Domain-Driven Design

Tackling Complexity in the Heart of Software

Eric Evans

eric@domainlanguage.com
(415) 902-7873

(Final Manuscript, April 15, 2003)

For current draft and other information, see
www.domainlanguage.com

To Mom and Dad

Table of Contents

Acknowledgements

I have been working on this book, in one form or another, for over four years, and many people have helped and supported me along the way.

I thank the many people who have read manuscripts and commented. This book would simply not have been possible without that feedback. A few have given their reviews especially generous attention. The Silicon Valley Patterns Group, led by Russ Rufer and Tracy Bialek, spent seven weeks scrutinizing the first complete draft of the book. The University of Illinois reading group led by Ralph Johnson also spent several weeks scrutinizing a later draft. Listening to the long, lively discussions of these groups had a profound effect. Kyle Brown and Martin Fowler contributed detailed feedback, valuable insights and invaluable moral support (while sitting on a fish). Ward Cunningham's feedback was important. Alistair Cockburn encouraged me and helped me find my way through the publication process. David Siegel and Eugene Wallingford have helped me avoid embarrassing myself in the more technical parts. Vibhu Mohindra and Vladimir Gitlevitch painstakingly checked all the code examples.

Rob Mee read some of my earliest explorations of the material, and brainstormed ideas with me when I was groping for some way to communicate this style of design. He then poured over a much later draft with me.

Josh Kerievsky is responsible for one of the major turning points in the book's development: He persuaded me to try out the "Alexandrian" pattern format, which became the backbone of the book's organization. He also helped me to bring some of the material now in Part II together into a coherent form for the first time, during the intensive "shepherding" process preceding the PLoP conference in 1999. This became a seed around which the much of the rest of the book formed.

Before I could have conceived of this book, I had to form my view and understanding of software development. That development owed a lot to the generosity of a few brilliant people who acted as informal mentors to me, as well as friends. David Siegel, Eric Gold, and Iseult White, in very different ways, helped me form my way of thinking about software design. At more or less the same time, Bruce Gordon, Richard Freyberg, and Judith Segal, also in very different ways, helped me find my way in the world of successful project work.

There was also a body of ideas in the air at the crucial time that formed the basis of my own technical view. Some of those contributions will be clear in the main text and referenced where possible. Others are so fundamental that I don't even realize their influence on me.

My master's thesis advisor, Dr. Bala Subramanium, turned me on to mathematical modeling, which we applied to chemical reaction kinetics, but modeling is modeling, and that work was part of the path that led to this book.

And even before that, my way of thinking was formed thanks to my parents, Carol and Gary Evans, and a few special teachers, especially Dale Courier (a high school math teacher), Mary Brown (a high school English composition teacher), and a 6th grade science teacher whose name I have shamefully forgotten.

Finally, I thank my friends and family, and Fernando De Leon, for their encouragement throughout this long process.

Preface

Leading software designers have recognized domain modeling and design as critical topics for at least twenty years, yet surprisingly little has been written about what needs to be done or how to do it. Although it has never been clearly formulated, a philosophy has developed as an undercurrent in the object community, which I call “domain-driven design”.

I have spent the past decade focused on developing complex systems in several business and technical domains. I’ve tried best practices in design and development process as they have emerged from the leaders in the object-oriented development community. Some of my projects were very successful; a few failed. A feature common to the successes was a rich domain model that evolved through iterations of design and became part of the fabric of the project.

This book provides a framework for making design decisions and a technical vocabulary for discussing domain design. It is a synthesis of widely accepted best practices along with my own insights and experiences. Projects facing complex domains can use this framework to approach domain-driven design systematically.

Contrasting Three Projects

Three projects stand out in my memory as vivid examples of the dramatic effect domain design practice has on development results. Although all three delivered useful software, only one achieved its ambitious objectives and delivered complex software that continued to evolve to meet ongoing needs of the organization.

I watched one project get out of the gate fast with a useful, simple web-based trading system. Developers were flying by the seat of their pants, but simple software can be written with little attention to design. As a result of this initial success, expectations for future development were sky-high. It was at this point that I was approached to work on the second version. When I took a close look, I saw that they lacked a domain model, or even a common language on the project, and were saddled with an unstructured design. So when the project leaders did not agree with my assessment, I declined the job. A year later, they found themselves bogged down and unable to deliver a second version. Although their use of technology was not exemplary, it was the business logic that overcame them. Their first release had ossified prematurely into a high-maintenance legacy.

Lifting this ceiling on complexity calls for a more serious approach to the design of domain logic. Early in my career, I was fortunate to end up on a project that did emphasize domain design. This project, in a domain at least as complex as the one above, also started with a modest initial success, delivering a simple application for institutional traders. But this delivery was followed up with successive accelerations of development. Each successive iteration opened exciting new options for integration and elaboration of functionality. The team way able to respond to the needs of the traders with flexibility and expanding capability. This upward trajectory was directly attributable to an incisive domain model, repeatedly refined and expressed in code. As the team gained new insight into the domain, the model deepened. The quality of communication improved among developers and between developers and domain experts, and the design, far from imposing an ever-heavier maintenance burden, became easier to modify and extend.

Unfortunately, not all projects that start with this intention manage to arrive at this virtuous cycle. One project I joined started with lofty aspirations to build a global enterprise system based on a domain model, but finally had a disappointing result. The team had good tools, a good understanding of the business and gave serious attention to modeling. But a separation of developer roles led to a disconnect between the model and implementation, so the design did not reflect the deep analysis that was going on. In any case, the design of detailed business objects was not rigorous enough to support combining them in elaborate applications. Repeated iteration produced no improvement in the code, due to uneven skill-level among developers with no clear understanding of the particular kind of rigor needed. As months rolled by, development work became mired in complexity and the team lost its cohesive vision of the system. After

years of effort, the project did produce modest, useful software, but had given up the early ambitions along with the model focus.

Of course many things can put a project off course, bureaucracy, unclear objectives, lack of resources, to name a few, but it is the approach to design that largely determines how complex software can become. When complexity gets out of hand, the software can no longer be understood well enough to be easily changed or extended. By contrast, a good design can make opportunities out of those complex features.

Some of these design factors are technological, and a great deal of effort has gone into the design of networks, databases, and other technical dimension of software. Books have been written about how to solve these problems. Developers have cultivated their skills.

Yet the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user. When this domain complexity is not dealt with in the design, it won't matter that the infrastructural technology is well-conceived. A successful design must systematically deal with this central aspect of the software.

The premise of this book is that

1. For most software projects, the primary focus should be on the domain and domain logic.
2. Complex domain designs should be based on a model.

Domain-driven design is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains. To accomplish that goal, this book presents an extensive set of design practices, techniques and principles.

Design vs. Development Process

Design books. Process books. They seldom even reference each other. Each is a complex topic in its own right. This is a design book. But I believe that these two issues are inextricable if design concepts are to be put into successful practice and not dry up into academic discussion. When people learn design techniques, they feel excited by the possibilities, but then the messy realities of a real project descend on them. They don't see how to fit the new design ideas with the technology they must use. Or they don't know when to worry about a particular design aspect and when to let go in the interest of time. While it is possible to talk with other team members about the application of a design principle in the abstract, it is more natural to talk about the things we do together. So, while this is a design book, I'm going to barge right across that artificial boundary when I need to. This will place design in the context of a development process.

This book is not specific to a particular methodology, but it is oriented toward the new family of "Agile Development Processes". Specifically, it assumes a couple of process practices are in place on the project. These two practices are prerequisites for applying the approach in this book.

- Iterative development. The practice of iterative development has been advocated and practiced for decades, and is a corner stone of the Agile development methods. There are many good discussions in the literature of Agile development and Extreme Programming, among them, [Cockburn1998] and [Beck 1999].
- A close relationship between developers and domain experts. Domain-driven design crunches a huge amount of knowledge into a model that reflects deep insight into the domain and a focus on the key concepts. This is a collaboration between those who know the domain and those who know how to build software. Because it is iterative, this collaboration must continue throughout the project's life.

Extreme Programming (XP), conceived by Kent Beck, Ward Cunningham and others [Beck2000], is the most prominent of the agile processes and the one I have worked with most. To make the discussion

concrete, I will use XP throughout the book as the basis for discussion of the interaction of design and process. The principles illustrated are easily adapted to other Agile Processes.

In recent years there has been a rebellion against elaborate development methodologies that burden projects with useless, static documents and obsessive upfront planning and design. Instead, the Agile Processes, such as XP, emphasize the ability to cope with change and uncertainty.

XP recognizes the importance of design decisions, but strongly resists upfront design. Instead, it puts an admirable effort into increasing communication, and increasing the project's ability to change course rapidly. With that ability to react, developers can use the "simplest thing that could work" at any stage of a project and then continuously refactor, making many small design improvements, ultimately arriving at a design that fits the customer's true needs.

This has been a much-needed antidote to some of the excesses of design enthusiasts. Projects have bogged down in cumbersome documents that provided little value. They have suffered "analysis paralysis", so afraid of an imperfect design that they made no progress at all. Something had to change.

Unfortunately, some of these new process ideas can be easily misinterpreted. Each person has a different definition of "simplest". Continuous refactoring without design principles to guide these small redesigns developers can produce a code base hard to understand or change – the opposite of agility. And, while fear of unanticipated requirements often leads to over-engineering, the attempt to avoid over-engineering can develop into another fear: The fear of any deep design thinking at all.

In fact, XP works best for developers with a sharp design sense. The XP process assumes that you can improve a design by refactoring, and that you will do this often and rapidly. But design choices make refactoring itself easier or harder. The XP process attempts to increase team communication. But model and design choices clarify or confuse communication. What is needed is an approach to domain modeling and design that pulls its weight.

This book intertwines design and development practice and illustrates how domain-driven design and agile development reinforce each other. A sophisticated approach to domain modeling within the context of an agile development process will accelerate development. The interrelationship of process with domain development makes this approach more practical than any treatment of "pure" design in a vacuum.

The Structure of This Book

The book is divided into four major sections:

Part I: Putting the Domain Model to Work presents the basic goals of domain-driven development that motivate the practices in later sections. Since there are so many approaches to software development, *Part I* defines terms, and gives an overview of the implications of placing the domain model in the role of driving communication and design.

Part II: The Building Blocks of Model-driven Design condenses a core of best practices in object-oriented domain modeling into a set of basic building blocks. The focus of this section is on bridging the gap between models and practical, running software. Sharing these standard patterns brings order to the design and makes it easy for team members to understand each other's work. Using standard patterns also establishes a common language, which all team members can use to discuss model and design decisions.

But the main point of this section is on the kind of decisions that keep the model and implementation aligned with each other, reinforcing each other's effectiveness. This alignment requires attention to the detail of individual elements. Careful crafting at this small scale gives developers a steady platform to apply the modeling approaches of Parts III and IV.

Part III: Refactoring Toward Deeper Insight goes beyond the building blocks to the challenge of assembling them into practical models that provide the payoff. Rather than jumping directly into esoteric design principles, this section emphasizes the discovery process. Valuable models do not emerge

immediately. They require a deep understanding of the domain. That understanding comes from diving in, implementing an initial design based on a probably naïve model, and then transforming it again and again. Each time the team gains insight, the model is transformed to reveal that richer knowledge, and the code is refactored to reflect the deeper model and make it's potential available to the application. Then, once in a while, this onion pealing leads to an opportunity to break through to a much deeper model, attended by a rush of profound design changes.

Exploration is inherently open-ended, but it does not have to be random. *Part III* delves into modeling principles that can guide choices along the way, and techniques that help direct the search.

Part IV: Strategic Design deals with situations that arise in complex systems, larger organizations, interactions with external systems and legacy systems. This section explores a triad of principles that apply to the system as a whole: Bounded Context, Distillation, and Large-Scale Structure. Strategic design decisions are made by teams, or even between teams. Strategic design enables the goals of *Part I* to be realized on a larger scale, for a big system or in an application that fits in an enterprise-wide network.

Throughout the book, discussions are illustrated with realistic examples, drawn from actual projects, rather than oversimplified "toy" problems.

Much of the book is written as a set of "patterns". The reader should be able to fully understand the material without concern about this device, but those who are interested in the style and format of the patterns can read Appendix 1.

Who This Book is Written For

This book is primarily written for developers of object-oriented software. Most members of a software project team can benefit from some parts of it. It will make most sense to people who are on a project, trying to do some of these things as they go through, or who have deep experience already to relate it to.

Some knowledge of object-oriented modeling is necessary to benefit from this book. The examples include UML diagrams and Java code, so the ability to read those languages at a basic level is important, but it is unnecessary to have mastered the details of either UML or Java. Knowledge of Extreme Programming will add perspective to the discussions of development process, but the discussion should be understandable without background knowledge.

For an intermediate software developer, a reader who already knows something of object-oriented design and may have read one or two software design books, this book will fill in gaps and provide perspective on how object modeling fits into real life on a software project. It will help an intermediate developer make the jump to applying sophisticated modeling and design skills to practical problems.

An advanced or expert software developer should be interested in the comprehensive framework for dealing with the domain. The systematic approach to design will help them in leading teams down this path. The coherent terminology will help them communicate with peers.

Readers of various backgrounds may wish to take different paths through the book, shifting emphasis to different points. I recommend all readers to start with the introduction to Part I, and Chapter 1. This book is a narrative, and can be read beginning to end, or from the beginning of any chapter. A skimmer who already has some grasp of a topic should be able to pick up the main points by reading headings and bolded text. A very advanced reader may want to skim Parts I and II, and will probably be most interested in Parts III and IV.

In addition to this core readership, the book will be of interest to analysts and to relatively technical project managers. Analysts can draw on the connection between model and design to make more effective contributions in the context of an "Agile" project. Analysts may also use some of the principles of strategic design to better focus and organize their work.

Project managers should be interested in the emphasis on making a team more effective and more focused on designing software meaningful to business experts and users. And, since, strategic design decisions are

interrelated with team organization and work styles, these design decisions necessarily involve the leadership of the project, and have a major impact on the project's trajectory..

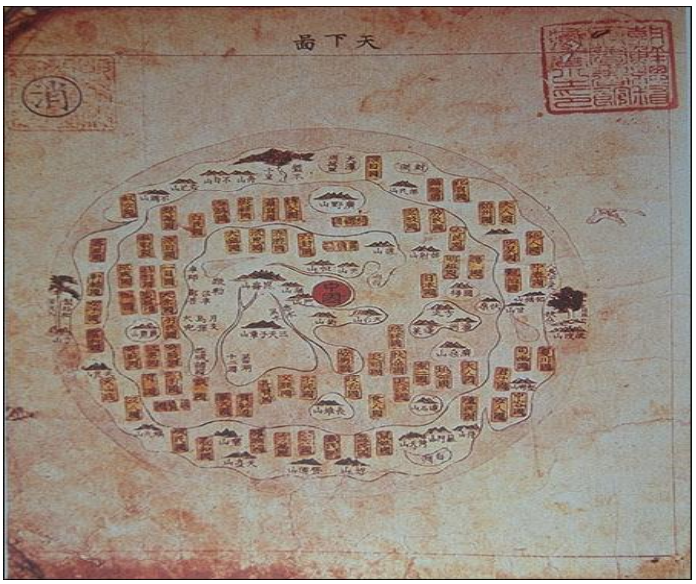
While an individual developer who understands domain-driven design will gain valuable design techniques and perspective, the biggest gains come when a team joins to apply a domain-driven design approach and move the domain model to the center of discourse of the project. The team members will share a language that enriches their communication and keeps it connected to the software. They will produce an implementation in step with the model, giving leverage to application development. They will share a map of how design work of different teams relates, and will systematically focus attention on the most features most distinctive and valuable to the organization.

A domain-driven design is a difficult technical challenge that can pay off big, opening opportunities just at the stage when most software projects begin to ossify into legacy.

Eric Evans, San Francisco, California, March 2003

http://domainlanguage.com

Part I. Putting the Domain Model to Work



The 18th century Chinese map above represents the whole world. In the center and taking up most of the space is China, surrounded by perfunctory representations of other countries. This was a model of the world appropriate to that society, which had intentionally turned inward. The worldview that the map represents must not have been helpful in dealing with foreigners. Certainly it would not serve modern China at all. Maps are models, and every model represents some aspect of reality or of an idea that is of interest its. It is a simplification. It is an interpretation of reality that abstracts the aspects relevant to solving the problem at hand and ignores extraneous detail.

Every software program relates to some activity or interest of its user. That subject area to which the user applies the program is the "domain" of the software. Some domains involve the physical world. The domain of an airline-booking program involves real people getting on real aircraft. Some domains are intangible. The domain of an accounting program is money and finance. Software domains usually have little to do with computers, though there are exceptions. The domain of a source-code control system is software development itself.

To create valuable software, we must bring to bear a body of knowledge related to the activities the software will be involved in. The amount of knowledge required can be daunting. The volume and complexity of information can be overwhelming. This is when a development team can use modeling to wrestle with that overload. A model is a selectively simplified and consciously structured form of knowledge. An appropriate model makes sense of information and brings it to bear on a problem.

The domain model is not a particular diagram; it is the idea that the diagram is intended to convey. It is not just the knowledge in a domain expert's head; *it is a rigorously organized and selective abstraction of that knowledge*. A diagram can represent and communicate a model, as can carefully written code, as can an English sentence.

Domain modeling is not a matter of making as "realistic" a model as possible. Even in a domain of tangible real-world things, our model is an artificial creation. Nor is it just the construction of a software mechanism

that gives the necessary results. It is more like movie making, loosely representing reality to a particular purpose. Even a documentary film does not show unedited real life. Just as a moviemaker selects aspects of experience and presents them in an idiosyncratic way to tell a story or make a point, a domain model is chosen for its utility.

The Utility of a Model in Domain-Driven Design

In domain-driven design, there are three basic uses that determine the choice of a model.

1. The model dictates the form of the design of the heart of the software. It is the intimate link between the model and the implementation that makes the model relevant and ensures that the analysis that went into it applies to the final product, a running program. This binding of model and implementation also helps maintenance and continuing development because the code can be interpreted based on understanding the model. (Chapter 3)
2. The model is the backbone of a language used by all team members. Because of the binding of model and implementation, developers can talk about the program in this language. They can communicate with domain experts without translation. And because the language is based on the model, our natural linguistic abilities can be turned to refining the model itself. (Chapter 2)
3. The model is distilled knowledge. The model is the team's agreed-upon way of structuring domain knowledge, and distinguishing the elements of most interest. A model captures how we choose to think about the domain as we select terms, break down concepts, and relate them. The shared language allows effective collaboration of developers and domain experts to wrestle information into this form. The binding of model and implementation ensures means that experience with early versions of the software are also valid feedback into the modeling process. (Chapter 1)

The next three chapters set out to examine the meaning and value of each of these contributions in turn, and the ways they are intertwined. Using a model in these ways can support the development of software with rich functionality that would otherwise take a massive investment of ad hoc development.

Yet most projects get very little out of their domain model. This book will examine an array of potential obstacles to effective domain development, and ways of getting past those obstacles with design principles ranging from high-level concepts to concrete techniques.

The Centrality of Domain Functionality

The heart of software is its ability to solve domain related problems for the user. All other features, vital though they may be, support this basic purpose. When the domain is complex, this is a difficult task calling for the concentrated effort of talented and skilled people. Developers have to steep themselves in the domain to build up knowledge of the business. They must hone their modeling skills and master domain design.

Yet this is not the priority on most software projects. Most talented developers do not have much interest in learning about the specific domain they are working in, much less making a major commitment to expand their domain modeling skills. Technical people enjoy quantifiable technical problems that exercise their technical skills. The domain is messy and demands a lot of complicated new knowledge that doesn't seem to develop a computer scientist's capabilities.

Instead, the technical talent goes to work on elaborate frameworks, trying to solve domain problems with technology. Complexity in the heart of software has to be tackled head on. To not have this focus is a project risk.

In a TV talk show interview, comedian John Cleese told a story of an event during the filming of "Monty Python and the Holy Grail". They had been shooting a particular scene over and over, but somehow it wasn't funny. Finally, he took a break and consulted with fellow comedian Michael Palin (the other actor in the scene), and they came up with a slight variation. They shot one more take, and it turned out funny, so they called it a day.

The next morning Mr. Cleese was looking at the rough cut the film editor had put together of the previous day's work. Coming to the scene they had struggled with, he found that it wasn't funny; one of the earlier takes had been used.

He asked the film editor why he hadn't used the last take, as directed. "Couldn't use it. Someone walked in-shot," the editor replied. Mr. Cleese watched the scene again, and then again. Still he could see nothing wrong. Finally the editor stopped the film and pointed out a coat sleeve that was visible for a moment at the edge of the picture.

The film editor was concerned that other film editors who saw the movie would judge his work based on its technical perfection. He was focused on the precise execution of his own specialty, and, in the process, the heart of the scene had been lost. ["The Late Late Show with Craig Kilborn", CBS, September, 2001]

Fortunately, the funny scene was restored by a director who understood comedy. In just the same way, leaders within a team who understand the centrality of the domain can put their software project back on course when enthusiastic developers get caught up in develop elaborate technical frameworks that do not serve, or actually get in the way of domain development, while development of a model that reflects deep understanding of the domain is lost in the shuffle.

This book will show that domain development holds opportunities to cultivate very sophisticated design skills. The messiness of most business domains is an interesting technical challenge. In fact, in many scientific disciplines, "complexity" is one of the most exciting current topics, as researchers attempt to tackle the messiness of the real world. A software developer has that same prospect when facing a complicated domain that has never been formalized. Creating a lucid model that cuts through that complexity is exciting.

There are systematic ways of thinking that developer's can employ to search for insight and produce effective models. There are design techniques that can bring order to a sprawling software application. Cultivation of these skills make a developer much more valuable, even in an initially unfamiliar domain.

1. Crunching Knowledge

A few years ago, I set out to design a specialized software tool for Printed Circuit Board (PCB) design. One catch: I didn't know anything about electronic hardware. I had access to some PCB designers, of course, but they typically got my head spinning in three minutes. How was I going to understand enough to write this software? I certainly wasn't going to become an electrical engineer before the delivery deadline!

We tried the approach of having them tell me exactly what the software should do. Bad idea. They were great circuit designers, but their software ideas usually involved reading in an ASCII file, sorting it, writing it back out with some annotation and producing a report. This was clearly not going to lead to the leap forward in productivity that they were looking for.

These first few meetings were discouraging, but there was a glimmer of hope in the reports they asked for. They always involved "nets" and various details about them. A net, in this domain, is essentially a wire conductor that can connect any number of components on a PCB and carry an electronic signal to everything it is connected to. We had the first element of the domain model.

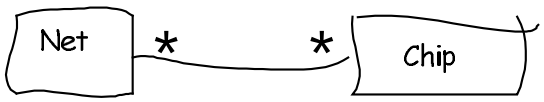


Figure 1. 1

I started drawing diagrams for them as we discussed the things they wanted the software to do. I used an informal variant of object interaction diagrams to walk through scenarios.

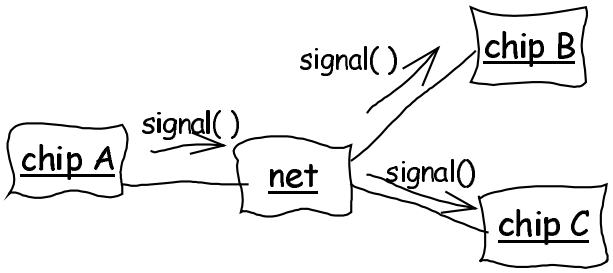


Figure 1. 2

PCB EXPERT 1: The components wouldn't have to be chips.

DEVELOPER (ME): So I should just call them components?

EXPERT 1: We call them "component instances". There could be many of the same component.

EXPERT 2: The "net" box looks just like a component instance.

EXPERT 1: He's not using our notation. Everything is a box for them, I guess.

DEVELOPER: Sorry to say, yes. I guess I'd better explain this notation a little more.

They constantly corrected me, and as they did I started to learn. We ironed out collisions and ambiguities in their terminology and differences between their technical opinions, and they learned. They began to explain things more precisely and consistently, and we started to develop a model together.

EXPERT 1: It isn't enough to say a signal arrives at a ref-des, we have to know the pin.

DEVELOPER: Ref-des?

EXPERT 2: Same thing as a component instance. Ref-des is what it's called in a particular tool we use.

EXPERT 1: Anyhow, a net connects a particular pin of one instance to a particular pin of another.

DEVELOPER: Are you saying that a pin belongs to only one component instance and connects to only one net?

EXPERT 1: Yes, that's right.

EXPERT 2: Also, every net has a topology, an arrangement that determines the way the elements of the net connect.

DEVELOPER: Ok, how about this?

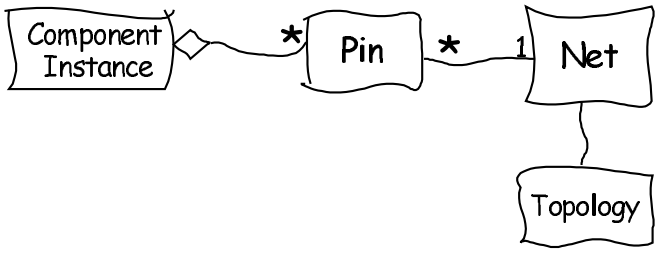


Figure 1. 3

To focus our exploration, we limited ourselves, for a while, to studying one particular feature. A "probe simulation" would trace the propagation of a signal to detect likely sites of certain kinds of problems in the design.

DEVELOPER: I understand how the signal gets carried by the **Net** to all the **Pins** attached, but how does it go any further than that? Does the **Topology** have something to do with it?

EXPERT 2: No. The component pushes the signal through.

DEVELOPER: We certainly can't model the internal behavior of a chip. That's way too complicated.

EXPERT 2: We don't have to. We can use a simplification. Just a list of pushes through the component from certain **Pins** to certain others.

DEVELOPER: Something like this?

[With considerable trial-and-error, together we sketched out a scenario.]

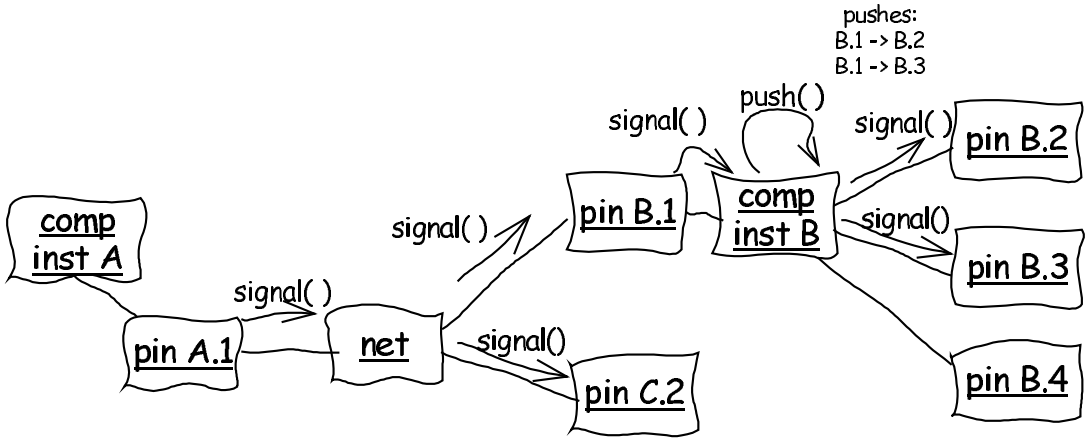


Figure 1. 4

DEVELOPER: But what exactly do you need to know from this computation?

EXPERT 2: We'd be looking for long signal delays – say, any signal path that was more that two or three hops. It's a rule of thumb. If the path is too long, the signal may not arrive during the clock cycle.

DEVELOPER: More than three hops… So we need to calculate the path lengths. And what counts as a hop?

EXPERT 2: Each time the signal goes over a **Net**, that's one hop.

DEVELOPER: So we could pass the number of hops along, and a **Net** could increment it, like this.

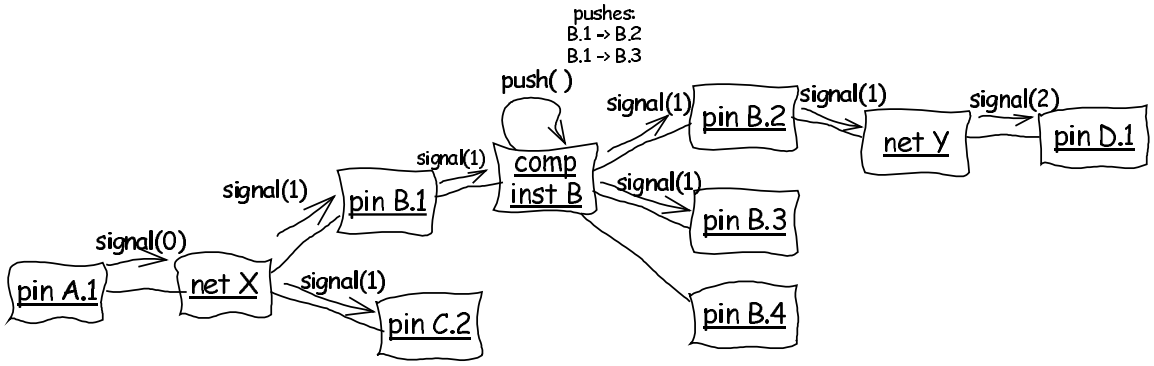


Figure 1. 5

DEVELOPER: The only part that isn't clear to me is where the "pushes" come from. Do we store that data for every **Component Instance**?

EXPERT 2: The pushes would be the same for all the instances of a component.

DEVELOPER: So the type of component determines the pushes. They'll be the same for every instance?

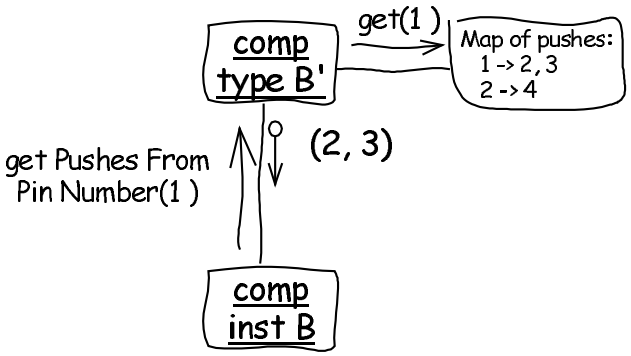


Figure 1. 6

EXPERT 2: I'm not sure exactly what some of this means, but I would imagine storing push-throughs for each component would look something like that.

DEVELOPER: Sorry, I got a little too detailed there. I was just thinking it through.

DEVELOPER: So, now, where does the **Topology** come into it?

EXPERT 1: That's not used for the probe simulation.

DEVELOPER: Then I'm going to drop it out for now, ok? We can bring it back when we get to those features.

And so it went (with much more stumbling than is shown here). Brainstorming and refining; questioning and explaining. The model developed along with my understanding of the domain and their understanding of how the model would play into the solution. A class diagram representing that early model looked something like this.

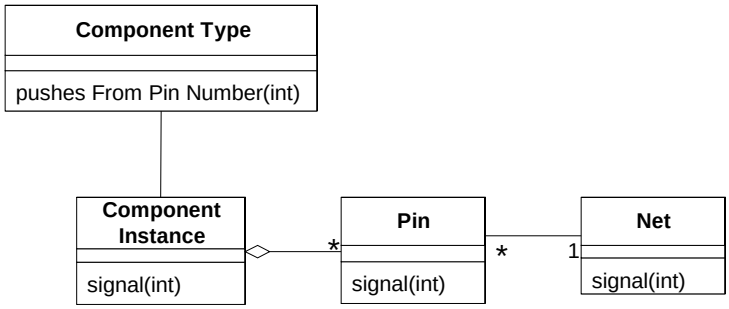


Figure 1. 7

After a couple more part-time days of this, I felt I understood enough to attempt some code. I wrote a very simple prototype, driven by an automated test framework. I avoided all infrastructure. There was no persistence, and no U.I. This allowed me to concentrate on the behavior. I was able to demonstrate a simple probe simulation in just a few more days. Although it used dummy data and wrote raw text to the console, it was, none-the-less, doing the actual computation of path-lengths using Java objects. Those Java objects reflected a model shared by the domain experts and myself.

The concreteness of this prototype made clearer to them what the model meant and how it related to the functioning software. From that point, our model discussions became more interactive, as they could see how I incorporated my newly acquired knowledge into the model and then into the software. And they had concrete feedback from the prototype to evaluate their own thoughts.

Embedded in that model, which naturally became much more complicated that the one shown here, was knowledge about the domain of PCB relevant to the problems we were solving. It consolidated many synonyms and slight variations in descriptions. It excluded hundreds of facts that the engineers understood but that were not directly relevant, such as the actual digital features of the components. A software specialist like me could look at the diagrams and in minutes start to get a grip on what the software was about. He or she would have a framework to organize new information and learn faster, to make better guesses about what was important and what was not, and to communicate better with the PCB engineers.

As the engineers described new features they needed, I made them walk me through scenarios of interactions of the objects. When the model objects couldn't carry us through an important scenario, we brainstormed new ones or changes to old ones crunched their knowledge. We refined the model; the code coevolved. A few months later they had a rich tool that exceeded their expectations.

Why It Worked

Certain things we did led to this success.

1. Binding model and implementation as soon as possible. That crude prototype forged the essential link.
2. Cultivating a language based on the model. At first, they had to explain elementary PCB issues to me while I had to explain what a class diagram meant. But as the project proceeded, terms taken straight out of the model, organized into sentences consistent with the structure of the model tested the viability of the model immediately upon reaching their ears or mine.
3. Knowledge rich model. The objects had behavior and enforced rules. The model wasn't just a data schema, it was integral to solving a complex problem. It captured knowledge of various kinds.

4. The model was distilled. Important concepts were added to the model as it became more complete, but equally important, concepts were dropped when they didn't prove useful or central. When na unneeded concept was tied to one that was needed, a new model was found that distinguished the essential concept so that the other could be dropped.
5. Knowledge crunching. The language combined with sketches and a brainstorming attitude turned our discussions into laboratories of the model, in which hundreds of experimental variations could be exercised, tried and judged.

It is this last point, knowledge crunching, that makes it possible to find a knowledge rich model and ways to distill it. It requires the creativity of brainstorming and massive experimentation.

Financial analysts crunch numbers. They sift through reams of detailed figures, combining and recombining them looking for the underlying meaning, looking for a simple presentation that brings out what is really important – an understanding that can be the basis of a financial decision.

Effective domain modelers are knowledge crunchers. They take a torrent of information and probe for the relevant trickle. They try one organizing idea after another, searching for the simple view that makes sense of the mass. Many models are tried and rejected or transformed. Success comes in an emerging set of abstract concepts that make sense of all the detail. This distillation is a rigorous expression of the particular knowledge that has been found most relevant.

Knowledge crunching is not a solitary activity. A team of developers and domain experts collaborate, typically led by developers. Together they draw in information and crunch it into a useful form. The raw material comes from the minds of domain experts, from users of existing systems, from the prior experience of the technical team with a related legacy system or another project in the same domain. It comes in the form of documents written for the project or used in the business, and lots and lots of talk. Early versions or prototypes feed experience back into the team and change earlier interpretations.

In the old waterfall method, the business experts talked to the analysts, analysts digested and abstracted and passed the result along to the programmers who coded the software. This failed because it completely lacks feedback. The analysts have full responsibility for creating the model based only on input from the business experts. They have no opportunity to learn from the programmers or gain experience with early versions. Knowledge trickles in one direction, but does not accumulate.

Other projects have iteration, but don't build up knowledge because they don't abstract. They get the experts to describe a desired feature and they go build it. They show the experts the result and ask them what to do next. If the programmers practice refactoring they can keep the software clean enough to continue extending it, but if programmers are not interested in the domain, they only learn what the application should do, not the principles behind it. Useful software can be built that way, but the project will never gain the kind of leverage where powerful new features unfold as corollaries to older features.

Good programmers will naturally start to abstract and develop a model that can do more work. But when this happens only in a technical setting, without collaboration with domain experts, the concepts are naïve. That shallowness of knowledge produces software that does a basic job but lacks a deep connection to the domain expert's way of thinking.

The interaction between team members changes as all members crunch the model together. The constant refinement of the domain model forces the developers to learn the important principles of the business they are assisting, rather than mechanically producing functions. The domain experts often refine their own understanding by being forced to distill what they know to essentials, and they come to understand the conceptual rigor software projects require.

All this makes the team more competent knowledge crunchers. They winnow out the extraneous. They recast the model into an ever more useful form. Because analysts and programmers are feeding into it, it is cleanly organized and abstracted, and can provide leverage for the implementation.

Because the domain experts are feeding into it, it reflects deep knowledge of the business and those abstractions are true business principles.

As the model improves, it becomes a tool for organizing the information that continues to flow through the project. It focuses requirements analysis. It intimately interacts with programming and design. And, in a virtuous cycle, it deepens team member's insight into the domain, letting them see more clearly and leading to further refinement of the model. These models are never perfect. They evolve. They must be practical and useful in making sense of the domain. They must be rigorous enough to make the application simple to implement and understand.

Continuous Learning

When we set out to write software, we never know enough. Knowledge on the project is fragmented, scattered among many people and documents, and mixed with other information so that we don't even know which bits of knowledge we really need. Domains that seem less technically daunting can be deceiving – we don't realize how much we don't know. This leads us to make false assumptions.

Meanwhile, all projects leak knowledge. People who have learned something move on. Reorganization scatters the team and the knowledge is fragmented again. Crucial subsystems are outsourced in such a way that code is delivered but not knowledge. And with typical design approaches, the code and documents don't express the knowledge that was so hard earned, so when the oral tradition is interrupted for any reason, the knowledge is lost.

Highly productive teams consciously grow their knowledge, practicing "continuous learning" [Kerievsky 2001]. For developers, this means improving technical knowledge, along with general domain modeling skills (such as those in this book). But it also includes serious learning about the specific domain they are working in. These self-educated team-members form a stable core of people to focus on the development tasks that involve the most critical areas (see Chapter 15, "Distillation").

Knowledge is accumulated in the minds of the core team.

At this point, stop and ask yourself a question. Did you learn something about the PCB design process? Although this has been a superficial treatment of the topic, there should be some learning when a domain model is discussed. I learned an enormous amount. I did not learn how to be a PCB engineer. That was not the goal. I learned to talk to PCB experts, understand the major concepts relevant to the application, and sanity-check what we were building.

In fact, we discovered that the probe simulation was a low priority for development and was eventually dropped altogether. With it went the parts of the model that captured understanding of pushing signals through components and counting hops. The core of the application turned out to lay elsewhere, and the model changed to bring those aspects onto center stage. The domain experts had learned more and had clarified the goal of the application. (Chapter 15, "Distillation" will go these issues in depth.)

Even so, the early work was essential. Important model elements were retained, but more importantly, it set in motion the process that made all subsequent work effective: the knowledge gained by team members, developers and domain experts alike, the beginnings of a shared language, and closing the feedback loop through an implementation. A voyage of discovery has to start somewhere.

Knowledge Rich Design

The kind of knowledge captured in such a model goes beyond "find the nouns". Business activities and rules are as central to a domain as are the entities involved, and any domain will have various categories of concepts. Knowledge crunching yields models that reflect this kind of insight. In parallel with model changes, developers refactor the implementation to express the model, and giving the application use of that knowledge.

It is with this move beyond entities and values that knowledge crunching can get intense because there may be actual inconsistency among business rules. Domain experts are usually not aware of how complex their mental processes are as, in the course of their work, they navigate all these rules, reconciling contradictions, and filling in gaps with common sense. Software can't do this. It is through knowledge crunching in close collaboration with software experts that the rules are clarified, fleshed out, reconciled, or placed out of scope.

Example: Extracting a Hidden Concept

Let's start with a very simple domain model that is to be used as the basis of an application for booking cargos onto a voyage of a ship.

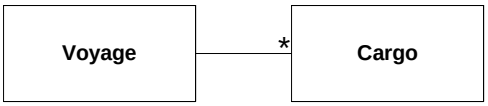


Figure 1. 8

We can state that the booking application's responsibility is to associate each **Cargo** with a **Voyage**, recording and tracking that relationship. So far so good. Somewhere in the application code there could be a method like this:

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

Standard practice in the shipping industry is to accept more cargo than a particular vessel can carry on a voyage. This is called "over-booking". Sometimes a simple percentage of capacity is used, such as booking 110% of capacity. In other cases complex rules are applied, favoring major customers or certain kinds of cargo.

This is a basic rule in the shipping domain that would be known to any businessperson in the shipping industry, but might not be understood by all technical people on a software team.

The requirements document contains this line:

```
Allow 10% overbooking.
```

The class diagram and code now look like this:

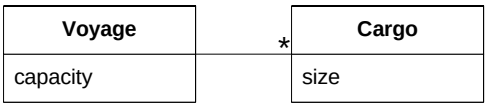


Figure 1. 9

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   double maxBooking = voyage.capacity() * 1.1;
   if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking) return -1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

Now an important business rule is hidden as a guard clause in an application method. Later we'll look at the principle of the LAYERED ARCHITECTURE (Chapter 4), which would guide us to refactor the overbooking rule into a domain object, but for now let's concentrate on how we could make this knowledge more explicit and accessible to everyone on the project.

1. As written, it is unlikely any business expert could read this code to verify the rule.
2. It would be difficult for a technical, non-business person to connect the requirement text with the code.

If the rule were more complex, that much more is at stake.

We can change the design to better capture this knowledge. The overbooking rule is a policy, a commonly used design pattern also known as STRATEGY [GHJV95]. It is usually motivated by the need to substitute different rules, which is not needed here, as far as we know. But the concept we are trying to capture does fit the *meaning* of a policy, which is an equally important motivation in domain-driven design. (See Chapter ##, "Relating Design Patterns to the Model".)

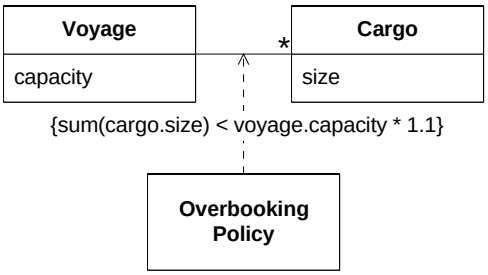


Figure 1. 10

The code is now:

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   if (!overbookingPolicy.isAllowed(cargo, voyage)) return –1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

The new **Overbooking Policy** class contains this method:

```
public boolean isAllowed(Cargo cargo, Voyage voyage) {
   return (cargo.size() + voyage.bookedCargoSize()) <= (voyage.capacity() * 1.1);
}
```

It will now be clear to all that overbooking is a distinct policy, and the implementation of that rule is explicit and separate.

Now, *I am not recommending that such an elaborate design be applied to every detail of the domain*. Chapter 15, Distillation, goes into depth on how to focus on the important and minimize or separate everything else. This example is meant to show that domain model and design can be used to secure and share knowledge.

1. In order to bring it to this stage, the programmers and everyone involved will have come to understand the nature of overbooking as a distinct and important business rule, not just an obscure calculation.
2. Programmers can show business experts technical artifacts, even code, that should be intelligible to them (with guidance), closing the feedback loop.

Deep Models

Useful models seldom lie on the surface. As we understand the domain and the needs of the application, we usually discard superficial model elements that seemed important in the beginning, or shift their perspective.

One of the projects that I'll be drawing on for examples throughout the book was a container shipping system. Since the beginning of a shipment is booking cargo, we developed a model that allowed us to describe the cargo, its itinerary, and so on. This was all necessary and useful, yet the domain experts felt dissatisfied. There was a way they looked at their business that we were missing.

Eventually, we realized that our focus on the handling of cargo, the physical loading and unloading, the movements from place to place, was largely handled by subcontractors or by operational people in the company. The view of our customers was of a series of transfers of responsibility between parties. A process governed transfer of legal and practical responsibility from the shipper to some local carrier, from one carrier to another, and to the consignee. Often, the cargo would sit in a warehouse while important steps were being taken. At other times, the cargo would move through complex physical steps that were not relevant to shipping company's business decisions. Rather than the logistical emphasis of the itinerary, what came to the fore was legal documents like the bill of lading, and processes leading to release of payments.

This deeper view of the shipping business did not mean there was no itinerary object, but the model changed profoundly. Our view of shipping changed from moving containers from place to place, to transferring responsibility for cargo from entity to entity. Features for handling these transfers of responsibility were no longer awkwardly attached to loading operations, but were supported by a model that came out of an understanding of the significant relationship between those operations and those responsibilities.

Knowledge crunching is an exploration, and you can't know where you will end up.

2. Communication and the Use of Language

The domain model can be the core of a common language for a software project. The model is the set of concepts built up in the heads of people on the project, with terms and relationships that reflect domain insight.

Traditionally emphasis has been on written text documents and diagrams, such as UML diagrams. Agile processes reemphasize less formal diagrams and casual conversation. XP in particular, stresses communication through the code itself and through the tests for that code.

All of these can be valuable in different situations. But to make any of these media work, we have to have to share concepts and a language to express them. The use of language on the project is subtle and all-important.

UBIQUITOUS LANGUAGE

For first you write a sentence,
And then you chop it small;
Then mix the bits, and sort them out
Just as they chance to fall:
The order of the phrases makes
No difference at all.

-Lewis Carroll, from "*Poeta Fit, Non Nascitur*"

To create a supple, knowledge-rich design calls for a versatile, shared team language, and a lively experimentation with language that seldom happens on software projects. The core of such a language comes from the domain model. The UBIQUITOUS LANGUAGE carries knowledge in a dynamic form.

♦ ♦ ♦

Domain experts have limited understanding of the technical jargon of software development, but they use the jargon of their field – probably in various flavors. Developers may understand and discuss the system in descriptive, functional terms, devoid of the meaning carried by the expert's language, or they may create abstractions that provide the basis for their design, yet are not understood by the domain experts. Developers working on different parts of the problem work out their own design concepts and ways of describing things.

Across this linguistic divide, the domain experts vaguely describe what they want. Developers, struggling to understand a domain new to them, vaguely understand. A few members of the team manage to become bilingual, and speak to the domain experts in their language and the developers in theirs, but they become bottlenecks of information flow, and their translations are inexact.

On a project without a common language, developers have to translate for domain experts. Domain experts translate between developers and still other domain experts. Translation is always inaccurate and hides disconnects in understanding between the domain experts and developers, between different developers and between different domain experts. Translation muddles model concepts, which leads to destructive refactoring of code. The indirectness of communication conceals the formation of schisms, where different team members use terms differently but don't realize it, which leads to unreliable software that doesn't fit together (see Chapter 12, "Maintaining Model Integrity"). The effort of translation prevents the interplay of knowledge and ideas that lead to deep model insights.

It is a serious problem when the language used on a project is fractured. Domain experts use their jargon while technical team members have their own language tuned for discussing the domain in terms of design. Translation blunts communication and makes knowledge crunching anemic. Yet none of these dialects can be a common language because none serves all needs.

The terminology of day-to-day discussions is disconnected from the terminology embedded in the code (ultimately the most important product of a software project). Even the same person uses different language in speech and in writing, and so the most incisive expressions of the domain often emerge in a transient form that is never captured in the code or even in writing.

The overhead cost of all the translation, plus the risk of misunderstanding, is simply too high. A project needs a common language that is more than the lowest common denominator. With a conscious effort by

the team, the domain model can provide the backbone for that common language, while connecting team communication to the software implementation. That language can be ubiquitous in the teams work.

A model can be viewed as a language. The vocabulary includes the names of classes and prominent operations. The language provides the means to discuss rules that have been made explicit in the model. It can be supplemented with terms from high-level organizing principles imposed on the model (such as CONTEXT MAPS and large-scale structures, which will be discussed in Chapters 14 and 16). Finally, this language is enriched with the names of patterns the team commonly applies to the domain model.

The model-based language should provide the primary language among developers to describe artifacts in the system, and also to describe tasks and functionality. This same model should provide the language for the developers and domain experts to communicate with each other, and for the domain experts to communicate among themselves about requirements, development planning, and features. The more pervasively the language is used, the more smoothly understanding will flow.

But, the model may simply not be good enough to fill these roles. The specialized jargons of the field are usually more semantically complete and tailored to the problem than the formalized domain model the developers create (though they often contain contradictions and ambiguities). The more subtle and active features of the model often only emerge in the code.

Although it is circular, a prerequisite for an adequate model is commitment by the team to rely on that model. Persistent use of the language will force its weaknesses into the open. The team will experiment and find alternatives to awkward terms or structure. As gaps are found, new words will enter the discussion. With a commitment to the model-based language, these changes will be recognized as changes in the domain model and will lead to updates of class diagrams and renaming of classes and methods in the code. Committed to using this language in the context of implementation, the developers will point out imprecision or contradictions, engaging the domain experts in discovering workable alternatives. When the team forces itself to rely on the model for their language, members will also be motivated to cultivate their modeling skills and work together to apply them.

Of course, domain experts will speak outside the scope of the UBIQUITOUS LANGUAGE, to explain and give context. But within its scope, they should use it, and raise concerns when they find it awkward or incomplete… or wrong. Repeated use in conversation exposes differences in interpretation of terms. By using it pervasively and not being satisfied until it flows, we approach a model that is complete and comprehensible, made up of simple elements that combine to express complex ideas.

Therefore,

Use the model as the backbone of a language. Commit the team to using that language relentlessly in all communication within the team and in the code. Use the same language in diagrams, writing, and, especially speech.

Iron out difficulties by experimenting with alternative expressions, which reflect alternative models. Then refactor the code, renaming classes, methods and modules to conform to the new model. Resolve confusion over terms in conversation, in just the way we converge on agreed meaning of ordinary words. Domain experts object to terms or structures that are awkward or inadequate to convey domain understanding, while developers watch for ambiguity or inconsistency that will trip up design.

With a UBIQUITOUS LANGUAGE, the model is not just a design artifact. It becomes integral to everything the developers and domain experts do together. Discussion in the LANGUAGE brings to life the meaning behind the diagrams and code.

♦ ♦ ♦

A UBIQUITOUS LANGUAGE based on the domain model assumes there is just one model in play. Chapter 14, Maintaining Model Integrity, deals with coexistence of different models and how to keep a model from splintering.

The LANGUAGE is the primary carrier of the aspects of design that don't appear in code – large-scale structures that organize the whole system (chapter 16), and BOUNDED CONTEXTS that define the relationships of different systems and models (Chapter 14).

Example Dialogs: Working out a Commercial Shipping Cargo Router

The differences between the two dialogs that follow are subtle, but important. Watch for how much they talk about what the software means to the business versus how it works technically. Are they speaking the same language? Is that language rich enough to carry the discussion of what the application must do?

Scenario 1: Minimal abstraction of the domain

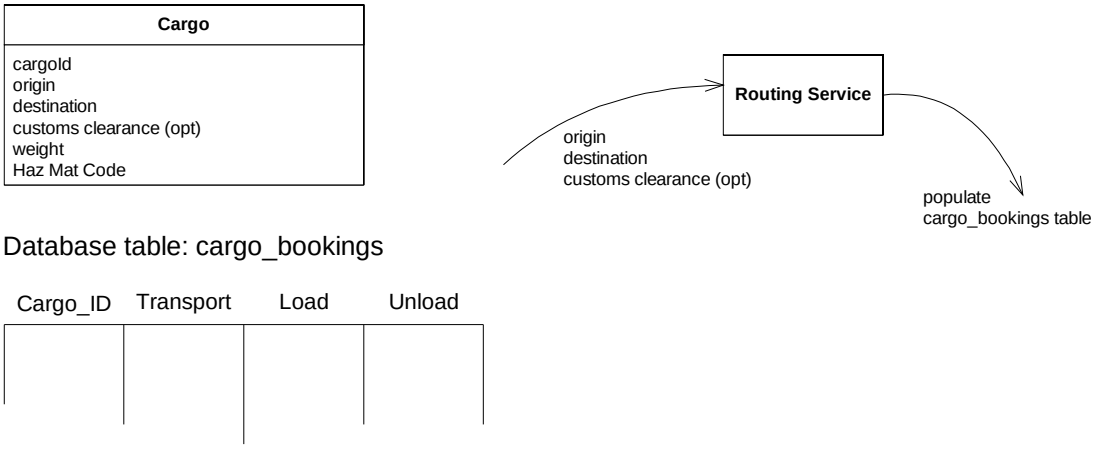


Figure 2. 1

USER: So when we change the customs clearance point, we need to redo the whole routing plan.

DEVELOPER: Right. We'll delete all the rows in the shipment table with that cargo id, then we'll pass the origin, destination and the new customs clearance point into the **Routing Service** and it will repopulate the table. We'll have to have a Boolean in the **Cargo** so we'll know there is data in the shipment table.

USER: Delete the rows? Ok, whatever. Anyway, if we didn't have a customs clearance point at all before, we'll have to do the same thing.

DEVELOPER: Sure, any time you change the origin, destination or customs clearance point (or enter one for the first time) we'll check to see if we have shipment data and then we'll delete it and then let the **Routing Service** regenerate it.

USER: Of course, if the old customs clearance just happened to be the right one, we wouldn't want to do that.

DEVELOPER: Oh, no problem. It is easier to just make the **Routing Service** redo the loads and unloads every time.

USER: Yes, but it is extra work for us to make all the supporting plans for a new itinerary, so we don't want to reroute unless the change necessitates it.

DEVELOPER: Ugh. Well, then, if you are entering a customs clearance point for the first time, we'll have to query the table to find the old derived customs clearance point, and then compare it to the new one. Then we'll know if we need to redo it.

USER: You won't have to worry about this on origin or destination, since the itinerary would always change then.

DEVELOPER: Good. We won't.

Or…

Scenario 2: Domain model enriched to support discussion

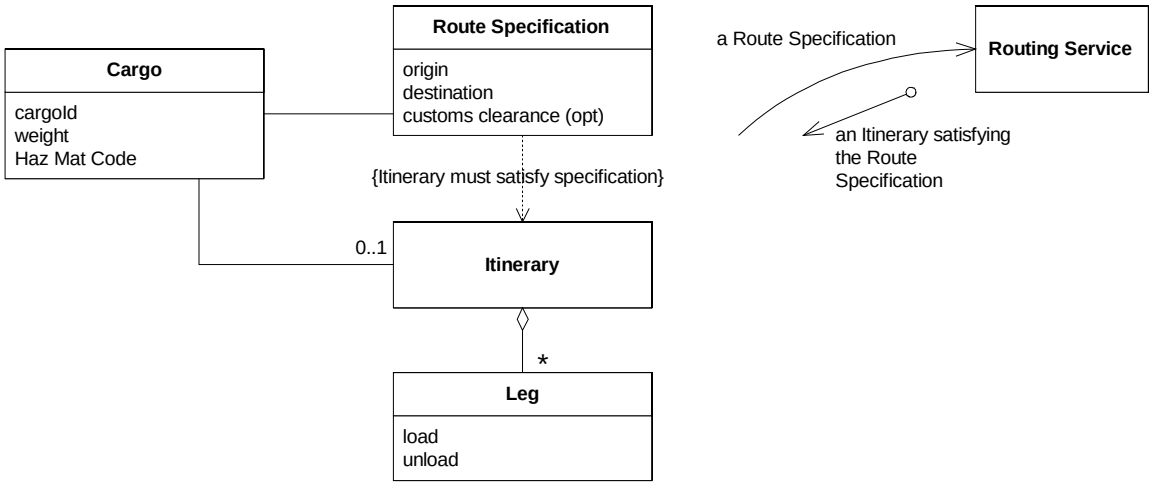


Figure 2. 2

USER: So when we change the customs clearance point, we need to redo the whole routing plan.

DEVELOPER: Right. When you change any of the attributes in the **Route Specification**, we'll delete the old **Itinerary** and ask the **Routing Service** to generate a new one based on the new **Route Specification**.

USER: If we hadn't specified a customs clearance point at all before, we'll have to do the same time.

DEVELOPER: Sure, any time you change anything in the **Route Spec**, we'll regenerate the **Itinerary**. That includes entering something for the first time.

USER: Of course, if the old customs clearance just happened to be the right one, we wouldn't want to do that.

DEVELOPER: Oh, no problem. It is easier to just make the **Routing Service** redo the **Itinerary** every time.

USER: Yes, but it is extra work for us to make all the supporting plans for a new **Itinerary**, so we don't want to reroute unless the change necessitates it.

DEVELOPER: Oh. Then we'll have to add some functionality to the **Route Specification**. Then, whenever you change anything in the **Spec**, we'll see if the **Itinerary** still satisfies the **Specification**. If it doesn't, we'll have the **Routing Service** regenerate the **Itinerary**.

USER: You won't have to worry about this on origin or destination, since the **Itinerary** would always change then.

DEVELOPER: Fine, but it will be simpler for us to just do the comparison every time. The **Itinerary** will only be generated when the **Route Specification** is no longer satisfied.

The second dialog conveys more of the intent of the domain expert. The user employed the word "itinerary" in both dialogs, but in the second it was an object the two could discuss precisely, concretely. They discussed the “route specification” explicitly, instead of describing it each time in terms of attributes and procedures.

The two dialogs were deliberately constructed to closely parallel each other. More realistically, the first would have been much more verbose, bloated with explanations of application features and miscommunications. The domain model based terminology of the second design makes the second dialog economical.

Modeling Out Loud

The disconnect between speech and other forms of communication is particularly harmful because we humans have a genius for spoken language. Unfortunately, when people speak, they usually don’t use the language of the domain model.

That may not ring true for you initially, and there are exceptions. But really listen in the next requirements or design discussion you attend. You’ll hear descriptions of features in business jargon or layman’s versions of the jargon. You’ll hear talk about technical artifacts and concrete functionality. Sure, you’ll hear terms from the domain model. The business jargon contains (we hope) many of these terms. Major nouns in this language are coded as objects that will tend to be mentioned. But do you hear phrases that could even remotely be described in terms of relationships and interactions in your current domain model?

One of the best ways of refining a model is to explore with speech, trying out loud various constructs from possible model variations. Rough edges are easy to hear.

1. If we give the Routing Service an origin, destination, and arrival time, it can look up the stops the cargo will have to make, and, well… stick them in the database.
2. The origin, destination, and so on… it all feeds into the Routing Service, and we get back an Itinerary that has everything we need in it.
3. A Routing Service finds an Itinerary that satisfies a Route Specification.

Playing around with words and phrases is a vital tool to complement the visual (sketching diagrams), code feel, and methodical analysis and design. (Part III of this book will delve into this discovery process and show this interplay in several dialogs.)

In fact, our brains seem to be somewhat specialized for dealing with complexity in spoken language (one good treatment for laymen, like myself, is [Pinker 94]). For example, when people of different language backgrounds come together for commerce, if they don’t have a common language they invent one, a “pidgin”, not as comprehensive as their original languages, but suited to the task at hand. When people are talking, they naturally discover differences in interpretation and the meaning of their words, and they naturally resolve those differences. They find rough spots in the language and smooth them out.

Once I took an intensive Spanish class in college. The rule in the classroom was that not a word of English could be spoken. At first, it was frustrating, felt very unnatural, and required a lot of self-discipline. But eventually we broke through to a level of fluency that we could never have reached through exercises on paper.

As we use the UBIQUITOUS LANGUAGE of the domain model in discussions, especially those with domain experts, with whom we discuss scenarios and requirements, we become more fluent in the language and teach each other its nuances. We naturally come to share the language that we speak in a way that never happens with diagrams and documents.

Bringing about a UBIQUITOUS LANGUAGE on a software project is easier said than done, and we have to fully employ our natural talents to pull it off. Just as a graphical overview can be conveyed rapidly because we take advantage of our specialized visual and spatial capabilities, we can use our innate talent for grammatical, meaningful language to drive model development.

Play with the model as you talk about the system. Describe scenarios out loud using the elements and interactions of the model, combining concepts in ways allowed by the model. Find easier ways to say what you need to say and then take those new ideas back down to the diagrams and code.

One Team, One Language

Technical people often feel the need to "shield" the business experts from the domain model. "Too abstract for them", "they don't understand objects", and "we have to collect requirements in their terminology". These are just a few of the reasons I've heard for having two languages on the team. Forget them. Of course there are technical components of the design that may not be of interest to them, but the core of the model had better be. Too abstract? Then how do you know the abstractions are sound? Do you understand the domain as deeply than they do? Sometimes specific requirements are collected from lower level users, and a subset of the more concrete terminology may be needed for them, but a domain expert is assumed to be capable of thinking somewhat deeply about his or her field. If sophisticated domain experts don't understand the model, there is something wrong with the model.

Now, at the beginning, when the users are discussing future capabilities of the system that haven't been modeled yet, there is no model for them to use. But as soon as they begin to discuss these new ideas with the developers, the process of groping toward a shared model begins. It may start out awkward and incomplete, but it will gradually get refined. As the new language evolves, the domain experts must make the extra effort to adopt it, and retrofit any old documents that are still important.

By using this language in discussions with domain experts, you quickly discover areas where your language is inadequate for their needs, or seems wrong to them. You also find areas where the precision of the domain language exposes contradictions or vagueness in their thinking.

The developers and user experts can informally test the model by walking through scenarios, using the model objects step-by-step. This is an opportunity for the developers and user experts to play with the model together, deepening each other's understanding and refining concepts as they go.

The domain experts can use the language of the model in writing use cases, and can work even more directly with the model by specifying acceptance tests.

The objection to collecting requirements in another terminology is subtler. The domain model will typically derive from the domain experts' jargon, but will be "cleaned up", to have sharper, narrower, definitions. The domain experts can and should learn these modified definitions, and should object if these definitions diverge from the meanings in the field. We'll deal with the coexistence of models on the same project in Chapter 12, Maintaining Model Integrity. Multiplicity of language is often necessary, but the division should never be between the domain experts and the developers.

This is not to say that a developer never uses technical terminology that a domain expert wouldn't understand. Of course, they have a huge jargon that they need to discuss the technical aspects of the system. Almost certainly, the users will also have specialized jargon that goes well beyond the narrow scope of the application and the understanding of the developers. But these are *extensions* to the language. These dialects should not contain alternative vocabularies for the same domain that reflect distinct models.

With a UBIQUITOUS LANGUAGE, conversations among developers, among domain experts, and expressions in the code itself are all based on the same language, derived from a shared domain model.

UBIQUITOUS LANGUAGE is Cultivated in the Intersection of Jargons

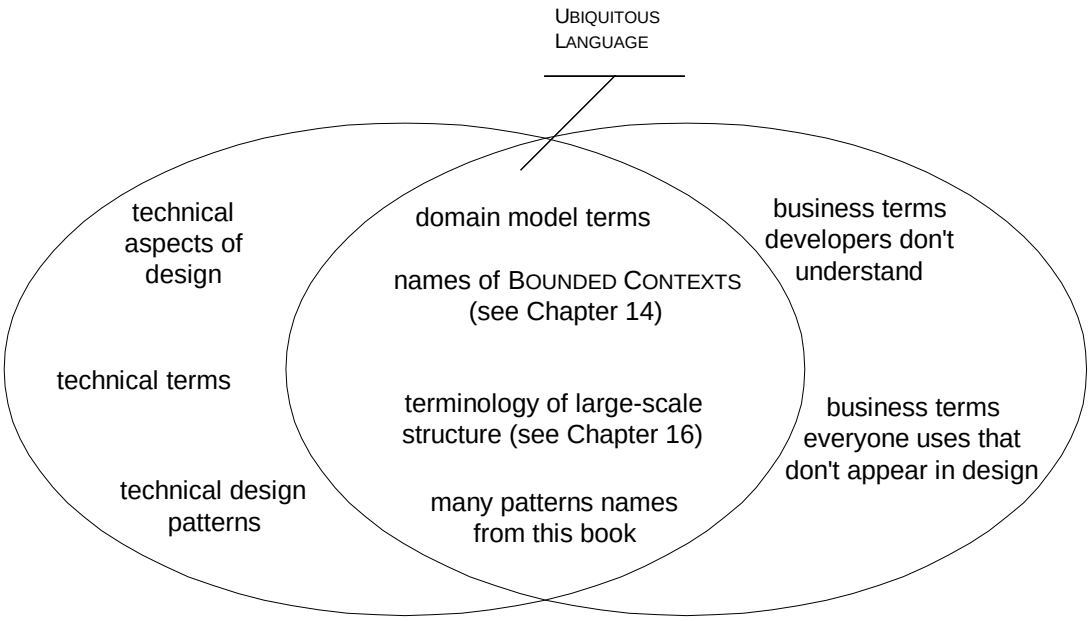


Figure 2. 3

Documents and Diagrams

When I'm in a meeting discussing a software design, I can hardly function without drawing on a white board or sketchpad. A good part of what I draw is UML diagrams, mostly class diagrams or object-interactions.

Some people are naturally visual, and diagrams help people visualize certain kinds of information. UML diagrams are pretty good at communicating relationships between objects, and fair at showing interactions. They do not convey the conceptual definitions of those objects. In a meeting, I would flesh that out in speech as I sketched the diagram, or it would emerge in a dialog with other participants.

Simple, informal UML diagrams can anchor a discussion. Sketch a diagram of three to five objects central to the issue at hand, and everyone can stay focused. Everyone will share a view of the relationships between the objects and, significantly, their names. The spoken discussion can be more effective with this aid. A diagram can be changed as people try different thought experiments, and the sketch takes on some of the fluidity of spoken words, a true part of the discussion. After all, UML stands for Unified Modeling *Language*.

The trouble comes when people feel compelled to convey the whole model or design through UML. A lot of object model diagrams are too complete and, simultaneously, leave too much out. They are too complete because people feel they have to put all the objects that they are going to code into a modeling tool. As soon as you have all that detail no one can see the forest for the trees.

Yet, in spite of all this detail, the attributes and relationships are only half the story of an object model. That is the half that UML handles most effectively. But the behavior of those objects and the constraints on them are not so easily illustrated. Object interaction diagrams can illustrate some tricky hotspots in the design, but the bulk of the interactions can't be shown that way. It is just too much work, both to create the diagrams and to read them. And an interaction diagram can still only imply the purpose behind the model.

To include constraints and assertions, UML falls back on text, placed in little brackets, inserted into the diagram.

Behavioral responsibilities of an object can be hinted at through operations names, and implicitly demonstrated with object interaction (or sequence) diagrams, but they cannot be *stated*. So, this falls to supplemental text or conversation. In other words, UML diagrams cannot convey two of the most important aspects of a model: the meaning of concepts it represents, and what they are meant to do. This doesn't trouble me because careful use of English (or Spanish, or whatever) can fill this role pretty well.

UML is not a very satisfying programming language. If you feel constrained by the capabilities of UML, you will often have to leave out the most crucial part of the model because it is some rule that doesn't fit into a box and line diagram. And, of course, a code generator cannot make use of those textual annotations. Every attempt I've seen to use the code generation capabilities of the modeling tools has been counterproductive. Even if you use some technology that allows executable programs to be written in a UML-like diagramming language, then the UML diagram is merely another way to view the program itself, and the very meaning of "model" is lost. If you use UML as your implementation language, you will still need other means of communicating the uncluttered model.

Diagrams are a means of communication and explanation, and they facilitate brainstorming. They serve these ends best if they are minimal. Comprehensive diagrams of the entire object model fail to communicate or explain. They overwhelm the reader with detail and they lack meaning. This leads us away from the all-encompassing object model diagram, or even the all-encompassing database of the model. It leads us toward simplified diagrams of conceptually important parts of the object model that are essential to understanding the design. The kinds of diagrams I've used in this book are representative of the kind of diagrams I use on projects. They simplify, they explain, and even incorporate a little non-standard notation when it clarifies. They show design constraints, but they are not design specifications in every detail. They represent the skeletons of ideas.

The vital detail about the design is captured in the code. A well-written implementation should transparently reveal the model underlying it. (Seeing that this happens is the subject of the next chapter and much of the rest of this book.) Supplemental diagrams and documents can guide people's attention to the central points. Natural language discussion can fill in the nuances of meaning. This is why I prefer to turn things inside out from the way a typical UML handles them. Rather than a diagram annotated with text, I write a text document illustrated with selective and simplified diagrams.

Always remember that *the model is not the diagram*. The diagram's purpose is to help communicate and explain the model. The code can serve as a repository of the details of the design. Well-written Java is as expressive as UML in its way. Carefully selected and constructed diagrams can serve to focus attention and aide navigation if they are not obscured by a compulsion to represent the model or design completely.

Written Design Documents

Once a document takes on a persistent form, it often loses its connection with the flow of the project. It is left behind by the evolution of the code, or it is left behind by the evolution of the language of the project. Still, there are valuable roles design documents can fill.

Each Agile process has its own philosophy about documents. Extreme Programming advocates using no extra design documents at all, and letting the code speak for itself. Running code doesn't lie, as any other document might. The behavior of running code is unambiguous.

Extreme Programming concentrates exclusively on the *active* elements of a program and executable tests. Comments added to the code do not effect program behavior, so they always fall out of synch with the active code and its driving model. External documents and diagrams do not affect the behavior of the program, so they fall out of synch. On the other hand, spoken communication and ephemeral diagrams on whiteboards do not linger to create confusion. This dependence on the code as communication medium motivates developers to keep the code clean and transparent.

Code as a design document does have its limits. It can overwhelm the reader with detail. While its behavior is unambiguous, that doesn't mean it is obvious. And the meaning behind a behavior can be hard to convey. In other words, it has some of the same basic problems as comprehensive UML diagrams. And developers are not the only people who need to understand the model.

Verbal communication supplements the code with meaning. But while talking is critical to connecting everyone to the model, a group of any size will probably need the stability and sharability of some written documents.

Many approaches can work. A few specific documents will be suggested much later, in Part IV of this book, but here I'll describe two general guidelines for evaluating a document.

First, a document shouldn't try to do what the code already does well. The code already supplies the detail. It already is the ultimately exact specification of program behavior.

It falls to other documents to illuminate meaning, to give insight into large-scale structures, and to focus attention on core elements. It can clarify design intent in a case where the programming language does not support a straight-forward implementation of a concept. Written documents should compliment the code and the verbal communication.

When I document a model in writing, I diagram small, carefully selected subsets of the model and surround them with text. I define the classes and their responsibility in words and frame them in a context of meaning as only a natural language can. But the diagram shows some of the choices that have been made in formalizing and paring down the concepts into an object model. These diagrams can be somewhat casual – even hand-drawn. In addition to saving labor, hand-drawn diagrams have the advantage of *feeling* casual and temporary. These are good things to communicate because they are generally true of our model ideas.

The greatest value of a design document is to explain the concepts of the model, perhaps its intended style of use, and to help in navigating the detail of the code. Depending on the philosophy of the team, the whole design document could be as simple as a set of sketches posted on the walls, or it could be substantial.

The second criterion is that a document must be involved in project activities. The easiest way to judge this is to observe its interaction with the UBIQUITOUS LANGUAGE. Is the document written in the language people speak on the project (now)? Is it written in the language embedded in the code?

Listen to the UBIQUITOUS LANGUAGE and how it is changing. If the terms explained in a design document don't start showing up in conversations and code, it is not fulfilling its purpose. Maybe the document is too big or complicated. Maybe it is not focused on a sufficiently important topic. Either people are not reading it or they are not finding it compelling. If it is having no impact on the UBIQUITOUS LANGUAGE, something is wrong.

You may hear the UBIQUITOUS LANGUAGE changing naturally while a document is being left behind. Evidently it does not seem relevant to people, or important enough to update. It could be archived as history, but if it is left active it could actually create confusion and hurt the project. Keeping it up to date through sheer will and discipline wastes effort, if the document isn't playing an important role.

The UBIQUITOUS LANGUAGE allows other documents, such as requirements specifications, to be more precise and unambiguous. As the domain model comes to reflect the most relevant knowledge of the business, application requirements become scenarios within that model, and the UBIQUITOUS LANGUAGE can be used to describe such a scenario in terms that directly connect to the MODEL-DRIVEN DESIGN. Specifications can often be simpler, since they do not have to convey business knowledge that is behind the model.

Now let's return to that desire of the XP community and some others to rely on the bedrock of the executable code and its tests. Much of this book discusses ways to make the code convey meaning through a MODEL-DRIVEN DESIGN (Chapter 3), and well written code can be very communicative, but this does not guarantee that what it communicates is accurate. Unlike the inescapable accuracy of the behavior caused by a section of code, a method name can be ambiguous, misleading, or out of date with the internals of the method.

This is a major selling point of approaches like declarative design (discussed in Chapter ##) in which a statement of the purpose of a program element determines its actual behavior in the program. The drive to generate programs from UML is partly motivated by this, though it generally hasn't worked out too well so far.

Accurately communicating the intent of code with current standard technology requires discipline and a certain way of thinking about design (discussed at length in Part III). To communicate effectively, the code is based on the same language that the requirements are written in -- the same language that the developers speak with each other and the customers.

It also takes discipline to use documents effectively. By keeping them minimal and focusing on complimenting code and conversation, documents can stay connected to the project. Let the UBIQUITOUS LANGUAGE and its evolution be your guide to choosing documents that live and get woven into the project's activity.

Explanatory Models

The thrust of this book is that one model should underlie implementation, design, and team communication. Having separate models for these separate purposes poses a hazard.

Models can also be valuable as education aides to teach about the domain. The model that drives the design is one view of the domain, but it may aide learning to have other views, used only as educational tools, to communicate general knowledge of the domain. For this purpose, people can use pictures that are not models, or other kinds of models unrelated to software design.

One particular reason other models are needed is scope. The technical model that drives the software development process must be strictly pared down to the necessary minimum to fulfill its functions. An explanatory model can include aspects of the domain that provide context that clarifies the more narrowly scoped model.

Explanatory models provide the freedom to create much more communicative styles tailored to a particular topic. Visual metaphors used by the domain experts in a field often provide clearer explanations, educating developers and harmonizing experts. It also presents the domain in a way that is simply different, and multiple different explanations help people learn.

There is no need for these to be object models, and it is generally best they not be. It is especially important to avoid UML. This avoids any false impression of correspondence with the software design. While there often is some correspondence between an explanatory model and the model that drives design, it will seldom be exact. To avoid confusion, everyone must be conscious of the distinction.

Example: Shipping Operations and Routes

Consider an application that tracks cargos for a shipping company. The model includes a detailed view of how port operations and vessel voyages are assembled into an operational plan for a cargo (a "route"). But to the uninitiated, a class diagram may not be too illuminating.

Class Diagram for Shipping Route

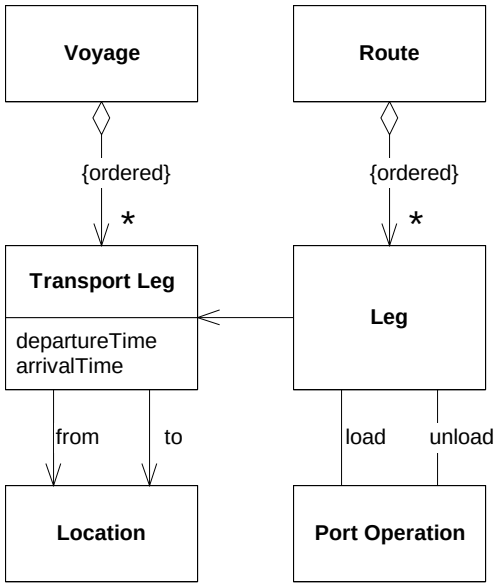


Figure 2. 4

In a case like this, an explanatory model can go a long way to helping team members understand what this actually means. Here is another way of looking at the same concepts.

Explanatory Model for Shipping Route

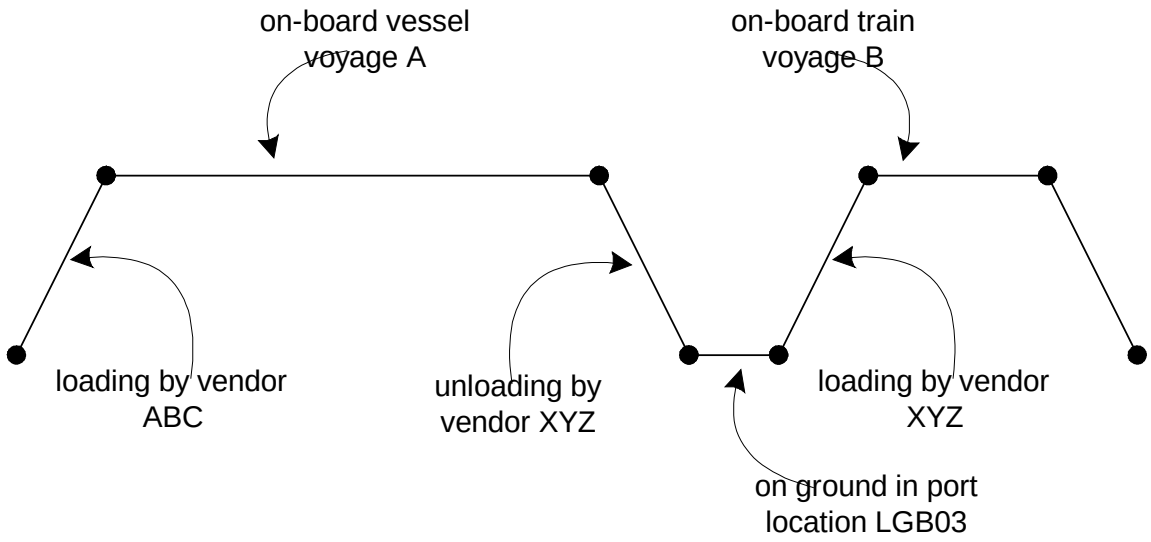


Figure 2. 5

Each line represents either a port operation (loading or unloading the cargo), or cargo sitting in storage on the ground, or cargo sitting on a ship in route.

This sort of diagram can help a developer, or a domain expert, understand the more rigorous software model diagrams. Together they are easier to understand than either of them alone.

3. Binding Model and Implementation

The first thing I saw as I walked in the door was a complete class diagram was printed on large sheets of paper that covered a large wall. It was my first day on a project where smart people had spent months carefully researching and developing a detailed model of the domain. The typical object in the model had intricate associations with three or four others, and this web of associations had few natural borders. In this, the analysts had been true to the nature of the domain.

As overwhelming as the diagram was, the model did capture some knowledge. After a moderate amount of study, I learned quite a bit (though that learning was hard to direct, much like randomly browsing the world-wide web.) I was more troubled to find that the study gave no insight into the application's code and design.

When the developers had undertaken to implement the application, they had quickly discovered that the tangle of associations, while navigable by a human analyst, didn't translate into storable, retrievable units that could be manipulated with transactional integrity. Mind you, this project was using an object database, so they did not even face the challenges of mapping objects into relational tables. At a fundamental level, the model did not provide a guide to implementation.

Since the model was "correct", the result of extensive collaboration between technical analysts and business experts, the developers reached the conclusion that conceptually based objects could not be the foundation of their design and they proceeded to develop an ad-hoc design that, while using a few of the same class names and attributes for data storage, was not based on that, or any, model.

The project had a domain model, but what good is a model on paper unless it directly aids the development of running software?

A few years later, I saw the same end result come from a completely different process. This project was to replace an existing C++ application with a new design implemented in Java. The old application had been hacked together without any regard for object modeling. The design, if there was one, had accreted as one capability after another had been laid on top of the existing code, without any noticeable generalization or abstraction.

The eerie thing was that the end product of these two processes was very similar! Both had functionality, but were bloated, very hard to understand, and eventually unmaintainable. Though the implementations had, in places, a kind of directness, you couldn't gain much insight about the purpose of the system by reading the code. Neither took any advantage of the object paradigm available in their development environment, except as fancy data structures.

Models come in many varieties and serve many roles, even restricted to the context of a software development project. Domain-driven design calls for a model that doesn't just aide early analysis, but is the very foundation of the design. This has some important implications for the code. What is less obvious is that it requires a different approach to modeling.

The astrolabe was a mechanical implementation of a model of the sky used to compute star positions.

Some development processes create an "analysis model", quite distinct from the design, and usually developed by different people. It is called an analysis model because it is the product of analyzing the business domain to organize its concepts without any consideration of the part it will play in a software system. It is meant as a tool for understanding only, and mixing in implementation concerns is thought to muddy this. Later, a design is created that may have only a loose correspondence to the analysis model. The independence of the two is, in such a process, a necessity. The analysis model was not created with design issues in mind and therefore is likely to be quite impractical for those needs.

Some knowledge crunching happens during such an analysis, but most of it is lost since the design cannot be directly based on the model. The developers are forced to reconceptualize the domain on their own, and there is no guarantee that the insights gained by the analysts and embedded in the model will be retained or rediscovered. And maintaining any mapping between the design and the loosely connected model is a labor that is not cost effective.

The pure analysis model even falls short of its primary goal of understanding the domain, because crucial discoveries always emerge during the design/implementation effort as very specific problems are encountered that were not anticipated. An upfront model will go into depth in some irrelevant places and will have overlooked

Ancient Greek astronomers devised an instrument called an astrolabe, which was perfected by medieval Islamic scientists. It incorporated an intricate arrangement of rotating disks that could be used to compute and predict the position in the sky of the sun and stars. Conversely, given a stellar or solar position (measured using sights optionally attached to the back of the astrolabe) the time could be calculated. A representation of a local spherical coordinate system was engraved on a plate, which could be replaced to represent the view from a different latitude. A rotating web (rete) represented the positions of the fixed stars on the celestial sphere. Rotating the rete against the plate enabled a calculation of celestial positions for any time and day of the year. The astrolabe was a mechanical implementation of an object-oriented model of the sky. More recently, star positions are calculated with computer software based on physical/mathematical models of Copernicus and Newton, but the astrolabe served well for over a thousand years.

others or taken a view of them that is not useful. The result is that pure analysis models get abandoned soon after coding starts, and most of the ground has to be covered again. But, if the perception is that analysis is a separate process, the second time around, analysis happens in a less disciplined way and may not be given adequate access to domain experts.

Projects that have no domain model at all, but just write code to fulfill one function after another, gain few of the advantages of knowledge crunching and communication discussed in the last two chapters. A complex domain will swamp them.

Many complex projects do attempt some sort of domain model, but don't maintain a tight connection between the model and the code. The model, possibly useful as an exploratory tool at the outset, becomes increasingly irrelevant and even misleading. All the care lavished on the model provides little reassurance that the design is correct, since the two are distinct. Meanwhile, software without a concept at the foundation of its design is, at best, a machine that does useful things without explaining its actions.

If the design, or some central part of it, does not map to the conceptual domain model, that model is of little value, and the correctness of the software is suspect. At the same time, complex mappings between models and design functions are difficult to understand, and, in practice, impossible to maintain as the design changes. A deadly divide opens between analysis and design so that insight gained in each of those activities does not feed into the other.

An analysis must capture fundamental concepts from the domain in a comprehensible, expressive way. The design has to specify a set of components that can be constructed with the programming tools we use that will perform efficiently in our deployment environment and will correctly solve the problems posed for the application.

MODEL-DRIVEN DESIGN discards the dichotomy of analysis model versus design to search out a single model that serves both purposes. Setting aside purely technical issues, each object in the design plays a conceptual role described in the model. This requires us to be more demanding of the chosen model, since it must fulfill two quite different objectives.

This can't be at the cost of a weakened analysis, fatally compromised by technical considerations. Nor can we accept clumsy designs, reflecting domain ideas but eschewing software design principles. It demands a model that works as both analysis and design. This is possible because there are always many ways of abstracting a domain. There are always many designs that could solve a stated problem. When the model doesn't seem to be practical for implementation, we must search for a new one. When the model doesn't faithfully express the key concepts of the domain, we must search for a new one. The modeling and design process then becomes a single iterative loop.

The design should retain as many of the concepts of the model as possible, as literally as possible, while the model should be chosen such that a practical design model can be created that corresponds to it. The imperative to closely relate the conceptual domain model to the design adds one more criterion for choosing the more useful models out of the universe of possible models. It calls for hard thinking and usually takes multiple iterations and a lot of refactoring, but it makes the model relevant.

Therefore,

Design a portion of the software system to reflect the domain model in a very literal way, so that mapping is obvious. Revisit the model and modify it to be implemented more naturally in software. Demand a single model that serves both purposes well. Have one model and tie the implementation slavishly to it. To do this usually requires software development tools and languages that support a modeling paradigm, such as object-oriented programming.

Although sometimes different models will exist to support different subsystems (see Chapter 14, Maintaining Model Integrity), sharing one set of concepts from analysis through all aspects of implementation within a given development effort, as reflected in the UBIQUITOUS LANGUAGE, gives the team leverage to solve problems using the model.

The model provides the language of the domain and the basic assignment of responsibilities. If new language is invented during implementation, *the model has changed*, and the change needs to be integrated back into model discussion.

The single model reduces the chances of error, since the design is now a direct outgrowth of the carefully considered model. The design, and even the code itself, has the communicativeness of a model. Design elements that are broken along conceptual lines tend to provide natural lines of division making refactoring easier.

♦ ♦ ♦

Developing a single model that captures the problem and provides a practical design is easier said than done. You can't just take any model and turn it into a workable design. The model has to be carefully crafted to make for a practical implementation. Design and implementation techniques have to be employed that allow code to express a model effectively (see Part II). Knowledge crunchers explore model options and refine them into practical software elements (Chapter 1). Development becomes an iterative process of refining the model, design and code as a single activity (see Part III).

Modeling Paradigms and Tool Support

To make a MODEL-DRIVEN DESIGN pay, the correspondence must be literal, exact within bounds of human error. To make such a close correspondence of model and design possible, it is almost essential to work within a modeling paradigm that has software tools that directly support it by allowing creation of direct analogs to the concepts in the paradigm.

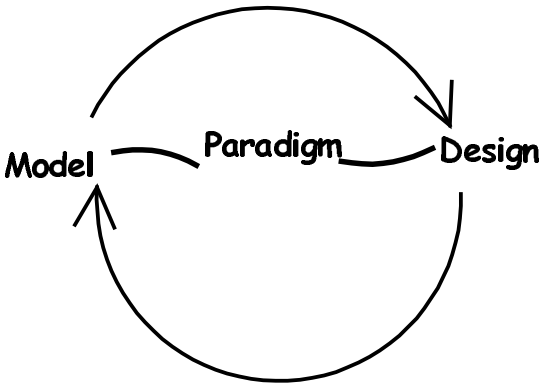


Figure 3. 1

Object-oriented programming is powerful because it is based on a modeling paradigm and provides implementations of the model constructs. As far as the programmer is concerned, objects really exist in memory, have associations with other objects, are organized into classes, and provide behavior available by messaging. Although many developers get benefit from just applying the technical capabilities of objects to organize program code, the real breakthrough of object design comes when the code expresses the concepts of a model. Java and many other tools allow the creation of objects and relationships directly analogous to conceptual object models.

Although it has never reached the mass usage that object-oriented languages have, the Prolog language is a natural fit for MODEL-DRIVEN DESIGN. In this case, the paradigm is logic, and the model is a set of logical rules and facts they operate on.

MODEL-DRIVEN DESIGN has limited applicability using languages like C because there is no modeling paradigm that corresponds to a purely procedural language. Those languages are "procedural", in the sense that the programmer tells the computer a series of steps to follow. While the programmer may be thinking

about the concepts of the domain, the program itself is a series of technical manipulations of data. The result may be useful, but the program doesn't capture much of the meaning. Procedural languages often support complex data types that begin to correspond to more natural conceptions of the domain, but these complex types are only organized data, and don't capture the active aspects of the domain. The result is that software is written as complicated functions linked together based on anticipated paths of execution, rather than by conceptual connections in the domain model.

Before I ever heard of object-oriented programming, I wrote FORTRAN programs to solved mathematical models have, which is just the sort of domain FORTRAN is targeted at. Mathematical functions are the main conceptual component of such a model and can be cleanly expressed. Even so, there was no way to capture higher-level meaning. Most non-mathematical domains don't lend themselves to MODEL-DRIVEN DESIGN in procedural languages because they are not conceptualized as math functions or as steps in a procedure.

Object-oriented design, the paradigm that currently dominates the majority of ambitious projects, is primarily used in this book.

Example: From Procedural to Model-Driven

As discussed in Chapter 1, a printed circuit board (PCB) can be viewed as a big collection of electrical conductors (called "nets") connecting the pins of various components. There are often tens of thousands of nets. Special software, called a "PCB layout tool", finds a physical arrangement for all the nets so that they don't cross or interfere with each other. It does this by optimizing their paths while satisfying an enormous number of constraints placed by the human designers that restrict the way they can be laid out. Although this is very sophisticated software, it still has some shortcomings.

One problem is that each of these thousands of nets has its own set of layout rules. PCB engineers see many of these nets as belonging to natural groupings that should share the same rules. For example, some nets form "buses".

Explanatory diagram of buses and nets

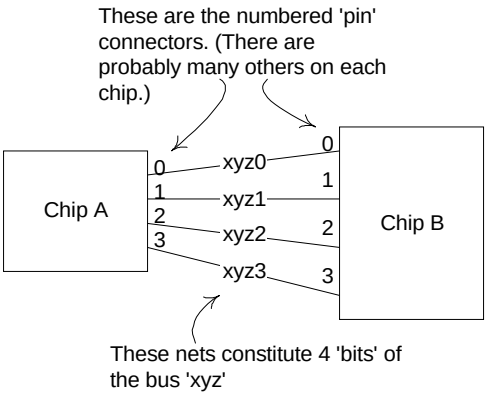


Figure 3. 2

By lumping nets into a bus, perhaps 8 or 16 or 256 at a time, the engineer cuts the job down to a more manageable size, improving productivity and reducing errors. The trouble is, the layout tool has no such concept as a "bus". Rules have to be assigned to tens of thousands of nets, one net at a time.

A Mechanistic Design

Desperate engineers worked around this limitation in the layout tool by writing scripts that parse the layout tool's data files and insert rules directly into the file, applying them to an entire bus at a time.

The layout tool stores each circuit connection in a "net list" file something like this:

```
Net Name    Component.Pin
--------    -------------
Xyz0        A.0, B.0
Xyz1        A.1, B.1
Xyz2        A.2, B.2
…
```

It stores the layout rules in a file format something like this:

```
Net Name    Rule Type       Parameters
--------    ---------       ----------
Xyz1         min_linewidth    5
Xyz1         max_delay        15
Xyz2         min_linewidth    5
Xyz2         max_delay        15
…
```

The engineers carefully use a naming convention for the nets so that an alphabetical sort of the data file will place the nets of a bus together in a sorted file. Then their script can parse the file, modify each net based on its bus. Actual code to parse, manipulate and write the files is just too verbose and opaque to serve this example, so I'll just list the steps in the procedure.

1. Sort net list file by net name
2. Read each line in file seeking first that starts with bus name pattern
3. For each line with matching name, parse line to get net name
4. Append net name with rule text to rules file
5. Repeat from 3 until left of line no longer matches bus name

So the input of a bus rule like this

```
Bus Name    Rule Type       Parameters
--------    ---------       ----------
Xyz          max_vias         3
```

would result in adding net rules to the file like these

```
Net Name    Rule Type       Parameters
--------    ---------       ----------
…
Xyz0         max_vias         3
Xyz1         max_vias         3
Xyz2         max_vias         3
…
```

I imagine that the person who first wrote such a script had only this simple need, and, if this were the only requirement, a script like this would make a lot of sense. But in practice there are now dozens of scripts. They could, of course, be refactored to share sorting, string matching, and if the language supported function calls to encapsulate the details, the scripts could begin to read almost like the summary steps above. But still, they are just file

manipulations. A different file format (and there are several) would require starting from scratch, even though the concept of grouping buses and applying rules to them is the same. If you wanted richer functionality or interactivity, you would have to pay for every inch.

What the scriptwriters were trying to do was to supplement the tool's domain model with the concept of "bus". Their implementation infers the bus's existence through sorts and string matches, but does not explicitly deal with the concept.

A MODEL-DRIVEN DESIGN

We've already discussed the concepts the domain experts use to think about their problems. Now we need to organize them explicitly into a model we can base software on.

Class diagram oriented toward efficient layout rule assignment

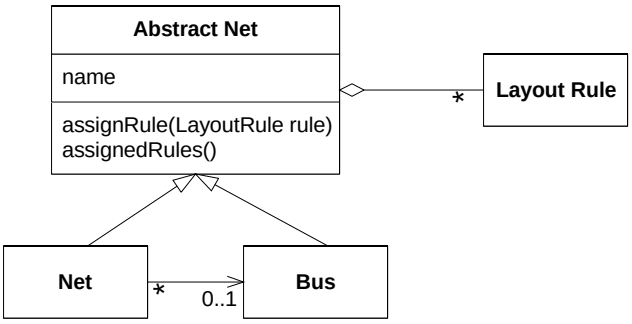


Figure 3. 3

With these objects implemented in an object-oriented language, the core functionality becomes almost trivial.

The `assignRule()` method can be implemented on **Abstract Net.** , The `assignedRules()` method on **Net** takes its own rules and it's **Bus**'s rules.

```
abstract class AbstractNet {
    private Set rules;

    void assignRule(LayoutRule rule) {
        rules.add(rule);
    }

    Set assignedRules() {
        return rules;
    }
}

class Net extends AbstractNet {
    private Bus bus;

    Set assignedRules() {
        Set result = new HashSet();
        result.addAll(super.assignedRules());
        result.addAll(bus.assignedRules());
        return result;
    }
}
```

Of course, there would be a great deal of supporting code, but this covers the basic functionality of the script.

The application requires import/export logic, which we'll encapsulate into some simple services.

Service	Responsibility
Net List import	Reads Net List file, creates instance of Net for each entry
Net Rule export	Given a collection of Nets, writes all attached rules into the Rules File

We'll also need a few utilities:

Class	Responsibility
Net Repository	Provides access to Nets by name
Inferred Bus Factory	Given a collection of Nets, uses naming conventions to infer Buses, creates instances
Bus Repository	Provides access to Buses by name

Now, starting the application is a matter of initializing the repositories with imported data:

```
Collection nets = NetListImportService.read(aFile);
NetRepository.addAll(nets);
Collection buses = InferredBusFactory.groupIntoBuses(nets);
BusRepository.addAll(buses);
```

Each of the services and repositories can be unit tested. Even more important, the core domain logic can be tested. Here is a unit test of the most central behavior (using the JUnit test framework):

```
public void testBusRuleAssignment() {
    Net a0 = new Net("a0");
    Net a1 = new Net("a1");
    Bus a = new Bus("a"); //Bus is not conceptually dependent on the
    a.addNet(a0);         //name-based recognition, and so its tests
    a.addNet(a1);         //should not be either.

    NetRule minWidth4 = NetRule.create(MIN_WIDTH, 4);
    a.assignRule(minWidth4);

    assertTrue(a0.assignedRules().contains(midWith4));
    assertEquals(minWidth4, a0.getRule(MIN_WIDTH));
    assertEquals(minWidth4, a1.getRule(MIN_WIDTH));
}
```

An interactive user interface could present a list of buses, allowing the user to assign rules to each, or it could read from a file of rules for backward compatibility. A façade makes access simple for either interface. Its implementation echoes the test:

```
public void assignBusRule(String busName, String ruleType, decimal parameter);
    Bus bus = BusRepository.getByName(aBusName);
```

```
    bus.assignRule(NetRule.create(ruleType, parameter));
}
```

Finishing:

```
NetRuleExport.write(aFileName, NetRepository.allNets());
```
(Each **Net** will be asked for `assignedRules()`)

Of course, if there were only one operation (as in the example), the script-based approach might be just as practical. But in reality, there were be 20 or more. The MODEL-DRIVEN DESIGN scales easily and can include constraints on combining rules and other enhancements.

The second design also accommodates testing. It is broken into components with well-defined interfaces that can be unit tested. The only way to test the script is an end-to-end file-in/file-out comparison.

Keep in mind that such a design does not emerge in a single step. It would take several iterations of refactoring and knowledge crunching to distill the important concepts of the domain into a simple, incisive model.

Letting the Bones Show; Why Models Matter to Users

In theory, perhaps, you could present a user with any view of the system, regardless of what lies beneath. But in practice, a mismatch causes confusion at best -- bugs at worst. Consider a very simple example of how users are misled by superimposed models of bookmarks for websites in current releases of Microsoft Internet Explorer.[1]

A user of Internet Explorer thinks of “favorites” as a list of names of websites that persist from session to session. But the implementation treats a favorite as a file containing a URL, and whose filename is put in the favorites list. That's a problem if the web page title contains characters that are illegal in Windows file names. Suppose a user tries to store a favorite and types the following name for it: “Laziness: The Secret to Happiness”. The error message will say, “A filename cannot contain any of the following characters: \ / : * ? ” < > | ”. What file name? On the other hand, if the website title already contains an illegal character, Internet Explorer will just quietly strip it out. The loss of data may be benign in this case, but not what the user would have expected. Quietly changing data is a big risk in most applications

MODEL-DRIVEN DESIGN calls for working with only one model (within any single context, as will be discussed in Chapter 15). Most of the advice and examples go to the problems of having separate analysis models and design models, but here we have a problem arising from a different pair of models: the user model and the design/implementation model.

Of course, an unadorned view of the domain model would definitely not be convenient for the user in most cases. But trying to create in the UI an illusion of a model other than the domain model will cause confusion unless the illusion is perfect. If web favorites are actually just a collection of shortcuts files, then expose this to the user and eliminate the confusing alternative model. Not only will it be less confusing, but the user can then leverage what he knows about the file-system to deal with web favorites. He can reorganize them with the file explorer, for example, rather than using awkward tools built into the web browser. They would be more aware of the possibility of storing web shortcuts anywhere in the file system. Just by removing the misleading extra model, the power of the application would be increased and made easier to understand. Why make the user learn a new model when the programmers felt the old model was good enough?

[1] Brian Marick mentioned this example to me.

Alternatively, store the favorites in a different way, say in a data file, so that they can have their own rules. Those rules would, presumably, be the naming rules that apply to web pages. That would again provide a single model. This one tells the user that everything he knows about naming websites applies to favorites.

When a design is based on a model that reflects the basic concerns of the users and domain experts, the bones of the design can be revealed to the user to a greater extent than with other design approaches. Revealing the model gives the user more access to the potential of the software and yields consistent, predictable behavior.

HANDS-ON MODELERS

Manufacturing is a popular metaphor for software development. One inference from this metaphor: highly skilled engineers design; less-skilled laborers assemble the products. This metaphor has messed up a lot of projects for one simple reason – software development is *all* design. All teams have specialized roles, but over-separation of responsibility for analysis, modeling, design, and programming interferes with MODEL-DRIVEN DESIGN.

On one project my job was to coordinate different application teams and help develop the domain model that would drive the design. But the management thought that modelers should be modeling, and coding was a waste of those skills, so I was, in effect, forbidden to program or work on details with programmers.

Things seemed to be ok for a while. Working with domain experts and the development leads of the different teams, we crunched knowledge and refined a nice core model. But that model was never put to work for two reasons.

First, some of the intent of the model was lost in the handoff. The overall effect of a model can be very sensitive to details (as will be discussed in Parts II and III), and those don't always come across in a UML diagram or general discussion. If I could have rolled up my sleeves and worked with the other developers directly, providing some code to follow as examples, and providing some close support, the team could have taken up the abstractions of the model and run with them.

The other problem was the indirectness of feedback from the interaction of the model with the implementation and the technology. For example, certain aspects of the model turned out to be wildly inefficient on our technology platform, but the full implications didn't trickle back to me for months. Relatively minor changes could have fixed the problem, but, by then, it didn't matter. The developers were well on their way to writing software that did work – without the model, which had been reduced to a data structure where it was still used at all. They had thrown the baby out with the bathwater, but what choice did they have? They could no longer risk being saddled with the dictates of the ivory-tower architect.

The circumstances of this project were about as favorable to a hands-off modeler as they ever are. I already had extensive hands-on experience with most of the technology used on the project. I had even led a small development team on the same project before my role changed, so I was familiar with the project's development process and programming environment. Those factors were not enough to make me effective.

When a modeler is separated from the implementation process, he or she never acquires, or quickly loses, a feel for the constraints of implementation. The basic constraint of MODEL-DRIVEN DESIGN – that the model supports an effective implementation and abstracts key insights into the domain – is half gone, and the resulting models will be impractical. Meanwhile, if the people who write the code do not feel responsible for the model, or don't understand how to make the model work for an application, then the model has nothing to do with the software. If developers don't realize that changing code changes the model, then their refactoring will weaken the model rather than strengthen it. Finally, the knowledge and skills of experienced designers won't be transferred to other developers if the division of labor prevents the kind of collaboration that conveys the subtleties of coding a MODEL-DRIVEN DESIGN.

The need for HANDS-ON MODELERS does not mean that team members cannot have specialized roles. Every Agile process, including Extreme Programming, defines roles for team members, and other informal specializations tend to emerge naturally. The problem arises from separating two tasks that are coupled in a MODEL-DRIVEN DESIGN, modeling and the implementation.

The effectiveness of an overall design is very sensitive to the quality and consistency of the fine-grained design and implementation decisions. With MODEL-DRIVEN DESIGN, a portion of the code is an expression

of the model, so changing that code changes the model. Programmers are modelers, whether anyone likes it or not. So it is better to set up the project so that the programmers do good modeling work.

Therefore,

Any technical person contributing to the model must spend some time touching the code, whatever primary role he or she plays on the project. Anyone responsible for changing code must learn to express a model through the code. Every developer must be involved in some level of discussion about the model and have contact with domain experts. Those who contribute in different ways must consciously engage those who touch the code in a dynamic exchange of model ideas through the UBIQUITOUS LANGUAGE.

♦ ♦ ♦

The sharp separation of modeling and programming doesn't work, yet large projects still need technical leaders who coordinate high-level design and modeling and help work out the most difficult or most critical decisions. Section IV, "Strategic Design", deals with such decisions and should stimulate ideas for more productive ways to define roles and responsibilities of high-level technical people.

Domain-driven design puts a model to work to solve problems for an application. Through knowledge crunching, a team distills a torrent of chaotic information into a practical model. A MODEL-DRIVEN DESIGN intimately connects the model and the implementation. The UBIQUITOUS LANGUAGE is the channel for all that information to flow between developers, domain experts and the software.

The result is software that provides rich functionality based on a fundamental understanding of the core domain.

Part II. Building Blocks of a Model-Driven Design

To keep the implementation crisp and in lock-step with the model, in spite of messy realities, calls for applying the best practices of object modeling and design. Domain-driven design involves a shift of emphasis of conventional ideas. The following chapters collect specific best practices of object modeling and cast them in the light of domain-driven design.

But the main point of this section is on the kind of decisions that keep the model and implementation aligned with each other, reinforcing each other's effectiveness. This alignment requires attention to the detail of individual elements. Careful crafting at this small scale gives developers a steady platform to apply the modeling approaches of Parts III and IV

This book is not an introduction to object-oriented design. Nor does it propose radical design fundamentals. The design style largely follows the principle of "responsibility-driven design", put forward in [WWW 1990] and updated in [WM 2003]. It also relies heavily (especially in Chapter 10) on the ideas of "design by contract", described in [Meyer 1988]. It draws on the general background of other widely held best practices of object-oriented design, which are described in such books as [Larman 1998].

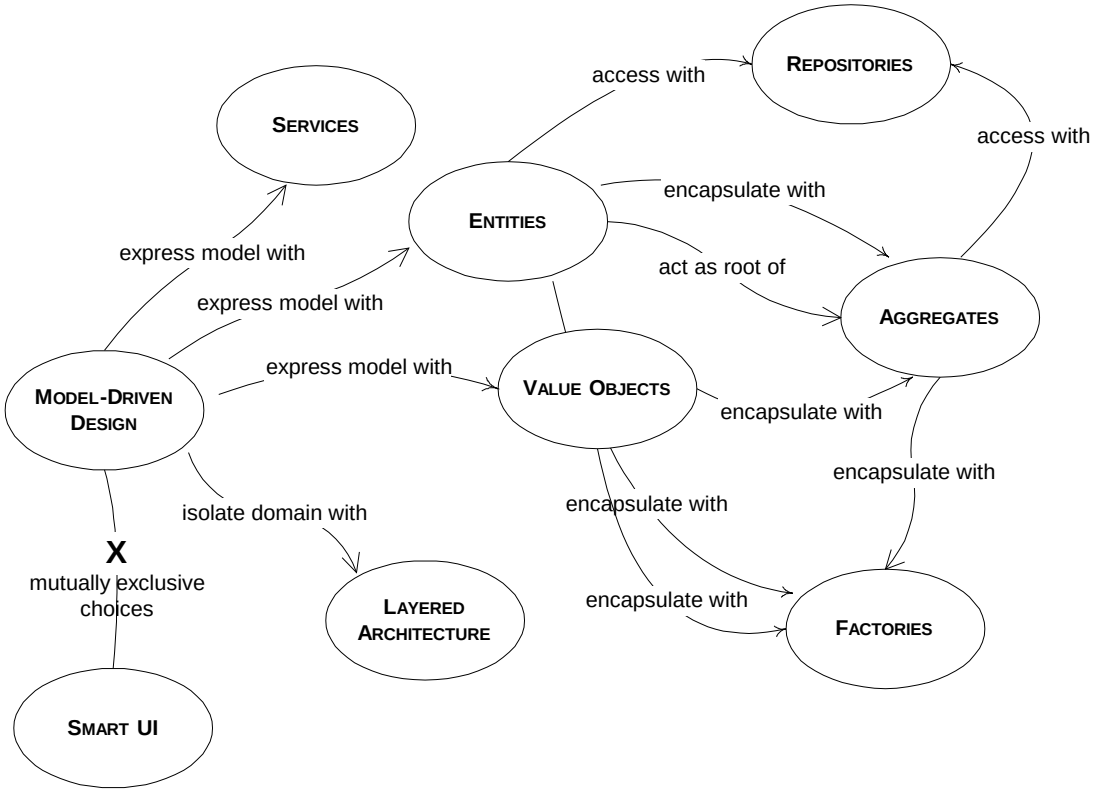
From that starting point, Part II looks at what it takes to keep a focus on the model in the face of bumpy realities that may seem to make those principles inapplicable. These techniques are organized as a "pattern language" (see Appendix 1). Sharing these standard patterns brings order to the design and makes it easy for team members to understand each other's work. Using standard patterns also adds to the UBIQUITOUS LANGUAGE, which all team members can use to discuss model and design decisions.

Developing a good domain model is an art. But practical design and implementation of the individual elements of a model can be relatively systematic, and the modeling process is improved by clarity of distinctions and definitions within the model, and by isolating the domain design from the mass of other concerns in the software system.

Elaborate models only cut through complexity if care is taken with the fundamentals, producing detailed elements that can be confidently combined.

.

Navigation Map of the Language of MODEL-DRIVEN DESIGN



Part II presents a pattern *language* of current best practices for the basic construction of domain models. This diagram shows the patterns that will be presented in Part II, and gives a quick indication of how they relate to each other.

4. Isolating the Domain

The part of the software that specifically solves problems from the domain usually constitutes only a small portion of the software of a system, although its importance is disproportionate to its size. To apply our best thinking, we need to be able to look at the elements of our model and see them as a system, not be distracted by the process of picking them out of a much larger mix of objects, like identifying constellations in the sky. This requires decoupling the domain objects from other functions of the system so they are not lost in the mass and so domain concepts are not blurred and confused with concepts related to software technology.

Sophisticated techniques for this isolation have emerged. This is well-trodden ground, but it is so critical to the successful application of domain modeling principles that it must be reviewed briefly, from a domain-driven point of view.

LAYERED ARCHITECTURE

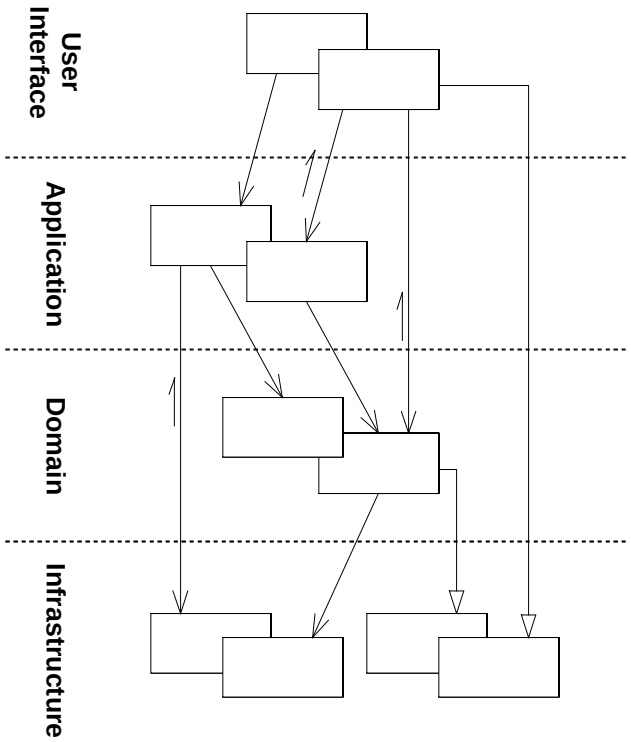


Figure 4. 1

For a shipping application to support the simple user act of selecting a cargo's destination from a list of cities, there must be program code to draw a widget on the screen, query the database for all the possible cities, interpret the user's input and validate it, associate the selected city with the cargo and commit the change to the database. All of this code is part of the same program, but only a little of it is related to the business of shipping.

Software programs involve design and code to carry out many different kinds of tasks. They take user input, carry out business logic, access databases, communicate over networks, display information to users, and so on. So the code involved in each program function can be substantial.

In an object-oriented program, UI, database and other support code often gets written directly into the business objects. Additional business logic is embedded in the behavior of UI widgets and database scripts. This happens because it is the easiest way to make things work, in the short run.

When the domain related code is diffused through this large amount of other code, it becomes extremely difficult to see and to reason about. Superficial changes to the UI can actually change business logic. Changing business rules may require meticulous tracing of UI code, database code, or other program elements. Implementing coherent model-driven objects becomes impractical. Automated testing is awkward. With all those technologies and logic involved in each activity, a program must be kept very simple or it becomes impossible to understand.

Creating programs that can handle very complex tasks calls for separation of concerns, allowing concentration on different parts of the design in isolation. At the same time, the intricate interactions within the system must be maintained in spite of the separation.

Many divisions of a system can be imagined, but through experience and convention, the industry has converged on LAYERED ARCHITECTURES, and specifically a few fairly standard layers. The metaphor of layering is so widely used that it feels intuitive to most people. Many good discussions of layering are available in the literature, sometimes in the format of a pattern as in [BMRSS96], pp. 31-51. The essential principle is that an element of a LAYER has dependencies only on other elements in the same layer or on elements of the layers "beneath" it. Communication upward must be through some indirect mechanism, which I'll discuss a little later.

The value of layers is that each specializes in a particular aspect of a computer program. This allows more cohesive designs of each aspect, and makes these designs much easier to interpret. Of course, it is vital to choose layers that isolate the most important cohesive design aspects. Again, experience and convention has led to some convergence. While there are many variations, most successful architectures use some version of these four conceptual layers:

User Interface **(aka Presentation Layer)**	Responsible for showing information to the user and interpreting the user's commands. The external actor might sometimes be another computer system rather than a human user.
Application Layer	Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.
Domain Layer **(aka Model Layer)**	Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. *This layer is the heart of business software.*
Infrastructure Layer	Provide generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, etc. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.

Some projects don't make a sharp distinction between User Interface and Application. Others have multiple Infrastructure layers. But it is the crucial separation of the domain layer that enables MODEL-DRIVEN DESIGN.

Therefore,

Partition a complex program into LAYERS. Develop a design within each LAYER that is cohesive and is dependent only on the layers below. Follow standard architectural patterns to provide loose coupling to the layers above. Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code. The domain objects, free of the

responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model. This allows a model to evolve rich enough and clear enough to effectively capture essential business knowledge and put it to work.

Separating the domain from the infrastructure and user interface layers allows a much cleaner design of each layer. Isolated, they will be much less expensive to maintain, since they tend to evolve at different rates and respond to different needs. The separation also helps with deployment in a distributed system, by allowing different layers to flexibly be placed in different servers or clients, in order to minimize communication overhead and improve performance [Fowler96].

Example: Partitioning Online Banking Functionality into Layers

An application provides various capabilities for maintaining bank accounts. One feature is funds transfer, where the user enters or chooses two account numbers and an amount of money and then initiates the transfer.

To make this example manageable, major technical features, most notably security, are omitted. The domain design is also oversimplified. (Realistic complexity would only increase the need for layered architecture.) Furthermore, the particular infrastructure implied here is only for example purposes—not a suggested design. But, given all those caveats, the responsibilities of the remaining functionality are in the appropriate layers.

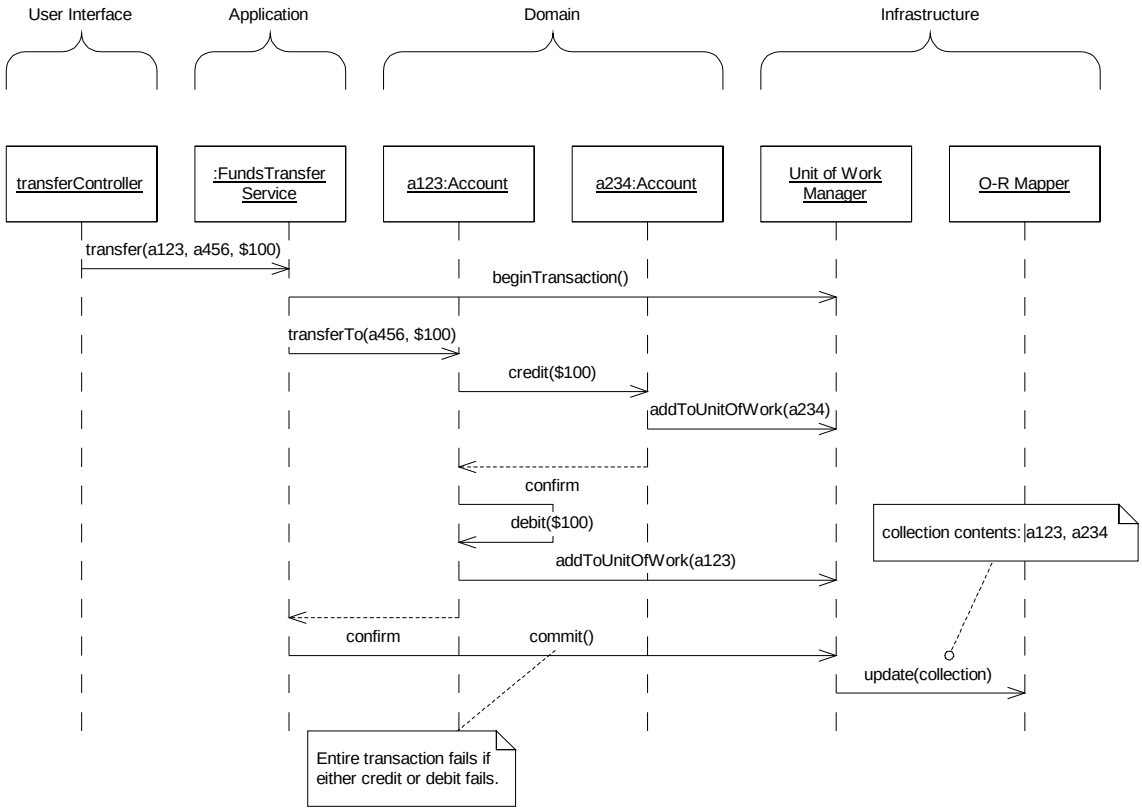


Figure 4. 2

Note that the domain layer, *not the application layer*, is responsible for fundamental business rules – in this case “every credit has a matching debit”.

The application also makes no assumptions about the source of the transfer request. The example initially described an application that presumably had entry fields and buttons for

account numbers, amounts and commands. But that user interface could be replaced by a wire request in XML without affecting the application or any of the lower layers. This is important not because projects frequently need to replace user interfaces with wire requests, but because a clean separation of concerns keeps the design of each layer easy to understand and maintain.

In fact, this diagram partially illustrates the problem of not isolating the domain. Because everything from the request to transaction control had to be included, the domain layer had to be dumbed down to keep the overall interaction simple enough to follow. If we were focused on the design of the isolated domain layer, we would have space on the page and in our heads for a model that better represented the domain's rules, perhaps including ledgers, credit and debit objects, or monetary transaction objects.

Relating the Layers

So far the discussion has focused on the separation of LAYERS and the way in which that improves the design of each aspect of the program, particularly the domain layer. But, of course, the LAYERS have to be connected. To do this without losing the benefit of the separation is the motivation of a number of patterns.

LAYERS are meant to be loosely coupled, with design dependencies in only one direction. Upper LAYERS can use or manipulate elements of lower ones straight-forwardly by calling their public interfaces, holding references to them (at least temporarily) and generally using conventional means of interactions. But when an object of a lower level needs to communicate upward (beyond answering a direct query) we need another mechanism, drawing on architectural patterns for relating layers such as callbacks or OBSERVERS [GHJV95].

The grandfather of patterns for connecting the UI to the application and domain layers is MODEL-VIEW-CONTROLLER (MVC). It was pioneered in the Smalltalk world back in the 1970s and has inspired many of the UI architectures that followed. [Fowler2002] discusses this pattern and several useful variations on the theme. [Larman98] discusses these concerns in the MODEL-VIEW SEPARATION PATTERN, and his APPLICATION COORDINATOR is one approach to connecting the application layer.

There are other styles connecting the UI and the application, and for our purposes they are all fine as long as they maintain the isolation of the domain layer, allowing domain objects to be designed without simultaneously thinking about the user interface that might interact with them.

The infrastructure layer usually does not initiate action in the domain layer. Being "below" the domain layer, it should have no specific knowledge of the domain it is serving. Indeed, such technical capabilities are most often offered as SERVICES. For example, if an application needs to send an email, some message-sending interface can be located in the infrastructure layer and the application layer elements can request the transmission of the message. This decoupling gives some extra versatility. The message-sending interface might be connected to an email sender, a fax sender or whatever is available. But the main benefit is simplifying the application layer, keeping it narrowly focused on its job – knowing *when* to send a message, but not burdened with *how*.

The application layer and domain layer call on the services provided by the infrastructure layer, and when the scope of a service has been well chosen and its interface well-designed, the caller can remain loosely coupled and uncomplicated by the elaborate behavior encapsulated by the interface.

But not all infrastructure is in the form of services callable from the higher layers. Some technical components are designed to directly support the basic functions of other layers (such as domain objects in the domain layer) and provide the mechanisms for them to relate (implementations of MVC and the like). Such an "architectural framework" has much more impact on the design of the other parts of the program.

Architectural Frameworks

When infrastructure is provided in the form of SERVICES called on through interfaces, it is fairly intuitive how the layering works and how to keep the layers loosely coupled. But some technical problems call for

more intrusive forms of infrastructure. Frameworks that integrate many infrastructure needs often require the other layers to be implemented in very particular ways, as a subclass of a framework class or with structured method signatures. (It is counterintuitive that a subclass is in a higher layer that the parent class, but keep in mind which class reflects more knowledge of the other.) The best architectural frameworks solve complex technical problems while allowing the domain developer to concentrate on expressing a model. But they can easily get in the way, either by making too many assumptions that constrain domain design choices or by making the implementation so heavyweight that development slows down.

Some form of architectural framework usually is needed (though sometimes teams choose frameworks that don't serve them very well). When applying a framework, the team needs to keep focused on the goal of an implementation that expresses a domain model and uses it to solve important problems. The team must seek ways of employing the framework to those ends. This may mean not using all the framework's features. For example, early J2EE applications often implemented all domain objects as "entity beans". This approach bogged down both performance and development pace. Instead, current best-practice is to use the J2EE framework for larger grain objects, implementing most business logic with generic Java objects. A lot of the downside of frameworks can be avoided by selectively applying them to solve difficult problems without looking for a one-size-fits-all solution. This keeps the implementation less coupled to the framework, which gives some flexibility in later design decisions, but more importantly, since many of the current frameworks are very complicated to use, keeps the business objects readable and expressive.

Architectural frameworks and other tools will continue to evolve that will automate or prefabricate more and more of the technical aspects of an application so that, increasingly, the effort of application development will be spent on modeling the core business problems, greatly improving productivity and quality. As we move in this direction, we must guard against our enthusiasm for technical solutions that straightjacket application developers.

The Domain Layer is Where the Model Lives

LAYERED ARCHITECTURE is used in most systems today with various layering schemes. It can be valuable to other styles of development. However DOMAIN-DRIVEN DESIGN requires one particular layer to exist.

The domain model is a set of concepts. The "domain layer" is the manifestation of that model and all directly related design elements. The design and implementation of business logic constitute the domain layer. In a MODEL-DRIVEN DESIGN, the software constructs of the domain layer will mirror the model concepts.

It is not practical to achieve that correspondence when the domain logic is mixed with other concerns of the program. Isolating the domain implementation is a prerequisite for a domain-driven design.

…That sums up the widely accepted LAYERED ARCHITECTURE pattern for object applications. But this separation of UI, application, and domain is so often attempted and so seldom accomplished that its negation deserves a discussion in its own right. The truth is that many software projects do take and should continue to take a much less sophisticated design approach that I call the SMART UI. But if that road is taken, most of what is in this book is not applicable. This is an alternate, mutually exclusive branch in the road, incompatible with the approach of domain-driven design. My interest is in the situations where the SMART UI does not apply, which is why I call it, with tongue in cheek, an "anti-pattern". Discussing it here provides a useful contrast and will help clarify the circumstances that justify the more difficult path taken in the rest of the book.

♦ ♦ ♦

A project needs to deliver simple functionality, dominated by data-entry and display with few business rules. Staff is not composed of advanced object modelers.

If an unsophisticated team with a simple project decides to try a MODEL-DRIVEN DESIGN with LAYERED ARCHITECTURE, they will have a difficult learning curve. They will have to master complex new technologies and stumble through the process of learning object modeling (which is challenging even with the help of this book!). The overhead of managing infrastructure and layers makes very simple tasks take longer. Simple projects come with short time lines and modest expectations. Long before they complete the assigned task, much less demonstrate the exciting possibilities of their approach, the project will have been canceled.

Even if they are given more time, they are likely to fail to master the techniques without expert help. And in the end, if they do surmount these challenges, they will have produced a simple system. Rich capabilities were never requested.

This is not to say that a more experienced team would face the same tradeoffs. Their learning curve would be short and the extra time taken to manage layers reduces with experience. Also, projects with big ambitions need to start with simple functionality and work their way up. But even those first tentative steps will be MODEL-DRIVEN and isolate the domain layer. The point is that the approach advocated in the rest of this book pays off for ambitious projects, and requires strong skills. And not all projects are ambitious or can muster those skills.

Therefore, when circumstances warrant,

Put all the business logic into the user interface. Chop the application into small functions and implement them as separate user interfaces, embedding the business rules into them. Use a relational database as a shared repository of the data. Use the most automated UI building and visual programming tools available.

Heresy! The gospel (as advocated everywhere, including elsewhere in this book) is that domain and UI should be separate. In fact, it is difficult to apply any of the methods discussed later in this book without that separation. Therefore, this might be considered an "anti-pattern", but it isn't always, and it is important to understand why we want to separate application from domain, even including when we might not want to. In truth, there are advantages to the SMART UI and situations where it works best, which partially account for why it is so common.

Advantages

- Productivity is high and immediate for simple applications.
- Less capable developers can work this way with little training.
- Even deficiencies in requirements-analysis can be overcome by releasing a prototype to users and then quickly changing the product to fit their requests.

- Applications are decoupled from each other so that delivery schedules of small modules can be planned relatively accurately, and expansion of the system with additional, simple behavior is easy.
- Relational databases work well and provide integration at the data level.
- 4-GL tools work well.
- When applications are handed off, maintenance programmers will be able to quickly redo portions they can't figure out since the effects should be localized to the UI being worked on.

Disadvantages

- Integration of applications is difficult except through the database.
- There is no reuse of behavior and no abstraction of the business problem
- Rapid prototyping and iteration reach a natural limit because the lack of abstraction limits refactoring options.
- Complexity buries you quickly, so the growth path is strictly toward additional simple applications. There is no graceful path to richer behavior.

If this pattern is applied consciously, a team can avoid taking on a great deal of overhead that is required to attempt other approaches. The common, costly mistake is to build an infrastructure and use tools much heavier weight than are needed, or to undertake a sophisticated design approach that you aren't committed to carrying all the way.

Most flexible languages (such as Java) are overkill for these applications and will cost you dearly. A 4-GL style tool is the way to go. Remember, one of the consequences of this pattern is that you can't migrate to another design approach except by replacement of entire applications, and that integration is only through the database. Therefore, a later attempt to use Java will not be helped very much by the use of Java in the initial development. Don't think you're building a flexible system just because you're using a flexible language.

♦ ♦ ♦

The SMART UI is discussed only to clarify why and when a pattern such as LAYERED ARCHITECTURE is needed in order to isolate a domain layer. The lack of decoupling can really be a disaster if applied in an inappropriate setting, but if the project does not have the necessary expertise for the more sophisticated approaches, and if it can meet the other requirements of the SMART UI, it provides an option. If not, bite the bullet, get the necessary experts and *avoid* the SMART UI.

There are other solutions in between these two. For example, [Fowler2002] describes the TRANSACTION SCRIPT, which separates UI from application, but does not provide for an object model. The bottom line is this: *If the architecture isolates the domain related code in a way that allows a cohesive domain design loosely coupled to the rest of the system then it can probably support domain-driven design.* Other development styles have their place, but you must accept varying limits on complexity and flexibility.

Other Kinds of Isolation

Unfortunately, infrastructure, user interface and application are not the only things you need to protect your delicate domain model from. You must also control the interfaces to other domain components that are not fully integrated into your model. Chapter 14, "Maintaining Model Integrity", deals with this, introducing such patterns as BOUNDED CONTEXT and ANTICORRUPTION LAYER. Chapter 15, "Distillation", will discuss distinctions within the domain layer that can unencumber the essential concepts of the domain from peripheral detail.

But that comes later. Next, we'll look at the nuts and bolts of coevolving the details of an effective domain model and an expressive implementation. After all, the best part of isolating the domain is getting all that other stuff out of the way so that we can really focus on the domain design…

5. A Model Expressed in Software

To compromise in implementation without losing the punch of a MODEL-DRIVEN DESIGN requires a reframing of the basics. Connecting model and implementation has to be done at the detail level. This chapter focuses on those individual model elements, getting them in shape to support the activities in later chapters.

This will start with the issues of designing and streamlining associations. Associations between the objects are simple to draw and conceive, but a potential quagmire to implement. They illustrate how crucial detailed implementation decisions are to the viability of a MODEL-DRIVEN DESIGN.

Turning to the objects themselves, but continuing the scrutiny of the relationship of detailed model choices and implementation concerns, I'll focus on making distinctions among the three patterns of model elements that express the model: ENTITIES, VALUE OBJECTS and SERVICES.

Defining objects that capture concepts of the domain seems very intuitive on the surface, but serious challenges are lurking in the shades of meaning. Certain distinctions have emerged that clarify the meaning of model elements and tie into a body of design practices for carving out specific kinds of objects.

Does an object represent something with continuity and identity -- something that is tracked through different states or even across different implementations? Or is it an attribute that describes the state of something else? This is the basic distinction between an ENTITY and a VALUE OBJECT. Defining objects that clearly follow one pattern or the other makes the objects less ambiguous and lays out the path to specific choices for robust design.

Then there are those aspects of the domain that are more clearly expressed as actions or operations, rather than as objects. Although it is a slight departure from object-oriented modeling tradition, it is often best to express these as SERVICES, rather than forcing responsibility for an operation onto some ENTITY or VALUE OBJECT. A SERVICE is something that will be done for a client on request. In the technical layers of the software there are many SERVICES. They emerge in the domain also, when some activity is modeled that corresponds to something the software must do, but not to state, as with an object.

Also, there are guidelines for staying on course when you are forced to deal with the messy realities of implementation, where the purity of the object model must be compromised, as for storage in a relational database.

Finally, a discussion of MODULES will drive home the point that every design decision should be motivated by some insight into the domain. The ideas of high-cohesion and low coupling, often thought of as technical metrics, can be applied to the concepts themselves. In a MODEL-DRIVEN DESIGN, MODULES are part of the model, and should reflect concepts in the domain.

This chapter brings together these building blocks, which embody the model in software. These ideas are conventional, and the modeling and design biases that follow from them have been written about before. But framing them in this context will help a developer create detailed components that will serve the priorities of domain-driven design when tackling the larger model and design issues. Also, developers need a sense of the basic principles in order to stay on course through the inevitable compromises.

Associations

The interaction between modeling and implementation is particularly tricky with the associations between objects.

For every traversable association in the model there is a mechanism with the same properties.

A model that shows an association between a customer and a sales representative corresponds to two things. On one hand, it abstracts a relationship developers deemed relevant between two real people. On the other hand, it corresponds to an object pointer between two Java objects, or an encapsulation of a database lookup, or some comparable implementation.

For example, a one-to-many association might be implemented as a collection in that instance variable. But the design is not necessarily so direct. There may be no collection; an accessor method may query a database to find the appropriate records and instantiate objects based on them. Both of these designs would reflect the same model. The design has to specify a particular traversal mechanism whose behavior is consistent with the association in the model.

In real life, there are lots of many-to-many associations, and many are naturally bi-directional. The same tends to be true of early forms of a model as we brainstorm and explore the domain. But these general associations complicate implementation and maintenance. A generic many-to-many relationship is uncommunicative as well as difficult to implement.

There are at least three ways of making associations more tractable:

- Imposition of a traversal direction
- Addition of a qualifier, effectively reducing multiplicity
- Elimination of non-essential associations

It is important to constrain relationships as much as possible. Bi-directional associations are tricky to implement, and they also mean that both objects can only be understood together. When application requirements do not require traversal in both directions, adding a traversal direction reduces interdependence and simplifies the design. Understanding the domain may reveal a natural directional bias.

The United States has had many presidents, as have many other countries. This is a bi-directional, one-to many relationship. Yet, we seldom would start out with the name “George Washington” and ask the question, “Which country was he president of?” Pragmatically, we can reduce it unidirectional association, traversable from country to president. This actually reflects insight into the domain as well as making a more practical design. It captures the understanding that one direction of the association is much more meaningful and important than the other. It keeps the “Person” class independent of the far less fundamental concept of “President”.

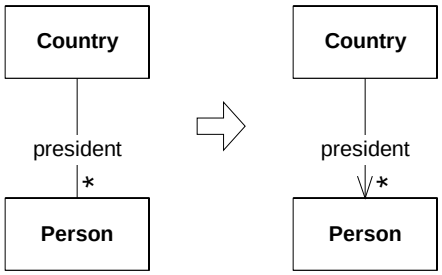


Figure 4. 3

Constraining traversal direction of a many-to-many association effectively reduces it’s implementation to one-to-many -- a *much* easier design.

Very often, deeper understanding leads to a "qualified" relationship. Looking deeper into presidents, we realize that (except in a civil war, perhaps) a country only has one president at a time. This qualifier reduces the multiplicity to one-to-one, and explicitly embeds an important rule into the model. Who was president of the United States in 1790? George Washington.

Constrained associations communicate more knowledge and are more practical designs.

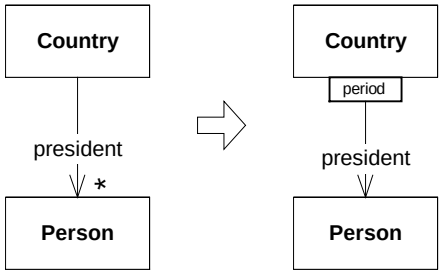


Figure 4. 4

Of course, the ultimate simplification is to eliminate an association altogether, if it is not essential to the job at hand or the fundamental meaning of the model objects.

Consistently constraining associations in ways that reflect the bias of the domain not only makes those associations more communicative and simpler to implement. It also means that when the bi-directionality of a relationship is a semantic characteristic of the domain, and needed for application functionality, its presence also conveys something.

Example: Associations in a Brokerage Account

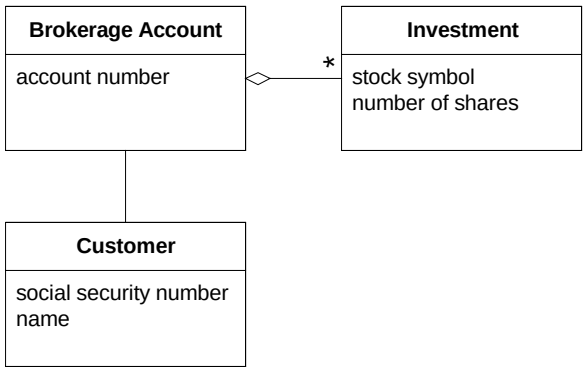


Figure 4. 5

One Java implementation of **Brokerage Account** in this model would be

```
public class BrokerageAccount {
    String accountNumber;
    Customer customer;
    Set investments;
  // Constructors, etc. ommitted.

  public Customer getCustomer() {
    return customer;
  }
  public Set getInvestments() {
```

```
    return investments;
  }
}
```

But if we need to fetch the data from a relational database, another implementation, equally consistent with the model, would be

Table: BROKERAGE_ACCOUNT

ACCOUNT_NUMBER	CUSTOMER_SS_NUMBER

Table: CUSTOMER

SS_NUMBER	NAME

Table: INVESTMENT

ACCOUNT_NUMBER	STOCK_SYMBOL	AMOUNT

```
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  // Omit constructors, etc.

  public Customer getCustomer() {
    String sqlQuery =
      "SELECT * FROM CUSTOMER WHERE SS_NUMBER='"+customerSocialSecurityNumber+"'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }
  public Set getInvestments () {
    String sqlQuery =
      "SELECT * FROM INVESTMENT WHERE BROKERAGE_ACCOUNT='"+accountNumber+"'";
    return QueryService.findInvestmentsFor(sqlQuery);
  }
}
```

(NOTE: The QueryService, a utility for fetching rows from the database and creating objects, is simple for explaining examples, not necessarily a good design for a real project.)

Let's refine the model by qualifying the association between Brokerage Account and Investment, reducing its multiplicity. This says there can only be one investment per stock.

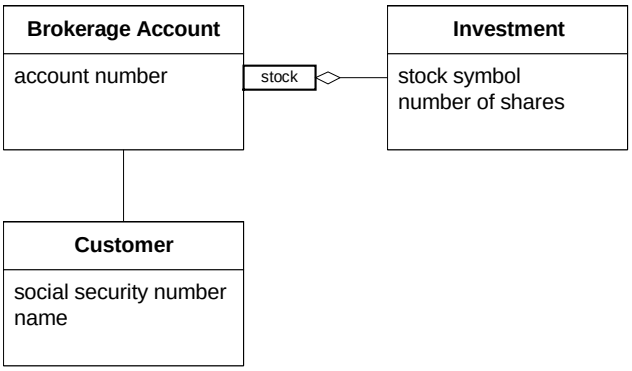


Figure 4. 6

This wouldn't be true of all business situations (e.g. if the lots need to be tracked), but, whatever the particular rules, as constraints on associations are discovered they should be included in the model and implementation. They make the model more precise and the implementation easier to maintain.

The Java implementation could become

```
public class BrokerageAccount {
  String accountNumber;
  Customer customer;
  Map investments;

  // Omitting constructors, etc.

  public Customer getCustomer() {
    return customer;
  }
  public Investment getInvestment(String stockSymbol) {
    return (Investment)investments.get(stockSymbol);
  }
}
```

And a SQL based implementation would be

```
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  //Omitting constructors, etc.

  public Customer getCustomer() {
    String sqlQuery = "SELECT * FROM CUSTOMER WHERE SS_NUMBER='"
      + customerSocialSecurityNumber + "'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }
  public Set getInvestment(String stockSymbol) {
    String sqlQuery = "SELECT * FROM INVESTMENT "
      + "WHERE BROKERAGE_ACCOUNT='" + accountNumber + "'"
      + "AND STOCK_SYMBOL='" + stockSymbol +"'";
    return QueryService.findInvestmentsFor(sqlQuery);
  }
}
```

ENTITIES (AKA REFERENCE OBJECTS)

Many objects are not fundamentally defined by their attributes, but by a thread of continuity and identity.

♦ ♦ ♦

A landlady sued me, claiming major damages to her property. The papers I was served described an apartment with holes in the walls, stains on the carpet and a noxious liquid in the sink that gave off caustic fumes that had caused the kitchen wallpaper to peal. The court documents named me as the tenant responsible for the damages, identifying me by name and by my then-current address. This was confusing to me, since I had never even visited that ruined place.

After a moment, I realized that it must be a case of mistaken identity. I called the plaintiff and told her this, but she didn't believe me. The former tenant had been eluding her for months. How could I prove that I was not the same person who had cost her so much money? I was the only Eric Evans in the phone book.

Well, the phone book turned out to be my salvation. Since I had been living in the same apartment for two years, I asked her if she still had the previous year's book. After she found it and verified that my listing was the same (right next to my namesake's listing), she realized that I was not the person she wanted to sue, apologized, and promised to drop the case.

Computers are not that resourceful. A case of mistaken identity in a software system leads to data corruption and program errors.

There are special technical challenges here, which I'll discuss in a bit, but first let's look at the fundamental issue: Many things are defined by their identity, and not by any attribute. In our typical conception, a person (to continue with the non-technical example) has an identity that stretches from birth to death and even beyond. That person's physical attributes transform and ultimately disappear. The name may change. Financial relationships come and go. There is not a single attribute of a person that cannot change; yet the identity persists. Am I the same person I was at age 5? This kind of metaphysical question is important in the search for effective domain models. Slightly rephrased: Does the user of the application *care* if I am the same person I was at age 5?

In a software system for tracking accounts due, that modest "customer" object may have a more colorful side. It accumulates status by prompt payment, or is turned over to a bill-collection agency for failure to pay. It may lead a double-life in another system altogether when the sales force extracts customer data into

its contact management software. In any case, it is unceremoniously squashed flat to be stored in a database table. When new business stops flowing from that source, the customer object will be retired to an archive, a shadow of its former self.

Each of these forms of the customer is a different implementation based on a different programming language and technology. But when a phone call comes in with an order, it is important to know: Is this the customer who has the delinquent account? Is this the customer that Jack (a particular sales representative) has been working with for weeks? Is this a completely new customer?

A conceptual identity has to be matched between multiple implementations of the objects, its stored forms, and real-world actors like the phone caller. Attributes may not match. A sales representative may have entered an address update into the contact software, which is just being propagated, to accounts-due. Two customer contacts may have the same name. In distributed software, multiple users could entering data from different sources, causing update transactions to propagate through the system to be reconciled in different databases asynchronously.

Object modeling tends to lead us to focus on the attributes of an object, but the fundamental concept of an ENTITY is an abstract continuity threading through a lifecycle and even multiple forms.

Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations. Sometimes such an object must be matched with another object even though attributes differ. An object must be distinguished from other objects even though they might have the same attributes. Mistaken identity can lead to data corruption.

An object defined primarily by its identity is called an "ENTITY"[2]. ENTITIES have special modeling and design considerations. They have lifecycles that can radically change their form and content, while a thread of continuity must be maintained. Their identities must be defined so that they can be effectively tracked. Their class definitions, responsibilities, attributes and associations should revolve around who they are, rather than the particular attributes they carry. Even for ENTITIES that don't transform so radically or have such complicated lifecycles, applying the semantic category leads to more lucid models and more robust implementations.

Of course, most "ENTITIES" in a software system are not people or entities in the usual sense of the word. They are things important to the application user that have continuity through a lifecycle and distinctions independent of attributes.

The issue of identity is confused by the fact that object-oriented languages build "identity" operations into every object (e.g. the '= =' operator in Java). These operations determine if two references point to the same object by comparing their location in memory or by some other mechanism. In this sense, every object instance has identity. In the domain of, say, creating a Java runtime environment or a technical framework for caching remote objects locally, every object instance may indeed be an ENTITY. But this identity mechanism means very little in other application domains.

In a banking application, two deposits of the same amount to the same account on the same day are still distinct transactions, so they have identity and are ENTITIES. On the other hand, the amount attributes of those two transactions are probably instances of some money object. These values have no identity, since there is no usefulness in distinguishing them. In fact, some objects can have the same identity without having the same attributes, or even, necessarily, being of the same class. When the bank customer is reconciling the transactions of the bank statement with the transactions of the check registry, the task is, specifically, to match transactions that have the same identity, even though the dates are recorded differently (the bank clearing date being later than the date on the check). The check number serves as a

[2] A model ENTITY is not the same thing as a Java "Entity Bean". Entity Beans were meant as a framework for implementing ENTITIES, more or less, but it hasn't worked out that way. Most ENTITIES will be implemented as ordinary objects. Regardless of how they are implemented, ENTITIES are a fundamental distinction in a domain model.

unique identifier in this case. Deposits, and cash withdrawals, which don't have an identifying number can be trickier.

In the examples above, apartment tenants and customers, the identity is significant outside a particular software system. This is true in many cases, such as people, cities, cars, or lottery tickets, but sometimes the identity is only important in the context of the system, such as the identity of computer processes.

Therefore,

When an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. Keep the class definition simple and focused on lifecycle continuity and identity. Define a means of distinguishing each object regardless of its form or history. Be alert to requirements to match by attributes. Define an operation that is guaranteed to produce a unique result for each object, possibly by attaching a symbol that is guaranteed unique. This means of identification may come from the outside, or may be an arbitrary identifier created by and for the system, but it must correspond to the identity distinctions in the model. The model must define what it *means* to be the same thing.

Identity is not intrinsic to things in the world, it is a meaning superimposed because it is useful. In fact, the same real-world thing might or might not be represented as an ENTITY in a domain model.

An application for booking seats in a stadium might treat seats and attendees as ENTITIES. In the case of assigned seating, in which each ticket has a seat-number on it, the seat is an ENTITY. Its identifier is the seat number, which is unique within the stadium. The seat may have many other attributes, such as its location, whether the view is obstructed, and the price, but only the seat number, or a unique row and position, is used to identify and distinguish seats.

On the other hand, if the event is "general admission", meaning ticket-holders sit wherever they find an empty seat, there is no need to distinguish individual seats. Only the total number of seats is important. Although the seat numbers are still engraved on the physical seats, there is no need for the software to track them, and, in fact, it would be an error in the model to associate specific seat numbers with tickets, since there is no such constraint at the event. Then seats are *not* ENTITIES, and no identifier is needed.

♦ ♦ ♦

Modeling ENTITIES

ENTITIES are defined by their identities. Attributes are attached and change. Therefore, strip the ENTITY object's definition down to the most intrinsic characteristics, particularly those that identify it, or are commonly used to find or match it. Separate other characteristics into other objects associated with the core ENTITY.

Attributes associated with identity stay with the ENTITY

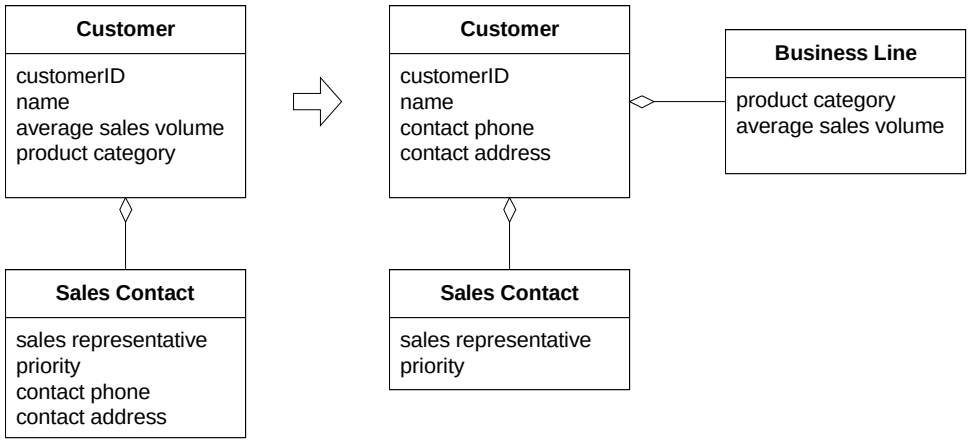


Figure 5. 1

The name does not *define* a person's identity, but it is often used as part of the means of determining it.

Many would dispute moving phone and address into **Customer**. It depends on how Customers are typically matched or distinguished. If a **Customer** has many contact phone numbers for different purposes, then the phone number is not associated with identity and should stay with the **Sales Contact**. The customerID is the one and only identifier of the ENTITY, but the phone number, name and address would often be used to find or match a **Customer**.

Beyond those, look for the attributes that characterize the object. These VALUE OBJECTS are the next pattern in this chapter.

In addition to identity related responsibilities, ENTITIES tend to have major conceptual responsibilities that they fulfill by coordinating operations of objects they own. ENTITY objects will also need operations for these activities.

Designing the Identity Operation

Each ENTITY must have an operational way of establishing its identity with another object; distinguishable even from another object with the same descriptive attributes. An identifying attribute must be guaranteed to be unique within the system however that system is defined -- even if distributed, even when objects are archived.

Object-oriented languages have "identity" operations that determine if two references are to the same object by comparing their location in memory, but this kind of identity is so fragile that, in most technologies for persistent storage of objects, every time an object is retrieved from a database a new instance is created, and so this identity is lost. Every time an object is transmitted across a network, a new instance is created on the destination, and this identity is lost. The problem can be even worse when multiple versions of the same object exist in the system, as when updates propagate through a distributed database. Even with frameworks that simplify these technical problems, the fundamental issue exists: How do you know that two objects represent the same conceptual entity? The definition of identity emerges from the model. Defining it demands understanding of the domain.

Sometimes certain data attributes, or combinations of attributes can be guaranteed to be, or simply constrained to be, unique within the system, and this provides a unique key for the ENTITY. Daily newspapers might be identified by the name of the newspaper, the city and the date of publication. (But watch out for extra editions and name changes!)

When there is no true unique key made of the attributes of an object, the most common solution is to attach to each instance a symbol (such as a number or string) that is unique within the class. Once this ID symbol is obtained and stored as an attribute of the ENTITY, it is designated immutable. It must never change, even if the development system is unable to directly enforce this rule. For example, the ID attribute is preserved as the object gets flattened into a database and reconstructed. Sometimes a technical framework helps with this, but otherwise it just takes engineering discipline.

Sometimes, the uniqueness of the ID must be enforced beyond the computer system's boundaries. For example, if medical records are being exchanged among hospitals that have separate computer systems, ideally the patient ID will be the same in both hospitals and there will never be duplication. IDs assigned by the government, such as social security numbers, are often used, along with many other sources and systems. Such methods are not foolproof. Not everyone has a Social Security number (children and non-US residents, especially), and many people object to its use for privacy reasons. Telephone numbers can change. For these reasons, specially assigned identifiers are often used (e.g. frequent flier numbers). In any case, when an external ID is necessary, the users have the responsibility for supplying IDs and ensuring uniqueness, and they must be provided with adequate tools to handle exceptions that arise.

Often the ID is generated by the system. The generation algorithm must guarantee uniqueness within the system, which can be a challenge with concurrent processing and in distributed systems.

When the ID is automatically generated, the user may be uninterested in it. Often it is needed only internally, as would be the case in a contact management application that lets the user find records by a person's name. The program needs to be able to distinguish two contacts with exactly the same name in a simple, unambiguous way. They will be shown to the user as separate items, but the ID may not be shown. The user will distinguish them on the basis of their company or location, etc.

Finally, there are cases where a generated ID *is* of interest to the users. When I ship a package through a parcel delivery service, I'm given a tracking number, generated by their software, that I can use to identify and follow up on my package. When I book airline tickets or reserve a hotel, I'm given confirmation numbers that are unique IDs for the transaction.

Seeing all these technical problems, it is easy to loose sight of the underlying conceptual problem: What does it mean for two objects to be the same thing? It is easy enough to stamp each object with an ID, or to write an operation that compares two instances, but if these ID's or operations don't correspond to some meaningful distinction in the domain, they just confuse matters more. This is why identity-assigning operations often involve human input. Checkbook reconciliation software may offer likely matches, but the user is expected to make the final determination.

The goal here is to point out when the considerations arise, so that developers are aware they have a problem to solve and know how to narrow their concerns down to the critical areas. Then they can work out a solution, possibly employing techniques that are out of the scope of this book. The key is to recognize that identity concerns emerge from the model. Often, the means of identification does too.

VALUE OBJECTS

Many objects have no conceptual identity. These objects describe some characteristic of a thing.

♦ ♦ ♦

If a child is drawing, he cares about the color of the marker he chooses. He may care about the sharpness of the tip. But if there are two markers of the same color and shape, he won't care which he uses. If a marker is lost and replaced by another of the same color from a new pack, he can resume his work unconcerned about the switch.

Ask the child about the various drawings on the refrigerator and he will quickly distinguish those he made from those his sister made. He and his sister have useful identities, as do their complete drawings. But imagine how complicated it would be if he had to track which lines in a drawing were made by each marker. Drawing would no longer be child's play.

Since the most conspicuous objects in a model are usually ENTITIES, and it is so important to track their identity, it is natural to start thinking about assigning an identity to all domain objects. Sometimes frameworks are created to assign every object a unique ID. Extra analytical effort goes into working out foolproof ways of tracking objects across distributed systems and in database storage.

This can be costly in terms of system performance, as the system has to cope with all that tracking, and many possible performance optimizations are ruled out. Equally important, it muddles the model, forcing all objects into the same mold, and tacking on misleading artificial identities.

Tracking the identity of ENTITIES is essential, but attaching identity to other objects can hurt system performance, add analytical work, and muddle the model by making all objects look the same.

Software design is a constant battle with complexity. We must make distinctions so that any special handling is applied only where necessary.

However, if we think of this category of object as just the absence of identity, we haven't added much to our toolbox or vocabulary. In fact, these objects have characteristics of their own, and their own significance to the model. *These are the objects that describe the nature of things.*

An object that represents a descriptive aspect of the domain that has no conceptual identity is called a VALUE OBJECT. VALUE OBJECTS are instantiated to represent elements of the design that we care about only for *what* they are, not *who* they are.

Colors are an example of VALUES OBJECTS that are provided in the base libraries of many modern development systems, as are strings and numbers. (You don't care which "4" you have or which "Q".) These basic examples are simple, but VALUE OBJECTS are not necessarily simple. For example, a color mixing program might have a rich model in which enhanced color objects could be combined to produce other colors. These colors could have complex algorithms for collaborating to derive the new resulting VALUE OBJECT.

A VALUE OBJECT can be an assemblage of other objects. In software for designing house plans, an object could be created for each window style. This "window style" could be incorporated into a "window" object along with height and width as well as rules governing how these attributes can be changed and combined. These windows are intricate VALUE OBJECTS made up of other VALUE OBJECTS. They in turn would be incorporated into larger elements of a plan, such as "wall" objects.

VALUE OBJECTS can even reference ENTITIES. For example, if I ask an online map service for a scenic driving route from San Francisco to Los Angeles, it might derive a route object linking L.A. and San Francisco via the Pacific Coast Highway. That route object would be a VALUE, even though the three objects it references (two cities and a highway) are all ENTITIES.

VALUE OBJECTS are often passed as parameters in messages between objects. They are frequently transient, created for an operation and then discarded. VALUE OBJECTS are used as attributes of ENTITIES (and other VALUES). A person may be modeled as an ENTITY with an identity, but that person's *name* is a VALUE.

When you care only about the attributes of an element of the model, classify it as a VALUE OBJECT. Making it express the meaning of attributes it conveys and give it related functionality. Treat the VALUE OBJECT as immutable. Don't give it any identity and avoid the design complexities necessary to maintain ENTITIES.

The attributes that make up a value object should form a conceptual whole[3]. For example, street, city, and postal code shouldn't be separate attributes of a Person object. They are part of a single, whole address, which makes a simpler Person, and a more coherent VALUE OBJECT.

An attribute should be conceptually whole

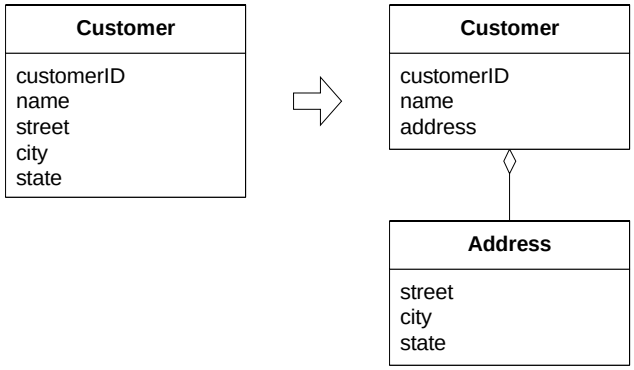


Figure 5. 2

<<BEGIN SIDEBAR>>

[3] The WHOLE VALUE pattern, by Ward Cunningham.

Is "Address" a VALUE OBJECT? Who's asking?

In software for a mail-order company, an address is needed to confirm the credit card, and to address the parcel. But if the customer has a roommate who also orders from the same company, it is not particularly important to realize they are in the same location. Address is a VALUE OBJECT.

In software for the postal service, intended to organize delivery routes, the country could be formed into a hierarchy of regions, cities, postal zones and blocks, terminating in individual addresses. These address objects would derive their zip code from their parent in the hierarchy, and if the postal service decided to reassign postal zones, all the addresses within would go along for the ride. Address is an ENTITY.

In software for the electric utility company, an address corresponds to a destination for their lines and service. Roommates do not order separate electrical service, and if they did, the company would need to realize it. Address is an ENTITY. Or… "Dwelling" could be an ENTITY, with an attribute of address. Then Address is a VALUE OBJECT.

<<END SIDEBAR>>

Designing VALUE OBJECTS

We don't care which instance we have of a VALUE OBJECT, and this lack of constraints gives us design freedom that can be used to simplify the design or optimize performance. This mainly involves making choices about copying, sharing, and immutability.

If two people have the same name, that does not make them the same person, or interchangeable. But the object representing the name is interchangeable since only the spelling of the name matters. A name object could be *copied* from the first person object to the second.

In fact, the two person objects might not need their own name instances. The same name object could be *shared* between the two person objects (each with a pointer to the same name instance) with no change in their behavior or identity. That is, until some change is made to the name of one person. Then the other person's name would change also! To protect from this, in order for an object to be safely shared, it must be *immutable*, meaning it cannot be changed except by full replacement.

The same issues arise when an object passes one of its attributes to another object as an argument or return value. Anything could happen to the wandering object while it is out of control of its owner. The VALUE could be changed in a way that corrupts the owner, by violating the owner's invariants. This is avoided by making the passed object immutable, or by passing a copy.

These extra options for performance tuning can be important because VALUE OBJECTS tend to be numerous. The example of the house design software hints at this. If each electrical outlet is a separate VALUE OBJECT, there might be a hundred of them in a single version of a single house plan. But, if all outlets are considered interchangeable, we could share just one instance of an outlet and point to it a hundred times (an example of FLYWEIGHT, [GHJV95]). In large systems, this kind of effect can be multiplied by thousands, and such an optimization can make the difference between a usable system and one that slows to a crawl, choked on millions of redundant objects.

This is just one example of an optimization trick that is not available for ENTITIES.

The economy of copying versus sharing depends on the implementation environment. While copies may clog up the system with huge numbers of objects, sharing can slow down a distributed system. When a copy is passed between two machines, one message is sent and the copy lives independently on the receiving machine. But if a single instance is being shared, only a reference is passed, requiring a message back to the object for any interaction.

Sharing is best restricted to cases where it is most valuable and least troublesome.

- When saving space or object count in the database is critical.
- When communication overhead is low (e.g. centralized server).
- When the shared object is strictly immutable.

<<Begin Sidebar>>

Special Cases: When to Allow Mutability

Immutability is a great simplifier in an implementation, making sharing and reference passing safe. It is also consistent with the meaning of a value. If the value of an attribute changes, you use a different VALUE OBJECT, rather than modifying the existing one. Even so, there are cases when performance considerations will favor allowing a VALUE OBJECT to be mutable. Forces that would lead to that decision are:

- If the VALUE is frequently changed
- If object creation or deletion is expensive
- If replacement (rather than modification) will disturb clustering (as discussed in the example above)
- If there is not much sharing of VALUEs, or if such sharing is forgone to improve clustering or for some other reason

Just to reiterate: If a VALUE's implementation is to be mutable, then it *must not* be shared. Whether you will be sharing or not, design VALUE OBJECTS as immutable when you can -- that is unless some of the reasons above are strong.

<<END Sidebar>>

Immutability of an attribute or object can be declared in some languages and environments and not in others. These features are helpful for communication of the design decision, but not essential. Many of the distinctions we are making in the model cannot be explicitly declared in the implementation with most current tools and programming languages. You can't declare ENTITIES, for example, and have an identity operation automatically enforced. The lack of direct language support for a conceptual distinction does not mean that it is not useful. It just means that more discipline is needed maintain the rules that are only implicit in the implementation. Communication can be augmented by naming conventions, selective documentation, and *lots of discussion*.

As long as a VALUE OBJECT is immutable, change management is simple, since there isn't any change except full replacement. Immutable objects can be freely shared, as in the electrical outlet example. If garbage collection is reliable, deletion is just a matter of dropping all references to the object. When a VALUE OBJECT is designated immutable in the design, developers are free to make decisions about issues such as copying and sharing on purely technical basis, secure in the knowledge that the application does not rely on particular instances of the objects.

Example: Tuning a database with VALUE OBJECTS

I'll give one example of a purely technical performance tuning decision to demonstrate the usefulness of a model and design that sharply define what is constrained and leaves as much as possible unconstrained. This may be a little hard to follow for those unfamiliar with database technology, but don't worry. This is not a book about specific technologies, and following this example isn't essential. This example is making the point that on real projects decisions have to be made based on specific technologies, and the model can make this easier and safer by avoiding unneeded constraints and by clearly defining the constraints that are in place.

Databases, at the lowest level, have to place data in a physical location on a disk, and it takes time for physical parts to move around and read that data. Sophisticated databases attempt to cluster these physical addresses so that related data can be fetched from the disk in a single physical operation.

If an object is referenced by many other objects, some of those objects will not be located nearby (on the same page), requiring an additional physical operation to get the data. By making a copy, rather than sharing a reference to the same instance, a VALUE OBJECT that is acting as an attribute of many ENTITIES can be stored on the same page as each ENTITY that uses it. This technique of storing multiple copies of the same data is called

"denormalization" and is often used when access time is more critical than storage space or simplicity of maintenance.

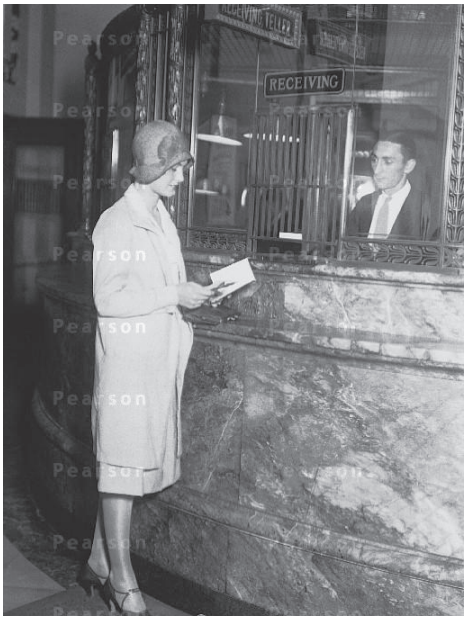
In a relational database, you might want to put a particular VALUE in the table of the ENTITY that owns it, rather than creating an association to a separate table. In a distributed system, holding a reference to a VALUE OBJECT on another server is probably wasteful; instead, a copy of the whole object should be passed to the other server. We can freely make these copies because no identity tracking is needed for VALUE OBJECTS.

Designing associations that involve VALUE OBJECTS

Most of the earlier discussion of associations applies to ENTITIES and VALUE OBJECTS alike. The fewer and simpler the associations in model, the better.

But, while bi-directional associations between ENTITIES may be hard to maintain, bi-directional associations between two VALUE OBJECTS just make no sense. Without identity, it is meaningless to say that an object points back to the same VALUE OBJECT that points to it. The most you could say is that it points to an object that is *equal* to the one pointing to it, but you would have to enforce that invariant somewhere. And although you could do this, and set up pointers going both ways, it is hard to think of examples where it is useful. Try to completely eliminate bi-directional associations between VALUE OBJECTS. If, in the end, something like that seems necessary, rethink the decision to declare the object a VALUE OBJECT in the first place. Maybe it has an identity that hasn't been explicitly recognized yet.

SERVICES



Sometimes, it just isn't a thing.

There are cases where the clearest and most pragmatic design includes operations that do not conceptually belong to an object. Rather than forcing the issue, we can follow the natural contours of the problem space and include SERVICES in the model.

♦ ♦ ♦

There are important domain operations that can't find a natural home in an ENTITY or VALUE OBJECT. Some of these are intrinsically activities or actions, not things, but since our modeling paradigm is objects, we try to fit them into objects anyway.

Now, the more typical mistake is to give up on fitting the behavior into an appropriate object, gradually slipping toward procedural programming. But when we force an operation into an object that doesn't fit the object's definition, the object loses its conceptual clarity and becomes hard to understand or refactor. Complex operations can easily swamp a simple object, obscuring its role. And these operations often draw together many domain objects, coordinating them and putting them into action; the added responsibility will create dependencies on all those objects, tangling concepts that could be understood independently.

Sometimes these services masquerade as model objects, appearing as objects with no meaning beyond doing some operation. These "doers" end up with names ending in "Manager" and the like, and have no state of their own, nor any meaning in the domain beyond the operation they host. Still, at least this gives these distinct behaviors a home without messing up a real model object.

Some concepts from the domain aren't natural to model as objects. Forcing then required domain functionality to be assigned as a responsibility of an ENTITY or VALUE either distorts the definition of a model based object or adds meaningless artificial objects.

A SERVICE is an operation offered as an interface that stands alone in the model, without encapsulating state as ENTITIES and VALUE OBJECTS do. SERVICES are a common pattern in technical frameworks, but they can also apply in the domain layer.

The name "service" emphasizes the relationship with other objects. Unlike ENTITIES and VALUE OBJECTS, it is defined purely in terms of what it can do for a client. A SERVICE tends to be named for an activity, rather than an entity, a "verb" rather than a "noun". A SERVICE can still have an abstract, intentional definition; it just has a different flavor than the definition of an object. A SERVICE should still have a defined responsibility, and that responsibility and the interface fulfilling it should be defined in terms of the domain language. Operation names should come from the UBIQUITOUS LANGUAGE or be introduced into it. Parameters and results should be domain objects.

SERVICES should be used judiciously and not allowed to strip the ENTITIES and VALUE OBJECTS of all their behavior. But, when an operation is actually an important domain concept, a SERVICE forms a natural part of a MODEL-DRIVEN DESIGN. Declared in the model as a SERVICE, rather than as a phony object that doesn't actually represent anything, the standalone operation will not mislead anyone.

A good SERVICE has three characteristics:

- The operation relates to a domain concept that is not a natural part of an ENTITY or VALUE OBJECT.
- The interface is defined in terms of other elements of the domain model.
- The operation is stateless.

Statelessness here means that any client can use any instance of the SERVICE without regard to the instance's individual history. The execution of the SERVICE will use information that is accessible globally, and may even change that global information (have side-effects). But it does not hold state of its own that affects its behavior, as most domain objects do.

When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as a standalone interface declared as a SERVICE. Define the interface in terms of the language of the model and make sure the operation name is part of the UBIQUITOUS LANGUAGE. Make the SERVICE stateless.

♦ ♦ ♦

SERVICES and the Isolated Domain Layer

Although the focus here is on those SERVICES that have an important meaning in the domain in their own right, of course services are not used only in the domain layer. It takes care to distinguish SERVICES that belong to the domain layer from those of other layers, and to factor responsibilities to keep that distinction sharp.

Most SERVICES discussed in the literature are the purely technical and belong in the infrastructure layer. Domain and application SERVICES collaborate with these. For example, a bank might have an application that sends out an email to a customer when an account balance falls below a specific threshold. The interface that encapsulates the email system, and perhaps alternate means of notification, is a SERVICE in the infrastructure layer. Technical SERVICES should lack any business meaning at all. It can be harder to distinguish application SERVICES from domain SERVICES. The application layer is responsible for ordering the notification. The domain layer is responsible for determining if a threshold was met, though this probably does not call for a SERVICE, since it fits the responsibility of the account object.

That banking application could be responsible for funds transfers. If a SERVICE were devised to make appropriate debits and credits for a funds transfer, that capability would belong in the domain layer. Funds transfer has a meaning in the banking domain language, and involves fundamental business logic.

Many SERVICES are built on top of the populations of ENTITIES and VALUES, behaving like scripts that organize the potential of the domain to actually get something done. ENTITIES and VALUE OBJECTS are often too fine-grained to provide a convenient access to the capabilities of the domain layer. Here we encounter a very fine line between the domain layer and the application layer. For example, if the banking application can convert and export our transactions into a spreadsheet file for us to analyze, this is an application

SERVICE. There is no meaning of "file formats" in the domain of banking, and there are no business rules involved

On the other hand, a feature that can transfer funds from one account to another is a domain SERVICE because it embeds significant business rules (crediting and debiting the appropriate accounts, for example) and because a "funds transfer" is a meaningful banking term. In this case, the SERVICE does not do much on its own; it would ask the two account objects to do most of the work. But to put the "transfer" operation on Account is awkward, since it involves two accounts and some global rules. We might like to create a Funds Transfer object to represent the two entries plus the rules and history around the transfer.

But we are still left with calls to SERVICES in the interbank networks. Besides, in most development systems it is awkward to make a direct interface between a domain object and external resources. We dress up those external services with a FAÇADE that would take inputs in terms of the model, perhaps returning a Funds Transfer object as its result. But whatever intermediaries we might have, and even though they don't belong to us, those SERVICES are carrying out domain responsibility of funds transfer.

Partitioning Services into Layers

Application	Funds Transfer App Service: 1.Digests input (e.g. XML request), 2.sends message to domain service for fulfillment, 3.listens for confirmation, 4.decides to send notification using infrastructure service.
Domain	Funds Transfer Domain Service: Interacts with necessary Account and Ledger objects, making appropriate debits and credits, supplies confirmation of result (transfer allowed or not, etc.)
Infrastructure	Send Notification Service: Sends emails, letters, etc. as directed by application.

Granularity

Although the pattern discussion emphasized the expression of a concept as a service, this pattern is also valuable as a means of controlling granularity in the interfaces of the domain layer, as well as decoupling clients from the ENTITIES and VALUE OBJECTS.

Medium-grain, stateless SERVICES can be easier to reuse in large systems because they encapsulate significant functionality behind a simple interface. Also, fine-grained objects can lead to inefficient messaging in a distributed system.

And, as previously discussed, fine-grained domain objects can contribute to knowledge leaks from the domain into the application layer, where the domain object's behavior is coordinated. The complexity of dealing with a highly detailed interaction ends up being handled in the application layer allowing domain knowledge to creep into the application or user interface code and be lost from the domain layer. The judicious introduction of domain services can help keep the bright line between layers.

This pattern favors interface simplicity over client control and versatility. It provides a medium-grain of functionality very useful in packaging components of large or distributing systems. And sometimes it is the most natural way to express a domain concept.

Access to Services

Distributed system architectures, such as J2EE and CORBA, provide special publishing mechanisms for SERVICES, with conventions for their use, and they add distribution and access capabilities. But such frameworks are not always in use on a project, and even when they are, they are likely to be overkill when the motivation is just a logical separation of concerns.

The means of providing access to the SERVICE is not as important as the design decision to carve off specific responsibilities. The "doer" objects may be a satisfactory as implementers of a service's interface. A very simple SINGLETON [GHJV95] style access can be written easily. Coding conventions can make it clear that these objects are just delivery mechanisms for service interfaces, and not meaningful domain objects. The elaborate architectures should be used when there is a real need to distribute the system or otherwise draw on the framework's capabilities.

MODULES (AKA PACKAGES)

MODULES are an old, established design element. There are technical considerations, but cognitive overload is the primary motivation for modularity. MODULES give people two views of the model. They can look at detail within a MODULE without being overwhelmed by the whole, or they can look at relationships between MODULES in views that exclude interior detail.

The MODULES in the domain layer should emerge as a meaningful part of the model, telling the story of the domain on a larger scale.

♦ ♦ ♦

Everyone uses MODULES, but few treat them as a full-fledged part of the model. Code gets broken down by all sorts of categories, from aspects of the technical architecture to developer's work assignments. Even developers who refactor a lot, tend to content themselves with MODULES conceived early in the project.

It is a truism that there should be low coupling between MODULES and high cohesion within them. Explanations of coupling and cohesion tend to make them sound like technical metrics, to be judged mechanically based on the distributions of associations and interactions. Yet it isn't just code being divided into MODULES, but concepts. There is a limit to how many things a person can think about at once (hence low coupling). Incoherent fragments of ideas are even harder to understand than an undifferentiated soup of ideas (hence high cohesion).

Low coupling and high cohesion are general design principles that apply as much to individual objects as to MODULES, but they are particularly important at this larger grain of modeling and design. These terms have been around for a long time. One patterns-style explanation can be found in [Larman 1998].

Whenever two model elements are separated into different modules, the relationships between them become less direct than they were, which increases the overhead of understanding their place in the design. Low coupling between MODULES minimizes this cost, and makes it possible to analyze the contents of one MODULE with a minimum of reference to others that interact.

The elements of a good model have synergy. Well-chosen MODULES bring together elements of the model with particularly rich conceptual relationships. This high cohesion of objects with related responsibilities allows modeling and design work to concentrate within a single MODULE, a scale of complexity a human mind can easily handle.

Modules and the smaller elements should coevolve. Modules tend to be chosen to organize an early form of the objects. After that, the objects tend to change in ways that keep it in the bounds of the existing module definition. Refactoring modules is typically more work and disruption than refactoring classes, and probably should be as frequent. But, just as model objects tend to start out naïve and concrete and gradually transform to reveal deeper insight, modules can become subtle and abstract. Letting the modules reflect changing understanding of the domain will also allow more freedom for the objects within them to evolve.

Like everything else in a domain-driven design, MODULES are a *communications mechanism*. The *meaning* of the objects being divided needs to drive the choice of MODULES. When you place some classes together in a MODULE, you are telling the next developer who looks at your design to think about them together. If your model is telling a story, the MODULES are chapters. The name of the MODULE conveys its meaning. These names enter the UBIQUITOUS LANGUAGE. "Now let's talk about the 'customer' module," you might say to a business expert, and the context is set for your conversation.

Therefore,

When choosing MODULES, focus on conceptual cohesion and telling the story of the system. If this results in tight coupling between MODULES, look to see if an overlooked concept would bring the elements together in a coherent MODULE, or if a change in model concepts would disentangle them. Seek low coupling in the sense of concepts that can be understood and reasoned about independently of each other.

The MODULE should reflect insight into the domain. Refine the model until the concepts partition according to high-level domain concepts and the corresponding code is decoupled as well. Give the MODULES names that become part of the UBIQUITOUS LANGUAGE.

Looking at conceptual relationships is not an alternative to the technical measures. They are different levels of the same issue, and both have to be accomplished. But model-focused thinking produces a deeper solution, rather than an incidental one. And, when there has to be a tradeoff, it is best to go with the conceptual clarity, even if it means more references between MODULES or occasional ripple effects when changes are made to a MODULE. Developers can handle these problems if they understand the story the model is telling them.

♦ ♦ ♦

<<BEGIN SIDEBAR>>

Agile MODULES

The MODULES need to co-evolve with the rest of the model. This means that the MODULES will be refactored along with the model and code. This often doesn't happen. Changing MODULES tends to require widespread updates to the code. It can be disruptive to team communication. It can even throw a monkey wrench into development tools, such as source code control systems. As a result, MODULE structures and names often reflect much earlier forms of the model than the classes do.

This inertia gets compounded. Inevitable early mistakes in MODULE choices lead to high coupling, which makes it hard to refactor. The lack of refactoring just keeps increasing the inertia.

Particular development tools and programming systems exacerbate the problem. For example, in Java, a package must be imported into each class that is dependent on it, mixing two scales, whereas a modeler probably thinks of packages as depending on other packages. This means that changing a package requires attention to class level detail in other parts of the system. To make matters worse, common coding conventions encourage the import of specific classes. Resulting in code like this:

```
ClassA1
import packageB.ClassB1;
import packageB.ClassB2;
import packageB.ClassB3;
import packageC.ClassC1;
import packageC.ClassC2;
import packageC.ClassC3;
…
```

In whatever development technology the implementation will be based on, look for ways of minimizing the work of refactoring MODULES. In Java, there is no escape from importing *into* individual classes, but you can at least import entire packages at a time, reflecting the intention that packages are highly cohesive units while simultaneously reducing the effort of changing package names.

```
ClassA1
import packageB.*;
```

```
import packageC.*;
…
```

This communicates more than the voluminous list of classes above because it conveys the intent to create a dependency on those particular MODULES. If there really is only one class to be used, and the local MODULE doesn't seem to have a conceptual dependency on the other MODULE, then maybe one of the classes doesn't belong in its MODULE.

<<END SIDEBAR>>

The Pitfalls of Infrastructure Driven Packaging

Strong forces on our packaging decisions come from technical frameworks. Some of these are helpful, while others need to be resisted.

One of the best is the division of infrastructure and user interface code into separate groups of packages, leaving the domain layer physically separated into its own set of packages.

On the other hand, "tiered" architectures can fragment the implementation of the model objects. Some frameworks create tiers by spreading the responsibilities of a single domain object into multiple objects and then placing those objects in separate packages. For example, with J2EE a common practice is to place data and data access into an "entity bean" while placing associated business logic into a "session bean". In addition to the implementational complexity of each component, this separation immediately robs an object model of cohesion. One of the most fundamental concepts of objects is to encapsulate data with the logic that operates on that data. It is not fatal, since both components could be viewed as an implementation of a single model element, but, to make matters worse, the entity and session beans are often separated into different packages. At that point, the effort of viewing the various objects and mentally fitting them back together as a single conceptual ENTITY is just too great, and we lose the connection between the model and design. Best practice is to use EJBs at a larger-grain than ENTITY objects, reducing the downside of separating tiers. But fine-grain objects are often split into tiers also.

For example, these problems arose on a rather smart project where each conceptual object was actually broken into four tiers. Each division had a good rationale. The tier was a data persistence layer, handling mapping and access to the relational database. Then came a layer that handled behavior intrinsic to the object in all situations. Next was a layer for superimposing application-specific functionality. The fourth tier was meant as a public interface, decoupled from all the implementation below. This scheme was a bit too complicated, but the layers were well defined and there was some tidiness to the separation of concerns. We could have lived with mentally connecting all the physical objects making up one conceptual object. The separation of aspects even helped, at times. In particular, having the persistence code moved out removed a lot of clutter.

But, on top of all this, the framework required each tier to be in a separate set of packages, named according to a convention that identified the tier. This took up all the mental room for partitioning. As a result, that domain developers tended to avoid making too many MODULES (each of which was multiplied by four) and hardly ever changed one, since the effort of refactoring a MODULE was prohibitive. Worse, the effort of hunting down all the data and behavior that defined a single conceptual class was so difficult (combined with the indirectness of the layering) that developers didn't have much mental space left to think about models. The application was delivered, but with an anemic domain model that basically fulfilled the database access requirements of the application, with behavior supplied by a few SERVICES. The leverage that should have derived from MODEL-DRIVEN DESIGN was limited because the code did not transparently reveal the model and allow a developer to work with it.

This kind of framework design is attempting to address two legitimate issues. One is the logical division of concerns: One object has responsibility for database access, another for business logic, etc. This makes it easier to understand the functioning of each tier (on a technical level) and makes it easier to switch out layers. The trouble is that the cost to application development is not recognized. This is not a book on

framework design, so I won't go into alternative solutions to that problem, but they do exist. And even if there were no options, it would be better to trade this off for a more cohesive domain layer.

The other motivation for these packaging schemes is the distribution of tiers. This could be a strong argument if the code actually gets deployed on different servers. Usually it does not. The flexibility is sought just in case it is needed. On a project that hopes to get leverage from MODEL-DRIVEN DESIGN, this sacrifice is too great.

The costs of elaborate technically driven packaging schemes are:

- If the frameworks partitioning conventions pull apart the elements implementing the conceptual objects, the code no longer reveals the model.
- There is only so much partitioning a mind can stitch back together, and if the framework uses it all up, the domain developers lose their ability to chunk the model into meaningful pieces.

It is best to keep things simple. Choose a minimum of technical partitioning rules that are essential to the technical environment or actually aid development. For example, decoupling complicated data persistence code from the behavioral aspects of the objects may make refactoring easier.

Unless there is a real intention to distribute code on different servers, keep all the code that implements a single conceptual object in the same MODULE, if not the same object.

We could have come to the same conclusion by drawing on the old standard "high cohesion/low coupling". The connections between an "object" implementing the business logic and the one responsible for database access are so extensive that the coupling is very high.

There are other pitfalls where framework design or just conventions of a company or project can undermine MODEL-DRIVEN DESIGN by obscuring the natural cohesion of the domain objects, but the bottom line is the same. The restrictions, or just the large number of required packages, rules out the use of other packaging schemes that are tailored to the needs of the domain model.

Use packaging to separate the domain layer from other code. Otherwise, leave as much freedom as possible to the domain developers to package the domain objects in ways that support their model and design choices.

One exception arises when code is generated based on a declarative design (discussed in Chapter 10). In that case, the developers do not need to read the code, and it is better to put it into a separate package so that it is out of the way, not cluttering up the design elements developers actually have to work with.

♦ ♦ ♦

Modularity becomes more critical as the design gets bigger and more complex. This section presents the basic considerations. Much of *Part III: Strategic Design*, provides approaches to packaging and breaking down big models and designs, and ways to give people focal points to guide understanding.

The ENTITIES, VALUE OBJECTS and their associations, along with a few domain SERVICES and the organizing MODULES are points of direct correspondence between the implementation and the model. The objects, pointers and retrieval mechanisms in the implementation must map to model elements fairly straightforwardly. If it they do not, clean up the code and/or go back and change the model.

It is important to resist the temptation to add anything to these objects that does not closely relate to the concept they represent. These design elements have their job to do: expressing the model. There are other domain related responsibilities that must be carried out and other data that must be managed in order to make the system work, but they don't go in these objects. The next chapter will discuss some supporting objects that fulfill the technical responsibilities of the domain layer, such as defining database searches and encapsulating complex object creation. Employing these four patterns produces an implementation that expresses a model. But the lifecycle of objects demands more of both the model and the design, which is the subject of the next chapter.

Each concept from the domain model should be reflected in an element of implementation. This does not mean forcing everything into an object mold. Chapter 9 will discuss the modeling of unconventional types of concepts using object technology. There are also entire other model paradigms supported by tools, such as rules engines. Projects have to make pragmatic tradeoffs between them. These other tools and techniques are means to the end of a MODEL-DRIVEN DESIGN, not alternatives to it.

Modeling Paradigms

MODEL-DRIVEN DESIGN calls for an implementation technology in tune with the particular modeling paradigm being applied. Many such paradigms have been experimented with, but only a few have been widely used in practice. At present, the dominant paradigm is object-oriented design, and most complex projects these days set out to use objects. This predominance has come about for a variety of reasons, some intrinsic to objects, some circumstantial, and others passed on the advantages that come from wide usage itself.

Why the Object Paradigm Predominates

Many of the reasons teams choose the object paradigm are not technical, or even intrinsic to objects. Intrinsically, object modeling does strike a nice balance of simplicity and sophistication.

If a modeling paradigm is too esoteric, not enough developers will master it, and they will use it badly. If the non-technical members of the team can't grasp at least the rudiments of the paradigm, they will not understand the model and the UBIQUITOUS LANGUAGE will be lost. The fundamentals of object-oriented design seem to come naturally to most people. While some developers miss the subtleties of modeling, even non-technologists can follow a diagram of an object model.

Yet, simple as the concept of object modeling is, it has proven rich enough to capture important domain knowledge. And it was supported from the outset by development tools that allowed a model to be expressed in software.

At this point, the object paradigm also has some significant cicumstancial advantages deriving from maturity and wide-spread adoption. Without mature infrastructure and tool support, a project gets side-tracked into technological R&D, delaying and diverting resources away from application development and introducing technical risks. Some technologies don't play well with others, and it may not be possible to integrate with industry standard solutions, forcing the team to reinvent common utilities. But over the years many of these problems have been solved for objects, or made irrelevant by wide-spread adoption. (Now it falls on other approaches to integrate with main-stream object technology.) Most new technologies provide the means to integrate with the popular object-oriented platforms. This makes integration easier and even leaves open the option of mixing in subsystems based on other modeling paradigms (discussed later in this chapter).

Equally important is the maturity of the *developer community and the design culture itself*. A project that adopts a novel paradigm may be unable to find developers with expertise in the technology, or with the experience to create effective models in the chosen paradigm. It may not be feasible to educate them in a reasonable amount of time because the patterns for making the most of the paradigm and technology haven't gelled yet. Perhaps the pioneers of the field are effective but haven't yet published their insights in an accessible form.

Objects are already understood by a community of thousands of developers, project managers, and all the other specialists involved in a project.

The risks of working in an immature paradigm can be illustrated by a story from an object-oriented project of only a decade ago. In the early-1990s this project committed itself to several cutting edge technologies including use of an object-oriented database on a large scale. It was exciting. People on the team would proudly tell visitors that we were deploying the biggest database this technology had ever been used for. When I joined the project, different teams were spinning out object-oriented designs and storing their objects in the database effortlessly. But gradually the realization crept upon us that we were beginning to absorb a significant fraction of the database's capacity… with test data! The actual database would be dozens of times larger. The actual transaction volume would be dozens of times higher. Was it impossible to use this technology for this application? Had we used it improperly? We were out of our depth.

Fortunately, we were able to bring into the team one of a handful of people in the world with the skills to extricate us from the problem. He named his price and we paid it. There were three sources of the problem.

First, the off-the-shelf infrastructure provided with the database simply didn't scale up to our needs. Second, storage of fine-grain objects turned out to be much more costly than we had realized. Third, parts of the object model had such a tangle of interdependencies that contention became a problem with a relatively small number of concurrent transactions.

With the help of this hired expert, we enhanced the infrastructure. The team, now aware of the impact of fine-grained objects, began to find models that worked better with this technology. And all of us deepened our thinking about the importance of limiting the web of relationships in a model, and began applying this new understanding to making better models with more decoupling between closely interrelated aggregates.

Several months were lost in this recovery, in addition to the earlier months spent going down a failed path. And this had not been the first setback resulting from the learning curve or the immaturity of the chosen technologies and our own lack of experience with it. Sadly, this project eventually retrenched and became quite conservative. To this day they use the exotic technologies, but for cautiously scoped applications that probably don't really benefit from them.

Contrast

A decade later, object-oriented technology is relatively mature. Most common infrastructure needs can be met with off-the-shelf solutions that have been used in the field. Mission critical tools come from major vendors, often multiple vendors, or from stable open-source projects. Many of these infrastructure pieces themselves are themselves widely enough used that there is a base of people who already understand them, as well as books explaining them, and so forth. The limitations of these technologies are fairly well understood, so that knowledgeable teams are less likely to overreach.

Other interesting modeling paradigms just don't have this maturity. Some are too hard to master, and will never be used outside small specialties. Others have potential, but the technical infrastructure is still patchy or shaky, and few people understand the subtleties of creating good models for them. These may come of age, but they are not ready for most projects.

This is why most projects attempting MODEL-DRIVEN DESIGN at present are wise to use object-oriented technology as the core of their system. Nor are they locked into an object-only system. Because objects have become the mainstream of the industry, integration tools are available to connect with almost any other technology in current use.

Yet this doesn't mean everyone should always restrict themselves to objects forever. Traveling with the crowd provides some safety, but it isn't always the way to go.

Object models address a large number of practical software problems. There are, however, domains that are not natural to model as discrete elements. For example, intensely mathematical domains or domains where global logical reasoning dominates are examples that do not fit well into the object-oriented paradigm. Some projects try to have the best of both worlds, with varying results.

Non-objects in an Object World

A domain model does not have to be an object model. MODEL-DRIVEN DESIGN is not specific to object-oriented programming, but it does work best when the implementation technology directly supports the modeling paradigm. For example, a project using Prolog as its implementation language would be driven by a model made up of logical rules and facts. Those cases where it is possible to make the design a mirror of the model provide opportunity to create the most lucid software. Some model paradigm dominates a project. Typically, as discussed above, it is objects, though it could be another.

There are cases where some specific parts of a domain are much easier to express in a paradigm other than the dominant one. When there are just a few isolated elements of a domain that otherwise conforms to the paradigm, developers we can do their best to model them in the same way. They can live with a few awkward objects in an otherwise consistent model. And, of course, if the greater part of the problem domain is more naturally expressed in another paradigm, it may make sense to switch paradigms altogether

and choose a different implementation platform. But when major parts of the domain seem to belong to different paradigms, there is an attraction to modeling each part in the paradigm that fits, using a mixture of tool-sets to support implementation.

Most new projects nowadays use objects as their basic paradigm of modeling and implementation, yet most have to face at least one major part of the system that does not fit easily into the object paradigm. Some of the most common are:

- Relational databases
- Business rules engines
- Workflow engines
- Complex mathematical models

This mixture of paradigms allows developers to model particular concepts with the style that best fits. But it is complicated, and it makes it much harder to keep the design strictly tied to a single model. When developers run up into difficulty seeing the coherent model embodied in the software, MODEL-DRIVEN DESIGN can go out the window, even as this mixture increases the need for it.

Keeping to MODEL-DRIVEN DESIGN when paradigms are being mixed

This is a topic that deserves a book of its own. The goal of this section is merely to show that it isn't necessary to give up MODEL-DRIVEN DESIGN, and that it is worth the effort to keep it. Rules engines will serve as an example of a technology sometimes mixed into an object-oriented application development project.

A knowledge-rich domain model contains explicit rules, yet the object paradigm lacks specific semantics for stating rules and their interactions. Although rules can be successfully modeled as objects, and often are, object encapsulation makes it awkward to apply global rules that cross the whole system. Rules engine technology is appealing because it promises to provide a more natural and declarative way to define rules, effectively allowing the rules paradigm to be mixed into the object paradigm. The logic paradigm is well developed and powerful and seems like a good complement to the strengths and weaknesses of objects.

But people don't always get what they hope for out of rules engines. Some of the products just don't work very well. When the technical problems are solved, there are more fundamental problems stemming from the lack of a seamless view of the program elements that express model concepts running between the two implementation environments. For example, an application can fracture into two – a very static data storage system using objects, and an ad hoc rules processing application that had lost almost all connection with the object model, therefore hard to understand or keep synchronized with the object system.

When the domain model is bridging two paradigms, it is crucial to keep it cohesive, showing the relationships between, in this example, the rules and the objects. When the implementation tools dictate that the domain be implemented in two separate environments, we become that much more dependent on a model made up of clear, fundamental concepts to hold the whole design together.

Applying the same language and the same model to both parts will help hold them together, even though their implementations are distinct. In the model, rules apply to objects or relationships between objects or their attributes. This connection can be maintained by meticulous consistency in the names applied in the two environments and attention to keeping those names in the UBIQUITOUS LANGUAGE.

The team has to find a single model that can work with both implementation paradigms. This is not easy, but it should be possible if the rules engine allows an expressive implementation of rules. This model is *not* going to be entirely object-oriented. Sometimes the fixation on a tool, like UML diagrams, leads people to distort the model to make it fit what can easily be drawn. UML does have some features for representing constraints, but where they are not sufficient, some other style of drawing, or simple English descriptions are better than tortuous adaptation of a drawing style intended for objects. In fact, if the implementation is very expressive, it greatly reduces the need for diagrams to explain the model. The diagram is not the model, and the limits of what can be drawn should not limit what we can achieve with a model.

It is important to continue to think in terms of models while working with rules. Otherwise, the data and the rules become unconnected. The rules in the engine end up more like little programs than conceptual rules in the domain model.

Other paradigm mixtures arise because some major part of the problem domain is not natural to model in the dominant paradigm. When the interdependence is small, a subsystem in the other paradigm can be encapsulated, such as a complex math calculation that simply needs to be called by an object. Other times the different aspects are more intertwined, as when the interaction of the objects is dependent on some mathematical relationships. Then a model must be devised with elements of both paradigms.

While a MODEL-DRIVEN DESIGN does not have to be object-oriented, it *does* depend on having an expressive implementation of the model constructs, be they objects, rules, or workflows. If the available tool does not facilitate that expressiveness, reconsider the choice of tools. An unexpressive implementation obviates the advantage of the extra paradigm

Rules of thumb for mixing non-object elements into a predominantly object-oriented system:

- Don't fight the implementation paradigm. When modeling, find concepts that fit the paradigm that they will be implemented in.
- Lean on the UBIQUITOUS LANGUAGE. Even when there is no rigorous connection between tools, very consistent use of language can keep parts of the design from diverging.
- Be skeptical. Is the tool really pulling its weight? Just because you have some rules, that doesn't necessarily mean you need the overhead of a rules engine. Rules can be expressed as objects if a little less neatly, and multiple paradigms complicate matters enormously.

The relational paradigm is a special case. The most common non-object technology, the relational database is also more intimately related to the object model than other components because it is acting as the persistent store of the data that makes up the objects themselves. This will be discussed in Chapter 6.

6. The Lifecycle of a Domain Object

Every object has a lifecycle. It is born, it may go through various states, it eventually dies and is either archived or deleted. Of course there are simple, transient objects that are created by an easy call to their constructor, used in some computation, and then abandoned to the garbage collector. There is no need to complicate these. But we tend to spend most of our time on more complicated objects that have longer lives, not all of which are spent in active memory. Managing these persistent objects presents challenges that can easily derail an attempt at MODEL-DRIVEN DESIGN.

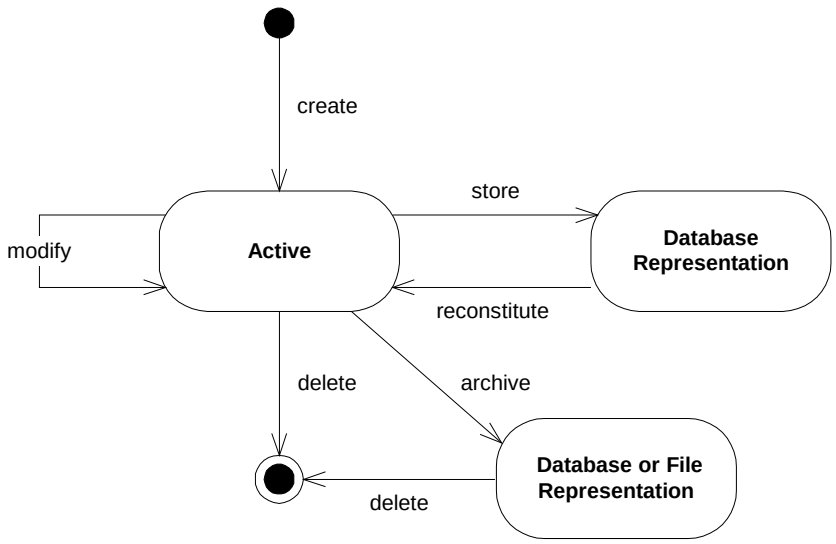


Figure 6. 1 Lifecycle of a Domain Object

The problems fall into two categories:

- Maintaining integrity throughout the lifecycle
- Preventing the model from getting swamped by the complexity of managing the lifecycle.

Three patterns will address these issues. First, AGGREGATES tighten up the model itself by defining clear ownership and boundaries, avoiding a chaotic tangled web of objects. This is crucial to maintaining integrity in all phases of the lifecycle.

Then, we focus on the beginning of the lifecycle, using FACTORIES to create and reconstitute complex objects and AGGREGATES, keeping their internal structure encapsulated. Finally, REPOSITORIES address the middle and end of the lifecycle, providing the means of finding and retrieving persistent objects while encapsulating the immense infrastructure involved.

Although REPOSITORIES and FACTORIES do not themselves come from the domain, they have meaningful roles in the domain design. These constructs complete the MODEL-DRIVEN DESIGN by giving us accessible handles on the model objects.

Modeling AGGREGATES and adding FACTORIES and REPOSITORIES to the design gives us the ability to manipulate the model objects systematically and in meaningful units throughout their lifecycle. AGGREGATES mark off the scope within which invariants have to be maintained at every stage of the lifecycle. FACTORIES and REPOSITORIES, operate on AGGREGATES, encapsulating the complexity of specific lifecycle transitions.

Minimalist design of associations between ENTITIES and VALUE OBJECTS helps simplify traversal and limit the explosion of relationships somewhat, but most business domains are so interconnected that we still end up tracing long, deep paths through object references. In a way, this reflects the realities of the world, which seldom obliges us with sharp boundaries. It is a problem in a software design.

Say you were deleting a "person" object from a database. Along with the person go a name, birth date, and job description. But what about the address? Couldn't there be other people at the same address? If you delete the address anyway, those person objects will have references to a deleted object. Should that be allowed? If you leave it, you accumulate junk addresses in the database. Automatic garbage collection could eliminate the junk addresses, but that technical fix, even if it were available in most database systems, ignores a basic modeling issue.

Even when considering an isolated transaction, the web of relationships in a typical object model gives no clear limit to the potential effect of a change. It is not practical to refresh every object in the system, just in case there is some dependency.

The problem is acute in a system with concurrent access to the same objects by multiple clients. With many users consulting and updating different objects in the system we have to prevent simultaneous changes to interdependent objects. Getting the scope wrong has serious consequences.

It is difficult to guarantee the consistency of changes to objects in a model with complex associations. Invariants need to be maintained that apply to closely related groups of objects, not just discrete objects. Yet cautious locking schemes cause multiple users to interfere pointlessly with each other and make a system unusable.

Put another way, how do we know where an object made up of other objects begins and ends? In any system with persistent storage of data, there must be a way of knowing the scope of a transaction that changes that data, and a way of maintaining consistency of the data (maintaining its invariants). Relational

databases allow various locking schemes, and special tests can be programmed. But these ad-hoc solutions quickly divert attention away from the model, and soon you are back to hacking and hoping.

In fact, finding a balanced solution to these kinds of problems calls for deeper understanding of the domain, this time extending to factors like the density of associations with instances of certain classes, or the frequency of change of those instances. We need find a model that leaves high-contention points looser, and strict invariants tighter.

Although the problem surfaces as technical difficulties involving details of database transactions, it is rooted in the model -- in its lack of defined boundaries. A solution driven from the model will make the model easier to understand and make the design easier to communicate. As the model is revised we will know how to change the implementation.

To address this, schemes have been developed for defining ownership relationships in the model along with a set of rules for implementing transactions that modify the objects and their owners. David Siegel developed the following simple but rigorous system, distilled from those concepts, in the mid 1990s[4].

First we need an abstraction for encapsulating references within the model. An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each AGGREGATE has a root and a boundary. The boundary defines what is inside the AGGREGATE. The root is a single specific ENTITY contained in the AGGREGATE. The root is the only member of the AGGREGATE that outside objects are allowed to hold references to, although objects within the boundary may hold references to each other. ENTITIES other than the root have local identity, but it only needs to be unique within the aggregate, since no outside object can ever see it out of the context of the root ENTITY.

A model of a car might be used in software for an auto repair shop. The car is an ENTITY with global identity – we want to distinguish that car from all other cars in the world, even very similar ones. We can use the Vehicle Identification Number for this, a unique identifier assigned to each new car. We might want to track the rotation history of the tires through the four wheel positions. We might want to know mileage and tread wear of each tire. To know which tire is which, the tires must be identified ENTITIES also. But it is very unlikely that we care about the identity of those tires outside of the context of that particular car. If we replace the tires and send the old ones to a recycling plant, either our software will no longer track them at all, or they will become anonymous members of a heap of tires. No one will care about their rotation histories. More to the point, even while they are attached to the car, no one will try to query the system to find a particular tire and then see which car it is on. They will query the database to find a car and then ask it for a transient reference to the tires. Therefore, the car is the root ENTITY of the AGGREGATE whose boundary encloses the tires also. On the other hand, engine blocks have serial numbers engraved on them and are sometimes tracked independently of the car. In some applications, the engine might be the root of its own AGGREGATE.

[4] David Siegel used this system on projects and related it to me, but has not published it.

Local vs. Global Identity and Object References

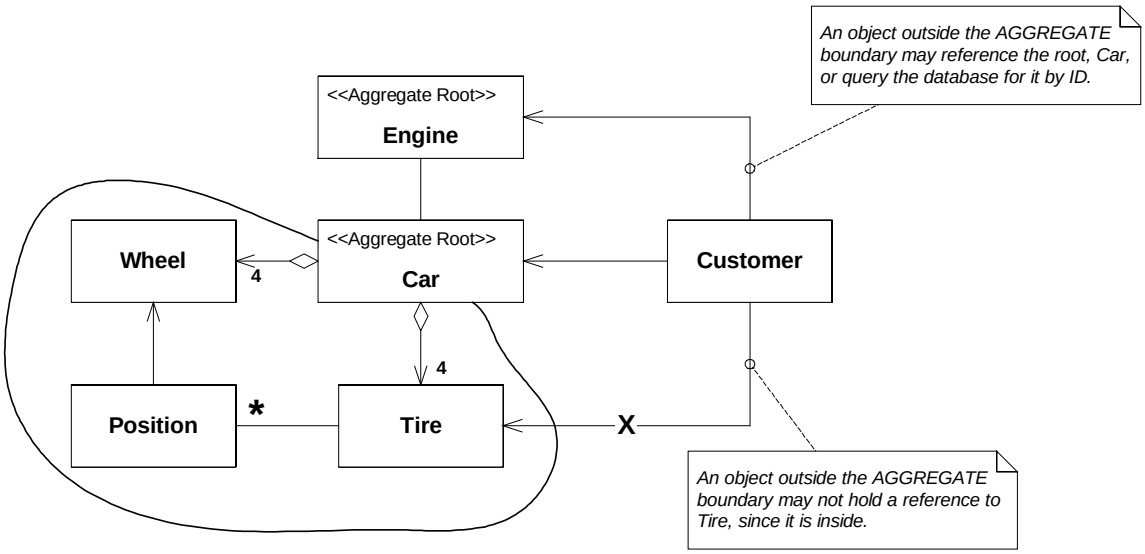


Figure 6. 2

Invariants, consistency rules that must be maintained whenever data changes, will involve relationships between members of the AGGREGATE. Any rule that spans AGGREGATES will not be expected to be always up to date. Through event processing, batch processing, or other update mechanisms, other dependencies can be resolved within some specified time. But the invariants applied within an AGGREGATE will be enforced with the completion of each transaction.

AGGREGATE Invariants

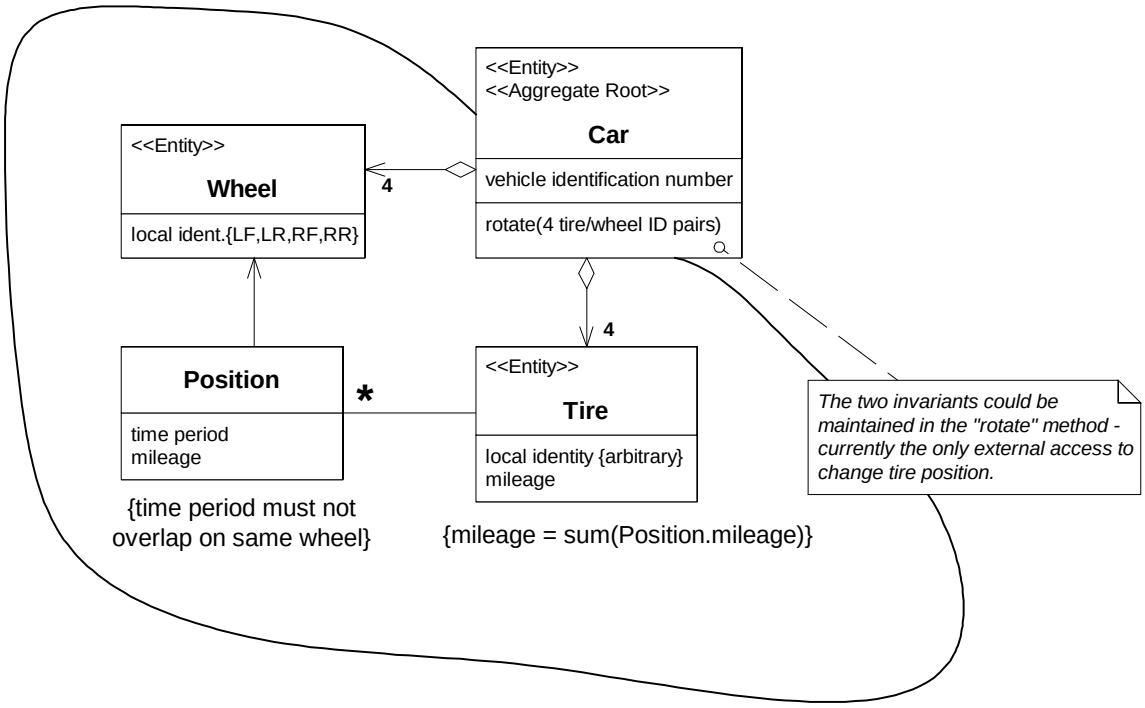


Figure 6. 3

Now, to translate that conceptual AGGREGATE into the implementation, we need a set of rules to apply to all transactions:

- The root ENTITY has global identity, and is ultimately responsible for checking invariants.
- Root ENTITIES have global identity. ENTITIES inside the boundary have local identity, unique only within the AGGREGATE.
- Nothing outside the AGGREGATE boundary can hold a reference to anything inside, except to the root ENTITY. The root ENTITY can hand references to the internal ENTITIES to other objects, but those objects can only use them transiently, and may not hold onto the reference. The root may hand a copy of a value to another object, and it doesn't matter what happens to it, since it's just a value and no longer will have any association with the AGGREGATE.
- As a corollary to the above rule, only AGGREGATE roots can be obtained directly with database queries. All other objects must be found by traversal of associations.
- Objects within the AGGREGATE can hold references to other AGGREGATE roots.
- A delete operation must remove everything within the AGGREGATE boundary at once. (With garbage collection, this is easy. Since there are no outside references to anything but the root, delete the root and everything else will be collected.)
- When a change to any object within the AGGREGATE boundary is committed, all invariants of the whole AGGREGATE must be satisfied.

Cluster the ENTITIES and VALUE OBJECTS into "AGGREGATES" and define boundaries around each. Choose one ENTITY to be the "root" of each AGGREGATE, and control all access to the objects inside the boundary through the root. Only allow references to the root to be held by external objects. Transient references to internal members can be passed out for use within a single operation only. Because the root controls access it cannot be blind-sided by changes to the internals. This makes it practical to enforce all invariants for objects in the AGGREGATE and for the AGGREGATE as a whole in any state-change.

It can be very helpful to have a technical framework that allows you to declare AGGREGATES and then automatically carries out the locking scheme and so forth. Without that, it calls for self-discipline within the team, to pay attention to the AGGREGATES and code consistent with them.

Example: Purchase Order Integrity

Consider the complications possible in a simplified purchase order system.

Model a Purchase Order System

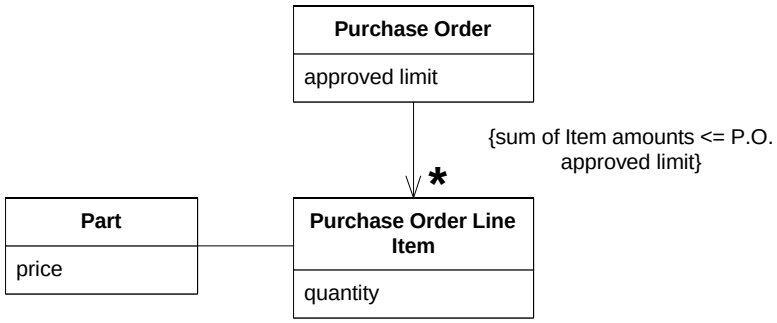


Figure 6. 4

This presents a pretty conventional view of a purchase order (P.O.) broken down into line-items, with an invariant rule that the sum of the line-items can't exceed the limit for the P.O. as a whole. The existing implementation has three interrelated problems, which will be explained in detail:

1. Invariant enforcement. When a new line-item is added, the P.O. checks the total and marks itself invalid if an item pushes it over the limit. As we'll see, this is not adequate protection.
2. When deleting or archiving the P.O., the line-items are taken along, but the model gives no guidance on where to stop following the relationships. There is also confusion about the impact of changing the Part price at different times.
3. Multiple users are creating contention problems in the database.

Multiple users will be entering and updating various P.O.s concurrently, and we have to prevent them from messing up each other's work. Let's start with a very simple strategy in which we lock any object a user begins to edit until they commit their transaction. So, when George is editing line-item #001, Amanda cannot get access to it. She can edit any other line-item on any P.O.

Initial Condition of P.O Stored in Database

P.O. #0012946		Approved Limit: $1,000.00		
Line Item	Quantity	Part	Price	Amount
001	3	Guitars	@ 100	300.00

002	2	Trombones	@ 200	400.00
			Total:	700.00

Objects will be read from the database and instantiated in each users memory space so that they can be viewed and edited, but database locks will only be requested when an edit begins, so they can both work as long as they stay away from each other's items. All is well…That is, until both George and Amanda start working on separate line items in the same P.O.

George edits his view

P.O. #0012946		Approved Limit: $1000.00		
Line Item	Quantity	Part	Price	Amount
001	**5**	Guitars	@ 100	**500.00**
002	2	Trombones	@ 200	400.00
			Total:	900.00

Amanda edits her view

P.O. #0012946	
Line Item	Quantity
001	3
002	**3**

Everything looks fine to both users and to their software because they ignore changes to the database that happen during the transaction, and the locked line-item is not involved in the other user's change.

Resulting P.O. Violates Approval Limit (broken invariant)

P.O. #0012946		Approved Limit: **$1,000.00**		
Line Item	Quantity	Part	Price	Amount
001	5	Guitars	@ 100	500.00
002	3	Trombones	@ 200	600.00
			Total:	**1,100.00**

When both users have saved their changes, a P.O. will be stored in the database that violates the invariant of the domain model. An important business rule has been broken. And nobody even knows.

Clearly, locking a single line-item isn't an adequate safeguard. If, instead, we locked an entire P.O. at a time, the problem would have been prevented.

George edits his view

Amanda is locked out o

P.O. #0012946	Approved Limit: $1000.00

Line Item	Quantity	Part	Price	Amount
001	**6**	Guitars	@ 100	**600.00**
002	2	Trombones	@ 200	400.00
			Total:	1,000.00

Amanda gets access; G

P.O. #0012946	
Line Item	Quantity
001	6
002	**3**

The program will not allow this to be saved until Amanda has resolved the problem, perhaps with a higher limit or by eliminating a guitar. This prevents the problem, and it may be a fine solution if work is mostly spread widely across many P.O.s. But if multiple people typically work simultaneously on different parts of a large P.O., then this locking will get cumbersome.

Even assuming many small P.O.s, there are other ways to violate the assertion. Consider that "Part". If someone changed the price of a trombone while Amanda was adding to her order, wouldn't that violate the invariant too?

Let's try locking the part in addition to the entire P.O.

George editing P.O.

Guitars and Trombones locked

P.O. #0012946		Approved Limit: $1,000.00		
Line Item	Quantity	Part	Price	Amount
001	**2**	Guitars	@ 100	**200.00**
002	2	Trombones	@ 200	400.00
			Total:	600.00

Amanda adds trombones

Violins locked

P.O. #0012932	
Line Item	Quantity
001	3
002	**2**

Sam adds trombones; must wait on George

P.O. #0013003		Approved Limit: $15,000.00		
Line Item	Quantity	Part	Price	Amount
001	1	Piano	@1,000	1,000.00
002	**2**	**Trombones**	**@ 200**	**400.00**
			Total:	1,400.00

The inconvenience is mounting, because there is a lot of contention for the instruments (the "parts"). And then…

George adds violins; must wait on Amanda (!)

P.O. #0012946		Approved Limit: $1,000.00		
Line Item	Quantity	Part	Price	Amount
001	**2**	Guitars	@ 100	**200.00**
002	2	Trombones	@ 200	400.00
003	**1**	**Violins**	**@ 400**	**400.00**
			Total:	1,000.00

Returning to the purchase order problem, we can begin to incorporate our knowledge of the business into the model to recognize that

- Parts are used in many P.O.s (high contention)
- There are fewer changes to parts than there are to P.O.s
- Changes to part prices do not necessarily propagate to existing P.O.s. It depends on the time of a price change relative to the status of the P.O.

Point 3 is particularly obvious when we consider archived P.O.s that have already been delivered.

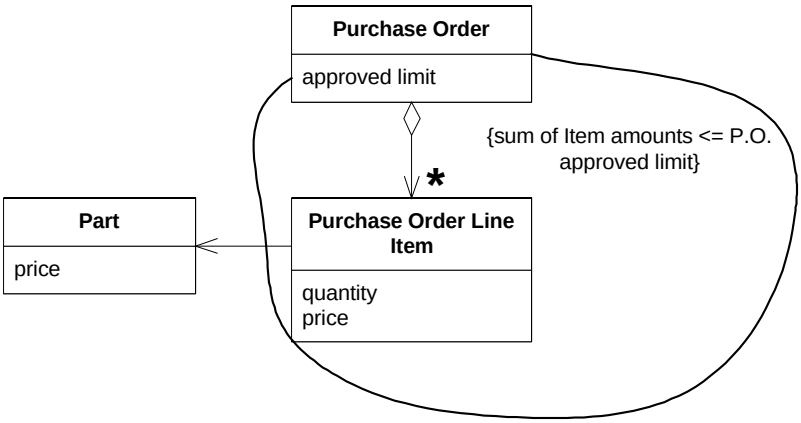


Figure 6. 5

An implementation consistent with this model would guarantee the invariant relating P.O. and its items, but changes to the price of a part would not have to immediately affect the items that reference it. The item now holds its own price for this reason. For example, the system could present a queue of items with outdated prices to the users each day, so they could update them or exempt them. By making the dependency of line-items on parts looser we avoid contention and reflect the realities of the business better, while tightening the relationship of the P.O. and its line-items guarantees that an important business rule will be followed.

It imposes an ownership of the P.O. and its items that is consistent with business practice. The creation and deletion of a P.O. and items are naturally tied together, while the creation and deletion of parts is independent.

♦ ♦ ♦

AGGREGATES mark off the scope within which invariants have to be maintained at every stage of the lifecycle. The following patterns, FACTORIES and REPOSITORIES, operate on AGGREGATES, encapsulating the complexity of specific lifecycle transitions.

FACTORIES

When creation of an object, or an entire AGGREGATE becomes complicated or reveals too much of the internal structure, FACTORIES provide encapsulation.

♦ ♦ ♦

Much of the power of objects rests in the intricate configuration of their internals and their associations. An object that expresses some model concept should be distilled to the point that nothing remains that does not relate to its meaning or support its role in those interactions. This basic mid-lifecycle responsibility is plenty. Problems arise from giving a complex object responsibility for its own creation.

Consider as an analogy a car engine (a real one, not a computer model). It is an intricate piece of machinery with dozens of parts collaborating to perform the engine's responsibility: to turn a shaft. One could imagine trying to design an engine block that could grab onto a set of pistons and insert them into its cylinders, spark plugs that would find their sockets and screw themselves in, and so on. But it seems unlikely that such a complicated machine would be as reliable or as efficient as our typical engines are. Instead, we accept that something else will assemble the pieces. Perhaps it will be a human mechanic or perhaps it will be an industrial robot. Either the robot or the human is actually more complex than the engine it assembles. The job of assembling parts is completely unrelated to the job of spinning a shaft. The assemblers are only used during creation of the car -- you don't need a robot or mechanic with you when you are driving. Since cars are never assembled and driven at the same time, there is no value in combining both of these functions into the same mechanism. Likewise, assembling a complex compound object is a job that is best separated from whatever job that object will have to do when it is finished.

But shifting responsibility to the other interested party, the client, may lead to even worse problems. The client knows what job needs to be done and relies on the domain objects to carrying out the necessary computations. It should not reflect any understanding of how the task is done or the internal nature of the object doing it. If the client is expected to assemble the domain objects it needs, it must know something about the internal structure of the object. In order to enforce all the invariants that apply to the relationship of parts in the domain object it must know some of the object's rules. Even calling constructors couples the

client to the concrete classes of the objects it is building. No change to the implementation of the domain objects can be made without changing the client, making refactoring harder.

A client taking on object creation becomes unnecessarily complicated and blurs its responsibility. It breaches the encapsulation of the domain objects and AGGREGATES being created. Even worse, if the client is part of the application layer, then responsibilities have leaked out of the domain layer altogether. This tight coupling of the application to the specifics of the implementation, strips away most of the benefits of abstraction in the domain layer, and makes continuing changes ever more expensive.

Creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the created objects. Combining these responsibilities can produce ungainly designs that are hard to understand. Shifting the responsibility of directing construction to the client muddies the design of the client, breaches encapsulation of the assembled object or AGGREGATE and overly couples the client to the implementation of the created object.

Complex object creation is a responsibility of the domain layer that does not belong to the objects that express the model. It has no meaning in the domain, but is a necessity of the implementation. To solve this problem, we have to add constructs to the domain design that are not ENTITIES, VALUE OBJECTS, or SERVICES. There are some cases where the factory corresponds to a milestone significant in the domain, such as "open a bank account", but it is usually motivated by design considerations. This is a departure from the previous chapter and it is important to make the point clear: We are adding elements to the design that do not correspond to anything in the model, but that nonetheless are part of the domain layer's responsibility.

Every object-oriented language provides a mechanism for creating objects (constructors in Java and C++, instance creation class methods in Smalltalk), but there is a need for more abstract construction mechanisms that are decoupled from the other objects. A program element whose responsibility is the creation of other objects is called a FACTORY.

Basic Interactions with a FACTORY

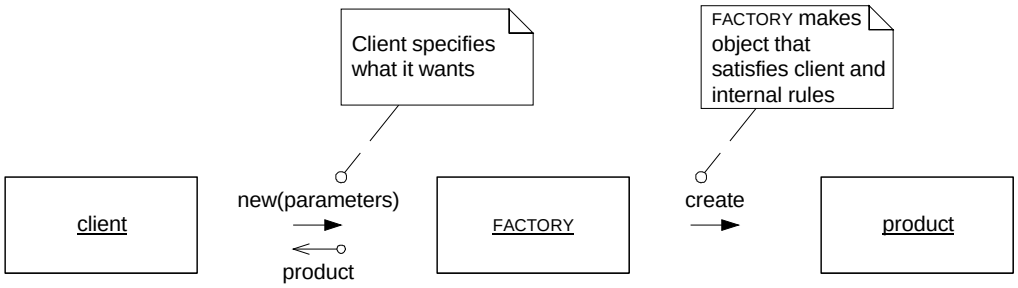


Figure 6. 6

Just as the interface of an object should encapsulate its implementation, allowing a client to use its behavior without knowing how it works, a FACTORY encapsulates the knowledge needed to create a complex object or AGGREGATE. It provides an interface that reflects the goals of the client and an abstract view of the created object.

Therefore,

Shift the responsibility for creating instances of complex objects and AGGREGATES to a separate object, which may itself have no responsibility in the domain model, but is still part of the domain design. Provide an interface that encapsulates all complex assembly and that does not require the client to reference the concrete classes of the objects being instantiated. Create entire AGGREGATES as a piece, enforcing their invariants.

♦ ♦ ♦

There are many ways to design FACTORIES. Several special purpose creation patterns were thoroughly treated in [GHJV95]: FACTORY METHOD, ABSTRACT FACTORY, and BUILDER. That book mostly explored patterns for the most difficult object construction problems. The point here is not to delve deeply into designing FACTORIES, but to get across that that FACTORIES are important components of a domain design, even though they are not expressing the domain model, and their proper use can help keep a MODEL-DRIVEN DESIGN on track.

The two basic requirements for any good FACTORY are:

- Each creation method is atomic and enforces all invariants of the created object or AGGREGATE. It should only be able to produce an object in a consistent state. For an ENTITY this means the creation of the entire AGGREGATE, with all invariants satisfied, but probably with optional elements still to be added. For an immutable VALUE this means all attributes are initialized to their correct final state. If the interface makes it possible to request an object that can't be created correctly, then an exception should be raised or some other mechanism should be invoked that will ensure that no improper return value is possible.
- The FACTORY should be abstracted to the type desired, rather than the concrete class(es) involved. The sophisticated FACTORY patterns in [GHJV95] help with this.

Choosing FACTORIES and Their Sites

Generally speaking, you create a factory to build something whose details you want to hide, and you place the FACTORY where you want the control to be. These decisions usually revolve around AGGREGATES.

For example, if you needed to add elements inside a preexisting AGGREGATE, you might create a FACTORY METHOD on the root of the AGGREGATE. This hides the implementation of the interior of the AGGREGATE from any external client, while giving the root responsibility for ensuring the integrity of the AGGREGATE as elements are added.

FACTORY METHOD Encapsulates Expansion of an AGGREGATE

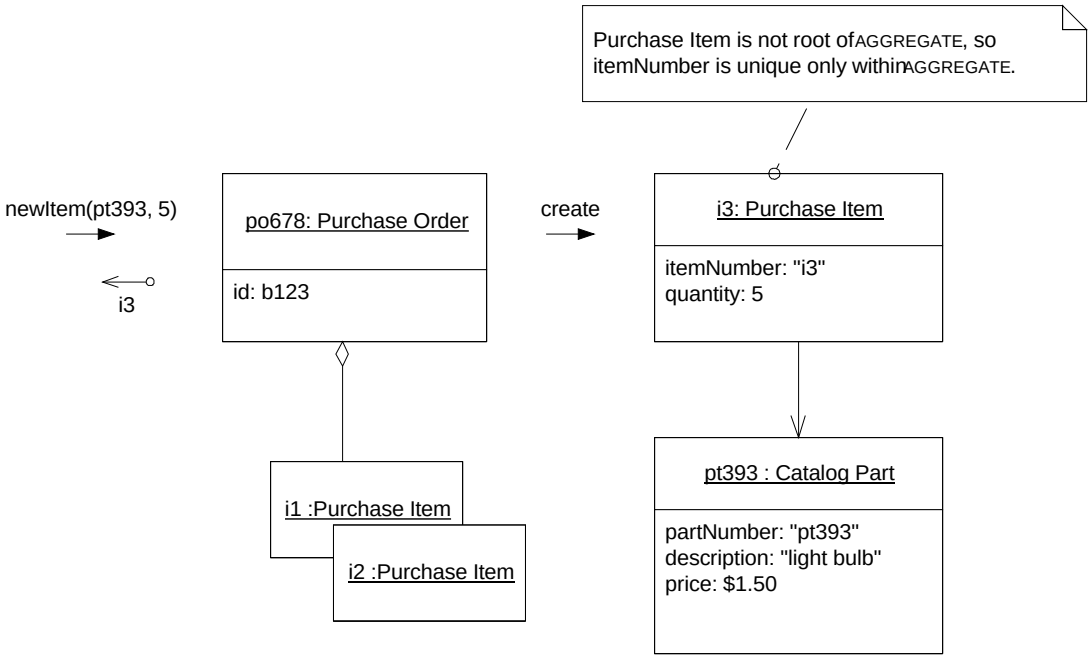


Figure 6. 7

Another example would be to place a FACTORY METHOD on an object that is closely involved in spawning another object, although it doesn't own the product once it is created. When the data and possibly the rules of one object are very dominant in the creation of an object, this saves pulling information out of the spawner to be used elsewhere to create the object. It also communicates the special relationship between the spawner and the product.

FACTORY METHOD Spawns ENTITY That is Not Part of the Same AGGREGATE

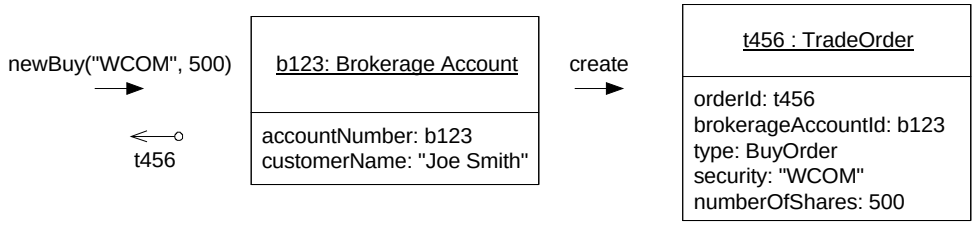


Figure 6. 8

The **Trade Order** is not part of the same AGGREGATE as the **Brokerage Account** because, for a start, it will go on to interact with the trade execution application, where the **Brokerage Account** would only be in the way. Even so, it seems natural to give it control over the creation of **Trade Orders**. The **Brokerage Account** contains information that will be embedded in the **Trade Order** (starting with its own identity) and also rules that govern what trades are allowed. We might also benefit from hiding the implementation of **Trade Order**. For example, it might be refactored into a hierarchy with separate subclasses for **Buy Order** and **Sell Order**. The FACTORY keeps the client from being coupled to the concrete classes.

A FACTORY is very tightly coupled to its product, so a FACTORY should only be attached to an object that has a close natural relationship with the product. When there is something we want to hide, either the concrete implementation, or the sheer complexity of construction, yet there doesn't seem to be a natural host, we must create a dedicated FACTORY object or SERVICE. These factories are usually provided for an AGGREGATE roots, and they enforce the AGGREGATE'S invariant on the product. If an object interior to an AGGREGATE needs a FACTORY, and the AGGREGATE root is not a good home for it, then go ahead and make a standalone FACTORY, but be mindful of the limits on what you are allowed to do with an object owned by an AGGREGATE. (In Smalltalk, the class, which is an object in its own right, is used as the FACTORY. The details of a class hierarchy are encapsulated by using the super-class of the hierarchy as an ABSTRACT FACTORY.)

Standalone FACTORY Builds AGGREGATE

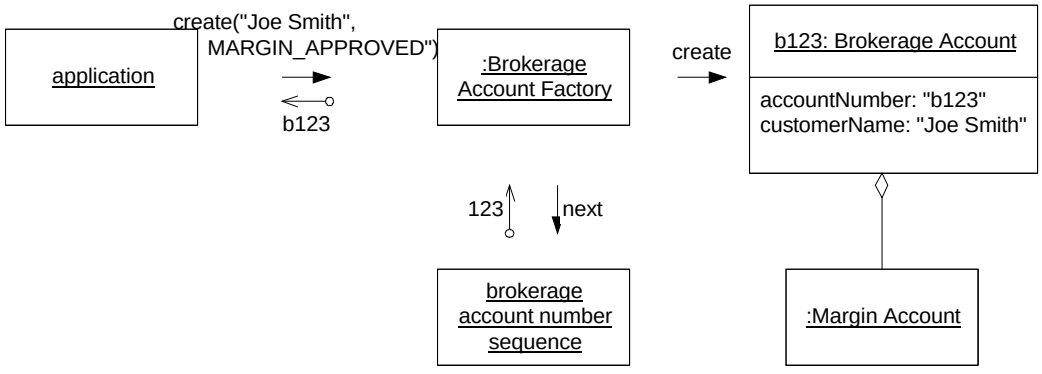


Figure 6. 9

When a constructor is all you need

I've seen far too much code where *all* instances are created by directly calling class constructors, or whatever the primitive level of instance creation is for the programming language. The introduction of FACTORIES has great advantages, and is generally underused. Yet there are cases where the directness of a constructor is the best choice. FACTORIES can actually obscure simple objects that don't use polymorphism.

The tradeoffs favor a bare, public constructor in the following circumstances:

- The class is the type. It is not part of any interesting hierarchy, and isn't used polymorphically by implementing an interface.
- The client cares about the implementation, perhaps as a way of choosing a STRATEGY.
- All of the attributes of the object are available to the client so that no object creation gets nested inside the constructor exposed to the client.
- The construction is not complicated.

A public constructor must follow the same rules as a factory: It must be an atomic operation that satisfies all invariants of the created object.

Be careful about calling constructors within constructors of other classes, unless they are dead simple. Complex assemblies, especially of AGGREGATES, call for FACTORIES.

The Java class library offers interesting examples. All collections implement interfaces that decouple the client from the concrete implementation. Yet they are all created by direct calls to constructors. A FACTORY could have encapsulated the collection hierarchy. The FACTORY's methods could have allowed a client to ask for the features it needed, with the FACTORY selecting the appropriate class to instantiate. Code that created collections would be more expressive, and new collection classes could be installed without breaking every Java program.

But there is a case in favor of the concrete constructors. First, the choice of implementation can be a performance-sensitive for many applications, so an application might want control. (Even so, a really smart FACTORY could accommodate such factors.) Anyway, there aren't very many collection classes, so it isn't that complicated to choose.

The abstract collection types preserve some value in spite of the lack of a FACTORY because of their usage patterns. Collections are very often created in one place and used in another. This means that the client that ultimately uses the collection, adding, removing, and retrieving its contents, can still talk to the interface and be decoupled from the implementation. The selection of a collection class typically falls to the object that owns the collection, or to the owning object's FACTORY.

Designing the Interface

When the specific method signatures of a FACTORIES, whether standalone or FACTORY METHOD, keep in mind these two points:

Each operation must be atomic. You have to pass in everything needed to create a complete product in a single interaction with the FACTORY. You also have to decide what will happen if creation fails, in the event that some invariant isn't satisfied. You could throw an exception or just return a null. To be consistent, consider adopting a coding standard for failures in factories.

The FACTORY will be coupled to its arguments. If you are not careful in your selection of input parameters, you can create a rat's nest of dependencies. The degree of coupling will depend on what you do with the argument. If it is simply plugged into the product, you've created a modest dependency. If you are picking parts out of the argument to use in the construction, the coupling gets tighter.

The safest parameters are those from a lower design layer. Even within a layer, there tend to be natural strata with more basic objects that are used by higher-level objects. (Such layering will be discussed in different ways in Chapter 10, "Supple Design", and again in Chapter 16, "Large-scale Structure")

Another good choice of parameter is an object that is closely related to the product in the model, so that no new dependency is being added. In the earlier example of a **Purchase Order Item**, the FACTORY METHOD takes a **Catalog Part** as an argument, which is an essential association for the **Item**. This adds a direct dependency between the **Purchase Order** class and the **Part**. But these three objects form a close conceptual group. The **Purchase Order's** AGGREGATE already referenced the **Part**, anyway. So giving control to the AGGREGATE root and encapsulating the AGGREGATE'S internal structure is a good tradeoff.

Use the abstract type of the arguments, not their concrete classes. The FACTORY is coupled to the concrete class of the products, it does not need to be coupled to concrete parameters.

Where Does Invariant Logic Go?

A FACTORY is responsible for ensuring all invariants are met for the object or AGGREGATE it creates; yet you should always think twice before removing the rules applying to an object outside that object. The FACTORY can delegate invariant checking to the product, and this is often best.

But FACTORIES have a special relationship with their products. They already know their product's internal structure, and their entire reason for being involves the implementation of their product. Under some circumstances, there are advantages to placing invariant logic in the FACTORY and reducing clutter in the product. This is especially appealing with AGGREGATE rules (which span many objects). It is especially *unappealing* with FACTORY METHODS attached to other domain objects.

Although in principle invariants apply at the end of every operation, often the transformations allowed to the object can never bring them into play. There might be a rule that applies to the assignment of the identity attributes of an ENTITY. But after creation that identity is immutable. VALUE OBJECTS are completely immutable. An object doesn't need to carry around logic that will never be applied in its active lifetime. In these cases, the FACTORY is a logical place to put invariants, keeping the product simpler.

ENTITY FACTORIES Versus VALUE OBJECT FACTORIES

ENTITY FACTORIES differ from VALUE OBJECT FACTORIES in two ways. VALUE OBJECTS are immutable and so the product comes out complete, so the FACTORY operations have to allow for a full description of the product. ENTITY FACTORIES tend to take just the essential attributes required to make a valid AGGREGATE. Details can be added later if they are not required by an invariant.

Then there are the issues involved in assigning identity to an ENTITY – irrelevant to a VALUE OBJECT. As pointed out in Chapter 5, an identifier can either be assigned automatically by the program or supplied from the outside, typically by the user. If a customer's identity is to be tracked by the telephone number, then that telephone number must obviously be passed in as an argument to the FACTORY. When the program is assigning an identifier, the FACTORY is a good place to control it. Although the actual generation of a unique tracking id is typically done by a database "sequence" or other infrastructure mechanism, the FACTORY knows what to ask for and where to put it.

Reconstitution of Stored Objects

Up to this point, the FACTORY has played its part in the very beginning of an object's lifecycle. At some point, most objects get stored in databases or transmitted through a network, and few current database technologies retain the object character of their contents. Most transmission methods flatten an object into an even more limited presentation. Therefore, upon retrieval, there is a potentially complex process of reassembling the parts into a live object.

A factory used for reconstitution is very similar to one used for creation, with two major differences:

- An ENTITY FACTORY used for reconstitution does not assign a new tracking id. To do so would lose the continuity with the object's previous incarnation. So identifying attributes must be part of the input parameters in a factory reconstituting a stored object.
- A FACTORY reconstituting an object will handle violation of an invariant differently. During creation of a new object, a FACTORY should simply balk at when an invariant isn't met, but a more

flexible response may be necessary in reconstitution. If an object already exists somewhere in the system (the database, etc.) this cannot be ignored. Yet we also can't ignore the rule violation. There has to be some strategy for repairing such inconsistencies, which can make reconstitution more challenging that the creation of new objects.

Reconstituting an ENTITY Retrieved from a Relational Database

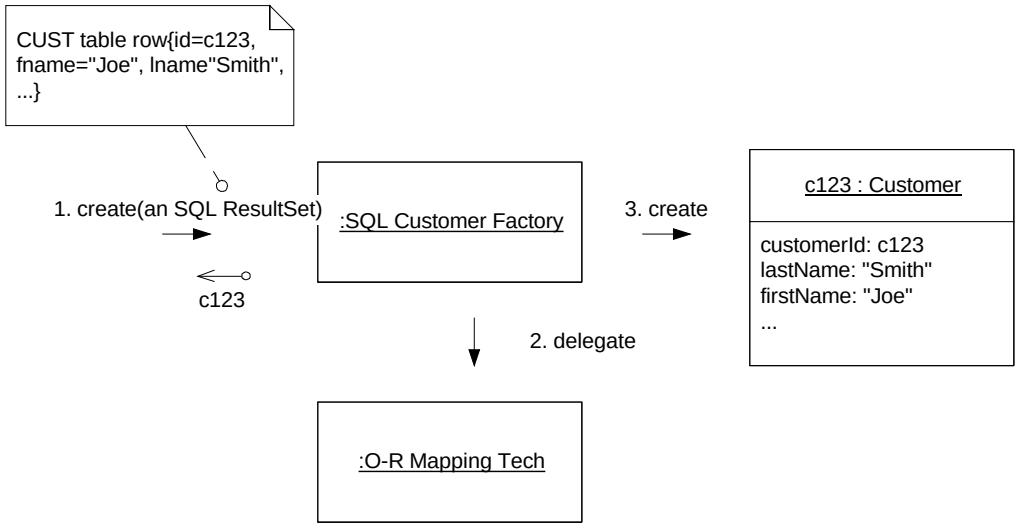


Figure 6. 10

Reconstituting an ENTITY transmitted as XML

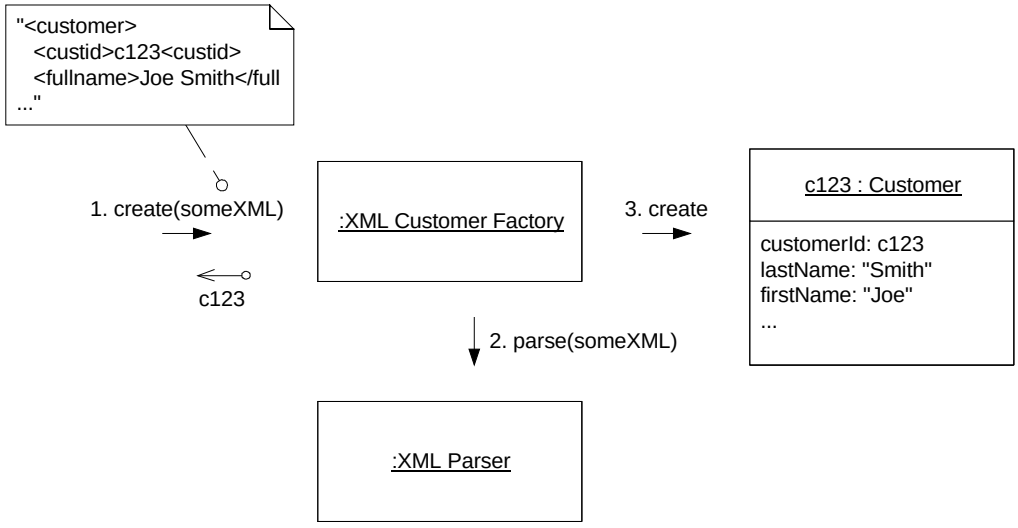


Figure 6. 11

Object mapping technologies may provide some or all of these services in the case of database reconstitution, which is convenient. Whenever there is exposed complexity in reconstituting an object from another medium, the FACTORY is a good option.

♦ ♦ ♦

To sum up, the access points for creation of instances must be identified and their scope defined explicitly. They may simply be constructors, but often there is a need for a more abstract or elaborate instance creation mechanism. This introduces new constructs into the design - FACTORIES. FACTORIES do not express any part of the model, yet they are a part of the domain design that helps keep the model expressing objects sharp.

A FACTORY encapsulates the lifecycle transitions of creation and reconstitution. Another transition that exposes technical complexity that can swamp the domain design is the transition too and from storage. This is the responsibility of another domain design construct, the REPOSITORY …

REPOSITORIES

…Associations in the OBJECT MODEL allow us to find an object based on its relationship to another. But we must have a means of finding the starting point for such a traversal for an ENTITY or VALUE in the middle of its lifecycle.

♦ ♦ ♦

To do anything with an object, you have to hold a reference to it. How do you get that reference? There are three ways. The first way is to create it, since the creation operation will return a reference to the new object. The second way is to traverse an association. You start with an object you already know, and ask it for an associated object. Any object-oriented program is going to do a lot of this, and these links give object models much of their expressive power. But you have to get that first object.

I actually encountered a project once where, in an enthusiastic embrace of MODEL-DRIVEN DESIGN, they were attempting to do *all* object access by creation or traversal! Their objects resided in an object database, and they reasoned that conceptual relationships would exist that would provide all necessary associations, if only they analyzed them enough, making their entire domain model cohesive. This forced them to create just the kind of endless tangle that the last few chapters have been intended to avert through careful implementation of ENTITIES and the use of AGGREGATES. They didn't stick with this strategy long, but they never replaced it with another coherent approach. They cobbled together ad hoc solutions and became less ambitious.

Few would even think of this approach, much less be tempted by it, because they store most of their objects in relational databases. This makes it natural to use the third way of getting a reference: execute a query to find the object in a database based on its attributes, or find the constituents of an object and then reconstitute it. Such queries are globally accessible, so we can get a reference to any stored object.

A database search makes it possible to go directly to any object. There is no need for all objects to be interconnected, which allows us to keep the web of objects manageable. Whether to provide a traversal or depend on a search becomes a design decision, trading off the decoupling of the search against the cohesiveness of the association. Should the "customer" object hold a collection of all the "orders" placed, or should the orders be found in the database, searching on the "customer id" field? The right combination of search and association makes the design comprehensible.

Unfortunately, developers don't usually get to think much about such design subtleties because they are swimming in the sea of mechanisms needed to pull off the trick of storing an object and bringing it back, and eventually removing it from storage.

Now, from a technical point of view, retrieval of a stored object is really a subset of creation, since the data from the database is used to assemble new objects. Indeed, the steps a client usually has to take make it hard to forget this reality. But conceptually, this is the *middle* of the lifecycle of an ENTITY. A "customer" object does not represent a new customer just because we stored it in a database and retrieved it. To keep this distinction in mind, I refer to the creation of an instance from stored data as "reconstitution".

The goal of domain-driven design is to create software by focusing on a model of the domain rather than the technology. By the time a developer has constructed an SQL query, passed it to a query service in the infrastructure layer, obtained a result set of table rows, pulled the necessary information out and passed it to a constructor or FACTORY, the model focus is gone. It becomes natural to think of the objects as containers for the data that the queries provide and the whole design shifts toward a data processing style. The details of the technology vary, but the problem remains that the client is dealing with technology, rather than model concepts. Infrastructure like METADATA MAPPING LAYERS help a great deal by making the conversion of the query result into objects easier, but the developer is still thinking about technical mechanisms, not the domain. Worse, as clients use the database directly, developers are tempted to bypass model features like AGGREGATES, directly taking and manipulating what they need. More and more domain rules are embedded in query code or simply lost. Object databases do eliminate the conversion problem, but search mechanisms are usually still mechanistic, and developers are still tempted to grab whatever objects they want.

A client needs a practical means of acquiring references to preexisting domain objects. If the infrastructure makes it easy to do so, the developers of the client may add more traversable associations, muddling the model. On the other hand, they may use queries to pull the exact data they need from the database, or a few specific objects rather than navigating the modeled associations. Domain logic moves into queries and client code, and the ENTITIES and VALUE OBJECTS become mere data containers. The sheer technical complexity of applying most database access infrastructure quickly swamps the client code, leading to the dumbing-down of the domain layer and the irrelevance of the model.

Drawing on the design principles discussed to this point, we can reduce the scope of the object access problem somewhat, assuming that we find a method of access that keeps the model focus sharp enough to employ those principles. Using the pattern language unfolded so far, we can then restate the dilemma more precisely. For starters, we need not concern ourselves with transient objects. Transients (typically VALUE OBJECTS) live brief lives, used in the client operation that created them and then discarded. Then there are the persistent objects that don't need access because they are more convenient to find by traversal. For example, the address of a person could be requested from the "person" object. And, most importantly, *any object internal to an aggregate is prohibited from access except by traversal from the root*.

Generally speaking, VALUE OBJECTS do not need global search access. Persistent VALUE OBJECTS are usually found by traversal from some ENTITY that acts as the root of the AGGREGATE that encapsulates them. In fact, a global access to a value is often meaningless, as finding a VALUE by its properties would be equivalent to creating a new instance with those properties. There are exceptions, though. For example, when I am planning travel online I sometimes save a few prospective itineraries and return later to select one to book. Those itineraries are VALUES (if there were two made up of the same flights, I would not care which was which), but they have been associated with my user name and retrieved for me intact. Another case would be an "enumeration", when a type has a strictly limited predetermined set of possible values. Global access to VALUE OBJECTS is much less common than for ENTITIES, though, and if you find you need to search the database for a preexisting VALUE, it is worth considering the possibility that you've really got an ENTITY whose identity you haven't recognized.

From this discussion, it is clear that most objects should not be accessed by a global search. It would be nice for the design to communicate those that do.

A subset of persistent objects must be globally accessible through a search based on object attributes. Such access is needed for the roots of AGGREGATES that are not convenient to reach by traversal. They are usually ENTITIES, sometimes VALUE OBJECTS with complex internal structure, and sometimes enumerated VALUES. Providing access to other objects muddies important distinctions. Free database queries can actually breach the encapsulation of domain objects and AGGREGATES. Exposure of technical infrastructure and database access mechanisms complicates the client and obscures the MODEL-DRIVEN DESIGN.

There is a raft of techniques for dealing with the technical challenges of database access. Examples include encapsulating SQL into QUERY OBJECTS or translating between objects and tables with METADATA MAPPING layers [Fowler2002]. FACTORIES can help reconstitute stored objects (as discussed later in this chapter). These and many other techniques help keep a lid on complexity.

But even so, take note of what has been lost. We are no longer thinking about concepts in our domain model. Our code will not be communicating about the business; it will be manipulating the technology of data retrieval. The REPOSITORY pattern provides us a simple conceptual framework to encapsulate those solutions and bring back our model focus.

A REPOSITORY represents all objects of a certain type as a conceptual set (usually simulated). It acts like a collection, except with more elaborate querying capability. Objects of the appropriate type are added and removed, and the machinery behind the REPOSITORY inserts them or deletes them from the database. This definition gathers a cohesive set of responsibilities for providing access to the roots of AGGREGATES from early-lifecycle through the end.

Clients request objects from the REPOSITORY using query methods that select objects based on criteria specified by the client, typically specifying the value of certain attributes. The REPOSITORY retrieves the requested object, encapsulating the machinery of database queries and metadata mapping. Repositories can implement a variety of queries that select objects based on whatever criteria the client requires. They can also return summary information, such as a count of how many instances meet some criteria. They can even return summary calculations, such as the total across all matching objects of some numerical attribute.

A REPOSITORY Doing a Search for a Client

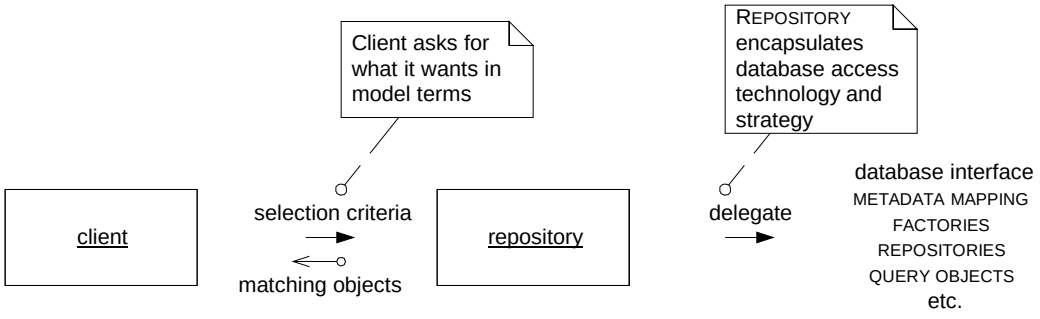


Figure 6. 12

This lifts a huge burden from the client, which can now talk to a simple, intention-revealing interface, and ask for what it needs in terms of the model. To support all this requires a lot of complex technical infrastructure, but the interface is simple and conceptually connected to the domain model.

Therefore,

For each type of object that needs global access, create an object that can provide the illusion of an in-memory collection of all objects of that type. Set up access through a well-known global interface. Provide methods to add and remove objects, which will encapsulate the actual insertion or removal of data in the data store. Provide methods that select objects based on some criteria, and return fully instantiated objects or collections of objects whose attribute values meet the criteria, thereby encapsulating the actual storage and query technology. Only provide repositories for AGGREGATE

roots that actually need direct access. Keep the client focused on the model, delegating all object storage and access to the REPOSITORIES.

REPOSITORIES have many advantages, including

- Simple concept for obtaining persistent objects and managing their lifecycle
- Decouples application and domain design from persistence technology, multiple database strategies, or even multiple data sources
- Communicates design decisions about object access
- Easy substitution of dummy implementation, for use in testing, using in-memory collection

Querying a REPOSITORY

All repositories provide methods that allow a client to request objects matching some criteria, but there is a range of options of how to design this interface.

The easiest REPOSITORY to build has hard-coded queries with specific parameters. These queries could range from a very simple request for an ENTITY by its identity (provided by almost all REPOSITORIES), or for a collection of objects with a particular attribute value, to complex combinations of parameters, value ranges (such as date ranges), or even some calculations that can be supported by the underlying database and that fall within the general responsibility of a REPOSITORY.

Hard-Coded Queries in a Simple REPOSITORY

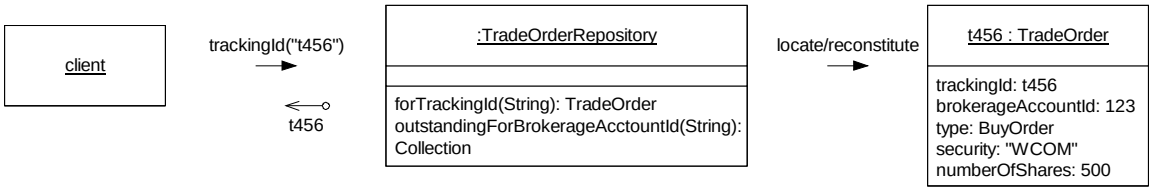


Figure 6. 13

Although most queries return an object or a collection of objects, it also fits within the concept to return some types of summary calculations, such as an object count, or a sum of a numerical attribute that was intended by the model to be tallied.

Hard-coded queries can be built on top of any infrastructure and without a lot of investment, since they just do what some client would have had to do anyway.

On projects with a lot of querying, more supportive infrastructure, and/or a staff familiar with the necessary technology, a REPOSITORY framework can be built that allows more flexible queries. An approach to this that is particularly harmonious with domain-driven design is the specification-based query. A SPECIFICATION is a way of allowing a client to describe (specify) what it wants without concern for how it will be obtained in the process creating an object that can actually carry out the selection. This pattern will be discussed in depth in Chapter 11.

Flexible, Declarative SPECIFICATION of Search Criteria in a Sophisticated REPOSITORY

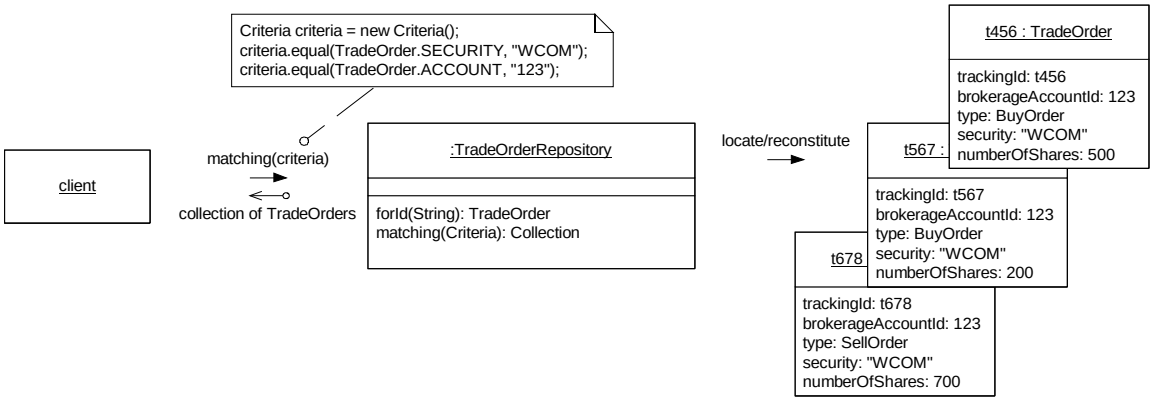


Figure 6. 14

The specification-based query is elegant and flexible. Depending on the infrastructure available, it may be a modest framework or it may be prohibitively difficult. Rob Mee and Edward Hieatt discuss more of the technical issues involved in designing such REPOSITORIES in [Fowler2002].

Even a REPOSITORY design with flexible queries should allow for the addition of specialized hard-coded queries. These might be convenience methods that encapsulate an often-used query, or a non-object query such as mathematical summaries of selected objects. Frameworks that don't allow for such contingencies tend to distort the domain design or get bypassed by developers.

Client Code Ignores REPOSITORY Implementation; Developer Does Not

Encapsulation of the persistence technology allows the client to be very simple, completely decoupled from the implementation of the REPOSITORY. But, as is often the case with encapsulation, the developer must understand what is happening under the hood. The performance implications can be extreme when REPOSITORIES are used in different ways or work in different ways.

Kyle Brown tells the story of getting called in on a manufacturing application based on WebSphere that was being rolled out to production. The system was mysteriously running out of memory after a few hours of use. Kyle browsed through the code and found the reason. At one point, they were summarizing some information about every item in the plant. They had done this using a query called "all objects" which instantiated each of the objects and then selected the bits they needed. This had the effect of bringing the entire database into memory at once! The problem hadn't shown up in testing because of the small amount of test data.

This is an obvious no-no, but much more subtle oversights can present serious problems. Developers need to understand the implications of using encapsulated behavior. That does not have to mean detailed familiarity with the implementation. Well designed components can be characterized. (This is one of the main points of Chapter 10, "Supple Design".)

Chapter 5 pointed out, in the discussion of the design of software objects that express the model, that the underlying technology might constrain your modeling choices. A relational database can place a practical limit on deep compositional object structures, for example. In just the same way, there must be feedback in both directions between the use of the REPOSITORY and the implementation of its queries.

Implementing a REPOSITORY

Implementation will vary greatly depending on the technology being used for persistence and the infrastructure you have. The ideal is to hide all this from the client (although not the developer of the client) so that client code will be the same whether the data is stored in an object database, a relational database, or simply held in memory. The REPOSITORY will delegate to the appropriate infrastructure services to get the

job done. Encapsulating the mechanisms of storage, retrieval and query is the most basic feature of a REPOSITORY implementation.

The Repository Encapsulates the Underlying Data Store

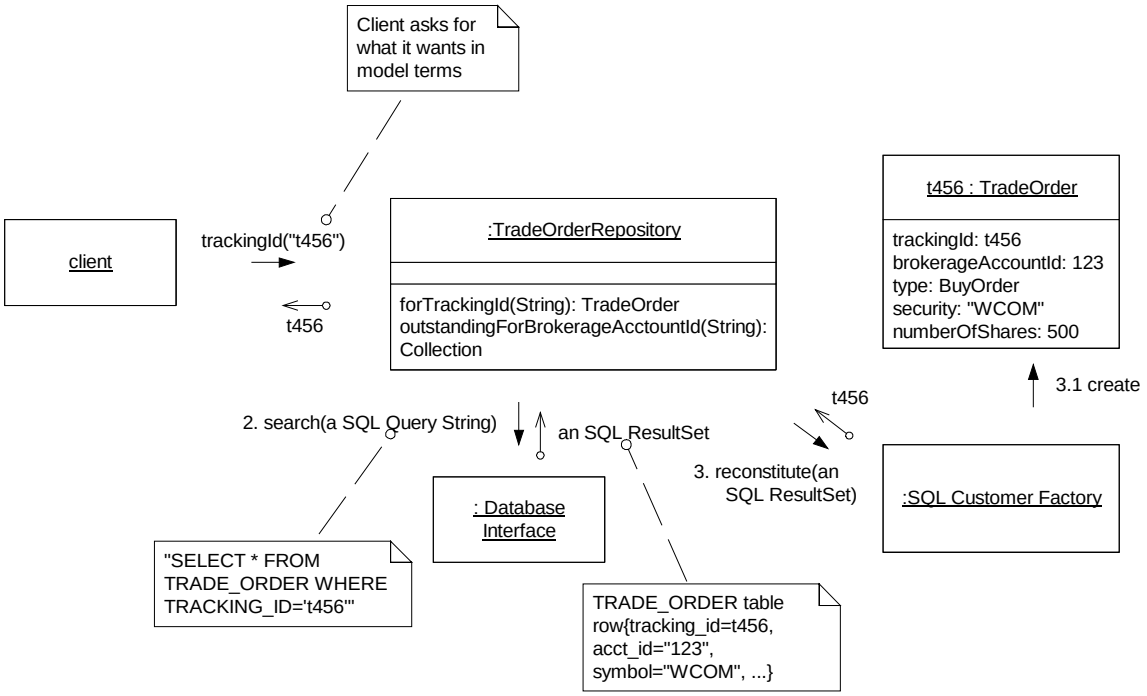


Figure 6. 15

The possibilities are so diverse, that I can only list of concerns to keep in mind. The concept is the essential thing, and should be adaptable to many situations.

- **Abstract the type.** A REPOSITORY "contains" all instances of a specific type, but this does not have to mean one REPOSITORY for each class. The type could be an abstract superclass of a hierarchy (e.g. a TradeOrder could be a BuyOrder or a SellOrder). It could be an interface whose implementers are not even hierarchically related. Or it could be a specific concrete class. Keep in mind you may well face constraints imposed by the lack of such polymorphism in your database technology.
- **Take advantage of the decoupling from the client.** You have more freedom to change the implementation of a REPOSITORY than you would if the client were calling the mechanisms directly. You can take advantage of this to optimize for performance, by varying the query technique or by caching objects in memory, freely switching persistence strategies at any time. You can facilitate testing of the client code and the domain objects by providing an easily manipulated dummy in-memory strategy.
- **Leave transaction control to the client.** While the REPOSITORY will insert and delete from the database, it will ordinarily not commit anything. It is tempting to commit after saving, for example, but the client presumably has the context to correctly initiate and commit units of work. This will be simpler if the REPOSITORY keeps hands-off.

Typically, teams add a framework to the infrastructure layer to support the implementation of repositories. In addition to the collaboration with the lower level infrastructure components, the repository superclass might implement some basic queries, especially when a flexible query is being implemented. Unfortunately, with a type system like Java has, this would force you to type returned objects as "Object", leaving the client to cast them to the repository's contained type. Of course, this will have to be done with queries that return collections anyway.

Some additional guidance to implementing REPOSITORIES and some of their supporting technical patterns like QUERY OBJECT can be found in [Fowler2002].

Working Within Your Frameworks

Before implementing something like a REPOSITORY, you need to think carefully about the infrastructure you are committed to, especially any architectural frameworks. You may find that the framework provides services you can use to easily create a repository, or you may find that it fights you all the way. You may find that the architectural framework has already defined a pattern of getting persistent objects that is equivalent. Or you may find that it has defined a pattern that is not like a REPOSITORY at all.

For example, your project might be committed to J2EE. Looking for conceptual affinities between the framework and the patterns of MODEL-DRIVEN DESIGN (and keeping in mind that an entity bean is *not* the same thing as an ENTITY), you may have chosen to use entity beans to correspond to AGGREGATE roots. The construct within the architectural framework of J2EE that is responsible for providing access to these objects is the "EJB Home". Trying to dress up the EJB Home to look like a REPOSITORY could lead to other problems.

In general, don't fight your frameworks. Seek ways to keep the fundamentals of domain-driven design and let go of the specifics when the framework is antagonistic. Look for affinities between the concepts of domain-driven design and the concepts in the framework. This is assuming that you have no choice but to use the framework. Many J2EE projects don't use entity beans at all. If you have the freedom, choose frameworks, or parts of frameworks, that are harmonious with the style of design you want to use.

Relationship with FACTORIES

A FACTORY is involved in the beginning of the life of an object, while a REPOSITORY helps manage the middle and end. When objects are being held in memory, or stored in an object database, this is straightforward. But in the typical situation, at least some object storage is in relational databases, files, or other, non-object-oriented systems. In these cases, the retrieved data must be reconstituted into object form.

Because the REPOSITORY is, in this case, creating objects based on data, many people consider the REPOSITORY to *be* a FACTORY, and indeed it is, from a technical point of view. But it is more useful in domain design to keep a view of the model, and, as pointed out before, the reconstitution of a stored object is not the creation of a new conceptual object. In this domain-driven view of the design, FACTORIES and REPOSITORIES have distinct responsibilities. The FACTORY makes new objects, the REPOSITORY finds old objects. The client of a REPOSITORY should be given the illusion that the objects are in memory. The object may have to be reconstituted (yes, a new instance created) but it is the same conceptual object, still in the middle of its lifecycle.

These two views can be reconciled by making the REPOSITORY *delegate* object creation to a FACTORY, which (in theory, though seldom in practice) could also be used to create objects from scratch.

This clear separation helps also by unloading all responsibility for persistence from the FACTORIES. A FACTORY'S job is to instantiate a potentially complex object from data. If this is a new object, the client will know this and can add it to the REPOSITORY, which will encapsulate the storage of the object in the database.

REPOSITORY uses FACTORY to reconstitute preexisting object

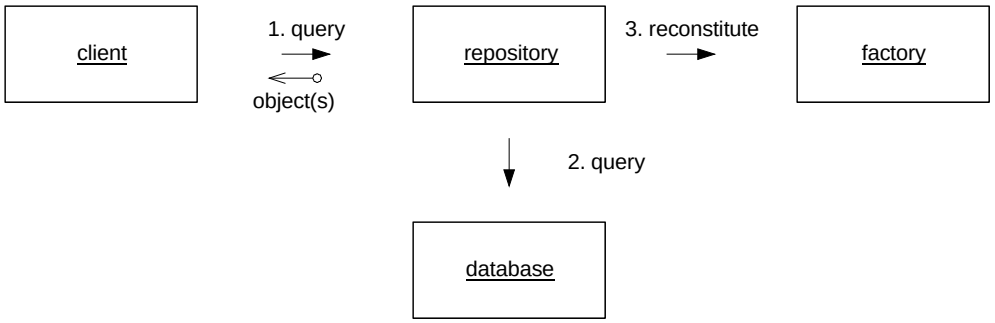


Figure 6. 16

REPOSITORY uses FACTORY to reconstitute preexisting object

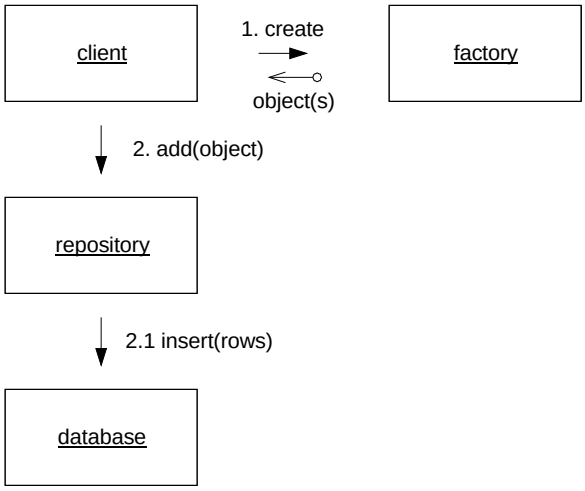


Figure 6. 17

One other case that drives people to combine FACTORY and REPOSITORY is the desire for "find or create" functionality, where a client can describe an object it wants and, if no such object is found, will be given a newly created one. This function should be avoided. It is a minor convenience at best. A lot of cases where it seems useful go away when ENTITIES and VALUE OBJECTS are distinguished. A client that wants a VALUE OBJECT can go straight to a FACTORY and ask for a new one. Usually, the distinction between a new object and an existing object is important in the domain, and a framework that transparently combines them will actually muddle the situation.

Designing Objects for Relational Databases

The most common non-object component of primarily object-oriented software systems is the relational database. This presents the usual problems of a mixture of paradigms (Chapter 5). But the database is more intimately related to the object model than most other components. It is not just interacting with the objects; it is storing the persistent form of the data that makes up the objects themselves. A good deal has been written about the technical challenges of mapping objects to relational tables and effectively storing and retrieving them. A recent discussion can be found in [Fowler2002]. There are reasonably refined tools for creating and managing mappings between the two. Apart from the technical concerns, this mismatch can have a significant impact on the object model.

There are three common cases:

- The database is primarily a repository for the objects
- The database was been designed for another system
- The database is designed for this system, but serves other roles than object store

When the database schema is being created specifically as a store for the objects, it is worth accepting some model limitations in order to keep the mapping very simple. Without other demands on schema design, the database can be structured to make aggregate integrity safer and more efficient as updates are made. Technically, the relational table design does not have to reflect the domain model. Mapping tools are sophisticated enough to bridge significant differences. The trouble is, multiple overlapping models are just too complicated. Many of the same arguments presented for MODEL-DRIVEN DESIGN, and avoiding separate analysis and design models apply to this mismatch. This does entail some sacrifice in the richness of the object model, and sometimes compromises have to be made in the database design (such as selective denormalization), but to do otherwise is to risk losing the tight coupling of model and implementation. This doesn't have to mean the one object/one table level of simplistic mapping. Depending on the power of the mapping tool, some aggregation or composition of objects may be possible. But it is crucial that the mappings be transparent, easily understood by inspection of the code or entries in the mapping tool.

When the database is being viewed as an object store, don't let the data model and the object model diverge far, regardless of the powers of the mapping tools. Sacrifice some richness of object relationships to keep close to the relational model. Compromise some formal relational standards, such as normalization, if it helps simplify the object mapping.

Processes outside the object system should not access such an object store. They could violate the invariants enforced by the objects. Also, their access will lock in the data model so that it is hard to change when the objects are refactored.

On the other hand, there are many cases when the data comes from a legacy or external system that was never intended as a store of objects. In this situation, there are, in reality, two domain models coexisting in the same system. Chapter 14, "Maintaining Model Integrity", deals with this issue in depth. It may make sense to conform to the model implicit in the other system, or it may be better to make the model completely distinct.

Another reason for exceptions is performance. Quirky design changes may have to be introduced to solve execution speed problems.

But for the important common case of a relational database acting as the persistent form of an object-oriented domain, simple directness is best. A table row contains an object, perhaps along with subsidiaries in an AGGREGATE. A foreign key in the table translates to a reference to another ENTITY object. The necessity of sometimes deviating from this should not lead to total abandonment of the principle of simple mappings.

The UBIQUITOUS LANGUAGE can help to tie the two components together. Although it is a lot of work, and mapping may seem to make it unnecessary, corresponding elements in the objects and the relational tables should be named meticulously the same and have the same associations. Subtle differences in relationships will cause a lot of confusion. The tradition of refactoring that has increasingly taken hold in the object world has not really affected relational database design much. What's more, serious data migration issues discourage frequent change. This may create a drag on the refactoring of the object model, but if the object model and the database model start to diverge, transparency can be lost quickly.

Finally, when the database is being created for your system, there are some reasons to go with a schema that is quite distinct from the object model. The database may be used by other software, that will not instantiate objects. The database may require little change, even while there is a lot of change or evolution of the behavior of the objects. Cutting the two loose from each other is a seductive path. It is often taken unintentionally, when the team fails to keep the database current with the model. If taken as a conscious decision, it allows creating a clean database schema, not an awkward one full of compromises to conform to last year's object model.

7. Using the Language in an Example: A Cargo Shipping System

This drastically simplified model of shipping walk through the steps to set the stage for a MODEL-DRIVEN DESIGN, showing the forces that apply and how the patterns of Part II resolve them.

A hypothetical cargo shipping company is developing some new software. Initial requirements are three basic functions:

1. Track key handling of customer cargo
2. Book cargo in advance
3. Send invoices to customers automatically when the cargo reaches some point in its handling.

We'll start from a model that seems to abstract the relevant part of the problem domain.

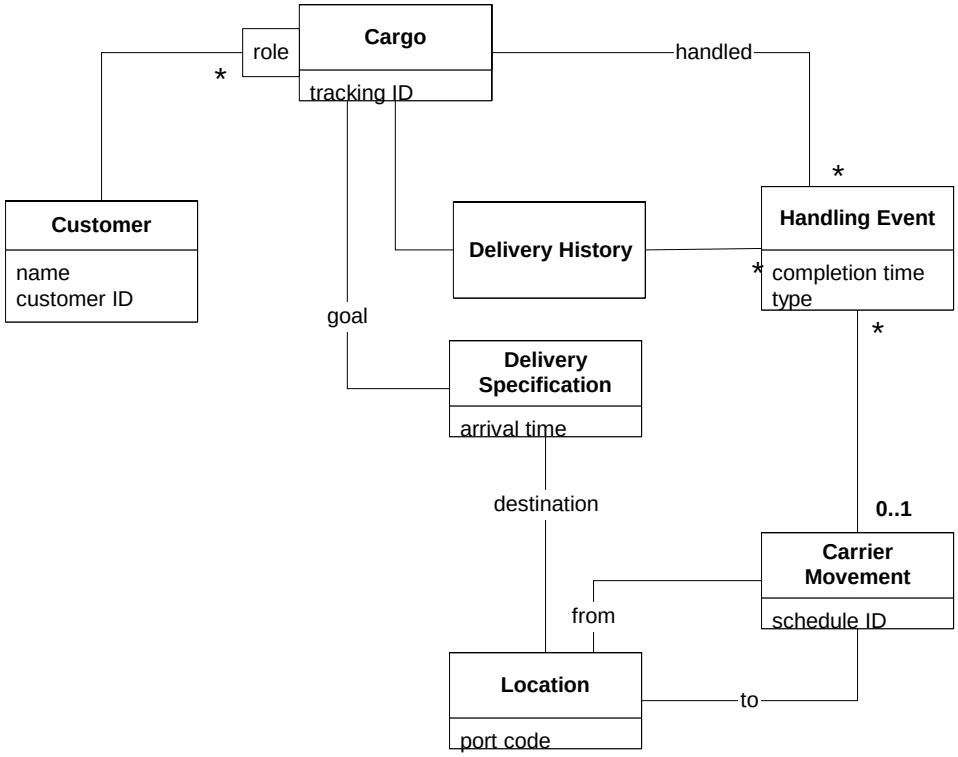


Figure 7. 1

The model represented by this class diagram organizes domain knowledge and provides a language for the team. Statements can be made like:

Multiple customers are involved with a cargo, each playing a different role.

The cargo delivery goal is specified.

A series of carrier movements satisfying the specification will fulfill the delivery goal.

A **Handling Event** is a discrete action taken with the Cargo, such as loading it onto a ship or clearing it through customs. It would probably be elaborated into a hierarchy of different kinds of incidents, such as loading, unloading, or claim by receiver.

Delivery Specification defines a delivery goal, which at can include a destination and an arrival date, but could be more complex. This is a use of the SPECIFICATION pattern (Chapter 9).

This could have been modeled as attributes of Cargo, but the abstraction of Delivery Specification gives at least three advantages. Without Delivery Specification, the Cargo object would be responsible for the detailed meaning of all those attributes, which would clutter it up and make it harder to change. Second, it makes it easy and safe to suppress detail. For example, there could be other criteria encapsulated in the Delivery Specification, but a diagram at this level of detail would not have to expose it. The diagram is telling the reader that there is a SPECIFICATION of delivery, and the details of that are not important to think about (and, in fact, could be easily changed later). Finally, this model is more expressive. Adding Delivery Specification says explicitly that the exact means of delivery of the Cargo is undetermined, but that it must accomplish the goal set out in the Delivery Specification.

A **role** distinguishes the different parts played by Customers in a shipment. One is the "shipper", one the "receiver", one the "payer", and so on. Since only one customer can play a given role in a particular Cargo, the association becomes a qualified many-to-one instead of many-to-many. It might be implemented as simply a string, or could be a class if other behavior was needed.

Carrier Movement represents one particular trip by a particular carrier (such as a truck or ship) from one Location to another. Cargoes can ride from place to place by being loaded onto carriers for the duration of one or more Carrier Movements.

Delivery History reflects what has actually happened to a Cargo, as apposed to the Delivery Specification, which describes goals. A Delivery History object can compute the current location of the Cargo by analyzing the last load or unload and the destination of the corresponding Carrier Movement. A successful delivery would end with a Delivery History that satisfied the goals of the Delivery Specification.

All the concepts needed to work through the requirements described above are present in this model, assuming appropriate mechanisms to persist the objects, find the relevant objects, etc. These issues are not dealt with in the model, but must be in the design.

In order to frame up a solid implementation, this model still needs some clarification and tightening. This example will illustrate that.

Ordinarily, simultaneous with refinement of the model to better support design, the model should be refined to reflect new insight into the domain. But for clarity of explanation, changes will be strictly motivated by the need to connect with a practical implementation, employing the building block patterns.

Once cleaned up, this model will serve for this. *That is because I cheated, for the sake of brevity.* This was not my initial model. I started with something that seemed reasonable, progressed through the steps toward design, found aspects of that model that made design difficult, and then I went back and changed it to make it a better basis for design. Basic iteration is the only way I know to produce useful models beyond the simplest, most obvious level.

Isolating the Domain: Simplified Shipping Applications

To prevent domain responsibilities from being mixed with those of other parts of the system, apply LAYERED ARCHITECTURE to mark off a domain layer.

Without any deep analysis, there seem to be three user level application functions:

1. A Tracking Query that can access past and present handling of a particular cargo.
2. A Booking application that allows a new Cargo to be registered and prepares the system for it.
3. An Incident Logging application that can record each handling of the cargo (providing the information that is found by the Tracking Query).

Remember, these application features are coordinators. They should not work out the answers to the questions they ask. That is the domain layer's job.

Distinguishing ENTITIES and VALUE OBJECTS

Considering each object in turn, we'll look for identity that must be tracked or a basic value that is represented. First we'll go through the clear-cut cases and then consider the more ambiguous ones.

Customer: Let's start with an easy one. A customer is a person or a company, an entity in the usual sense of the word, and it clearly has identity that matters to the user, making it an ENTITY in the model. How to track it? Tax ID might be appropriate in some cases, but an international company could not use that. This calls for consultation with a domain expert. We discuss the problem with a business person in the shipping company and discover that the company already has a customer database in which each customer is assigned an id number at first sales contact. This id is already used throughout the company, and using number in our software will establish continuity of identity between those systems. It will initially be a manual entry.

Cargo: Two identical boxes must be distinguishable, so they are ENTITIES. In practice all shipping companies assign tracking ID's to each. This id will be automatically generated, visible to the user, and, in this case, probably conveyed to the customer at booking time.

Handling Event and **Carrier Movement**: We care about such individual incidents because they allow us to keep track of what is going on. They reflect real-world events, and those are not usually interchangeable, so they are ENTITIES. Each Carrier Movement will be identified by a code obtained from a shipping schedule.

Another discussion with a domain expert concludes that Handling Events can be uniquely identified by the combination of Cargo Id, completion time, and type. For example, the same cargo cannot be both loaded and unloaded at the same time.

Location: Two places with the same name are not the same. Latitude and longitude could provide a unique key, but probably not a very practical one, since it is not of interest to most purposes of this system, and would be fairly complicated. More likely, the **Location** will be part of a geographical model of some kind that will relate places, and an arbitrary, internal, automatically generated identifier will suffice.

Delivery History: This is a tricky one. Delivery Histories are not interchangeable, so they are ENTITIES. But a Delivery History has a one-to-one relationship with its Cargo, so it doesn't really have an identity of its own. Its identity is borrowed from the Cargo that owns it. This will become clearer when we model the AGGREGATES.

Delivery Specification: Although it represents the goal of a Cargo, the abstraction does not depend on Cargo. It really expresses a hypothetical state of some Delivery History. We hope that the Delivery History attached to our Cargo will eventually satisfy the Delivery Specification attached to our Cargo. But if we had two cargoes going to the same place they could share the same Delivery Specification, while they could not share the same History, even though the histories start out the same (empty). Specifications are VALUE OBJECTS.

role: It says something about the association it qualifies, but has no history or continuity. It is a VALUE OBJECT, and could be shared among different Cargo/customer associations.

other attributes: (such as time stamps or names) are VALUE OBJECTS.

Designing Associations in the Shipping Domain

None of the associations in the original diagram specified a traversal direction, but bi-directional associations are problematic in a design. Also, traversal direction often captures insight into the domain, deepening the model itself.

If the Customer has a direct reference to every Cargo it has shipped it will become cumbersome for long-term repeat Customers. Also, the concept of a Customer is not specific to Cargo. In a large system, the customer may have roles to play with many objects. Best to keep it free of such specific responsibilities. If we need the ability to find cargoes by customer, this can be done through a database query. We'll return to this in the section on REPOSITORIES.

If our application was tracking the inventory of ships, traversal from Carrier Movement to Handling Event would be important. But our business only needs to track the Cargo. Making the association traversable only from Handling Event to Carrier Movement captures that understanding of our business. This also reduces the implementation to a simple object reference, since the direction with multiplicity was disallowed.

The rationale of the remaining decisions is marked on this diagram.

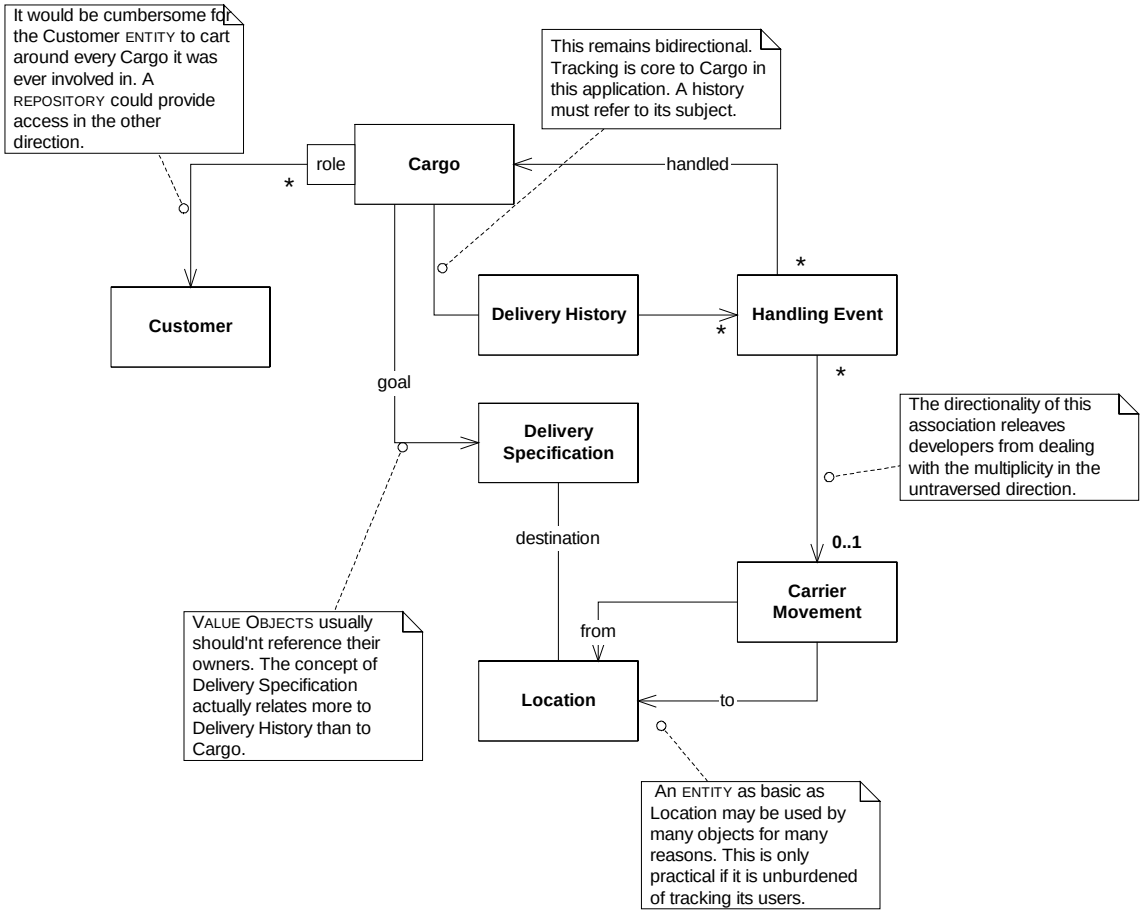


Figure 7. 2

There is one circular reference: Cargo knows its Delivery History which holds a series of Handling Event, which in turn point back to the Cargo. Circular references logically exist in many domains and are sometimes necessary in design as well, but they are tricky to maintain. Implementation choices can help by avoiding holding the same information in two places that mush be kept synchronized. In this case, we can use a simple but fragile implementation in an initial prototype, with an Array List (Java) of Handling Events in Delivery History. But at some point we'll probably want to drop the collection in favor of a database lookup with Cargo as the key. This discussion will be taken up again when choosing REPOSITORIES. If the query to see the history is relatively infrequent, this should give good performance and simplify maintenance and reduce the overhead of adding Handling Events. If this query is very frequent, then it is better to go ahead and maintain the direct pointer. These design tradeoffs are based on simplicity of implementation and on performance. The model is the same, and contains the cycle and the bi-directional association.

AGGREGATE Boundaries

Customer, Location and Carrier Movement have their own identity and are shared by many Cargoes, so they must be the roots of their own AGGREGATES, which contain their attributes and possibly other objects below the level of detail of these discussions. Cargo is also an obvious AGGREGATE root, but where to draw the boundary takes some thought.

We could sweep in everything that only exists because of this Cargo, which would include the Delivery History, the Delivery Specification, and the Handling Events. The Delivery History seems to be something no one would look up directly without the Cargo itself. With no need for direct global access, and with an identity that is really just derived from the Cargo, the Delivery History fits nicely inside of Cargo's boundary, and does not need to be a root. The Delivery Specification is a VALUE OBJECT, so that presents no complications.

The Handling Event is another matter. Previously we have considered two possible database queries that would search for these: one to find what was loaded onto a particular Carrier Movement; another as a possible alternative to finding the Handling Events for a Delivery History, which would be local within the Cargo AGGREGATE. It seems that the activity of handling the Cargo has some meaning even when considered apart from the Cargo itself. It could be drawn either way. It should be the root of its own AGGREGATE.

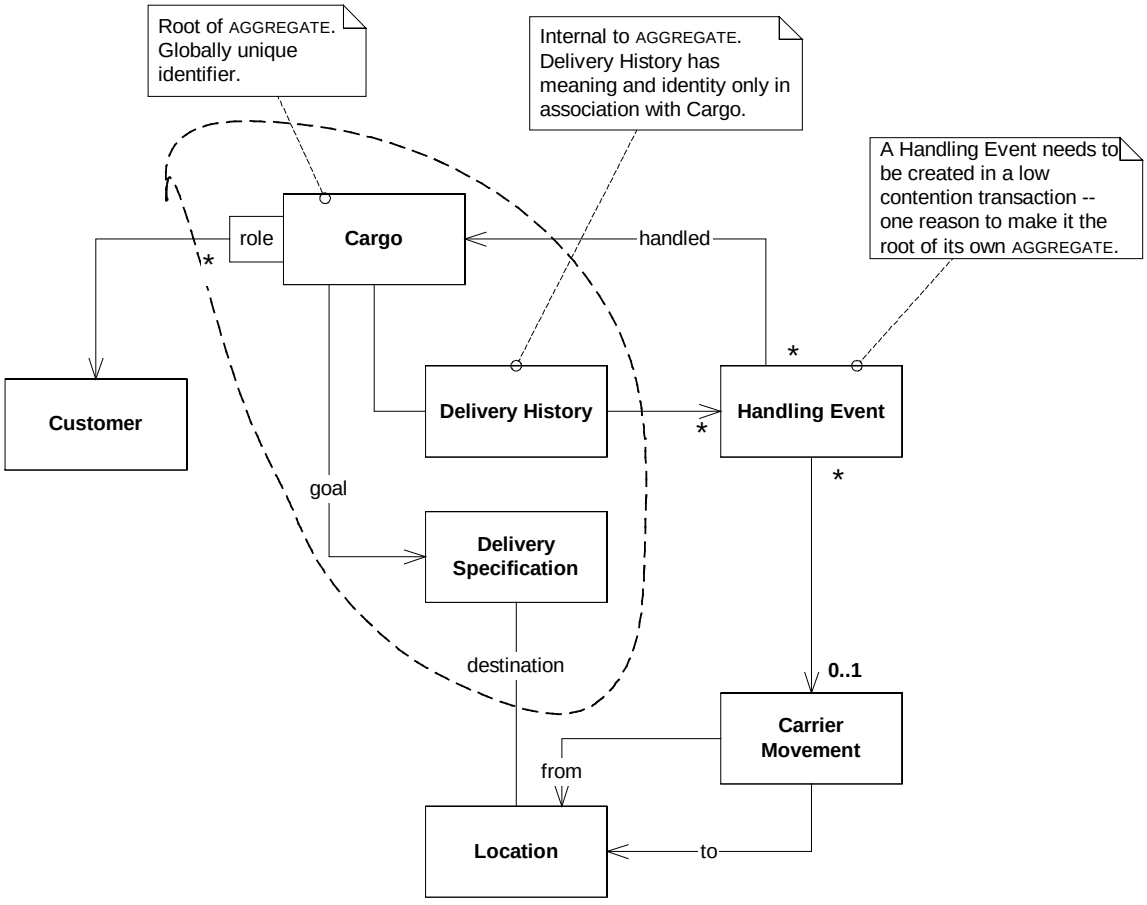


Figure 7. 3

Selecting REPOSITORIES

There are five ENTITIES in the design that are roots of AGGREGATES, so we can limit our consideration to these, since none of the other objects are allowed to have REPOSITORIES.

Now we must consider the application requirements. In order to take a booking through the Booking Application, the user to select the Customer(s) playing the various roles (shipper, receiver, etc), so we have a Customer Repository. We also need to find a Location to specify as the destination for the Cargo, so we have a Location Repository.

The Activity Logging Application needs to allow the user to look up the Carrier Movement that a Cargo is being loaded onto, so we need a Carrier Movement Repository. This user must also tell the system which Cargo has been loaded, so we need a Cargo Repository.

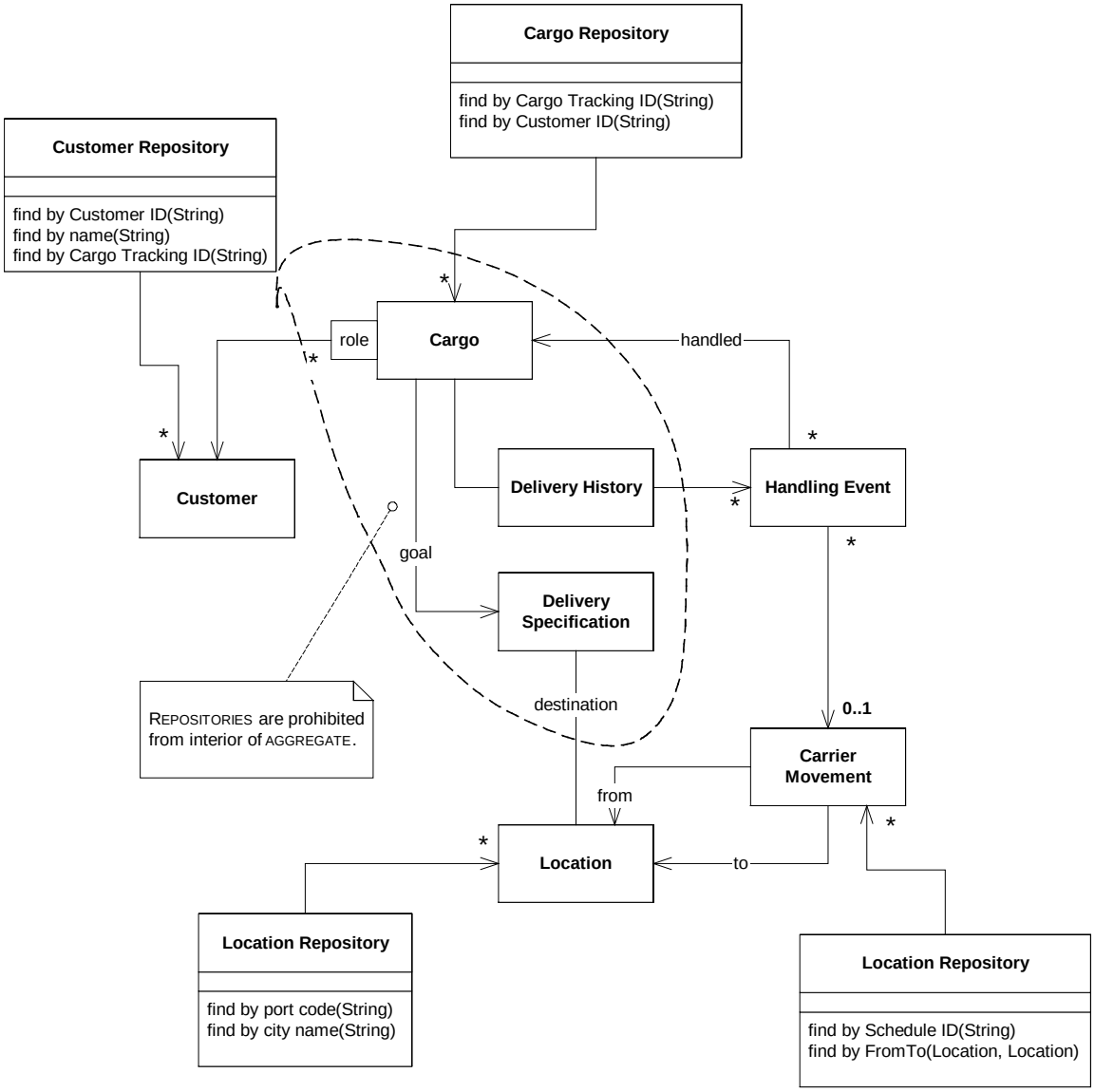


Figure 7. 4

For now there is no Handling Event Repository because we decided to implement the association with Delivery History as a collection in the first iteration, and we have no application requirement to find out what has been loaded onto a Carrier Movement. Either of these reasons could change, and then we would add a REPOSITORY.

To crosscheck all these decisions, we have to constantly step through scenarios to confirm that we can solve application problems effectively.

Sample Application Feature: Changing the Destination of a Cargo

Occasionally a customer calls up and says, "Oh, no. We said to send our Cargo to Hackensack, but we really need it in Hoboken". We are here to serve, so the system is required to provide for this change.

Delivery Specification is a VALUE OBJECT, so it would be simplest to just to throw it away and get a new one, then use setter method on Cargo to replace old one with new one.

Sample Application Feature: Repeat Business

The users say that repeated bookings from the same customers tend to be similar, so they want to use old Cargoes as prototypes for new ones. The application will allow them to find a Cargo in the REPOSITORY and then select a command to create a new cargo based on the selected one. We'll design this using the PROTOTYPE pattern [GHJV95].

Cargo is an ENTITY and is the root of an AGGREGATE. Therefore, it must be copied carefully, deciding what should happen to each object or attribute enclosed by its AGGREGATE boundary. Let's go over each one:

Delivery History: We should create a new, empty one, since the history of the old one doesn't apply. This is the usual case with ENTITIES inside the AGGREGATE boundary.

Customer Roles: We should copy the Map (or other collection) that holds the keyed references to Customers, including the keys, since they are likely to play the same roles in the new shipment, but we have to be careful *not* to copy the Customers. We must end up with references to the same Customers as the old Cargo referenced, since they are ENTITIES outside the AGGREGATE boundary.

Tracking ID: We must provide a new tracking ID from the same source as we would when creating a new Cargo from scratch.

Notice that we have copied everything inside the Cargo AGGREGATE boundary, made some modifications to the copy, and have *affected nothing outside the* AGGREGATE *boundary* at all.

FACTORIES and Constructors for Cargo

Even if we have a fancy FACTORY for Cargo, or use another Cargo as the FACTORY, as in Repeat Business, above, we still have to have a primitive constructor. We would like the constructor to produce an object that fulfills its invariants or at least, in the case of an ENTITY, has its identity intact.

Given these decisions, we might create a method on Cargo such as:

```
public Cargo copyPrototype(String newTrackingID)
```

Or we might make a new FACTORY METHOD such as:

```
public newCargo(Cargo prototype, String newTrackingID)
```

A FACTORY METHOD could also encapsulate the process of obtaining a new (automatically generated) id for a new Cargo, in which case it would need no arguments. The result would have an empty Delivery History, and a null Delivery Specification.

The two-way pointer between Cargo and Delivery History means that neither Cargo nor Delivery History is complete without pointing to its counterpart, so they must be created together. Remember that Cargo is the root of the AGGREGATE that includes Delivery

History. Therefore, we can allow Cargo's constructor to create a Delivery History. The Delivery History constructor will take a Cargo as an argument. Something like this:

```
public Cargo(String id) {
    trackingID = id;
    deliveryHistory = new DeliveryHistory(this);
    customerRoles = new HashMap();
}
```

The result is a new Cargo with a new Delivery History that points back to the Cargo. The Delivery History constructor is used exclusively by its AGGREGATE root, namely Cargo, so that the composition of Cargo is encapsulated.

Adding a Handling Event

Each time the cargo is handled in the real world, some user will enter a "Handling Event" using the Incident Logging Application. The basic constructor for Handling Event was presented above, but the story isn't quite that simple. The Delivery History holds a collection of Handling Events relevant to its Cargo, and the new object must be added to this collection as part of the transaction.

Figure 7. 5

The Handling Event model is an abstraction that might encapsulate a variety of specialized Handling Event classes, ranging from loading and unloading to sealing, storing, and other not related to carriers. They might be implemented as multiple subclasses and/or have complicated initialization. In this case, a single FACTORY, used by all Handling Events,

could abstract instance creation, freeing the client from knowledge of the implementation. Methods on the FACTORY would be provided to request instances to suit specific needs, and the factory would be responsible for knowing what class was to be instantiated.

Pause for Iteration: An Alternative Design of the Cargo Aggregate

Modeling and design is not a constant forward process. It will grind to a halt unless there is frequent refactoring to take advantage of new insights to improve the model and the design.

By now, there are a couple of cumbersome aspects to this design, although it does work and it does reflect the model. Problems that didn't seem important when starting the design are beginning to be annoying. Let's go back to one of them and, with the value of hindsight, stack the design deck in our favor.

The need to update Delivery History when adding a Handling Event gets the Cargo AGGREGATE involved in the transaction, which could cause delays if the Cargo is being updated by another transaction. An important application requirement is the ability to enter Handling Events without contention, since it is an operational activity that needs to be quick and simple. This pushes us to consider a different design.

Replacing the Delivery History's collection of Handling Events with a query would allow Handling Incidents to be added without any integrity issues outside its own aggregate, which would allow such transactions to complete without interference. If there are a lot of Handling Events being entered and relatively few queries, this design is more efficient. In fact, if a relational database is the underlying technology, a query was probably being used under the covers anyway to emulate the collection. It would, incidentally, reduce the difficulty of maintaining consistency in the cyclical reference between Cargo and Handling Incident.

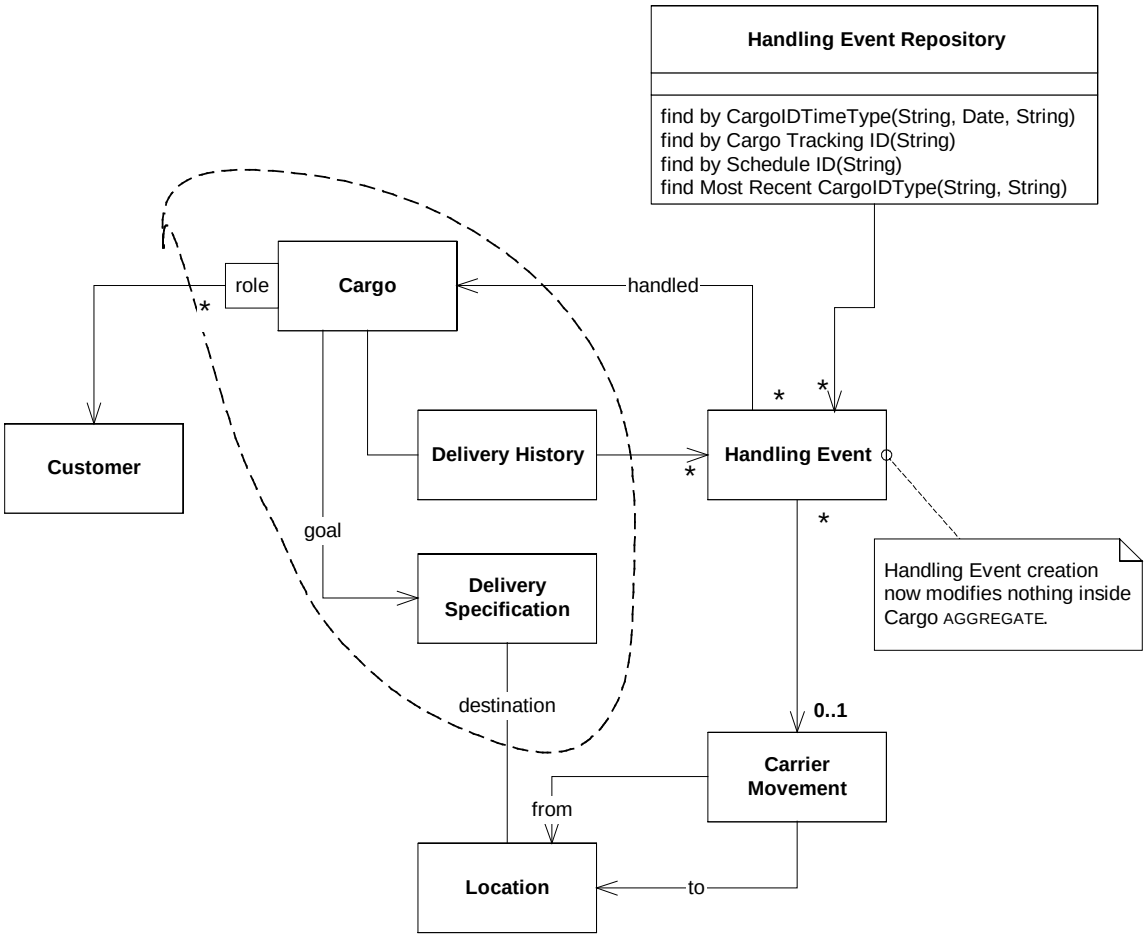


Figure 7. 6

To take responsibility for the queries, we'll add a REPOSITORY for Handling Events. The Handling Event Repository will support a query for the Events related to a certain Cargo. In addition, the REPOSITORY can provide queries optimized to answer specific questions efficiently. For example, if a frequent access path is the Delivery History finding the last reported load or unload, in order to infer the current status of the Cargo, a query could be devised to return just that relevant Handling Event. And if we wanted a query to find all Cargos loaded on a particular Carrier Movement, we could easily add it.

This leaves the Delivery History with no persistent state. At this point, there is no real need to keep it around. We could derive Delivery History itself whenever it is needed to answer some question. We can do this because, although the ENTITY will be repeatedly recreated, we can maintain the tread of continuity between its incarnations by each time associating it with the same Cargo.

The circular reference is no longer tricky to create and maintain. The Cargo FACTORY will be changed to no longer attach an empty Delivery History to new instances. Database space can be reduced slightly, and the actual number of persistent objects might be reduced considerably, which is a limited resource in some object databases. If the common usage pattern is that the user seldom queries for the status of a Cargo until it arrives, then a lot of unneeded work will be avoided altogether.

On the other hand, if we are using an object database, traversing an association or an explicit collection is probably much faster than a REPOSITORY query. If the access pattern includes frequent listing of the full history, rather than the occasional targeted query of last position, the performance tradeoff might favor the explicit collection. And remember that the added feature ("What is on this Carrier Movement?") hasn't been requested yet, and may never be, so we don't want to pay much for that option.

These kinds of alternatives and design tradeoffs are everywhere, and I could come up with lots of examples just in this little simplified system. But the important point is that these are degrees of freedom within the same model. By modeling VALUES, ENTITIES, and their AGGREGATES as we have, we have reduced the impact such design changes have. For example, in this case all changes are encapsulated within the Cargo's AGGREGATE boundary. It also required the addition of Handling Event Repository, but did not call for any redesign of the Handling Event itself (although some implementation changes might be involved, depending on details of the REPOSITORY framework).

Primitive Constructors for the Handling Event

Every class must have primitive constructors that are passed, in some form, the desired attributes and return the new instance. The Handling Event constructor must take a Cargo and a Carrier Movement as arguments.

```
public HandlingIncident(Cargo c, CarrierMovement m, Time t) {
    loaded = c;
    loadedOnto = m;
    timeStamp = t;
}
```

Another constructor could be used for handling other than loading.

```
public HandlingIncident(Cargo c, Time t)
```

MODULES in the Shipping Model

So far we've been looking at so few objects that the modularity is not an issue. Now let's look at a little bigger part of a shipping model (though still simplified, of course) to see its organization into MODULES will affect the model.

This next diagram shows a model neatly partitioned by a hypothetical enthusiastic reader of this book.

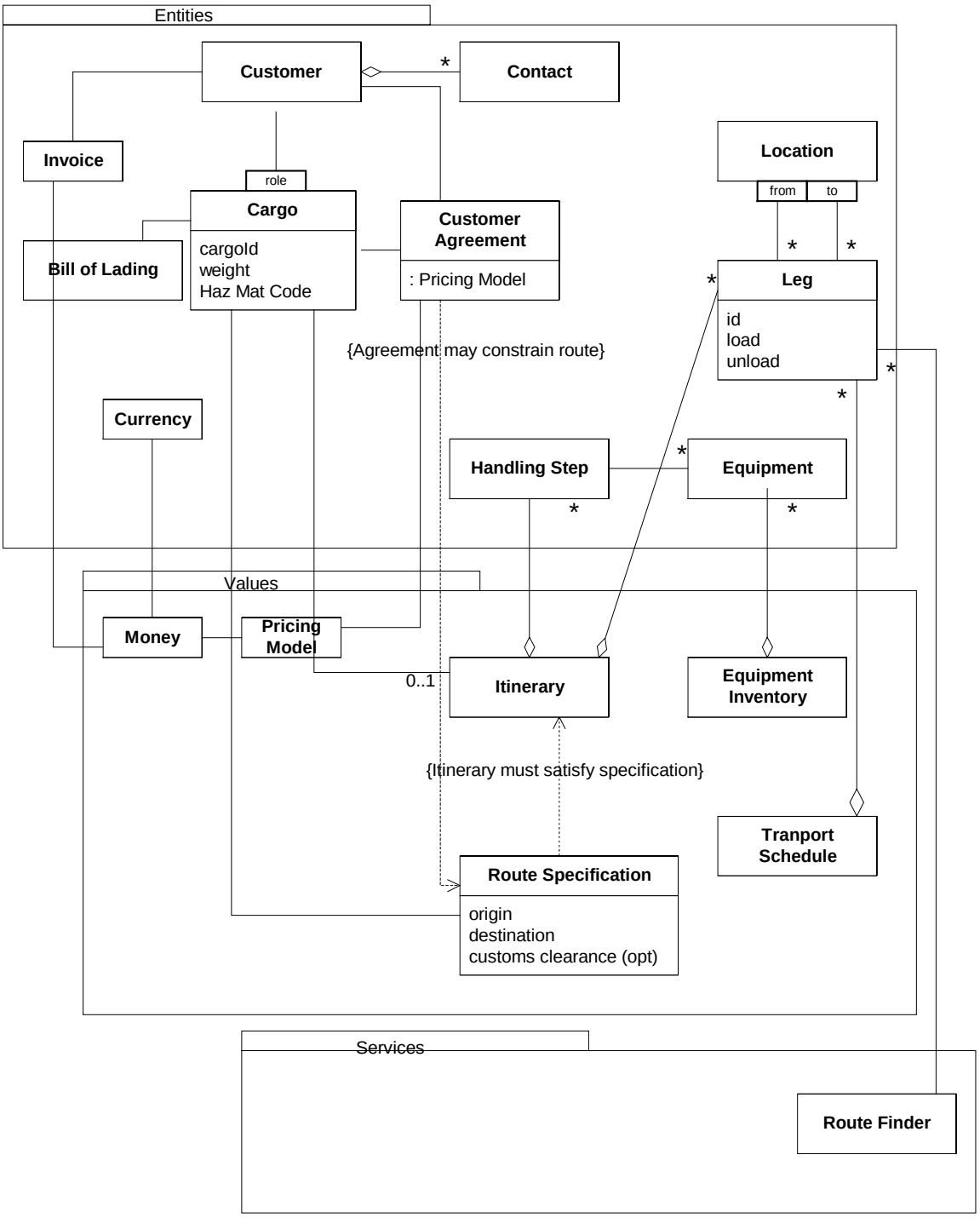


Figure 7. 7

This is a variation on the infrastructure driven packaging problem raised in Chapter ##. In this case, the objects have been grouped according to the pattern each follows. The result is that objects are crammed together that conceptually have little relationship (low cohesion), and associations run willy-nilly between all the MODULES (high coupling). The

packages tell a story, but it is not the story of shipping; it is the story of what the developer was reading at the time.

Partitioning by pattern may seem like an obvious error, but it is not really any less sensible than separating persistent objects from transient ones or any other methodical scheme that is not grounded in the meaning of the objects.

Instead, we should be looking for the cohesive concepts and focusing on what we want to communicate to others on the project. As with smaller-scale modeling decision, there are many ways to do it. Here is a straightforward one.

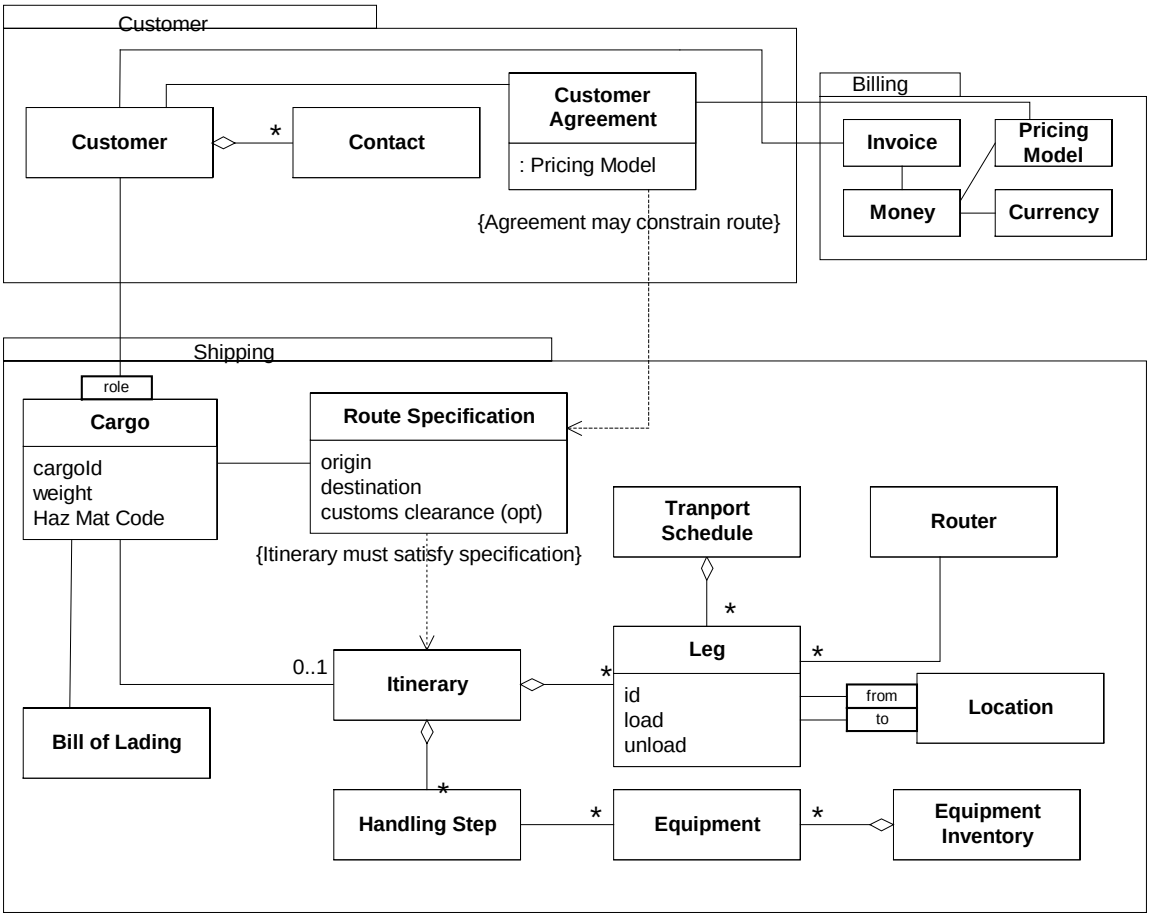


Figure 7. 8

Our company does *shipping* for *customers* so that we can *bill* them. Our sales and marketing people deal with *customers*, and make agreements with them. The operations people do the *shipping*, getting the cargo to its specified destination. The back-office takes care of *billing*, submitting invoices according to the pricing in the agreement. That's one story I can tell with this set of MODULES.

This is one intuitive breakdown. It could be refined, certainly, in successive iterations, but it is now aiding MODEL-DRIVEN DESIGN and contributing to the UBIQUITOUS LANGUAGE.

New Feature: Allocation Checking

Now the first major new functions are going to be added. In this design we will reabstract the domain of an external system and hide it behind an ANTICORRUPTION LAYER (a pattern discussed in Chapter ##).

The sales division of the imaginary shipping company uses other software to manage client relationships, sales projections, and so forth. One feature supports yield management by allowing them to allocate how much cargo of specific types they will attempt to book based on the type of goods, the origin and destination, or any other factor they may choose that can be entered as a category name. These constitute goals of how much will be sold of each type, so that more profitable types of business will not be crowded out by less profitable cargoes, while at the same time avoiding under-booking, (not fully utilizing their shipping capacity) or excessive overbooking (so that so much cargo gets bumped that it affects customer relationships).

Now they want this to be integrated with the booking system. When a booking comes in, they want it checked against these allocations to see if it should be accepted.

The information needed resides in two places, which will have to be queried by the booking application coordinator so that it can either accept or reject the requested booking. A first stab might look like this.

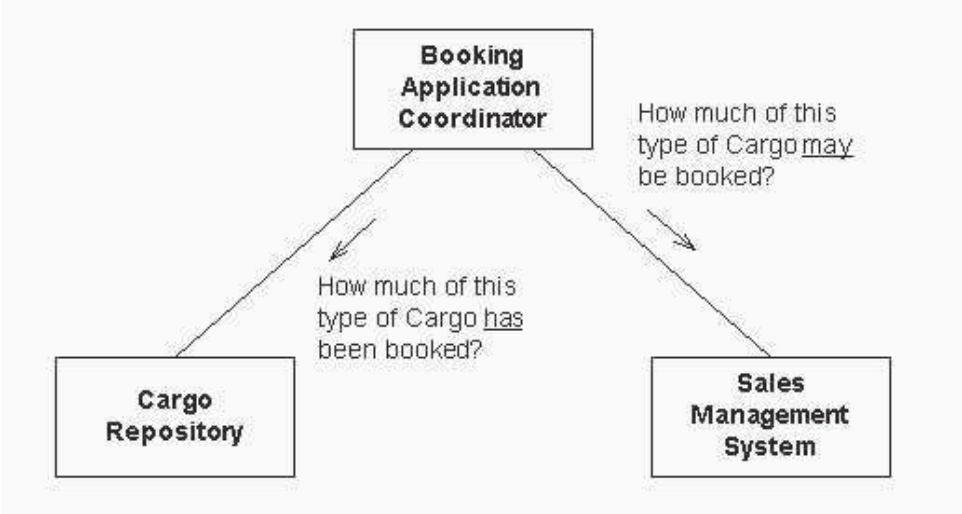


Figure 7. 9

The anti-corruption layer pattern should be applied, giving us a façade and adapter to control the interface to the sales management system. We could call it something like "Sales Management Interface". This would allow us to hide all the mechanics of talking to the other program and give us a global access point. But we would be missing an opportunity to recast the problem along lines more useful to us. Instead, lets give it a name related to its responsibility in our system, and call it "Allocation Manager", and present it as a SERVICE. All services related to allocation will be channeled through here.

If some other integration is needed (for example, using the sales management system's customer database instead of our own customer repository), another façade can be created with services fulfilling that responsibility. The lower level "Sales Management Interface" might be useful for shared machinery of talking to the other program, but it would be hidden behind the other façade(s), and wouldn't show up in the domain design.

Now, what kind of interface are we going to supply that can answer the question, "How much of this type of Cargo may be booked?" The tricky issue is to define what "type" it is,

since our domain model does not categorize cargoes yet. In the sales management system, this is just a set of category key words, and we could conform to this. We could pass a collection of strings in as an argument. But we would be passing up another opportunity – this time, to reabstract the domain of the other system. Let's do a little domain modeling.

An analysis pattern (see Chapter 11) that addresses this kind of problem is the ENTERPRISE SEGMENT, described by [Fowler96], p64. An ENTERPRISE SEGMENT is a set of dimensions that define a way of breaking down a business. These dimensions could include all those mentioned above and also time dimensions, such as month-to-date. Using this concept in our model of allocation makes the model more expressive and simplifies the interfaces. The Enterprise Segment will appear in our domain model and design as an additional value object which will have to be derived for each Cargo.

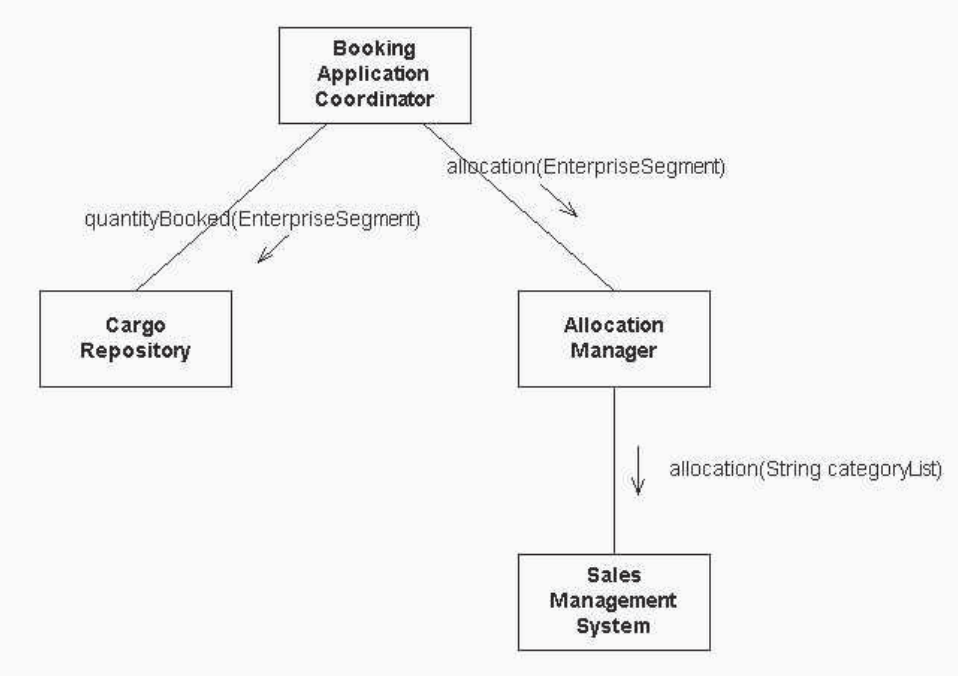


Figure 7. 10

It will be necessary for the Allocation Manager to translate between Enterprise Segments and the category names of the external system. The Cargo Repository must also provide a query based on enterprise segment. In both cases, collaboration with the Enterprise Segment object can be used to perform the operations without breaching the Segment's encapsulation and complicating their own implementations. (Notice that the Cargo Repository is answering a query with a count, rather than a collection of instances

There are still a few problems with this design. We have given the Booking Application Coordinator the job of applying a business rule, which is a domain responsibility and shouldn't be performed by an application coordinator. Also, it isn't clear who is responsible for deriving the Enterprise Segment. Both of these seem to belong to the Allocation Manager.

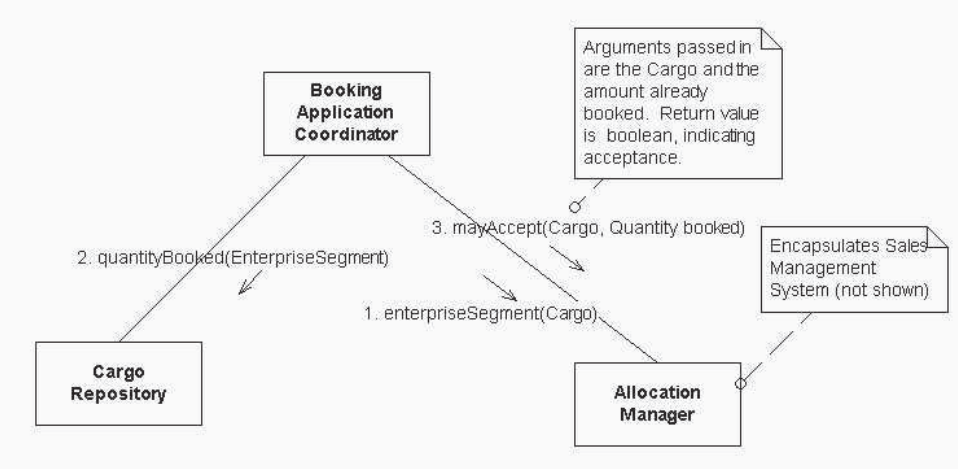


Figure 7. 11

The interface of the Allocation Manager is the only part that needs to be considered in the rest of the domain design. It does have its own internal design, which can present opportunities. One worth contemplating is making the objects responsible for deriving enterprise segment relocatable. If there is communications overhead from going to the sales management system (which may be on another server) it would be useful to cache on the client the data and behavior needed to derive this value, which should be relatively static compared to the allocations themselves. Flexible deployment is an important design goal in distributed systems.

The only serious constraint imposed by this integration will be that the sales system mustn't use dimensions that the Allocation Manager can't turn into Enterprise Segments. (Without Enterprise Segment, the same constraint would take the form that the sales system mustn't use dimensions that the Cargo Repository can't use in a query. This is feasible, but the sales system spills into other parts of our domain, whereas in this design, the Cargo Repository need only be designed to handle Enterprise Segment, and changes in the sales system ripple only as far as the Allocation Manager, which was conceived as a façade in the first place.)

That's it. This integration could have turned our simple, conceptually consistent design into a tangled mess, but now, using ANTI-CORRUPTION LAYER, SERVICE and ENTERPRISE SEGMENT, we have integrated the functionality of the Sales Management System into our booking system cleanly, enriching the domain.

A final design question: Why not give Cargo the responsibility of deriving the Enterprise Segment. At first glance it seems elegant, if all the data the derivation is based on is in the Cargo, to make it a derived attribute of Cargo. Unfortunately, it is not that simple. Enterprise Segments are defined arbitrarily to divide along lines useful for business strategy. The same ENTITIES could be segmented differently for different purposes. We are deriving the segment for a particular cargo for `*booking allocation* purposes, but it could have a completely different Enterprise Segment for tax accounting purposes. Even the allocation enterprise segment could change if the sales management system is reconfigured because of a new sales strategy. So the Cargo would have to know about the Allocation Manager, which is well outside its conceptual responsibility, and it would be laden with methods for deriving specific types of enterprise segment. Therefore, the responsibility for deriving this value lies properly with the object that knows the rules for segmentation, rather than the object that has the data that those rules are applied to. Those rules could be split out into a separate "strategy" object, which could be passed to a Cargo to allow it to derive an Enterprise Segment. That seems to go beyond the

requirements we have here, but it would be an option for a later design and shouldn't be a very disruptive change.

Part III. Refactoring Toward Deeper Insight

The previous section laid a foundation for maintaining the correspondence between model and implementation. The use of a proven set of basic building blocks along with consistent language brings some sanity to the development effort.

Of course, that is only the foundation. The real challenge is to actually *find* an incisive model, one that captures subtle concerns of the domain experts and that can drive a practical design. Ultimately, we hope to develop a model that captures a deep understanding of the domain. This should make the software more in tune with the way the domain experts think and more responsive to the user's needs. This section will clarify that goal, describe the process by which it can be approached, and explain some design principles and patterns to apply to make the design accommodate the needs of the application and the developers themselves.

Success with domain-driven design comes down to three points.

1. Sophisticated domain models are achievable and worth the trouble.
2. They are seldom developed except through an iterative process of refactoring, including close involvement of the domain experts with developers interested in learning about the domain.
3. They may call for sophisticated design skills to implement and use effectively.

Levels of Refactoring

Refactoring is the redesign of software in ways that do not change its functionality. Rather than making elaborate upfront design decisions, code is taken through a continuous series of small, discrete design changes, each leaving existing functionality unchanged while making the design more flexible or easier to understand. A suite of automated unit tests allow relatively safe experimentation with the code. The process frees the developers from the need to look far ahead.

But nearly all the literature on how to refactor focuses on mechanical changes to the code that make it easier to read or to enhance at a very detailed level. The approach of "refactoring to patterns"[5] can give a higher-level target to the refactoring process when a developer recognizes an opportunity to apply an established design pattern. Still, it is a primarily technical view of the quality of the design.

The refactorings that have the greatest impact on the viability of the system are those motivated by new insights into the domain or those that clarify the model's expression through the code. This does not in any way replace the refactorings to design patterns or the micro-refactorings, which should proceed continuously. It superimposes another level -- refactoring to a deeper model. Executing a refactoring to a deeper insight often involves a series of micro-refactorings. But the motivation is not just the state of the code. Rather, the micro-refactorings provide convenient units of change toward a more insightful model.

The result is that not only can a developer understand what the code does; he or she can understand *why* it does what it does and can relate that to the ongoing communication with the user community. The catalog in *Refactoring* [Fowler 1999] covers most of the micro-refactorings that come up regularly. Each is motivated primarily by some problem that can be observed in the code itself.

By contrast, domain models are transformed in such a range of ways as new insights emerge that a comprehensive catalog does not seem to be a near-term prospect. Modeling is as inherently unstructured as any exploration. Refactoring to deeper insight should follow wherever learning and deep thinking leads. As we'll see, published collections of successful models can be helpful, but we shouldn't get sidetracked trying to reduce domain modeling to a cookbook or a toolkit. It is an opportunity for creativity. The next few

[5] Patterns as targets for refactoring was briefly mentioned in [GHJV 1995]. Joshua Kerievsky has developed refactoring to patterns into a more mature and useful form.

chapters will suggest some specific approaches to thinking about improving a domain model, along with the design that brings it to life.

Deep Models

The traditional way of explaining object analysis involves identifying nouns and verbs in the requirements documents and using them as the initial objects and methods. This is recognized as an oversimplification that can be useful for explaining object modeling to beginners. The truth is, though, that initial models usually are naïve and superficial, based on shallow knowledge.

For example, I once worked on a shipping application where my initial idea of an object model involved ships and containers. Ships moved from place to place. Containers were associated and disassociated through load and unload operations. That is an accurate description of some physical shipping activities. It does not turn out to be a very useful model for shipping business software.

Eventually, after months working with shipping experts through many iterations, we evolved a quite different model. It was less obvious, but much more relevant to the experts. It was refocused on the business of delivering cargo.

The ships were still there, but highly abstracted in the form of a "vessel voyage", a particular trip scheduled for a ship, train, or other carrier. The ship itself was secondary, and could be substituted at the last minute for maintenance or a slipping schedule, while the vessel voyage went on as planned. The container all but disappeared. It did emerge in a cargo handling application in a different, very complex form, but in the context of the original application, it was an operational detail. The physical movement of the cargo took a back seat to the transfers of legal responsibility for that cargo. Less obvious objects, like the "bill of lading" came to the fore.

Whenever new object modelers showed up on the project, what was their first suggestion? The missing classes -- ship and container. They were smart people. They just hadn't gone through the processes of discovery of a deep model.

A "deep model" provides a lucid expression of the primary concerns of the domain experts and their most relevant knowledge while it sloughs off the superficial aspects of the domain. This definition doesn't mention abstraction. A deep model usually has abstract elements, but it may well have concrete elements where those cut to the heart of the problem. You find it by living in the domain and continuing to crunch knowledge and refactor until you have a model fits your needs.

Versatility, simplicity, and explanatory power come from a model that is truly in tune with the domain. One feature these models almost always have is a simple, though possibly abstract, language that the business experts like to use.

Deep Model – Supple Design

In a process of constant refactoring, the design itself needs to support change. Chapter 10 looks at ways to make a design easy to work with, both for those changing it and those integrating it with other parts of the system.

Certain characteristics of a design make it easier to change and use. They are not complicated, but they are challenging. "Supple design" and ways to approach it are the subjects of Chapter 11.

One bit of luck is that the very act of transforming the model and code again and again, if each change reflects new understanding, can bring about flexibility at just the points where change is most needed, along with easy ways of doing the common things. A well-worn glove becomes supple at the points where the fingers bend, while other parts are stiff and protective. So, although there is a lot of trial and error involved in this approach to modeling and design, the changes can actually become easier to make, and the repeated changes actually move us toward a supple design.

In addition to facilitating change, a supple design contributes to the refinement of the model itself. A MODEL-DRIVEN DESIGN stands on two legs. A deep model makes possible an expressive design. At the same time, a design with the flexibility to let a developer experiment and the clarity to show a developer what is happening actually feeds insight into the model discovery process. This half of the feedback loop is essential because the model we are looking for is not just a nice set of ideas, it is the foundation of the system.

The Discovery Process

To create a design really fitted to the problem at hand, you must first have a model that captures the central relevant concepts of the domain. Actively searching for these and bringing them into the design is the subject of Chapter 9, “Making Implicit Concepts Explicit”.

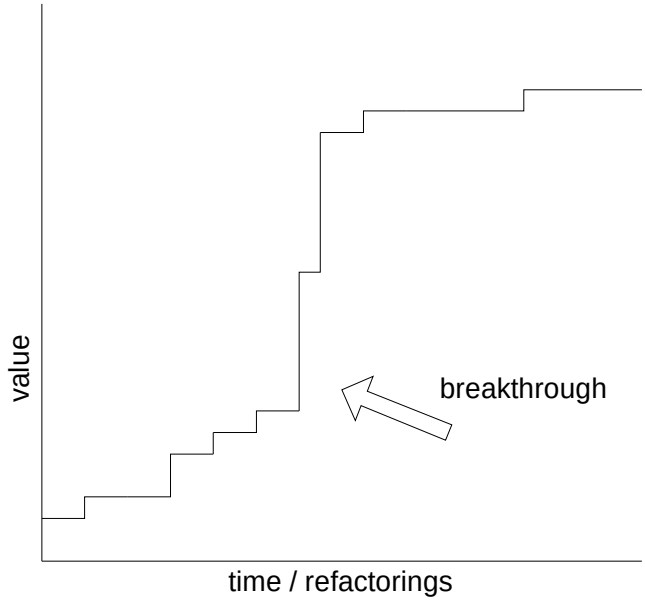
Because of the close relationship between model and design, the modeling process comes to a halt when the code is hard to refactor. Chapter 10, “Supple Design” discusses how to write software for software developers, not least yourself, so that it is productive to extend and change. This goes hand in hand with further refinements to the model. It often entails more advanced design techniques and more rigor in model definitions.

You will usually have to call on creativity and trial and error to find good ways to model the concepts you discover, but sometimes someone has laid down a pattern you can follow. Chapters 11 and 12 discuss the application of “analysis patterns” and “design patterns”. These few patterns have some value in themselves, but they are only a sampling. They serve as examples of refactorings to more useful models, to help develop habits of thought, and to provide points of departure in looking for ways to deepen your own models.

But I’ll start Part III with the most exciting event in domain-driven design. Sometimes, when the stage is set with a MODEL-DRIVEN DESIGN and explicit concepts, you have a breakthrough. An opportunity opens up to transform your software into something more expressive and versatile than you expected. This can mean new features or it can just mean the replacement of a big chunk of rigid code with a simple, flexible expression of a deeper model. While this does not happen every day, it is so valuable when it does come up that the opportunity needs to be recognized and grasped.

Chapter 8 tells the true story of a project on which a process of refactoring toward deeper insight led to a breakthrough. While this is not something you can set out to do or plan, it provides a good context for thinking about domain refactoring.

8. Breakthrough



The returns from refactoring are not linear. Usually there is a marginal return for a small effort. These small improvements add up and fight entropy and are the number one protection from a fossilized legacy.

Slowly but surely, the team assimilates knowledge and crunches it into a model. Deep models can emerge gradually through a sequence of small refactorings, an object at a time, an association tweak here, a shifted responsibility there. But they are often shocks.

As insight deepens, refinement of code and model gives a clearer view. This new clarity creates the potential for a breakthrough of insights. A rush of change leads to a model that corresponds on a deeper level to the realities and priorities of the users. Versatility and explanatory power suddenly increase even as complexity evaporates.

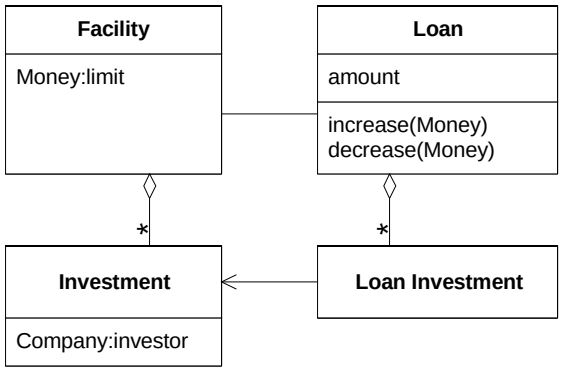
This is not a technique; it is an event. It calls for recognizing what is happening and deciding how to deal with it. To convey what this experience feels like, I'll tell a true story of a project I worked on some years ago, and how we arrived at a very valuable deep model.

A Decent Model, and Yet…

After a long, New York winter of refactoring, we had arrived at a model that captured some of the key knowledge of the domain and a design that did some real work for the application. We were developing a core part of a large application for managing syndicated loans in an investment bank.

Model That Assumes Lender Shares are Fixed

Figure 8.1



What is a "facility"?

A "facility" in this context is not a building. As on most projects, specialized terminology from the domain experts entered our vocabulary and became part of the UBIQUITOUS LANGUAGE. In the domain of commercial banking, **a facility is a commitment by a company to lend**. Your credit card is a facility that entitles you to borrow on demand up to a prearranged limit at a predetermined interest rate. When you use the card, you create an outstanding loan, and each additional charge is a "**draw-down**" against your "facility" that increases the loan. Finally you pay back the loan principle. You may also pay an annual fee. This is a fee for the privilege of having the card (the facility), and is independent of your loan.

When Intel wants to build a billion dollar factory, they need a loan that is too big for any single lending company to take on, so the lenders form a "**syndicate**" that pools its resources to support a facility. An investment bank usually acts as syndicate leader, coordinating transactions and other services.

Given that four months before we had been in deep trouble with a completely unworkable, inherited design, we were feeling pretty good, but there were disconcerting signs. We kept stumbling over unexpected requirements that complicated the design. A major example was the creeping understanding that the shares in a **Facility** (see sidebar for definition) were only a *guideline* to participation in any particular loan draw-down. When the borrower requests its money, the leader of the syndicate calls all members for their shares.

The model above makes the common case very simple. The **Loan Investment** is a derived object that represents a particular investor's contribution to the **Loan**. When called, the investors usually cough up their share, but often they negotiate with other members of the syndicate and invest less (or more). We had accommodated this by adding **Loan Adjustments** to the model.

Model Incrementally Changed to Solve Problems

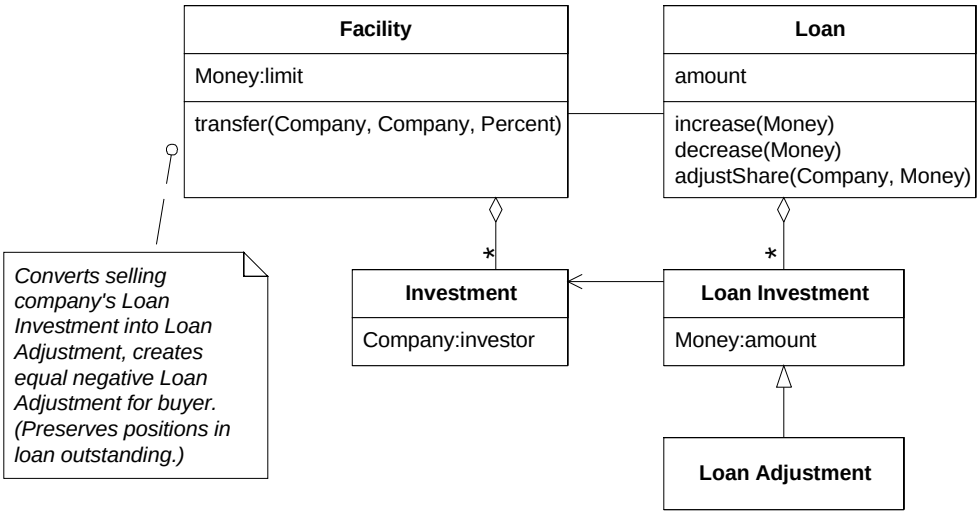


Figure 8. 2

Refinements of this kind allowed us to keep up as the rules of various transactions became clearer. But complexity was increasing and we did not seem to be converging quickly onto really solid functionality.

Even more troubling were subtle rounding inconsistencies that we had not been able to squash with increasingly complex algorithms. True, in a $100 MM (million) deal no one cares about where the extra pennies go, but they don't trust software that cannot meticulously account for it. We began to suspect that our difficulties in doing this were symptomatic of a basic design problem.

The Breakthrough

Suddenly one week it dawned on us what was wrong. Our model tied the facility and loan shares in a way that was *not appropriate to the business*. This had wide repercussions. With the business experts nodding, enthusiastically helping – and, I dare say, wondering what took us so long – we hashed out a new model on a white board. We walked through numerous scenarios using the new model, something like this:

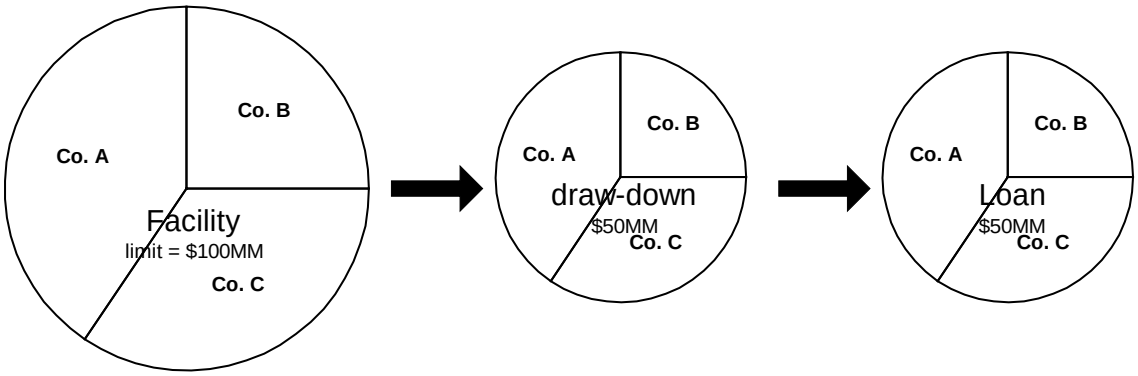


Figure 8. 3

This diagram says that the borrower has chosen to draw an initial $50MM from the $100MM committed under the Facility. The three lenders chip in their shares in exact proportion to the Facility shares, resulting in a $50MM Loan divided among the lenders.

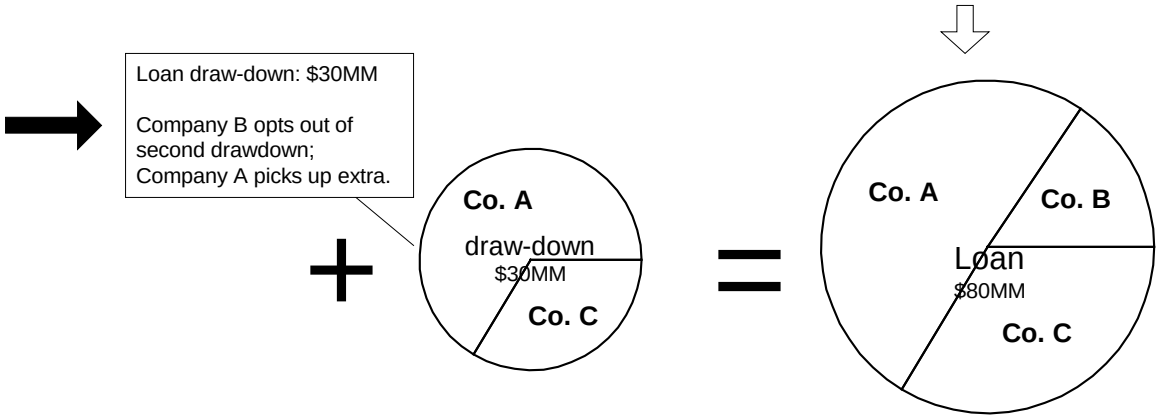


Figure 8. 4

Next, the borrower draws an additional $30MM, bringing his outstanding Loan to $80MM, still under the $100MM limit of the Facility. This time, Company B chooses not to participate, letting Company A take an extra share. The Shares of the Draw-down reflect these investment choices. When the Draw-down amounts are added to the Loan, the shares of the Loan are no longer proportional to the shares of the Facility. This is common.

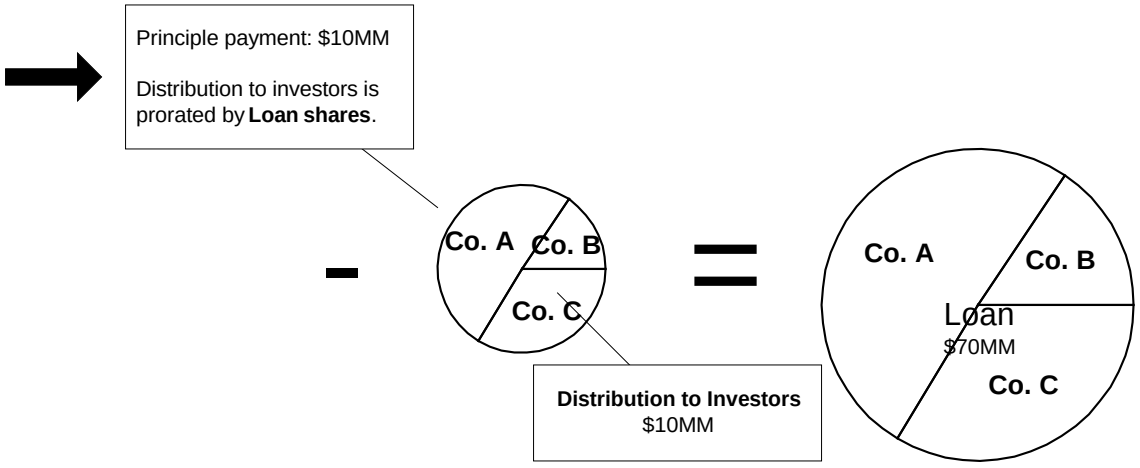


Figure 8. 5

When the borrower pays down the Loan, the money is divided among the lenders according to the shares of the Loan, not the Facility. Likewise, interest payments will be divided according to the Loan shares.

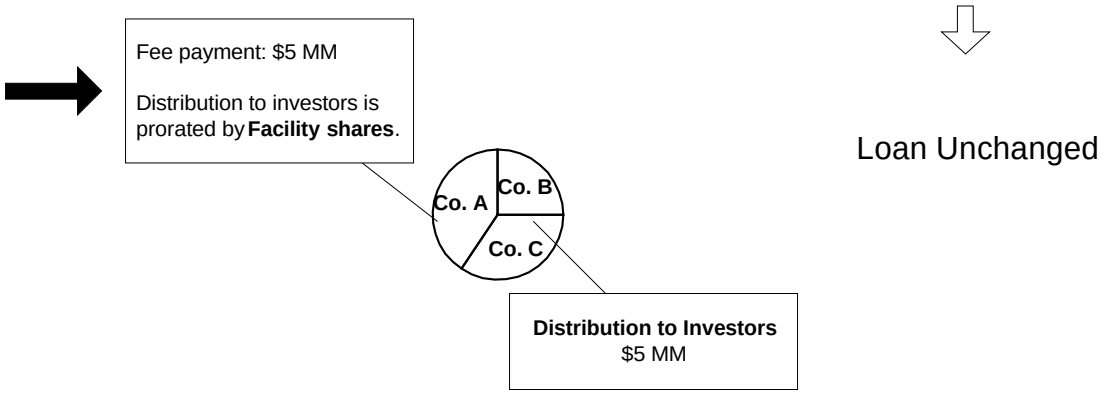


Figure 8. 6

On the other hand, when the borrower pays a fee for the privilege of having the Facility available, this money is divided according to the Facility shares, regardless of who actually has lent money. The Loan is unchanged by fee payments. There are even scenarios in which lenders trade shares of fees separately from their shares of the Facility, and so on.

A Deeper Model

We had two deep insights. First was the realization that our "Investments" and "Loan Investments" were just two special cases of a general and fundamental concept: shares. Shares of a facility, shares of a loan, shares of a *payment distribution*. Shares, shares everywhere. Shares of any divisible value.

A few tumultuous days later I had sketched a model of shares, drawing on the language used in the discussions with experts and the scenarios we had explored together.

An Abstract Model of Shares

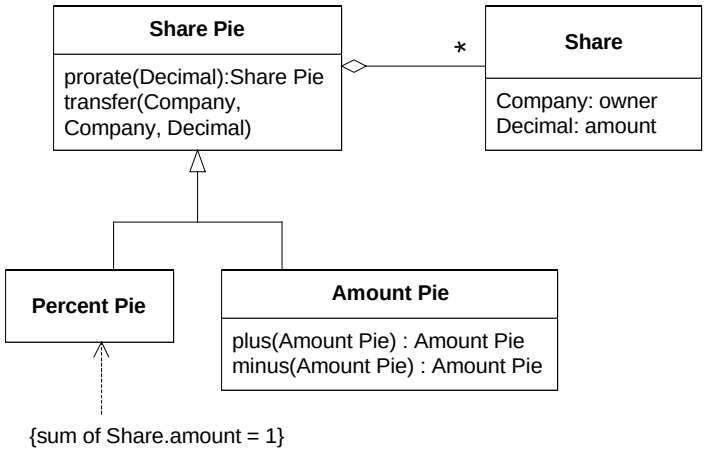


Figure 8. 7

I also sketched a new loan model to go with it.

Breakthrough Model Using Share Pie

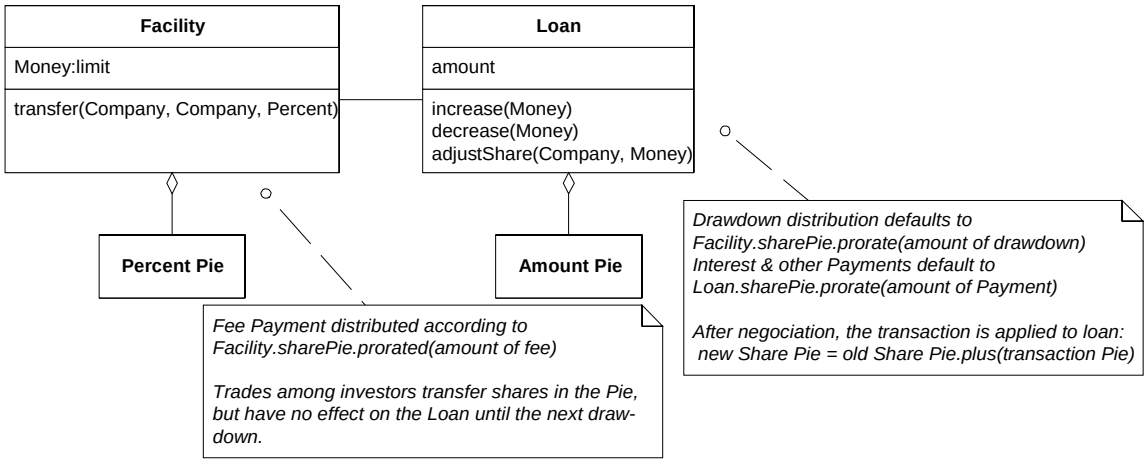


Figure 8. 8

There were no longer specialized objects for the shares of a **Facility** or a **Loan**. They are both broken down into the more intuitive "**Share Pie**". This generalization allowed the introduction of "shares math", vastly simplifying the calculation of shares in any transaction, and making those calculations more expressive, concise and easily combined. The **Loan Investment** had disappeared, and at this point we realized that "loan investment" was not a banking term. In fact, the business experts had told us a number of times they didn't understand it. They had deferred to our software knowledge, and assumed it was useful to the technical design. Actually, we had created it based on our incomplete understanding of the domain.

Even more important, this model did not directly tie the **Loan** shares and **Facility** shares together. The **Share Pie** of the **Loan** could be adjusted directly, so the **Loan Adjustment** was not needed and a large amount of special-case logic was eliminated.

Suddenly we could run through every scenario we had ever encountered relatively effortlessly, much more simply than ever before. *And our model diagrams made perfect sense to the business experts, who had often indicated that they were "too technical" for them.* I have since come to consider such disclaimers a warning that the model isn't right or isn't well expressed. Even sketching on the whiteboard we could see that some of our most persistent rounding problems would be pulled out by the roots, allowing us to scrap some of the complicated rounding code.

Our new model worked well. Really, really well.

And we all felt sick!

A Sobering Decision

You might reasonably assume that we would have been elated at this point. We were not. We were under a severe deadline; the project was already dangerously behind schedule. Our dominant emotion was fear.

The gospel of refactoring is that you always go in small steps, always keeping everything working. But to refactor our code to this new model would require changing a lot of supporting code, and there would be few, if any, stable stopping points in between. We could see some small improvements we could make, but none that would take us closer to the new concept. We could see a sequence of small steps to get there, but parts of the application would be disabled along the way. And this was before the age when automated

tests were widely used on such projects. We had none, so there was bound to be unforeseen breakage.

And it was going to take effort. We were already exhausted from months of pushing.

At this point we had a meeting with our project manager that I will never forget. Our manager was an intelligent and bold man. He asked a series of questions.

Q: How long would it take to get back to current functionality with the new design?

A: About 3 weeks.

Q: Could we solve the problems without it?

A: Probably. But no way to be sure.

Q: Would we be able to move forward in the next release if we didn't do it now?

A: Forward movement would be slow without the change. And the change would be much harder once we had an installed base.

Q: Did *we* think it was the right thing to do?

A: We knew the political situation was unstable, so we'd cope if we had to. And we were tired. But, yes, it was a simpler solution that fit the business much better. In the long run it was lower risk.

He gave us the go-ahead and told us he would handle the heat. I've always had a tremendous admiration for the courage and trust it took for him to make this decision.

We busted our butts and got it done in three weeks. It was a big job, but it went surprisingly smoothly.

The Payoff

The mystifyingly unexpected requirement changes stopped. The rounding logic, though never exactly simple, stabilized and made sense. We delivered version one and the way was clear to version two. My nervous breakdown was narrowly averted.

As version two evolved, this **Share Pie** became the unifying theme of the whole application. Technical people and business experts used it to discuss the system. *Marketing people used it to explain the features to prospective customers.* Those prospects and customers immediately grasped it and used it to discuss features. It truly became part of the UBIQUITOUS LANGUAGE because it got to the heart of what loan syndication is about.

Opportunities

When the prospect of a breakthrough to a deeper model presents itself, it is often scary. Such a change has higher opportunity *and* higher risk than most refactorings. And timing may be inopportune.

Much as we might like it to be otherwise, progress isn't a smooth ride. The transition to a really deep model is a profound shift in your thinking and demands a major change to the design. On many projects the most important progress in model and design come in these breakthroughs.

Focus on Basics

Don't become paralyzed trying to bring about a breakthrough. The possibility usually comes after many modest refactorings. Most of the time is spent making piecemeal improvements, model insights emerging gradually with each successive refinement.

To set the stage for a breakthrough, concentrate on knowledge crunching and cultivating a robust UBIQUITOUS LANGUAGE. Probe for important domain concepts and make them explicit in the model

(Chapter 9). Refine the design to be suppler (Chapter 10). Distill the model (Chapter 15). Push on these more predictable levers which increase clarity, usually a precursor of breakthroughs.

Don't hold back from modest improvements, which gradually deepen the model, even if confined within the same general conceptual framework. Don't be paralyzed by looking too far forward. Just be watchful for the opportunity.

Epilogue: A Cascade of New Insights

That breakthrough got us out of the woods, but it was not the end of the story. The deeper model opened unexpected opportunities to make the application richer and the design clearer.

Just weeks after the release of the **Share Pie** version of the software, we noticed another awkward aspect of the model that was complicating the design. An important ENTITY was missing, its absence filled by extra responsibilities taken up by other objects. Specifically, there were significant rules governing loan draw-downs, fee payments, etc, and all this logic was crammed into various methods on the **Facility** and **Loan**. These design problems, which had been barely noticeable before the **Share Pie** breakthrough, became obvious with our clearer field of vision. Now we noticed terms popping up in our discussions that were nowhere to be found in the model, terms like "transaction" (meaning financial transaction), that we started to realize were being implied by all those complicated methods.

Following a process similar to the one described above (although, thankfully, under much less time pressure) led to yet another round of insights and a still deeper model. This new model made these implicit concepts explicit, as **Transactions**, and at the same time simplified the **Positions** (an abstraction including the **Facility** and **Loan**). It became easy to define the diverse transaction types we had, along with their rules, negotiating procedures and approval processes, and all in relatively self-explanatory code.

Another Model Breakthrough That Followed Several Weeks Later

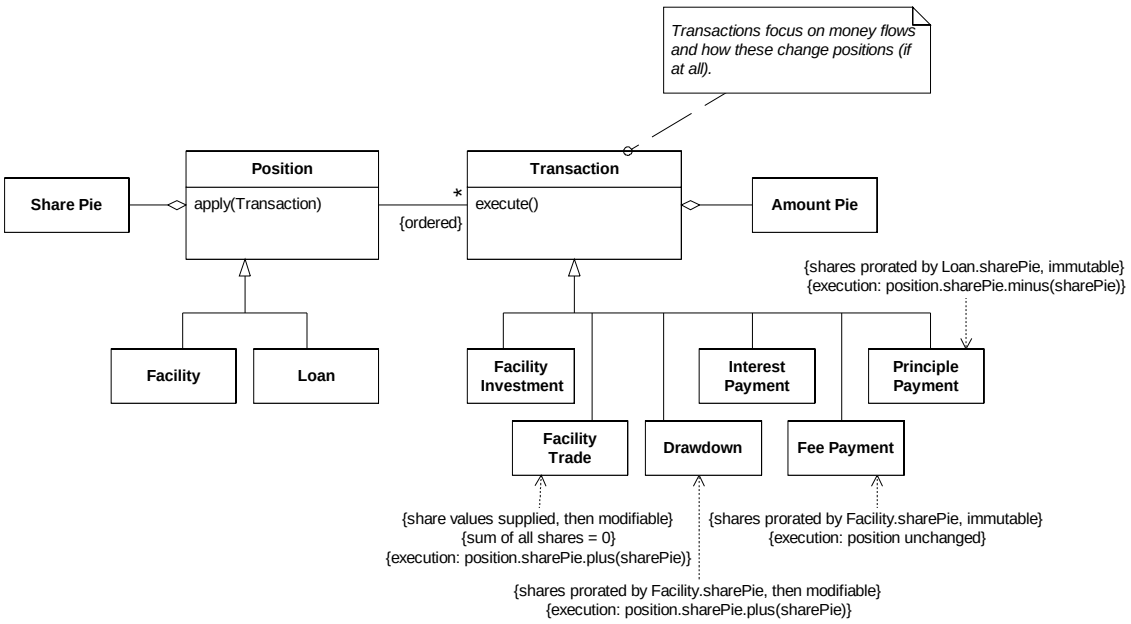


Figure 8. 9

(**The constraints in the diagram illustrate the easy precision gained with the new model.**)

As is commonly the case after a real breakthrough to a deep model, the clarity (and typically simplicity) of the new design, combined with the enhanced communication based on the new UBIQUITOUS LANGUAGE had led to yet another modeling breakthrough.

Our pace of development was accelerating at a stage where most projects are beginning to be bogged down by the mass and complexity of what they have already built.

9. Making Implicit Concepts Explicit

Deep modeling sounds great, but how do you actually do it? A deep model has power because it contains the central concepts and abstractions that can succinctly and flexibly express essential knowledge of activities of the users, their problems, and their solutions. The first step toward this is to have the essential concepts of the domain represented in the model in some form. Refinement comes later, after successive iterations of knowledge crunching and refactoring. But this process really gets into gear when an important concept is recognized and made explicit in the model and design.

Many transformations of domain models and the corresponding code take the form of recognizing a concept that has been hinted at, present implicitly, and representing it with one or more explicit objects or relationships in the model.

Occasionally, this change of making a formerly implicit concept explicit is a breakthrough that leads to a deep model. More often, though, the breakthrough comes later, after a number of important concepts are explicit in the model, after successive refactorings have tweaked their responsibilities repeatedly, changed their relationships with other objects, even changed their names a few times. It finally snaps into focus. But it starts with recognizing the implied concepts in however crude a form.

Such recognition comes from listening to the language of the team, scrutinizing awkwardness in the design and seeming contradictions in the statements of experts, mining the literature of the domain, and lots and lots of experimentation.

Listen to Language

You may remember an experience like this: The users have always talked about some item on a report. The item is compiled from a set of data your application collects in various other situations as well. Attributes of different objects or even direct database queries. The data are assembled in order to present or report or derive something. But you have never seen the need for an object. Probably, you have never really understood what the users meant by a term, or have not realized it was important.

Then suddenly a light comes on in your head. You excitedly talk with your experts about your new insight. Maybe they show relief that you finally got it. Maybe they yawn because they've taken it for granted all along. Either way, you start to draw model diagrams on the board that fill in for some hand-waving that you've always done before. The users correct you on the details of how the new model connects, but you can tell that there is a change in quality of the discussion. You and the users are understanding each other more precisely, and demonstrations of model interactions to solve specific scenarios have become more natural. The language of the domain model has become more powerful. You refactor the code to reflect the new model and find you have a cleaner design.

Listen to the language the domain experts use. Are there terms that succinctly state something complicated? Are they correcting your word choice (perhaps diplomatically)? Does the puzzled look on their face go away with the use of a particular phrase? These are hints of a concept that might benefit the model.

This is *not* the old "nouns are objects" notion. Hearing a new word just produces a lead to be followed up with conversation and knowledge crunching with the goal of carving out a clean, useful concept. When the users or domain experts use vocabulary that is nowhere in the design, that is a warning sign. It is a doubly strong warning when the developers and the domain experts are both using terms that are not in the design.

Or perhaps it is better to look at it as an opportunity. The UBIQUITOUS LANGUAGE is made up of the vocabulary that pervades speech, documents, model diagrams, and even code. If a term is absent from the design, it is an opportunity to improve the model and design by including it.

Example: Hearing a Missing Concept in the Shipping Model

The team had already developed a working application that could book a cargo. They were starting to build an "operations support" application that would help juggle the work-orders

for loading and unloading of cargos at the origin and destination and at transfers between ships.

The booking application used a routing engine to plan the trip for a cargo. Each leg of the journey was stored in a row of a database table, indicating the vessel voyage ID (a unique identifier for a particular voyage by a particular ship) which a cargo would be carried on, the location where it would be loaded, and the location where it would be unloaded.

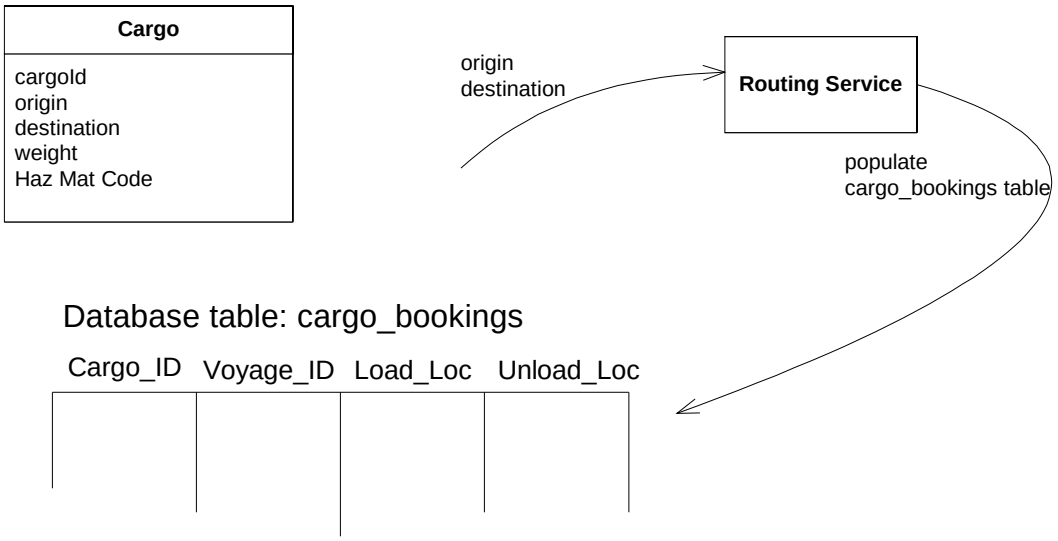


Figure 9. 1

Let's eavesdrop on a conversation (heavily abbreviated) between the developer (**DEV**) and a shipping expert (**EXP**).

DEV: I want to make sure the "cargo bookings" table has all the data that the operations application will need.

EXP: They're going to need the whole itinerary for the cargo. What information does it have now?

DEV: The cargo Id, the vessel voyage, the loading port, and the unloading port for each leg.

EXP: What about the date? Operations will need to contract handling work based on the expected times.

DEV: Well, that can be derived from the schedule of the vessel voyage. The table data is normalized.

EXP: Yes, it is normal to need the date. Operations people use these kinds of itineraries to plan for upcoming handling work.

DEV: Yeah... Ok, they'll definitely have access to the dates. The operations management application will be able to provide the whole loading and unloading sequence, with the date of each handling operation. The "itinerary", I guess you would say.

EXP: Good. The itinerary is the main thing they'll need. Actually, you know, the booking application has a menu item that will print an itinerary or email it to the customer. Can you use that somehow?

DEV: That's just a report, I think. We won't be able to base the operations application on that.

[**DEV** looks thoughtful, then excited.]

DEV: So this itinerary is really the link between booking and operations.

EXP: Yes, and some customer relations, too.

DEV: [Sketching a diagram on the whiteboard.] So would you say it is something like this?

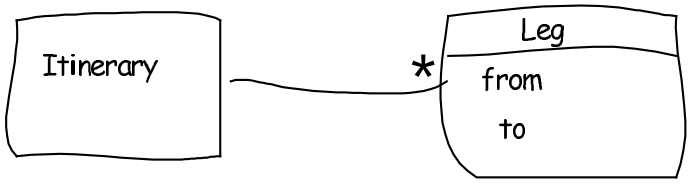


Figure 9. 2

EXP: Yes, that looks basically right. For each leg you'd like to see the vessel voyage, the load and unload location and time.

DEV: So once we create this object, it can derive the times from the vessel voyage schedule. We can make this object our main point of contact with the operations application. And we can rewrite that itinerary report to use this, so we'll get the domain logic back into the domain layer.

EXP: I didn't follow all of that, but you are right that the two main uses for the itinerary are in the report in booking and in the operations application.

DEV: Hey! We can make the routing service interface return an itinerary object instead of putting the data in the database table. That way the routing engine doesn't need to know about our tables.

EXP: Huh?

DEV: I mean, I'll make the routing engine just return an itinerary. Then it can be saved in the database by the booking application when the rest of the booking is saved.

EXP: You mean it isn't that way now?!

At this point, the developer went off to talk with the other developers involved in the routing process. They hashed out the details of the changes to the model and the implications for the design, calling on the shipping experts when needed. They came up with the following.

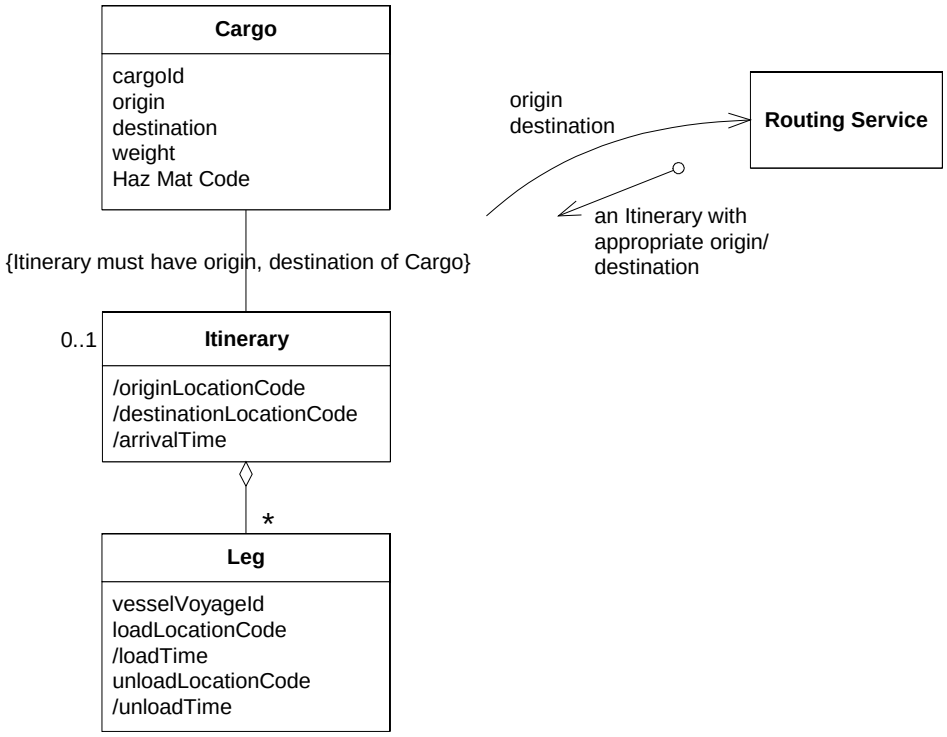


Figure 9. 3

Next, the developers refactored the code to reflect the new model. They did it in a series of two or three refactorings, but in quick succession, within a week, except for simplifying the itinerary report in the booking application, which they took care of early the following week.

The developer had been listening closely enough to the shipping expert to notice how important the concept of an "itinerary" was to him. True, all the data was already being collected, and the behavior was implicit in the itinerary report, but the explicit **Itinerary** as part of the model opened up opportunities.

Benefits of refactoring to the explicit **Itinerary** object:

1. A more expressive definition of the interface of the **Routing Service**.
2. Decoupling of the **Routing Service** from the booking database tables.
3. A clear relationship between the booking application and the operations support application (the sharing of the **Itinerary** object).
4. Reduction of duplication, as the **Itinerary** derives loading/unloading times for both the booking report and the operations support application.
5. Removal of domain logic from the booking report into the isolated domain layer.
6. Expansion of the UBIQUITOUS LANGUAGE, allowing a more precise discussion of the model and design between developers and domain experts and among the developers themselves.

Scrutinize Awkwardness

The concept you need is not always floating on the surface, emerging in conversation or documents. You may have to dig and invent. The place to dig is the most awkward part of your design. The place where

procedures are doing complicated things that are hard to explain. The place where every new requirement seems to add complexity.

Sometimes it can be hard to recognize that there even is a missing concept. You may have objects doing all the work, but find some of the responsibilities awkward. Or, if you do realize something is missing, a model solution may elude you.

Now you have to actively engage the domain experts in the search. If you are lucky, they may enjoy playing with ideas and experimenting with the model. If you are not that lucky, you and your fellow developers will have to come up with the ideas, using the domain expert as a validator, watching for discomfort or recognition on his face.

Example: Earning Interest a Hard Way

The next story takes place in a hypothetical financial company that invests in commercial loans and other interest-bearing assets. An application has been evolving incrementally, feature by feature that tracks those investments and the earnings from them. Each night, a batch script would run that would cause all interest and fees earned to be calculated and then recorded appropriately in the company's accounting software.

The Awkward Model

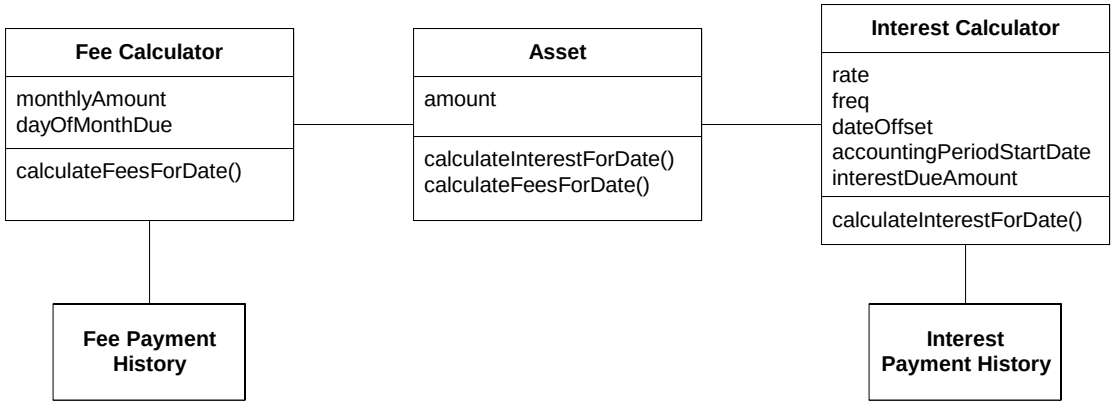


Figure 9. 4

In terms of the model, the nightly batch script iterated through each **Asset**, telling each to `calculateInterestForDate()` on that day's date. The script took the return value (the amount earned) and passed this amount, along with the name of a specific ledger, to a SERVICE that provided public interface of the accounting program. That software posted the amount to the named ledger. The script went through a similar process to get the day's fees from each ASSET, posting them to a different ledger.

A developer had been struggling with the increasing complexity of calculating interest. She started to suspect an opportunity for a model better suited to the task. This developer(**DEV**) asked her favorite domain expert (**EXP**) to help her dig into the problem area.

DEV: Our **Interest Calculator** is getting out of hand.

EXP: That is a complicated part. We still have more cases we've been holding off with.

DEV: I know. We can add new interest types by substituting a different **Interest Calculator.** But what we're having the most trouble with right now is all these special cases when they don't pay the interest on schedule.

EXP: Those really aren't special cases. There's a lot of flexibility in when people pay.

DEV: Back when we factored out the **Interest Calculator** from the **Asset**, it helped a lot. We may need to break it up more.

EXP: Ok.

DEV: I was thinking you might have a way of talking about this interest calculation.

EXP: What do you mean?

DEV: Well, for example, we're tracking the interest due but unpaid within an accounting period. Do you have a name for that?

EXP: Well, we don't really do it like that. The interest earned and the payment are quite separate postings.

DEV: So you don't need that number?

EXP: Well, sometimes we might look at it, but it isn't the way we do business.

DEV: Ok, so if the payment and interest are separate, maybe we should model them that way. How does this look? [Sketching on whiteboard]

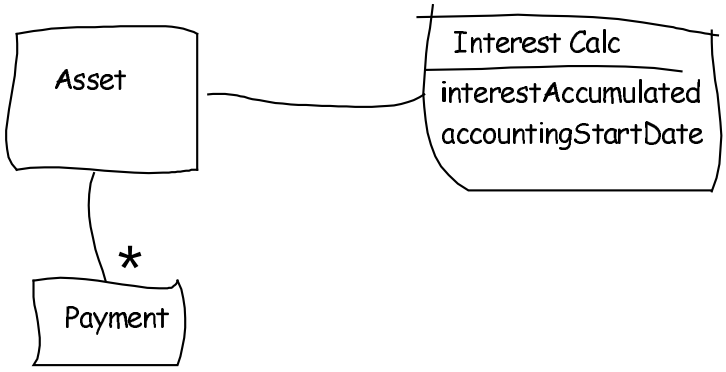


Figure 9. 5

EXP: It makes sense, I guess. But you just moved it from one place to another.

DEV: Except now the **Interest Calculator** only keeps track of interest earned, and the **Payment** keeps that number separately. It hasn't simplified it a lot, but does it better reflect your business practice?

EXP: Ah. I see. Could we have interest history, too? Like the **Payment History**.

DEV: Yes, that has been requested as a new feature. But that could have been added onto the original design.

EXP: Oh. Well, when I saw interest and **Payment History** separated like that, I thought you were breaking up the interest to organize it more like the **Payment History**.

Do you know anything about accrual basis accounting?

DEV: Please explain?

EXP: Each day, or whenever the schedule calls for, we have an interest accrual that gets posted to a ledger. The payments are posted a different way. This aggregate you have here is a little awkward.

DEV: You're saying that if we keep a list of accruals, they could be aggregated or…"posted" as needed.

EXP: Probably posted on the accrual date, but yes, aggregated any time. Fees work the same way, posted to a different ledger, of course.

DEV: Actually, the interest calculation would be simpler if it was done just for one day, or period. And then we could just hang on to them all. How about this?

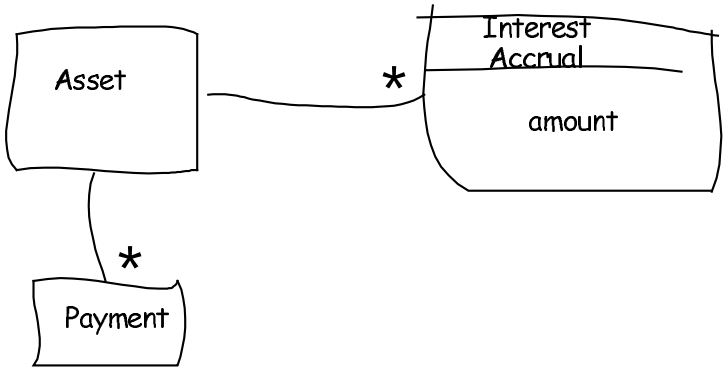


Figure 9. 6

EXP: Sure. It looks good. I'm not sure why this would be easier for you. But basically, what makes any asset valuable is what it can accrue in interest, fees, etc.

DEV: You said fees work the same way? They… what was it…post to different ledgers?

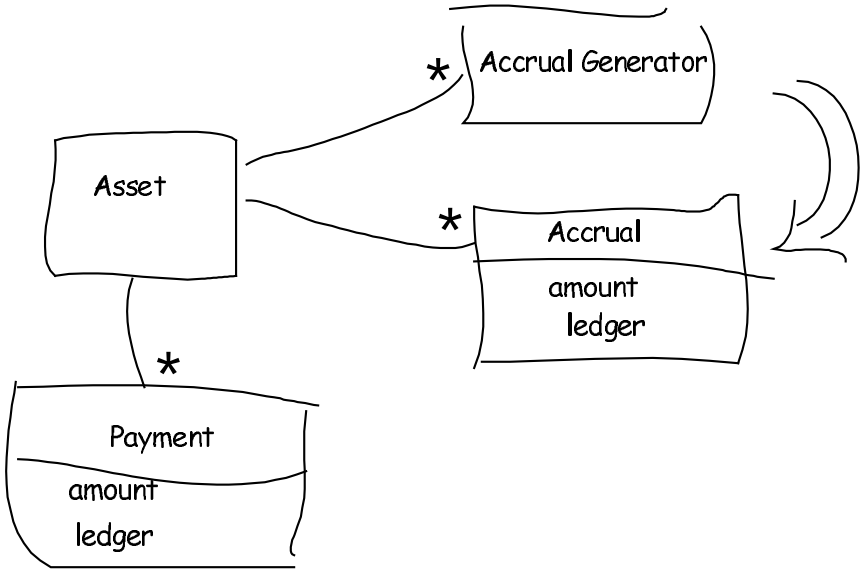


Figure 9. 7

DEV: With this model, we get the interest calculation, or rather, the accrual calculation logic that was in the Interest Calculator separated from tracking. And I hadn't noticed until now how much duplication there is in the Fee Calculator. Also, now the different kinds of fees can easily be added.

EXP: Yes, the calculation was correct before, but I can see everything now.

Since the Calculator classes hadn't been directly coupled with other parts of the design, this was a fairly easy refactoring. The developer was able to rewrite the unit tests to use the new language in a few hours and had the new design working late the next day. She ended up with this.

A Deeper Model After Refactoring

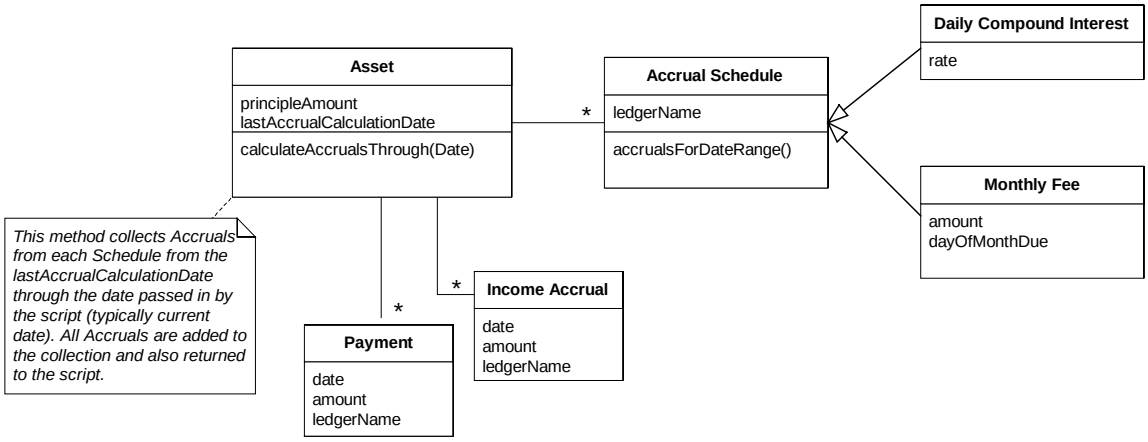


Figure 9. 8

In the refactored application, the nightly batch script tells each **Asset** to `calculateAccrualsThroughDate()`. The return value is a collection of **Accruals**, each of whose amounts it posts to the indicated ledger.

The new model has these advantages:

1. Enriches the UBIQUITOUS LANGUAGE with the term “accrual”.
2. Decouples accrual from payment.
3. Removes domain knowledge (e.g., which ledger to post to) from the script and into the domain layer.
4. Brings fees and interest together in a way that fits the business and eliminates duplication in the code.
5. Provides a straight-forward path for adding new variations of fees and interest as Accrual Schedules.

This time, the developer had to dig for the new concepts she needed. She could see the awkwardness of the interest calculations and made a committed effort to looking for a deeper answer.

She was lucky to have an intelligent and motivated partner in the banking expert. With a more passive source of expertise, she would have made more false starts and depended more on other developers as brainstorming partners. Progress would have been slower, but still possible.

Contemplate Contradictions

Different domain experts see things different ways based on their experience and needs. Even the same person provides information that is logically inconsistent after careful analysis. These pesky contradictions we encounter all the time when digging into program requirements can be great clues to deeper models. Some are just variations in terminology, or are based on misunderstanding. But there is a residue where two factual statements by experts seem to contradict.

The astronomer Galileo once posed a paradox. The evidence of the senses clearly indicates that the Earth is stationary -- people are not being blown off and falling behind. Yet Copernicus had made a compelling argument that the Earth was moving around the sun quite rapidly. Reconciling this might reveal something profound about how nature works.

Galileo devised a thought experiment. If a rider dropped a ball from a running horse, where would it fall? Of course, the ball would move along with the horse until it hit the ground by the horse's feet, just as if the horse were standing still. From this he deduced an early form of the idea of inertial frames of reference, solving the paradox and leading to a much more useful model of the physics of motion.

Ok. Our contradictions are usually not so interesting, nor the implications so profound. Even so, this same pattern of thought often helps pierce through the superficial layers of a problem domain to a deeper insight.

It is not practical to reconcile all contradictions, and it may not even be desirable. (Chapter 14 delves into how to decide and how to manage the result.) However, even when a contradiction is left in place, contemplation of how two statements could both apply to the same external reality can be revealing.

Read the Book

Don't overlook the obvious when seeking model concepts. In many fields, you can find books that explain the fundamental concepts and conventional wisdom. You still have to work with your own domain experts to distill the part relevant to your problem and to crunch it into something suited to object-oriented software. But you may be able to start with a coherent, deeply thought-out view.

Example: Earning Interest Revisited

Let's imagine a different scenario for the investment tracking application discussed in the previous example. Just as before, the story starts with the developer realizing that the design is getting unwieldy, particularly the **Interest Calculator**. But in this scenario, the domain expert's primary responsibilities lie elsewhere, and he doesn't have much interest in helping the software development project. In this scenario, the developer couldn't turn to the expert for a brainstorming session to probe for the missing concepts she suspected to be lurking under the surface.

Instead, she went to the bookstore. After a little browsing, she found an introductory accounting book she liked and skimmed it. She discovered a whole system of well defined concepts. An excerpt that particularly fired her thinking:

> **Accrual Basis Accounting.** This method recognizes income when it is earned, even if it is not paid. *All* expenses also show when they are incurred whether they have been paid for or billed to be paid at a later date. Any obligation due, including taxes, will be shown as expense.
>
> *Finance and Accounting: How to Keep Your Books and Manage Your Finances Without an MBA, a CPA or a Ph.D.*, Suzanne Caplan, Streetwise, 2000.

She no longer needed to reinvent accounting. After some brainstorming with another developer, she came up with a model.

A Somewhat Deeper Model Based on Book Learning

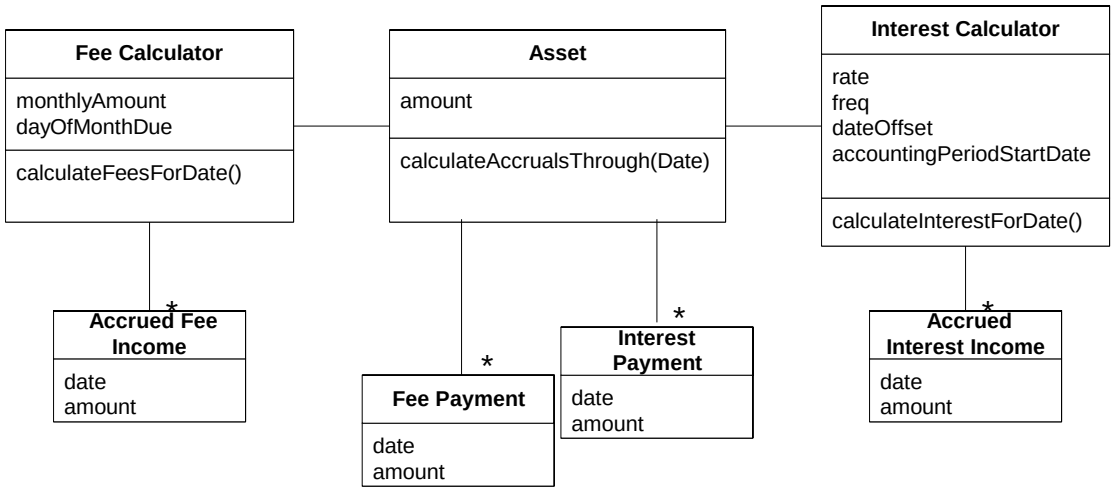


Figure 9. 9

She did not have the insight that **Assets** are income generators, and so the "**Calculators**" are still there. The knowledge of ledgers is still in the application, rather than domain layer where it probably belongs. But she did separate the issue of payment from the accrual of income, which was the primary source of complexity, and she introduced the word "accrual" into the model and the UBIQUITOUS LANGUAGE. Further refinement could come with later iterations.

When she did finally have the chance to talk with the domain expert, he was quite surprised. It was the first time a programmer had shown a glimmer of interest in what he did. Due to the way his responsibilities were assigned, he never engaged with her as happened in the other scenario. However, her knowledge allowed her to ask better questions, and he did make a special effort to answer her questions promptly in the future, and he listened to her more carefully.

Of course, this is not an either-or proposition. Even with ample support from domain experts, it pays to look at the literature to get a grasp of the theory of the field. Most businesses do not have models refined to the level of accounting or finance, but in many there have been thinkers in the field who have organized and abstracted the common practices of the business.

Yet another option she had was to read something like Chapter 6 in *Analysis Patterns: Reusable Object Models* [Fowler 1997]. This would have sent her in quite a different direction, not necessarily better or worse. These models would not have given her an off-the-shelf solution, they would have given her several new starting points for her own experiments, along with the distilled experience of people who have written this kind of software before. She would have been spared reinventing the wheel. Chapter 11, "Applying Analysis Patterns" will delve into this option more.

Try, Try Again

These examples don't convey the amount of trial and error involved. I might follow half a dozen leads in conversation before finding one that seems clear and useful enough to try out in the model. I'll end up

replacing that one at least once later, as additional experience and knowledge crunching serve up better ideas. A modeler/designer cannot afford to get attached to his own ideas.

All these changes of direction are not just thrashing. Each change embeds deeper insight into the model. Each refactoring leaves the design more supple, easier to change the next time, ready to bend in the places that turn out to need to bend.

There really is no choice anyway. Experimentation is the way to learn what works and doesn't. Trying to avoid missteps in design will result in a lower quality result because it will be based on less experience. And it can easily take longer than a series of quick experiments.

Expressing Less Obvious Categories of Concepts

The object-oriented paradigm leads us to look for and invent certain kinds of concepts. Things, even very abstract ones like "accruals", are the meat of most object models, along with the actions those things take. These are the "nouns and verbs" that introductory object-oriented design books talk about. But other important categories of concepts can be made explicit in a model as well.

I'll discuss three such categories which were not obvious to me when I started with objects. My designs became sharper with each one of these I learned.

Explicit Constraints

Constraints make up a particularly important category of model concepts. They often emerge implicitly, and expressing them explicitly can greatly improve a design.

Sometimes constraints find a natural home in an object or method. A "Bucket" object must guarantee the invariant that it does not hold more than its capacity.

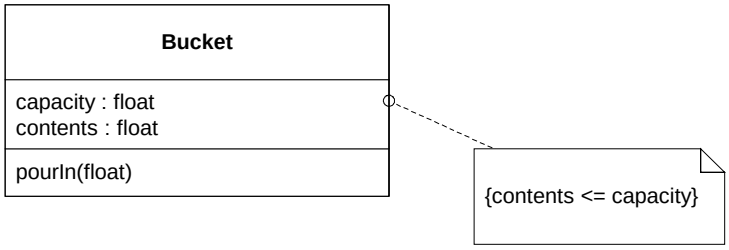


Figure 9. 10

A simple invariant like this can be enforced using case logic in each operation capable of changing contents.

```
class Bucket  {
   private float capacity;
   private float contents;

   public void pourIn(float addedVolume) {
      if (contents + addedVolume > capacity) {
         contents = capacity;
      } else {
         contents = contents + addedVolume;
      }
   }
}
```

This is so simple that the rule is obvious. But you can easily imagine that this constraint could get lost in a more complicated class. Let's factor it into a separate method, with a name that clearly and explicitly expresses the significance of the constraint.

```
class Bucket  {
   private float capacity;
   private float contents;

   public void pourIn(float addedVolume) {
      float volumePresent = contents + addedVolume;
      contents = constrainedToCapacity(volumePresent);
   }

   private float constrainedToCapacity(float volumePlacedIn) {
      if (volumePlacedIn > capacity) return capacity;
      return volumePlacedIn;
   }
}
```

Both versions of this code enforce the constraint, but the second makes a more obvious relationship to the model (the basic requirement of MODEL-DRIVEN DESIGN).This very simple rule was understandable in its original form, but when the rules being enforced are more complex, they start to overwhelm the object or operation they apply to, as any implicit concept does. Factoring the constraint into its own method allows us to give it an intention revealing name that makes the constraint explicit in our design. It is now a named thing we can discuss. It also gives the constraint room. A more complex rule than this might easily produce a longer method than its caller. This way, the caller stays simple and focused on its task while the constraint can grow in complexity if need be.

This separate method gives the constraint some room, but there are lots of cases when a constraint just can't fit comfortably in a single method. Or, even if the method stays simple, it may call on information that the object doesn't need for its primary responsibility. The rule may just have no good home in an existing object. Here are some warning signs that a constraint is distorting the design of its host object:

1. Evaluating a constraint requires data that does not otherwise fit the object's definition.
2. Related rules appear in multiple objects, forcing duplication or inheritance between objects that are not otherwise a family.
3. A lot of design and requirements conversation revolves around the constraints, but in the implementation, they are hidden away in procedural code.

When the constraints are obscuring the object's basic responsibility, or when the constraint is prominent in the domain yet not prominent in the model, you can factor it out into an explicit object or even model it as a set of objects and relationships. (One in-depth semi-formal treatment of this can be found in *The Object Constraint Language: Precise Modeling with UML* [WK1999].)

Review Example: Overbooking Policy

In the example in Chapter 1, p. ##, a shipping company had a business practice of booking 10% more cargo than the transports could handle. (Experience has taught them that this compensates for last-minute cancellations, so their ships sail nearly full.)

This constraint on the association between Voyage and Cargo was made explicit, both in the diagrams and in the code, by adding a new class that represented the constraint.

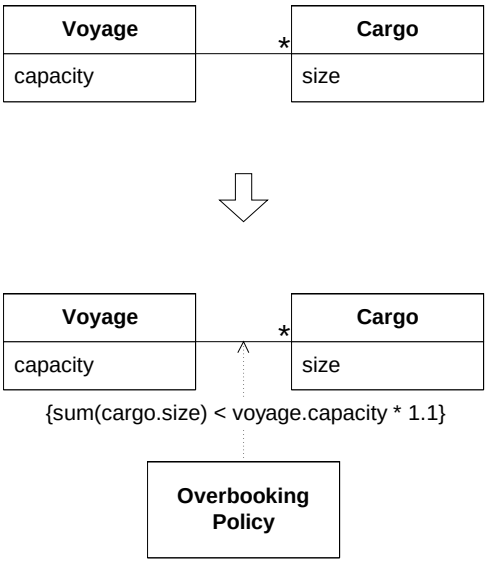


Figure 9. 11

To review the code and reasoning, see p. ##.

Representing Processes as Domain Objects

Right up front, let's agree that we do *not* want to make procedures a prominent aspect of our model. Objects are meant to encapsulate the procedures and let us think about their goals or intentions instead.

What I am talking about here are processes that exist in the domain, which we have to represent in the model. When these emerge, they tend to make for awkwardness in object designs.

Back in Chapter 5, the example of a domain SERVICE described a shipping system that routed cargo. This routing process was something the business did. A SERVICE is one way of expressing such a process explicitly, while still encapsulating the extremely complex algorithms.

When there is more than one way to carry out a process, another approach is to make the algorithm itself, or some key part of it, an object in its own right. The choice between processes becomes a choice between these objects, each of which represents a different STRATEGY. (Chapter 12 will look in more detail at the use of STRATEGIES in the domain.)

The key to distinguishing a process that aught to be made explicit from one that should be hidden is simple: Is this something the domain experts talk about, or is it just part of the mechanism of the computer program?

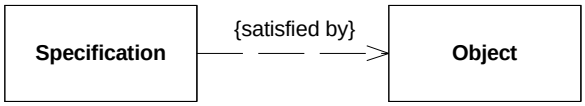
♦ ♦ ♦

Constraints and processes are two broad categories of model concepts that don't come leaping to mind when programming in an object-oriented language, yet can really sharpen up a design once we start thinking about them as model elements.

Some useful categories of concepts are much narrower. The next one is much more specific, yet quite common. SPECIFICATION provides a concise way of expressing certain kinds of rules, extricating them from conditional logic and making them explicit in the model.

I developed SPECIFICATION in collaboration with Martin Fowler [EF1997]. The simplicity of the concept belies the subtlety in application and implementation, so there is a lot of detail in this section. There will be even more discussion in Chapter 10, where it is extended. Beyond the initial statement of the pattern, you may want to skim, until such time as you are actually attempting to apply the pattern.

SPECIFICATION



In all kinds of applications, Boolean test methods appear that are really parts of little rules. As long as they are simple, we handle them with testing methods, such as `anIterator.hasNext()` or `anInvoice.isOverdue()`. The code in `isOverdue()` is an algorithm that evaluates a rule. For example,

```
public boolean isOverdue() {
   Date currentDate = new Date();
   return currentDate.after(dueDate);
}
```

But not all rules are so simple. On the same **Invoice** class, another rule, `anInvoice.isDelinquent()` would presumably start with testing if the **Invoice** is overdue, but that would just be the beginning. A policy on grace periods could depend on the status of the customer's account. Some delinquent invoices will be ready for a second notice, while others will be ready to be sent to a collection agency. Payment history of the customer, company policy on different product lines… The clarity of **Invoice** as a request for payment will soon be lost in the sheer mass of rule evaluation code. The **Invoice** will also develop all sorts of dependencies on domain classes and subsystems that do not support that basic meaning.

At this point, in an attempt to save the **Invoice** class, a developer will often refractor the rule evaluation code into the application layer (in this case, the bill collection application). Now the rules have been separated from the domain layer altogether, leaving behind a dead data object that does not express the rules inherent in the business model. These rules need to stay in the domain layer, but they don't fit into the object being evaluated (the **Invoice** in this case). Not only that, but evaluating methods swell with conditional code, which make the rule hard to read.

Developers working in the logic-programming paradigm would handle this differently. Such rules would be expressed as "predicates". Predicates are functions that evaluate to "true" or "false" and can be combined using operators like "and" and "or" to express more complex rules. With predicates, we could declare rules explicitly and use them with the **Invoice**. If only we were in the logic paradigm.

Seeing this, people have made attempts at implementing logical rules in terms of objects. Some of these attempts were very sophisticated, others naïve. Some were ambitious, others modest. Some turned out valuable, some were tossed aside as failed experiments. A few were allowed to derail their projects. One thing is clear: As appealing as the idea is, full implementation of logic in objects is a major undertaking. (After all, logic programming is a whole modeling and design paradigm in its own right.)

Business rules often do not fit the responsibility of any of the obvious ENTITIES or VALUE OBJECTS, and their variety and combinations can overwhelm the basic meaning of the domain object. But moving the rules out of the domain layer is even worse, since the domain code no longer expresses the model.

Logic programming provides the concept of separate, combinable, rule objects called "predicates", but full implementation of this concept with objects is cumbersome. It is also so general that it doesn't communicate intent as much as more specialized designs.

Fortunately, we don't really need to fully implement it to get a large benefit. Most of our rules fall into a few special cases. We can borrow the concept of predicates and create specialized objects that evaluate to a Boolean. Those testing methods that get out of hand neatly expand into objects of their own. They are little truth tests that can be factored out into a separate VALUE OBJECT. This new object can evaluate another object to see if the predicate is true for that object.

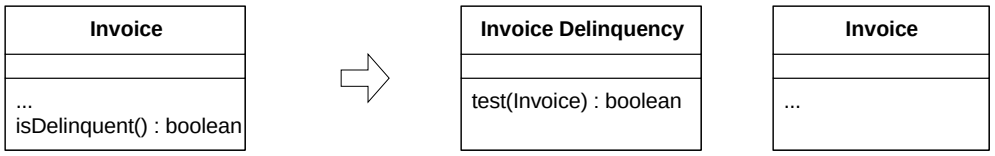


Figure 9. 12

To put it another way, the new object is a "specification". A SPECIFICATION states a constraint on the state of another object, which may or may not be present. It has multiple uses, but the most basic concept of it is that a SPECIFICATION can test any object to see if it satisfies the specified criteria.

Therefore,

Create explicit predicate-like VALUE OBJECTS for specialized purposes. A SPECIFICATION is a predicate that determines if an object does or does not satisfy some criteria.

Many SPECIFICATIONS are simple, special-purpose tests, as in the delinquent invoice example. In cases where the rules become more complex, more predicate logic capability can be applied, such as combining with logical operators. This will be discussed in the next chapter. The fundamental pattern stays the same and provides a path from the simpler to more complex models.

The case of the delinquent invoice can be modeled using a SPECIFICATION that states what it means to be delinquent, and which can evaluate any Invoice and make that determination.

More Complex Delinquency Rule Factored Into SPECIFICATION

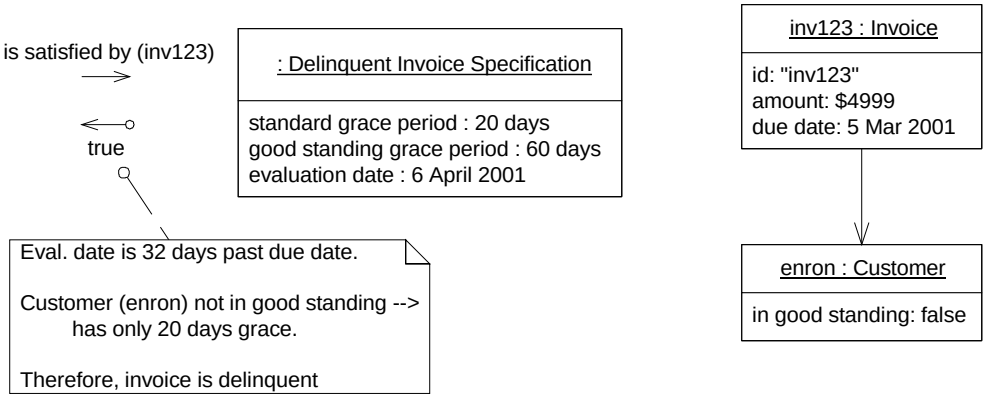


Figure 9. 13

The SPECIFICATION keeps the rule in the DOMAIN LAYER. Since the rule is a full-fledged object, the design can be more expressive. The FACTORY of a SPECIFICATION can configure it using information from other sources, such as the customer's account or the corporate policy database. Providing direct access to these from the **Invoice** would couple the objects in a way that does not relate to the request for payment (the basic responsibility of **Invoice**). The SPECIFICATION can be given the information it will need to do its job in a simple, straightforward way. In this case, the **Delinquent Invoice Specification** was meant to be transient, so it was created with a specific evaluation date built in.

♦ ♦ ♦

The basic concept of SPECIFICATION is very simple and helps us think about a domain modeling problem. But a MODEL-DRIVEN DESIGN requires an effective implementation that also expresses the concept. To pull that off requires digging a little deeper into how the pattern will be applied. A domain pattern is not just a neat idea for a UML diagram; it is a solution to a programming problem that retains a MODEL-DRIVEN DESIGN.

When you apply a pattern appropriately, you can tap into a whole body of thought about how to approach a class of domain modeling problem and experience in finding effective implementations. There is a lot of detail in this pattern description—many options for features and approaches to implementation. A pattern is not a cookbook. It lets you start from a base of experience to develop your solution, and it gives you some language to talk about what you are doing.

You may want to skim the key concepts when first reading. Later, when you run into the situation, you can come back and draw on the experience captured in the detailed discussion. Then you go and figure out a solution to your problem.

Applying and Implementing SPECIFICATION

Much of the value of SPECIFICATION is that it unifies application functionality that may seem quite different. We need to specify the state of an object for three reasons:

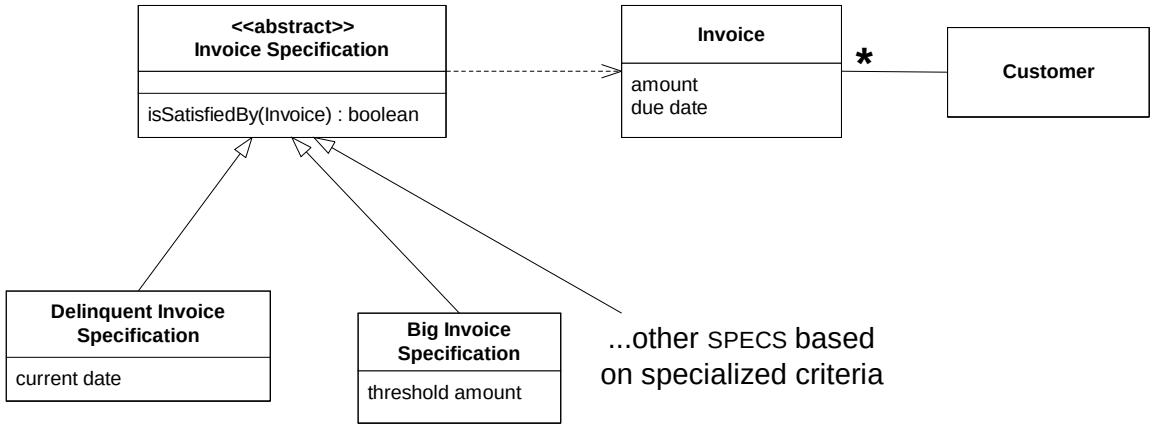
- validation of an object to see if it fulfills some need or is ready for some purpose
- selection of an object from a collection (as in the case of the overdue invoices)
- specifying the creation of a new object to fit some need.

These three uses: validation, selection, and building-to-order, are the same on a conceptual level. Without a pattern like SPECIFICATION, the same rule may show up in different guises, and possibly contradictory forms. The conceptual unity can be lost. Applying the SPECIFICATION pattern allows a consistent model to be used, even when the implementation may have to diverge.

Validation

The simplest use of a SPECIFICATION is validation, and it is the one that demonstrates the concept most straightforwardly.

Figure 9. 14 An Implementation of SPECIFICATION for Validation




```
class DelinquentInvoiceSpecification extends InvoiceSpecification {
   private Date currentDate;
   // An instance is used and discarded on a single date.

   public DelinquentInvoiceSpecification(Date currentDate) {
      this.currentDate = currentDate;
   }

   public boolean isSatisfiedBy(Invoice candidate) {
```

```
        int gracePeriod = candidate.customer().getPaymentGracePeriod();
        Date firmDeadline =
           DateUtility.addDaysToDate(candidate.dueDate(), gracePeriod);
        return currentDate.after(firmDeadline);
     }

  }
```

Now, suppose we need to display a red flag whenever a salesman brings up a customer with delinquent bills. We just have to write a method in a client class something like this.

```
  public boolean accountIsDelinquent(Customer customer) {
     Date today = new Date();
     Specification delinquentSpec =
        new DelinquentInvoiceSpecification(today);

     Iterator it = customer.getInvoices().iterator();
     while (it.hasNext()) {
        Invoice candidate = (Invoice)it.next();
        if (delinquentSpec.isSatisfiedBy(candidate)) return true;
     }
     return false;
  }
```

Querying

Validation tests an individual object to see if it meets some criteria, presumably so that the client can act on the conclusion. Another common need is to select a subset of a collection of objects based on some criteria. The same concept of SPECIFICATION can be applied here, but implementation issues are different.

Suppose there was an application requirement to list all customers with delinquent **Invoices.** In theory, the **Delinquent Invoice Specification** that we defined before will still serve, but in practice its implementation would probably have to change. To demonstrate that the *concept* is the same, let's assume first that the number of **Invoices** is small, maybe already in memory. In this case, the straight-forward implementation developed for validation still serves. The **Invoice Repository** could have a generalized method to select **Invoices** based on a SPECIFICATION:

```
  public Set selectSatisfying(InvoiceSpecification spec) {

     Set results = new HashSet();
     Iterator it = invoices.iterator();
     while (it.hasNext()) {
        Invoice candidate = (Invoice) it.next();
        if (spec.isSatisfiedBy(candidate)) results.add(candidate);
     }

     return results;
  }
```

So a client could obtain a collection of all delinquent **Invoices** with a single code statement:

```
  Set delinquentInvoices = invoiceRepository.selectSatisfying(
     new DelinquentInvoiceSpecification(currentDate));
```

That establishes the concept behind the operation. Of course, the **Invoice** objects probably aren't in memory. There may be thousand of them. In a typical business system, the data are probably in a relational

database. And, as pointed out in earlier chapters, the model focus tends to get lost at these intersections with other technologies.

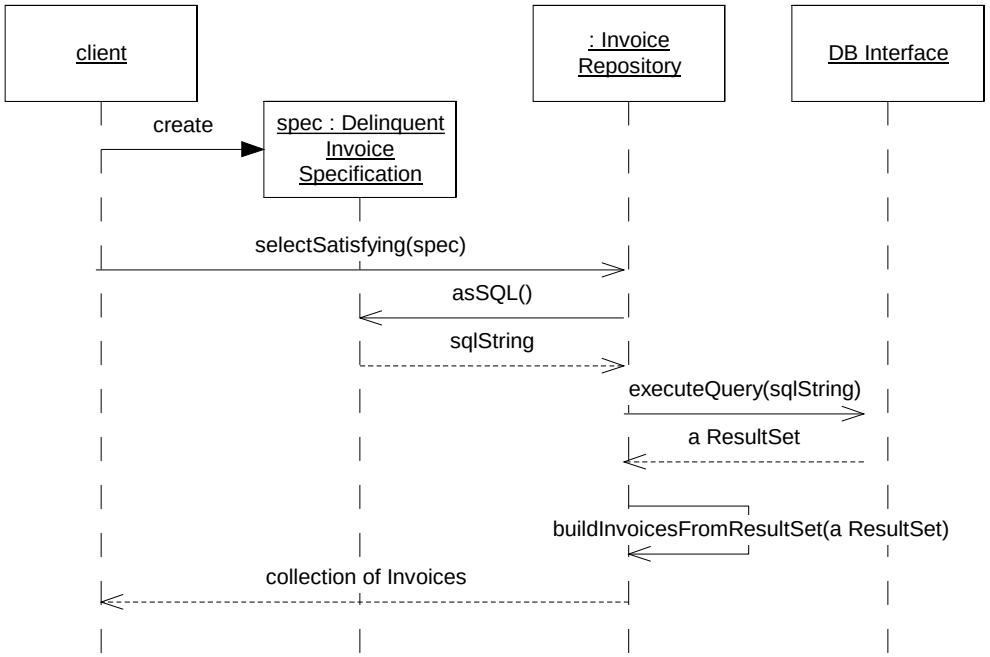
Relational databases have powerful search capabilities. How can we take advantage of that power to solve this problem efficiently while retaining the model of a SPECIFICATION? MODEL-DRIVEN DESIGN demands that the model stay in lock step with the implementation, but it allows freedom to choose any implementation that faithfully captures the meaning of the model. Lucky for us, SQL is a very natural way to write SPECIFICATIONS.

Here is a very simple example in which the query is encapsulated in the same class as the validation rule. A single method is added to the **Invoice Specification** and is implemented in the **Delinquent Invoice Specification** subclass as:

```
public String asSQL() {
   return "SELECT *" +
          " FROM INVOICE, CUSTOMER" +
          " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
          " AND TO_DAYS(INVOICE.DUE_DATE) + CUSTOMER.GRACE_PERIOD" +
          "   < TO_DAYS(" + SQLUtility.dateAsSQL(currentDate) + ")";
}
```

SPECIFICATIONS mesh smoothly with REPOSITORIES, which the building-block mechanism for providing query access to domain objects and encapsulating the interface to the database.

Figure 9. 15 Interaction Between REPOSITORY and SPECIFICATION



Now, this design has some problems. Most importantly, the details of the table structure have leaked into the DOMAIN LAYER, and this is typically isolated in a mapping layer that relates the domain objects to the relational tables. Implicitly duplicating that information here could hurt modifiability and maintainability of the **Invoice** and **Customer** objects, since any change to their mappings now have to be tracked in more than one place. But it is a simple illustration of a way to keep the rule in just one place. Some object-relational mapping frameworks provide the means to express such a query in terms of the model objects

and attributes, generating the actual SQL in the infrastructure layer. This would let us have our cake and eat it too.

When there the infrastructure doesn't come to the rescue, we can refactor the SQL out of the expressive domain objects by adding a specialized query method to the **Invoice Repository**. To avoid embedding the rule into the REPOSITORY, we have to express the query in a more generic way that doesn't capture the rule but can be combined or placed in context to work the rule out (in this example, by using a double dispatch).

```
public class InvoiceRepository {

   public Set selectWhereGracePeriodPast(){
      //This is not a rule, just a specialized query.
      String sql = whereGracePeriodPast_SQL();
      ResultSet queryResultSet =
         SQLDatabaseInterface.instance().executeQuery(sql);
      return buildInvoicesFromResultSet(queryResultSet);
   }

   public String whereGracePeriodPast_SQL() {
      return "SELECT *" +
             " FROM INVOICE, CUSTOMER" +
             " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
             " AND TO_DAYS(INVOICE.DUE_DATE) + CUSTOMER.GRACE_PERIOD" +
             "   < TO_DAYS(" + SQLUtility.dateAsSQL(currentDate) + ")";
   }

   public Set selectSatisfying(InvoiceSpecification spec) {
      return spec.satisfyingElementsFrom(this);
   }
}
```

The `asSql()` method on **Invoice Specification** is replaced with `satisfyingElementsFrom(InvoiceRepository)`, which is implemented on **Delinquent Invoice Specification** as:

```
public class DelinquentInvoiceSpecification {
   // basic DelinquentInvoiceSpecification code here

   public Set satisfyingElementsFrom(InvoiceRepository repository) {
      //Delinquency rule is defined as due date and grace period past.
      return repository.selectWhereGracePeriodPast();
   }
}
```

This puts the SQL in the REPOSITORY, while the SPECIFICATION controls what query should be used. It doesn't collect the rules as neatly into the SPECIFICATION, but the essential declaration is there of what constitutes delinquency (i.e. past grace period).

The REPOSITORY now has a very specialized query that most likely will only be used in this case. That is acceptable, but, depending on the relative numbers of **Invoices** that are overdue to those that are delinquent, an intermediate solution that leaves the REPOSITORY methods more generic may still give good performance, while keeping the SPECIFICATION more self-explanatory.

```
public class InvoiceRepository {
```

```
   public Set selectWhereDueDateIsBefore(Date aDate) {
      String sql = whereDueDateIsBefore_SQL();
      ResultSet queryResultSet =
         SQLDatabaseInterface.instance().executeQuery(sql);
      return buildInvoicesFromResultSet(queryResultSet);
   }

   public String whereDueDateIsBefore_SQL() {
      return "SELECT *" +
             " FROM INVOICE" +
             " WHERE TO_DAYS(INVOICE.DUE_DATE)" +
             "    < TO_DAYS(" + SQLUtility.dateAsSQL(currentDate) +")";
   }

   public Set selectSatisfying(InvoiceSpecification spec) {
      return spec.satisfyingElementsFrom(this);
   }
}

public class DelinquentInvoiceSpecification {
   //basic DelinquentInvoiceSpecification code here

   public Set satisfyingElementsFrom(InvoiceRepository repository) {
      Collection pastDueInvoices =
         repository.selectWhereDueDateBefore(currentDate);

      Set delinquentInvoices = new HashSet();
      Iterator it = pastDueInvoices.iterator();
      while (it.hasNext()) {
         Invoice anInvoice = (Invoice)it.next();
         if (this.isSatisfiedBy(anInvoice))
            delinquentInvoices.add(anInvoice);
      }
      return delinquentInvoices;
   }
}
```

There is a performance hit since we'll pull out more **Invoices** and then have to select from them in memory. Whether this is an acceptable performance cost for the better factoring of responsibility purely depends on circumstances. There are many ways to implement the interactions between SPECIFICATIONS and REPOSITORIES, to take advantage of the development platform, while keeping the basic responsibilities.

Sometimes, to improve performance, or more likely to tighten security, queries may be implemented on the server as stored procedures. In that case, the SPECIFICATION could only carry the parameters allowed by the stored procedure. Other than that, there is no difference in the model between these various implementations. The choice of implementation is free except where specifically constrained by the model. The price comes in a more cumbersome way of writing and maintaining queries.

This barely scratches the surface of the challenges of combining SPECIFICATIONS with databases, and I'll make no attempt to cover all the considerations that may arise. I just want to give a taste of the kind of choices that have to be made. Mee and Hieatt discuss some of the technical issues involved in designing REPOSITORIES with SPECIFICATIONS in [Fowler2002].

Building to Order (Generating)

When the Pentagon wants a new fighter jet, they write a specification. This specification may require that the jet reach Mach 2, that it have a range of 1800 miles, that it cost no more than $50 million. But, however detailed it is, the specification is not a design, much less a plane. An aerospace engineering company will take the specification and create one or more designs based on it. Competing companies may produce different designs, all of which presumably satisfy the original spec.

Many computer programs generate things, and those things have to be specified. When you place a picture into a word-processing document, the text flows around it. You have specified the location of the picture, and perhaps the style of text flow. The exact placement of the words on the page is then worked out by the word processor in such a way that it meets your specification.

Once again, this is the same concept of a SPECIFICATION because we are creating a constraint for objects that are not yet present. The implementation will be quite different, however. This is not a filter for preexisting objects, as with querying. It is not a test for an existing object, as with validation. This time, a whole new object or set of objects will be made or reconfigured to satisfy the SPECIFICATION.

Without using SPECIFICATION, a generator can be written that has procedures or a set of instructions that create the needed objects. This implicitly defines the behavior of the generator.

Instead, an interface of the generator that is defined in terms of a descriptive SPECIFICATION explicitly constrains the generated objects. This will have several advantages.

- The generator's implementation is decoupled from its interface. The SPECIFICATION declares the requirements for the output, but does not define how that result is reached.
- The interface communicates its rules explicitly, so developers can know what to expect from the generator without understanding all details of its operation. The only way to predict the behavior of a procedurally defined generator is to run cases or to understand every line of code.
- The interface is more flexible, or can be enhanced with more flexibility, since the statement of the request is in the hands of the client, while the server (generator) is only obligated to fulfill the letter of the SPECIFICATION.
- Last, but not least, this kind of interface is easier to test, since the model contains an explicit way to define input into the generator *that is also a validation of the output*. That is, the same SPECIFICATION that is passed into the generator's interface to constrain the creation process can also be used, in its validation role (if the implementation supports it) to confirm the created object is correct.

(This is an example of an INTENTION REVEALING INTERFACE and ASSERTIONS, discussed in depth in Chapter 10.)

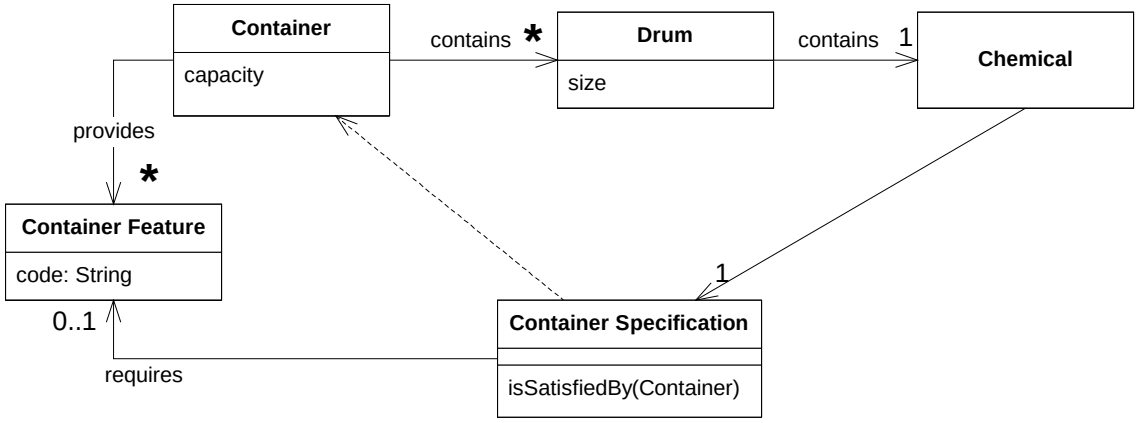
"Building-to-order" can mean creation of an object from scratch, but it is often a configuration of existing objects to satisfy the SPEC'.

Example: Chemical Warehouse Packer

There is a warehouse in which various chemicals are stored in stacks of large containers, similar to boxcars. Some chemicals are inert, and can be stored about anywhere. Some are volatile, and have to be stored in specially ventilated containers. Some are explosive and have to be stored in specially armored containers. There are also rules about the combinations allowed in a container.

The goal is to write software that will find an efficient and safe way to put the chemicals in the containers.

Figure 9. 16 Model for Warehouse Inventory



We could start writing a procedure to take a chemical and place it in a container, but, instead, let's start with the validation problem. This will force us to make the rules explicit, and will give us a way to test the final implementation.

Each chemical will have a container SPECIFICATION:

Chemical	Container Specification
TNT	Armored container
Sand	
Biological Samples	Must not share container with explosives
Ammonia	Ventilated container

Now, if we write these as **Container Specifications**, we should be able to take a configuration of packed containers and tests to see if it meets these constraints.

Container Features	Contents	Specification Satisfied?
Armored	20 lbs TNT 500 lbs Sand	√
	50 lbs biological samples	√
	Ammonia	X

A method on ContainerSpecification, isSatisfied(), would have to be implemented to check for needed ContainerFeatures. For example, the SPEC attached to an explosive chemical would look for the “armored” feature:

```
public class ContainerSpecification {
   private ContainerFeature requiredFeature;

   public ContainerSpecification(ContainerFeature requiredFeature) {
      this.requiredFeature = requiredFeature;
   }
```

```
    boolean isSatisfiedBy(Container aContainer){
       return aContainer.getFeatures().contains(requiredFeature);
    }
 }
```

Sample client code to set up an explosive chemical:

```
 tnt.setContainerSpecification(new ContainerSpecification(ARMORED));
```

A method on Container, `isSafelyPacked()`, would confirm that the drum current contained in the Container met the requirements for

```
 boolean isSafelyPacked(){
    ContainerSpecification packingSpec =
       getDrum().getContainerSpecification();
    Iterator it = contents.iterator();
    while (it.hasNext()) {
       Drum drum = (Drum)it.next();
       if (!drum.containerSpecification().isSatisfiedBy(this))
          return false;
    }
    return true;
 }
```

At this point, we could write a monitoring application that would take the inventory database and report any unsafe situations.

```
 Iterator it = containers.iterator();
 while (it.hasNext()) {
    Container container = (Container)it.next();
    if (!container.isSafelyPacked())
       unsafeContainers.add(container);
 }
```

This is not the software we've been asked to write. It would be good to let the business people know of the opportunity, but we have been charged with designing a packer. What we have is a test for a packer. This understanding of the domain and our SPECIFICATION based model put us in a position to define a clear and simple interface for a SERVICE that will take collections of Drums and Containers and pack them in compliance with the rules.

```
 public interface WarehousePacker {
    public void pack(Collection containersToFill, Collection drumsToPack)
                                                 throws NoAnswerFoundException;
       /* ASSERTION: At end of pack, the ContainerSpecification
       of each Drum shall be satisfied by its Container. If no
       complete solution can be found, an exception shall be thrown. */

 }
```

Now the task of designing an optimized constraint solver to fulfill the responsibilities of the Packer service has been decoupled from the rest of the application, and those mechanisms will not clutter the part of the design that expresses the model. (See Declarative Style of Design, Chapter 10, and COHESIVE MECHANISM, Chapter 15) Yet the rules *governing* packing have not been pulled out of the domain objects.

<<Begin Side Bar>>

Clearing Development Logjams with Working Prototypes

Often, a development process is helped along by a working prototype, even if it does not satisfy all requirements. With the parts made in the warehouse packer example, we could build a very simple implementation of a Packer that could help the development process move along.

```
public class Container {
   private double capacity;
   private Set contents; //Drums

   public boolean hasSpaceFor(Drum aDrum) {
      return remainingSpace() >= aDrum.getSize();
   }

   public double remainingSpace() {
      double totalContentSize = 0.0;
      Iterator it = contents.iterator();
      while (it.hasNext()) {
         Drum aDrum = (Drum) it.next();
         totalContentSize = totalContentSize + aDrum.getSize();
      }
      return capacity – totalContentSize;
   }

   public boolean canAccommodate(Drum aDrum) {
      return hasSpaceFor(aDrum) &&
         aDrum.getContainerSpecification().isSatisfiedBy(this);
   }

}


public class PrototypePacker implements WarehousePacker {

   public void pack(Collection containers, Collection drums)
                                 throws NoAnswerFoundException {
      /* This method fulfills the ASSERTION as written. However,
         when an exception is thrown, Containers contents may
         have changed. Rollback must be handled at a higher level. */

      Iterator it = drums.iterator();
      while (it.hasNext()) {
         Drum drum = (Drum) it.next();
         Container container = findContainerFor(containers, drum);
         container.add(drum);
      }
   }

   public Container findContainerFor(Collection containers, Drum drum)
                                    throws NoAnswerFoundException {
      Iterator it = containers.iterator();
      while (it.hasNext()) {
         Container container = (Container) it.next();
         if (container.canAccommodate(drum))
```

```
                return container;
            }
            throw new NoAnswerFoundException();
        }

    }
```

Granted that this leaves a lot to be desired. It may pack sand into specialty containers and then run out of room before it packs the hazardous chemicals. It certainly doesn't optimize revenues. But a lot of optimization problems are never solved perfectly anyway. This does follow the rules that have been stated so far.

The point is that having any working implementation at all allows project work to go on much more flexibly. When the time is right, it can (and probably must) be replaced by a more effective implementation, quite possibly developed by specialists in optimization algorithms. In the mean time, all other parts of the system have something to interact with during development.

Here is an example of a "simplest thing that could possibly work" that actually becomes possible because of a more sophisticated model. We can have a functioning prototype of a very complex component in a couple dozen lines of easily understood code. A less model-driven approach would be harder to understand, be harder to upgrade (because the **Packer** would be more coupled to the rest of the design), and, in this case, would likely take longer to prototype.

<<End Sidebar>>

10.Supple Design

The ultimate purpose of software is to serve users. But first it has to serve developers. This is especially true in a process that emphasizes refactoring. As the program evolves, developers will rearrange and rewrite every part. They will integrate the domain objects into the application and with new domain objects. Even years later, maintenance programmers will be changing and extending the code. People have to work with this stuff. But will they want to?

When software with complex behavior lacks a good design, it becomes hard to refactor or combine elements. Duplication starts to appear as soon as a developer isn't confident of predicting the full implications of a computation. Duplication is forced when design elements are monolithic so that the parts cannot be recombined. Classes and methods can be broken down for better reuse, but it gets hard to keep track of what all the little parts do. When software doesn't have a clean design, developers dread even looking at the existing mess, much less making a change that could aggravate the tangle or break something through an unforeseen dependency. In any but the smallest systems, this places a ceiling on the richness of behavior it is feasible to build. It stops refactoring and iterative refinement.

To have a project accelerate as development proceeds rather than get weighed down by its own legacy demands a design that is a pleasure to work with; inviting to change. A supple design.

Supple design is the complement to deep modeling. Once you've dug out implicit concepts and made them explicit, you have the raw material. Through the iterative cycle you hammer that material into a useful shape, cultivating a model that simply and clearly captures the key concerns and shaping a design that allows a client developer to really put that model to work. Development of the design and code leads to insight that refines model concepts. Round and round – we're back to the iterative cycle and refactoring toward deeper insight. But what kind of design are you trying to arrive at? What kind of experiments should you try along the way? That is what this chapter is about.

A lot of over-engineering has been justified in the name of flexibility. But, more often than not, excessive layers of abstraction and indirection get in the way. Look at the design of software that really empowers the people who handle it, and you will usually see something simple. Simple is not easy. To make complex systems work, a dedication to MODEL-DRIVEN DESIGN has to be joined with a moderately rigorous design style. It may well require relatively sophisticated design skill to create *or to use*.

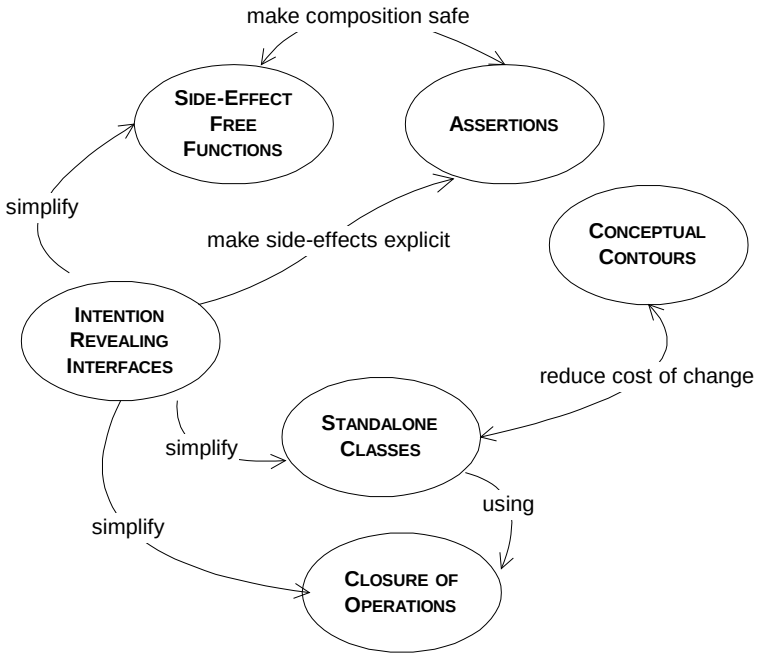
Developers play two roles, each of which must be served by the design. The same person might well play both roles -- might even switch back and forth in minutes -- but the relationship to the code is different nonetheless. One role is the developer of a client, who weaves the domain objects into the application code or other domain layer code, utilizing capabilities of the design. A supple design reveals a deep underlying model that makes its potential clear. The client developer can flexibly use a minimal set of concepts to express a range of scenarios in the domain. Design elements fit together in a natural way with a result that is predictable, clearly characterized and robust.

Equally important, the design must serve the developer working to change it. To be open to change, a design must be easy to understand, revealing that *same* underlying model that the client developer is drawing on. It must follow the contours of a deep model of the domain, so most changes bend the design at flexible points. The effects of its code must be obvious, so the consequences of a change will be easy to anticipate.

Early versions of a design are usually stiff. Many never acquire any suppleness in the time-frame/budget of the project. I've never seen a large program that had this quality throughout. But when complexity is holding back progress, honing the most crucial, intricate parts to a supple design makes the difference between getting sucked down into legacy maintenance and punching through the complexity ceiling.

There is no formula for designing software like this, but I have culled a set of patterns that, in my experience, tend to lend suppleness to a design when they fit. These patterns and examples should give a feel for what a supple design is like and the kind of thinking that goes into it.

Some Patterns That Contribute to Supple Design



We want to think about meaningful domain logic. Code that produces the effect of a rule without explicitly stating the rule forces us to think of step-by-step software procedures. The same applies to a calculation that just results from running some code, but isn't explicit. Without a clear connection to the model, it is difficult to understand the effect of the code or anticipate the effect of a change. The previous chapter delved into modeling rules and calculations explicitly. Implementing such objects requires a lot of understanding of the gritty details of the calculation or fine print of the rule. The beauty of objects is their ability to encapsulate all that so that client code is simple and can be interpreted in terms of higher-level concepts.

But if the interface doesn't tell the client developer what he needs to know in order to use the object effectively, he will have to dig into the internals and understand the details. A reader of the client code will have to do that same. Then most of the value of the encapsulation is lost. We are always fighting cognitive overload. If the client developer's mind is flooded with detail about how a component does its job, his mind isn't clear to work out the intricacies of the client design. This is true even when the same person is playing both roles, developing and using his own code, because even if he doesn't have to learn those details, there is a limit to how many factors he can consider at once.

If a developer must consider the implementation of a component in order to use it, the value of encapsulation is lost. If someone other than the original developer must infer the purpose of an object or operation based on its implementation, that new developer may infer a purpose that the operation or class fulfills only by chance. If that was not the intent, the code may work for the moment, but the conceptual basis of the design will have been corrupted, and the two developers will be working at cross-purposes.

To obtain the value of explicitly modeling a concept in the form of a class or method, we must give these program elements names that reflect those concepts. The names of classes and methods are great opportunities for improving communication between developers, and for improving the abstraction of the system.

Kent Beck wrote of an INTENTION REVEALING SELECTOR [Beck 1997]. All public elements of a design together make up its interface, and the name of each of those elements presents an opportunity to reveal the intention of the design. Type names, method names, and argument names all combine to form an INTENTION REVEALING INTERFACE.

Therefore,

Name classes and operations to describe their effect and purpose, without reference to the means by which they do what they promise. This relieves the client developer of the need to understand the internals. These names should conform to the UBIQUITOUS LANGUAGE so that team members can quickly infer their meaning. Write a test for a behavior before creating it, to force your thinking into client developer mode.

All the tricky mechanism should be encapsulated behind abstract interfaces that speak in terms of intentions, rather than means.

In the public interfaces of the domain, state relationships and rules, but not how they are enforced; describe events and actions, but not how they are carried out; formulate the equation but not the numerical method to solve it. Pose the question but don't present the means by which the answer shall be found.

Example Refactoring: Paint Mixing Application

A program for paint stores can show a customer the result of mixing standard paints. Here is the initial design, which has a single domain class.

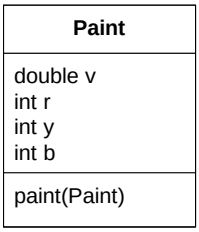


Figure 10. 1

The only way to even guess what the `paint(Paint)` method does is to read the code.

```
public void paint(Paint paint) {
   v = v + paint.getV(); //After mixing, volume is summed
   // Ommitted -- many lines of complicated color mixing logic
   // ending with the assignment of new r, b, and y values.
}
```

Ok, so it looks like this method combines two Paints together, the result having a larger volume and a mixed color.

To shift our perspective, let's write a test for this method. (This code is based on the JUnit test framework.)

```
public void testPaint() {
    // Create a pure yellow paint with volume=100
    Paint yellow = new Paint(100.0, 0, 50, 0);
    // Create a pure blue paint with volume=100
    Paint blue = new Paint(100.0, 0, 50, 0);

    // Mix the blue into the yellow
    yellow.paint(blue);

    // Result should be volume of 200.0 of green paint
    assertEquals(200.0, yellow.getV(), 0.01);
    assertEquals(25, yellow.getB());
    assertEquals(25, yellow.getY());
    assertEquals(0, yellow.getR());
}
```

The passing test is the starting point. It is unsatisfying at this point because the code in the test doesn't tell us what it is doing. Let's rewrite the test to reflect the way we would *like* to use the **Paint** objects. Initially, this test will fail. In fact, it won't even compile. We are writing it to explore the interface design of the **Paint** object from the client developer's point of view.

```
public void testPaint() {
    // Create a pure yellow paint with volume=100
    Paint ourPaint = new Paint(100.0, 0, 50, 0);
    // Create a pure blue paint with volume=100
    Paint blue = new Paint(100.0, 0, 50, 0);

    // Mix the blue into the yellow
    ourPaint.mixIn(blue);

    // Result should be volume of 200.0 of green paint
```

```
        assertEquals(200.0, ourPaint.getVolume(), 0.01);
        assertEquals(25, ourPaint.getBlue());
        assertEquals(25, ourPaint.getYellow());
        assertEquals(0, ourPaint.getRed());
    }
```

We should take our time to write a test that reflects the way we would like to talk to these objects. After that, we refactor the **Paint** class to make the test pass.

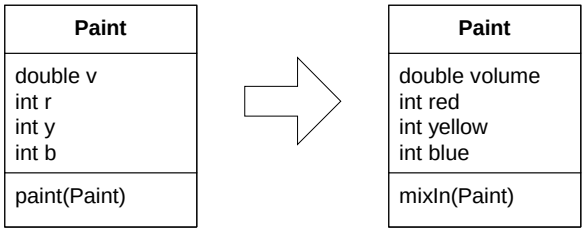


Figure 10. 2

The new method name may not tell the reader everything about the effect of "mixing in" another Paint (for that we'll need ASSERTIONS, coming up in a few pages), but it will clue them in enough to get started using the class, especially with the example the test provides. And it will allow the reader of the client code to interpret the client's intent. In the next few sections, we'll refactor this class again to make it even clearer.

♦ ♦ ♦

Entire subdomains can be carved off into separate modules and encapsulated behind an INTENTION REVEALING INTERFACES. Using this to focus a project and manage the complexity of a large system will be discussed more in Chapter 15, "Distillation", with COHESIVE MECHANISMS and GENERIC SUBDOMAINS.

But next, we'll continue with making the consequences of using a method very predictable, first by greatly simplifying some of them as with SIDE-EFFECT-FREE FUNCTIONS, and then by characterizing others with ASSERTIONS.

SIDE-EFFECT-FREE FUNCTIONS

Operations can be broadly divided into two categories, commands and queries. Queries obtain information from the system, possibly by a simply accessing data in a variable, possibly performing a calculation based on that data. Commands (also known as modifiers) are operations that affect some change to the systems, (for a simple example, by setting a variable). In standard English, the word "side effect" implies an unintended consequence, but in computer science it means any effect on the state of the system. For our purposes, let's narrow that meaning to any change in the state of the system that will affect future operations.

Why was the term "side effect" adopted and applied to quite intentional changes affected by operations? I assume this was based on experience with complex systems. Most operations call on other operations. The operations they call invoke still other operations. As soon as this arbitrarily deep nesting is involved, it becomes very hard to anticipate all the consequences of invoking an operation. The developer of the client may not have intended the effects of the second and third tier operations – they've become side effects in every sense of the word. There are other ways that elements of a complex design interact that are likely to produce the same result. The use of the term side effect underlines the inevitability of that.

Interactions of multiple rules or compositions of calculations become extremely difficult to predict. The developer calling an operation must understand its implementation and the implementation of all its delegations in order to anticipate the result. The value of any abstraction of interfaces is limited if the developers are forced to pierce the veil. Without safely predictable abstractions, the developers must limit the combinatory explosion, placing a low ceiling on the richness of behavior that is feasible to build.

Operations that return results without side effects are called "functions". A function can be called multiple times and return the same value each time. A function can call on other functions without worrying about the depth of nesting. Functions are much easier to test than operations that have side effects. For these reasons, functions lower risk.

Obviously, you can't avoid commands in most software systems, but the problem can be mitigated in two ways. First, you can keep the commands and queries strictly segregated in different operations. The methods that cause changes do not return domain data, and are kept as simple as possible. All queries and calculations are done in methods that cause no observable side effects [Meyer 1988].

Second, there are often alternative models and designs that do not call for an existing object to be modified at all. Instead a new VALUE OBJECT, representing the result of the computation, is created and returned. This is a common technique, which will be illustrated in the example. A VALUE OBJECT can be created in answer to a query, handed off, and forgotten, unlike an ENTITY, whose lifecycle is carefully regulated.

VALUE OBJECTS are immutable, which implies that, apart from initializers called only during creation, *all* their operations are functions. VALUE OBJECTS, like functions, are safer to use and easier to test. An operation that mixes logic or calculations with state change should be refactored into two separate operations [Fowler1999] (p. 279). But this, by definition, only applies to ENTITIES. *After* completing the first refactoring, consider a second refactoring to move the responsibility for the complex calculations into a VALUE OBJECT. Can the side effect be completely eliminated by deriving a VALUE OBJECT instead of changing existing state? Can the entire responsibility be moved into a VALUE OBJECT?

Therefore,

Place as much of the logic of the program as possible into functions, operations that return results with no observable side effects. Strictly segregate commands (resulting in modifications to observable state) into very simple operations that do not return domain information. Further control side effects by moving complex logic into VALUE OBJECTS with conceptual definitions fitting the responsibility.

SIDE-EFFECT-FREE FUNCTIONS, especially in immutable VALUE OBJECTS, allow safe combination of operations. When a FUNCTION is presented through an INTENTION REVEALING INTERFACE, a developer can use it without understanding the detail of its implementation.

Example: Refactoring Paint Mixing Application Again

A program for paint stores can show a customer the result of mixing standard paints. Here is the single domain class.

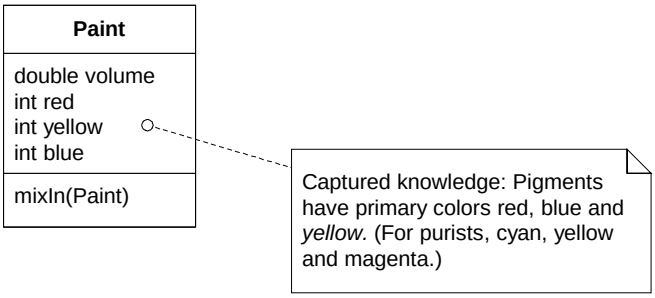


Figure 10. 3

```
public void mixIn(Paint other) {
   volume = volume.plus(other.getVolume());
   // Many lines of complicated color mixing logic
   // ending with the assignment of new red, blue,
   // and yellow values.
}
```

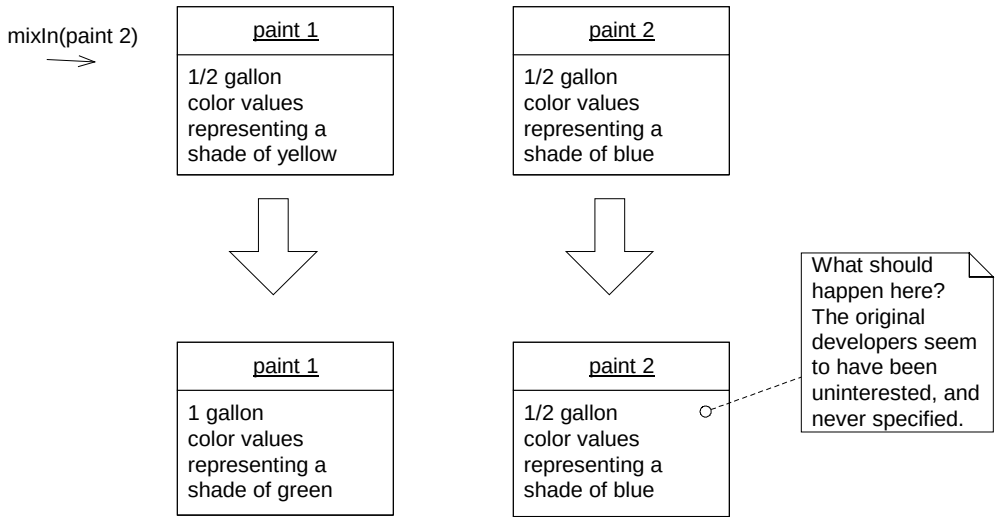


Figure 10. 4

A lot is happening in the `mixIn( )` method, but this design does follow the rule of separating modification from querying. One concern, which we'll take up later, is that the volume of the paint 2 object, the argument of the `mixIn( )` method, has been left in limbo. Paint 2's volume is unchanged by the operation, which doesn't seem quite logical in the context of this conceptual model. This was not a problem for the original developers because, as near

as we can tell, they had no interest in the paint 2 object after the operation, but it is hard to anticipate the consequences of side-effects or their absence. We'll return to this question soon in the discussion of ASSERTIONS. For now, we'll look at color.

Color is an important concept in this domain. Let's try the experiment of making it an explicit object. What should it be called? "Color" comes to mind first, but earlier knowledge crunching had already yielded the important insight that color mixing is different for paint than it is for the more familiar RGB light display. The name needs to reflect this.

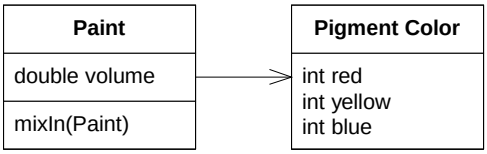


Figure 10. 5

This does communicate more than the earlier version, but the computation is the same, still in they `mixIn( )` method. When we moved out the color data, we should have taken related behavior with it. Before we do, note that **Pigment Color** is a VALUE OBJECT. Therefore, it should be treated as immutable. When we mixed paint, the **Paint** object itself was changed. It was an ENTITY with an ongoing life-story. In contrast, a **Pigment Color** representing a particular shade of yellow is always exactly that. Instead, mixing will result in a new **Pigment Color** object representing the new color.

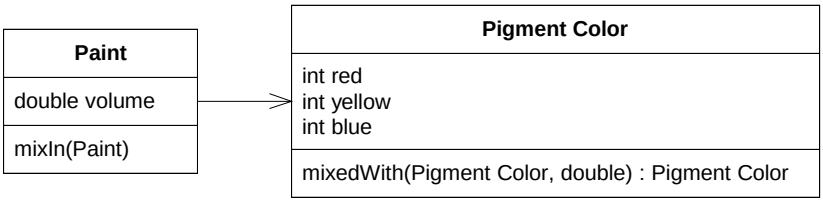


Figure 10. 6

```
public class PigmentColor {

   public PigmentColor mixedWith(PigmentColor other,
                                 double ratio) {
      // Many lines of complicated color mixing logic
      // ending with the creation of a new PigmentColor object
      // with appropriate new red, blue, and yellow values.
   }
}

public class Paint {

   public void mixIn(Paint other) {
      volume = volume + other.getVolume();
      double ratio = other.getVolume() / volume;
      pigmentColor =
         pigmentColor.mixedWith(other.pigmentColor(),ratio);
   }
}
```

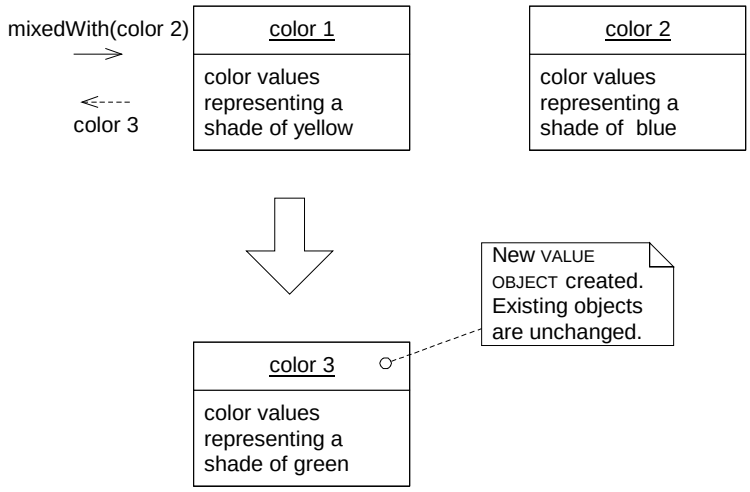


Figure 10. 7

Now the modification code in **Paint** is as simple as possible. The new **Pigment Color** class captures knowledge and communicates it explicitly, and provides a SIDE-EFFECT-FREE FUNCTION whose result is easy to understand, *easy to test*, and safe to use or combine with other operations. Because it is so safe, the complex logic of color mixing is truly encapsulated. Developers using this class don't have to understand the implementation.

ASSERTIONS

Separating complex computations into SIDE-EFFECT-FREE FUNCTIONS cuts the problem down to size, but there is still a residue of commands on the ENTITIES that produce side effects, and anyone using them must understand their consequences. ASSERTIONS make side effects explicit and easier to deal with.

True, a command containing no complex computations may be fairly easy to interpret by inspection. But in a design where larger parts are built of smaller ones, a command may invoke other commands. The developer using the high-level command must understand the consequences of each underlying command. So much for encapsulation. And since object interfaces do not restrict side effects, two subclasses that implement the same interface can have different side effects. The developer using them will want to know which is which to anticipate the consequences. So much for abstraction and polymorphism.

When the side effects of operations are only defined implicitly by their implementation, designs with a lot of delegation become a tangle of cause and effect. The only way to understand a program is to trace execution through branching paths. The value of encapsulation is lost. The necessity of tracing concrete execution defeats abstraction.

We need a way of understanding the meaning of a design element and the consequences of executing an operation without delving into its internals. INTENTION REVEALING INTERFACES carry us part of the way there, but informal suggestions of intentions are not always enough. The "design by contract" school goes the next step, making "assertions" about classes and methods that the developer guarantees will be true. This is discussed in detail in [Meyer 1988]. Briefly, "post conditions" describe the side effects of an operation, the guaranteed outcome of calling a method. "Preconditions" are like the fine print on the contract, the conditions that must be satisfied in order for the post-condition guarantee to hold. Class invariants make assertions about the state of an object at the end of any operation. Invariants can also be declared for entire AGGREGATES, rigorously defining integrity rules.

All these assertions describe state, not procedures, so they are easier to analyze. Class invariants help characterize the meaning of a class, and simplify the client developer's job by making the objects more predictable. If you trust the guarantee of a post-condition, you don't have to worry about how a method works. The effects of delegations should already be incorporated into the assertions.

Therefore,

State post-conditions of operations and invariants of classes and AGGREGATES. If ASSERTIONS cannot be coded directly in your programming language, write automated unit tests for them. Write them into documentation or diagrams where it fits the style of the project's development process.

Seek models with coherent sets of concepts, which lead a developer to infer the intended ASSERTIONS, accelerating the learning curve and reducing the risk of contradictory code.

Even though most object-oriented languages don't currently support ASSERTIONS directly, they can be a powerful way of thinking about a design. Automated unit tests can partially compensate for the lack of language support. Since ASSERTIONS are all in terms of states, rather than procedures, they make tests very easy to write. The test setup puts the preconditions in place, then, after execution, the test checks to see if the post-conditions hold.

Clearly stated invariants and pre- and post-conditions allow a developer to understand the consequences of using an operation or object. Theoretically, any non-contradictory set of assertions would work. But humans don't just compile predicates in their heads. They will be extrapolating and interpolating the concepts of the model, so it is important to find models that make sense to people as well as satisfying the needs of the application.

Example: Back to Paint Mixing

Recall that, in the example of SIDE-EFFECT-FREE FUNCTIONS, I was concerned about the ambiguity of what happens to the argument of the `mixIn(Paint)` operation on the **Paint** class.

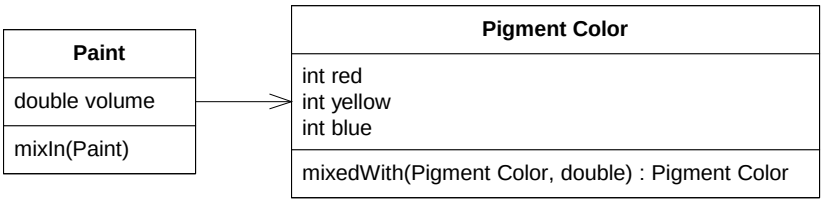


Figure 10. 8

The receiver's volume is increased by the amount of the argument's volume. Drawing on our general understanding of physical paint, this should deplete the other paint by the same amount, draining it to zero volume, or eliminating it completely. The current implementation does not modify the argument, and modifying arguments is a particularly risky kind of side effect anyway.

To start on a solid footing, let's state the post-condition of the `mixIn()` method *as it is*:

- after `p1.mixIn(p2)`, `p1` volume is increased by amount of p2.volume. p2 volume is unchanged

The trouble is, developers are going to make mistakes because these properties don't fit the concepts we have invited them to think about. The straightforward fix would be change the volume of the other paint to zero. Changing an argument is a bad practice, but it would be easy and intuitive. We could state an invariant:

- total volume of paint is unchanged by mixing

But wait! While developers were pondering this option, they made a discovery. It turns out that there was a compelling reason the original designers made it this way. At the end, the program *reports the list of unmixed paints that were added*. After all, the ultimate purpose of this application is to help a user figure out which paints to *put into* a mixture.

So, to make the volume model logically consistent would make it unsuitable for its application requirements. There seems to be a dilemma. Are we stuck with documenting the weird post-condition and trying to compensate with good communication? Not everything in this world is intuitive, and sometimes that is the best answer. But in this case, the awkwardness seems to point to missing concepts. Let's look for a new model.

As we search for a better model, we have significant advantages over the original designers because of the knowledge crunching and refactoring to deeper insight that has happened in the interim. For example, we compute color using a SIDE-EFFECT-FREE FUNCTION on a VALUE OBJECT. This means we can repeat the calculation any time we need to. We should take advantage of that.

We seem to be giving **Paint** two different basic responsibilities. Let's try splitting them.

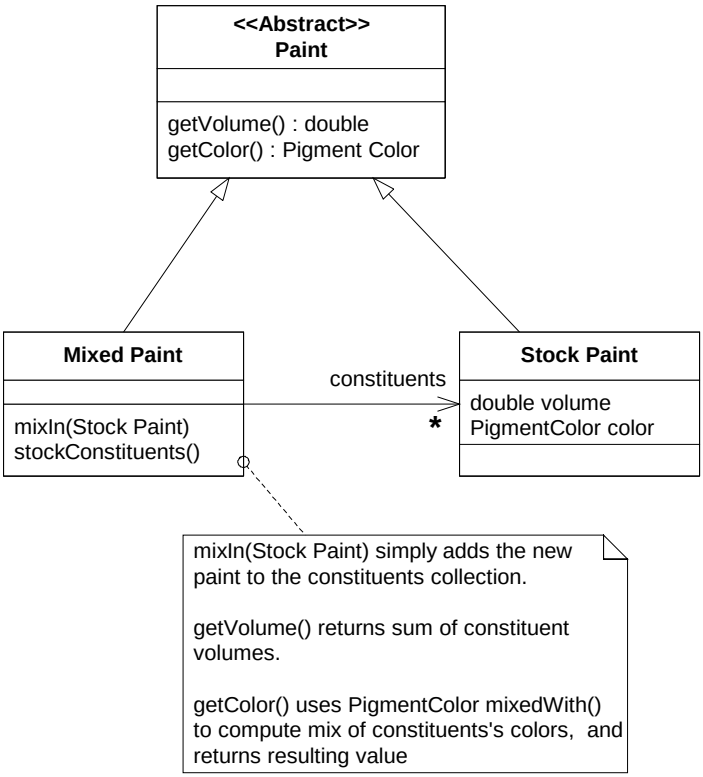


Figure 10. 9

Now there is only one command, `mixIn()`. It just adds an object to a collection, an effect apparent from an understanding of the model. All other operations are SIDE-EFFECT-FREE FUNCTIONS.

A test method confirming one of the ASSERTIONS listed in the diagram above could look something like this (using the JUnit test framework).

```
public void testMixingVolume {
   PigmentColor yellow = new PigmentColor(0, 50, 0);
   PigmentColor blue = new PigmentColor(0, 0, 50);

   StockPaint paint1 = new StockPaint(1.0, yellow);
   StockPaint paint2 = new StockPaint(1.5, blue);
   MixedPaint mix = new MixedPaint();

   mix.mixIn(paint1);
   mix.mixIn(paint2);
   assertEquals(2.5, mix.getVolume());
}
```

This model captures and communicates more of the domain. The invariants and post-conditions make common sense, which will make them easier to maintain and use.

♦ ♦ ♦

The design elements should now be predictable and the communicativeness of the INTENTION REVEALING INTERFACES combined with the predictability given by ASSERTIONS should make encapsulation and abstraction safe.

The next ingredient in recombinable elements is effective decomposition…

CONCEPTUAL CONTOURS

Sometimes people chop functionality fine to allow flexible combination. Sometimes they lump it large to encapsulate complexity. Sometimes they seek a consistent granularity, making all classes and operations to a similar scale. These are oversimplifications that don't work well as general rules. But they are motivated by a basic set of problems.

When elements of a model or design are embedded in a monolithic construct, their functionality gets duplicated. The external interface doesn't say everything a client might care about. Their meaning is hard to understand, because different concepts are mixed together.

On the other hand, breaking classes and methods down can pointlessly complicate the client, forcing client objects to understanding how tiny pieces fit together. Worse, a concept can be lost completely. Half of a uranium atom is not uranium. And, of course, it isn't just grain size that counts, but just where the grain runs.

Cookbook rules don't work. But there is a logical consistency deep in most domains, or they would not be viable in their own sphere. This is not to say that they are perfectly consistent, and certainly the ways people talk about them are not consistent. But there is rhyme and reason somewhere or modeling would be pointless. Because of this underlying consistency, when we find a model that resonates with some part of the domain, it is more likely to be consistent with other parts that we discover later. Sometimes the new discovery isn't easy for the model to adapt to, in which case we refactor to deeper insight, and hope to conform to the *next* discovery.

This is one reason why repeated refactoring eventually leads to suppleness. The CONCEPTUAL CONTOURS emerge as the code is adapted to newly understood concepts or requirements.

At all scales, from individual methods up through classes and modules to large-scale structures (Chapter ##), design employs the conventional criteria of high-cohesion and low coupling of both concepts and code. This can be tempered by always touching base with your intuition for the domain. With each decision, ask yourself, Is this an expedient based on a particular set of relationships in the current model and code, or does it echo some contour of the underlying domain?

Find the conceptually meaningful unit of functionality, and the design will be both flexible and easily understood. For example, if an "addition" of two objects has a coherent meaning in the domain, then implement methods at that level. Don't break the `add()` into two steps. Don't proceed to the next step in the same operation. Each object should be a single complete concept, a "WHOLE VALUE"[6].

By the same token, there are areas in any domain where detail isn't interesting to the kind of people the software serves. A paint mixer would never want to add red pigment or blue pigment; they combine complete paints, which contain all three pigments. While a paint chemist might need this kind of control, it would involve a whole other analysis, probably producing a much more detailed model of the makeup of paint, than the abstracted pigment color that serves paint mixing. And it is simply irrelevant to anyone involved in the paint mixing application project. Clumping things that don't need to be dissected or rearranged avoids clutter and makes it easier to see the elements that really are meant to recombine.

Therefore,

Decompose design elements (operations, interfaces, classes, and AGGREGATES) into cohesive units, taking into consideration your intuition of the important divisions in the domain. Observe the axes of change and stability through successive refactorings and look for the underlying CONCEPTUAL CONTOURS that explain these shearing patterns. Align the model with the consistent aspects of the domain that make it viable in the first place.

---

[6] The WHOLE VALUE pattern, by Ward Cunningham.

The goal is a simple set of interfaces that combine logically to make sensible statements in the UBIQUITOUS LANGUAGE, and without the distraction and maintenance burden of irrelevant options. This is typically an outcome of refactoring, hard to produce up front. But it doesn't necessarily ever emerge from technically oriented refactoring. It emerges from refactoring toward deeper insight.

Even when the design follows CONCEPTUAL CONTOURS, there will need to be modifications and refactoring. When successive refactoring tend to be localized, not shaking multiple broad concepts of the model, it is an indicator of model fit. Encountering a requirement that forces extensive changes in the breakdown of the objects and methods is a message. Our understanding of the domain needs refinement. It presents an opportunity to deepen the model and make the design more supple.

Example: Accruals

In the Chapter 9, a loan tracking system was refactored based on deeper insight into accounting concepts:

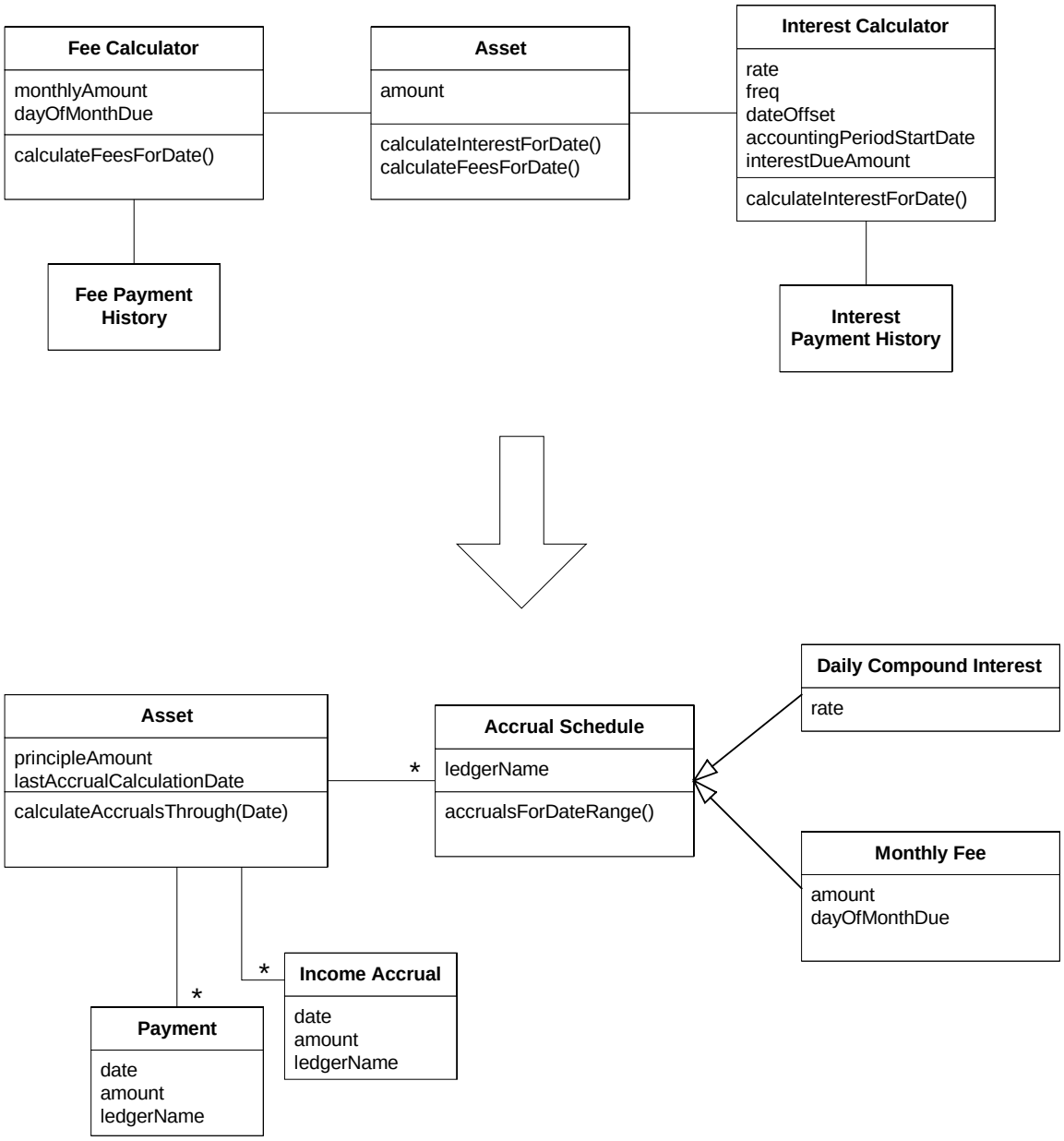


Figure 10. 10

The new model contained only one more object than the old one, yet the partitioning of responsibility had been greatly changed.

Schedules, which had been worked out through case-logic in the “Calculator” classes, were now exploded out into discrete classes for different types of fees and interest. On the other hand, payments of fees and interest, previously kept separate, are now lumped together.

Because of the cohesiveness of the hierarchy of **Accrual Schedules**, the developer believed that this model follows some CONCEPTUAL CONTOURS of the domain. The test will be how it responds to later modification.

Addition of a new kind of Accrual Schedule

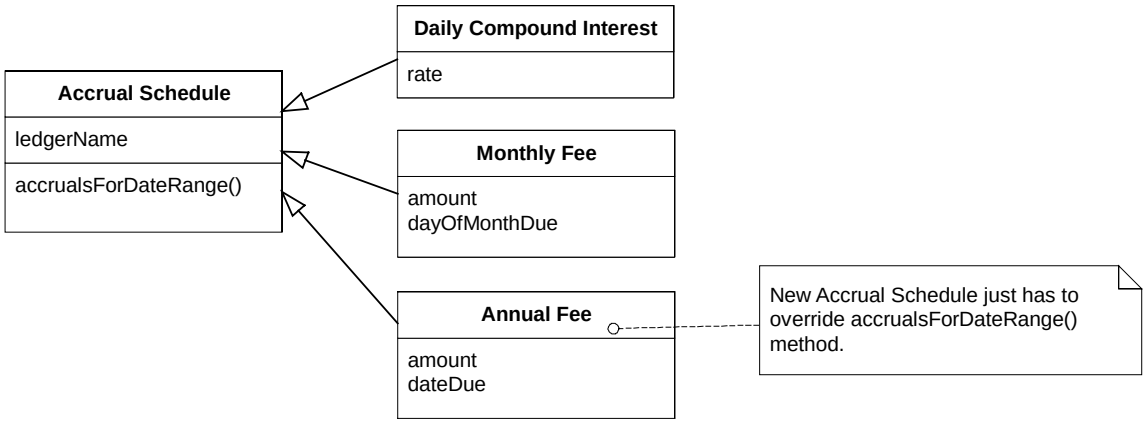


Figure 10. 11

The developer already knew of additional schedules that would be required, so, in addition to making the existing design clearer, she could anticipate that the chosen model would make it easy to extend for the other schedules. There can be no guarantees about how a design will handle unanticipated change, but she thinks it has improved the odds.

An Unanticipated Change

As the project proceeded, a requirement emerged for detailed rules for handling early and late payments. As she studied the problem, the developer was pleased to see that the same rules to payments on interest and for payments on fees. This meant that the new model elements would connect naturally to the single **Payment** class.

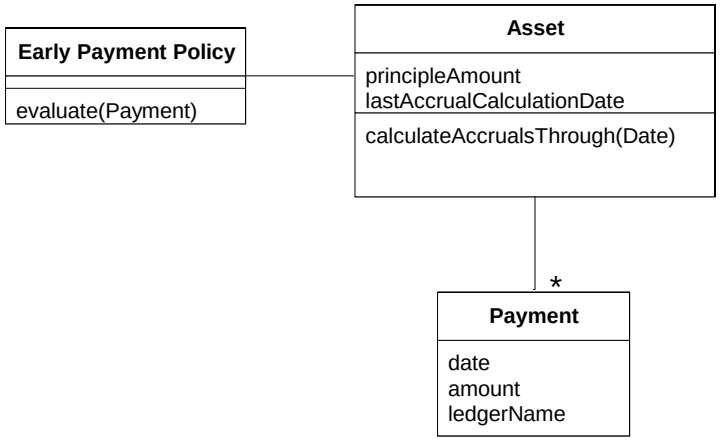


Figure 10. 12

The old design would have forced duplication between the two **Payment History** classes. (This difficulty might have triggered an insight that the **Payment** class should be shared, leading by another path to a similar model.) This ease of extension did not come because she anticipated the change. Nor did it come because she made a design so versatile it could accommodate any conceivable change. It happened because in the previous refactoring, the design was aligned with underlying concepts of the domain.

♦ ♦ ♦

INTENTION REVEALING INTERFACES allow clients to present objects as units of meaning rather than just mechanisms. SIDE-EFFECT FREE FUNCTIONS and ASSERTIONS make it safe to use those units and make complex combinations. Emergence of CONCEPTUAL CONTOURS stabilizes parts of the model and also make the units more intuitive to use and combine.

We still can run into conceptual overload when interdependencies force us to think about too many of these things at a time…

Interdependencies make models and designs hard to understand. They also make them hard to test and maintain. And interdependencies pile up easily.

Every association is, of course, a dependency, and understanding a class requires understanding what it is attached to. Those attached things will be attached to still more things, and they have to be understood too. The type of every argument of every method is also a dependency. So is every return value.

With one dependency, you have to think about two classes at the same time, and the nature of their relationship. With two dependencies, you have to think about each of the three classes, the nature of the class's relationship to each of them, and any relationship they might have to each other. If they in turn have dependencies, you have to be wary of those, also. With three dependencies… It snowballs.

MODULES and AGGREGATES are both aimed at limiting the web of interdependencies. When a highly-cohesive subdomain is carved out into a MODULE, a set of objects are decoupled from the rest of the system so there are a finite number of interrelated concepts. But even a MODULE can be a lot to think about without an almost fanatical commitment to controlling dependencies within it.

Even within a MODULE, the difficulty of interpreting a design increases wildly as dependencies are added. This adds to mental overload, limiting the design complexity a developer can handle. Implicit concepts contribute to this load even more than explicit references.

Refined models are distilled until every remaining connection between concepts represents something fundamental to the meaning of those concepts. In an important subset, the number of dependencies can be reduced to zero, resulting in a class that can be fully understood all by itself, along with a few primitives and basic library concepts.

In every programming environment, a few basics are so pervasive that they are always in mind. For example, in Java development, primitives and a few standard libraries provide basics like numbers, strings and collections. Practically speaking, "integers" don't add to the intellectual load. Beyond that, every additional concept that has to be held in mind in order to understand an object contributes to mental overload.

Implicit concepts, recognized or unrecognized, count just as much as explicit references. While we can generally ignore dependencies on primitive values like integers and strings, we can't ignore *what they represent*. For example, in the first paint mixing examples, the Paint object held three public integers representing red, yellow, and blue color values. The creation of the Pigment Color object did not increase the number of concepts involved or the dependencies. It did make the ones that were already there more explicit and easier to understand. On the other hand, the Collection `size()` operation returns an int that is simply a count, the basic meaning of an integer, so no new concept is implied.

Every dependency is suspect until proven basic to the concept behind the object. It starts with the factoring of the model concepts themselves. Then it requires attention to each individual association and operation. Model and design choices can chip away at dependencies – often to zero.

Low coupling is a fundamental to object design. When you can, go all the way. Eliminate *all* other concepts from the picture. Then the class will be completely self-contained and can be studied and understood alone. Every such self-contained class significantly eases the burden of understanding a MODULE.

Dependencies on other classes within the same module are less harmful than those outside. Likewise when two objects are naturally tightly coupled, multiple operations involving the same pair can actually clarify the nature of the relationship. The goal is not to eliminate all dependencies, but to eliminate all nonessential ones. If every dependency can't be eliminated, each one that is frees the developer to concentrate on the remaining conceptual dependencies.

Try to factor the most intricate computations into STAND ALONE CLASSES, perhaps by modeling value objects held by the more connected classes.

The concept of Paint is fundamentally related to the concept of color. But color, even of pigment, can be considered without paint. By making these two concepts explicit and distilling the relationship, the remaining one-way association says something important, and the Pigment Color class, where most of the computational complexity lies, can be studied *and tested* alone.

♦ ♦ ♦

Low coupling is a basic way to reduce conceptual overload. A STANDALONE CLASS is an extreme of low coupling.

Eliminating dependencies should not mean dumbing-down the model by arbitrarily reducing everything to primitives. The final pattern, CLOSURE OF OPERATIONS, is an example of a technique for reducing dependency while keeping a rich interface…

CLOSURE OF OPERATIONS

> If we take two real numbers and multiply them together, we get another real number. (The real numbers are all the rational numbers and all the irrational numbers.) Because this is always true, we say that the real numbers are "closed under the operation of multiplication": there is no way to escape the set. When you combine any two elements of the set, the result is also included in the set.
>
> -The Math Forum, Drexel University

Of course, there will be dependencies, and that isn't a bad thing when the dependency is fundamental to the concept. Stripping interfaces down to deal with nothing but primitives can impoverish them. But a lot of unnecessary dependencies, and even entire concepts, get introduced at interfaces.

Most interesting objects end up doing things that can't be characterized by primitives alone.

Another common practice in refined designs I refer to as "CLOSURE OF OPERATIONS". The name comes from that most refined of conceptual systems, mathematics. 1+1=2. The addition operation is closed under the set of real numbers. Mathematicians are fanatical about not introducing extraneous concepts, and the property of closure provides them a way of defining an operation without involving any other concepts. We are so accustomed to the refinement of mathematics that it can be hard to grasp how powerful their little tricks are. But this one is used extensively in software designs as well. The basic use of XSLT is to transform one XML document into another XML document. This sort of XSLT operation is closed under the set of XML documents. The property of closure tremendously simplifies the interpretation of an operation, and it is easy to think about chaining together or combining closed operations.

Therefore,

Where it fits, define an operation whose return type is the same as the type of its argument(s). If the implementer's has state that is used in the computation, then the implementer is effectively an argument of the operation, so the argument(s) and return value should be of the same type as the implementer. Such an operation is closed under the set of instances of that type. A closed operation provides a high-level interface without introducing any dependency on other concepts.

This pattern is most often applied to the operations of a VALUE OBJECT. Since the lifecycle of an ENTITY has significance in the domain, so you can't just conjure up a new one to answer a question. There are operations that are closed under an ENTITY type. You could ask an **Employee** object for it's supervisor and get back another **Employee**. But in general, ENTITIES are not the sort of concepts that are likely to be the result of a computation. So, for the most part, this is an opportunity to look for in the VALUE OBJECTS.

An operation can be closed under an abstract type, in which case specific arguments can be of different concrete classes. After all, addition is closed under real numbers, which could be either rational or irrational.

When experimenting, looking for this pattern, you sometimes get half way there. Sometimes the argument can be matched to the implementer, but the return type is different, or the return type matches the receiver and the argument is different. These operations are not fully closed, but they do give some of the advantage of closure. When the extra type is a primitive or basic library class, it frees the mind almost as much as closure.

In the earlier example, the **Pigment Color** `mixedWith()` operation was closed under **Pigment Colors**, and there are several other examples scattered through the book. Here's one more.

Example: Collections

In Java, if you want to select a subset of elements from a **Collection**, you request an **Iterator**. Then you iterate through the elements, testing each one, probably accumulating the matches into a new **Collection**.

```
Set employees = (some Set of Employee objects);
Set lowPaidEmployees = new HashSet();
Iterator it = employees.iterator();
while (it.hasNext()) {
   Employee anEmployee = it.next();
   if (anEmployee.salary() < 40000) lowPaidEmployees.add(anEmployee);
}
```

Conceptually, I've selected a subset of a set. What do I need with this extra concept, **Iterator** and all its mechanical complexity? In Smalltalk, I would call the "select" operation on the **Collection**, passing the test in as an argument. The return would be a new **Collection** containing just the elements that passed the test.

```
employees := (some Set of Employee objects).
lowPaidEmployees := employees select: [:anEmployee | anEmployee salary < 40000].
```

The Smalltalk **Collections** provide other such FUNCTIONS that return **Collections**, which can be of several concrete classes. The operations are not fully closed, since they take a block (a basic library class in Smalltalk) as an argument, but since blocks are a basic library type in Smalltalk, they don't add to the developer's mental load. Because the return value matches the implementer, they can be strung together, like a series of filters. They are easy to write and easy to read. They do not introduce extraneous concepts that are not relevant to the problem of selecting subsets.

♦ ♦ ♦

This collection of patterns should illustrate a general style of design and a way of thinking about it. Making software obvious, predictable and communicative makes abstraction and encapsulation effective. Models can be factored so that objects are simple to use and understand yet still have rich high-level interfaces.

These techniques require fairly advanced design skills to apply and sometimes even to write a client for. The usefulness of a MODEL-DRIVEN DESIGN is sensitive to the quality of the detailed design and implementation decisions, and it only takes a few confused developers to derail a project from the goal.

That said, for the team willing to cultivate its modeling and design skills, these patterns and the way of thinking they reflect yield software that developers can work and rework to create complex software.

Declarative Design

We've discussed assertions, and our relatively informal way of testing them. They can lead to much better designs, but there can't be any real guarantees in handwritten software. To name just one way of evading the ASSERTIONS, there could be additional side effects that were not specifically excluded. No matter how MODEL-DRIVEN our design is, we still end up writing procedures to produce the effect of the conceptual interactions. And we always spend so much of our time writing boilerplate code that doesn't really add any meaning *or* behavior. This is tedious and fraught with error, and the bulk of it obscures the meaning of our model. (Some languages are better than others, but all require us to do a lot of grunt-work.) INTENTION REVEALING INTERFACES and the other patterns in this chapter help, but can never give conventional object-oriented programs formal rigor.

These are some of the motivations behind declarative design. This term means many things to many people, but usually it indicates a way to write a program, or some part of it, as a kind of executable specification. A very precise description of properties actually controls the software. In its various forms this could be done through a reflection mechanism or at compile-time through code generation (producing conventional code automatically, based on the declaration). This allows another developer to take the declaration at face-value. It is an absolute guarantee.

This is a kind of Holy Grail of model-driven design. It does have its pitfalls in practice. For example, just two particular problems I've encountered more than once are

- a declaration language not expressive enough to do everything needed, but a framework that makes it very difficult to extend the software beyond the automated portion.
- code generation techniques that cripple the iterative cycle by merging generated code into hand-written code in a way that makes regeneration very destructive.

Oddly, these attempts at declarative design often ultimately end in the dumbing-down of the model and application, as developers, trapped by the limitations of the framework, enact design triage in order to get *something* delivered.

I've seen the greatest value when a narrowly scoped framework automates a particular tedious and error-prone aspect of the design, such as persistence and object-relational mapping. The best of these unburden the developer of drudgework while leaving them complete freedom to design.

<<Sidebar>>

Rule-based programming with an inference engine and declarative rule base aims at these ideals too, and, as I've mentioned in other chapters, presents an enticing approach to domain-driven design. It also provides an example of how subtle the issues can be.

While a rules based program is declarative in principle, most systems have added "control predicates" that were added to allow performance tuning. This control code introduces side-effects, and means that the behavior is no longer dictated completely by the declared rules. Adding, removing or reordering the rules can cause unexpected, incorrect results.

Many declarative approaches can be corrupted if the developers bypass them intentionally or unintentionally. This is likely when the system is difficult to use or overly restrictive. Everyone has to follow the rules of the framework in order to get the benefits of a declarative program.

<<End sidebar>>

Domain-Specific Languages

An interesting approach that is sometimes declarative is the "domain specific language". Client code is written in a programming language tailored to a particular model of a particular domain. A language for shipping systems might include terms like cargo and route, along with syntax for associating them. The

program is then compiled, often into a conventional object-oriented language, where a library of classes provides implementations for the terms in the language.

In such a language, programs can be extremely expressive, and make the strongest connection with the UBIQUITOUS LANGUAGE. This is an exciting concept, but domain-specific languages do have their drawbacks in the approaches I've seen based on object-oriented technology.

In order to refine the model, a developer needs to have the capability of modifying the language. This may involve modifying grammar declarations and other language interpreting features as well as modifying underlying class libraries. I don't object to developers learning advanced technology and concepts, but this there is value in the seamlessness of an application and model implemented in the same language. To make matters worse, it can be difficult to refactor client code to conform to the new model and the domain-specific language, since there are no refactoring tools and an indirect mapping of language semantics. Of course, someone may come up with a technical fix for the refactoring problems.

This technique might be most useful for very mature models, perhaps where client code is being written by a different team. Generally, such setups lead to the poisonous distinction between highly technical framework builders and technically unskilled application builders, but it wouldn't have to.

In the scheme programming language, something very similar is part of standard programming style, so that the expressiveness of a domain-specific language can be created without bifurcating the system.

Declarative Style of Design

Once your design has INTENTION-REVEALING INTERFACES, SIDE-EFFECT-FREE FUNCTIONS, and ASSERTIONS, you are edging into this territory. Many of the benefits of declarative design are obtained once you have combinable elements that communicate their meaning.

A supple design can make it possible for the client code to use a declarative *style* of design.

Extending SPECIFICATIONS in a Declarative Style

In Chapter 9 we covered the basic concept of SPECIFICATION, the roles it can play in a program, and some sense of what is involved in implementation. Now let's take a look at a few bells and whistles that can be very useful in some situations with complicated rules.

SPECIFICATION is an adaptation of an established formalism, the predicate. Predicates have other useful properties that we can draw on, selectively.

Combining Specifications Using Logical Operators

When using SPECIFICATIONS, you quickly come across situations in which you would like to combine them. As mentioned above, a SPECIFICATION is an example of a predicate, and predicates can be combined and modified with the operations "and", "or" and "not". These logical operations are closed under predicates, so SPECIFICATION combinations will exhibit CLOSURE OF OPERATIONS.

As significant generalized capability is built into SPECIFICATIONS, it becomes very useful to create an abstract class or interface that can be used for SPECIFICATIONS of all sorts. This means typing arguments as some high-level abstract class.

```
public interface Specification {
   boolean isSatisfiedBy(Object candidate);
}
```

This calls for a guard clause at the beginning of the method, but otherwise does not affect functionality. For example, the **Container Specification** (from the example in Building-to-Order in Chapter 9) would be modified this way.

```
    public class ContainerSpecification implements Specification {
       private ContainerFeature requiredFeature;

       public ContainerSpecification(ContainerFeature requiredFeature) {
          this.requiredFeature = requiredFeature;
       }

       boolean isSatisfiedBy(Object candidate){
          if (!candidate instanceof Container) return false;

          return (Container)aContainer.getFeatures().contains(requiredFeature);
       }
}
```

Now, let's extend the **Specification** interface by adding the three new operations:

```
    public interface Specification {
       boolean isSatisfiedBy(Object candidate);

       Specification and(Specification other);
       Specification or(Specification other);
       Specification not();
    }
```

Recall that some **Container Specifications** were configured to require ventilated **Containers** and others to require armored **Containers**. A chemical that is both volatile *and* explosive would, presumably, need *both* of these SPECIFICATIONS. Easily done, using the new methods.

```
    Specification ventilated = new ContainerSpecification(VENTILATED);
    Specification armored = new ContainerSpecification(ARMORED);

    Specification both = ventilated.and(armored);
```

This would have required a more complicated **Container Specification**, and would still be special purpose.

Suppose we had more than one kind of ventilated **Container**. It might not matter for some items which kind they were packed into. They could be placed in either type.

```
    Specification ventilatedType1 =
          new ContainerSpecification(VENTILATED_TYPE_1);
    Specification ventilatedType2 =
          new ContainerSpecification(VENTILATED_TYPE_2);

    Specification either = ventilatedType1.or(ventilatedType2);
```

If it was considered wasteful to store sand in specialized containers, we could prohibit it by SPECIFYING a container with no special features.

```
    Specification cheap = (ventilated.not()).and(armored.not());
```

This would have prevented some of the suboptimal behavior of the prototype warehouse packer discussed in Chapter 9.

The ability to build complex specifications out of simple elements increases the expressiveness of the code. The combinations are written in a declarative style.

Depending on how SPECIFICATIONS are implemented, these operators may be easy or difficult to provide. What follows is a very simple implementation, which would be inefficient in some situations, and quite practical in others. It is meant as an *explanatory example*. Like any pattern, there are many ways to implement it.

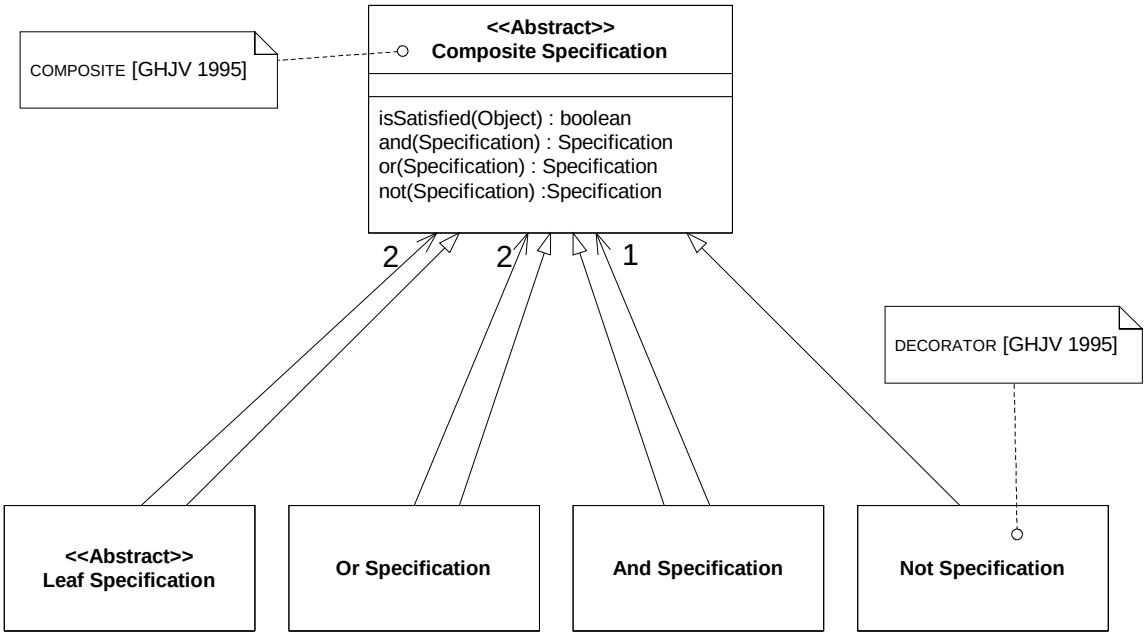


Figure 10. 13 COMPOSITE Design of SPECIFICATION

```
public abstract class AbstractSpecification implements Specification {
   public Specification and(Specification other) {
      return new AndSpecification(this, other);
   }
   public Specification or(Specification other) {
      return new OrSpecification(this, other);
   }
   public Specification not() {
      return new NotSpecification(this);
   }
}

public class AndSpecification extends AbstractSpecification {
   Specification one;
   Specification other;
   public AndSpecification(Specification x, Specification y) {
      one=x;
      other=y;
   }
   public boolean isSatisfiedBy(Object candidate) {
      return one.isSatisfiedBy(candidate)
                && other.isSatisfiedBy(candidate);
   }
}
```

```
public class OrSpecification extends AbstractSpecification {
   Specification one;
   Specification other;
   public OrSpecification(Specification x, Specification y) {
      one=x;
      other=y;
   }
   public boolean isSatisfiedBy(Object candidate) {
      return one.isSatisfiedBy(candidate)
                  || other.isSatisfiedBy(candidate);
   }
}

public class NotSpecification extends AbstractSpecification {
   Specification wrapped;

   public NotSpecification(Specification x) {
      wrapped=x;
   }
   public boolean isSatisfiedBy(Object candidate) {
      return !wrapped.isSatisfiedBy(candidate);
   }
}
```

This code was written to be as simple as possible. As I said, there may be situations in which this is inefficient. However, other implementation options are possible that would minimize object count or boost speed, or perhaps be compatible with idiosyncratic technologies present in some project. The important thing is a model that captures the key concepts of the domain, along with an implementation that is faithful to that model. That leaves a lot of room to solve performance problems.

There are also cases where this full generality is not needed. In particular, "and" tends to be used a lot more than the others, and also tends to create less implementation complexity. Don't be afraid to implement only "and", if that is all you need.

Way back in Chapter 2, in the example dialog, the developers had apparently not implemented the "satisfied by" behavior of their SPECIFICATION. Up to that point, the SPECIFICATION had been used only for building-to-order. Even so, the abstraction was intact, and adding functionality was relatively easy. Using a pattern doesn't mean building features you don't need. They can be added later, as long as the concepts don't get muddled.

<<SIDEBAR: One Alternative Implementation>>

Some implementation environments don't accommodate very fine-grain objects very well. I once worked on a project with an object database that insisted on giving and tracking an object id to every object. Each object had lot of overhead in space and performance, and total address space was a limiting factor. I applied SPECIFICATION pattern to important points in the domain design, which I think was a good decision. But I used a slightly more elaborate version of the implementation described in this chapter, which was definitely a mistake. It resulted in millions of very fine-grained objects that contributed to bogging the system down.

Here is an example of an alternative implementation that encodes the composite SPECIFICATION as a string or array encoding the logical expression, to be interpreted at runtime.

(Don't worry if you do not see how you would implement this. The important thing is to realize that there are many ways of implementing a SPECIFICATION with logical operators, and so if the simple one is not practical in your situation, you have options.)

SPECIFICATION Stack Content for "cheap container"	
Top	AndSpecificationOperator (FLY WEIGHT)
	NotSpecificationOperator (FLY WEIGHT)
	Armored
	NotSpecificationOperator
	Ventilated

When you want to test a candidate, you have to interpret this structure, which could be done by popping each element off, evaluating it or popping the next off as required by an operator. You would end up with

```
and(not(armored), not(ventilated))
```

+ Low object count

+ Efficient use of memory

- Requires more sophisticated developers

<<end sidebar>>

Subsumption

This final feature is not usually needed and can be difficult to implement, but it can be useful, and it elucidates the meaning of a SPECIFICATION.

Consider the chemical warehouse packer example again. Recall that each **Chemical** had a **Container Specification** and the **Packer** SERVICE guaranteed that all these would be satisfied when **Drums** are assigned to **Containers**. All is well…until they change the regulations.

Every few months a new set of rules is issued, and our users would like to be able to produce a list of the chemical types that now have more stringent requirements.

Of course, we could give a partial answer (and one the users probably also want) by running a validation of each **Drum** in the inventory, with the new SPECIFICATIONS in place, and finding all those that no longer meet the SPEC. This would tell the users which **Drums** in the existing inventory they needed to move.

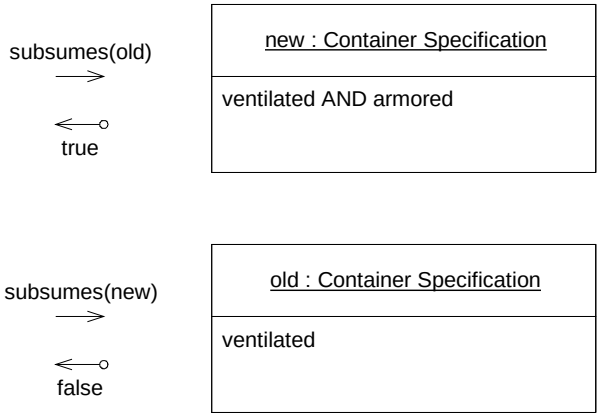
But what they *asked for* was a list of chemicals whose handling has become more stringent. Perhaps there are none in-house right now, or perhaps they just happened to be packed into a more stringent container. In either case, the report above would not list them.

Let's introduce a new operation for comparing two SPECIFICATIONS directly to each other.

```
boolean subsumes(Specification other);
```

A more stringent SPEC subsumes a less stringent one. It could take its place without any previous requirement being neglected.

Figure 10. 14 Container Specification for gasoline has been tightened



In the language of SPECIFICATION, we would say that the new SPECIFICATION "subsumes" the old SPECIFICATION because any candidate that would satisfy the new SPEC would also satisfy the old. Not necessarily the other way around.

If each of these SPECIFICATIONS is viewed as a predicate, subsumption is equivalent to logical implication. Using a conventional notation, a → b means that statement “a” implies statement “b”, so that if “a” is true, “b” is also true. (Not necessarily the other way around, though.)

To apply this to our container matching needs, when a SPECIFICATION is being changed we would like to know if the proposed new SPEC meets all the conditions of the old one.

New Spec → Old Spec (if the new spec is true, the old is also true)

Proving a logical implication in a general way is very difficult, but special cases can be easy. For example, particular parameterized SPECS can define their own subsumption rule.

```
public class MinimumAgeSpecification {
   int threshold;

   public boolean isSatisfiedBy(Person candidate) {
      return candidate.getAge() >= threshold;
   }

   public boolean subumes(MinimumAgeSpecification other) {
      return threshold >= other.getThreshold();
   }
}
```

So that a JUnit test might contain

```
drivingAge = new MinimumAgeSpecification(16);
votingAge = new MinimumAgeSpecification(18);
assertTrue(votingAge.subsumes(drivingAge));
```

Another practical special case, one suited to address the **Container Specification** problem, is a SPECIFICATION interface combining “subsumption” with the single logical operator, "and".

```
public interface Specification {
   boolean isSatisfiedBy(Object candidate);
   Specification and(Specification other);
   boolean subsumes(Specification other);
```

```
}
```

Proving implication with only the “and” operator is simple.

a and b → a

or, in a more complicated case

a and b and c → a and b

So if the **Composite Specification** is able to collect together all the leaf SPECIFICATIONS that are "anded" together, then all we have to do is check that the subsuming SPECIFICATION has all the leaves that the subsumed one has, and maybe some additional criteria.

```
public boolean subsumes(Specification other) {
   if (other instanceof CompositeSpecification) {
      Iterator it =
         (CompositeSpecification)other.leafSpecifications().iterator();
      while (it.hasNext()) {
         if (!leafSpecifications().contains(it.next())) return false;
      }
   } else {
      if (!leafSpecifications().contains(other)) return false;
   }
   return true;
}
```

This interaction could be enhanced to compare carefully chosenparameterized leaf SPECIFICATIONS and some other complications. Unfortunately, when "or" and "not" are included, these proofs become much more involved. In most situations it is best to avoid such complexity by making a choice, either forgoing some of the operators or foregoing subsumption. If both are needed, consider carefully if the benefit is great enough to justify the difficulty.

Socrates on SPECIFICATIONS

All men are mortal.	`Specification manSpec = new ManSpecification();`
	`Specification mortalSpec = new MortalSpecification();`
	`assert manSpec.subsumes(mortalSpec);`

Socrates is a man.	`Man socrates = new Man();` `assert manSpec.isSatisfiedBy(socrates);`
Therefore, Socrates is mortal.	`assert mortalSpec.isSatisfiedBy(socrates)`

Carve off Subdomains

You don't have to tackle the whole design at once. Pick away at it. At first you may see a part of the model that is a specialized math and separate that. Maybe you pull error handling into a framework. With each such step, not only is the new module clean, but the part left behind is smaller and clearer because a part of it is written in a declarative style, a declaration in terms of the special math or error handler.

This issue is taken up again in Chapter 15.

Draw on Established Formalisms, When You Can

Many business applications involve accounting, for example. Accounting has a well-developed set of ENTITIES and rules that make for an easy adaptation to a deep model and a supple design.

There are many such formalized conceptual frameworks, but my personal favorite is specialized math. Many domains have math in them somewhere. Look for it. Dig it out. Specialized math is clean, combinable by clear rules, and people find it easy to understand. One that I've used in the past is the "Shares Math" example that ends this chapter.

Example Integrating the Patterns: Shares Math

Back in Chapter 8, I introduced the example of a syndicated loan system. One basic requirement of that application was that when the borrower makes a principle payment, the money is, by default, prorated according to the lenders' shares in the loan.

Initial Design for Payment Distribution

(As it is refactored, this code will get easier to understand, so don't get stuck on this version.)

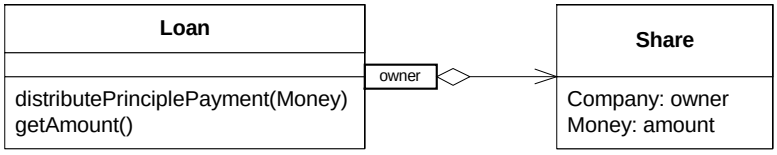


Figure 10. 15

```
public class Loan {
   private Map shares;

   //Accessors, constructors, and very simple methods are being excluded.

   public Map distributePrinciplePayment(double paymentAmount) {
      Map paymentShares = new HashMap();
      Map loanShares = getShares();
      double total = getAmount();
      Iterator it = loanShares.keySet().iterator();
      while(it.hasNext()) {
         Object owner = it.next();
```

```
            double initialLoanShareAmount = getShareAmount(owner);
            double paymentShareAmount =
               initialLoanShareAmount/total*paymentAmount;
            Share paymentShare = new Share(owner, paymentShareAmount);
            paymentShares.put(owner, paymentShare);

            double newLoanShareAmount =
               initialLoanShareAmount-paymentShareAmount;
            Share newLoanShare = new Share(owner, newLoanShareAmount);
            loanShares.put(owner, newLoanShare);
         }
         return paymentShares;
      }

      public double getAmount() {
         Map loanShares = getShares();
         double total = 0.0;
         Iterator it = loanShares.keySet().iterator();
         while(it.hasNext()) {
            Share loanShare = (Share)loanShares.get(it.next());
            total = total + loanShare.getAmount();
         }
         return total;
      }
   }
```

Separating Commands and Side-effect-free Functions

This design already has intention revealing interfaces. But the `distributePaymentPrinciple()` method does a dangerous thing: It calculates the shares for distribution, and also modifies the **Loan**. Let's refactor to separate the query from the modifier.

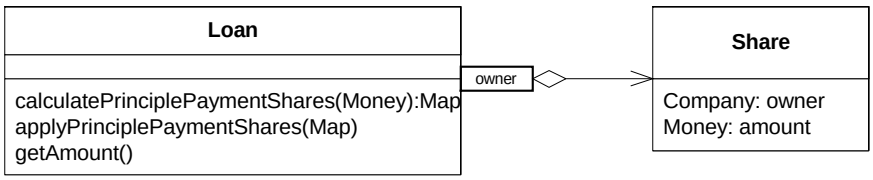


Figure 10. 16

```
   public void applyPrinciplePaymentShares(Map paymentShares) {
      Map loanShares = getShares();
      Iterator it = paymentShares.keySet().iterator();
      while(it.hasNext()) {
         Object lender = it.next();
         Share paymentShare = (Share)paymentShares.get(lender);
         Share loanShare = (Share)loanShares.get(lender);
         double newLoanShareAmount = loanShare.getAmount()-paymentShare.getAmount();
         Share newLoanShare = new Share(lender, newLoanShareAmount);
         loanShares.put(lender, newLoanShare);
      }
   }
   public Map calculatePrinciplePaymentShares(double paymentAmount) {
```

```
    Map paymentShares = new HashMap();
    Map loanShares = getShares();
    double total = getAmount();
    Iterator it = loanShares.keySet().iterator();
    while(it.hasNext()) {
       Object lender = it.next();
       Share loanShare = (Share)loanShares.get(lender);
       double paymentShareAmount = loanShare.getAmount()/total*paymentAmount;
       Share paymentShare = new Share(lender, paymentShareAmount);
       paymentShares.put(lender, paymentShare);
    }
    return paymentShares;
}
```

Client code now looks like this.

```
Map distribution = aLoan.calculatePrinciplePaymentShares(paymentAmount);
aLoan.applyPrinciplePaymentShares(distribution);
```

Not too bad. The FUNCTIONS have encapsulated a lot of complexity behind INTENTION REVEALING INTERFACES. But the code does begin to multiply some when we add the "`applyDrawdown()`", "`calculateFeePayentShares()`", etc. Each extension complicates it and weighs it down with more. This might be a point where the granularity is too coarse. The conventional approach would be to break the calculation methods down into subroutines. That could well be a good step along the way, but we ultimately want to see the underlying conceptual boundaries and deepen the model. The elements of a design with such a concept contouring grain could be combined to produce the needed variations.

Making an Implicit Concept Explicit

There are enough pointers now to start probing for that new model. The **Share** objects are passive in this implementation, and they are being manipulated in complex low-level ways. This is because most of the rules and calculations about shares don't apply to single shares, but to groups of them. There is a missing concept – shares are related to each other as making up a part of a whole. Making this concept explicit will let us express those rules and calculations more succinctly.

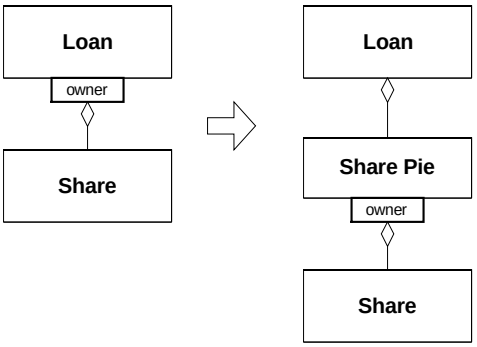


Figure 10. 17

The **Share Pie** represents the total distribution of a specific loan. It is an ENTITY whose identity is local within the AGGREGATE of the **Loan**. The actual distribution calculations can be delegated to the **Share Pie**.

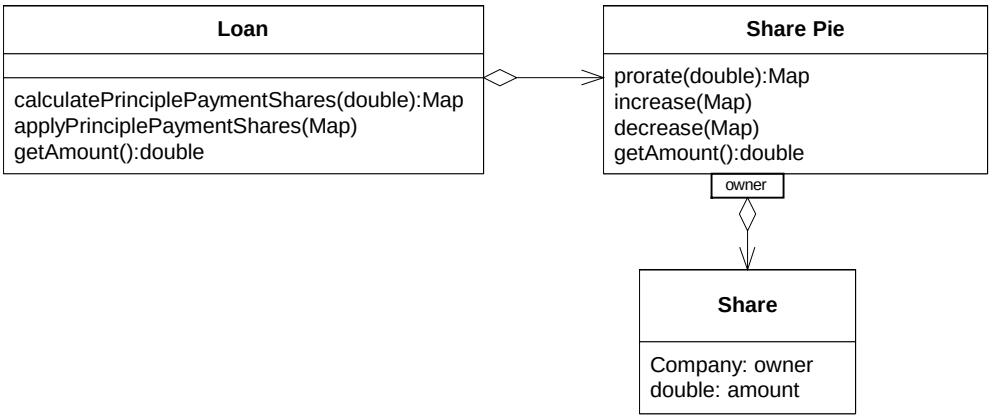


Figure 10. 18

```
public class Loan {
   private SharePie shares;

   //Accessors, constructors, and straight-forward methods are omitted.

   public Map calculatePrinciplePaymentDistribution(double paymentAmount) {
      return getShares().prorated(paymentAmount);
   }
   public void applyPrinciplePayment(Map paymentShares) {
      shares.decrease(paymentShares);
   }
}
```

The **Loan** is simplified, and the **Share** calculations are centralized in a VALUE OBJECT focused on that responsibility. Still, the calculations haven't really become more versatile or easier to use.

Share Pie Becomes a VALUE OBJECT: Cascade of Insights

Often, the hands-on experience of implementing a new design will trigger a new insight into the model itself. In this case, the tight coupling of the **Loan** and **Share Pie** seems to be obscuring the relationship of the **Share Pie** and the **Shares**. What would happen if we made **Share Pie** a VALUE OBJECT?

This would mean that `increase(Map)` and `decrease(Map)` would not be allowed because the **Share Pie** would be immutable. The whole **Pie** would have to be replaced to change its value. So you could have an operations like `addShares(Map)` that would return a whole new, larger Share Pie.

Let's go all the way to CLOSURE OF OPERATIONS. Instead of "increasing" a **Share Pie** or adding **Shares** to it, just add two **Share Pies** together and the result is the new larger **Share Pie**.

We can close the `prorate(double)` operation over **Share Pie** just by changing the return type. "Shares Math" starts to take shape, initially with four operations.

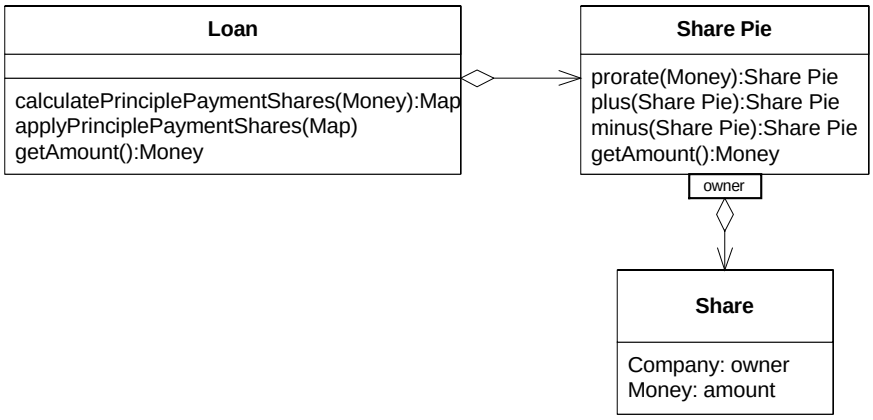


Figure 10. 19

We can make some well-defined ASSERTIONS about our new VALUE OBJECTS, the **Share Pies**.

```
public class SharePie {
   private Map shares = new HashMap();

   //Again, accessors and other straight-forward methods are omitted.

   public double getAmount() {
      double total = 0.0;
      Iterator it = shares.keySet().iterator();
      while(it.hasNext()) {
         Share loanShare = getShare(it.next());
         total = total + loanShare.getAmount();
      }
      return total;
 }

public SharePie minus(SharePie otherShares) {
   SharePie result = new SharePie();
   Set owners = new HashSet();
   owners.addAll(getOwners());
   owners.addAll(otherShares.getOwners());
   Iterator it = owners.iterator();
   while(it.hasNext()) {
      Object owner = it.next();
      double resultShareAmount = getShareAmount(owner) –
         otherShares.getShareAmount(owner);
      result.add(owner, resultShareAmount);
   }
   return result;
}

public SharePie plus(SharePie otherShares) {
   //Similar to implementation of minus()
}
```

The whole is equal to the sum of its parts.

The difference between two Pies is the difference between each owner's share.

The combination of two Pies is the combination of each owner's share.

```
public SharePie prorated(double amountToProrate){
   SharePie proration = new SharePie();
   double basis = getAmount();
   Iterator it = shares.keySet().iterator();
   while(it.hasNext()) {
      Object owner = it.next();
      Share share = getShare(owner);
      double proratedShareAmount = share.getAmount()/basis * amountToProrate;
      proration.add(owner, proratedShareAmount);
   }
   return proration;
}
}
```

An amount can be divided proportionately among all share holders

The Suppleness of the New Design

At this point, the methods in the all-important **Loan** class could be as simple as

```
public class Loan {
   private SharePie shares;

   //Accessors, constructors, and straight-forward methods are omitted.

   public SharePie calculatePrinciplePaymentDistribution(double paymentAmount) {
      return getShares().prorated(paymentAmount);
   }
   public void applyPrinciplePayment(SharePie paymentShares) {
      setShares(shares.minus(paymentShares));
   }
```

Each of these short methods states its *meaning*. Applying a principle payment means that you subtract the payment from the loan, share by share. Distributing a principle payment is done by dividing the amount *prorata* among the shareholders. The design of the **Share Pie** has allowed us to use a declarative style in the **Loan** code, producing code that begins to read like a conceptual definition of the business transaction, rather than a calculation.

Other transaction types (too complicated to list before) can be declared easily. For example, loan drawdowns are divided among lenders based on their shares of the **Facility**. The new drawdown is added to the outstanding **Loan**. In our new domain language:

```
public class Facility {
   private SharePie shares;
   …
   public SharePie calculateLoanDrawdownDistribution(double drawdownAmount) {
      return shares.prorated(paymentAmount);
   }
}

public class Loan {
   …
   public void applyLoanDrawdown(SharePie drawdownShares) {
      setShares(shares.plus(drawdownShares));
   }
}
```

To see the deviation of each lender from its agreed contribution, take the theoretical distribution of the outstanding **Loan** amount and subtract it from the actual **Loan** shares:

```
SharePie originalAgreement = aFacility.getShares().prorated(aLoan.getAmount());
SharePie actual = aLoan.getShares();
SharePie deviation = actual.minus(originalAgreement);
```

Certain characteristics of the **Share Pie** design make for this easy recombination and communication.

- Complex logic encapsulated in specialized VALUE OBJECTS

  Since **Share Pies** are VALUE OBJECTS, these math operations can create new instances which we can freely use to replace outdated instances.

- SIDE-EFFECT-FREE FUNCTIONS

  None of the **Share Pie** methods causes any change to any existing object. This allows us to use `plus()`, `minus()`, and `prorated()` freely in intermediate calculations, combining them, expecting them to do what their names suggest, and nothing more. It also allows us to build analytical features based on the same methods. (Before, they could only be called when an actual distribution was made, since the data would change after each call.)

- CLOSURE OF OPERATIONS

  Each **Share Pie** operation produces either another **Share Pie** or a simple amount. As a result, the **Share Pie** is self-contained, easily understood, easily tested, and easily combined (e.g. the deviation is the actual pie minus the loan amount prorated based on the facility shares).

Pulling out the part of the problem that corresponded to the formalism of math, we arrived at a supple design for shares that further distills the core **Loan** and **Facility** methods. (Chapter 15 will delve further into distillation of a CORE DOMAIN.)

♦ ♦ ♦

Supple design has a profound effect on the ability of software to cope with change and complexity. The impact can go beyond a specific modeling and design problem. Chapter 15, Distillation, will discuss the strategic value of supple design as one of several tools for distilling a domain model to make large and complex projects more tractable.

11.Applying Analysis Patterns

Deep models and supple designs don't come easy. Progress comes from lots of learning about the domain, lots of talking, and lots of trial and error. Sometimes, though, we can get a leg up.

When an experienced developer looking at a domain problem sees a familiar sort of responsibility or a familiar web of relationships, he or she can draw on the memory of how the problem was solved before. What models were tried and which worked? What difficulties arose in implementation and how were they resolved? The trial and error of that earlier experience is suddenly relevant to the new situation. Some of these patterns have been documented and shared, allowing the rest of us to draw on the accumulated experience.

In contrast to the fundamental building block patterns presented in Part II, and the supple design principles of Chapter 10, these patterns are higher-level and more specialized, involving the use of a few objects to represent some concept. They let us cut through expensive trial and error to start with a model that is already expressive and implementable and addresses subtleties that might be costly to learn. From that starting point we refactor and experiment. These are not out-of-the-box solutions.

In *Analysis Patterns: Reusable Object Models*, Martin Fowler, defined his patterns this way.

> Analysis patterns are groups of concepts that represent a common construction in business modeling. It may be relevant to only one domain or it may span many domains. [Fowler1997, p. 8]

The analysis patterns Fowler presents are the result of experience in the field, and so they are practical, in the right situation. Such patterns provide someone facing a challenging domain with very valuable starting points for their iterative development process. The name emphasizes their conceptual nature. These are not technological solutions; they are guides to help you work out a model in a particular domain.

What the name unfortunately does *not* convey is that there is significant discussion of implementation, including some code. Fowler understands the pitfalls of analysis without thought for practical design. Here is an interesting example where he is looking even beyond deployment, to the implications of specific model choices on the long-term maintenance of the system in the field.

> … When we build a new [accounting] practice, we create a network of new instances of the posting rule. We can do this without any recompilation or rebuilding of the system, while it is still up and running. There will be unavoidable occasions when we need a new subtype of posting rule, but these will be rare. [p. 151]

On a mature project, model choices are often informed by experience with the application. Multiple implementations of various components will have been tried. Some of these will have been carried into production and even faced the maintenance phase. Many problems can be avoided when such experience is available. Analysis patterns at their best can carry that kind of experience from other projects, combining model insights with extensive discussions of design directions and implementation consequences. To discuss model ideas out of that context makes them harder to apply and risks opening the deadly divide between analysis and design, which is the primary motivation of MODEL-DRIVEN DESIGN.

The principle and application of analysis patterns can be better explained by example than through abstract explanations, so this chapter will give two examples describing developers making use of a small representative sample of models from the chapter "Inventory and Accounting" in [Fowler 1997]. The analysis patterns will be summarized just enough to support the examples. This is obviously not an attempt to catalog patterns of this kind or even to fully explain the sample patterns. The point is to illustrate their integration into the domain driven design process.

Example: Earning Interest with Accounts

Chapter 10, showed two possible ways that a developer might search for a deeper model for a particular application. Here is yet a third scenario. This time, the developers will mine *Analysis Patterns* for useful ideas. separate examples treated different scenarios in which an awkward design was improved by defining explicit concepts to deepen the model. Here is a third scenario.

To review, an application for tracking loans and other interest bearing assets calculates the interest and fees generated and tracks payments from the borrower. A nightly batch process takes those figures and passes them to the legacy accounting system, indicating the specific ledger each amount should be posted to. The design works, but is awkward to use, tricky to change and does not communicate well.

Initial class diagram

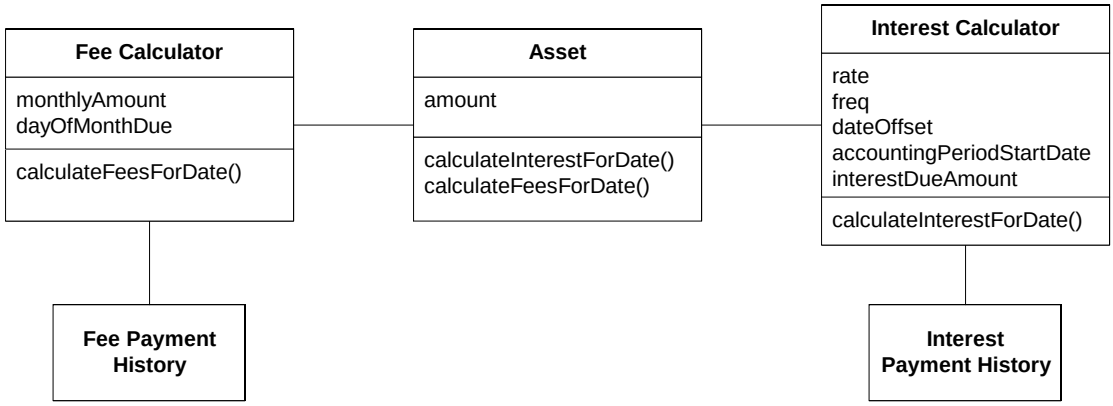


Figure 11. 1

The developer (**DEV1**) decides to read *Analysis Patterns* Chapter 6, “Inventory and Accounting”. Here is a summary of the part she found most relevant.

<<Some kind of box starts here>>

Accounting Models in *Analysis Patterns*

Business applications of all sorts track “accounts”, which hold things of value, typically money. In a lot of applications, it isn’t enough to keep track of the amount in an account. It is essential to account for and control each change to that amount. That is the motivation for the most basic of the accounting models.

Account Model

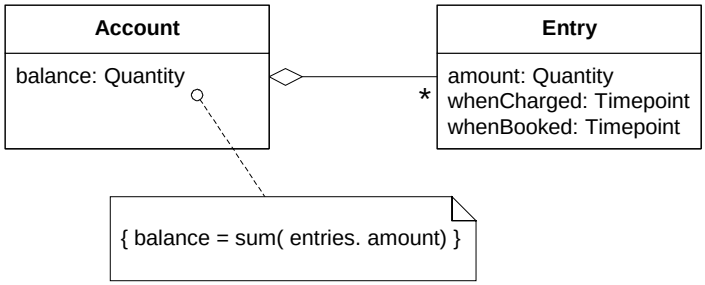


Figure 11. 2

Value can be added by inserting an "**Entry**". Value can be removed by inserting a negative **Entry**. **Entries** are never removed, so the whole history is retained. The balance is the combined effect of all **Entries**. This balance could be computed on demand or cached, an implementation decision that is encapsulated by the **Account** interface.

A basic principle of accounting is "conservation". Money doesn't appear out of nowhere, nor does it disappear without a trace. It is only moved from one **Account** to another.

Transaction Model

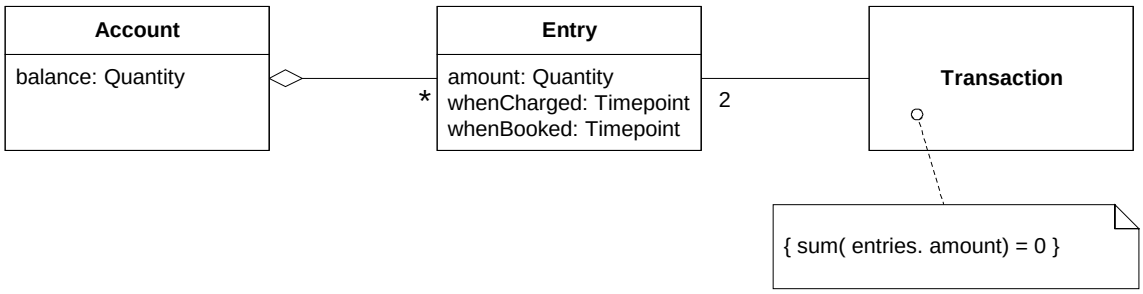


Figure 11. 3

This is the basic concept of "double-entry bookkeeping" – every credit has a matching debit – a well-established way of maintaining consistency. Of course, like other conservation principles, it only applies to a closed system, one that includes all sources and sinks. Many simple applications do not require this rigor.

Fowler includes more elaborate forms of these models and considerable discussion of the tradeoffs.

<<End of box>>

This reading gives her several new ideas. She shows the chapter to a colleague (**DEV2**) who has been working on some of the interest calculation logic with her and who wrote the nightly batch program. Together, they rough out a change to their model, incorporating some of the model elements they've read about.

New model proposal

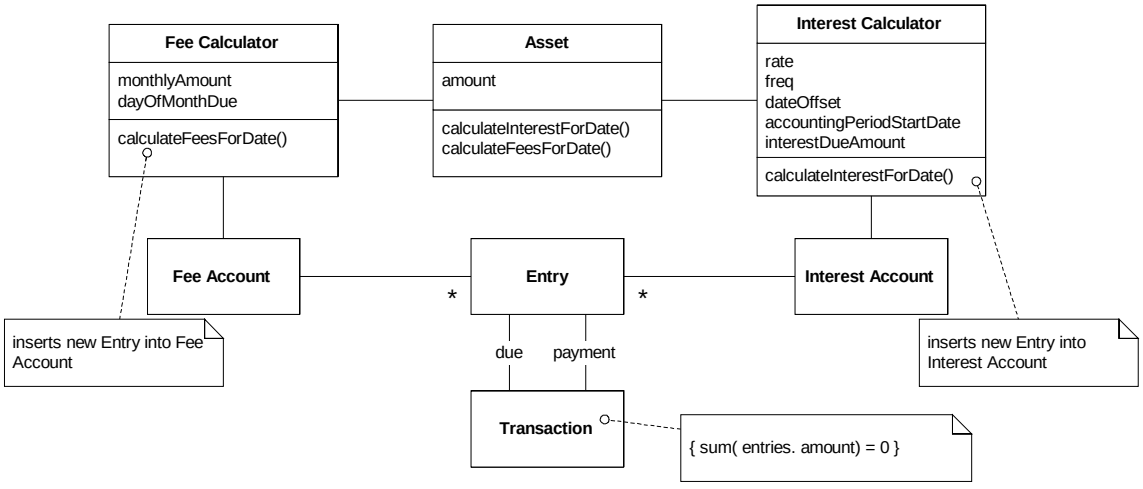


Figure 11. 4

Then they pull in their domain expert (**EXP**) for a discussion of their new model ideas.

DEV1: With this new model, we make an "**Entry**" into the **Interest Account** for the interest earned, rather than just adjusting the interestDueAmount. Then, another entry for the payment balances it out.

EXP: So now we'd be able to see a history of all the interest accruals as well as the payment history? That's something we've been wanting.

DEV2: I'm not sure we've used "**Transaction**" quite right. The definition talks about moving money from one Account to another, not two entries that balance each other in the same account.

DEV1: That's a good point. I was also worried that the book seems to make quite a point about the transaction being created all at once. The interest payments can be several days late.

EXP: Those payments aren't necessarily late. There is a lot of flexibility in when they pay.

DEV1: So this may be a blind alley. I was thinking we might have identified some implicit concepts. Having the **Interest Calculator** create **Entry** objects does seem to communicate better. And **Transaction** seemed to neatly tie together the calculated interest with the payment.

EXP: Why do we need to tie together the accrual to the payment? They are separate postings in the accounting system. The balance on the **Account** is the main thing. Along with the individual **Entries** we really have what we need.

DEV2: You mean you don't track whether they've made the interest payment?

EXP: Well, of course we do. But it isn't as simple as this one-accrual/one-payment scheme of yours.

DEV2: It could actually simplify a lot of things to stop worrying about that connection.

DEV1: Ok, how about this? [Takes copy of *old* class diagram and starts sketching modifications]

By the way, you used the word "accruals" a few times. Could you clarify what it means?

EXP: Sure. An "accrual" is just when you account for an expense or income at the time it is incurred, never mind when money actually changes hands. So, we accrue interest every day, but at the end of the month (for example) we receive a payment against it.

DEV1: Yes, we really needed a word like that. Ok, how does this look?

Original class diagram, accruals separated from payment

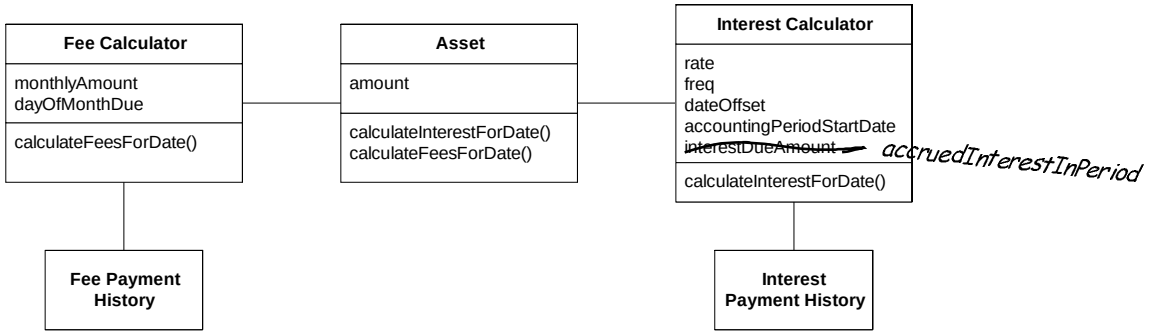


Figure 11. 5

Now we can get rid of all the complications that were in the calculator from relating payments, and we've introduced the term "accruals", which reveals the intent better.

EXP: So we're not going to have the **Account** object? I was looking forward to being able to see everything together there, with the accruals and the payments and a balance.

DEV1: Really! Well in that case, maybe *this* would work. [Takes other diagram and sketches]

Account based diagram, without Transaction

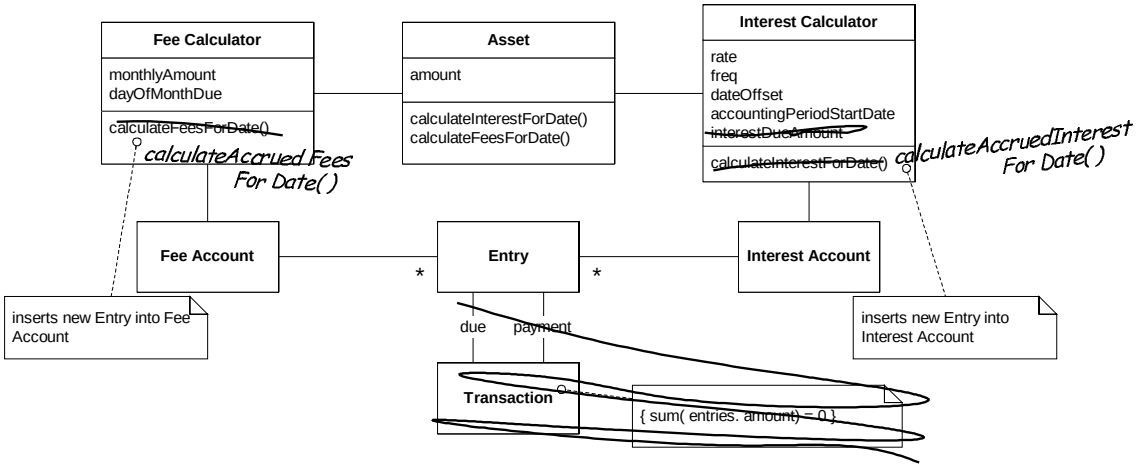


Figure 11. 6

EXP: That actually looks pretty good!

DEV2: The batch script will be easy to change to use these new objects.

DEV1: It will take a few days to get the new **Interest Calculator** working. There are quite a few tests to change. But the test will read clearer afterward.

The two developers went off and started refactoring based on the new model. As they got their hands on the code, tightening up the design, they had insights that refined the model.

Class diagram after the implementation

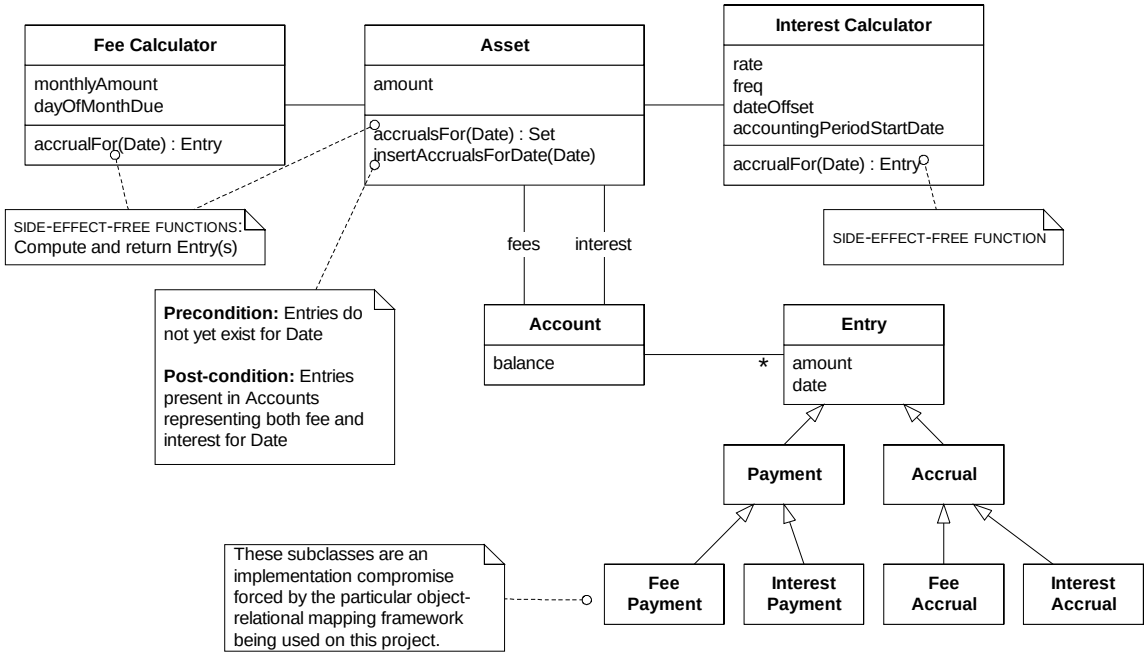


Figure 11. 7

Entries were subclassed into **Payment** and **Accrual** because closer inspection revealed slightly different responsibilities in the application for these, and because they were both important domain concepts.

On the other hand, the distinction between **Fee Payments**, **Interest Payments**, and so on, was a compromise for the implementation. Data was stored in relational tables, and the project standard was to make those tables interpretable without running the program. This meant keeping fee entries and interest entries in separate tables. The only way for them to do this, using their particular object-relational mapping framework, was to make distinct concrete subclasses. With different infrastructure, they might have avoided these subclasses. (Although this example is largely fictional, I've encountered similar situations. I threw this in to represent the rub of reality that we encounter all the time. We have to make the compromise that has an acceptable affect on our MODEL-DRIVEN DESIGN and then move on without letting it throw us off.)

The new design was much easier to analyze and test because the most complex functionality is in SIDE-EFFECT-FREE FUNCTIONS. The remaining command has simple code (because it calls various FUNCTIONS) and is characterized by ASSERTIONS.

Sometimes there are parts of our programs that we don't even suspect have the potential of useful domain models. They may have started very simply and evolved mechanistically. They seem like complicated application code. Analysis patterns can be particularly helpful in showing us these blind spots.

In the following example, a developer has a new insight into the black-box of the nightly batch, which had not been thought of as being domain oriented.

Continuing Example: Insight into the Nightly Batch

After a few weeks, the improved Account-based model had started to settle in. As often happens, the clarity of the new design made other problems more visible. The developer

(**DEV2**) who was adapting the nightly batch to interact with the new design began to see connections between the behavior of the batch and some of the concepts in *Analysis Patterns*. Here is a summary of some of the concepts he found most relevant.

<<Some kind of box starts here>>

Posting Rules

Accounting systems often provide multiple views of the same basic financial information. One account might track income while another might track an estimated tax on that income. If the system is expected to automatically update the estimated tax account, the implementation of those two accounts becomes very intertwined. There are systems where the majority of account entries are the result of such rules, and in such a system, the dependency logic gets to be a mess. Even in more modest systems, such cross-posting can be tricky. The first step toward taming the tangle of dependencies is to make these rules explicit by introducing a new object.

Class diagram of basic Posting Rule

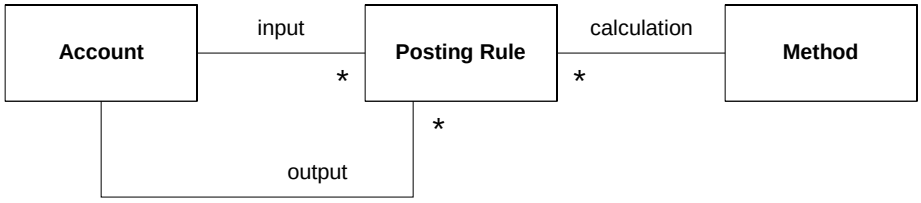


Figure 11. 8

A posting rule is triggered by a new **Entry** in its "input" account. It then derives a new **Entry** (based on its own calculation **Method**) and inserts the new **Entry** into its "output" **Account**. In a payroll system, an **Entry** in a salary **Account** might trigger a **Posting Rule** that would calculate a 30% estimated income tax and insert it as an **Entry** in the tax withholding **Account**.

Executing Posting Rules

The **Posting Rule** has established the conceptual dependency between **Accounts**, but if the pattern stopped there, it could be difficult to follow. One of the trickiest parts of dependency designs is the timing and control of updates. Fowler discusses three options.

"Eager firing" is the most obvious, but typically the least practical. Whenever an **Entry** is inserted into an Account, it immediately triggers the **Posting Rules** and all update are made immediately.

"Account-based firing" allows processing to be deferred. At some point, a message is sent to an **Account** and it triggers its **Posting Rules** to process all **Entries** inserted since its last firing.

Finally, "Posting Rule-based firing" is initiated by an external agent, which tells the **Posting Rule** to fire. The **Posting Rule** is responsible for looking up all **Entries** made to is input accounts since the last time it fired.

Although firing modes can be mixed in a system, each particular set of rules needs to have one clearly defined point of initiation and responsibility for identifying input **Account Entries**. The addition of the three firing modes to the UBIQUITOUS LANGUAGE is as important to the success of the pattern as the model object definitions themselves. It eliminates ambiguity and guides decision-making directly to a clearly defined set of choices. It identifies an easily overlooked challenge and provides vocabulary to support clear discussion.

<<end box>>

DEV2 needed a sounding board to discuss his new ideas. He met up his colleague (**DEV1**) the developer who had been primarily responsible for modeling the accruals.

DEV2: At some point, the nightly batch started being a place we swept stuff under the rug. There is domain logic implicit in what the script does, and its been getting more and more complicated. For a long time I've wanted to do a MODEL-DRIVEN DESIGN for the batch, separate out a DOMAIN LAYER and make the script itself a simple layer on top of the domain. But I could never figure out what that domain model would be like. It seemed like maybe it was just some procedures that didn't really make sense as objects.

As I've been reading the section in *Analysis Patterns* on Posting Rules, I've been getting some ideas. Here's what I had in mind. [Hands over a sketch]

A shot at using Posting Rules in the batch

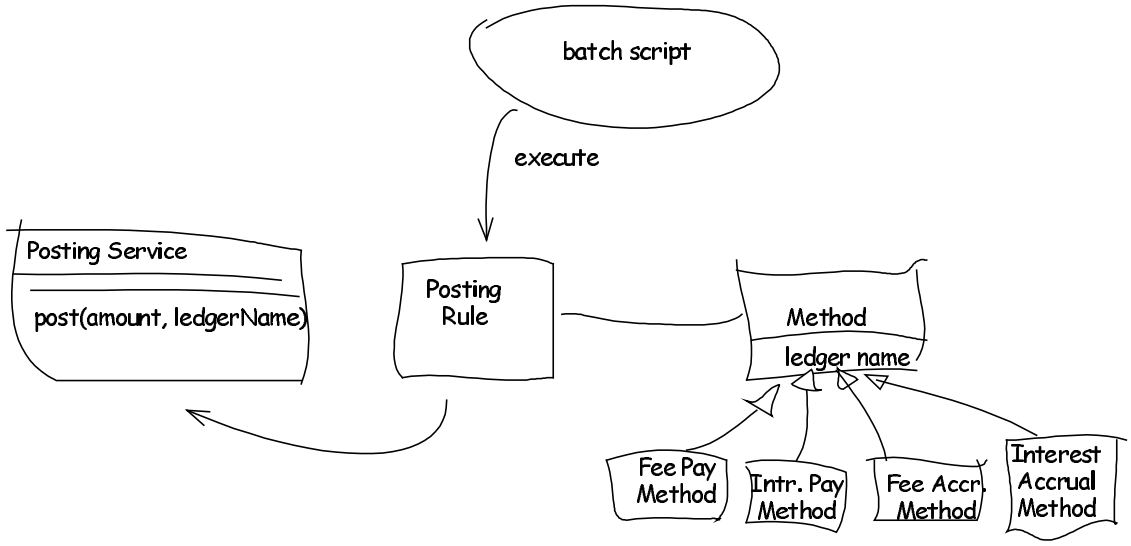


Figure 11. 9

DEV1: What is this "**Posting Service**"?

DEV2: That is a FAÇADE that presents the accounting application's API that presents it as a SERVICE. I actually made that a while back to simplify the batch code, and it also gave me an INTENTION-REVEALING INTERFACE for posting to the legacy system.

DEV1: Interesting. So, which firing style do you plan to use for these **Posting Rules**?

DEV2: I hadn't really gotten that far.

DEV1: Eager Firing would work for **Accruals**, since the batch actually tells the **Asset** to insert them, but it wouldn't work for **Payments**, which get entered during the day.

DEV2: I don't think we would want to couple the calculation method that tightly to the batch anyway. If we ever decided to trigger interest calculations at a different time, it would mess things up. And it just doesn't seem right, conceptually.

DEV1: It sounds like Posting-rule-based Firing. The batch tells each **Posting Rule** to execute and the rule goes and looks for appropriate new **Entries** and then does its thing. That's pretty much the way you've drawn it.

DEV2: So then we avoid creating a lot of dependencies on the batch design, and the batch keeps control. That sounds right.

DEV1: I'm still a little vague on the interaction of these objects with the **Accounts** and **Entries**.

DEV2: You and me both. The examples in the book create a direct link between the **Accounts** and the **Posting Rules**. That is kind of logical, but I don't think it will work very well for us. We have to instantiate these objects from data each time, so we would have to figure out which rule applies in order to associate it. Meanwhile, the **Asset** object is the one that knows the content of each **Account**, and therefore which rule to apply. Anyway, what about the rest of this?

DEV1: I hate to nitpick, but I don't think that we're using "**Method**" right. I think the concept is that the **Method** computes the amount to be posted, like say a 20% tax withholding on income. But in our case, that's simple – it's always the full amount being posted. I think the **Posting Rule** itself is supposed to know which **Account** to post to, which corresponds to our "ledger name".

DEV2: Oh. So if the **Posting Rule** is responsible for knowing the correct ledger name, we probably don't need **Method** at all.

Actually, this whole business of choosing the right ledger name is getting more and more complicated. It is already a combination of the type of income (fee or interest) with the "asset class" (a category the business applies to each **Asset**). That is one place I'm hoping this new model will help.

DEV1: Ok, let's focus there. The **Posting Rule** is responsible for choosing the ledger based on attributes of the **Account**. For now, we can make it a straight-forward way to handle asset class and the distinction between interest and fees. In the future, you'll have an OBJECT MODEL you can enhance to handle more complex cases.

DEV2: I need to think about this some more. Let me mull it over, and also reread the patterns, and then I'll take another stab at it. Could I talk with you about this again tomorrow afternoon?

Over the next few days, the two developers worked out a model and refactored the code so that the batch simply iterated through the **Assets**, sending a few self-explanatory messages too each and then committing the database transactions. The complexity was shifted into the DOMAIN LAYER where an OBJECT MODEL made it both more explicit and more abstract.

Class Diagram With Posting Rules

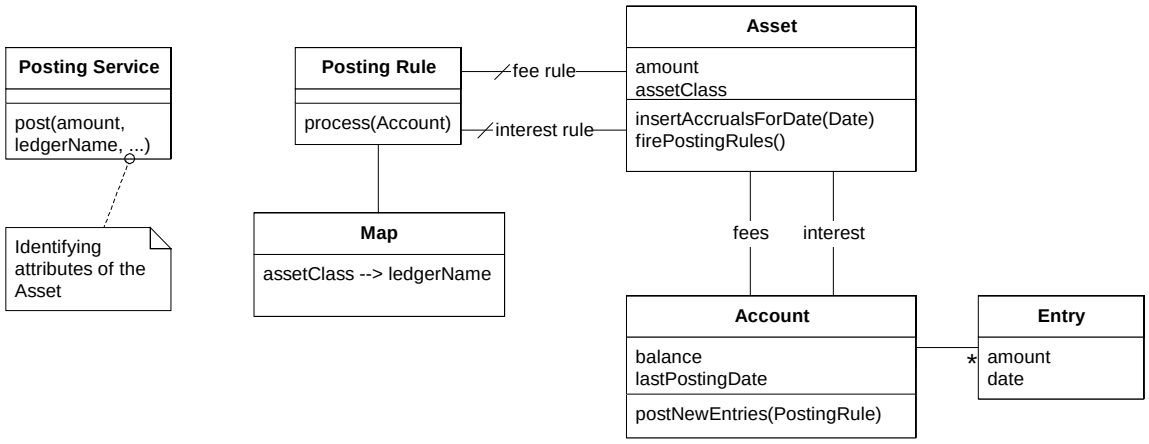


Figure 11. 10

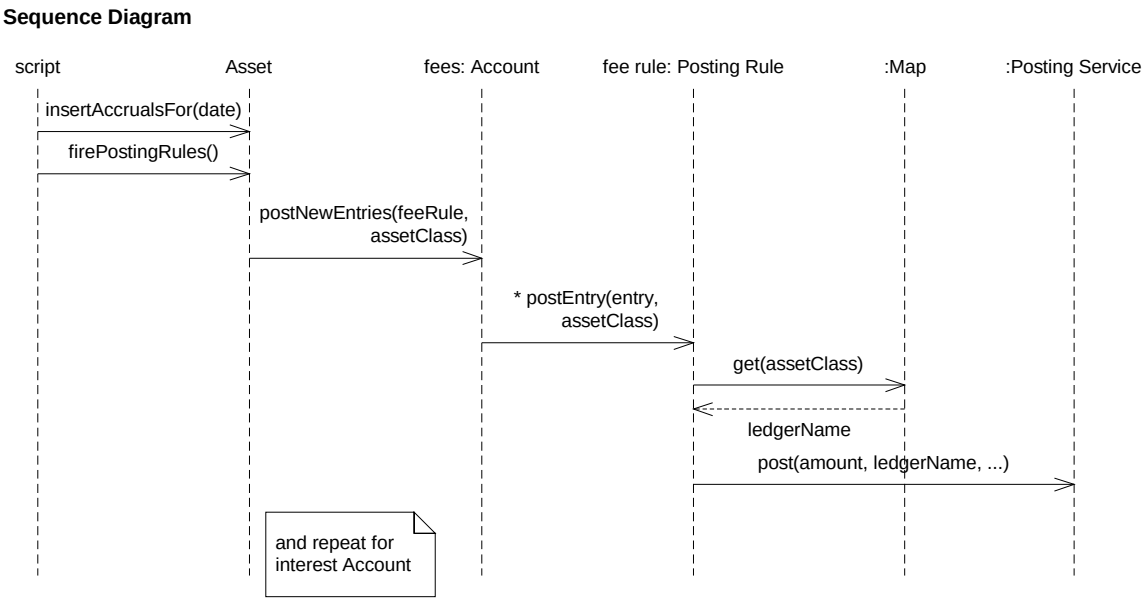


Figure 11. 11

The developers departed considerably from the details of the models presented in *Analysis Patterns,* yet they felt they had preserved the essence of the concepts. They were a little uncomfortable about involving the **Asset** in the selection of the **Posting Rule**. They went that way because the **Asset** had the knowledge of the nature of each **Account** (fee or interest), and was also the natural access point for the script. To have associated the rule object directly with the **Account** would have required a collaboration with the **Asset** object on each instantiation of the objects (each time the batch was run). Instead, they let the **Asset** object look up the two relevant rules through their SINGLETON access and pass them the appropriate **Account**. It seemed to make the code much more direct and so they made a pragmatic decision.

They both felt that conceptually it would have been better to associate **Posting Rules** only with **Accounts**, while keeping the **Asset** focused on its job of generating **Accruals**. They hoped that subsequent refactorings and deeper insight would bring them back to this and show them a way to make this clean division without loosing the obviousness of the code.

Analysis Patterns Are Knowledge to Draw On

When you are lucky enough to have an analysis pattern, it hardly ever is the answer to your particular needs. Yet it provides valuable leads in your investigation. It provides cleanly abstracted vocabulary. It should provide guidance about implementation consequences that will save you pain down the road.

All this feeds into the dynamo of knowledge crunching and refactoring toward deeper insight and stimulates development. The result often resembles the form documented in the analysis pattern, but adapted to circumstances. Sometimes the result doesn't even obviously relate to the analysis pattern itself, yet was stimulated by the insights from the pattern.

There is one kind of change you should avoid. When you use a term from a well known analysis pattern, take care to keep the basic concept it designates intact, however much the superficial form might change. There are two reasons for this. First, the pattern may embed understanding that will help you avoid problems. Second, and more important, your UBIQUITOUS LANGUAGE is enhanced when it includes terms that are widely understood or at least well explained. If your model definitions change through the natural evolution of the model, take the trouble to change the names too.

Quite a lot of object models have been written about, some specialized for one kind of application in one industry and some quite general. Most of them provide the seed of an idea, but only a few have captured the reasoning behind the choices and the consequences that follow, which are the most useful parts of an analysis pattern. More of these refined analysis patterns would be valuable, to help save us from reinventing the wheel again and again. I'd be surprised to ever see a comprehensive catalog. But industry-specific catalogs might arise. And some domains cross many applications and could be widely shared.

This kind of reapplication of organized knowledge is completely different from attempts to reuse code through frameworks or components, except that either could provide the seed of an idea that is not obvious. A model, even a generalized framework, is a complete working whole, while an analysis is a kit of model fragments. Analysis patterns focus on the most critical and difficult decisions and illuminate alternatives and choices. They anticipate down-stream consequences that are expensive if you have to discover them for yourself.

12.Relating Design Patterns to the Model

The patterns so far are specifically for solving problems in a domain model in the context of a MODEL-DRIVEN DESIGN. Actually, though, most of the patterns published to date are more technical in focus. What is the difference between a design pattern and a domain pattern? For starters, the authors of the seminal book, *Design Patterns*, had this to say.

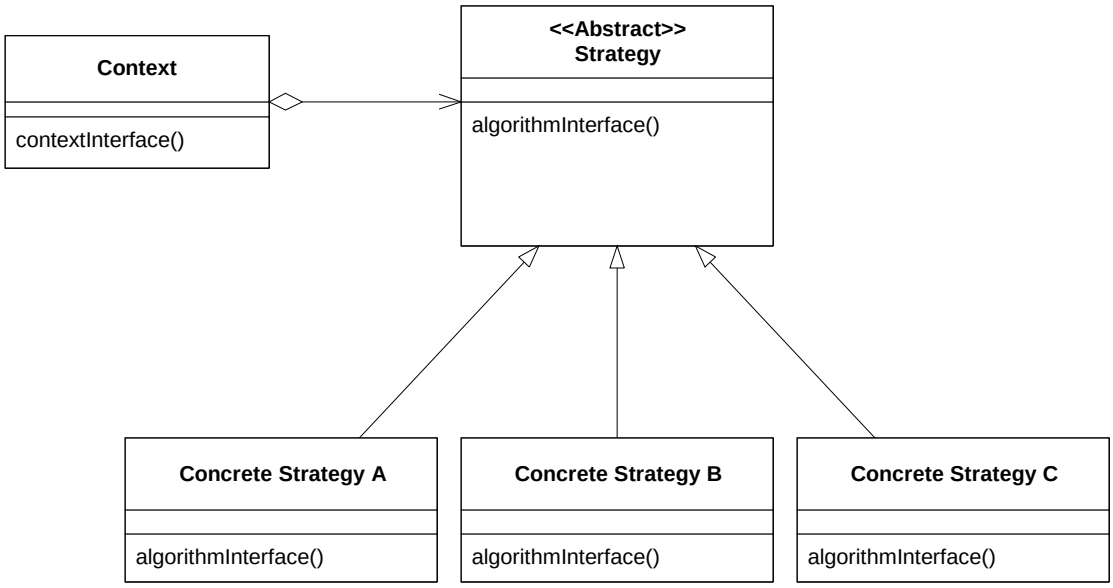
> Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block. For this book we have concentrated on patterns at a certain level of abstraction. Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem. The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. [GHJV1995, p.3]

Some, not all, of the design patterns in [GHJV1995] can be used as domain patterns. It requires a shift in emphasis. *Design Patterns* presented a catalog of design elements that solved problems commonly encountered in a variety of contexts. The motivations of these patterns and the patterns themselves were presented in purely technical terms. But a subset of these elements can be applied in the broader context of domain modeling and design, because they correspond to general concepts that emerge in many domains.

In addition to *Design Patterns*, there have been many other technical design patterns presented over the years. Some of them correspond to deep concepts that emerge in domains. It would be nice to draw on this work. To make use of them in domain-driven design, we have to look at the patterns on two levels simultaneously. On one level, they are technical design patterns in the code. On the other level, they are conceptual patterns in the model.

A sample of specific patterns from *Design Patterns* will show how a pattern conceived as a design pattern can be applied in the domain model, and will clarify the distinction between a technical design pattern and a domain pattern. COMPOSITE and STRATEGY demonstrate how some of the classic design patterns of [GHJV1995] can be applied to domain problems by thinking about them in a different way.

STRATEGY AKA POLICY



Define a family of algorithms, encapsulate each one, and make them interchangeable. STRATEGY lets the algorithm vary independently from clients that use it. [GHJV95]

Domain models contain processes that are not technically motivated, but actually meaningful in the problem domain. When alternative processes must be provided, the complexity of choosing the appropriate process is combined with the complexity of the multiple processes themselves, and things get out of hand.

When we model processes, we often realize that there is more than one legitimate way of doing them. As we start to describe these options, our definition of the process becomes clumsy and complicated. The actual behavioral alternatives we are choosing between are obscured as they are mixed in with the rest of the behavior.

We would like to separate this variation from the main concept of the process. Then we can see both the main process and the options more clearly. The STRATEGY pattern, already well established in the software design community, addresses this very issue, though the focus is technical. Here it is being applied as a concept in a model and the reflected in the code implementation of that model. There is the same need to decouple the highly variable part of the process from the more stable part.

Therefore,

Factor the varying part of a process into a separate "strategy" object in the model. Factor apart a rule and the behavior it governs. Implement the rule or substitutable process following the STRATEGY design pattern. Multiple versions of the strategy object represent different ways the process can be done.

Whereas the conventional view of STRATEGY as a design pattern focuses on the ability to substitute different algorithms, its use as a domain pattern focuses on its ability to express an a concept, usually a process or a policy rule.

Example: Route Finding Policies

A **Route Specification** is being passed to a **Routing Service**, whose job is to construct a detailed **Itinerary** that satisfies the SPECIFICATION. It is an optimization engine that can be tuned to find the fastest route or the cheapest.

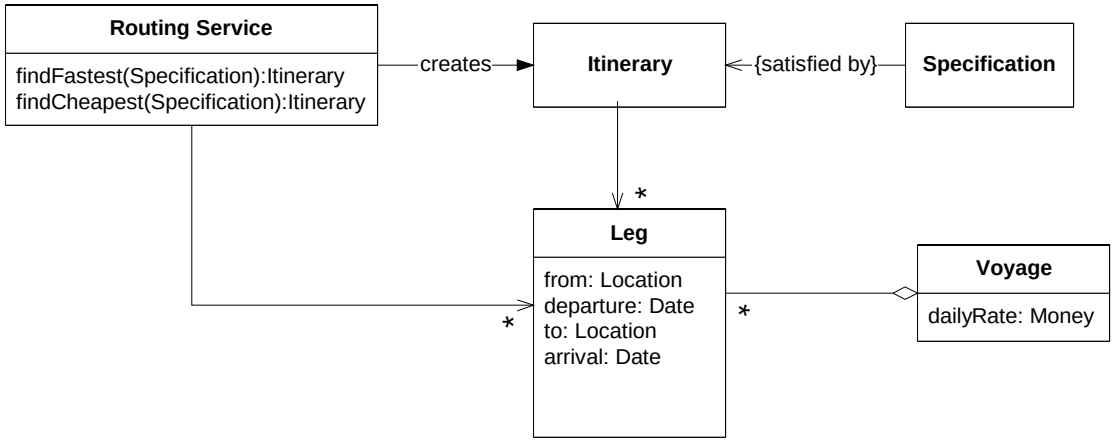


Figure 12. 1

This looks ok, but a detailed look at the routing code would reveal conditionals in every computation, making the decision between fastest or cheapest appear all over the place. More trouble will come when new criteria are added to make more subtle choices between routes.

One approach is to separate those tuning parameters into STRATEGIES. They can then be represented explicitly, passed into the **Routing Service** as a parameter.

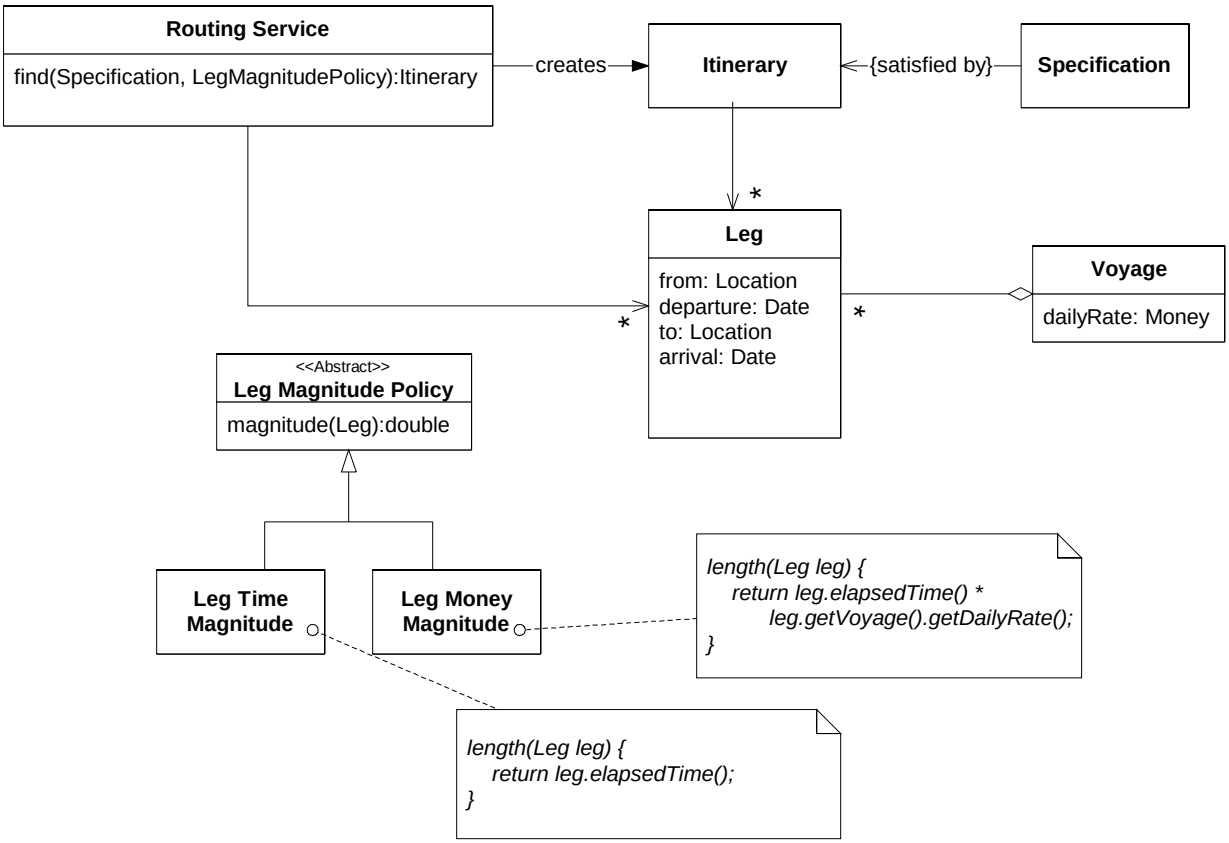


Figure 12. 2

The **Routing Service** now handles all requests in the same, unconditional way, looking for a sequence of **Legs** with a low magnitude, as computed by the **Leg Magnitude Policy**.

This design has the advantages that motivate the STRATEGY pattern in [GHJV1995]. On the level of application versatility and flexibility, the behavior of the **Routing Service** can now be controlled and extended by installing an appropriate **Leg Magnitude Policy**. The STRATEGIES illustrated in the diagram (fastest or cheapest) are the only the most obvious ones. Combinations that balance speed and cost are likely, and there may be other factors altogether, such as a bias toward booking cargo on the company's own transports rather than subcontracting to carry them on the transports of other shipping companies. These modifications could have been made anyway, but the logic would have wound through the internals of the **Routing Service** and bloated its interface. The decoupling does make it clear and easily testable.

A fundamentally important rule in the domain, the basis of choosing one **Leg** over another when building an **Itinerary**, is now explicit and distinct. It conveys the knowledge that a specific attribute (potentially derived) of an individual leg, boiled down to a single number, is the basis for routing. This makes possible a simple statement in the language of the domain that defines the **Routing Service's** behavior: The **Routing Service** chooses an **Itinerary** with a minimum total magnitude of the **Legs** based on the chosen STRATEGY.

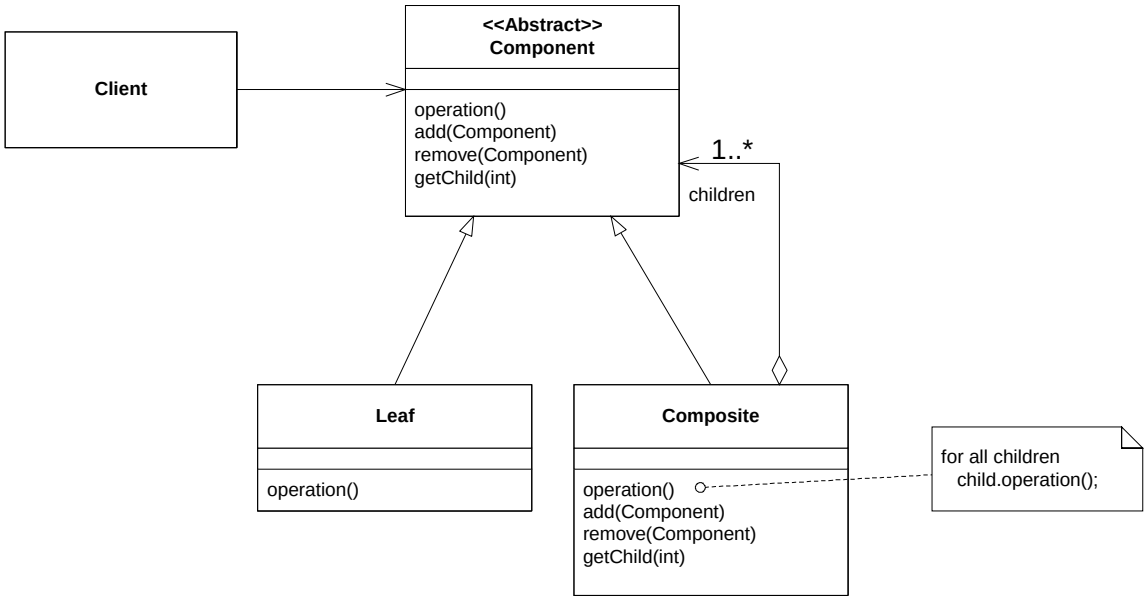
(Note: This implies that the **Routing Service** is actually evaluating **Legs** as it searches for an **Itinerary**. This is conceptually straight-forward, and could make reasonable prototype

implementation, but it is probably unacceptably inefficient. This example will be taken up again in an example in Chapter 14 “Maintaining Model Integrity”, where the same interface will be used with a completely different implementation of the **Routing Service**.)

When we use the technical design pattern in the domain layer, we have to add an additional motivation, another layer of meaning. The STRATEGY becomes more than just a useful implementation technique (though that is valuable as far as it goes) when the STRATEGY corresponds to an actual business strategy or policy.

The *consequences* of the design pattern fully apply. For example, [GHJV95] points out that clients must be aware of different STRATEGIES, which is also a modeling concern. A concern purely of implementation is that STRATEGIES can increase the number of objects in the application. If it is a problem, the overhead can be reduced by implementing STRATEGIES as stateless objects that contexts can share. The extensive discussion of implementation approaches all applies. This is because we are still using a STRATEGY. Our motivations are partially different, which will affect some choices, but the experience embedded in the design pattern is at our disposal.

COMPOSITE



Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. [GHJV95]

We often encounter, in modeling complex domains, that an important object is composed of parts, which are themselves made up of parts, which are made up of parts… In some domains, each of these layers is conceptually distinct, but in other cases there is a conceptual relationship that is lost if the model does not recognize it. There are also cases in which the nesting can be of arbitrary depth.

Common behavior has to be duplicated at each layer of the hierarchy, and nesting is rigid (e.g. containers can't usually contain other containers at their own level). Clients must deal with different levels of the hierarchy through different interfaces, even though there may be no conceptual difference they care about. Recursion through the hierarchy to produce aggregate information is very complicated.

When applying any design pattern in the domain, the first concern should be whether the pattern idea really is a good fit for the domain concept. You might find a case where it is convenient to be able to move recursively through some associated objects, but is it a true whole-part hierarchy? Have you found an abstraction under which all the parts truly are the same conceptual type? If you have, COMPOSITE will make those aspects of the model clearer, while allowing you to tap into the carefully thought-out design and implementation considerations of the design pattern.

Define an abstract type that encompasses all members of the COMPOSITE. Methods that return information are implemented on containers to return aggregate information about their contents, while "leaf" nodes implement them based on their own values. Clients deal with the abstract type and have no need to distinguish leafs from containers.

This is a relatively obvious pattern on the structural level, but designers often do not push themselves to flesh out the operational level of the pattern. The COMPOSITE offers the same behavior at every structural level, and meaningful questions can be asked small or large parts that transparently reflect their makeup. That rigorous symmetry is the key to the power of the pattern.

Example: Shipment Routes Made of Routes

A complete cargo shipment route is complicated. First, the container must be trucked to a rail-head, then carried to a port, then on a ship to another port, possibly transferred to other ships, finishing with ground transportation on the other end.

Schematic of a "route" made up of "legs"

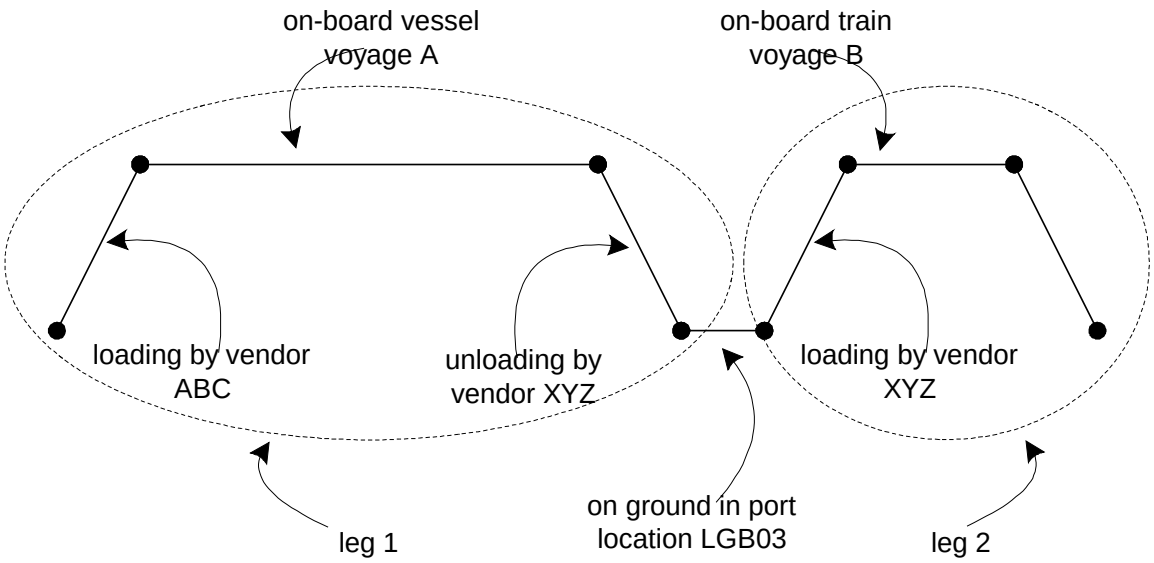


Figure 12. 3

An application development team has developed an OBJECT MODEL to express these arbitrarily long strings of legs that assemble into a route.

Class Diagram of Route made up of Legs

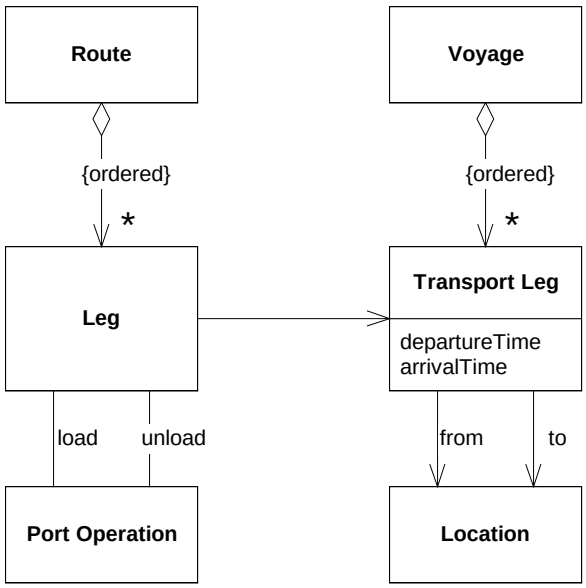


Figure 12. 4

Using this model, they are able to create **Route** objects based on booking requests. They are able to process the **Legs** into the operational plan for the step-by-step handling of the cargo. Then they discover something.

The developers had always thought of a route as an arbitrary, undifferentiated string of legs.

The developers' conception of a route

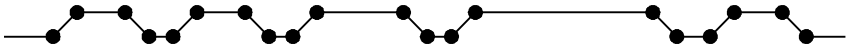


Figure 12. 5

It turns out the domain experts see the route as sequence of five logical segments.

The business experts' conception of a route

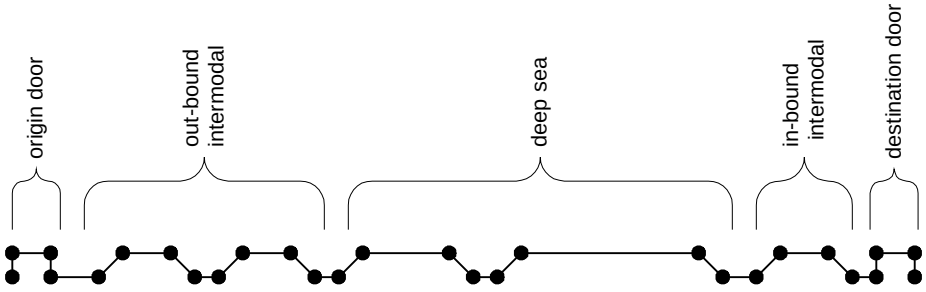


Figure 12. 6

Among other things, these may be planned at different times by different people, they have to be viewed in this way. And, on closer inspection, the "door legs" are quite different from the other legs, involving locally hired trucks or even customer haulage, in contrast to the elaborately scheduled rail and ship transports.

An OBJECT MODEL reflecting all these distinctions, starts to get complicated.

Elaborated class diagram of route

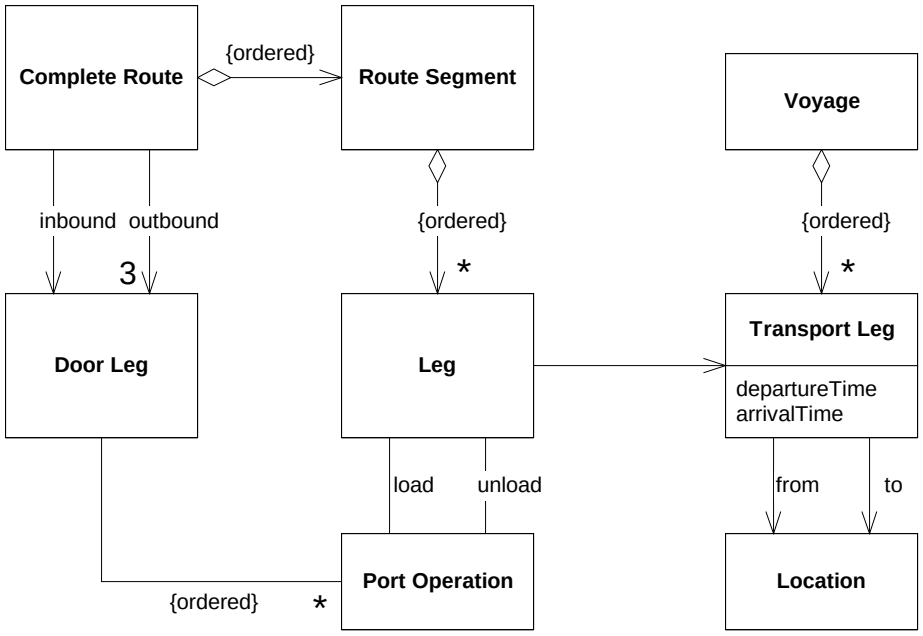


Figure 12. 7

Structurally it isn't so bad, but the uniformity of processing the operational plan is lost, so it becomes much more complicated. Other complications begin to surface, too. Any traversal of a route involves multiple collections of different types of objects.

Here is where a COMPOSITE comes in. It would be nice, for certain clients, to treat the different levels in this construct uniformly, as routes made up of routes. Conceptually it is sound. Every level is a movement of a container from one point to another, all the way down to an individual leg.

Class Diagram using COMPOSITE

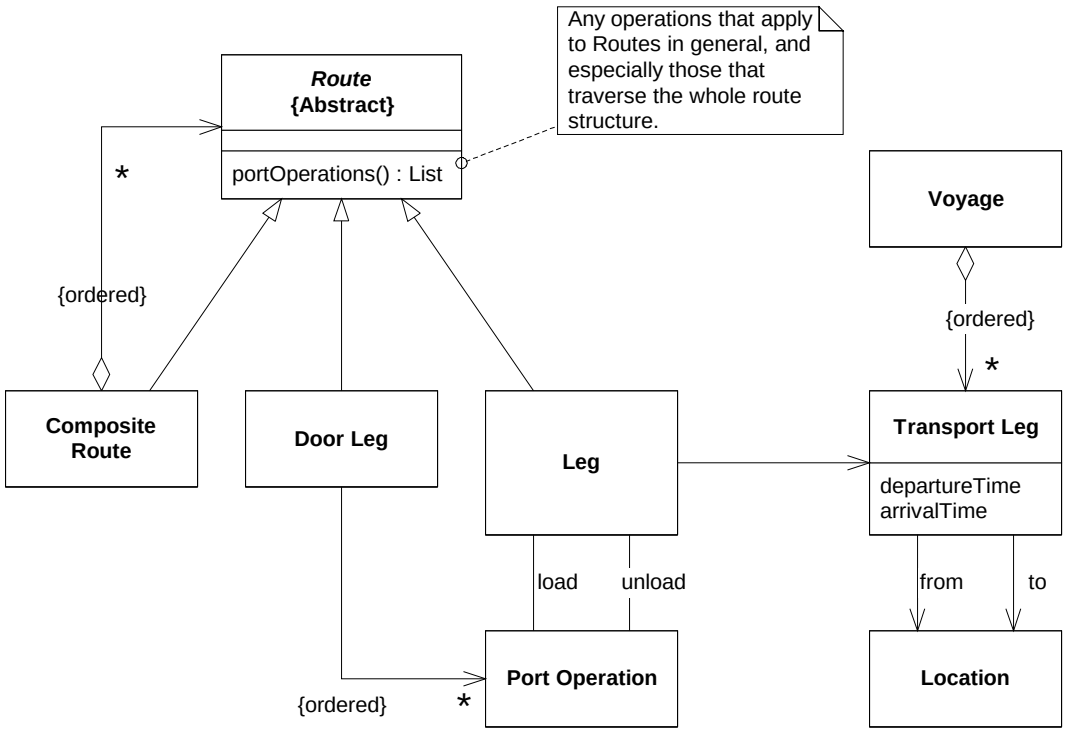


Figure 12. 8

Now, the static class diagram does not tell us as much about how door legs and other segments fit together as the previous one did. But the model is more than a static class diagram. We'll convey that assembly information through other diagrams and through the (now much simpler) code. This model captures the deep relatedness of all these different kinds of "Route". Generating the operational plan is simple again, as are other route traversing operations.

Instances representing a complete route

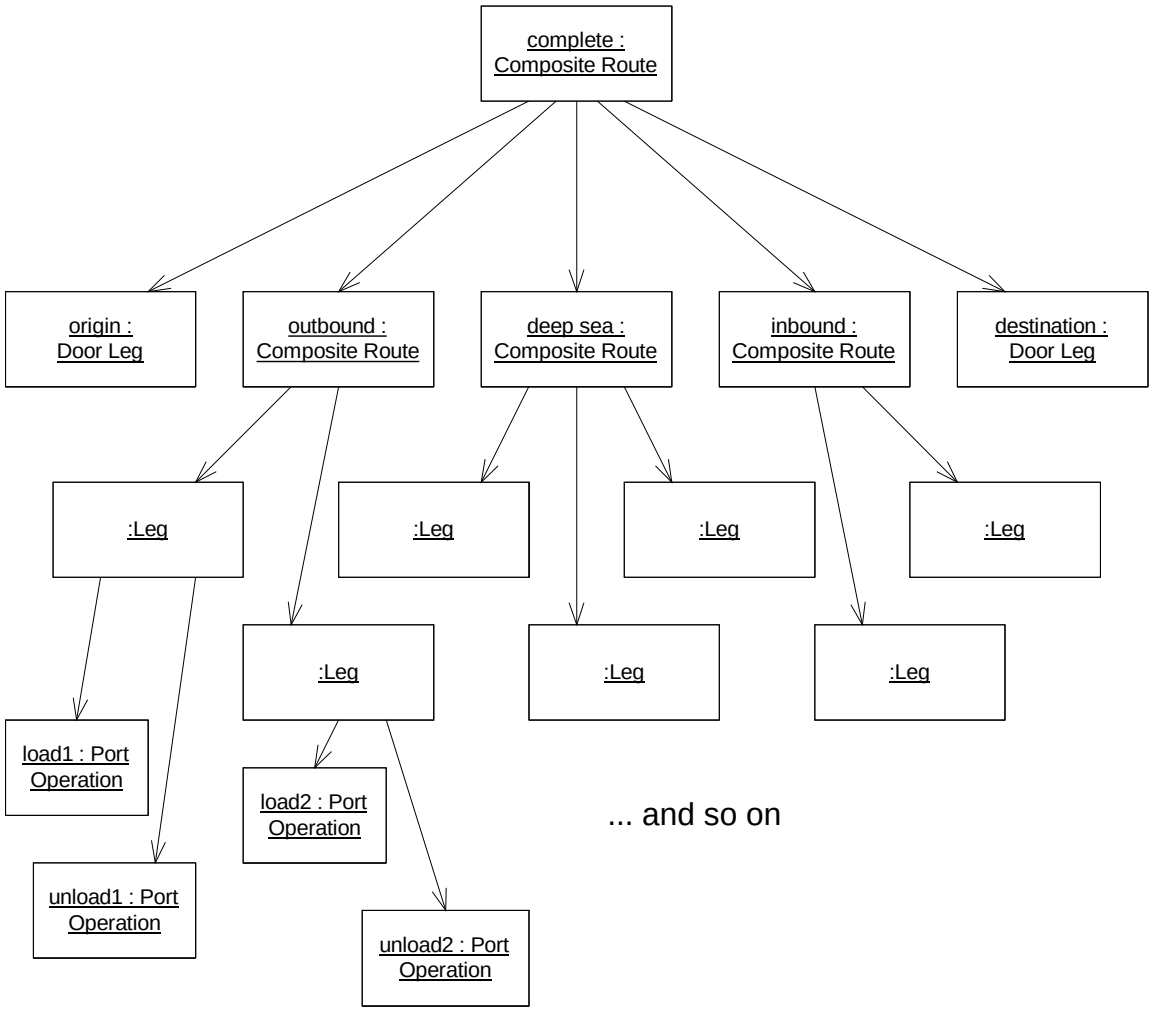


Figure 12. 9

With a route made of other routes, pieced together end-to-end to get from one place to another, you can have route implementations of varying detail, you can chop the end off a route and splice on a new ending, have arbitrary nesting of detail, and all sorts of possibly useful options.

Of course, we don't yet need such options. And before we needed those route segments and distinct door legs, we were doing just fine without COMPOSITE. A design pattern should only be applied when it is needed.

Why Not FLYWEIGHT?

Since I've referred to using the FLYWEIGHT pattern earlier (Chapter 5), you might expect it to be an example of a pattern to be applied to domain models. In fact, it is a good example of a design pattern that has *no correspondence* to the domain model.

When a limited set of VALUE OBJECTS is used many times (as in the example of electrical outlets on p. xx) it may make sense to implement them as FLYWEIGHTS. This is an *implementation* option available for VALUE OBJECTS and not for ENTITIES. Contrast this with COMPOSITE, in which conceptual objects are composed of other conceptual objects. In that case, the pattern applies to both views of the model, which is an essential trait of a domain pattern.

♦ ♦ ♦

I'm not going to provide a list of the design patterns that can be used as domain patterns. While I can't think of an example of using an interpreter as a domain pattern, I'm not prepared to say that there is no conception of any domain that would fit. The only requirement is that the pattern should say something about the conceptual domain, and not be only a technical solution to a technical problem.

13.Bringing the Pieces Together

Refactoring toward deeper insight is a multifaceted process. It will be helpful to stop for a moment to pull together the major points. There are three things you have to focus on: Live in the domain; Keep looking at things a different way; And maintain an unbroken dialog with domain experts. This creates a broader context for the process of refactoring.

The classic description of refactoring is a developer or two sitting at the keyboard, recognizing that some code can be improved, and changing it on the fly (with unit tests to verify their results, of course). This practice should happen all the time, but it isn't the whole story.

The last 5 chapters present an expanded view of refactoring, superimposed on the conventional micro-refactoring approach.

Initiation

Refactoring toward deeper insight can begin in many ways. It may be a response to a problem in the code – some complexity or awkwardness – but rather than a applying a standard transformation of the code, the developer(s) senses that the root of the problem is in the domain model. Perhaps a concept is missing. Maybe some relationship is wrong.

Unlike the conventional view of refactoring, this same realization could come when the code looks tidy, if the language of the model seems disconnected from the domain experts, or if new requirements are not fitting in naturally. It might result from learning, as a developer whose understanding has deepened sees an opportunity for a more lucid or useful model.

Seeing the trouble spot is often the hardest and most uncertain part. After that, developers can systematically seek out the elements of a new model. They can brainstorm with colleagues and domain experts. They can draw on systematized knowledge written as analysis patterns or design patterns.

Exploration Teams

Whatever the source of dissatisfaction, the next step is to seek the model refinement that will make it communicate clearly and naturally. This might require only some modest change that is immediately evident and can be accomplished in a few hours. In this case it resembles the traditional case. But it may well call for more time and involvement of more people.

The initiators of the change pick a couple of other developers who are good at thinking through that kind of problem, who know that area of the domain, or who have strong modeling skills. If there are subtleties, they make sure a domain expert is involved. This group of four or five people goes to a conference room or a coffee shop and brainstorms for half an hour to an hour and a half. They sketch UML diagrams; they try walking through scenarios using the objects. They make sure the subject matter expert understands the model and finds it useful. When they find something they are happy with, they go back and code it. Or they decide to mull it over for a few days, and they go back and work on something else. A couple days later the group reconvenes and goes through the exercise again. This time they are more confident, having slept on their earlier thoughts, and they reach some conclusions. They go back to their computer and code the new design.

There are a few keys to keeping this process productive.

- Self determination. A small team can be assembled on the fly to explore a design problem. The team can operate for a few days and then disband. There is no need for long-term elaborate organizational structures.

- Scope and sleep. Two or three short meetings spaced out over a few days should produce a design worth trying. Dragging it out doesn't help. If you get stuck, you may be taking on too much at once. Pick a smaller aspect of the design and focus on that.
- Exercising the UBIQUITOUS LANGUAGE. The involvement of the other developers and particularly the subject matter expert in the brainstorming session creates an opportunity to exercise and refine the UBIQUITOUS LANGUAGE. The end result of the effort is a refinement of that LANGUAGE which the original developer or pair will take back and formalize in code.

The preceding chapters have shown several dialogs in which developers and domain experts probe for better models. A full-blown brainstorming session is dynamic, unstructured, and incredibly productive.

Prior Art

It isn't always necessary to reinvent the wheel. The process of brainstorming for missing concepts and better models has a great capacity to absorb ideas from any source, combine them with local knowledge, and continue crunching to find answers to the current situation.

You can get ideas from books and other sources of knowledge about the domain itself. Although the people in the field may not have created a model suitable for running software, they may well have organized the concepts and found some useful abstractions. Feeding the knowledge-crunching process this way leads to richer, quicker results that will probably also seem more familiar to domain experts.

Sometimes prior experience with modeling for software is available in the form of analysis patterns. This kind of input has some of the effect of reading about the domain, but in this case there should be experience directly relevant to software implementation. Analysis patterns can give you subtle model concepts and let you avoid lots of mistakes. They don't give you a cook-book answer. They feed the knowledge-crunching process.

As the pieces are fit together, model concerns and design concerns must be dealt with in parallel. Again, it doesn't always mean inventing everything from scratch. Design patterns can often be employed in the domain layer when they fit both an implementation need and fit the model concept.

Likewise, when a common formalism, such as arithmetic or predicate logic, fits some part of a domain, you can factor that part out and adapt the rules of the formal system. This provides very tight and readily understood models.

A Design for Developers

Software isn't just for users. It is also for developers. Developers have to integrate code with other parts of the system. In a refactoring process, developers change the code again and again. Refactoring toward deeper insight both leads to and benefits from a supple design.

A supple design communicates its intent. It is easy to anticipate the effect of running code and therefore easy to anticipate the consequences of changing it. It helps limit mental overload, primarily by reducing dependencies and side-effects. It is based on a deep model of the domain that is fine-grained only where most critical to the users. This makes for flexibility where change is most common, and simplicity elsewhere.

Timing

If you wait until you can make a complete justification for a change, you've waited too long. Your project is already incurring heavy costs, and the change will probably be harder because it is more elaborated and embedded in other code.

Continuous refactoring has come to be considered a "best practice", but most projects are still too cautious about it. They see the risk of any code change and the cost of developer time to make a change, while the risk of keeping an awkward design and the cost of working around that design are hard to see in advance. Developers who want to refactor are often asked to justify the decision. While this seems reasonable, it makes an already difficult thing impossibly difficult, and tends to squelch refactoring (or drive it underground). Software development is not such a predictable process that the benefits of a change or the costs of not making a change can be accurately calculated.

Refactoring toward deeper insight can become part of the ongoing exploration of the subject matter of the domain, education of the developers, and meeting of the minds of developers and domain experts. Therefore, refactor when

- The design does not express the team's current understanding of the domain
- Important concepts are implicit in the design (and you see a way to make them explicit)
- You see an opportunity to make some important part of the design suppler.

This does not justify any change at any time. Don't refactor the day before a release. Don't introduce "supple designs" that are just demonstrations of technical virtuosity, but fail to cut to the core of the domain. Don't introduce a "deeper model" that you couldn't convince a subject matter expert to use, no matter how elegant it seems. Don't be absolute about things, but push beyond the comfort zone in the direction of favoring refactoring.

Crisis As Opportunity

For over a century after Charles Darwin introduced it, the standard model of evolution was that species changed gradually, somewhat steadily, over time. Suddenly, in the 1970's, this model was displaced by the "punctuated equilibrium" model. In this expanded view of evolution, long periods of gradual change or stability are interrupted by relatively short bursts of rapid change. Then things settle down into a new equilibrium. Software development has an intentional direction behind it evolution lacks (although it may not be evident on some projects) but nonetheless it follows this kind of rhythm.

Classic descriptions of refactoring sound very steady. Refactoring toward deeper insight usually isn't. A period of steady refinement of a model can suddenly bring you to an insight that shakes up everything. These breakthroughs don't happen every day, yet a large proportion of the changes that lead to a deep model and supple design emerge from them.

Such a situation often does not look like an opportunity; it seems more like a crisis. Suddenly there is some obvious inadequacy in the model. There is a gaping hole in what it can express, or some critical area where it is opaque. Maybe it is makes statements that are just wrong.

This means the team has reached a new level of understanding. From their now-elevated viewpoint, the old model looks poor. From that viewpoint, they can conceive a far better one.

Refactoring toward deeper insight is a continuing process. Implicit concepts are recognized and made explicit. Parts of the design are made suppler, perhaps taking on a declarative style. Development suddenly comes to the brink of a breakthrough and plunges through to a deep model – and then steady refinement starts again.

Part IV. Strategic Design

As systems grow too complex to know completely at the level of individual objects, we need techniques for manipulating and comprehending large models. This section presents principles that enable the modeling process to scale up to very complicated domains. Most of these decisions must be made at team level or even negotiated between teams. These are the decisions where design and politics often intersect.

The goal of the most ambitious enterprise systems is a tightly integrated system spanning the entire business. Yet the entire business model for almost any such organization is too large and complex to manage or even understand as a single unit. The system must be broken into smaller parts, in both concept and implementation. The challenge is to accomplish this modularity **without losing the benefits of integration,** allowing different parts of the system to inter-operate to support the coordination of various business operations. A monolithic, spaghetti-like, all-encompassing domain model will be unwieldy and loaded with subtle duplications and contradictions. A set of small distinct subsystems glued together with ad-hoc interfaces will lack the power to solve enterprise wide problems, and allows consistency problems to arise at every integration point. The pitfalls of both extremes can be avoided with a systematic, evolving design strategy.

A good domain model captures an abstraction of the business in a form defined enough to be coded into software. Strategic design principles provide a guide to design decisions for the model that reduce interdependence of parts and improve clarity and ease of understanding and analysis without reducing their interoperability and synergy. It must also capture the conceptual core of the system, the "vision" of the system. **And it must do all this without bogging the project down.** The three broad themes explored in this section can help accomplish these goals: BOUNDED CONTEXTS, distillation, and large-scale structure.

BOUNDED CONTEXT, the least obvious of the principles, is actually the most fundamental. A successful model, large or small, has to be logically consistent throughout, without any contradictory or overlapping definitions. Enterprise systems sometimes integrate subsystems with varying origins or have applications so distinct that very little in the domain is viewed in the same light. It may be asking too much to unify the models implicit in these disparate parts. By explicitly defining a BOUNDED CONTEXT within which a model applies, and then, when necessary, defining its relationship with other contexts, the modeler can avoid bastardizing the model.

DISTILLATION reduces the clutter and focuses the attention appropriately. Often a great deal of effort is spent on peripheral issues in the domain. The overall domain model needs to make prominent the most value-adding and special aspects of your system and be structured to give that part as much power as possible. While some supporting components are critical, they must be put into their proper perspective. This not only helps to direct efforts toward vital parts of the system, but it keeps the vision of the system from being lost. DISTILLATION can bring clarity to an overall model. And with a clearer view the design of the core can be made more useful.

LARGE-SCALE STRUCTURE completes the picture. In a very complex model, you may not see the forest for the trees. Distillation helps, by focusing the attention on the core and presenting the other elements in their supporting roles, but the relationships can still be too confusing without some LARGE-SCALE STRUCTURE that allows system-wide design elements and patterns to be applied. I'll overview a few approaches to large scale structure and then go into depth on one such pattern, RESPONSIBILITY LAYERS, in which a small set of fundamental responsibilities are identified that can be organized into layers with defined relationships between layers, such as modes of communication and allowed references. These are examples of LARGE SCALE STRUCTURES, not a comprehensive catalog. New ones should be invented when needed. Some such structure can bring a uniformity to the design that can accelerate the design process and improve integration.

These principles, useful separately but particularly powerful taken together, help to produce good designs even when systems become too big to understand as a whole while simultaneously thinking about the detail level of individual objects. LARGE-SCALE STRUCTURE can bring consistency to the disparate parts to help those parts mesh. STRUCTURE and DISTILLATION make the complex relationships between the parts

comprehensible while keeping the big picture in view. BOUNDED CONTEXTS allow work to proceed in different parts without corrupting the model or unintentionally fragmenting it. Adding these concepts to the team's language will help teams work out their own solutions.

14.Maintaining Model Integrity

I once worked on a project where several teams were working in parallel on a major new system. One day, the team working on the customer-invoicing module was ready to implement an object they called "**Charge**", when they discovered that another team had already built one. Diligently, they set out to reuse the existing object. They discovered it didn't have an "expense code", so they added one. It already had the "posted amount" attribute they needed. They had been planning to call it "amount due", but what is in a name? They changed it. Adding a few more methods and associations, they got something that looked like what they wanted, without disturbing what was there. They had to ignore many associations they didn't need, but their application module ran.

A few days later, mysterious problems surfaced in the bill-payment application module for which the **Charge** had been originally written. Strange "**Charges**" appeared that no one remembered entering and which didn't make any sense. The program began to crash when some functions were used, particularly the month-to-date tax report. Investigation revealed that the crash resulted when a function was used that summed up the amount deductible for all the current month's payments. The mystery records had no value in the "percent deductible" field, although the validation of the data entry application required it and even put in a default value.

The problem was that these two groups had *different models*, but they did not realize it, and there were no processes in place to detect it. Each made assumptions about the nature of a charge that were useful in their context (billing customers versus paying vendors). When their code was combined without resolving these contradictions, the result was unreliable software.

If only they had been more aware of this reality, they could have consciously decided how to deal with it. That might have meant working together to hammer out a common model and then writing an automated test suite to prevent future surprises. Or it might simply have meant an agreement to develop separate models and keep hands off each other's code. Either way, it starts with an explicit agreement on the boundaries within which each model applies.

What did they do once they knew about the problem? They created separate **Customer Charge** and **Supplier Charge** classes and defined each according to the needs of the corresponding team. The immediate problem having been solved, they went back to doing things just as before. Oh, well.

Although we seldom think about it explicitly, the most fundamental requirement of a model is that it be internally consistent; that every term always have the same meaning, and that it contain no contradictory rules. Internal consistency of a model such that each term is unambiguous and no rules contradict is called *unification*. A model is meaningless unless it is logically consistent. In an ideal world, we would have a single model spanning the whole domain of the enterprise. This model would be unified, without any contradictory or overlapping definitions of terms. Every logical statement about the domain would be consistent. But the world of large systems development is not the ideal world. To maintain that level of unification in an entire enterprise system is more trouble than it is worth. It is necessary to allow multiple models to develop in different parts of the system. We need to make careful choices about which parts of the system will be allowed to diverge and what their relationship to each other will be. We need ways of keeping crucial parts of the model tightly unified. None of this happens by itself or through good intentions. It only happens through conscious design decisions and institution of specific processes.

Total unification of the domain model for a large system will not be feasible or cost-effective.

Sometimes people fight this. Most people see the price that multiple models exact by limiting integration and making communication cumbersome. On top of that, it somehow seems inelegant. This resistance to multiple models sometimes leads to very ambitious attempts to unify all the software in a large project under a single model. I know I've been guilty of this kind of overreaching. Consider the risks:

1. Too many legacy replacements may be attempted at once.

2. Large projects may bog down because the coordination overhead exceeds their abilities.
3. Applications with specialized needs may have to use models that don't fully satisfy their needs, forcing them to put behavior elsewhere.
4. Conversely, attempting to satisfy everyone with a single model may lead to complex options that make the model difficult to use.

What's more, model divergences are as likely to come from political fragmentation and differing management priorities as from technical concerns. And the emergence of different models can be a result of team organization and development process. So even when no technical factor prevents full integration, the project may still face multiple models.

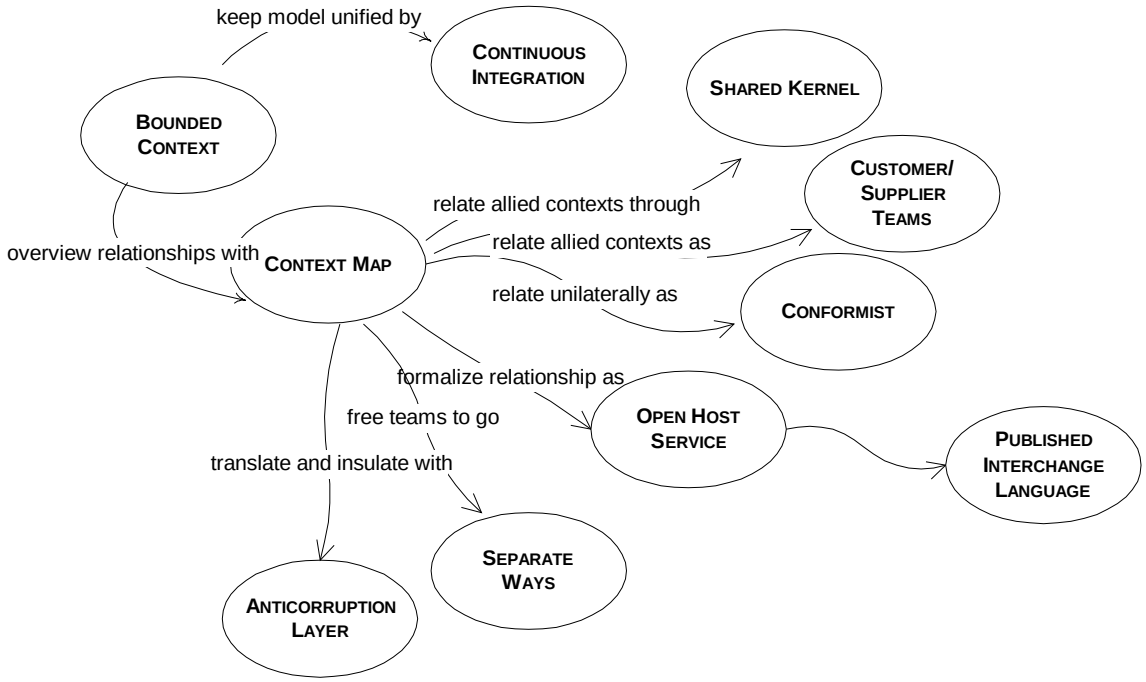
Given that it isn't practical to maintain a unified model for an entire enterprise, we don't have to leave ourselves at the mercy of events. Through a combination of proactive decisions about what should be unified and pragmatic recognition of what is not unified, we can create a clear, shared picture of the situation. With that in hand, we can systematically set about making sure that the parts we want to unify stay that way, and the parts that are not unified don't cause confusion or corruption.

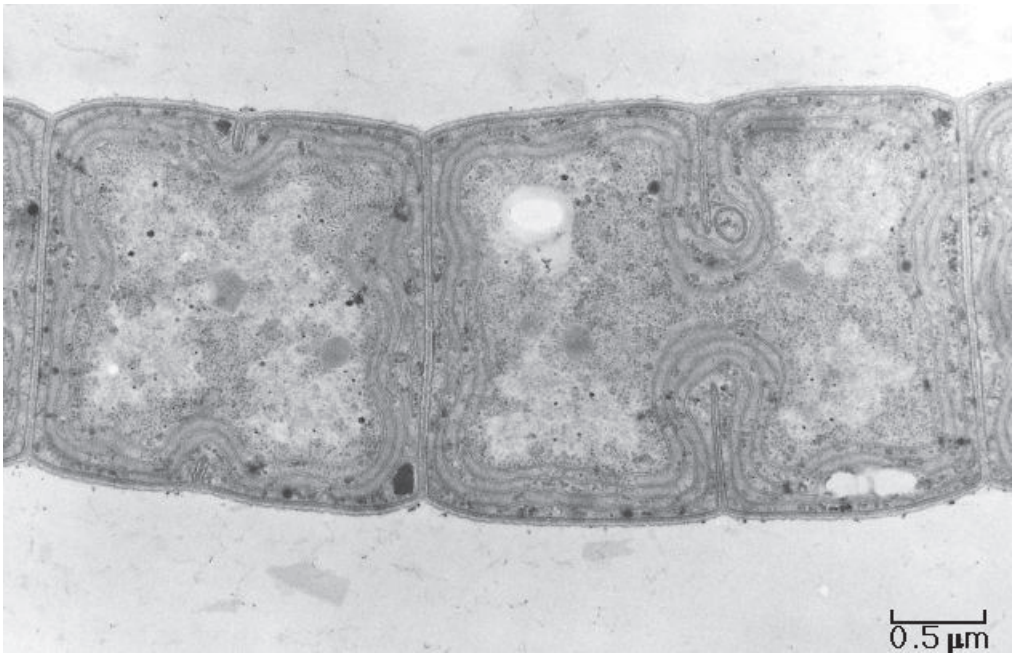
We need a way to make the boundaries and relationships between different models clear. We need to choose our strategy consciously and then follow our strategy consistently.

This chapter lays out techniques for recognizing, communicating, and choosing the limits of a model and its relationships to others. It all starts with mapping the current terrain of the project. A BOUNDED CONTEXT defines the range of applicability of each model, while a CONTEXT MAP gives a global overview of the project's contexts and the relationships between them. This reduction of ambiguity will, in and of itself, change the way things happen on the project, but it isn't necessarily enough. Once we have a CONTEXT BOUNDED a process of CONTINUOUS INTEGRATION will keep the model unified.

Then, starting from this stable situation, we can start to migrate toward more effective strategies for BOUNDING CONTEXTS and relating them, ranging from closely allied contexts with SHARED KERNELS to loosely coupled models that go their SEPARATE WAYS.

Navigation Map for Model Integrity Patterns





Cells can exist because their membranes define what is in and out and determine what can pass.

Multiple models coexist on big projects, and this works fine in many cases. Different models apply in different contexts. For example, you may have to integrate your new software with an external system that your team has no control over. This would probably be clear to everyone as a distinct context where the model under development doesn't apply. Other situations can be more confusing. In the story at the beginning of this chapter, two teams were working on different functionality for the same new system. Were they working on the same model? Their intention was to share at least part of what they did, but there was no demarcation to tell them what they did or did not share. And they had no process in place to hold a shared model together or quickly detect divergences. They realized they had diverged only after their system's behavior suddenly became unpredictable.

Even a single team can end up with multiple models. Communication can lapse, leading to subtly conflicting interpretations of the model. Older code often reflects an earlier conception of the model that is subtly different from the current model.

Everyone is aware that the data format of another system is different and calls for a data conversion, but this is only the mechanical dimension of the problem. More fundamental is the difference in the models implicit in the two systems. When the discrepancy is not with an external system, but within the same code base, it is even less likely to be recognized, yet this happens on *all* large team projects.

Multiple models are in play on any large project. Yet, when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confused. It is often unclear what context a model should not be applied in or where new work should be focused.

Failure to keep things straight is ultimately revealed when the running code doesn't work right, but it starts in the way teams are organized and the way people interact. Therefore, to clarify the context of a model, we have to look at both the project and their end products (code, database schemas, etc.)

A model applies in a *context.* This may be a certain part of the code, or the work of a particular team. For a model invented in a brainstorming session, the context could be limited to that particular conversation. The context of a model used in an example in this book is that particular example section and any discussion of it. The model context is whatever set of conditions must apply in order to be able to say that the terms in a model have a specific meaning.

To begin to solve the problems of multiple models, we need to explicitly define the scope of a particular model as a bounded part of a software system within which a single model will apply and will be kept as unified as possible. This has to be reconciled with the team organization.

Therefore,

Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside.

A BOUNDED CONTEXT delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other contexts. Within that context, work to keep the model logically unified, but do not worry about applicability outside those bounds. In other contexts, other models apply, with differences in terminology, concepts and rules, and different dialects of the UBIQUITOUS LANGUAGE. By drawing an explicit boundary, you can keep the model pure, and therefore potent, where it is applicable. At the same time you avoid confusion when shifting your attention to other CONTEXTS. Integration across the boundaries necessarily will involve some translation, which you can analyze explicitly.

<<BEGIN SIDEBAR>>

This issue sometimes gets confused with the motivations for MODULES. True, when it is recognized that two sets of objects make up different models they are typically placed in separate MODULES, and this does provide different name-spaces that allow them to compile even if they have name overlaps. But this is just an implementation mechanism for code separation of different models. This issue is preceded by the fundamental problems, recognizing model differences and deciding what to do with them. Furthermore, MODULES are also used to organize the elements within one model, so they don't communicate an intention to separate models. The separate name-spaces they create can actually make it harder to spot accidental model divergences. While technical tools can help us communicate and implement conceptual issues, they don't solve our conceptual problems.

<<END SIDEBAR>>

Example: Booking Context

A shipping company has an internal project to develop a new application for booking cargo. This application is to be driven by an object model. What is the outline of the BOUNDED CONTEXT within which this model applies? To answer this question, we have to look at what is happening on the project. Keep in mind, this is a look at the project *as it is*, not as it ideally should be.

One project team is working on the booking application itself. They are not expected to work on the model objects themselves, but the application they are building has to display and manipulate those objects. This team is a consumer of the model. The model is valid within the application (it's primary consumer) and therefore the booking application is in bounds.

The completed bookings have to be passed to the legacy cargo-tracking system. A decision was made up-front that the new model would depart from that of the legacy, so the legacy cargo tracking system is outside the boundary. Necessary translation between the new model and the legacy is to be the responsibility of the legacy maintenance team.

Translation mechanism is not driven by the model. It is not in the BOUNDED CONTEXT. (It is part of the boundary itself, which will be discussed in CONTEXT MAP.) This is a good thing. Because the primary work of the legacy team (which has responsibility for this translation) is out of context, it would be unrealistic to ask for any real use of the model.

The team responsible for the model deals with the whole lifecycle of the objects, including persistence. Since this team has control of the database schema, they've been deliberately keeping the object-relational mapping straight-forward. In other words, the schema is being driven by the model, and therefore is in bounds.

Yet another team is working on a model and application for scheduling the voyages of the cargo ships. The scheduling and booking teams were initiated together, and both teams had intended to produce a single, unified system. The two teams have casually coordinated with each other, and they occasionally share objects, but they are not systematic about it. They are *not* working in the same context. This is a risk, because they do not think of themselves as working on separate models. To the extent they integrate, there will be problems unless they put in place processes to manage the situation. (The SHARED KERNEL, discussed later in this chapter, might be a good choice.) The first step, though, is to recognize the situation *as it is*. They are not in the same bounded context and should stop trying to share code until some changes are made.

This BOUNDED CONTEXT is made up of all those aspects of the system that are driven by this particular model: the model objects, the database schema that persists the model objects, and the booking application. Two teams work primarily in this context, the modeling team and the application team. Information has to be exchanged with the legacy tracking system, and the legacy team has primary responsibility for the translation at this boundary, with cooperation from the modeling team. There is no clearly defined relationship between the booking model and the voyage schedule model, and defining that relationship should be one of the first their first actions. In the mean time, they should be very careful about sharing code or data.

So, what has been gained by defining this BOUNDED CONTEXT? For the teams working in context, clarity. Those two teams know they must stay consistent with one model. They make design decisions in that knowledge and watch for fractures. For the teams outside, freedom. They don't have to walk in the gray zone, not using the same model, yet somehow feeling they should. But the most concrete gain in this particular case is probably highlighting the risk of the informal sharing that was happening between the booking model team and the voyage schedule team. To avoid problems, they really need to decide on the cost/benefit tradeoffs of sharing and put in processes to make it work. This doesn't happen unless everyone understands where the bounds of the model contexts are.

♦ ♦ ♦

Of course, boundaries are special places. The relationships between a BOUNDED CONTEXT and its neighbors requires care and attention, which will be approached with a CONTEXT MAP and several patterns for relating BOUNDED CONTEXTS.

Within a BOUNDED CONTEXT, a process of CONTINUOUS INTEGRATION can help keep the model unified. But first, what does it look like when unification of a model is breaking down? How do you recognize these conceptual splinters?

Recognizing Splinters Within a BOUNDED CONTEXT

Many symptoms may indicate unrecognized model differences. When code clashes, either by interfaces that don't match or, more subtly, by behavior that is unexpected, that is a sure sign. The CONTINUOUS INTEGRATION process with automated tests can help catch these. But the early warning sign is usually a confusion of language.

Combining elements of distinct models causes two categories of problems: duplicate concepts and false cognates. Duplication of concepts means that there are two model elements (and attendant implementations) that actually represent the same concept. Every time this information changes, it has to be updated in two places with conversions. Every time new knowledge leads to a change in one of the objects, the other has to be reanalyzed and changed too. Except the reanalysis doesn't happen in reality, so the result is two versions of the same concept that follow different rules and even have different data. On top of that, the team members must learn not one but two ways of doing the same thing, along with all the ways they are being synchronized.

False cognates may be slightly less common, but more insidiously harmful. This is the case when two people who are using the same term (or implemented object) think they are talking about the same thing, but really are not. The example in the introduction (two different business activities both called a "charge") is typical, but conflicts can be even subtler when the two definitions are actually related to the same aspect in the domain, but have been conceptualized in slightly different ways. False cognates lead to development teams that step on each other's code, databases that have weird contradictions, and confusion in communication within the team. The term "false cognate" is ordinarily applied to natural languages. English speakers learning Spanish often misuse the word "*embarasada*". This word does not mean "embarrassed"; it means "pregnant". Oops.

When you detect these problems, your team will have to make a decision. You may want to pull the model back together and refine the processes to prevent fragmentation. Or, the fragmentation may be a result of groups who want to pull the model in different directions for good reasons, and you may decide to let them develop independently. Dealing with these issues is the subject of the remaining patterns in this chapter.

CONTINUOUS INTEGRATION

…Once a BOUNDED CONTEXT has been defined, we must keep it sound.

Whether we attack our system as a single unified model, or we break it into multiple models, the model of each BOUNDED CONTEXT must be kept unified. Typically at least one or two of those pieces will be larger than the work of one person and will be taken on by a team. Sometimes developers do not fully understand the intent of some object or interaction modeled by someone else, and they change it in a way that makes it unusable for its original purpose. Sometimes they don't realize that the concepts they are working on are already embodied in another part of the model and they duplicate (inexactly) those concepts and behavior. Sometimes they are aware of those other expressions, but are afraid to tamper with them, for fear of corrupting the existing functionality, and so they proceed to duplicate concepts and functionality.

When a number of people are working in the same BOUNDED CONTEXT there is a strong tendency for the model to fragment. The bigger the team, the bigger the problem, but as few as three or four people can encounter serious problems. Yet breaking down the system into ever-smaller contexts eventually loses a valuable level of integration and coherency.

It is very hard to maintain the level of communication needed to develop a unified system of any size. We need ways of increasing communication and reducing complexity. We also need safety nets that prevent overcautious behavior, like developers duplicating functionality because they are afraid they will break existing code.

It is in this environment that Extreme Programming (XP) really comes into its own. Many of the XP practices are aimed at this specific problem of maintaining a coherent design that is being constantly changed by many people. XP in its purest form is a nice fit for maintaining model integrity within a single BOUNDED CONTEXT. However, whether or not XP is being used, it is essential to have some process of CONTINUOUS INTEGRATION.

CONTINUOUS INTEGRATION means that all work within the context is being merged and made consistent frequently enough that when splinters happen they are caught and corrected quickly. CONTINUOUS INTEGRATION, like everything else in domain-driven design, operates at two levels: the integration of model concepts, and the integration of the implementation.

Concepts are integrated by constant communication among team members. The team must have very high communication, promoting a shared understanding of the ever-changing model. Many practices help, but the most fundamental is constantly hammering out the UBIQUITOUS LANGUAGE. Meanwhile, the implementation artifacts are being integrated by a systematic merge/build/test process that exposes model splinters early. Many processes for integration are used, but most of the effective ones share these characteristics:

- a step-by-step, reproducible merge/build technique
- automated test suites
- rules that set some reasonably small upper limit on the lifetime of unintegrated changes

The other side of the coin in effective processes, although it is seldom formally included, is conceptual integration:

- constant exercise of the UBIQUITOUS LANGUAGE in discussions of the model and application

Most Agile projects have at least daily merges of each developer's code changes. The frequency can be adjusted to the pace of change, as long as any unintegrated change would be merged before a significant amount of work incompatible work could be done by other team members.

In a MODEL-DRIVEN DESIGN, the integration of concepts smoothes the way for the integration of the implementation, while the integration of the implementation proves the validity and consistency of the model, and exposes splinters.

Therefore,

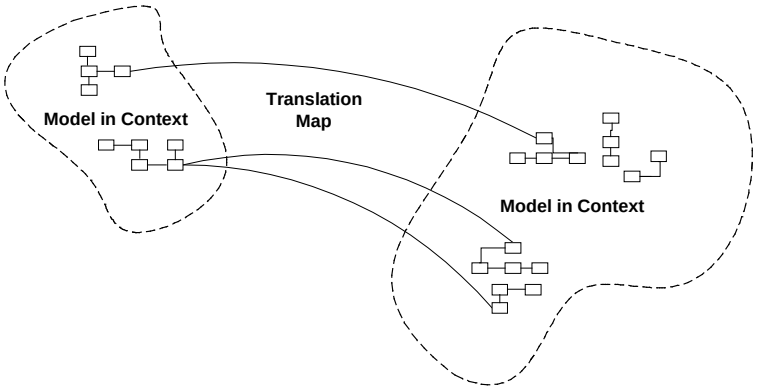
Institute a process of merging all code and other implementation artifacts frequently, with automated tests to flag fragmentation quickly. Relentlessly exercise the UBIQUITOUS LANGUAGE to hammer out a shared view of the model as the concepts evolve in different people's heads.

Finally, do not make the job any bigger than it has to be. CONTINUOUS INTEGRATION is only essential within a BOUNDED CONTEXT. Design issues involving neighboring CONTEXTS, including translation, don't have to be dealt with at the same pace.

♦ ♦ ♦

CONTINUOUS INTEGRATION would be applied within any individual BOUNDED CONTEXT that is larger than a two-person task and maintains integrity of that single model. When multiple BOUNDED CONTEXTS coexist, you have to decide on their relationships and design any necessary interfaces…

CONTEXT MAP



An individual BOUNDED CONTEXT leaves some problems in the absence of a global view. The context of other models may still be vague and in flux.

People on other teams won't be very aware of the context bounds, and unknowingly will make changes that blur the edges or complicate the interconnections. When connections must be made between different contexts, they tend to bleed into each other.

Code reuse between BOUNDED CONTEXTS is a hazard to be avoided. Integration of functionality and data must go through a translation. You can reduce confusion by defining the relationship between the different contexts and creating a global view of all the model contexts on the project.

A CONTEXT MAP is in the overlap between project management and software design. The natural course of events is for the boundaries to follow the contours of team organization. People who work closely will naturally share a model context. People on different teams, or those that don't talk, even if they are on the same team, will spit off into different contexts. Physical office space can have an impact too, as team members on opposite ends of a building, not to mention different cities, will probably diverge without extra integration effort. Most project managers intuitively recognize these factors and broadly organize teams around subsystems. But the interrelationship between team organization and software model and design is still not prominent enough. Both managers and team members need a clear view into of the ongoing conceptual subdivision of the software model and design.

Identify each model in play on the project and define its BOUNDED CONTEXT. This includes the implicit models of non-object-oriented subsystems. Name each BOUNDED CONTEXT, and make the names part of the UBIQUITOUS LANGUAGE.

Describe the points of contact between the models, outlining explicit translation for any communication and highlighting any sharing.

Map the *existing* terrain. Take up transformations later.

Within a BOUNDED CONTEXT, you will have a coherent dialect of the UBIQUITOUS LANGUAGE. The names of the BOUNDED CONTEXTS will themselves enter that LANGUAGE so that you can speak unambiguously about the model of any part of the design by making your CONTEXT clear.

The map does not have to be documented in any particular form. I find conceptual diagrams like the ones in this chapter to be helpful in visualizing and communicating the map. Others may prefer a more textual description or different graphical representation. In some situations, discussion among teammates may be sufficient. The level of detail can vary according to need. Whatever form it takes, it must be shared and understood by everyone on the project. It must provide a clear name for each BOUNDED CONTEXT, and it must make the points of contact and their natures clear.

Later in this chapter, various patterns of relationships between contexts will be explored that are effective in different situations, and which can provide names to describe the relationships you find. Keeping in mind that the CONTEXT MAP always represents *the situation as it stands*, the relationships you find may not fit these patterns initially. If they fall close, you may wish to use the name, but don't force it. Just describe the relationships you find. Later you can begin to migrate toward more standardized relationships.

So, what do you do if you've discovered a splinter – a model that is completely entangled but contains inconsistencies? Put a dragon on the map and finish describing everything. Then, with an accurate global view, address the points of confusion. A minor splinter can be repaired, and processes can be put in place to shore it up. If a relationship is vague, you can choose the nearest pattern and move toward it. Your first order of business is to arrive at a clear CONTEXT MAP, and this may mean fixing real problems you have found. But don't let this necessary repair lead to wholesale reorganization. Until you have an unambiguous CONTEXT MAP that places all your work into some BOUNDED CONTEXT, with explicit relationships between all connected models, change only the outright contradictions.

Once you have a coherent CONTEXT MAP, you'll see things you want to change. You can make considered changes to the organization of teams or to the design. Remember, don't change the map until the change in reality is *done*.

Example: Two CONTEXTS in a Shipping Application

We return again to the shipping system. One of the application's major features was to be automatic routing of cargos at booking time. The model was something like this:

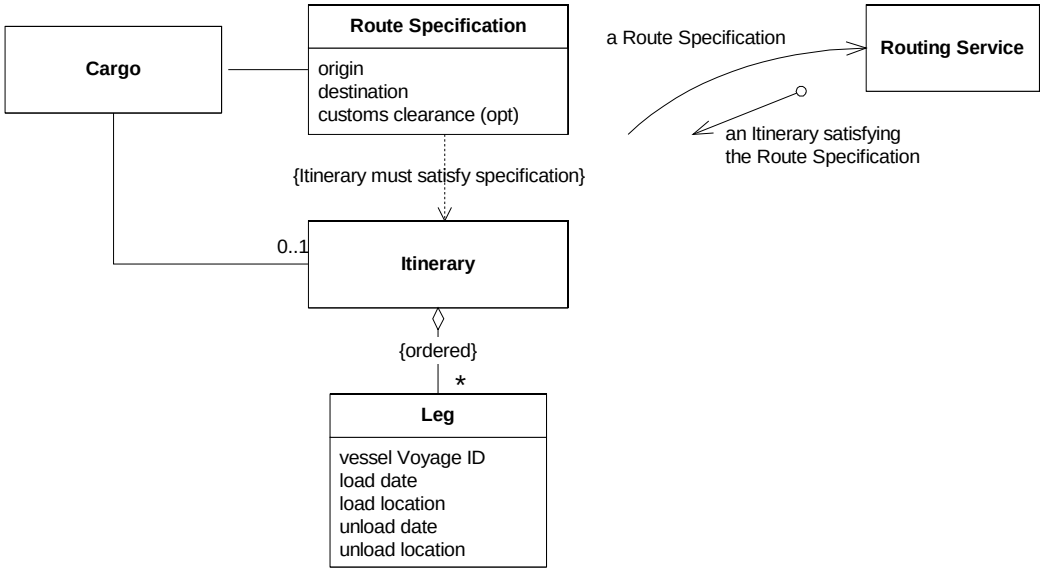


Figure 14. 1

The **Routing Service** is a SERVICE that encapsulates a mechanism behind an INTENTION-REVEALING INTERFACE made up of SIDE-EFFECT-FREE FUNCTIONS. The results of those functions are characterized with ASSERTIONS. The interface declares that:

1. The interface declares that when a **Route Specification** is passed in, an **Itinerary** will be returned.
2. The ASSERTION states that the returned **Itinerary** will satisfy the **Route Specification** passed in.

Nothing is stated about *how* this very difficult task is performed. Now lets go behind the curtain to see the mechanism.

Initially, on this project, I was too dogmatic about the internals of the **Routing Service**. I wanted the actual routing operation to be done with an extended domain model that would represent vessel voyages and directly relate them to the **Legs** in the **Itinerary**. But the team working on the routing problem pointed out that, to make it perform well and in order to draw on well established algorithms, it needed to be implemented as an optimized network with each leg of a voyage represented as an element in a matrix. They insisted that they needed a distinct model of shipping operations for this purpose.

They were clearly right about the computational demands of the routing process as then designed, and so, lacking any better idea, I yielded. In effect, we created two separate BOUNDED CONTEXTS, each of which had its own conceptual organization of shipping operations.

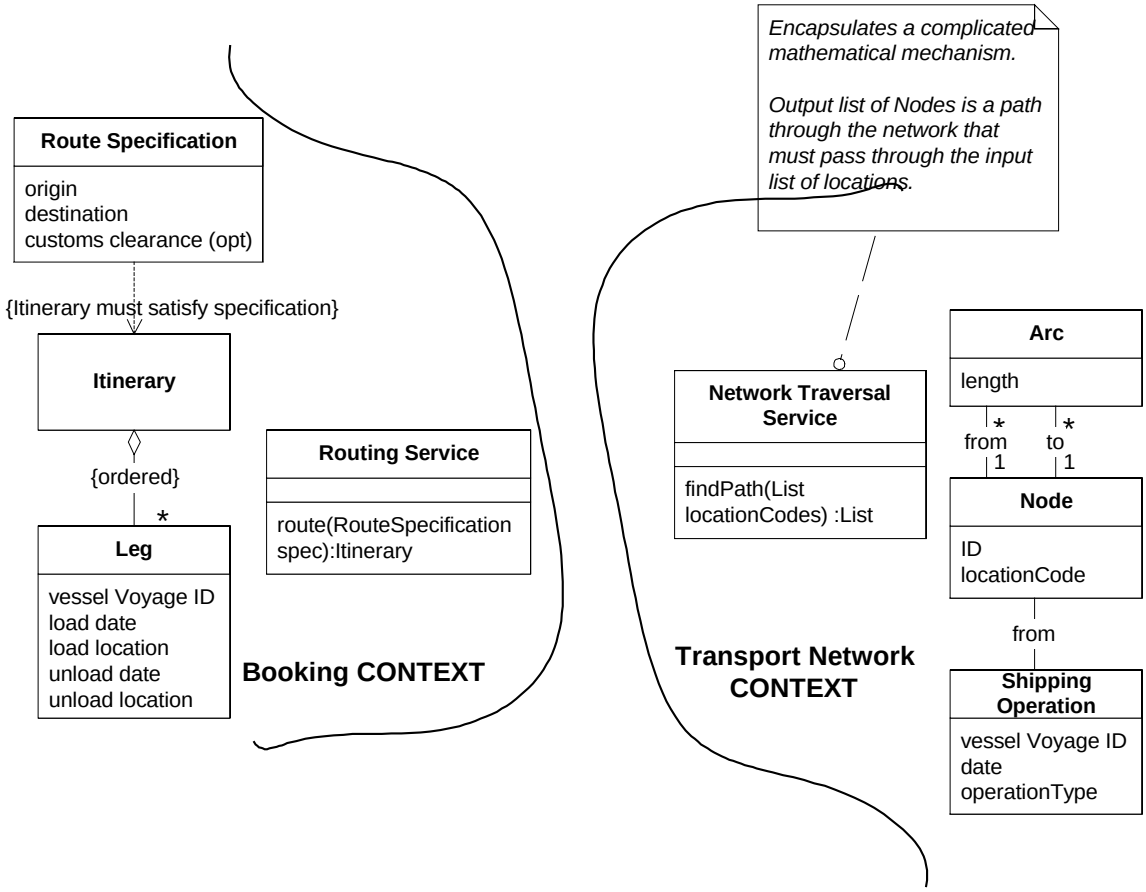


Figure 14. 2

Our requirement was to take a **Routing Service** request, translate it into terms the Graph **Traversal Service** could understand, then take the result and translate it into the form a **Routing Service** is expected to give.

This means it was not necessary to map everything in these two models, but only to be able to make two specific translations:

Route Specification → **List** of location codes

List of **Node** IDs → **Itinerary**

To do this, we have to look at the meaning of an element of one model and figure out how to express it in terms of the other.

Starting with the first (**Route Specification** → **List** of location codes), we have to think about the meaning of the sequence of locations in the list. The first in the list will be the beginning of the path, which will then be forced to pass through each location in turn until it reaches the last location in the list. So the origin and destination are the first and last in the list, with the customs clearance location (if there is one) in the middle.

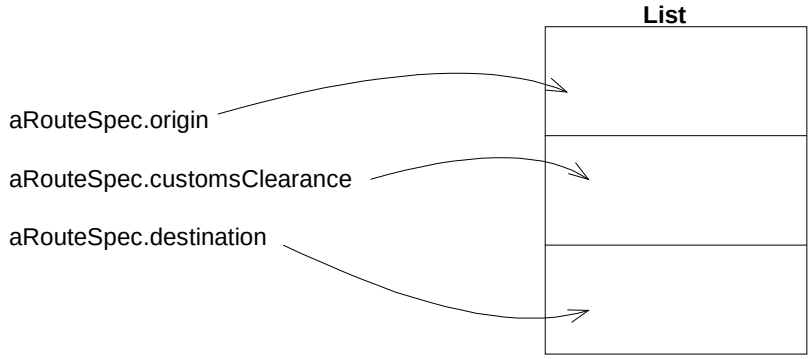


Figure 14. 3

(Mercifully, the two teams used the same location codes, so we don't have to deal with that level of translation.)

Notice that the reverse translation would be ambiguous, since the network traversal input allows any number of intermediate points, not just one specifically designated as customs clearance point. Fortunately, this is no problem because we don't need to translate in that direction, but it gives a glimpse of why some translations are impossible.

Now, let's do result translation (**List** of **Node** IDs → **Itinerary).** We'll assuming we can use a REPOSITORY to look up the **Node** and **Shipping Operation** objects based on the **Node** IDs we receive. So, how do those **Nodes** map to **Legs**? Based on the `operationTypeCode`, we can break the list of **Nodes** into departure/arrival pairs. Each pair then relates to one **Leg**.

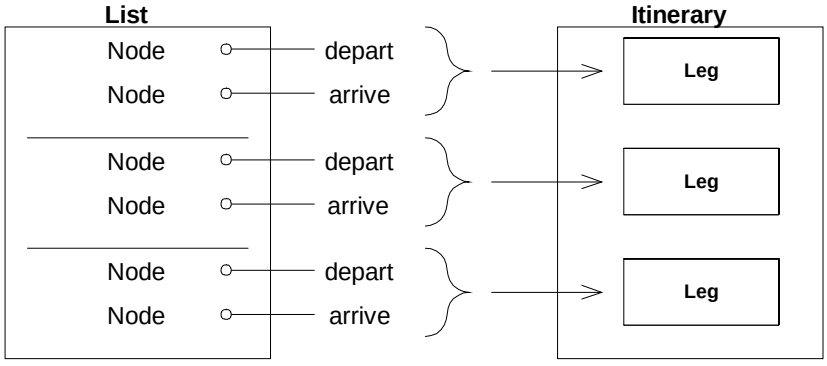


Figure 14. 4

The attributes for each **Node** pair would be mapped as:

```
departureNode.shippingOperation.vesselVoyageId → leg.vesselVoyageId
departureNode.shippingOperation.date → leg.loadDate
departureNode.locationCode → leg.loadLocationCode
arrivalNode.shippingOperation.date → leg.unloadDate
arrivalNode.locationCode → leg.unloadLocationCode
```

This is the conceptual translation map between these two models. Now we have to implement something that can do the translation for us. In a simple case like this, I typically creating an object for the purpose, and then find or create another object to provide the service to the rest of our subsystem.

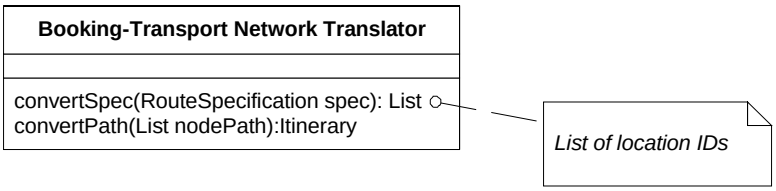


Figure 14. 5

This is the one object that both teams have to work together to maintain. The design should make it very easy to unit test, and it would be a particularly good idea for the teams to collaborate on a test suite for it. Other than that, they can go their separate ways.

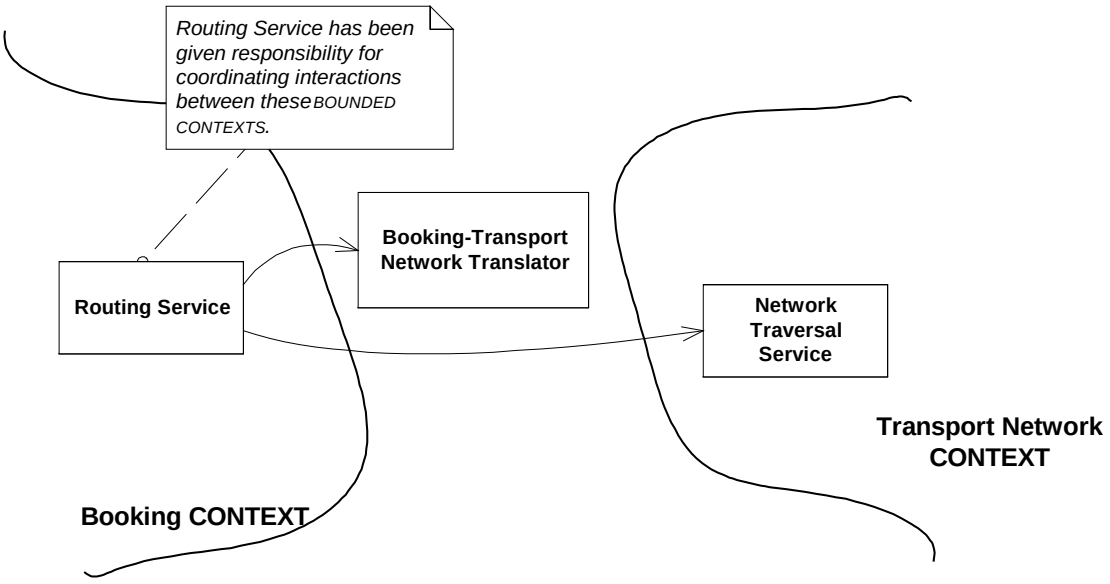


Figure 14. 6

The **Routing Service** implementation now becomes a matter of delegating to the Translator and the Graph Traversal Service. It's single operation would look something like:

```
public Itinerary route(RouteSpecification spec) {
   Booking_TransportNetwork_Translator translator =
      new Booking_TransportNetwork_Translator();
```

```
    List constraintLocations = translator.convertConstraints(spec);

    // Get access to the NetworkTraversalService
    List pathNodes =
       traversalService.findPath(constraintLocations);

    Itinerary result = translator.convert(pathNodes);
    return result;
}
```

Not bad. The BOUNDED CONTEXTS served to keep each of the models relatively clean, let the teams work largely independently, and, if initial assumptions had been correct, would probably have served well. (We'll return to that later in this chapter.)

The interface between the two contexts is fairly small. The interface of the **Routing Service** insulates the rest of the Booking CONTEXT's design from events in the route finding world. The interface is easy to test because it is made up of SIDE-EFFECT-FREE FUNCTIONS. One of the secrets to comfortable coexistence with other CONTEXTS is to have effective sets of tests for the interfaces. "Trust, but verify," said President Reagan when negotiating arms reductions.

It should be easy to devise a set of automated tests that would feed **Route Specifications** into the **Routing Service** and check the returned **Itinerary**.

♦ ♦ ♦

Model contexts always exist, but may overlap and fluctuate. By explicitly defining BOUNDED CONTEXTS your team can begin to direct the process of unifying models and connecting distinct ones.

Testing at the Context Boundaries

Contact points with other BOUNDED CONTEXTS are particularly important to test because this helps compensate for the subtleties of translation and the lower level of communication that typically exists. It is a valuable early warning system, especially reassuring in cases where you depend on details of a model you don't control.

Organizing and Documenting CONTEXT MAPS

There are only two important points here:

1. The BOUNDED CONTEXTS should have names so that you can talk about them. Those names should enter the UBIQUITOUS LANGUAGE of the team.
2. Everyone has to know where the boundaries lie including being able to recognize the CONTEXT of any piece of code or any situation.

The second requirement could be met in many ways depending on the culture of the team. Once the BOUNDED CONTEXTS have been defined, it comes naturally to segregate the code of different CONTEXTS into different MODULES, which leaves the question of how to keep track of which MODULE belongs in which CONTEXT. A naming convention might be used to indicate this, or any other mechanism that is easy and avoids confusion.

Equally important is communicating the conceptual boundaries in such a way that everyone on the team understands them the same way. For this communication purpose, I like informal diagrams like the ones in the example. More rigorous diagrams or textual lists could be made showing all packages in each CONTEXT along with the points of contact and the mechanisms responsible for connecting and translating. Some

teams will be more comfortable with this approach, while others will get by fine based on verbal agreement and lots of discussion.

In any case, working the CONTEXT MAP into discussions is essential if the names are to enter the UBIQUITOUS LANGUAGE. Don't say, "George's team's stuff is changing so we're going to have to change our stuff that talks to it." Say instead, "The *Transport Network* model is changing so we're going to have to change the *translator* for the *Booking context*."

Relationships Between BOUNDED CONTEXTS

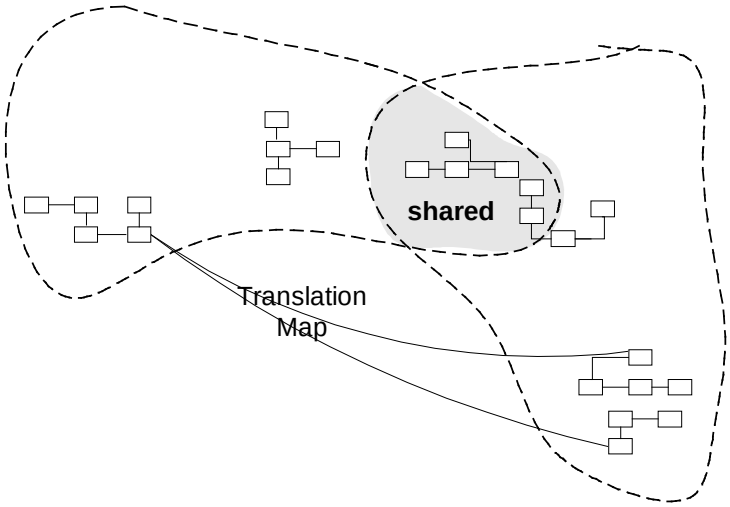
The following patterns cover a range of strategies for relating two models that can be composed to encompass an entire enterprise. These served the dual purpose of providing targets for successfully organizing development work, and providing vocabulary for describing the existing organization.

An existing relationship may, by chance or by design, fall near one of these patterns, in which case you can describe it using that term, variations duly noted. Then, with each small design change, the relationship can be drawn closer to the chosen pattern.

On the other hand, you may find an existing relationship is muddled or overcomplicated. Some reorganization might be necessary just to make an unambiguous CONTEXT MAP possible. In this situation, or an any situations in which you are considering reorganization, these patterns present a range of choices that are favored in different circumstances. Prominent variables include the level of control you have over the other model, and the level and type of cooperation between teams, and the degree of integration of features and data.

The following set of patterns cover some of the most common and important cases, which should give a good idea of how to approach other cases. A crack team working closely on a tightly integrated product you can deploy a large unified model. The need to serve different user communities or a limitation on the coordination abilities of the team might lead to a SHARED KERNEL or CUSTOMER-SUPPLIER relationships. Sometimes a good hard look at the requirements reveals that integration is not essential and it is best for systems to go their SEPARATE WAYS. And, of course, most projects have to integrate to some degree with legacy and external systems, which can lead to OPEN HOST SERVICES or ANTICORRUPTION LAYERS…

SHARED KERNEL



… When functional integration is limited, the overhead of CONTINUOUS INTEGRATION may be deemed too high. This may especially be true when the teams do not have the skill or the political organization to maintain continuous integration, or when a single team is simply too big and unwieldy. So separate BOUNDED CONTEXTS might be defined and multiple teams formed.

Uncoordinated teams working on closely related applications can go racing forward for a while, but what they produce may not fit together. They can end up spending more on translation layers and retrofitting than they would have on CONTINUOUS INTEGRATION in the first place, meanwhile duplicating effort and losing the benefits of a common UBIQUITOUS LANGUAGE.

On many projects I've seen the infrastructure layer shared among teams that worked largely independently. An analogy to this can work well within the domain as well. It may be too much overhead to fully synchronize the entire model and code base, but a carefully selected subset can provide much of the benefit for less cost.

Therefore,

Designate some subset of the domain model that the two teams agree to share. Of course this includes, along with this subset of the model, the subset of code or of the database design associated with that part of the model. This explicitly shared stuff has special status, and shouldn't be changed without consultation with the other team.

Integrate a functional system frequently, but somewhat less than the pace of CONTINUOUS INTEGRATION within the teams. At these integrations, run the tests of both teams.

It is a careful balance. The SHARED KERNEL cannot be changed as freely as other stuff. Decisions involve consultation with another team. Automated test suites must be integrated because all tests of both teams must pass when changes are made. Usually, teams make changes on separate copies of the KERNEL, integrating with the other team at intervals. (For example, on a team that CONTINUOUSLY INTEGRATES daily ore better, the kernel merger might be weekly). As usual, communication is a good thing, and the sooner both teams talk about the changes the better.

The SHARED KERNEL is often the CORE DOMAIN and/or some set of GENERIC SUBDOMAINS, but it can be any part of the model that is needed by both teams. The goal is to reduce duplication (but not to eliminate it, as would be the case if there were just one BOUNDED CONTEXT) and make integration between the two subsystems relatively easy.

… Often functionality is partitioned such that one subsystem essentially feeds another, while the second performs analysis or other functions that don't feed back into the first very much. It is also common in these cases that the two subsystems serve very different user communities, with different jobs, where different models may be useful. The tool set may also be completely different, meaning that program code cannot be shared.

♦ ♦ ♦

Upstream and downstream subsystems, where the downstream component takes the output from the upstream component and all dependencies go one way, separate naturally into two BOUNDED CONTEXTS. Translation is easier for having to operate in one direction only. This is especially true when the two components require different skills or employ a different tool set for implementation. But there are problems that can emerge in this situation, depending on the political relationship of the two teams.

The freewheeling development of the upstream team can be cramped if the downstream has veto power over changes or if procedures for requesting changes are too cumbersome. The upstream team may even be inhibited by worries of breaking the downstream system. Meanwhile, the downstream team can be helpless, at the mercy of upstream priorities.

Downstream needs things from upstream, but upstream is not responsible for downstream deliverables. It takes a lot of extra effort to anticipate what will affect the other team, and human nature being what it is, and time pressures being what they are, well… It makes everyone's life easier to formalize the relationship between the teams and organize the system for balancing the needs of the two user communities and scheduling work on features needed downstream.

On an Extreme Programming project, there already is a mechanism in place for doing just that: the iteration planning process. All we have to do is define the relationship between the two teams in the terms of the planning process. Representatives of the downstream team can function much like the user representatives, joining them in planning sessions, discussing the tradeoffs for the tasks they want directly with the other customers. The result is an iteration plan for the supplier team that includes tasks the downstream team most needs or defers tasks by agreement, so there is no expectation of delivery.

If a process other than XP is used, whatever analogous method is used to balance the concerns of different users can be expanded to include the downstream application's needs.

Therefore,

Establish a clear customer/supplier relationship between the two teams. In planning sessions, make the downstream team play the customer role to the upstream team. Negotiate and budget tasks for downstream requirements so that everyone understands the commitment and schedule.

Jointly develop automated acceptance tests that will validate the interface they expect. Add these tests to the upstream team's test suite to be run as part of their continuous integration. This will free the upstream team to make changes without fear of side effects downstream.

During the iteration, the downstream team members need to be available to the upstream developers just as the conventional customer is, to answer questions and help resolve problems.

Automating the acceptance tests is a vital part of this customer relationship. Even on the most cooperative project without tests, though the customer can identify its dependencies and communicate them, and the supplier tries to communicate changes, surprises will happen that will disrupt the downstream team's work and force the upstream to take on unscheduled emergency fixes. Instead, have the customer team, in collaboration with the supplier team, develop automated acceptance tests that will validate the interface they expect. The upstream team will run these tests as part of their standard test suite. Any change to these tests calls for communication, since it implies change in interface.

I've also seen customer/supplier relationships emerge between projects in separate companies, in situations where a single customer is very important to the business of the supplier. Then the tail can wag the dog and get what they need. Both parties can benefit from the formalization of the process, since the cost/benefit tradeoffs are even harder to see than with the internal customers in the internal IT situation.

There are two crucial elements to this pattern.

1. The relationship must be that of customer and supplier, with the implication that the customer's needs are paramount. Since the downstream application is not the only customer, their needs will have to be balanced, but they are still priorities. This is in contrast to the poor cousin relationship that often emerges, where the downstream has to come begging to the upstream team for their needs.
2. There needs to be an automated test suite that can give the upstream team the ability to change their code without fear of breaking the downstream, and lets the downstream team concentrate on their own work without constantly monitoring the upstream team.

In a relay race, the forward runner can't be looking behind himself all the time, checking. He or she has to be able to trust the baton carrier to make the handoff precisely, or the team will be hopelessly slowed down.

Example: Yield Analysis vs. Booking

Back to our trusty shipping example. A highly specialized team has been set up analyze the all the bookings that flow through to see how to maximize income. They might find that ships have empty space and recommend overbooking more. They might find that the ships are filling up with bulk freight early, forcing the company to turn away more lucrative specialty cargos, and recommend reserving space for these, or recommend raising prices on the bulk freight.

To do this analysis, they use their own complex models. For implementation, they use a data warehouse with tools for building analytical models. And they need lots of information from the booking application.

From the start, it is clear that these are two BOUNDED CONTEXTS, since they use different implementation tools, and, most importantly, different domain models. What should the relationship be between them?

A SHARED KERNEL might seem logical, since yield analysis is interested in a subset of the booking's model, and their own model has some overlapping concepts of cargos, prices, etc. But SHARED KERNEL is difficult in a case where different implementation technologies are being used. Besides, the modeling needs of the yield analysis team are quite

specialized, and they continuously play with their models and try alternative ones. They would really be better off translating what they need from the booking CONTEXT into their own. (On the other hand, if they can use a SHARED KERNEL, their translation burden will be much lighter. They will still have to reimplement the model and translate the data to the new implementation, but if the model is the same, the transfer should be simple.)

The booking application has no dependency on the yield analysis, since there is no intention of automatically adjusting policies. Human specialists will make the decisions and convey them to the needed people and systems. So we have an upstream/downstream relationship. What they need is:

1. Some data not needed by any booking operation
2. Some stability in database schema (or at least reliable notification of change) or an export utility.

Fortunately, the project manager of the booking application development team is motivated to help the yield analysis team. This could have been a problem, since the operations department that actually does day-to-day booking reports to a different vice president than the people who actually do yield analysis. But the upper management cares deeply about yield management and, having seen past cooperation problems between the two departments, structured the software development project so that the project managers of both teams report to the same person.

Therefore, all the requirements are in place to apply CUSTOMER/SUPPLIER DEVELOPMENT TEAMS.

I've seen this scenario evolve in multiple places, where analysis software developers and operations software developers had a customer-supplier relationship. When the upstream thought of their role as serving a customer, things worked out pretty well. It was almost always organized informally, and in each case it worked out about as well as the personal relationship of the two project managers.

On one XP project, I saw this formalized in the sense that, each iteration, representatives of the downstream team played the "planning game" in the role of customers, huddling with the more conventional customer representatives (of application functionality) to negotiate which tasks made it into the iteration plan. This project was at a small company, and so the nearest shared boss was not far up the chain.

♦ ♦ ♦

CUSTOMER/SUPPLIER TEAMS are more likely to work if the two teams work under the same management so that ultimately they do share goals, or where they are in different companies that actually have those roles. When there is nothing to motivate the upstream team, the situation is very different…

CONFORMIST

When two teams with an upstream/downstream relationship are not effectively being directed from the same source, such a cooperative pattern as CUSTOMER/SUPPLIER TEAMS is not going to work. Naively trying to apply it will get the downstream team into trouble. This can be the case in a large company in which the two teams are far apart in the hierarchy or where the shared management level is indifferent to the relationship of the two teams. It can also arise when the two teams are in different companies where the downstream team's company really is a customer to the upstream team's company, but where that particular customer is not individually important to the supplier. This could be because the supplier has many small customers, or because they are changing market direction and no longer value the old customers, or just because they are poorly run, or even out of business. Whatever the reason, the reality is, the downstream is on its own.

When two development teams have an upstream/ downstream relationship in which the upstream has no motivation to provide for the downstream team's needs, the downstream team is helpless. Altruism may lead upstream developers into making promises, but they are unlikely to be fulfilled. Belief in those good intentions leads the downstream team to make plans based on features that will never be available. Their project will be delayed until they ultimately learn to live with what they are given. An interface tailored to the needs of the downstream team is not in the cards.

In this situation, there are three possible paths. One is to abandon use of the upstream altogether. This should be evaluated realistically, making no assumptions that the upstream will accommodate downstream needs. Sometimes we overestimate the value or underestimate the cost of such a dependency. If the downstream team decides to cut the strings, they are going their SEPARATE WAYS (p. ##).

Sometimes the value is so great that the upstream dependency has to be maintained (or a political decision has been made that the team cannot change). In this case, two paths remain open, and which is taken depends on the quality and style of the upstream design. If the design is very difficult to work with, perhaps for lack of encapsulation, awkward abstractions, or modeling in a paradigm the team cannot use, then the downstream team will still need to develop its own model. They will have to take full responsibility for a translation layer that is likely to be complex. (See ANTICORRUPTION LAYER, p. ##).

On the other hand, if the quality is not so bad, and the style is reasonably compatible, then it may be best to give up on an independent model altogether. This is the circumstance that calls for a CONFORMIST.

Eliminate the complexity of translation between BOUNDED CONTEXTS by slavishly adhering to the model of the upstream team. This cramps the style of the downstream designers. It probably is not going to be the ideal model for the application. But it is a huge reduction in complexity. Plus, you will share a UBIQUITOUS LANGUAGE with your supplier team. They are in the driver's seat, so it is good to make communication easy for them. Altruism may be sufficient to get them to share information with you.

This decision deepens the dependency, and limits your application to the capabilities of the upstream model plus purely additive enhancements. It is very unappealing emotionally, which is why we choose it less than we probably should.

If these tradeoffs are not acceptable, but the upstream dependency is indispensable, the third option is to insulate yourself as much as possible by creating an ANTICORRUPTION LAYER, an aggressive approach to implementing a translation map that will be discussed later.

<<SIDEBAR BEGIN>>

Following isn't always bad.

When using an off-the-shelf component that has a large interface, you should typically CONFORM to the model implicit in that component. In other words, the model used in the application should be unified with the component's model. But since they are clearly different BOUNDED CONTEXTS, based on team organization and control, the only way to accomplish this unification is by making the BOUNDED CONTEXT of the application CONFORM to that of the component. Even though adapters may be needed for minor format changes, the model should be equivalent. Otherwise, you should question the value of having the component. If it is good enough to give you value, there is probably knowledge crunched into its design. There must be more to your model than this one thing, and those parts you evolve. But where they connect, your model is a CONFORMIST, following the lead of the component's model. In effect, you could be dragged into a better design.

When your interface with a component is small, sharing a unified model is less essential, and translation is a viable option. But when the interface is large and integration is more significant, it usually makes sense to follow the leader.

<<SIDEBAR END>>

♦ ♦ ♦

CONFORMIST resembles SHARED KERNEL in that they both have an overlapping area where the model is the same, areas where you have extended your model by addition and areas where the other model does not affect you. The difference between the patterns is in the decision making process and development process. Where the SHARED KERNEL is a collaboration between two teams that coordinate tightly, CONFORMIST deals with integration with a team that is not interested in collaboration.

We've been proceeding down a spectrum of cooperation in the integration between BOUNDED CONTEXTS, from highly cooperative SHARED KERNELS or CUSTOMER/SUPPLIER DEVELOPER TEAMS, to the one-sidedness of the CONFORMIST. Now we'll take the final step to an even more pessimistic view of the relationship, assuming neither cooperation nor a usable design on the other side…

ANTICORRUPTION LAYER

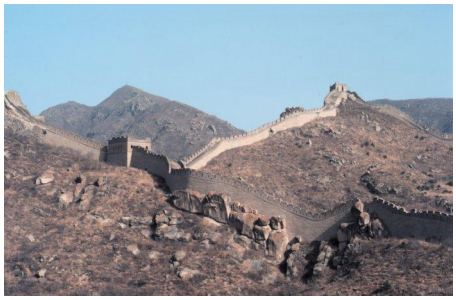


Photo: US National Oceanic and Atmospheric Administration

…New systems almost always have to be integrated with legacy or other systems that will have their own models. When control or communication is not adequate to pull off a SHARED KERNEL or CUSTOMER/SUPPLIER DEVELOPMENT TEAMS, the interface can become more complex. Translation layers can be simple, even elegant, when bridging well-designed BOUNDED CONTEXTS with cooperative teams. But when the other side of the boundary starts to leak through, the translation layer may take on a more defensive tone.

When a new system is being built that must have a large interface with another, the difficulty of relating the two models can eventually overwhelm the intent of the new model altogether, causing it to be modified to resemble the other system's model, in an ad hoc fashion. The models of legacy systems are usually weak, and even the exception that is well developed may not fit the needs of the current project. Yet there may be a lot of value in the integration, and sometimes it is an absolute requirement.

The answer is not to avoid all integration with other systems. I've been on projects where people's enthusiastically set out to replace all the legacy, but this is just too much to take on at once. Besides, integrating with existing systems is a valuable form of reuse. On a large project, one subsystem will often have to interface with several other, independently developed subsystems. These will reflect the problem domain differently. When systems based on different models are combined, the need for the new system to adapt to the semantics of the other system can lead to a corruption of the new system's own model. Even when the other system is well designed, it is not based on the *same* model as the client. And often the other system is not well designed.

There are many hurdles in interfacing with an external system. For example, the infrastructure layer must provide the means to communicate with another system that might be on a different platform or use different protocols. The data types of the other system must be translated into those of your system. But often overlooked is the certainty that the other system does not use the same conceptual domain model.

It seems clear enough that errors will result if you take some data from one system and misinterpret it in another. You may even corrupt the database. But even so, this problem tends to sneak up on us because we think that what we are transporting between systems is primitive data whose meaning is unambiguous and must be the same on both sides. This is usually wrong. Subtle yet important differences in meaning arise from the way the data are associated in each system. And even if some of the primitive data elements do have exactly the same meaning, it is a usually a mistake to make the interface to the other system operate at such a low level. A low-level interface takes away the power of the other system's model to explain the data and constrain its values and relationships, while saddling the new system with the burden of interpreting primitive data that is not in terms of to its own model.

So, combining systems with different models has pitfalls, but it is nonetheless unavoidable. A means is needed to provide a translation between the parts that adhere to different models, so that the models are not corrupted with undigested elements of foreign models.

Create an isolating layer to provide clients with functionality in terms of their own domain model. The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally the layer translates in both directions as necessary between the two models.

When I talk about a mechanism to link two systems, many will be thinking of the issues of transporting the data from one program to another or from one server to another. I'll discuss the incorporation of the technical communications mechanism shortly, but it shouldn't be confused with an ANTI-CORRUPTION LAYER, which is not a mechanism for sending messages to another system. It is a mechanism that translates conceptual objects and actions from one model and protocol to another.

An ANTICORRUPTION LAYER can become a complex piece of software in its own right. Next I'll outline some of the design considerations for creating one.

Designing the Interface of the ANTICORRUPTION LAYER

The public interface of the ANTICORRUPTION LAYER usually appears as a set of SERVICES, although occasionally it can take the form of an ENTITY. Building a whole new layer responsible for the translation between the semantics of the two systems gives us an opportunity to reabstract the other system's behavior and offer its services and information to our system consistently with our model. It may not even make sense, in our model, to represent the external system as a single component. It may be best to use multiple SERVICES (or occasionally ENTITIES), each of which has a coherent responsibility in terms of our model

Implementing the ANTICORRUPTION LAYER

One way of organizing the design of the ANTICORRUPTION LAYER is as a combination of FAÇADES, ADAPTERS [both from GHJV95], and translators, along with the communication and transport mechanisms usually needed to talk between systems.

Many of the systems we have to integrate with have large complicated messy interfaces. This is not the same problem as the conceptual model differences that motivate the use of ANTICORRUPTION LAYERS, but it is a problem you'll encounter trying to create them. Translating from one model to another (especially if one model is fuzzy) is a hard enough job without simultaneously dealing with a subsystem interface that is hard to talk to. Fortunately, that is what FACADES are for.

A FAÇADE is an alternative interface for a subsystem that simplifies access for the client and makes the subsystem easier to use. Since we know exactly what functionality of the other system we want to use, we can create a FAÇADE that facilitates and streamlines access to those features and hides the rest. The FAÇADE does *not* change the model of the underlying system. It should be written strictly in accordance with the other system's model. Otherwise, you will, at best, diffuse responsibility for translation into multiple objects and overload the FAÇADE, and, at worst, end up creating yet another model that doesn't belong to the other system or your own BOUNDED CONTEXT. The FAÇADE belongs in the BOUNDED CONTEXT of the other system. It just presents a friendlier face specialized for your needs.

An ADAPTER is a wrapper that allows a client to use a different protocol that understood by the implementer of the behavior. When a client sends a message to an ADAPTOR, it is converted to a semantically equivalent message and sent on to the "adaptee". The response is converted and passed back. I'm using "ADAPTER" a little bit loosely, since the emphasis in [GHJV95] is on making a wrapped object conform to a standard interface that clients expect, whereas we get to choose the adapted interface, and the adaptee is probably not even an object. Our emphasis is on translation between two models, but I think this is consistent with the intent of the pattern.

For each SERVICE we defined, we need an ADAPTOR that supports the SERVICE'S interface and knows how to make equivalent requests of the other system or its FAÇADE.

The remaining element is the translator. The adapter's job is to know how to make a request. The actual conversion of conceptual objects or data is a distinct complex task that can be placed in its own object, making them both much easier to understand. A translator can be a lightweight object instantiated when needed. It needs no state and does not need to be distributed, since it belongs with the adaptor(s) it serves.

Structure of an ANTICORRUPTION LAYER

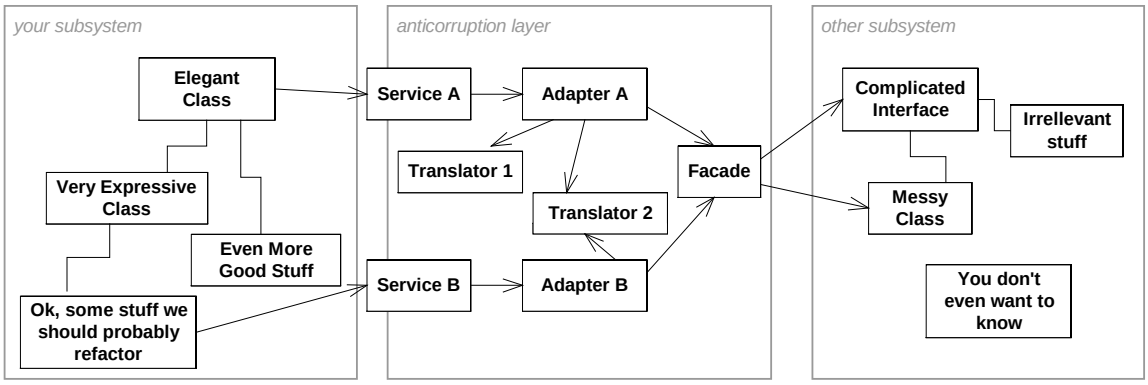


Figure 14. 7

Those are the basic elements I use to create an ANTICORRUPTION LAYER. There are a few other considerations.

- Typically, the system under design (your subsystem) will be initiating action, as implied by this diagram. There are cases, however, when the other subsystem may need to request something of the your subsystem or notify it of some event. An ANTICORRUPTION LAYER can be bi-directional, defining SERVICES on both interfaces with their own ADAPTERS, potentially using the same translators with symmetrical translations. While implementing the ANTICORRUPTION LAYER doesn't usually require any change to the other subsystem, it might be necessary in order to make it call on services.
- You'll usually need some communications mechanism to connect the two subsystems, and they could well be on separate servers. In this case, you have to decide where to place these communication links. If you have no access to the other subsystem, you may have to put the links between the FAÇADE and the other subsystem, but if the FAÇADE can be integrated directly with the other subsystem, then a good option is to put the communication link between the ADAPTER and FAÇADE, since this presumably simplifies those elements. There also would be cases where the entire ANTICORRUPTION LAYER could live with the other subsystem, placing communication links or distribution mechanisms between your subsystem and the SERVICES. These are implementation and deployment decisions to be made pragmatically. They have no bearing on the conceptual role of the ANTICORRUPTION LAYER.
- If you do have access to the other subsystem, you may find that a little refactoring over there can make your job easier. In particular, try to write more explicit interfaces for the functionality you'll be using, starting with automated tests, if possible.
- Where integration requirements are extensive, the cost of translation goes way up. It may be necessary to make choices in the model of the system under design that keep it closer to the external system, in order to make translation easier. Do this very carefully, without compromising the integrity of the model. It is only something to do selectively when translation difficulty gets out of hand. If this seems the most natural solution for much of the important part of the problem, consider using the CONFORMIST pattern.
- If the other subsystem is simple or has a clean interface, you may not need the FAÇADE.
- Functionality can be added to the ANTICORRUPTION LAYER if it is *specific to the relationship of the two subsystems*. An audit trail for use of the external system, or trace logic for debugging the calls to the other interface are two useful features that come to mind.

Remember, an ANTICORRUPTION LAYER is a means of linking two BOUNDED CONTEXTS. Ordinarily, we are thinking of a system created by someone else, which we have incomplete understanding of and little control

over. But that is not the only situation where you need a little padding between subsystems. There are even situations in which it makes sense to connect two subsystems of your own design with an ANTICORRUPTION LAYER, if they are based on different models. Presumably, in that situation you will have full control of both sides and may be able to simplify the layer, but if you have decided on SEPARATE WAYS between two BOUNDED CONTEXTS that still have some need of functional integration, an ANTICORRUPTION LAYER can help you keep each of them internally unified.

Example: The Legacy Shipping Booking Application

In order to have a small quick first release, a minimal application will be written that can set up a shipment and then pass that to the legacy system through a translation layer for booking and support operations. Since we specifically built the translation layer to protect our developing model from the influence of the legacy design, this translation is an ANTICORRUPTION LAYER.

Initially, the ANTICORRUPTION LAYER will take the objects representing a shipment, convert them and pass them to the legacy system and request a booking, capture the confirmation and translate it back into the confirmation object of the new design. This isolation will allow us to develop our new application mostly independently of the old one, though we'll have to invest quite a bit in translation.

With each successive release, the new system can either take over more functions of the legacy or simply add new value without replacing existing capabilities, depending on later decisions. This flexibility, and the ability to continually operate the combined system while gradually transitioning probably makes it worth the expense to build the anticorruption layer.

A Cautionary Tale

To protect their frontiers from raids by neighboring nomadic warrior tribes, the early Chinese built the Great Wall. It was not an impenetrable barrier, but it allowed a regulated commerce with neighbors while providing an impediment to invasion and other unwanted influence. For two thousand years it defined a boundary that helped the Chinese agricultural civilization to define itself with less disruption from the chaos outside.

Although China might not have become so distinct a culture without the Great Wall, the Wall's construction was immensely expensive and bankrupted at least one dynasty, probably contributing to its fall. The benefit of isolation strategies must be balanced against their cost. There is a time to be pragmatic and make measured revisions to the model to make a smoother fit to the foreign ones.

♦ ♦ ♦

There is overhead of any integration, from full-on CONTINUOUS INTEGRATION inside a single BOUNDED CONTEXT, through the lesser commitments of SHARED KERNELS or CUSTOMER/SUPPLIER DEVELOPER TEAMS, to the one-sidedness of the CONFORMIST and the defensive posture of ANTICORRUPTION LAYER. Integration can be very valuable, but it is always expensive. We should be sure it is really needed…

SEPARATE WAYS

We must be ruthless when it comes to defining requirements. If two sets of functionality have no significant relationship, they can be completely cut loose from each other.

Integration is always expensive, and sometimes the benefit is small.

In addition to the usual expense of coordinating teams, there are the compromises that have to be made. The simple specialized model that can satisfy a particular need must give way to the more abstract model that can handle all situations. Or perhaps some completely different technology could provide certain features very easily, but is very difficult to integrate. Or perhaps some team is just so hard to get along with that nothing works very well when other teams try to collaborate with them.

There are many circumstances where integration provides no significant benefit. If two functional parts do not call upon each other's functionality, or require interactions between objects that are touched by both, or share data during their operations, then integration, even through a translation layer, may not be necessary. Just because features are related in a use-case, does not mean they must be integrated.

Therefore,

Declare a BOUNDED CONTEXT to have no connection to the others at all, allowing developers to find simple specialized solutions within this small scope.

The features can still be organized in middle ware or the UI layer, but there will be no sharing of logic, and an absolute minimum of data transfer through translation layers -- preferably none.

Example: An Insurance Project Slims Down

One project had set out to develop new software for insurance claims that would integrate everything a customer service agent or a claims-adjuster needed into one system. After a year of effort they were stuck. A combination of analysis paralysis with a major upfront investment in infrastructure had found them with nothing to show an increasingly impatient management. More seriously, the scope of what they were trying to do was overwhelming them.

A new project manager forced everyone into a room for a week to form a new plan. First they made lists of requirements and tried to estimate their difficulty and assign importance.

They ruthlessly chopped the difficult and unimportant ones. Then they started to bring order to the remaining list. Many smart decisions were made in that room that week, but in the end, only one turned out to be important. At some point it was recognized that *there were some features for which integration provided little added value.* For example, adjusters needed access to some existing databases, and their current access was very inconvenient. *But, although the users needed to have this data, none of the other features of the proposed software system would use it*.

Team members proposed various ways of providing easy access. In one case, a key report could be exported to html and placed on the intranet. In another case, adjusters could be provided with a specialized query written using a standard software package. All these functions could be integrated by organizing links on an intranet page or by placing buttons on the user's desktop.

The team launched a set of small projects that attempted no more integration than launching from the same menu. Several valuable capabilities were delivered almost overnight. Dropping the baggage of these extraneous features left a distilled set of requirements that seemed for a while to give hope for delivery of the main application.

It could have gone that way, but unfortunately the team slipped back into old habits. They paralyzed themselves again. In the end, their only legacy turned out to be those small applications that had gone their SEPARATE WAYS.

♦ ♦ ♦

Taking SEPARATE WAYS forecloses some options. Although continuous refactoring can eventually undo any decision, it is hard to merge models that have developed in complete isolation. If integration turns out to be needed after all, translation layers will be necessary and may be complex.

Of course, this is something you will face anyway. We've talked about how cooperative teams solve the problem, or how to CONFORM to someone else's design. But sometimes you have neither of those options is available…

OPEN HOST SERVICE

Typically, within a BOUNDED CONTEXT you will define an translation layer for each component that you have to integrate with that is outside the CONTEXT. Where integration is one-off, this approach of inserting a translation layer for each external system avoids corruption of the models with a minimum of cost. When you find your subsystem in high demand, you may need a more flexible approach…

♦ ♦ ♦

When a subsystem has to be integrated with many others, customizing a translator for each can bog down the team. There is more and more to maintain, and more and more to worry about when changes are made.

The team may be doing the same thing again and again. If there is any coherence to the subsystem, it is probably possible to describe it as a set of SERVICES that cover the common needs of other subsystems.

Therefore,

Define a protocol that gives access to your subsystem as a set of SERVICES. Open the protocol so that all who need to integrate with you can use it. Enhance and expand the protocol to handle new integration requirements, except when a single team has idiosyncratic needs. Then, use a one-off translator to augment the open protocol so that the protocol can stay simple and coherent.

It is a lot harder to design a protocol clean enough to be understood and used by multiple teams, so it only pays off when the subsystem's resources can be described as a cohesive set of SERVICES and when there are a significant number of integrations. Under those circumstances, it can make the difference between maintenance mode and continuing development.

♦ ♦ ♦

This formalization of communication implies some shared model vocabulary – the basis of the SERVICE interfaces. This leads to the other subsystems being coupled to the model of the open host, and forces those other teams to learn the particular dialect used by the host team. In some situations, coupling can be reduced and ease of understanding enhanced by use of a well-known PUBLISHED LANGUAGE as this interchange model…

… The translation between the models of two BOUNDED CONTEXTS requires a common language.

When two domain models must coexist and information must pass between them, the translation process itself can become complex and hard to document and understand. If we are building a new system, we often will think that our new model is the best available, and so will think in terms of translating directly into it. But sometimes we are enhancing older systems and trying to integrate them. Choosing one messy model over the other may be choosing the lesser of two evils.

When it is not a one-to-one correspondence, the situation is more complicated. When businesses want to exchange information with one another, how do they do it? It is not only unrealistic to expect one to adopt the domain model of the other; it may be undesirable for both parties. The domain model is developed to solve problems for its users, and that may contain features that needlessly complicate communication with another system. Also, if the model is the communications medium, it cannot be changed freely to meet new needs, but must be very stable to support the ongoing communication role.

Direct translation to and from the existing domain models may not be a good solution. Those models may be overly complex or poorly factored. They are probably undocumented. If one is used as a data interchange language, it essentially becomes frozen and cannot respond to new development needs.

The OPEN HOST SERVICE is a basic use of a standardized protocol for multiparty integration. It employs a model of the domain for interchange between systems, even though that model may not be used internally by those systems. Here we go a step further and publish that language, or find one that is already published. By publish I simply mean that the language is readily available to the community that might be interested in using it, and is sufficiently documented to allow independent interpretations to be compatible.

Recently, the world of e-commerce has become very exited about a new technology, XML, that promises to make interchange of data much easier. There are many technical advantages of XML, but the major advantage is that, through the Document Type Definition (DTD) or XML Schemas it allows the formal definition of a specialized domain language into which data can be translated. Industry groups have begun to form to define a single standard DTD for their industry so that, say, chemical formula information or genetic coding can be communicated between many parties. Essentially they are creating a shared domain model in the form of a language definition.

Therefore,

Use a well-documented shared language that can express the necessary domain information as a common medium of communication, translating as necessary into and out of that language.

The language doesn't have to be created from scratch. Many years ago, I was contracted by a company that had a software product written in Smalltalk that used DB2 to store its data. The company wanted the flexibility to distribute the software to users without a DB2 license and contracted me to build an interface to Btrieve, a lighter-weight database engine that had a free runtime distribution license. Btrieve is not fully relational, but they were only using a small part of DB2's power and were within the lowest common denominator of the two databases. They had built some abstractions on top of DB2 that were in terms of the storage of objects. I decided to use this as the interface for my Btrieve component.

This did work. The software worked, and it smoothly integrated with their system, but since there was no formal specification or documentation of their abstractions of persistent objects, there was a lot of work involved in figuring out the requirements of the component. Also, there wasn't much opportunity for reuse of the component. The new software more deeply entrenched their model of persistence, so that refactoring that would have been much more difficult.

A better way would have been to identify the subset of the DB2 interface that they were using and then support that. The interface of DB2 is made up of SQL and a number of proprietary protocols. While it is very complex, it is tightly specified and thoroughly documented. The complexity would have been

mitigated because only a small subset was being used. If a component had been developed that emulated the necessary subset of the DB2 interface, it could have been very effectively documented for developers simply by identifying the subset. The application it was integrated into already knew how to talk to DB2, so little additional work would have been needed. Future redesign of the persistence layer would have been constrained only to the use of the DB2 subset, just as before the enhancement.

The DB2 interface is an example of a PUBLISHED LANGUAGE. In this case, the two models are not in the business domain, but all the principles apply just the same. Since one of the models in the collaboration is already a PUBLISHED LANGUAGE, there is no need to introduce a third language.

A Published Language for Chemistry

Innumerable programs are used to catalog, analyze, and manipulate chemical formulas in industry and academia. Exchanging data has always been difficult, since almost every program uses a different domain model to represent chemical structures. And, of course, most of them are written in languages, such as FORTRAN, that do not express the domain model very fully. Whenever anyone wanted to share data, they had to unravel the details of the other system's database and work out some sort of translation scheme.

Enter the Chemical Markup Language, a dialect of XML intended as a common interchange language for this domain, developed and managed by a group representing academics and industry [MRL95].

Chemical information is very complex, and various, and changes all the time with new discoveries, so they developed a language that could describe the basics, such as chemical formulas of organic and inorganic molecules, protean sequences, spectra or physical quantities. Here is a tiny sample. It is only vaguely intelligible to non-specialists like myself, but the principle is clear.

```
<CML.ARR ID="array3" EL.TYPE=FLOAT NAME="ATOMIC ORBITAL ELECTRON POPULATIONS"
    SIZE=  30 GLO.ENT=CML.THE.AOEPOPS>
   1.17947   0.95091   0.97175   1.00000   1.17947   0.95090   0.97174   1.00000
   1.17946   0.98215   0.94049   1.00000   1.17946   0.95091   0.97174   1.00000
   1.17946   0.95091   0.97174   1.00000   1.17946   0.98215   0.94049   1.00000
   0.89789   0.89790   0.89789   089789   0.89790   0.89788
</CML.ARR>
```

Now that the language has been published, tools can be developed that would never have been worth the trouble to write before, when they would have only been usable for one database. For example, a Java application, called the "JUMBO Browser", was developed that creates graphical views of chemical structures stored in CML. So that if you put your data in the CML format you'll have access to such visualization tools.

In fact, CML gained a double advantage by using XML, a sort of "published meta-language". The learning curve is shortened by people's previous familiarity with this standard, the implementation is eased by various off-the-shelf tools, such as parsers, and documentation is helped by the many books written on all aspects of handling XML.

Unifying an Elephant

It was six men of Indostan
To learning much inclined,
Who went to see the Elephant
(Though all of them were blind),
That each by observation
Might satisfy his mind.

The *First* approached the Elephant,
And happening to fall
Against his broad and sturdy side,
At once began to bawl:
`God bless me! but the Elephant
Is very like a wall!'

…

The *Third* approached the animal,
And happening to take
The squirming trunk within his hands,
Thus boldly up and spake:
`I see,' quoth he, `the Elephant
Is very like a snake.'

The *Fourth* reached out his eager hand,
And felt about the knee.
`What most this wondrous beast is like
Is mighty plain,' quoth he;
`'Tis clear enough the Elephant
Is very like a tree!'

…

The *Sixth* no sooner had begun
About the beast to grope,
Than, seizing on the swinging tail
That fell within his scope,
`I see,' quoth he, `the Elephant
Is very like a rope!'

And so these men of Indostan
Disputed loud and long,
Each in his own opinion
Exceeding stiff and strong,
Though each was partly in the right,
And all were in the wrong!

…

John Godfrey Saxe (1816-1887), based on a story in the Udana, a Hindu text.

Depending on their goals in interacting with the elephant, the various blind men may still be able to make progress, even if they don't fully agree on the nature of the elephant. If no integration is required, then it doesn't matter that the models are not unified. If they require some integration, they may not actually have to agree on what an elephant is, but they will get a lot of value from merely recognizing that they don't agree. This way, at least they don't unknowingly talk at cross-purposes.

Four Contexts: No Integration

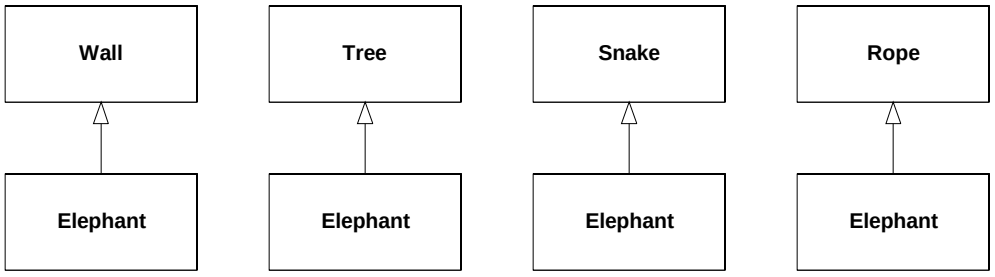
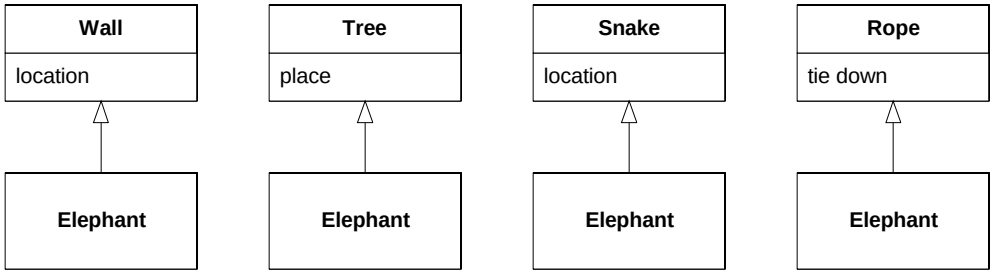


Figure 14. 8

The diagrams above are UML representations of the models the blind men have formed of the elephant. Having established separate BOUNDED CONTEXTS, the situation is clear enough to work out a way to communicate with each other about the few aspects they care about in common – the location of the elephant, perhaps.

Four Contexts: Minimal Integration



Translations: {Wall:location = Tree:place = Snake:location = Rope: tie down}

Figure 14. 9

As the blind men want to share more information about the elephant, the value of sharing a single BOUNDED CONTEXT goes up. But unifying the disparate models is a challenge. None of them is likely to give up his model and adopt one of the others. After all, the man who touched the tail *knows* the elephant is not like a tree, and that model would be meaningless and useless to him. Unifying multiple models almost always means creating a new model.

In this particular case, unification of the various elephant models is easier than most because we

With some imagination and continued discussion (probably heated) the blind men could eventually recognize that they have been describing and modeling different parts of a larger whole. For many purposes, a part-whole unification may not require much additional work. At least the first stage of integration only requires figuring out how the parts are related. It may be adequate for some needs to view an elephant as a wall, held up by tree trunks, with a rope at one end and a snake at the other.

One Context: Crude Integration

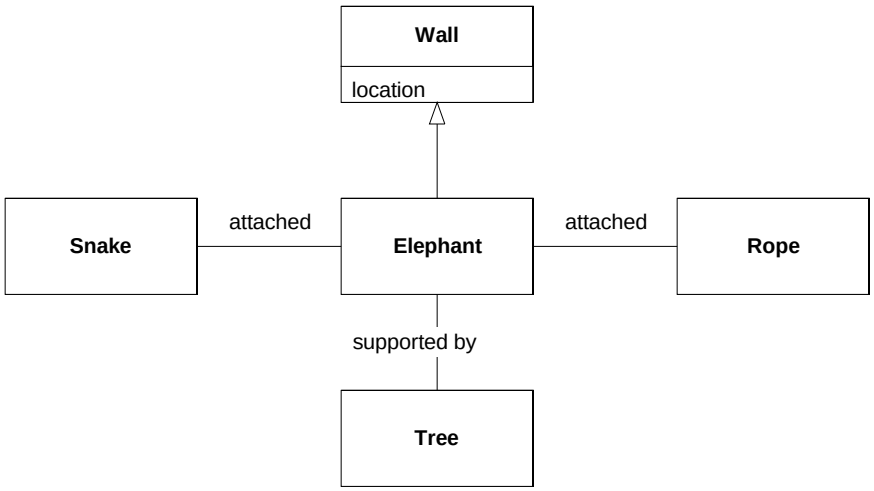


Figure 14. 10

The unification of the various elephant models easier than most such mergers. Unfortunately, it is the exception when two models purely describe different parts of the whole, although there are this is often one aspect of the difference. Matters are more difficult when two models are looking at the same part in a different way. If two men had touched the trunk and one described it as a snake and the other described it as a fire-hose, they would have had more difficulty. If one tried to accept the other's model, he could get into trouble. In fact, they need a new abstraction that incorporates the "aliveness" of a snake with the water

shooting functionality of a fire-hose, but leaves out inappropriate implications of the first models, such as the expectation of possibly venomous fangs, or the ability to detach from the body and roll up into a compartment in a fire truck.

Even though we have combined the parts into a whole, the resulting model is crude and new insights could lead to a deeper model in a process of continuous refinement. New application requirements can also force the move to a deeper model. If the elephant starts moving, the "tree" theory is out, and our blind modelers may break through to the concept of "legs".

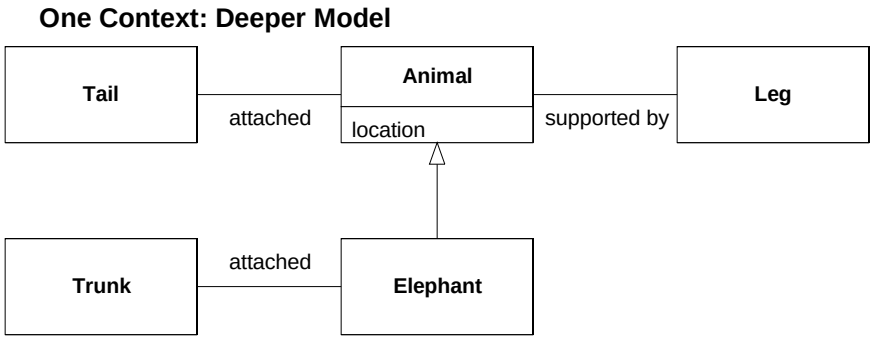


Figure 14. 11

This second pass of model integration tends to slough off incidental or incorrect aspects of the individual models and creates new concepts – in this case, "animal" with parts "trunk", "leg", "body", and "tail" -- that each have their properties and are connected in a particular relationship. Successful model unification, to a large extent, hinges on minimalism. An elephant trunk is both more and less than a snake, but the "less" is probably more important than the "more". Better to lack the water spewing ability than to have an incorrect poison fang feature.

Deep integration between different BOUNDED CONTEXTS is impractical. Translation is limited to those parts of one model that can be rigorously stated in terms of the other model, and even this level of integration may take a considerable effort. This makes sense when there will be a small interface between two systems. If the goal is simply to find the elephant, then translating between each model's expression of location will do.

When more integration is needed, the unified model doesn't have to reach full maturity in the first version. It may be adequate for some needs to view an elephant as a wall, held up by tree trunks, with a rope at one end and a snake at the other. Later, driven by new requirements and by improved understanding and communication, the model can be deepened and refined.

The recognition of multiple clashing domain models is really just accepting reality. By explicitly defining a context within which each model applies, you can maintain the integrity of each, and clearly see the implications of any particular interface you want to create between the two. There is no way for the blind men to see the whole elephant, but their problem would be manageable if only they recognized the incompleteness of their perception.

Choosing Your Model Context Strategy

It is important to always draw the CONTEXT MAP to reflect the current situation at any given time. Once that's done, though, you may very well want to change that reality. Now you will by consciously choosing context boundaries and relationships. Here are some guidelines.

First, teams have to make decisions about where to define BOUNDED CONTEXTS and what sort of relationships to have between them. *Teams* have to make these decisions, or at least the decisions have to be propagated to the entire team and understood by everyone. In fact, they often involve agreements beyond your own team. On the merits, decisions about expanding BOUNDED CONTEXTS or partitioning them

should be made based on the cost-benefit tradeoff of the value of independent action of the team and the value of simple and rich integration. In practice, political relationships between teams are often the deciding factor on what kinds of integration will be possible or forced.

When we are working on a software project, we are primarily interested in the system under design and secondarily in the systems it will communicate with. The system under design is probably going to get carved into one or two BOUNDED CONTEXTS that the main development teams will be working on, perhaps with another context or two in a supporting role. In addition to that are the relationships between these contexts and the external systems.

Transforming the Boundaries

There are an unlimited variety of situations and an unlimited number of options for drawing the boundaries of the BOUNDED CONTEXTS. But typically the struggle is to balance some subset of these forces:

Favoring Larger BOUNDED CONTEXTS

- Flow between user tasks is smoother when more is handled with a unified model.
- It is easier to understand one coherent model than two distinct ones plus mappings.
- Translation between two models can be difficult (sometimes impossible).
- Shared language fosters clear team communication.

Favoring Smaller BOUNDED CONTEXTS

- Communication overhead between developers is reduced.
- CONTINUOUS INTEGRATION is easier in smaller teams and code bases.
- Larger contexts may call for more versatile abstract models, requiring skills that are in short supply.
- Different models can cater to special needs or encompass the jargon of specialized groups of users, along with specialized dialects of the UBIQUITOUS LANGUAGE.

Deep integration of functionality between different BOUNDED CONTEXTS is impractical. Integration is limited to those parts of one model that can be rigorously stated in terms of the other model, and even this level of integration may take a considerable effort. This makes sense when there will be a small interface between two systems.

Accepting That Which We Cannot Change: Delineating the External Systems

It is best to start with the easiest decisions. Some subsystems will clearly not be in any BOUNDED CONTEXT of the system under development. Examples would be major legacy systems that you are not immediately replacing, external systems that provide services you'll need. You can identify these immediately and prepare to segregate them from your design.

Here we must be careful about our assumptions. It is convenient to think of each of these systems as constituting its own BOUNDED CONTEXT, but most external systems only weakly meet the definition. First, a BOUNDED CONTEXT is defined by an *intention* to unify the model within certain boundaries. You may have control of maintenance of the legacy system, in which case you can declare the intention, or the legacy team may be well coordinated and be carrying out an informal form of CONTINUOUS INTEGRATION, but don't take it for granted. Check into it and if the development is not well integrated be particularly cautious. It is not unusual to find semantic contradictions in different parts of such systems.

Relationships With the External Systems

There are three patterns that can apply here. The first to consider is SEPARATE WAYS. Yes, you wouldn't have included them if you didn't need integration. But be really sure. Would it be sufficient to give the user easy access to both systems? Integration is expensive and distracting, so unburden your project as much as you can.

If the integration is really essential, you can choose between two extremes: CONFORMIST or ANTICORRUPTION LAYER. It is not fun to be a CONFORMIST. Your creativity and your options for new functionality will be limited. In building a major new system, it is unlikely to be practical to adhere to model of a legacy or external system (after all, why are you building a new system?) but it may be appropriate in the case of peripheral extensions to a large system that will continue to be the dominant system. Examples of this choice include the lightweight decision-support tools that are often written in Excel or other simple tools. If your application is really an extension to the existing system and your interface with that system is going to be large, the translation between CONTEXTS can easily be a bigger job that the application functionality itself. And there is still some room for good design work, even though you have placed yourself in the BOUNDED CONTEXT of the other system. If there is a discernable domain model behind the other system, you can improve your implementation by making that model more explicit than it was in the old system, just as long as you strictly conform to the old model. If you decide on a conformist design, you must do it wholeheartedly. You restrict yourself to extension only, with no modification of the existing model.

When the functionality of the system under design is going to be more involved that an extension to an existing system, where your interface to other system is small, or where the other system is very badly designed, you'll really want your own BOUNDED CONTEXT, which means building a translation layer, or even an ANTICORRUPTION LAYER.

The System Under Design

The software your project is actually building is the "system under design". You can declare BOUNDED CONTEXTS within this zone and apply CONTINUOUS INTEGRATION with each to keep them unified. But how many should you have? What relationships should they have to each other? The answers are less cut and dried than with the external systems since we have more freedom in the design.

It could be quite simple – a single BOUNDED CONTEXT for the entire system under design. For example, this would be the clear choice for a team of fewer than ten people working on a set of highly interrelated functionality.

As the team grows larger, CONTINUOUS INTEGRATION may become difficult (although I have seen it maintained for somewhat larger teams). You may look for a SHARED KERNEL and break off relatively independent sets of functionality into separate BOUNDED CONTEXTS, each with fewer than ten people. If all of the dependencies between two of these go in one direction, you could set up CUSTOMER/SUPPLIER DEVELOPMENT TEAMS.

You may recognize that the mindset of two groups is so different that their modeling efforts constantly clash. It may be that they actually need quite different things from the model, it may just be a difference in background knowledge, or it may be a result of the management structure the project is embedded in. If the cause of the clash is something you can't change, or don't want to change, you may choose to allow the models to go SEPARATE WAYS. Where integration is needed, a translation layer can be developed and maintained jointly by the two teams as the single point of CONTINUOUS INTEGRATION. This is in contrast with integration with external systems, where the ANTICORRUPTION LAYER typically has to accommodate the other system as is and without much support from the other side.

Generally speaking, there is a correspondence of one team per BOUNDED CONTEXT. One team can maintain multiple BOUNDED CONTEXTS, but it is hard (though not impossible) for multiple teams to work on one together.

Catering to Special Needs With Distinct Models

Different groups within the same business have often developed their own specialized terminologies, which may have diverged from one another. These local jargons may be very precise and tailored to their needs. Changing them (for example, by imposing a standardized, enterprise-wide terminology) requires extensive training and extensive analysis to resolve the differences. Even then, the new terminology may not serve as well as the finely tuned version they already had.

You may decide to cater to these special needs in separate BOUNDED CONTEXTS, allowing the models to go SEPARATE WAYS, except for CONTINUOUS INTEGRATION of translation layers. Different dialects of the UBIQUITOUS LANGUAGE will evolve around these models and the specialized jargon they are based on. If the two dialects have a lot of overlap, a SHARED KERNEL may provide the needed specialization while minimizing the translation cost.

Where integration is not needed, or is relatively limited, this allows continued use of customary terminology and avoids corruption of the models. It also has its costs and risks.

- The loss of shared language will reduce communication
- There is extra overhead in integration
- There will be some duplication of effort, as different models of the same business activities and entities evolve.

But perhaps the biggest risk is that it can become an argument against change and a justification for any quirky parochial model. How much do you need to tailor this individual part of the system to meet specialized needs? Most importantly, *how valuable is the particular jargon of this user group*? You have to weigh the value of more independent action of teams against the risks of translation, keeping an eye out for rationalizing terminology variations that have no value.

Sometimes a deep model emerges that can unify these distinct languages and satisfy both groups. The catch is that deep models emerge later in the lifecycle, after a lot of development and knowledge crunching, if at all. You can't plan on a deep model; you just have to accept the opportunity when it arises, change your strategy, and refactor.

Keep in mind that, where integration requirements are extensive, the cost of translation goes way up. Some coordination of the teams, from the pinpoint modifications of one object that has a complicated translation ranging up to a SHARED KERNEL, can make translation easier while still not requiring full unification.

Deployment

Coordinating the packaging and deployment of complex systems is one of those boring tasks that is almost always a lot harder than it looks. The choice of BOUNDED CONTEXT strategy has an impact on the deployment. For example, when CUSTOMER/SUPPLIER TEAMS deploy new versions, they have to coordinate with each other to release versions that have been tested together. Both code and data migrations have to work in these combinations. In a distributed system, it may help to keep the translation layers between CONTEXTS together within a single process, so that you don't have multiple versions coexisting.

Even deployment of the components of a single BOUNDED CONTEXT can be challenging when data migration takes time or when distributed systems can't be updated instantaneously, resulting in two versions of the code and data coexisting.

Many technical considerations come into play depending on the deployment environment and technology. But the BOUNDED CONTEXT relationships can point you toward the hot spots. The translation interfaces have been marked out.

The feasibility of a deployment plan should feed back into the drawing of the CONTEXT boundaries. When two CONTEXTS are bridged by a translation layer, one CONTEXT can be updated just so a new translation layer provides the same interface to the other CONTEXT. A SHARED KERNEL imposes a much greater burden

of coordination not just in development but also in deployment. SEPARATE WAYS can make life much simpler.

The Tradeoff

To sum up these guidelines, there is a range of strategies for unifying or integrating models. In general terms, you will tradeoff the benefits of seamless integration of functionality against the additional effort of coordination and communication. You trade more independent action against smoother communication. More ambitious unification requires control over the design of the subsystems involved.

Relative Demands of CONTEXT Relationship Patterns

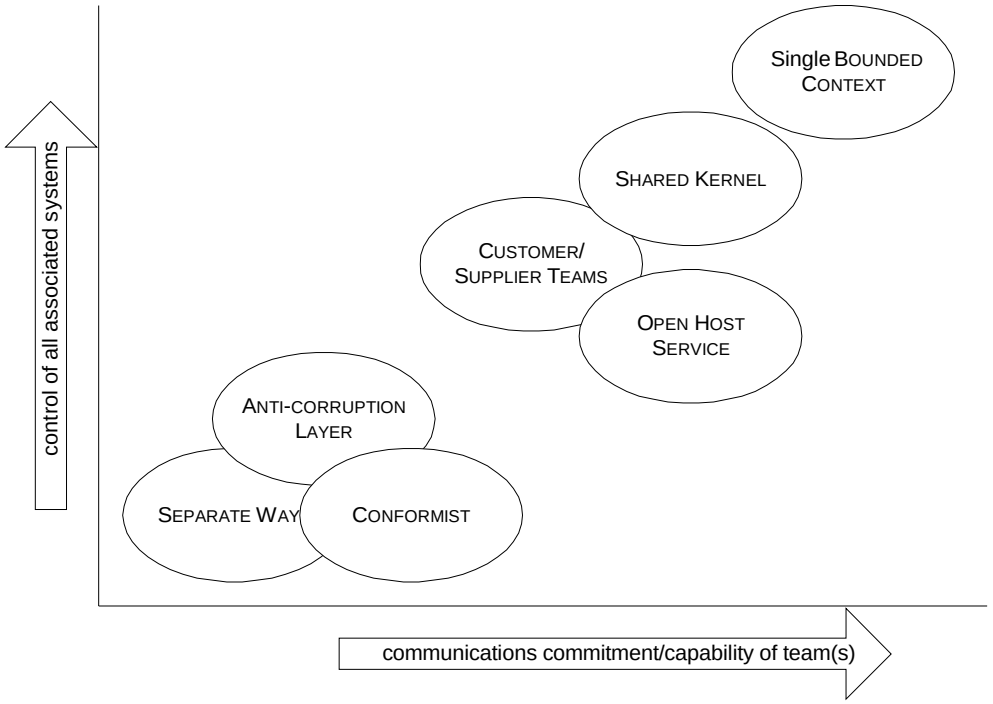


Figure 14. 12

When Your Project Is Already Underway

Most likely, you are not starting a project but are looking to improve a project that is already underway. In this case, the first step is to define BOUNDED CONTEXT *according to the way things are now.* This is crucial. To be effective, the unification contexts must reflect the true practice of the teams, *not* the ideal organization you might decide on following the guidelines above.

Once you have delineated your true current BOUNDED CONTEXTS and described the relationships they currently have, the next step is to tighten up the team's practices *around that current organization.* Improve your CONTINUOUS INTEGRATION within the CONTEXTS. Refactor any stray translation code into your ANTICORRUPTION LAYERS. Name the existing BOUNDED CONTEXTS and make sure they are in the UBIQUITOUS LANGUAGE of the project.

Now you are ready to consider changes to the boundaries and relationships themselves. These changes will naturally be driven by the same principles as described above for a new project, but they will have to be bitten off in small pieces, chosen pragmatically to give the most value for the least effort and disruption.

The next section discusses how to go about actually making changes to your CONTEXT boundaries once you have decided on a change you want to make.

Transformations

Like any other aspect of modeling and design, decisions about BOUNDED CONTEXTS are not irrevocable. Inevitably, there will be many cases in which you will have to change your initial decision about the boundaries and relationships between BOUNDED CONTEXTS. Generally speaking, breaking CONTEXTS up is pretty easy, but merging them or changing the relationships between them can be challenging. I'll describe a few changes that are difficult yet important. These transformations are usually much too big to be taken in a single refactoring or possibly even in a single project iteration. For that reason, I've outlined some game-plans for making these transformations as a series of manageable steps. These are, of course, guidelines that you will have to adapt to your particular circumstances and events.

Merging CONTEXTS (SEPARATE WAYS → SHARED KERNEL)

Translation overhead is too high. Duplication is too obvious. There are many motivations for merging BOUNDED CONTEXTS. This is hard to do. It's not too late, but it takes some patience.

Even if your eventual goal is to merge completely to a single CONTEXT with CONTINUOUS INTEGRATION, start by moving to a SHARED KERNEL.

1. Evaluate the initial situation. Be sure that the two CONTEXTS are indeed internally unified before beginning to unify them with each other.
2. Set up the process. You'll need to decide how the code will be shared and what the module naming conventions will be. There must be at least weekly integration of the SHARED KERNEL code. And it must have a test suite. Set this up before developing any shared code. (The test suite will be empty, so it should be easy to pass!)
3. Choose some small subdomain to start with—something duplicated in both CONTEXTS, but *not* part of the CORE DOMAIN. This first merger is going to establish the process, so it is best to use something simple and relatively generic or non-critical. Examine the integrations and translations that already exist. Choosing something that is being translated has the advantage of starting out with a proven translation, plus you'll be thinning your translation layer.

At this point, you have two models that address the same subdomain. There are basically two approaches to merging. You can choose one model or the other and refactor the other context to be compatible. This decision can be made wholesale, setting the intention of systematically replacing one contexts models and retaining the other's and retaining the coherence. Or it can be done one subdomain at a time, presumably ending up with the best of both (but taking care not to end up with a jumble).

The other option is to find a new model, presumably deeper than either of the originals, capable of assuming the responsibilities of both.

4. Form a group of 2-4 developers, drawn from both teams, to work out a shared model for the chosen subdomain. This includes the hard work of identifying synonyms and mapping any terms that are not already being translated. This joint team outlines a basic set of tests for the model.
5. Developers from either team take on the task of implementing the model (or adapting existing code to be shared), working out details and making it function. If these developers run into problems with the model, they reconvene the team from step 3 and participate in any necessary revisions of the concepts.
6. Developers of each team take on the task of integrating with the new SHARED KERNEL.
7. Remove translations that are no longer needed.

At this point, you will have a very small SHARED KERNEL, with a process in place to maintain it. In subsequent project iterations, repeat steps 3-7 to share more. As the processes firm up and the teams gain confidence, you can take on more complicated subdomains, multiple ones at the same time, or subdomains that are in the CORE DOMAIN.

A note: As you take on more domain-specific parts of the models, you may encounter cases where the two models have conformed to specialized jargon of different user communities. It is wise to defer merging these into the SHARED KERNEL unless *a breakthrough to a deep model* has occurred, providing you with a language capable of superseding both specialized ones. An advantage of SHARED KERNEL is that you can some of the advantages of CONTINUOUS INTEGRATION while retaining some of the advantages of SEPARATE WAYS.

Those are the guidelines to merging into a SHARED KERNEL. Before going ahead, consider one alternative that satisfies some of the needs addressed by this transformation. If one of the two models is definitely preferred, consider shifting toward it without integrating. Instead of sharing common subdomains just systematically transfer full responsibility for those subdomains from one BOUNDED CONTEXT to the other by refactoring the applications to call on the model of the more favored CONTEXT, and making any

enhancements that model needs. Without any ongoing integration overhead, you have eliminated redundancy. Potentially (but not necessarily), the more favored BOUNDED CONTEXT could eventually take over completely, and you'll have created the same effect as a merger. In the transition (which can be quite long or indefinite) this will have the usual advantages and disadvantages of going SEPARATE WAYS, and you have to weigh them against the pros and cons of a SHARED KERNEL.

Merging CONTEXTS (SHARED KERNEL → CONTINUOUS INTEGRATION)

If your SHARED KERNEL is expanding and you may be lured by the advantages of full unification of the two BOUNDED CONTEXTS. This is not just a matter of resolving the model differences. You are going to be changing team structures and ultimately the language people speak.

Start by preparing the people and the teams.

1. Be sure that all the processes needed for continuous integration (shared code ownership, frequent integration, etc) are in place on *each team*, separately. Harmonize integration procedures on the two teams so that everyone is doing things in the same way.
2. Start circulating team members between teams. This will create a pool of people who understand both models, and will begin to connect the people of the two teams.
3. Clarify the distillation (Chapter 15) of each model individually.
4. At this point, confidence should be high enough to begin merging the core domain into the SHARED KERNEL. This can take several iterations and sometimes temporary translation layers are needed between the newly shared parts and the not-yet-shared parts. Once into the core domain, it is best to go pretty fast. It is a high-overhead phase, fraught with errors, and should be shortened as much as possible, taking priority over most new development. But don't take on more than you can chew.

To merge the core models, you have a few choices. You can choose one, stick with it and modify the other to be compatible to it, or you can create a new model of the subdomain and adapt both contexts to use it. Watch out if the two models have been tailored to address distinct user needs. You may need the specialized power of both original models. This calls for developing a deeper model that can supercede both original models. Developing a deeper unifying model is very difficult, but if you are committed to full merger of the two CONTEXTS, you no longer have the option of multiple dialects. There will be a reward in terms of the power and clarity of the resulting model and code. Be careful that it doesn't come at the cost of your ability to address specialized needs of your users.

5. As the SHARED KERNEL grows, increase the integration frequency to daily and finally to CONTINUOUS INTEGRATION.
6. As the SHARED KERNEL approaches the point of encompassing all of the two former BOUNDED CONTEXTS, you will find yourself with either one large team or two smaller teams that have a shared code base that they integrate continuously, and that trade members back and forth frequently.

Phasing Out a Legacy System

All good things must come to an end, even legacy computer software. But it doesn't happen on its own. These old systems can be so woven into the business and other systems that extricating them can take many years. Fortunately, it doesn't have to be done all at once.

The possibilities are too various for me to do more than scratch the surface here. But I'll discuss a common case: An old system that is used daily in the business has been supplemented recently by a handful of more modern systems that communicate with the legacy system through an ANTI-CORRUPTION LAYER.

One of the first steps should be to decide on a testing strategy. Automated unit tests should be written for new functionality in the new systems, but phasing out legacy introduces special testing needs. Some organizations run new and old in parallel for some period of time.

In any given iteration:

1. Identify specific functionality of the legacy that could be added to one of the favored systems within a single iteration.

2. Identify additions that will be required in the ANTI-CORRUPTION LAYER

3. Implement

4. Deploy

Sometimes it will be necessary to spend more than one iteration writing equivalent functionality to a unit that can be phased out of the legacy, but still plan the new functions in small, iteration-sized units, only waiting multiple iterations for deployment.

Deployment is another point at which too much variation exists to cover all the bases. It would be nice for development if these small incremental changes could be rolled out to production, but usually it is necessary to organize bigger releases. The users must be trained to use the new software. A parallel period sometimes must be completed successfully. Many logistical problems will have to worked out.

Once it is finally running in the field,

1. Identify any unnecessary parts of the ANTICORRUPTION LAYER and remove them.

2. Consider excising the now unused modules of the legacy system, though this may not turn out to be practical. Ironically, the better designed the legacy system, the easier it will be to phase it out. But badly designed software is hard to dismantle a little at a time. It may be possible to just ignore the unused parts until a later time when the remainder has been phased out and the whole thing can be switched off.

Repeat this over and over. The legacy system should become less involved in the business, and eventually it will be possible to see the light at the end of the tunnel and finally switch off the old system. Meanwhile, the ANTICORRUPTION LAYER will shrink *and* swell as various combinations increase or decrease the interdependence between the systems. All else being equal, of course, you should migrate first those functions that lead to smaller ANTICORRUPTION LAYERS. But other factors are likely to dominate, and you may have to live with some hairy translations during some transitions.

OPEN HOST SERVICE → PUBLISHED LANGUAGE

You have been interfacing two systems with an ad-hoc protocol, but the maintenance burden is mounting as more systems want access, or if it is becoming very difficult to understand. You need to formalize the relationship between the systems with a PUBLISHED LANGUAGE.

1. If an industry standard language is available, evaluate it and use it if at all possible.

2. If no standard or prepublished language is available, then begin by sharpening up the distillation of the model of the system that will serve as the host.

3. Use the DISTILLED CORE as the basis of an interchange language, using a standard interchange paradigm such as XML, if at all possible.

4. Publish the new language to all involved in the collaboration (at least).

5. If a new system architecture is involved, publish that too.

6. Build translation layers for each collaborating system.

7. Switch over.

At this point, additional collaborators should be able to enter with minimal disruption.

Remember, the PUBLISHED LANGUAGE must be stable, yet you'll still need the freedom to change the host's model as you continue your relentless refactoring. Therefore, do not equate the interchange language and the model of the host. Keeping them close together will reduce translation overhead, and you may choose to make your host a CONFORMIST. But reserve the right to beef up the translation layer and diverge if the cost/benefit tradeoff favors that.

♦ ♦ ♦

Once BOUNDED CONTEXTS have been explicitly defined and respected, then logical consistency should be protected. Related communication problems will at least be exposed so they can be dealt with.

Decisions about the boundaries of BOUNDED CONTEXTS should be based on logical consistency problems between subsystems and on the coordination abilities of the teams, as discussed in this chapter. However, teams often find they have defined large BOUNDED CONTEXTS that seem too complex to fully comprehend as a whole, or to analyze completely. By choice or by chance, this often leads to breaking the CONTEXTS down into more manageable pieces. This fragmentation leads to lost opportunities. There are other means of making large models tractable that should be considered before making this sacrifice. The next two chapters are focused on managing complexity within a big model by applying two broad principles: Distillation and Large-scale Structure…

15.Distillation

$$\mathbf{div}\,\mathbf{D} = \rho$$

$$\mathbf{div}\,\mathbf{B} = 0$$

$$\mathbf{curl}\,\mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\mathbf{curl}\,\mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

--- James Clerk Maxwell, *A Treatise on Electricity and Magnetism*, 1873.

These four equations, along with the definitions of their terms and the body of mathematics they rest on, express the entirety of classical, nineteenth century, electromagnetism.

How do you focus on your central problem and keep from drowning in a sea of side issues? A LAYERED ARCHITECTURE separates business concepts from the technical logic that makes a computer system run, but in a large system even the isolated domain may be unmanageably complex.

Distillation is the process of separating the components of a mixture to extract the essence in a form that makes it more valuable and useful. Applying this principle to a domain model is a running theme through this book. Now this chapter lays out a set of patterns for distilling the domain model as a whole.

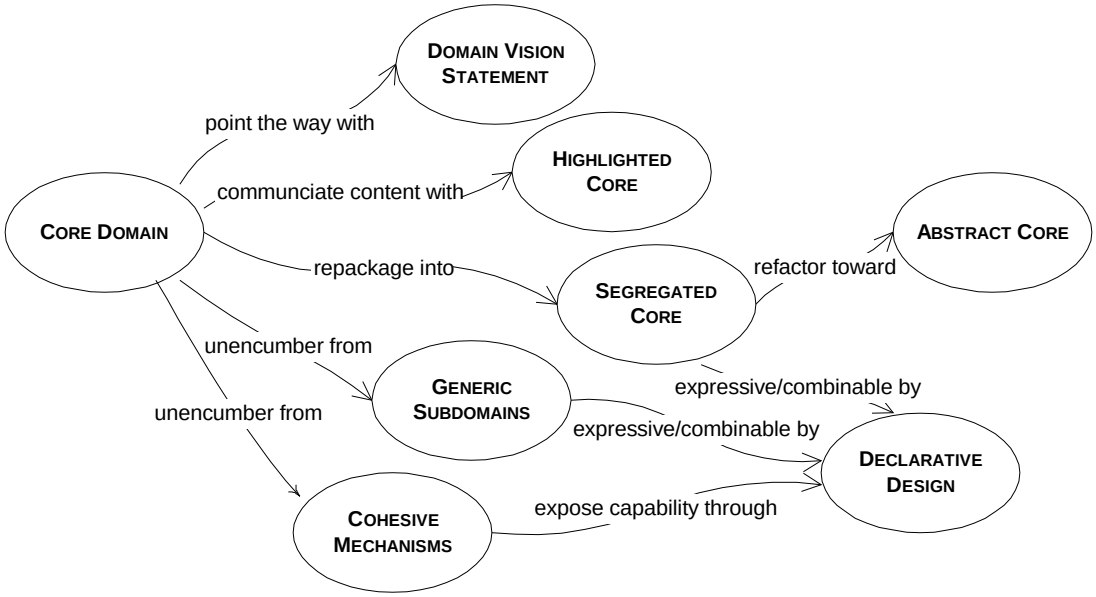
It starts with a renewed focus on the part of the model that matters most, the "CORE DOMAIN", and then examines a variety of techniques for separating the CORE DOMAIN from the other aspects of the model, making it clearer and more useful.

Distilling a domain model:

1. Aids all team members in grasping the overall design of the system and how it fits together.
2. Facilitates communication by identifying a core model of manageable size to enter the UBIQUITOUS LANGUAGE.
3. Guides refactoring.
4. Focuses work on areas of the model with the most value.
5. Guides outsourcing, use of off-the-shelf components, and decisions about assignments.

The patterns of this chapter are a roadmap to the goal of distilling the CORE DOMAIN and effectively communicating it.

Navigation Map for Strategic Distillation



In designing a large system, there are so many contributing components, all complicated and all absolutely necessary to success, that the essence of the business model, the real business asset, can be obscured and neglected.

A system that is hard to understand is hard to change or extend. The effect of a change is hard to foresee. A developer who wanders outside his or her own area of familiarity gets lost. (This is particularly true when bringing new people into a team, but even an established member of the team will struggle unless code is very expressive and organized.) This forces people to specialize. When developers confine their work to specific modules it further reduces knowledge transfer. With compartmentalization of work, smooth integration of the system suffers, and flexibility of assigning work is lost. Duplication crops up because a developer does not realize that some behavior already exists elsewhere, and so the system becomes even more complex.

Those are the consequences of any design that is hard to understand, but there is another, equal risk from losing the big picture of the domain. **The harsh reality is that not all parts of the design are going to be equally refined. Priorities must be set. To make the domain model an asset, the critical core of that model has to be sleek and fully leveraged to create application functionality. But scarce, highly-skilled developers tend to gravitate to technical infrastructure or neatly definable domain problems that can be understood without specialized domain knowledge.** Such parts of the system seem interesting to computer scientists, and are perceived to build transferable professional skills and provide better resume material.

The specialized core, that part of the model that really differentiates the application and makes it a business asset, typically ends up being put together by less skilled developers who work with DBA's to create a data model and then code feature-by-feature without drawing on any conceptual power in the model at all.

The consequence of a poor design or implementation of this part of the domain is that the application never does compelling things for the users, no matter how well the technical infrastructure works, no matter how nice the supporting features are. This problem is insidious without a sharp picture of the overall design and the relative significance of the various parts.

One of the most successful projects I've joined initially suffered from this syndrome. The goal was to develop a very complex syndicated loan system. Most of the strong talent was happily working on database mapping layers and messaging interfaces while the business model was in the hands of developers new to object technology.

The single exception, an experienced object developer working on a domain problem, devised a way of attaching comments to any of the long-lived domain objects. These comments could be organized in various ways so that traders could see the rationale they or others recorded for some past decision. He also built an elegant user interface that gave intuitive access to the flexible features of the comment model.

These features were useful and well designed. They went into production.

Unfortunately, they were peripheral. This talented developer had modeled his interesting, generic way of commenting activities, implemented it cleanly, and put it into users hands. Meanwhile an incompetent developer was turning the mission-critical "loan" module into an incomprehensible tangle that the project very nearly did not recover from.

The planning process must drive resources to the most crucial points in the model and design. To do that, those points must be identified and understood by everyone during planning and development.

Those parts of the model distinctive and central to the purposes of the intended applications make up the CORE DOMAIN. The CORE DOMAIN is where the most value should be added in your system. The patterns in this chapter make it easier to see and use and change.

Therefore,

Boil the model down. Find the CORE DOMAIN and provide a means of easily distinguishing it from the mass of supporting model and code. Bring the most valuable and specialized concepts into sharp relief. Make the CORE small.

Apply top talent to the CORE DOMAIN, and recruit accordingly. Spend the effort in the CORE to find a deep model and develop a supple design – sufficient to fulfill the vision of the system. Justify investment in any other part by how it supports the distilled CORE.

Keep other parts of the model as generic as practical, even using off-the-shelf components when appropriate.

Choosing the CORE

We are looking at those parts of the model particular to representing your business domain and solving your business problems.

The CORE DOMAIN you choose depends on your point of view. For example, many applications need a generic model of money that could represent various currencies and their exchange rates and conversions, but an application to support currency trading might need a more elaborate model of money, and would consider it part of the CORE. Even in such a case, there may be a part of the money model that is very generic. As insight into the domain deepens with experience, the distillation process can continue by separating the generic money concepts and retaining only the specialized aspects of the model in the CORE DOMAIN.

In a shipping application, the CORE could be the model of how cargoes are consolidated for shipping, how liability is transferred when containers change hands, or how a particular container is routed on various transports to reach its destination. In investment banking, the CORE could include the models of syndication of assets among assignees and participants.

One application's CORE DOMAIN is another application's generic supporting component. Still, throughout one project, and usually throughout one company, a consistent CORE can be defined. Like every other part of the design, the identification of the CORE DOMAIN should evolve through iterations. The importance of a particular set of relationships might not have been apparent at first. The objects that seemed obviously central at first may turn out to have supporting roles.

The discussion in the following sections, particularly GENERIC SUBDOMAINS, will give more guidelines for these decisions.

Who Does the Work

A corollary to this diversion of the strongest developers away from the core domain is the loss of the opportunity to build and accumulate domain knowledge.

With few exceptions, the most technically proficient members of projects seldom have much knowledge of the domain, which limits their usefulness and reinforces the tendency to put them onto supporting components, sustaining a vicious circle in which lack of knowledge keeps them away from the work that would build domain knowledge.

It is essential to break this cycle by assembling a team matching up a set of strong developers who have a long-term commitment and an interest in becoming repositories of domain knowledge with one or more domain experts who know the business deeply. Domain design is interesting, technically challenging work when approached seriously, and developers can be found who see it this way.

It is usually not practical to hire short-term outside design expertise to help in the nuts and bolts of creating the CORE DOMAIN because the team needs to accumulate domain knowledge, and a temporary member is a leak in the bucket. On the other hand, an expert in a teaching/mentoring role can be very valuable by helping the team build its domain design skills and facilitating the use of sophisticated principles that team members probably have not mastered.

For similar reasons, it is unlikely that the CORE DOMAIN can be purchased. Efforts have been made to build industry-specific model frameworks, conspicuous examples being the semiconductor industry consortium Sematech's CIM framework for semiconductor manufacturing automation, and IBM's "San Francisco" frameworks for a wide range of businesses. Although this is a very enticing idea, so far the results have not been compelling, except perhaps as PUBLISHED LANGUAGES (p. 264) facilitating data interchange. *Domain-Specific Domain Frameworks* [FJ2000] gives a thorough overview of the state of the art. As the field advances, more workable frameworks may be available.

Even so, there is a more fundamental reason for caution: The greatest value of custom software comes from the total control of the CORE DOMAIN. A well-designed framework may be able to provide high-level abstractions that you can specialize for your use. It may save you from developing the more generic parts and leave you free to concentrate on the CORE. If it constrains you more than that, then there are three likely possibilities.

1. You are losing an essential software asset. Back off restrictive frameworks in your CORE DOMAIN.
2. The area treated by the framework is not as pivotal as you thought. Redraw the boundaries of the CORE DOMAIN to the truly distinctive part of the model.
3. You don't have special needs in your CORE DOMAIN. Consider a lower risk solution, purchasing software to integrate with your applications.

One way or another, creating distinctive software comes back to a stable team accumulating specialized knowledge and crunching it into a rich model. No shortcuts. No magic bullets.

♦ ♦ ♦

Distilling the CORE DOMAIN is not easy, but it leads to some easy decisions. You'll put a lot of effort into making your CORE distinctive, while keeping the rest as generic as possible. If you need to keep some aspect of your design secret as a competitive advantage, it is the CORE DOMAIN. There is no need to waste effort concealing the rest. And whenever a choice has to be made (due to time limitations) between two desirable refactorings, the one that most affects the CORE DOMAIN is chosen first.

The rest of the patterns in this chapter provide techniques for arriving at that easily understood, powerful CORE DOMAIN. Circumstances will dictate the combination of techniques used.

A simple DOMAIN VISION STATEMENT communicates the basic concepts and their value with a minimum investment. The HIGHLIGHTED CORE can improve communication and help guide decision making, and still requires little or no modification to the design.

More aggressive refactoring and repackaging can explicitly separate GENERIC SUBDOMAINS, which can then be dealt with separately. Deep modeling leads to encapsulating COHESIVE MECHANISMS with a versatile, communicative, supple design. Removing these distractions disentangles the CORE. Repackaging a SEGREGATED CORE provides the easiest visibility of the CORE and facilitates future work on the core model. Most ambitious is the ABSTRACT CORE that expresses the most fundamental concepts and relationships in a pure form (and requires extensive reorganizing and refactoring of the model).

Each of these techniques requires a successively greater commitment, but a knife gets sharper as its blade is ground finer. Successive distillation of a domain model produces an asset that gives the project speed, agility and precision of execution.

To start, we can boil off the least distinctive aspects of the model. GENERIC SUBDOMAINS provide a contrast to the CORE DOMAIN that clarifies the meaning of each…

GENERIC SUBDOMAINS

Some parts of the model add complexity without capturing or communicating specialized knowledge. Anything extraneous makes the CORE DOMAIN harder to discern and understand. The model clogs up with details of general principles everyone knows or that belong to specialties that are not your primary focus but play a supporting role. Yet, however generic, these other elements are essential to the functioning of the system and the full expression of the model.

There is a part of your model that you would like to take for granted. It is undeniably of the domain model, but it abstracts concepts that would probably be needed for a great many businesses. For example, a corporate organization chart is needed in some form by businesses as diverse as shipping, banking, or manufacturing. For another example, many applications track receivables, expense ledgers, and other matters that could all be handled using a generic accounting model.

Often a great deal of effort is spent on peripheral issues in the domain. I personally have witnessed two separate projects that have employed their best developers for weeks in redesigning dates and times with time zones. While such components must work, they are not the conceptual core of the system.

Even if some such generic model element is deemed critical, the overall domain model needs to make prominent the most value-adding and special aspects of your system and be structured to give that part as much power as possible. And this is hard to do when it is mixed with all the interrelated factors.

Therefore,

Identify cohesive subdomains that are not the motivation for your project. Factor them into general models of the GENERIC SUBDOMAINS that have no trace of your specialties, and place them in separate packages. Consider off-the-shelf solutions or published models for these subdomains.

Once they have been separated, give their continuing development lower priority than the CORE DOMAIN, and avoid assigning your core developers to the tasks (since they will gain little domain knowledge).

You may have a few extra options when developing these packages.

1. Off-the-shelf solution. Sometimes you can buy an implementation, or use open source code.

    + less code to develop
    + maintenance burden externalized
    + code is probably more mature, used in multiple places, and therefore more bullet-proof and complete than home-grown code
    - you still have to spend the time to evaluate it and understand it before using it
    - quality control being what it is in our industry, you can't count on it being correct and stable
    - may be over-engineered for your purposes, integration could be more work than a minimalist home-grown implementation
    - foreign elements don't usually integrate smoothly. May be a distinct BOUNDED CONTEXT. Even if not, may be difficult to smoothly reference ENTITIES from your other packages.
    - may introduce platform dependencies, compiler version dependencies, etc.

    Off-the-shelf subdomain solutions usually are not worth the trouble, but are worth investigating. I've seen success stories in applications with very elaborate workflow

requirements that used commercially available external workflow systems with API hooks. I've also seen success with an error-logging package that was deeply integrated into the application. Sometimes generic subdomain packages are in the form of frameworks, which implement a very abstract model that can be integrated with and specialized for your application. The more generic the subcomponent, and the more distilled its own model, the better the chance that it will be useful.

2. Published design or published model

    + more mature than a homegrown model, and reflects many people's insights

    + Instant, high-quality documentation

    - may not quite fit your needs or may be over engineered for your needs

    Tom Lehrer (a comedic song-writer from the 1950s and '60s) said the secret to success in mathematics was, "Plagiarize! Plagiarize. Let no one's work evade your eyes. […] Only be sure always to call it please, "*research*"." Good advice in domain modeling, and especially when attacking a GENERIC SUBDOMAIN.

    This works best when there is a widely distributed model such as the ones in *Analysis Patterns* [Fowler1996].

    When the field already has a highly formalized and rigorous model, use it. Accounting and physics are two examples that come to mind. Not only are these very robust and streamlined, but they are widely understood by people everywhere, reducing your present and future training burden.

    Don't feel compelled to implement all aspects of a published model, if you can identify a simplified subset that is self-consistent and satisfies your needs. But in cases where there is a well traveled and documented, or, better yet, formalized model available, it makes no sense to reinvent the wheel.

3. Outsource implementation

    + keeps core team free to work on core domain, where most knowledge is needed and accumulated

    + allows more development to be done without permanently growing team, but without dissipating knowledge of the CORE DOMAIN

    + forces an interface-oriented design, and helps keep the subdomain generic, since the specification is being passed outside

    - still requires time from core team, since interface, coding standards, and any other important aspects need to be communicated

    - significant overhead of transferring ownership back inside, since code has to be understood (still, overhead is less than for specialized parts, since a generic model is presumably requires no special background to understand)

    +/- code quality can vary. This could be good or bad, depending on the relative caliber of the two teams

    Automated tests can play an important role in outsourcing. The implementers should be required to provide unit tests for the code they deliver. A really powerful approach, that helps ensure a degree of quality, clarifies the spec, and smoothes reintegration, is to specify or even write automated acceptance tests for the outsourced components. Note that "outsourced implementation" can be an excellent combination with "published design or model".

4. In-house implementation

+ easy integration

+ you get just what you want and no extra

+ Temporary contractors can be assigned

- ongoing maintenance burden and training burden

- it is easy to underestimate the time and cost of developing such packages

Of course, this too combines well with "published design or model"

GENERIC SUBDOMAINS are the place to try to apply outside design expertise, since they do not require deep understanding of your specialized CORE DOMAIN, and do not present a major opportunity to learn that domain. Confidentiality is less of a concern, since little proprietary information or business practice will be involved in these modules. A GENERIC SUBDOMAIN lessens the training burden for those not committed to deep knowledge of the domain.

Over time, I believe our ideas of what constitutes the CORE model will narrow, and more and more generic models will be available as implemented frameworks, or at least as published analysis models as in [Fowler96]. For now, we still have to develop most of these ourselves, but there is great value in partitioning them from the CORE MODEL.

A Tale of Two Time Zones

Twice I've watched as the best developers on a project spent weeks of their time solving the problem of storing and converting times with time zones. While I'm always suspicious of such activities, sometimes it is necessary, and these two projects provide almost perfect contrast.

The first was an effort to design cargo shipping scheduling software. To schedule international transports, it is critical to have accurate time calculations, and since all such schedules are tracked in local time, it is impossible to coordinate transports without conversions.

Having clearly established their need for this functionality, they proceeded with development of the CORE DOMAIN and some early iterations of the application using the available time classes and some dummy data. As the application began to mature, it was clear that the existing time classes were not adequate, and that the problem was very intricate because of the variations between the many countries and the complexity of the International Date Line. With their requirements by now even clearer, they searched for an off-the-shelf solution, but found none. They had no option but to build it themselves.

The task would require research, and precision engineering, so they assigned one of their best programmers. But the task did not require any special knowledge of shipping and would not cultivate that knowledge, so they chose a programmer who was on the project on a temporary contract.

This programmer did not start from scratch. He researched several existing implementations of time zones, most of which did not meet requirements, and decided to adapt the public domain solution from BSD Unix, which had an elaborate database and an implementation in C. He reverse-engineered the logic and wrote an import routine for the database.

The problem turned out to be even harder than expected (involving, for example, the import of databases of special cases), but the code got written and integrated with the CORE and the product was delivered.

Things went very differently on the other project. An insurance company was developing a new claims processing system, and planned to capture times of various events (when did the car crash, etc.) This would be recorded in local time, ergo time zone functionality was needed.

When I arrived, they had assigned a junior, but very smart, developer to the task, although the exact requirements of the app were still in play and not even an initial iteration had been attempted. He had dutifully set out to build a time zone model *a priori*.

Not knowing what would be needed, it was assumed that it should be flexible enough to handle anything. The programmer assigned to the task needed help with such a difficult task, so a senior developer was assigned to the task also. Complex code was written, but, since no specific application was using it, it was never clear that it worked correctly.

The project ran aground for various reasons, and the time zone code was never used, but if it had been, my assessment was that initially storing local times tagged with the time zone might have been sufficient, even with no conversion, because this was primarily reference data and not the basis of computations. Even if conversion had turned out to be necessary, all the data was going to be gathered from North America, where time zone conversions are relatively simple.

The main cost of this attention to the time zones was the neglect of the CORE DOMAIN model. If the same energy had been placed there, they might have produced a functioning prototype of their own app and a first cut at a working domain model. Furthermore, the developers involved, who were committed long-term to the project, should have been steeped in the insurance domain, building up critical knowledge within the team.

One thing both projects did right was to cleanly segregate the time zone model from the CORE DOMAIN. A shipping-specific or insurance-specific model of time zones would have coupled the model to this generic supporting model, making the CORE harder to understand (because it would contain irrelevant detail about time zones) and the time zone harder to maintain (because the maintainer would have to understand the CORE and its interrelationship with time zone).

Shipping Transport Scheduler	Insurance Claims Tracker
+ generic model decoupled from core	+ generic model decoupled from core
+ core model mature, so resources could be diverted without stunting it	- core model undeveloped, so attention to other issues continued this neglect
+ knew exactly what they needed	- unknown requirements led to attempt at full generality, where simpler North America-specific conversion might have sufficed
+ critical support functionality for international scheduling	
+ programmer on short term contract used for generic task	- long-term programmers were assigned who could have been repositories of domain knowledge
- diverted top programmer from core	

We technical people tend to enjoy definable problems like time zone conversion, and can easily justify spending our time on them. But a disciplined look at priorities usually points to the CORE DOMAIN.

Generic doesn't mean reusable

Note that while I have emphasized the generic quality of these subdomains, I have not emphasized reusability of code. Off-the-shelf solutions may or may not make sense for a particular situation, but assuming you are implementing the code yourself, in-house or out-sourced, you should specifically not concern yourself with the reusability of that code. This would go against the basic motivation of distillation – that you should be applying as much of your effort to the CORE DOMAIN as possible and investing only as necessary in supporting GENERIC SUBDOMAINS.

Reuse does happen, but not always code reuse. Model reuse is often a better level of reuse, as when you use a published design or model. And if you have to create your own model, it may well be valuable in a later related project. But while the concept of such a model may be applicable to many situations, you do not have to develop the model in its full generality. You can model and implement only the part you need for your business.

While you should seldom design for reusability, you must be strict about keeping within the generic concept. Introducing industry-specific model elements will have two costs. First, you may need to expand the model later. Although you need only a small part of it now, your needs will grow. By introducing anything to the design that is not part of the concept, you make it much more difficult to expand the system cleanly without completely rebuilding the older part and redesigning the other modules that use it.

The second, and more important, reason is that those industry-specific concepts belong either in the CORE DOMAIN or in their own, more specialized, subdomains, and those specialized models are even more valuable than the generic ones are.

Project Risk Management

Agile processes typically call for managing risk by tackling the riskiest tasks early. XP specifically calls for getting an end-to-end system up and running immediately. This initial system often proves a technical architecture, and it is tempting to build a peripheral system that handles some supporting GENERIC SUBDOMAIN, since these are usually easier to analyze. But be careful; this can defeat the purpose of risk management.

Projects face risk from both sides, with some projects having greater technical risks and others greater domain modeling risks. The end-to-end system mitigates risk only to the extent that it is an embryonic version of the challenging parts of the actual system. It is easy to underestimate the domain modeling risk. It can take the form of unforeseen complexity, of inadequate access to business experts, or gaps in key skills of the developers.

Therefore, except when the team has proven skills and the domain is very familiar, the first-cut system should be based on some part of the CORE DOMAIN, however simple.

The same principle applies to any process that tries to push high-risk tasks forward: the CORE DOMAIN is high risk because it is often unexpectedly difficult and because without it, the project cannot succeed.

♦ ♦ ♦

Most of the distillation patterns show how to change the model and code to distill the CORE DOMAIN. The next two patterns, DOMAIN VISION STATEMENT and HIGHLIGHTED CORE, show how the use of supplemental documents can, with a very minor investment, improve communication and awareness of the CORE and focus development effort…

DOMAIN VISION STATEMENT

At the beginning of a project, the model usually doesn't even exist, yet the need to focus its development is already there. In later stages of development there is a need for an explanation of the value of the system that does not require an in-depth study of the model. Also, the critical aspects of the domain model may span multiple BOUNDED CONTEXTS, but, by definition, these distinct models can't be structured to show their common focus.

Many projects write "vision statements" for management. The best of these documents lay out the specific value the application will bring to the organization. Some of these describe the creation of the domain model as a strategic asset. Usually the vision statement document is abandoned after the project gets funding, and is never used in the actual development process or even read by the technical staff.

These documents, or closely related ones that emphasize the nature of the domain model, can be used directly by the management and technical staff during all phases of development to guide resource allocation, to guide modeling choices, and to educate team members. If the domain model serves many masters, you can use this document to show how their interests are balanced.

Write a short (~1 page) description of the CORE DOMAIN and the value it will bring, the "value proposition". Ignore those aspects that do not distinguish this domain model from others. Show how the domain model serves and balances diverse interests. Keep it narrow. Write this statement early and revise it as you gain new insight.

This is part of a DOMAIN VISION STATEMENT	**This, though important, is <u>not</u> part of a DOMAIN VISION STATEMENT**
Airline booking system. The model can represent passenger priorities and airline booking strategies and balance these based on flexible policies. The model of a passenger should reflect the "relationship" the airline is striving to develop with repeat customers. Therefore, it should represent the history of the passenger in useful condensed form, participation in special programs, affiliation with strategic corporate clients, etc. Different roles of different users (e.g. passenger, agent, manager) are represented to enrich the model of relationships and to feed necessary information to the security framework. Model should support efficient route/seat search and integration with other established flight booking systems.	The UI should be streamlined for expert users but accessible for first time users. Access will be offered over the web, by data transfer to other systems, and maybe through other UI's, so interface will be designed around XML I.O. with transformation layers to serve web pages or translate to other systems. A colorful animated version of the logo needs to be cached on the client machine so that it can come up quickly on future visits. When customer submits a reservation, make visual confirmation within 5 seconds. A security framework will authenticate a user's identity and then limit access to specific features based on privileges assigned to defined user roles.

This is part of a DOMAIN VISION STATEMENT	**This, though important, is <u>not</u> part of a DOMAIN VISION STATEMENT**
Semiconductor factory automation The domain model will represent the status of materials and equipment within a wafer fab in such a way that necessary audit trails can be provided and automated product routing can be supported. The model will not include the human resources required in the process, but must allow selective process automation through recipe download. The representation of the state of the factory should be comprehensible to human managers, to give them deeper insight and support better decision making.	The software should be web enabled through a servlet, but structured to allow alternative interfaces. Industry standard technologies should be used whenever possible to avoid in-house development and maintenance costs and to maximize access to outside expertise. Open source solutions are preferred (e.g. Apache web server.) The web server will run on a dedicated server. The application will run on a single dedicated server.

The DOMAIN VISION STATEMENT can be used as a guide post that keeps the development team headed in a common direction in the ongoing process of distilling the model and code itself. It can be shared with non-technical team members, management, and even customers (except where it contains proprietary information, of course).

♦ ♦ ♦

A DOMAIN VISION STATEMENT sets a guidepost for to give the team a shared direction. Some bridge between the high-level statement and the full detail of the code or model will usually be needed…

A DOMAIN VISION STATEMENT identifies the core model in broad terms, but leaves the identification of the specific core elements up to the vagaries of individual interpretation. Unless there is exceptionally high communication on the team, the VISION STATEMENT alone will have little impact. In order to translate the high-level view into a concrete view requires each developer to continually mentally filter the large model to identify the CORE DOMAIN. This constant sifting takes up a lot of concentration, and requires a complete knowledge of the model. Even then, there is no guarantee that two people will choose the same elements, or that the same person will be consistent from day to day. The CORE DOMAIN must be made easier to see.

Structural changes in the organization of the model, such as partitioning GENERIC SUBDOMAINS and a few more to come later in this chapter, can allow the MODULES to tell the story. But it is usually too ambitious to shoot straight for that as the only means of communicating the CORE DOMAIN. To reiterate, the diagram is not the model, nor is the code. The model is a set of concepts that are held in team members heads and expressed through diagrams, code, and other documents. The problem of marking off a privileged part of that model, along with the implementation that embodies it, is a reflection on the model, not necessarily part of the model itself.

A lighter solution can supplement these more aggressive techniques. You may be starting out with existing code that does not differentiate the core well. You may be able to refactor toward these other patterns, but you really need to see the CORE, and share that view, to do that effectively. You may have design constraints that prevent you from making physical separations. Even at an advanced stage, a few carefully selected diagrams or documents may provide mental anchor points and entry points for the team. This problem arises equally for projects that use elaborate UML models and those (like XP projects) that keep few external documents and use the code as the primary repository of the model. An Extreme Programming team might be more minimalist, keep these supplements more casual and more transient (e.g. a hand drawn diagram on the wall for all to see), but these techniques can fold nicely into the process.

Even though we may know broadly what constitutes an element of the CORE DOMAIN, we won't come up with consistent choices from developer to developer or even from one day to the next. As we unravel the model for ourselves, we have no effective way of sharing our discoveries, or even of recording them for ourselves to aide memory. The labor of constantly sifting the model in our heads to identify the key parts is simply too hard. Significant structural changes to the code are the ideal way of identifying the core domain, but not always practical in the short-term. In fact, such major code changes are difficult to undertake without the very view we are lacking.

Any technique that makes it easy for everyone to know the core domain will do to solve this problem. Two specific techniques can represent this class of solutions.

Distillation Document

Often I create a separate document to describe and explain the CORE DOMAIN. It can be as simple as a list of the most essential conceptual objects. It can be a set of diagrams focused on those objects, showing their most critical relationships. It can walk through the fundamental interactions at an abstract level or by example. It can use UML class or sequence diagrams, nonstandard diagrams particular to the domain, carefully worded textual explanations, or combinations of these. *A distillation document is not a complete design document*. It is a minimalist entry point that points out the CORE and suggests reasons for closer scrutiny particular pieces. The reader is then guided to the appropriate point in the code, and is provided with a broad view of how the pieces fit, but still goes to the code to see details.

Write a very short (3-7 sparse pages) document that describes the CORE DOMAIN and the primary interactions among CORE elements.

All the usual risks of separate documents apply:

1. It may not be maintained
2. It may not be read

3. By multiplying the information sources, it may defeat the whole purpose of cutting through complexity

The best way to limit these risks is to be absolutely minimalist about this document. When I write these, they are usually 3-7 sparse pages, including diagrams. By staying away from mundane detail and focusing on the central abstractions and their interactions, the document will age more slowly, since this level of the model is usually more stable.

Write the document to be understood by the non-technical members of the team. Use it as a shared view that delineates what everyone needs to know, and a doorway for all team members to start their exploration of the model and code.

Flagged Core

On my first day on a project at a major insurance company, I was given a copy of the "domain model", a two hundred page document, purchased at great expense from an industry consortium. I spent a few days wading through a jumble of class diagrams covering everything from the detailed composition of insurance policies to extremely abstract models of relationships between people. The quality of the factoring of these models ranged from high-school project to rather good (a few even described business rules, at least in the accompanying text). But where to start? Two hundred pages.

The project culture heavily favored abstract framework building, and my predecessors had focused on a very abstract model of the relationship of people with each other, with things, activities or agreements. It was actually a nice analysis of these relationships, and their experiments with the model had the quality of an academic research project. But it wasn't getting us anywhere near an insurance application.

My first instinct was to start slashing, finding a small CORE DOMAIN to fall back on, then refactoring that and reintroducing other complexities as we went. But the management was alarmed by this attitude. The document was invested with great authority. Its production had involved experts from across the industry, and in any event they had paid the consortium far more than they were paying me, so they were unlikely to weigh my recommendations for radical change too heavily. But I knew we had to get a shared picture of our CORE DOMAIN and get everyone's efforts focused on that.

Instead of refactoring, I went through the document and, with the help of a business analyst who knew a great deal about the insurance business in general and the requirements of the application we were to build in particular, I identified the handful of sections that presented the essential, differentiating, concepts we needed to work with. I provided a navigation of the model that clearly showed the CORE and its relationship to supporting features.

A new prototyping effort started from this perspective, and quickly yielded a simplified application that demonstrated some of the required functionality.

Two pounds of recyclable paper was turned into a business asset by a few page tabs and some yellow highlighter.

This technique is not specific to object diagrams on paper. A team that uses UML diagrams extensively could use a "stereotype" to identify core elements. A team that uses the code as the sole repository of the model might use comments, maybe structured as Java Doc, or might use some tool in their development environment. The particular technique doesn't matter, as long as a developer can effortlessly see what is in and what is not in the CORE DOMAIN.

Flag the elements of the CORE DOMAIN within the primary repository of the model, without particularly trying to elucidate its role. Make it effortless for a developer to know what is in or out of the CORE.

The CORE DOMAIN is now clearly visible to those working with the model, with a fairly small effort and low maintenance, at least to the extent that the model is factored fine enough to distinguish the contributions of parts.

Distillation Document as Process Tool

Theoretically, on an XP project, any pair (two programmers working together) can change any code in the system. In practice, some changes have major implications, and call for more consultation and coordination. When working in the INFRASTRUCTURE LAYER, it may be clear that changing the procedure for storing objects will have a wide-ranging impact, but it may not be so obvious in the DOMAIN LAYER, as typically organized.

With the concept of the CORE DOMAIN, this can be clear. Changes to the CORE DOMAIN should have a big effect. Changes to widely used generic elements may require the update of a lot of code, but still shouldn't create the conceptual shift that CORE changes do.

Use the distillation document as a guide. When a programming pair changes code that does not require any update to the distillation document, either because they are working outside the CORE or because they are changing only details, they have the full autonomy that XP suggests.

When they realize that the distillation document requires change to stay in synch with their code, either because they are fundamentally changing the CORE DOMAIN elements or relationships, or because they are changing the boundaries of the CORE, including or excluding something different, then consultation is called for, and dissemination of the new model through a new version of the distillation document.

When a model or code change affects the Distillation Document, it requires consultation with other team members. When the change is made, it requires immediate notification of all team members, and the dissemination of a new version of the Distillation Document. Other changes can be integrated without consultation or notification, and will be encountered by other members in the course of their work.

♦ ♦ ♦

Although the VISION STATEMENT and HIGHLIGHTED CORE inform and guide, they do not actually modify the model or the code itself. Partitioning GENERIC SUBDOMAINS physically removes some distracting elements. Next we'll look at other ways to structurally change the model and the design itself to make the CORE DOMAIN more visible and manageable.

COHESIVE MECHANISMS

Encapsulating mechanisms is a standard principle of object-oriented design. By hiding complex algorithms in methods with intention revealing names separates the "what" from the "how". This technique makes a design simpler to understand and use. Yet it has natural limits.

The computations sometimes reach a level of complexity that begins to bloat the design. The conceptual "what" is swamped by the mechanistic "how". A large number of methods that provide algorithms for resolving the problem obscure the methods that express the problem.

This proliferation of procedures is usually a symptom of a problem in the model. Refactoring toward deeper insight can yield a model and design whose elements are better suited to solving. The first solution to seek is a model that makes the computation mechanism simple. But now and then the insight emerges that some part of the mechanism is itself conceptually coherent. This conceptual computation will probably not include all of the messy computations you need. But extracting it should make the remaining mechanism easier to understand. We are not talking about some kind of catch-all "calculator".

Partition a conceptually cohesive mechanism into a separate lightweight framework. Particularly watch for formalisms or well-documented categories of algorithms. Expose the capabilities of the framework with an INTENTION-REVEALING INTERFACE. Now the other elements of the domain focused on expressing the problem ("what"), delegating the intricacies of the solution ("how") to the framework.

These separated mechanisms are then placed in their supporting roles, leaving a smaller, more expressive CORE DOMAIN that uses the mechanism through the interface in a more declarative style.

Recognizing a formalism moves some of the complexity of the design into a studied set of concepts. We can implement a solution with confidence and little trial-and-error. We can count on other developers knowing about it or at least being able to look it up. This is similar to the benefits of a published GENERIC SUBDOMAIN model, but may happen more often because this level of computer science has been more studied. Still, more often than not you will have to create something new. Make it narrowly focused on the computation and avoid mixing in the expressive domain model. There is a separation of responsibilities. The expressive model, part of the CORE DOMAIN or a GENERIC SUBDOMAIN, formulates a fact, rule, or problem. The COHESIVE MECHANISM resolving the rule or completes the computation as specified by the expressive model.

A Mechanism in an Organization Chart

I went through this process on a project that needed a fairly elaborate model of an organization chart. This model represented that one person worked for another, and in which branches of the organization, and provided an interface by which relevant questions may be asked and answered. But with a INTENTION-REVEALING INTERFACE the means by which the answers are obtained were not a primary concern. Since most of these questions were along the lines of "Who, in this chain of command, has authority to approve this?" or "Who, in this department, is capable of handling an issue like this?" the team realized that most of the complexity involved traversing specific branches of the organizational tree searching for specific people or relationships. This is exactly the kind of problem solved by the well developed formalism of a "graph", a set of nodes connected by arcs (called "edges") and the rules and algorithms needed to traverse the graph.

We implemented a graph traversal framework as a GENERIC SUBDOMAIN. This framework used standard graph terminology and algorithms familiar to most computer scientists and abundantly documented in textbooks. By no means did we implement a fully general graph. It was a small subset of that conceptual framework that covered the features needed for our organization model.

> Now the organization model could simply state, using standard graph terminology, that each person is a node, and that each relationship between people is an edge (arc) connecting those nodes. After that, presumably, mechanisms within the graph framework could find the relationship between any two people.
>
> If this mechanism had been incorporated into the domain model, it would have cost us in two ways. The model would have been coupled to a particular method of solving the problem, limiting future options. More importantly, the model of organization would have been greatly complicated and muddied. Keeping mechanism and model separate allowed a declarative style of describing organizations that was much clearer. And the intricate code for graph manipulation was isolated in a purely mechanistic framework, based on proven algorithms, that could be maintained and unit tested in isolation.

Another example of a COHESIVE MECHANISM would be a framework for constructing SPECIFICATION objects that can support the basic comparison and combination operations expected of them. The SPECIFICATION pattern allows the client to use a declarative style. By providing a framework, the CORE DOMAIN and GENERIC SUBDOMAINS can declare their SPECIFICATIONS in the clear easily understood conceptual language described of that pattern. The intricate operations involved in carrying out the comparisons and combinations can be left to the framework.

GENERIC SUBDOMAIN V. COHESIVE MECHANISM

GENERIC SUBDOMAINS and COHESIVE MECHANISMS are motivated by the same desire to unburden the core domain. The difference is the nature of the responsibility taken on. A GENERIC SUBDOMAIN is based on an expressive model that represents some aspect of how the team views the business domain. In this it is no different than the CORE DOMAIN, just less central, less important, less specialized. The COHESIVE MECHANISM does not represent the business; it solves some sticky computational problem posed by the expressive modules.

An expressive model proposes; a COHESIVE MECHANISM disposes.

In practice, unless you recognize a formalized, published computation, this distinction is usually not pure, at least at not at first. In successive refactoring it should either be distilled into a purer mechanism, or be transformed into a GENERIC SUBDOMAIN with some previously unrecognized model concepts that would make the mechanism simple.

When a mechanism is part of the CORE DOMAIN

You almost always want to remove mechanisms from the CORE DOMAIN. The one exception is when a mechanism is itself proprietary and a key part of the value of the software. This is sometimes the case with highly specialized algorithms. For example if one of the distinguishing features of a shipping logistics application was a particularly effective algorithm for working out schedules, that mechanism could be considered part of the conceptual core. I once worked on a project at an investment bank in which highly proprietary algorithms for rating risk were definitely in the CORE DOMAIN. (In fact, they were held so closely that even most of the core developers were not allowed to see them). Of course, these algorithms are probably a particular implementation of a set of rules that really predict risk. Deeper analysis might lead to a deeper model that would allow those rules to be explicit, with an encapsulated solving mechanism.

But that would be another incremental improvement in the design, for another day. The decision as to whether to do go that next step would be based on cost benefit: How difficult would it be to work out that new design? How difficult is the current design to understand and modify? How much easier would it be with a more advanced design, for the type of people who would be expected to do the work?

Full Circle

> **Organization Chart Reabsorbs Its Mechanism**
>
> Actually, a year after we completed the organization model, other developers redesigned it to eliminate the separation of the graph framework. Performance problems in the deployment environment were emerging, and they felt the complication of separating the mechanism into a separate package was not warranted. Instead, they added node behavior to the parent class of the organizational ENTITIES. Still, they retained the declarative public interface of the organization model. They even kept the mechanism encapsulated, within the organizational ENTITIES.

These full circles are common, but they do not return to their starting point. The end result is usually a deeper model that more clearly differentiates facts, goals and mechanisms. Pragmatic refactoring retains the important virtues of the intermediate stages while shedding the unneeded complications.

Distilling to a Declarative Style

Declarative design and "declarative style" was the topic of Chapter 10, but that design style deserves special mention in this chapter on strategic distillation. The key to distillation is being able to see what you are doing – cutting to the essence without being distracted by irrelevant detail. Important parts of the CORE DOMAIN may be able to follow a declarative style, when the supporting design provides an economical language for expressing the concepts and rules of the CORE while encapsulating the means of computing or enforcing them.

COHESIVE MECHANISMS are by far most effective when they provide access through an INTENTION REVEALING INTERFACE, conceptually coherent ASSERTIONS and SIDE-EFFECT-FREE FUNCTIONS. GENERIC SUBDOMAINS with supple designs can allow the CORE DOMAIN to make meaningful statements rather than calling obscure functions. But an exceptional payoff comes when part of the CORE DOMAIN itself breaks through to a deep model and starts to function as a language that can express the most important situations the application deals with flexibly and concisely.

A deep model often comes with a corresponding supple design. When a supple design reaches maturity, it provides an easily understood set of elements that can be combined unambiguously to accomplish complex tasks or express complex information, just as words are combined into sentences. At that point, client code takes on a declarative style.

♦ ♦ ♦

Factoring out GENERIC SUBDOMAINS reduces clutter, and COHESIVE MECHANISMS serve up encapsulate complex operations. This leaves behind a more focused model, with fewer distractions that add no particular value to the way you do your business. But you are unlikely ever to find good homes for *everything* in the domain model that is not CORE. The SEGREGATED CORE takes a more direct approach to structurally marking off the CORE DOMAIN…

Elements in the model may partially serve the core and partially play supporting roles. Core elements may be tightly coupled to generic ones. The conceptual cohesion of the core may not be strong or visible. All this clutter and entanglement chokes the core. Designers can't clearly see the most important relationships, leading to a weak design.

By factoring out GENERIC SUBDOMAINS, you clear away some of the obscuring detail from the domain, making the core more visible. But it is hard work identifying and clarifying all these subdomains, and some of them don't seem worth the trouble. Meanwhile, the all important core domain is left entangled with the residue.

Refactor the model to separate the core concepts from supporting players (including ill-defined ones) and strengthen the cohesion of the core while reducing its coupling to other code. Factor all generic or supporting elements into other objects and place them into other packages, even if this means refactoring the model in ways that separate highly coupled elements.

This is basically taking the same principles we applied to GENERIC SUBDOMAINS but from the other direction. The cohesive subdomains that are central to our application can be identified and partitioned into coherent packages of their own. What is done with the undifferentiated mass left behind is important – but not as important. It can be left more or less where it was, or placed into packages based on prominent classes. Eventually, more and more of the residue can be factored into GENERIC SUBDOMAINS, but in the short term any easy solution will do, just so the focus on the SEGREGATED CORE is retained.

The steps to refactor to SEGREGATED CORE typically are something like this:

1. Identify a core subdomain (possibly drawing from the Distillation Document).
2. Move related classes to a new package, named for the concept that relates them.
3. Refactor code to sever data and functionality that are not directly expressions of the concept. Put the removed aspects into (possibly new) classes in other packages. Try to place them with conceptually related tasks, but don't waste too much time being perfect. Keep focused on scrubbing the core subdomain and making the references from it to other packages explicit and self-explanatory.
4. Refactor the remaining core to make its relationships and interactions simpler and more communicative, and to minimize and clarify its relationships with other packages. (This becomes an ongoing refactoring objective.)

Costs of Creating a SEGREGATED CORE

Segregating the core will sometimes make relationships with tightly coupled non-core classes more obscure or even more complicated, but the benefit of clarifying the core domain and making it much easier to work on outweighs this.

The SEGREGATED CORE will let you enhance the cohesion of that CORE DOMAIN. Since there are many meaningful ways of breaking down a model, sometimes in the creation of a SEGREGATED CORE a nicely cohesive package may be broken, sacrificing that cohesion for the sake of bringing out the cohesiveness of the CORE DOMAIN. This is a net gain since the greatest value-added of enterprise software comes from the enterprise specific aspects of the model.

And, of course, segregating the core is a lot of work. Although some of that effort also contributes to continuous refactoring on the finer scale, it must be acknowledged that a decision to go to a segregated core will potentially absorb developers in changes all over the system.

The time to use SEGREGATED CORE is when you have a large BOUNDED CONTEXT that is critical to the system, but where the essential part of the model is being obscured by a great deal of supporting capability.

Evolving Team Decision

As with many strategic design decisions, an entire team must move to a SEGREGATED CORE together. This means a team decision process and a team disciplined and coordinated enough to carry out the decision. The challenge is to constrain everyone to use the same definition of the CORE while not freezing that decision. Because the CORE DOMAIN evolves just like every other aspect of a design. Experience working with a segregated core will lead to new insights into what is essential and what is supporting. Those insights should feed back in the form of a refined definition of the core and refactoring of the code.

This means that new insights must be shared with the team on an ongoing basis, but an individual (or programming pair) cannot act on those insights unilaterally. Whatever the process is for joint decisions, whether consensus or team leader, it must be agile enough to make repeated course corrections. Communication must be effective enough to keep everyone together in one view of the CORE.

Segregating the Core of a Cargo Shipping Model

We start with the following model as the basis of software for cargo shipping coordination.

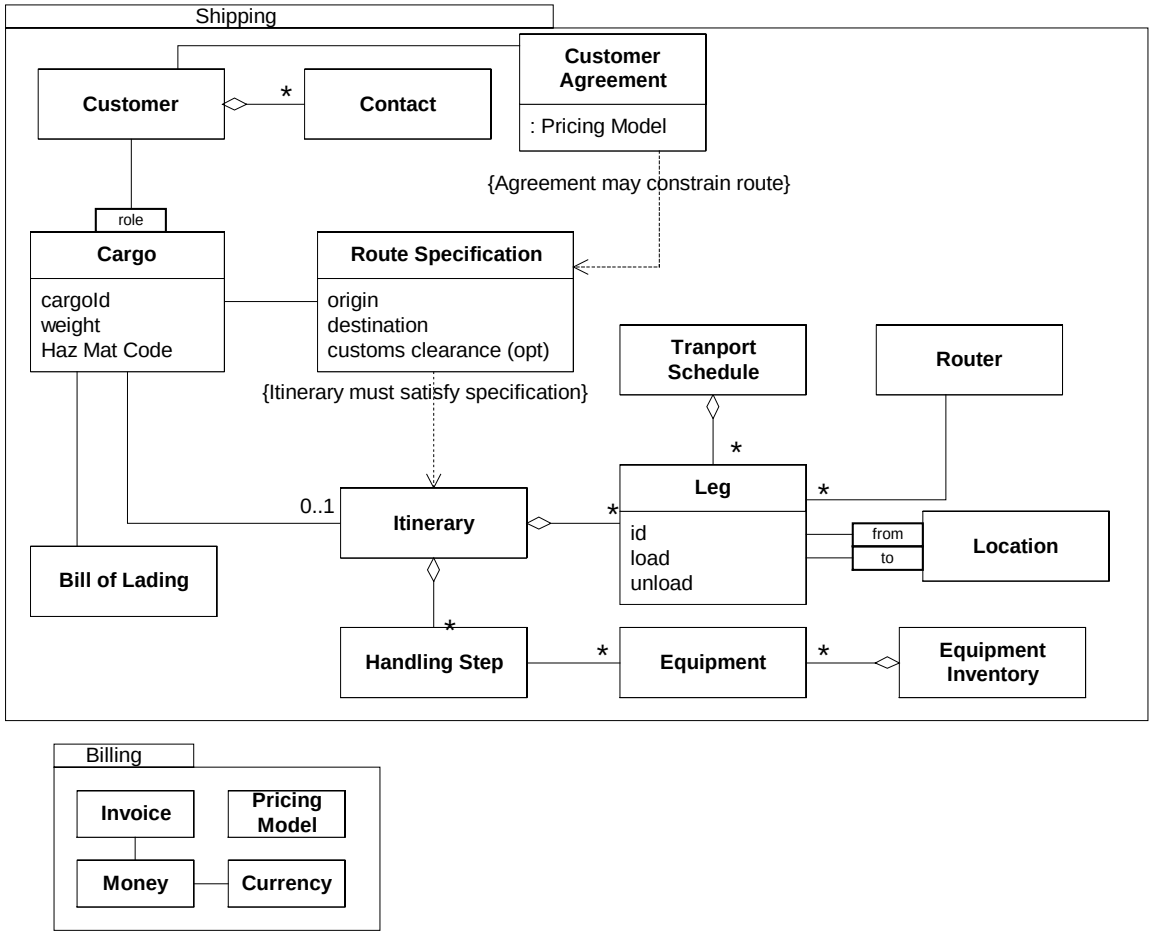


Figure 15. 1

Note that this is highly simplified compared to what would likely be needed for a real application. A realistic model would be too cumbersome for an example. Therefore, while this example may not be complicated enough to drive us to a SEGREGATED CORE, take a

leap of imagination to treat this model as being too complicated to easily interpret and deal with as a whole.

Now, what is the essence of the shipping model? Usually a good place to start looking is the "bottom line". This might lead us to focus on pricing and invoices. But we really need to look at the DOMAIN VISION STATEMENT. Here is an excerpt from this one.

> …Increase visibility of operations and provide tools to fulfill customer requirements faster and more reliably…

This application is not being designed for the sales department. It is going to be used by the front-line operators of the company. So let's relegate all money related issues to (admittedly important) supporting roles. Someone has already placed some of these items into a separate package (**Billing**). We can keep that, and further recognize that it is a supporting package.

The focus needs to be on the cargo handling – delivery of the cargo according to customer requirements. Extracting the classes most directly involved in these activities produces a SEGREGATED CORE in new package called "**Delivery**".

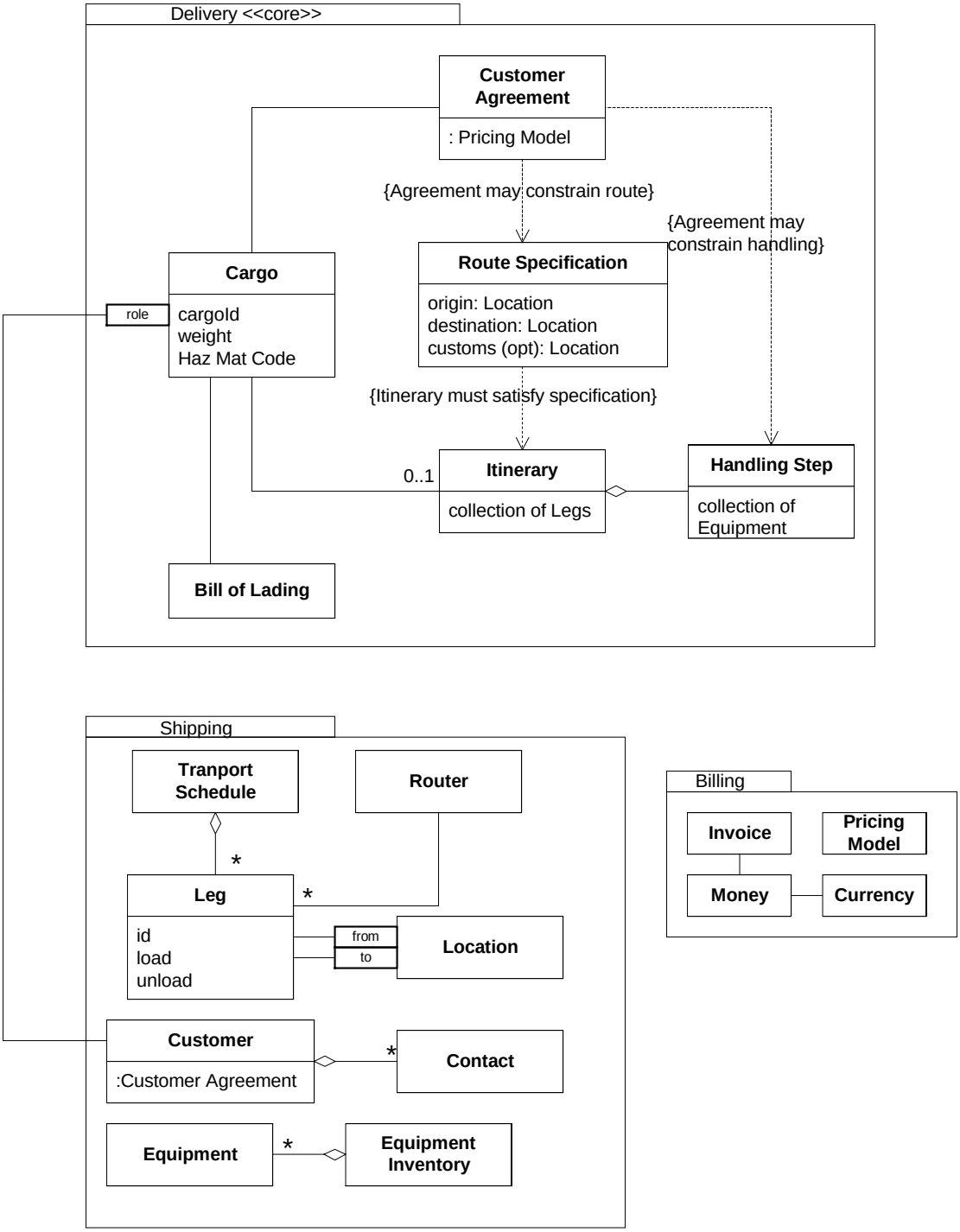


Figure 15. 2

For the most part, classes have just moved into the new package, but there have been a few changes to the model itself.

First, the **Customer Agreement** now constrains the **Handling Step**. This is typical of the insights that tend to arise as the team segregates the core. As attention is focused on

effective, correct delivery, it becomes clear that the delivery constraints in the **Customer Agreement** are fundamental and should be *explicit* in the model.

The other change is more pragmatic. In the refactored model, the **Customer Agreement** is attached directly to the **Cargo**, rather than requiring a navigation through the **Customer**. (It will have to be attached when the cargo is booked, just as the **Customer** is.) At actual delivery time, the customer is not as relevant to operations as the agreement itself, so the most important scenarios are as simple and direct as possible. In the other model, the correct **Customer** had to be found, according to the role it played in the shipment, and then queried for its **Customer Agreement**. This interaction would clog up every story you set out to tell about the model. Now it becomes easy to pull the Customer out of the core altogether.

And what about pulling customer out, anyway? Certainly the focus is on fulfilling the customer's requirements, so at first it seems to belong in the core. Yet the interactions during delivery do not usually need to involve the **Customer** class now that the **Customer Agreement** is available directly. And the basic model of a customer is pretty generic.

A strong argument could be made for **Leg** remaining in the core. I tend to be minimalist in the core, and the **Leg** has tighter cohesion with **Transport Schedule**, **Routing Service**, and **Location**, none of which needed to be in the core. But if a lot of the stories I wanted to tell about this model involved **Legs**, I'd move it into the **Delivery** package and suffer the awkwardness of its separation from those other classes.

In this example, all the class definitions are the same as before, but often distillation requires refactoring the classes themselves to separate the generic and domain specific responsibilities, which can then be segregated.

Now that we have a SEGREGATED CORE, the refactoring is complete. But the **Shipping** package we are left with is just "everything left-over after we pulled out the core". We can follow up with other refactorings to get more communicative packaging.

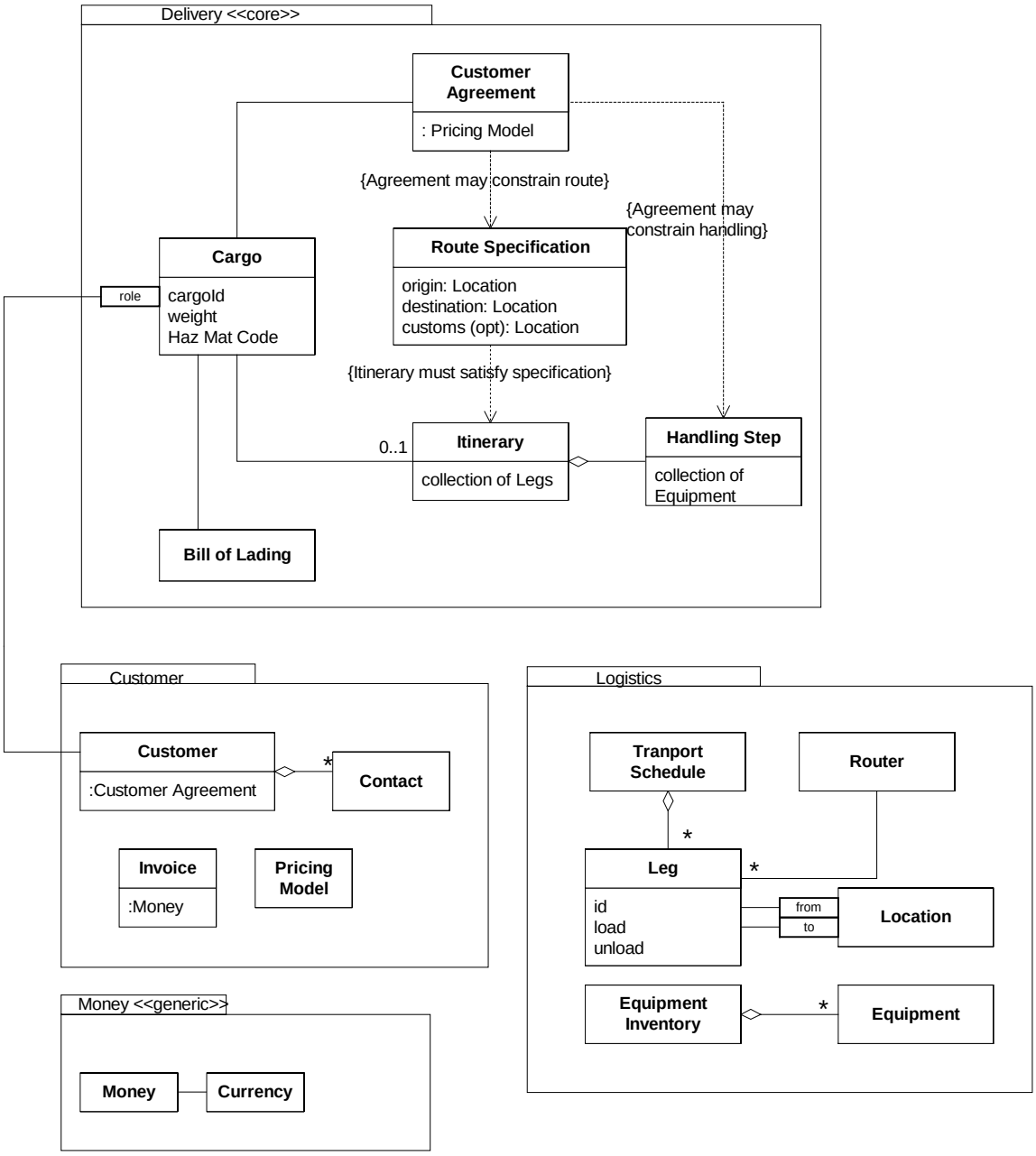


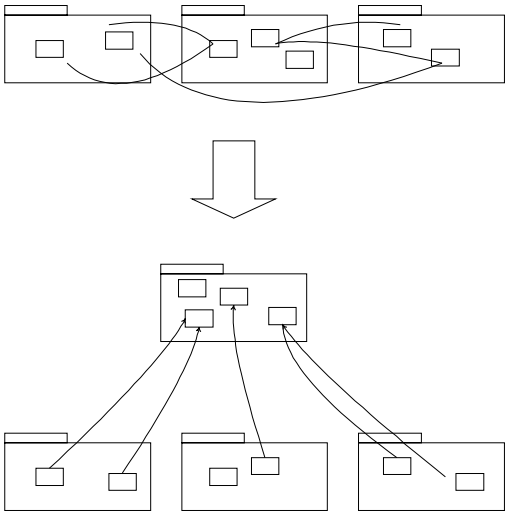
Figure 15. 3

It might take several refactorings to get to this point; it doesn't have to be done at once. Here, we've ended up with one SEGREGATED CORE package, one GENERIC SUBDOMAIN, and two domain specific packages in supporting roles. (Deeper insight might eventually produce a GENERIC SUBDOMAIN for **Customer**, or it might end up more specialized for shipping.)

Recognizing conceptual packages that communicate and provide convenience (by grouping cohesive, coupled elements) requires understanding and experience with the domain. The business experts can and should be involved in reaching such insights. As

the developers work with the model more and more they will come to also have insight. These insights, crunched knowledge, can be captured in the model.

ABSTRACT CORE



Even the CORE DOMAIN model usually has so much detail that communicating the big-picture can be difficult.

We usually deal with this by breaking the large model into narrower subdomains that are small enough to be grasped. This reductive style of packaging often works to make a complicated model manageable. But sometimes creating separate packages can obscure or even complicate the interactions between the subdomains.

When there is a lot of interaction between subdomains in separate packages, either many references will have to be created between packages, which defeats much of the value of the partitioning, or the interaction has to be made more indirect, which makes the model less communicative, harder to understand.

Consider slicing horizontally rather than vertically. Polymorphism gives us the power to ignore a lot of the detailed variation among instances of an abstract type. If most of the interactions across the conventional packages can be expressed at the level of these polymorphic interfaces, it may make sense to refactor these types into a special core package.

We are not looking for a technical trick here. This is only a valuable technique when the polymorphic interfaces correspond to fundamental concepts in the domain. In that case, separating these abstractions accomplishes both the technical advantages of tighter coupling within packages and looser coupling between them and the fundamental objective of a more communicative design.

Therefore,

Identify the most fundamental conceptual elements in the model and factor them into distinct classes, abstract classes, or interfaces. Design this abstract model so that it expresses most of the interaction between significant components. Place this abstract overall model in its own package, while the specialized, detailed implementation classes are placed in their own packages defined by subdomain.

Most of the specialized classes will now reference the core package, but not the other specialized packages. The abstract core gives a very succinct view of the main concepts and their interactions.

The process of factoring out the abstract core is not mechanical. For example, if all the classes that were frequently referenced across packages were simply moved into a separate package, the likely result would be a meaningless mess. It requires a deep understanding of the key concepts and roles they play in the major interactions of the system. And it usually requires some redesign. In other words, it is an example of refactoring to deeper insight.

The ABSTRACT CORE should end up looking a lot like the Distillation Document (if both were used on the same project, and the Distillation Document had evolved with the application as insight deepened). Of course, the ABSTRACT CORE will be code, and therefore more rigorous and more complete.

Deep Models Distill

Distillation does not operate only on the gross level of separating parts of the domain away from the core. It also means refining those subdomains, especially the core, through continuously refactoring toward deeper insight, driving toward a deep model and supple design. The goal is a design that makes the model obvious, a model that expresses the domain simply. A deep model distills the most essential aspects of a domain into simple elements that can be combined to solve the important problems of the application.

While a breakthrough to a deep model provides value anywhere it happens, it is in the CORE DOMAIN that it can change the trajectory of an entire project.

Choosing Refactoring Targets

When you encounter a large system that is poorly factored, where do you start? In the XP community, the answer tends to be either one of these:

1. Just start anywhere, since it all has to be refactored.
2. Wherever it is hurting. I'll refactor what I need to in order to get my specific task done.

I don't hold with either of these. The first is impractical except in a few projects staffed entirely with top programmers. The second tends to pick around the edges, treating symptoms and ignoring root causes, shying away from the worst tangles. Eventually the code becomes harder and harder to refactor.

So, if you can't do it all, and you can't be pain-driven, what do you do?

1. In a pain-driven refactoring, you look to see if the root involves the CORE DOMAIN or the relationship of the CORE to a supporting element. If it is, you bite the bullet and fix that first.
2. When you have the luxury of refactoring freely, you focus first on better factoring of the CORE DOMAIN, on improving the segregation of the CORE, and on purifying supporting subdomains to be GENERIC.

This is how to get the most bang for your refactoring buck.

16.Large-Scale Structure

Thousands of people worked independently to create the AIDS Quilt.

A small Silicon Valley design firm had been contracted to create a simulator for a satellite communications system. Work was progressing well. An object model was developing that could express and simulate a wide range of network conditions and failures.

But the lead developers on the project were uneasy. The problem was inherently complex. Driven by the need to clarify the intricate relationships in the model, they had decomposed the design into coherent modules of manageable size. Now there were a *lot* of modules. Which package should a developer look in to find a particular aspect of functionality? Where should a new class be placed? What did some of these little packages really mean? How did they all fit together? And there was still more to build.

The developers had good communication with each other and could still figure out what to do from day to day, but the project leaders were not content to skirt the edge of comprehensibility. They wanted some way of organizing the design so that it could be understood and manipulated as it moved to the next level of complexity.

They brainstormed. There were a lot of possibilities. Alternative packaging schemes were proposed. Maybe some document could give an overview of the system, or some new views of the class diagram in the modeling tool could guide a developer to the right module. But the project leaders weren't satisfied with these gimmicks.

They knew they could tell a simple story of their system, of the way data would be marshaled through an infrastructure, its integrity and routing assured by layers of telecommunications technology. Every detail of that story was in the model, yet the broad arc of the story could not be seen.

Some essential concept from the domain was missing. But this time it was not a class or two missing from the object model, it was a missing structure for the model as a whole.

After mulling over the problem for a week or two, the idea began to gel. They would impose a structure on the design. The entire simulator would be viewed as a series of layers related to aspects of the communications system being simulated. The bottom layer would represent the physical infrastructure, the basic ability to transmit bits from one node to another. Then there would be a packet routing layer that brought together the concerns of how a particular data stream would be directed. Other layers would identify other conceptual levels of the problem. *These layers would outline their story of the system.*

They set out to refactor the code to conform to the new structure. Modules had to be redefined so as not to span layers. In some cases, object responsibilities were refactored so that each object would clearly belong to one layer. Conversely, throughout this process the definitions of the conceptual layers themselves were refined based on the hands-on experience of applying them. The layers, modules and objects coevolved until, in the end, the entire design followed the contours of this layered structure.

These layers were not modules nor any other artifact in the code. They were an overarching set of rules that constrained the boundaries and relationships of any particular module or object throughout the design, even at interfaces with other systems.

Imposing this order brought the design back to comfortable intelligibility. People knew roughly where to look for a particular function. Individuals working independently could make design decisions that were broadly consistent with each other. The complexity ceiling had been lifted.

♦ ♦ ♦

Even with a MODULAR breakdown, a large model can be too complicated to grasp. The MODULES chunk the design into manageable bites, but there may be many of them. Also, this does not necessarily bring uniformity to the design. Object to object, package to package, a jumble of design decisions may be applied, each defensible, but idiosyncratic.

The strict segregation imposed by BOUNDED CONTEXTS prevents corruption and confusion, but does not, in itself, make it easier to see the system as a whole.

Distillation does help by focusing the attention on the core and presenting the other models in their supporting roles. But it is still necessary to understand the supporting elements and their relationships to the CORE DOMAIN and to each other. And, while the CORE DOMAIN would ideally be so clear and easily understood that no additional guidance would be needed, we are not always at that point.

On a project of any size, people must work somewhat independently on different parts of the system. Without any coordination or rules, a confusion of different styles and distinct solutions to the same problems arise, making it hard to understand how the parts fit together and impossible to see the big picture. Learning about one part of the design will not transfer to other parts, so the project will end up with specialists in different modules who cannot help each other outside their narrow range. CONTINUOUS INTEGRATION breaks down and the BOUNDED CONTEXT fragments.

In a large system without any overarching principle that allows elements to be interpreted in terms of their role in patterns that span the whole design, **developers cannot see the forest for the trees**. We need to be able to understand the role of an individual part in the whole without delving into details of the whole.

A " large-scale structure" is a language that lets you discuss and understand the system in broad strokes. A set of high-level concepts and/or rules establishes a pattern of design for an entire system. This organizing principle can guide design as well as aid understanding. It helps coordinate independent work because there is a shared concept of the big picture of the role of parts and the shape of the whole.

You can't represent most large-scale structures in UML, and you don't need to. Most large-scale structures shape and explain the model and design, but do not appear in it. They provide an extra level of communication about the design. In the examples of this chapter you'll see many informal UML diagrams on which I've superimposed information about the large-scale structure.

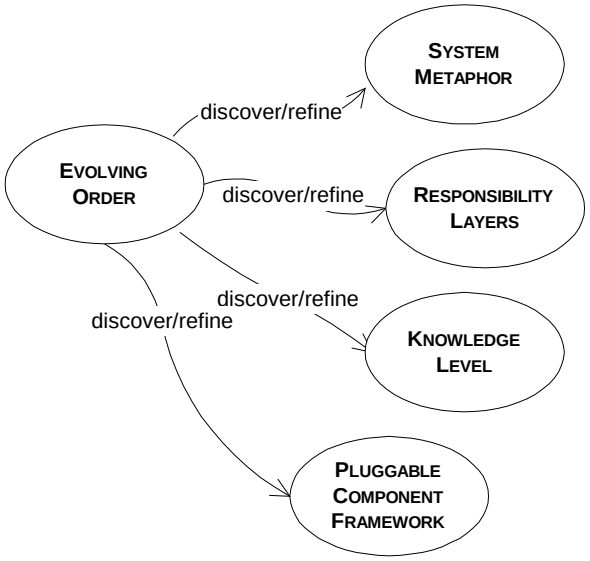
Devise a pattern of rules or roles and relationships that will span the entire system and which allows some understanding of a part's place in the whole even without detailed knowledge of the part's responsibility.

Structure may be confined to one BOUNDED CONTEXT but will usually span more than one, providing the conceptual organization to hold together all the teams and subsystems involved in the project. A good structure gives insight into the model and complements distillation.

When a team is reasonably small and the model is not too complicated, decomposition into well named MODULES, a certain amount of distillation, and informal coordination among developers can be sufficient to keep the model organized.

Large-scale structure can save a project, but an ill-fitting structure can severely hinder development. This chapter explores patterns for successfully structuring a design at this level.

Some Patterns of Large-Scale Structure



EVOLVING ORDER

Many developers have experienced the cost of an unstructured design. To avoid this anarchy, projects impose architectures that constrain development in various ways. Some technical architectures do solve technical problems, such as networking or data persistence, but when architectures start venturing into the arena of the application and domain model they can create problems of their own. They often prevent the developers from creating designs and models that work well for the specifics of the problem. The most ambitious ones can even take away from application developers familiarity and technical power of the programming language itself. And whether technical or domain oriented, architectures freeze a lot of up-front design decisions that can become a straight jacket as requirements change and as understanding deepens.

While some technical architectures have become prominent over the years (e.g., J2EE), large-scale structure in the domain layer has not been explored much. Needs vary widely from one application to the next. An upfront imposition of a large-scale structure is likely to be costly. As development proceeds, you will almost certainly find a more suitable structure, and you may even find that the prescribed structure is prohibiting you from taking a design route that would greatly clarify or simplify the application. You may be able to use some of the structure, but you're forgoing opportunities. Your work slows down as you try workarounds or try to negotiate with the architects. But your managers think the architecture is done. It was supposed to make this application easy, so why aren't you working on the application instead of all these architecture problems? The managers and architecture teams may even be open to input, but if each change is a heroic battle, it is too exhausting.

Design free-for-alls produce systems no one can make sense of as a whole, that are very difficult to maintain. But architectures can straightjacket a project with up-front design assumptions, and take too much power away from the developers/designers of particular parts of the application. Soon, developers will dumb down the application to fit the structure, or they will subvert it and have no structure at all, bringing along the problems of uncoordinated development.

The problem is not the existence of guiding rules, but rather the rigidity and source of those rules. If the rules governing the design really fit the circumstances, they will not get in the way but actually push development in a helpful direction, as well as providing consistency.

Therefore,

Let this conceptual large-scale structure evolve with the application, possibly changing to a completely different type of structure along the way. Don't over-constrain the detailed design and model decisions that must be made with detailed knowledge.

Individual parts have natural or useful ways of being organized and expressed that may not apply to the whole, so imposing global rules makes these parts less ideal. Choosing to use a large-scale structure favors manageability of the model as a whole over optimal structuring of the individual parts. Therefore, there will be some compromise between unifying structure and freedom to express individual components in the most natural way. This can be mitigated by careful selection of the structure and by avoiding over-constrictive structures. A really nice fit of structure to domain and requirements actually makes detailed modeling and design easier, by helping to quickly eliminate a lot of options.

It can also give shortcuts to design decisions that could, in principle, be found by working on the individual object level, but which would, in practice, take too long and have inconsistent results. Of course, continuous refactoring is still necessary, but this will make it a more manageable process, and can help make different people come up with consistent solutions.

A large-scale structure generally needs to be applicable across BOUNDED CONTEXTS. Through iteration on a real project, a structure will lose features that tightly bind it to a particular model and evolve features that correspond to deep insights into the domain. This doesn't mean that it will have *no* assumptions about the model, but not about the points that vary within the particular models in use on the project.

Designers may have no control over the model of some parts of the system, especially in the case of external or legacy systems. So large-scale structure must accommodate such constraints on development.

This may be done by changing the structure to better fit the specific external elements. It may be done by specifying ways in which the application relates to externals. It may be done by making the structure loose enough to flex around awkward realities.

Unlike the CONTEXT MAP, using a large-scale structure is optional. One should be applied when costs and benefits favor it, and when a fitting structure is found. In fact, it is not needed for systems that are simple enough to be understood when broken into MODULES. **Large-scale structure should be applied when a structure can be found that greatly clarifies the system without forcing an unnatural constraint into model development. Since an ill-fitting structure is worse than none, it is best not to shoot for comprehensiveness, but a minimal set that solves the problems that have emerged. Less is more.**

A large-scale structure can be very helpful and still have a few exceptions, but those exceptions need to be flagged somehow, so that developers can assume the structure is being followed unless otherwise noted. And if those exceptions start to get numerous, the structure needs to be changed or discarded.

♦ ♦ ♦

As mentioned, it is no mean feat to create a structure that gives the necessary freedom to developers while still averting chaos. Although a lot of work has been done on technical architectural for software systems, little has been published on the structuring of the DOMAIN LAYER. Some approaches that have been tried weaken the object-oriented paradigm, such as those that break down the domain by application task or by use-case. This whole area is still undeveloped. I've observed a few general patterns of LARGE-SCALE STRUCTURES that have emerged on various projects. I'll discuss four in this chapter. One of these may fit your needs or lead to ideas for a STRUCTURE tailored to your project.

SYSTEM METAPHOR

Metaphorical thinking is pervasive in software development, especially with models. But the Extreme Programming practice of "metaphor" has come to mean a particular way of using a metaphor to bring order to the development of a whole system.

♦ ♦ ♦

Just as a firewall can save a building from a fire raging through neighboring buildings, a software "firewall" protects the local network from the dangers of the larger networks outside. This metaphor has influenced network architectures and shaped a whole product category. Multiple competing firewalls are available for consumers, developed independently, understood to be somewhat interchangeable. Novices to networking readily grasp the concept. This shared understanding throughout the industry and among customers is due in no small part to the metaphor.

Yet it is an inexact analogy, and its power cuts both ways. The use of the firewall metaphor has led to development of software barriers that are sometimes insufficiently selective and impede desirable exchanges, while offering no protection against threats originating within the wall. Wireless LANs, for example, are vulnerable. The clarity of the firewall has been a boon, but all metaphors carry baggage.[7]

Software designs tend to be very abstract and hard to grasp. Developers and users alike need tangible ways to understand the system and share a view of the system as a whole.

On one level, metaphor runs so deeply in the way we think that it pervades every design. Systems have "layers" that "lay on top" of each other. They have "kernels" at their centers. But sometimes a metaphor comes along that can convey the central theme of a whole design and provide a shared understanding among all team members.

When this happens, the system is actually shaped by the metaphor. A developer will make design decisions consistent with the system metaphor. This consistency will enable other developers to interpret the many parts of a complex system in terms of the same metaphor. The developers and experts have a reference point in discussions that may be more concrete than the model itself.

A SYSTEM METAPHOR is a loose, easily understood large-scale structure that it is harmonious with the object paradigm. Because the SYSTEM METAPHOR is only an analogy to the domain anyway, different models can map to it in an approximate way, which allows it to be applied in multiple BOUNDED CONTEXTS, helping to coordinate work between them.

SYSTEM METAPHOR has become a popular approach because it is one of the core practices of Extreme Programming [Beck2000]. Unfortunately, few projects have found really useful metaphors, and people have tried to push it into domains where it is counterproductive. A persuasive metaphor introduces the risk that the design will take on aspects of the analogy that are not desirable for the problem at hand, or that the analogy, while seductive, may not be apt.

That said, it is a well-known form of large-scale structure that is useful on some projects, and it nicely illustrates the general concept of a structure.

When a concrete analogy to the system emerges that captures the imagination of team members and seems to lead thinking in a useful direction, adopt it as a large-scale structure. Organize the design around this metaphor and absorb it into the UBIQUITOUS LANGUAGE. The system metaphor should both facilitate communication about the system and guide development of it. This increases consistency in different parts of the system, potentially even across different BOUNDED CONTEXTS.

[7] System metaphor finally made sense to me when I heard Ward Cunningham use this firewall example in a workshop lecture.

Since all metaphors are inexact, continuously reexamine the metaphor for overextension or inaptness, and be ready to drop it if it gets in the way.

♦ ♦ ♦

The "Naïve Metaphor" and Why We Don't Need It

Because a useful metaphor doesn't present itself on most projects, the XP community has come to talk of the "naïve metaphor", by which they mean the domain model itself.

One trouble with this term is that a mature domain model is anything but naïve. In fact, "payroll processing is like an assembly line" is likely a much more naïve view than a model that is the product of many iterations of knowledge crunching with user experts, and that has been proven by being tightly woven into the implementation of a working application.

The term "naïve metaphor" should be retired.

SYSTEM METAPHORS are not useful on all projects. Large-scale structure in general is not essential. In the twelve practices of Extreme Programming, the role of SYSTEM METAPHOR could be fulfilled by UBIQUITOUS LANGUAGE. Projects should augment that LANGUAGE with METAPHORS or other large-scale structures when they find one that fits well.

RESPONSIBILITY LAYERS

Throughout this book, individual objects have been assigned narrow sets of related responsibilities. **When each individual object has handcrafted responsibilities, there are no guidelines, no uniformity, and no ability to handle large swaths of the domain together.** To give coherence to a large model, it is useful to impose some structure on the assignment of those responsibilities. Responsibility-driven design also applies to larger scales.

With a deep understanding of a domain, broad patterns start to become visible. Some domains have a natural stratification. Certain concepts and activities take place against a background of other elements that change independently and at a different rate for different reasons. How can we take advantage of this natural structure, to make it more visible and useful? It suggests layering, one of the most successful architectural design patterns [BMRSS1996 among others].

Layers are partitions of a system in which the members of each partition are aware of and able to use the services of the layers "below", but unaware of and independent of the layers "above". When the dependencies of modules are drawn, they are often laid out so that a module with dependents appears below its dependents. In this way, layers sometimes sort themselves out so that none of the objects in the lower levels are conceptually dependent on those in higher layers.

But this ad hoc layering, while it can make tracing dependencies easier, and sometimes makes some intuitive sense, doesn't give much insight into the model or guide modeling decisions. We need something more intentional.

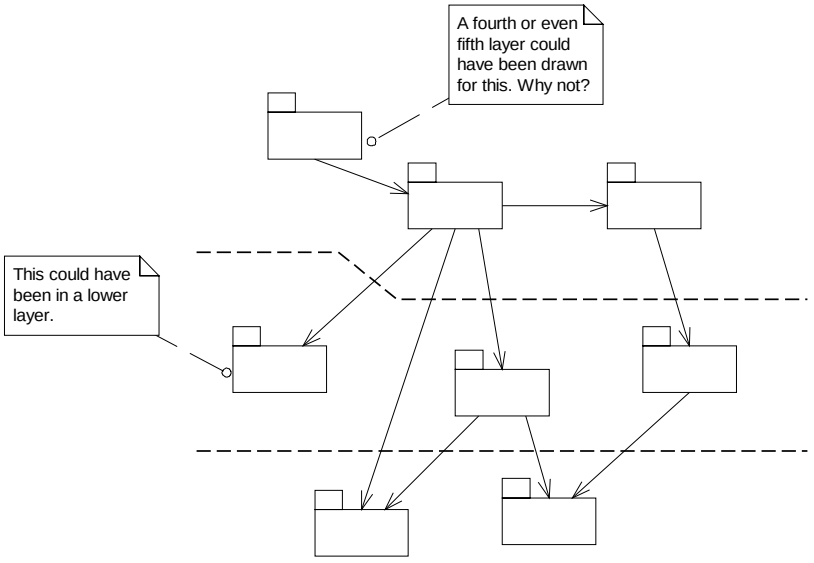


Figure 16. 1 Ad hoc layering: What are these packages about?

In a model with a natural stratification, conceptual layers can be defined around major responsibilities, uniting the two powerful principles of LAYERING and RESPONSIBILITY DRIVEN DESIGN.

These responsibilities must be considerably broader than those typically assigned to individual objects, as shall be seen below. As individual MODULES and AGGREGATES are designed, they are factored to keep them within the bounds of one of these major responsibilities. This named grouping of responsibilities by itself could enhance the comprehensibility of a modularized system, since the responsibilities of MODULES could be more readily understood. But combining high-level responsibilities with layering give us an organizing

principle for a system. (The layering pattern that best fits is the variant called RELAXED LAYERED SYSTEM, [BMRSS96], p. 45, which allows components of a layer access to any lower layer, not just the one immediately below.)

Look at the conceptual dependencies in your model and the varying rates and sources of change of different parts of your domain. Identify the natural strata in the domain and cast them as broad abstract responsibilities. These responsibilities should tell a story of the high-level purpose and design of your system. Refactor the model so that the responsibilities of each domain object, AGGREGATE and MODULE fits neatly within the responsibility of one layer.

This is a pretty abstract description, but it will become clear with a few examples. The satellite communications simulator whose story opened this chapter layered its responsibility. The following example explores responsibility layers in detail to give a feel for the discovery of a large-scale structure and the way that it guides and constrains modeling and design.

In Depth Example: Layering a Shipping System

Let's look at the implications of applying RESPONSIBILITY LAYERS to the cargo shipping application discussed in the examples of previous chapters.

As we rejoin the story, the team has made considerable progress creating a MODEL-DRIVEN DESIGN and distilling a CORE DOMAIN. But as the design fleshes out, they are having trouble keeping coordinated on how all the parts fit together. They are looking for a large-scale structure that can bring out the main themes of their system and keep everyone on the same page.

Here is a simplified look at a representative part of the model.

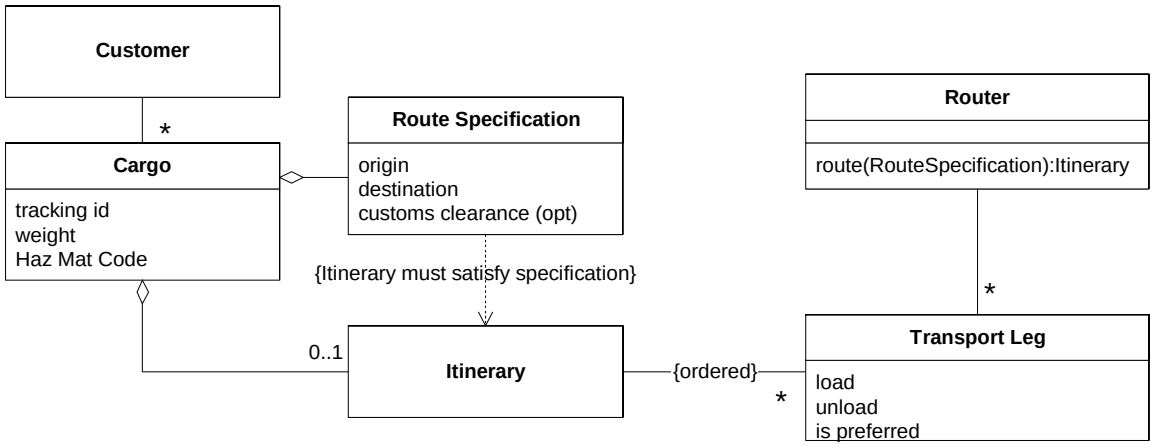


Figure 16. 2 Basic Shipping Domain Model for Routing Cargos

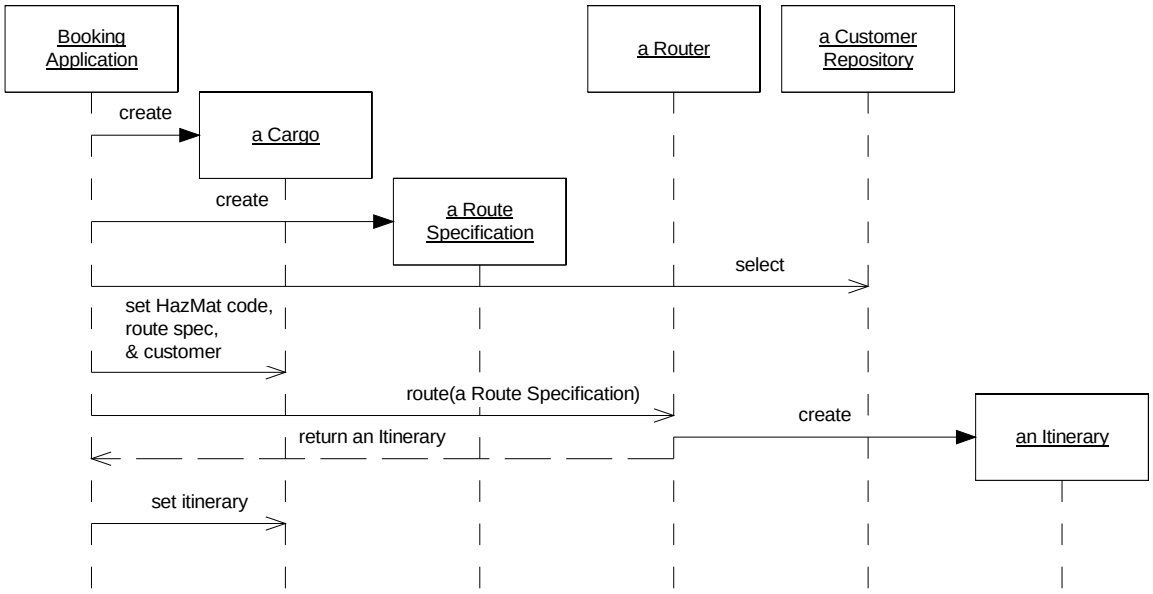


Figure 16. 3 Using the Model to Route a Cargo During Booking

The team members have been steeped in the domain of shipping for months, and they have noticed some natural stratification of its concepts. It is quite reasonable to discuss transport schedules (the scheduled voyages of ships and trains) without referring to the cargoes aboard those transports. It is harder to talk about tracking a cargo without referring to transport carrying it. The conceptual dependencies are pretty clear. The team can readily distinguish two layers: “Operations”, and the substrate of those operations, which they dub “Capability”.

“Operational” Responsibilities

Activities of the company, past, current, and planned, are the collected into the Operations layer. The most obvious Operations object is **Cargo**, which is the focus of most of the day-to-day activity of the company. The **Route Specification** is an integral part of **Cargo**, indicating delivery requirements. The **Itinerary** is the operational delivery plan. Both of these objects are part of the **Cargo’s** AGGREGATE, and their lifecycles are tied to the timeframe of an active delivery.

“Capability” Responsibilities

This layer reflects the resources the company draws upon in order to carry out operations. The **Transit Leg** is a classic example. The ships are scheduled to run and have a certain capacity to carry cargo which may or may not be fully utilized.

True, if we were focused on operating a shipping fleet, **Transit Leg** would be in the Operations layer. But the users of this system aren’t worried about that problem. (If the company were involved in both those activities and wanted the two coordinated, the development team might have to consider a different layering scheme, perhaps with two distinct layers such as “Transport Operations” and “Cargo Operations”.)

A trickier decision is where to place **Customer**. In some businesses, customers tend to be transient, interesting while a package is being delivered and then mostly forgotten until next time they send something. This makes them an operational concern. This would be

the typical case for a parcel delivery service aimed at individual consumers. But our hypothetical shipping company tends to cultivate long-term relationships with customers and most work comes from repeat business. *Given these intentions of the business users*, the **Customer** belongs in the potential layer. As you can see, this was *not a technical decision*. It was an attempt to capture and communicate knowledge of the domain.

Since the association between **Cargo** and **Customer** can only be traversed in one direction, the **Cargo** REPOSITORY will need a query that finds all **Cargos** for a particular **Customer**. There were good reasons to design it that way anyway, but with the imposition of the large-scale structure it is now a requirement.

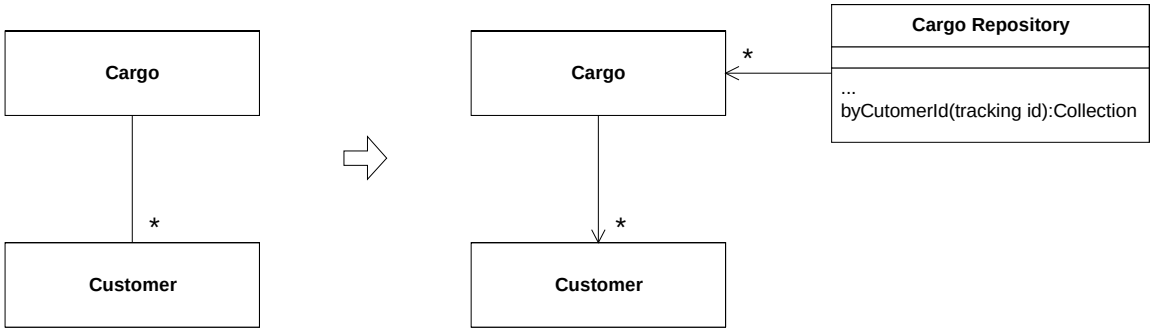


Figure 16. 4

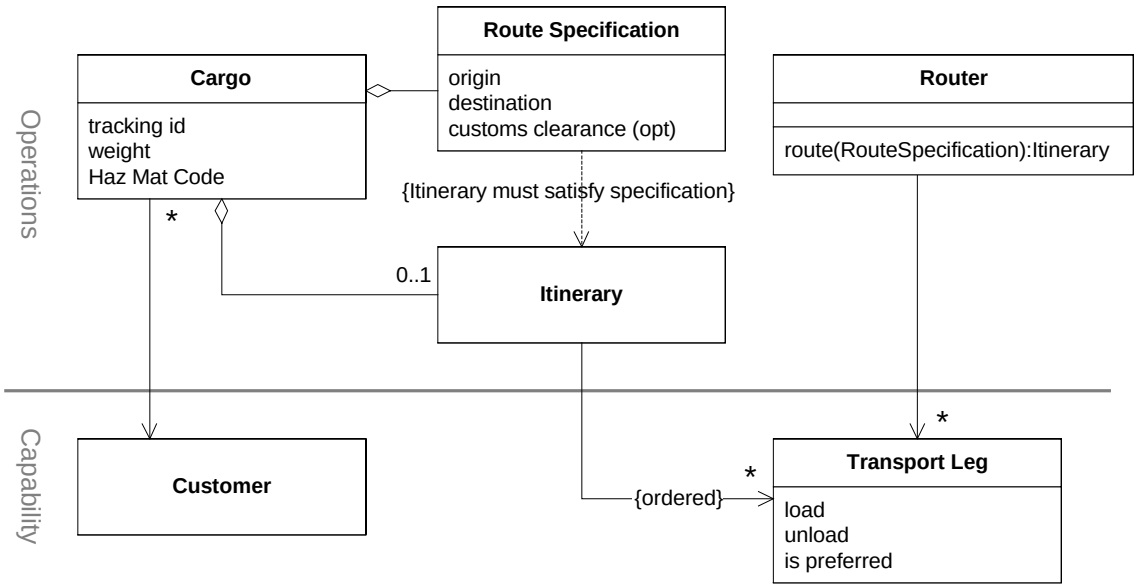


Figure 16. 5 First-pass Layered Model

While the distinction between Operations and Capability clarifies the picture, order continues to evolve. After a few weeks of experimentation, the team zeros in on another distinction. For the most part, the initial two layers both focus on situations or plans *as they are*. But the Router (and many other elements excluded from this example) isn't part of

current operational realities or plans. It helps make decisions about changing those plans. The team defines a new layer responsible for “Decision Support”.

“Decision Support” Responsibilities

This layer of the software provides the user with tools for planning and decision making, and could potentially automate some decisions. (e.g., automatically reroute cargoes when a transport schedule changes.)

The **Router** is a SERVICE that helps a booking agent choose the best way to send a cargo. This places it squarely in Decision Support.

The references within this model are all consistent with the three layers except for one discordant element: the “is preferred” attribute on **Transport Leg**. This attribute exists because the company prefers to use its own ships when it can, or the ships of certain other companies that it has favorable contracts with. The “is preferred” attribute is used to bias the Router toward these favored transports. This attribute has nothing to do with “Capability”. It is a policy that directs decision making. To use the new RESPONSIBILITY LAYERS, the model will have to be refactored.

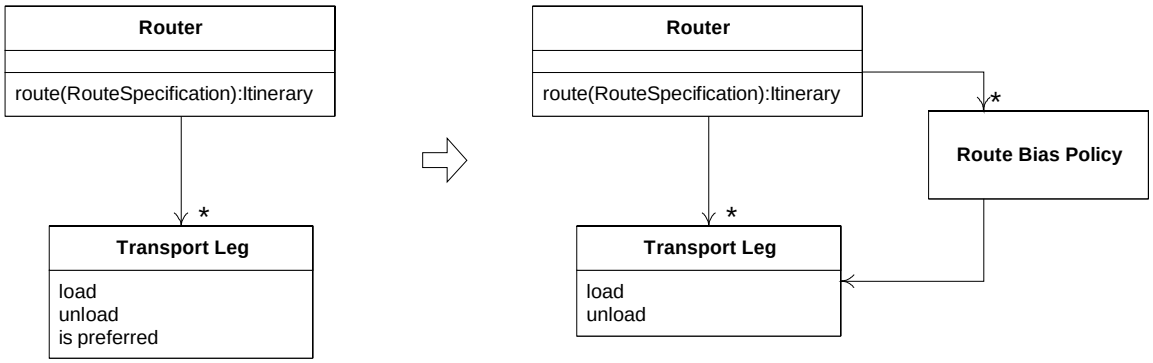


Figure 16. 6 Refactoring the Model to Conform to New Layering Structure

This factoring makes the **Route Bias Policy** more explicit while making **Transport Leg** more focused on the fundamental concept of transportation capability. A large-scale structure based on a deep understanding of the domain will often push the model in directions that clarify its meaning.

This new model now smoothly fits into the large-scale structure.

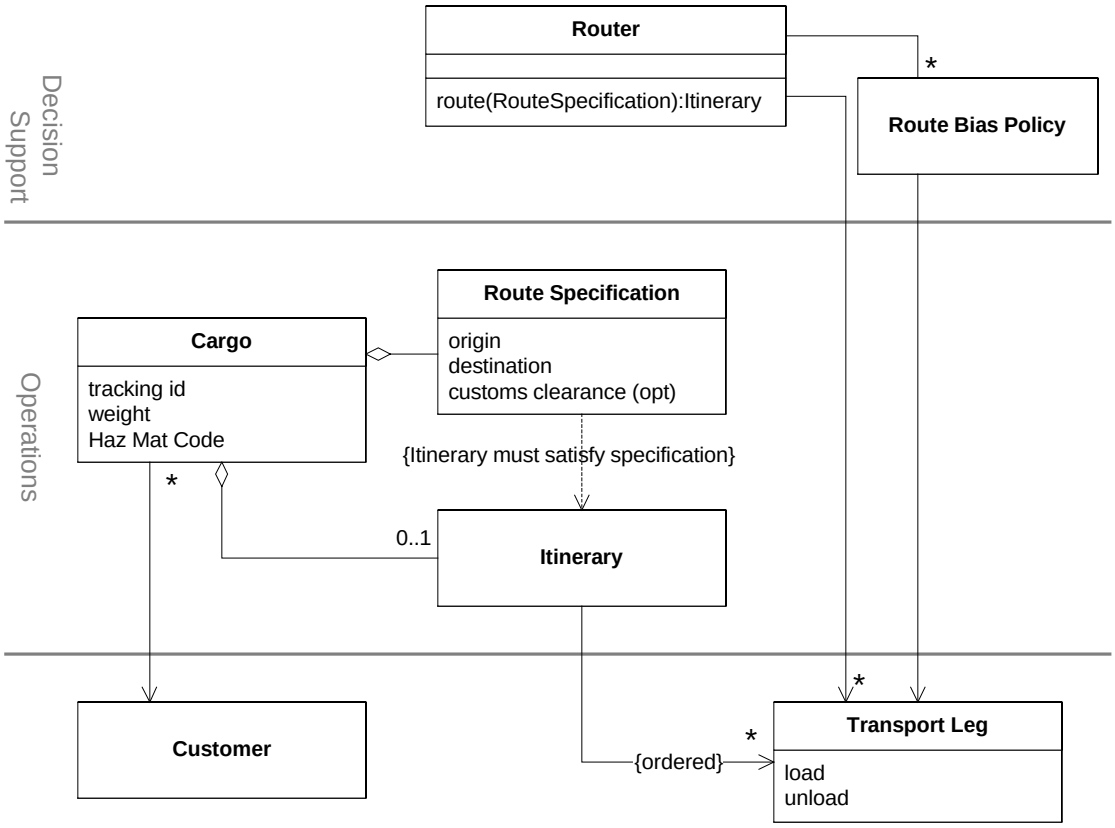


Figure 16. 7 Restructured and Refactored Model

A developer accustomed to the chosen layers can more readily discern the roles and dependencies of the parts. The value of the large-scale structure increases as the complexity grows.

Note that, although I'm illustrating this with a modified UML diagram, that is just a way of *communicating* the layering. UML doesn't include this notation, so this is additional information imposed for the sake of the reader. I'll say once again that the diagram is not the model. It is not the LARGE-SCALE STRUCTURE either. For each situation in which you would want to be able to differentiate the layers, you would need to devise a means for seeing them. If code is the ultimate design document for your project, it could be helpful to have a tool for browsing classes by layer or at least reporting them.

How Does This Structure Affect Ongoing Design?

Once a large-scale structure has been adopted, subsequent modeling and decision decisions must take it into account. To illustrate, suppose that we must add a new feature to this already layered design,. The domain experts have just told us that routing restrictions apply for certain categories of hazardous materials. Certain materials may not be allowed on some transports or in some ports. We have to make the **Router** obey these regulations.

There are many possible model and design approaches. In the absence of a LARGE-SCALE STRUCTURE, one appealing design would be to give the responsibility of incorporating these

routing rules to the object that owns the **Route Specification** and the Hazardous Material (HazMat) code – namely the **Cargo**.

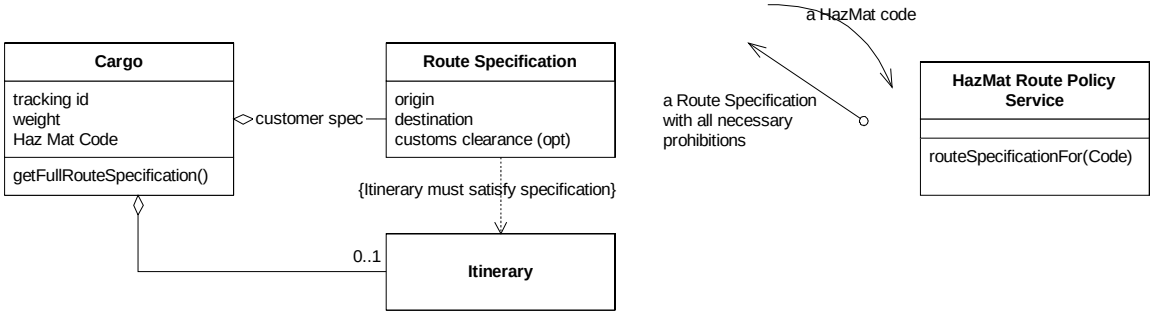


Figure 16. 8 A Possible Design for Routing Hazardous Cargo

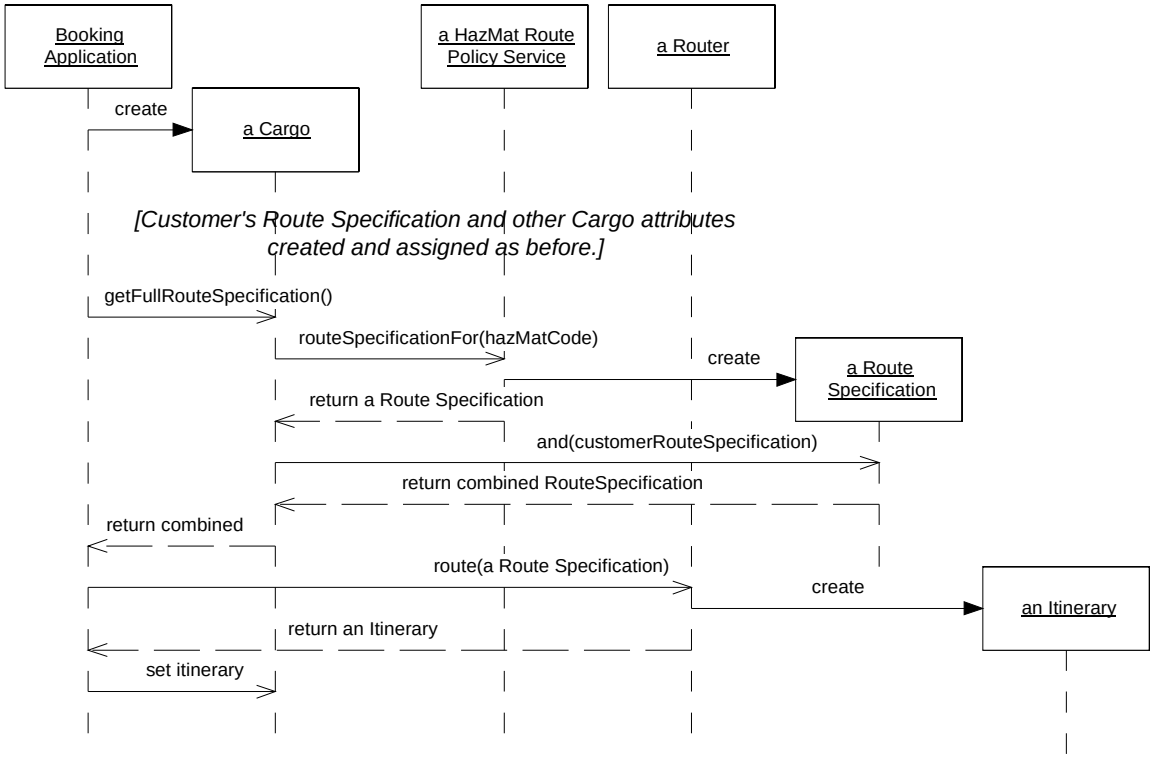


Figure 16. 9

The trouble is, this design doesn't fit the LARGE-SCALE STRUCTURE. The **HazMat Route Policy Service** is not the problem. This policy fits neatly into the responsibility of the "Decision Support" layer. The problem is the dependency of **Cargo** (an "operational" object) on **HazMat Route Policy Service** (a "Decision Support" object). As long as the project is committed to these layers, this is not an allowed design. It would confuse developers who expected the STRUCTURE to be followed.

There are always many design possibilities, and we'll just have to choose another one -- one that is follows the rules of the LARGE-SCALE STRUCTURE. The **HazMat Route Policy Service** is all right but we need to move the responsibility for using the policy. Let's try giving the **Router** the responsibility for collecting appropriate policies before searching for a route. This means a change to the **Router** interface to include objects that policies might depend on. Here is a possible design.

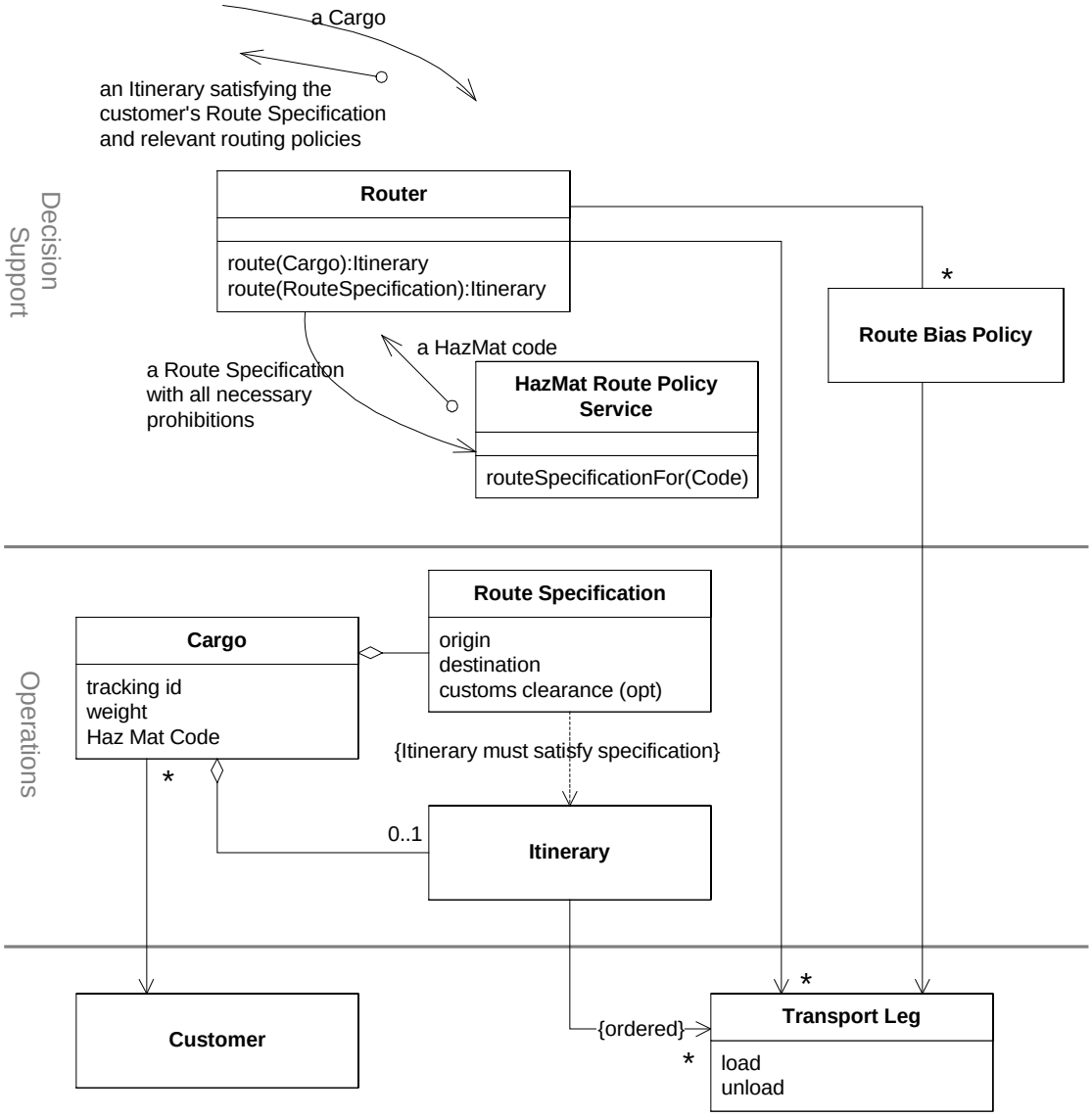


Figure 16. 10 A Design for Routing Hazardous Cargo Consistent with Layering

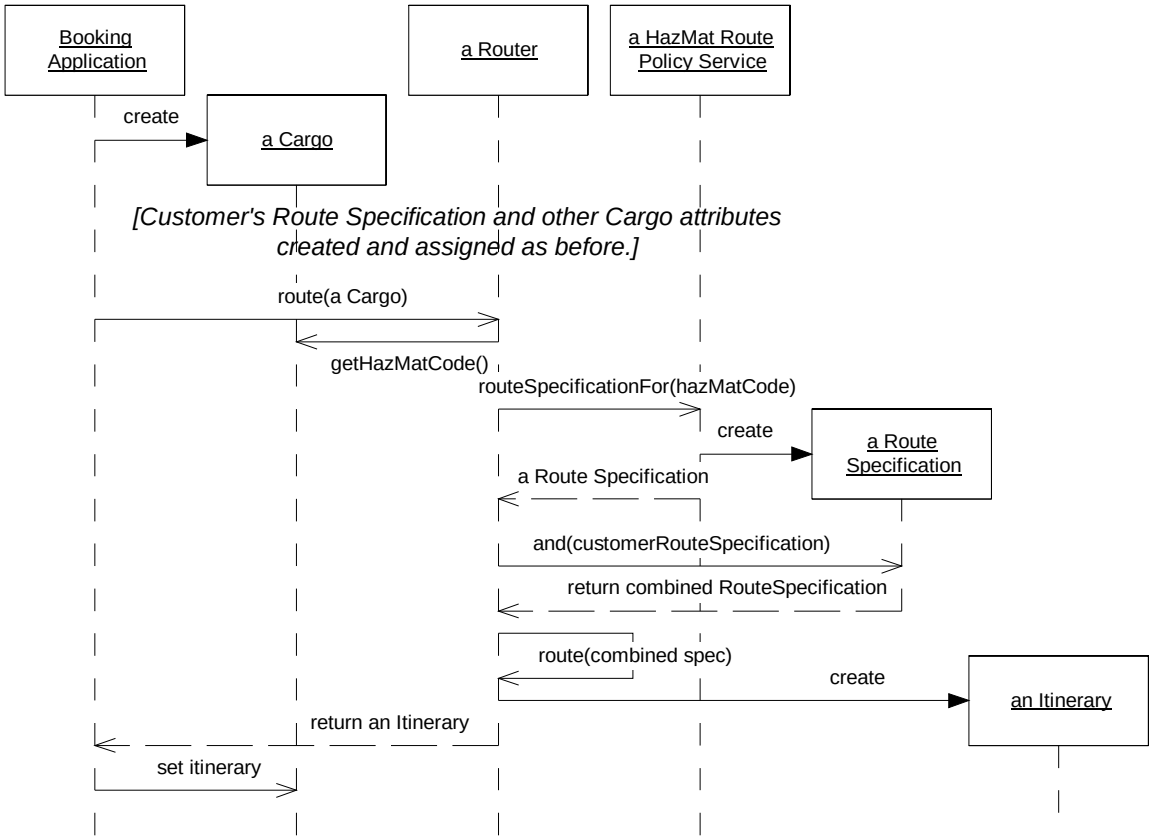


Figure 16. 11

Now, this isn't necessarily a *better* design than the other. They both have pros and cons. But if everyone on a project makes decisions in a consistent way, the design as a whole will be much more comprehensible, and that is worth some modest tradeoffs on detailed design choices.

If the structure were forcing many awkward design choices, then, in keeping with EVOLVING ORDER, it should be evaluated and perhaps modified or even discarded.

I have seen RESPONSIBILITY LAYERS used to good effect in domains as various as manufacturing control and financial management.

♦ ♦ ♦

Choosing Appropriate Layers

Finding good RESPONSIBILITY LAYERS is a matter of understanding the problem domain and of experimentation. If you allow EVOLVING ORDER, the initial starting point is not critical, although a poor choice does add work. The structure may well evolve into something unrecognizable. So the guidelines suggested here should be applied when considering transformations of the structure as much as when choosing from scratch.

As layers get switched out, merged, split, and redefined, useful characteristics to look for and preserve are:

- Storytelling. The layers should communicate basic realities or priorities of the domain. Choosing a large-scale structure is less a technical decision than a business modeling decisions. The layers should bring out the priorities of the business.
- Conceptual dependency. The concepts in the "upper" layers should have meaning against the backdrop of the "lower" layers, while the lower layer concepts should be meaningful standing alone.
- CONCEPTUAL CONTOURS. If the objects of different layers have should have different rates of change or different sources of change, the layer accommodates the shearing between them.

It isn't always necessary to start from scratch in defining layers for each new model. Certain layers show up in whole families of related domains.

In businesses based on exploiting large fixed capital assets, such as factories or cargo ships, logistical software can often be organized into a "Potential" layer (another name for the "Capability" layer in the example) and an "Operations" layer.

- *Potential*: What can be done? Never mind what we are planning to do. What *could* we do? The resources of the organization, including its people, and the way those resources are organized are the core of potential. Contracts with vendors also define potentials. This layer could be recognized in almost any business domain, but is a prominent part of the story in those, like transportation and manufacturing, that have a relatively large fixed capital investments that enable the business. Potential includes transient assets, as well, but a business primarily driven by transient assets might choose layers that emphasize this, as we'll see later.
- *Operation*: What is being done? What have we managed to make of those potentials? Like Potential, this should reflect the reality of the situation, rather than what we want it to be. In this layer we are trying to see our own efforts and activities: What we are selling, rather than what enables us to sell. It is very typical of Operational objects to reference or even be composed of Potential objects, while a potential object shouldn't reference the operations layer.

In many, perhaps most, existing systems, these two layers cover everything, although there could be some entirely different and more revealing breakdown. They track the current situation and active operational plans and issue reports or documents about it. But tracking is not always enough. When projects seek to guide or assist users, or to automate decision making, there is an additional set of responsibilities that can be organized into another layer, above Operations.

- *Decision Support*: What action should be taken or what policy should be set? This layer is for analysis and decision making. It relies on some lower layers, such as Potential or Operations to base its analysis on. Decision Support software may actively seek opportunities for current and future operations using historical information.

Decision Support systems have conceptual dependencies on another layer such as Operations or Potential because decisions aren't made in a vacuum. One of the intrinsic advantages of layers is that the lower layers can exist without the higher ones cannot exist without the lower. The higher ones cannot. A lot of projects implement "Decision Support" using data warehouse technology. The layer becomes a distinct BOUNDED CONTEXT, with a CUSTOMER/SUPPLIER relationship with the Operations software. In other projects, it is more deeply integrated, as in the extended example.

Another case is software that enforces very elaborate business rules or legal requirements, these can constitute a RESPONSIBILITY LAYER.

- *Policy*: What are the rules and goals? Rules and goals are mostly passive, but constrain the behavior in other layers. Designing these interactions can be subtle. Sometimes a Policy is passed in as an argument to a lower-level method. Sometimes the strategy pattern is applied. It works well in conjunction with a Decision Support layer which provides the means to seek the goals set by Policy, constrained by the rules set by Policy.

Policy layers can be written in the same language as the other layers, but they are sometimes implemented using rules engines. This doesn't necessarily place them in a separate BOUNDED CONTEXT. In fact, the difficulty of coordinating such different implementation technologies can be eased by fastidiously using the same model for both. When rules are written based on a different model than the objects they apply to, either the complexity goes way up or the objects get dumbed-down to keep it manageable.

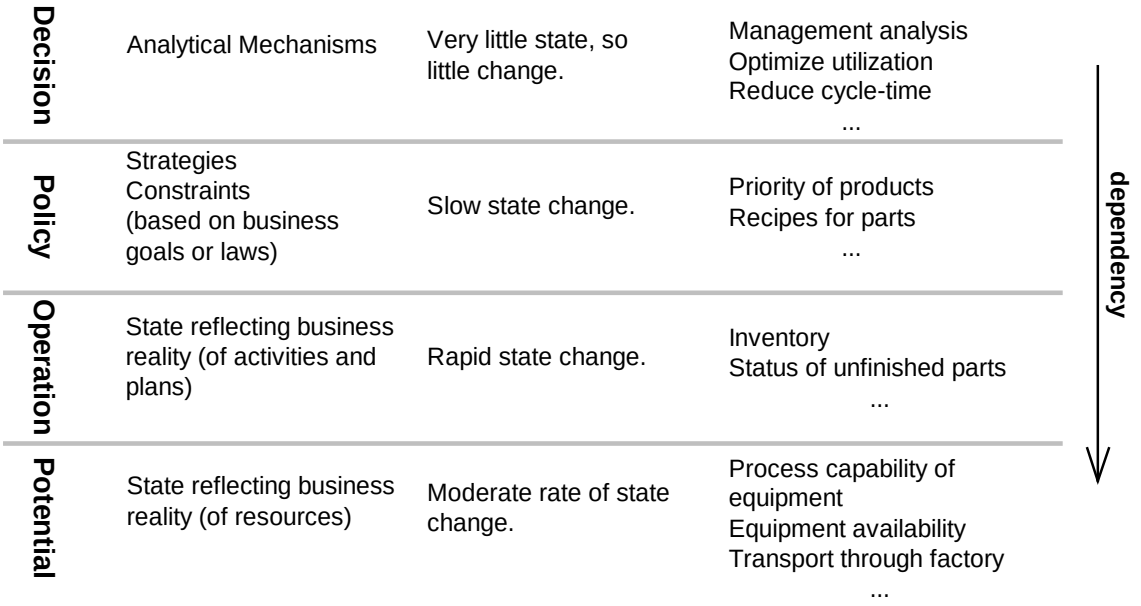


Figure 16. 12 Conceptual Dependencies and Shearing Points in a Factory Automation System

Many businesses do not base their capability on plant and equipment. In financial services or insurance, to name two, the potential is to a large extent determined by current operations. An insurance company's ability to take on a new risk by underwriting a new policy agreement is based on the diversification of its current business. The Potential layer would probably merge into Operations, and a different layering would evolve.

One area that often comes to the fore in these situations is commitments made to customers.

- *Commitment*: What have we promised? This layer has the nature of Policy, in that it states goals that direct future operations, but it has the nature of Operations in that Commitments emerge and change as a part of ongoing business activity.

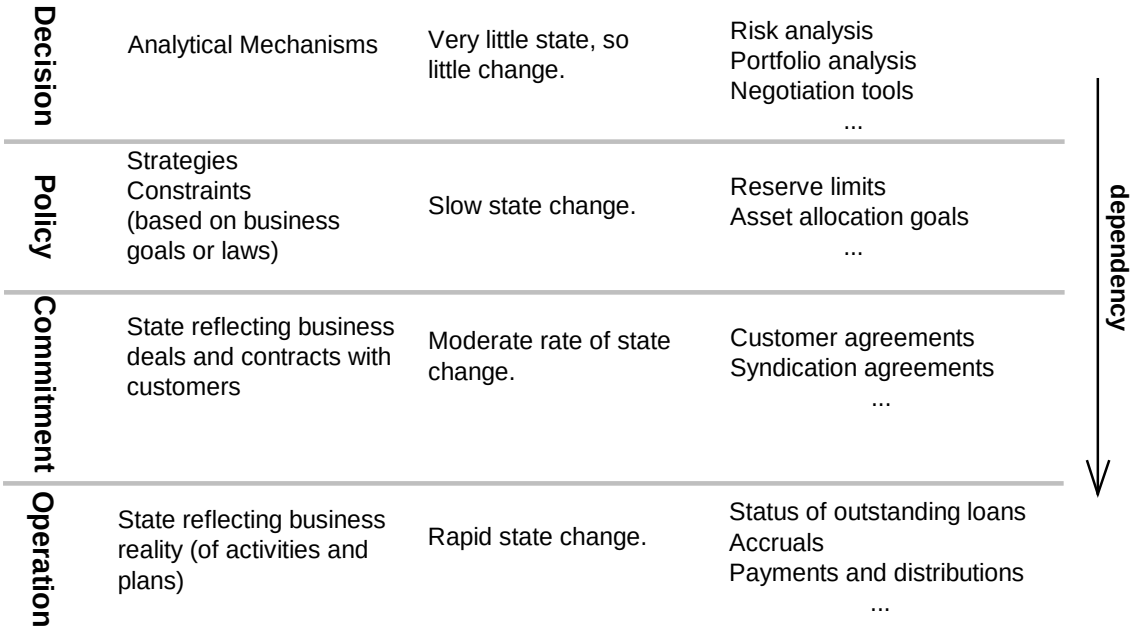
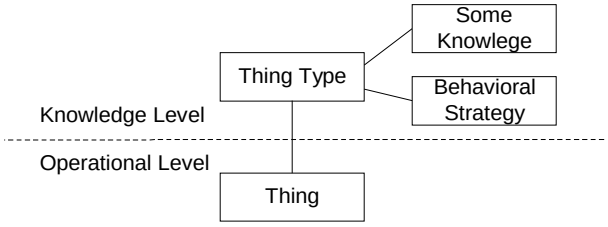


Figure 16. 13 Conceptual Dependencies and Shearing Points in an Investment Banking System

The Potential and Commitment layers are not mutually exclusive. A domain in which both were prominent, say a transportation company with a lot of custom shipping services, might use both. Other layers, more specific to those domains might be useful too. Change things. Experiment. But it is best to keep the layering system simple, and going beyond four or possibly five becomes unwieldy. It isn't as effective at telling the story, and the problems of complexity the large-scale structure was meant to solve will come back in a new form. The large-scale structure must be ferociously distilled.

While these five layers are applicable to a range of enterprise systems, they do not capture the salient responsibilities of all domains. In other cases, it would be counterproductive to try to force the design into this shape, but there may be a natural set of responsibility layers that do work. For a domain completely unrelated to those we've discussed, these might have to be completely original. Ultimately, you have to use your intuition, start somewhere, and let the ORDER EVOLVE.

KNOWLEDGE LEVEL



A group of objects that describe how another group of objects should behave. [www.martinfowler.com]

KNOWLEDGE LEVEL untangles things when we need to let some part of the model itself be plastic in the user's hands, yet constrained by a broader set of rules. It addresses requirements for software with configurable behavior, in which the roles and relationships among ENTITIES must be changed at installation or even at runtime.

In *Analysis Patterns*, [Fowler1996, pp 24-27], the pattern emerges from a discussion of modeling accountability within organizations, and is later applied to posting rules in accounting. Although it appears in several chapters, it doesn't have a chapter of its own. This is because it is a different sort of pattern than most in the book. Rather than modeling a domain, as the other analysis patterns do, it structures a model.

To see the problem concretely, consider models of "accountability". Organizations are made up of people, smaller organizations, the roles they play, and the relationships between them. The rules governing those roles and relationships vary greatly for different organizations. At one company, a "department" might be headed by a "Director" who reports to a "Vice President". In another company, a "module" is headed by a "Manager" who reports to a "Senior Manager". Then there are "matrix" organizations in which each person reports to different managers for different purposes.

A typical application would make some assumptions. When those didn't fit, users would start to use fields differently. Any behavior the application had would misfire, as the semantics were changed by the users. Users would develop workarounds for the behavior, or would get it shut off. They would learn complicated mappings between what they did in their jobs and the software features, and would never be served well.

When the system had to be changed or replaced, developers would discover, sooner or later) that the meaning of features were not what they seemed. They might mean very different things in different user communities or in different situations. Changing anything without breaking these overlaid usages would be daunting. Data migration to a more tailored system would require understanding and coding for all those quirks.

Example Part 1: Employee Payroll and Pension

The HR department of a medium sized company has a simple program for calculating payroll and pension contributions.

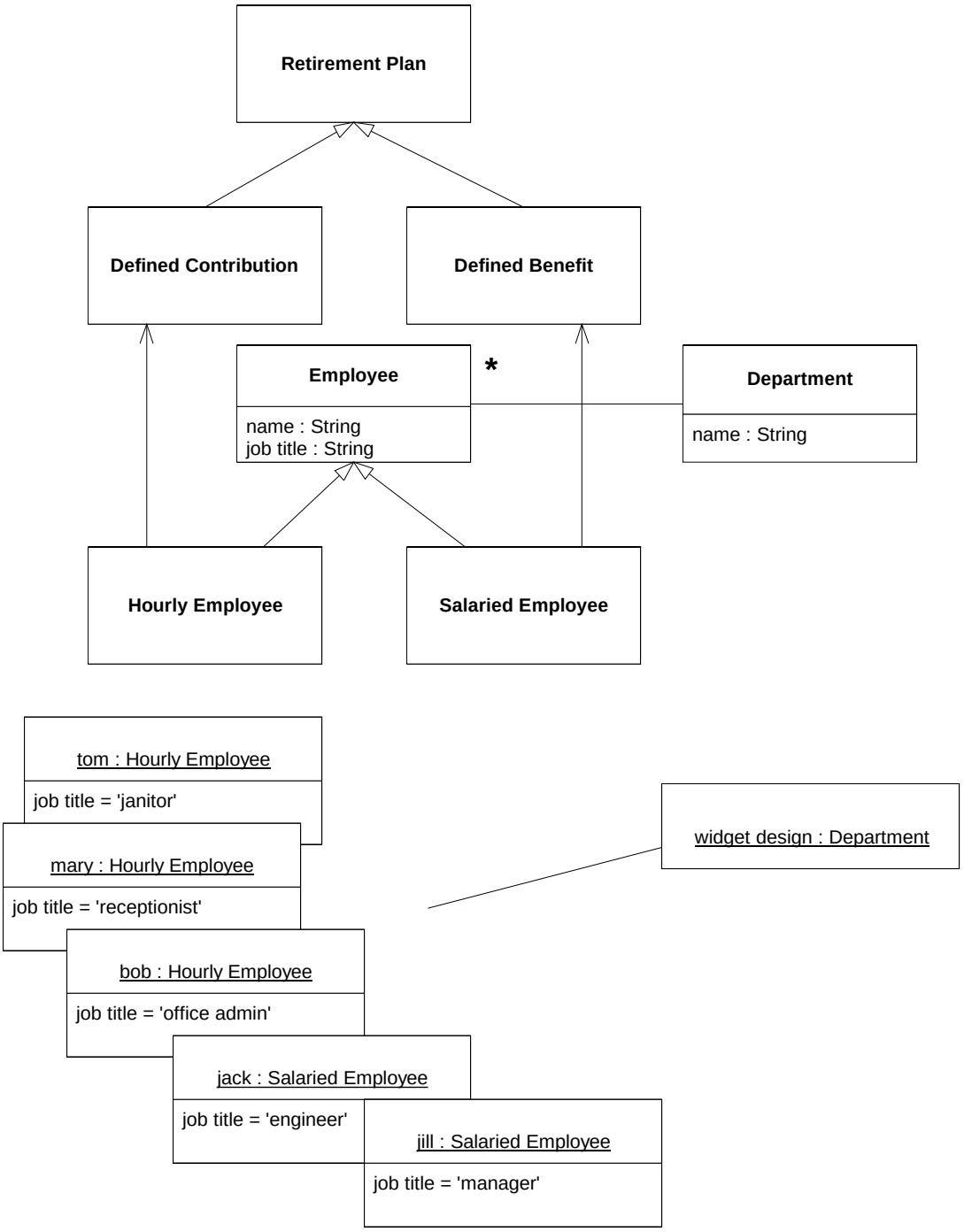


Figure 16. 14 Old model, over-constrained for new requirements

But now, the management has decided that the office administrators should go into the "defined benefit" retirement plan. The trouble is that office administrators are paid hourly, and this model does not allow mixing. The model will have to change.

The next model proposal is quite simple – just remove the constraints.

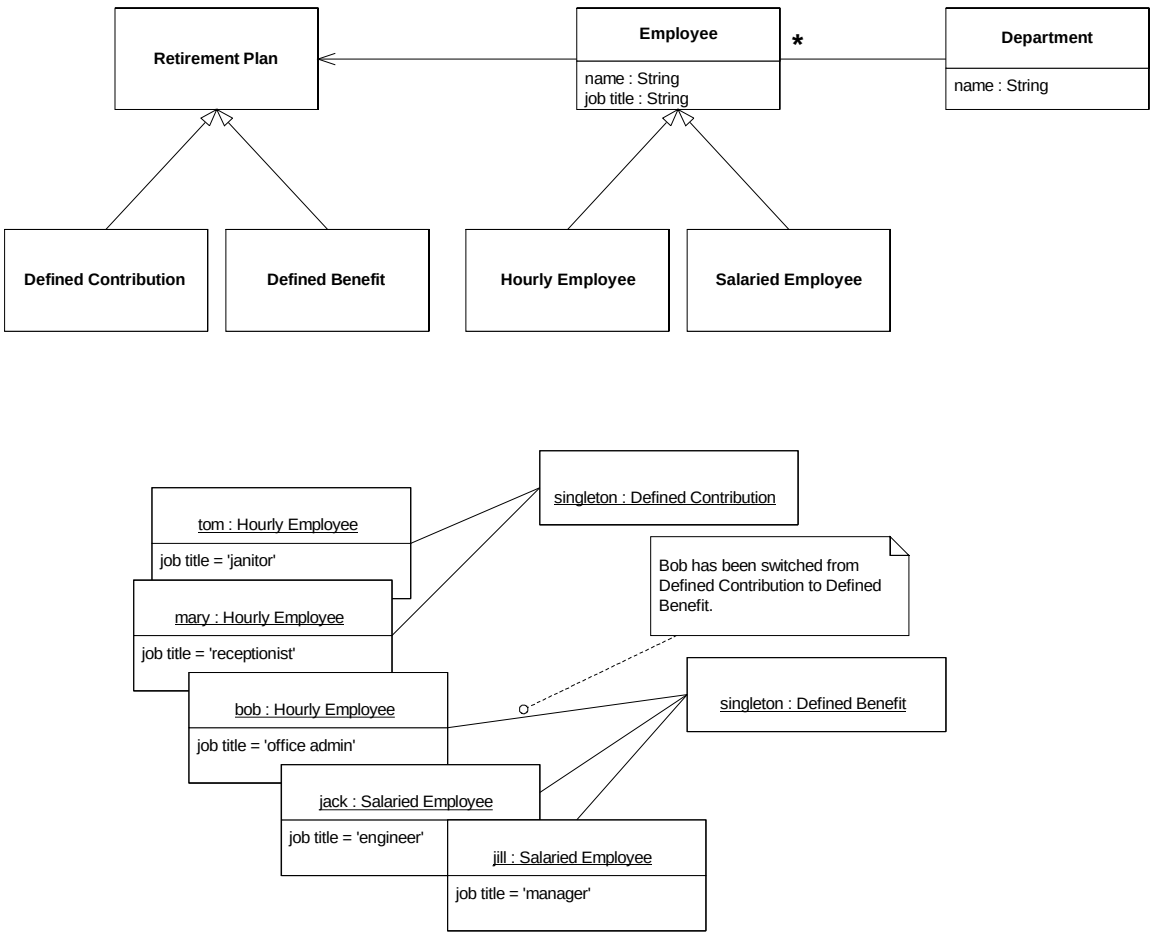


Figure 16. 15 Proposed model, now under-constrained

This allows each employee to be associated with either kind of retirement plan, so each office administrator could be switched. This model is rejected by management because it does not reflect company policy. Some administrators could be switched and others not. Or the janitor could be switched. They want a model that enforces the policy:

Office administrators are hourly employees with defined benefit retirement plans

This suggests that the “job title” field now represents an important domain concept. Refactoring to making that concept explicit as an “Employee Type”, produces

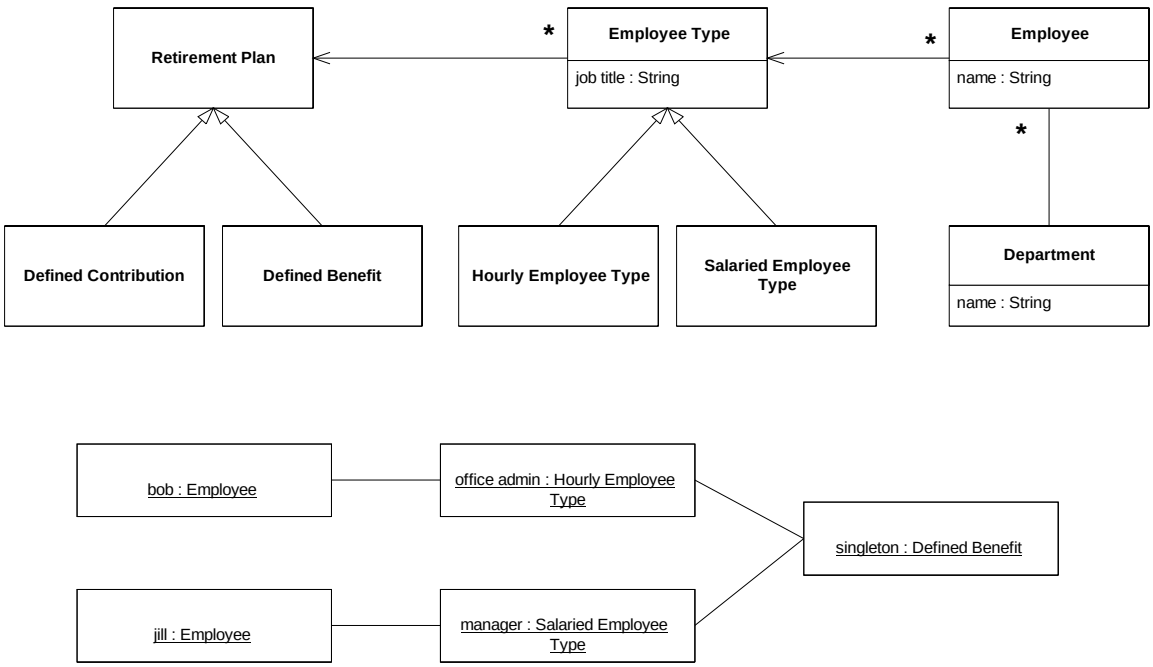


Figure 16. 16 Type object allows requirements to be met.

The requirements can be stated in the UBIQUITOUS LANGUAGE.

An Employee Type is assigned to either Retirement Plan or either payroll.

Employees are constrained by the Employee Type.

Access to edit the Employee Type object will be restricted to a "super user" who will only make changes when company policy changed. An ordinary user in the personnel department could change Employees or point them at a different Employee Type.

This model satisfies the requirements, but the developers sense an implicit concept or two. But it is just a nagging feeling at the moment. They don't have any ideas for what to do about it, so they call it a day.

A static model can cause problems. But problems could be just as bad with a fully flexible system that allowed any possible relationship to be presented. Such a system would be inconvenient to use and wouldn't allow the organization's own rules to be enforced.

Fully customizing software for each organization is not practical because, even if each organization could pay for custom software, the organizational structure will likely change frequently.

So such software must provide options to allow the *user* to configure it to reflect the current structure of the organization. The trouble is that adding such options to the model objects makes them unwieldy. The more flexibility you add, the more complex it all becomes.

In an application in which the roles and relationships between ENTITIES varies in different situations, complexity can explode. Neither fully general models nor highly customized ones serves the user's needs. Objects end up with references to other types to cover a variety of cases, or with attributes that are used in different ways in different situations. Or you may end up with many classes that have the same data and behavior, but just have different assembly rules.

Nestled into our model is another model that is *about* our model. A KNOWLEDGE LEVEL separates that self-defining aspect of the model and makes its constraints explicit.

KNOWLEDGE LEVEL is an application to the domain layer of the REFLECTION pattern, used in many software architectures and technical infrastructures and described well in [BMRSS1996]. REFLECTION accommodates changing needs by making the software "self-aware", and making selected aspects of its structure and behavior accessible for adaptation and change. This is done by splitting the software into a base level, which carries the operational responsibility for the application, and a "meta level", which represents knowledge of the structure and behavior of the software.

Java has some minimal built-in REFLECTION in the form of protocols for interrogating a class for its methods and so forth. These mechanisms allow a program to ask questions about its own design. CORBA has somewhat more extensive but similar REFLECTION protocols. Some persistence technologies extend the richness of that self-description to support partially automated mapping between database tables and objects. There are other technical examples. Fowler shows that this pattern can also be applied within the application domain.

The KNOWLEDGE LEVEL provides two useful distinctions. First, it focuses on the application domain, in contrast to familiar applications of REFLECTION. Second, it does not strive for full generality. Just as a SPECIFICATION can be more useful than a general predicate, a very specialized set of constraints on a set of objects and their relationships can be more useful than a generalized framework. It is simpler and can communicate the specific intent of the designer.

Fowler terminology versus POSA terminology

Knowledge Level	Meta Level
Operations Level	Base Level

Just to be clear, the reflection tools of the programming language are *not* for use in implementing the KNOWLEDGE LEVEL of a domain model. Those meta-objects describe the structure and behavior of the language constructs themselves. Instead, the KNOWLEDGE LEVEL must be built of ordinary objects.

Interestingly, the pattern is not called a knowledge "layer". As much as it resembles layering, reflection involves mutual dependencies running in both directions.

Create a distinct set of objects that can be used to describe and constrain the structure and behavior of the basic model. Keep these concerns distinct as two "levels", one very concrete, the other reflecting rules and knowledge that a user or super-user is able to customize.

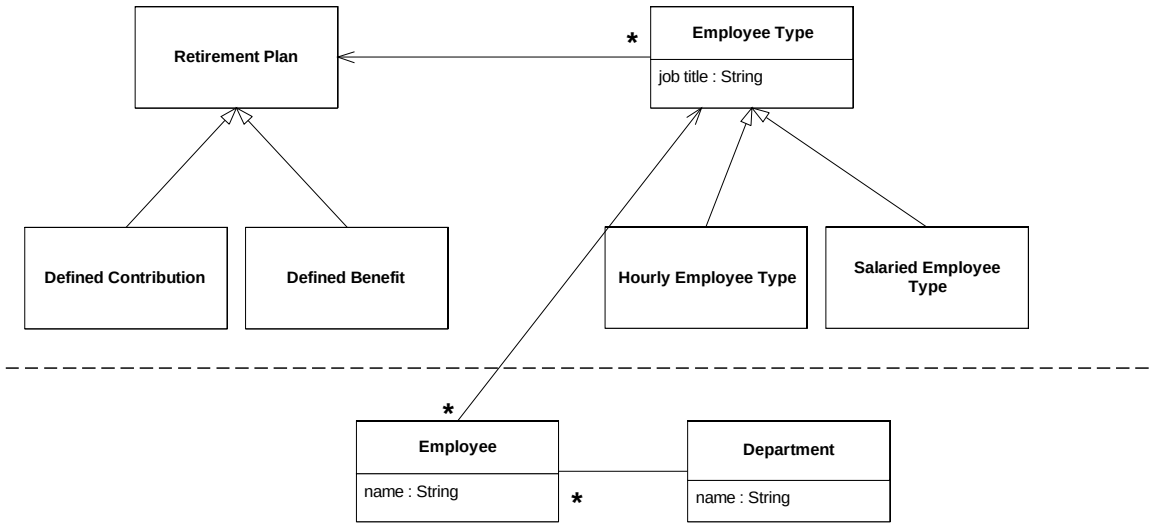
Like all powerful ideas, REFLECTION and KNOWLEDGE LEVELS can be intoxicating. This pattern should be used sparingly. It can unravel complexity by freeing operations objects from the need to be jacks-of-all-trades, but the indirection it introduces does add some of that obscurity back in. If the KNOWLEDGE LEVEL becomes complex, the system's behavior becomes hard to understand for developers and users alike. The users (or super-user) who configure it will end up needing the skills of a programmer, and a meta-level programmer at that. If they make mistakes, the application will behave incorrectly.

Also, the basic problems of data migration don't completely disappear. When a structure in the KNOWLEDGE LEVEL is changed, existing operations level objects have to be dealt with. It may be possible for old and new to coexist, but one way or another, careful analysis is needed.

All of these issues put a major burden on the designer of a KNOWLEDGE LEVEL. The design has to be robust enough not only to handle scenarios presented in development, but any scenario a user could configure the software for in the future. Applied judiciously, to the points where customization is crucial and would otherwise distort the design, it can solve problems that are very hard to handle any other way.

Example Continued: Payroll and Pension KNOWLEDGE LEVEL

Our team is back, and, refreshed from a night's sleep, one of them has started to close in on one of the awkward points. Why were certain objects being secured while others were freely edited? The cluster of restricted objects reminded him of the KNOWLEDGE LEVEL pattern, and he decided to try it as a way of viewing the model. He found that the existing model could already be viewed this way.



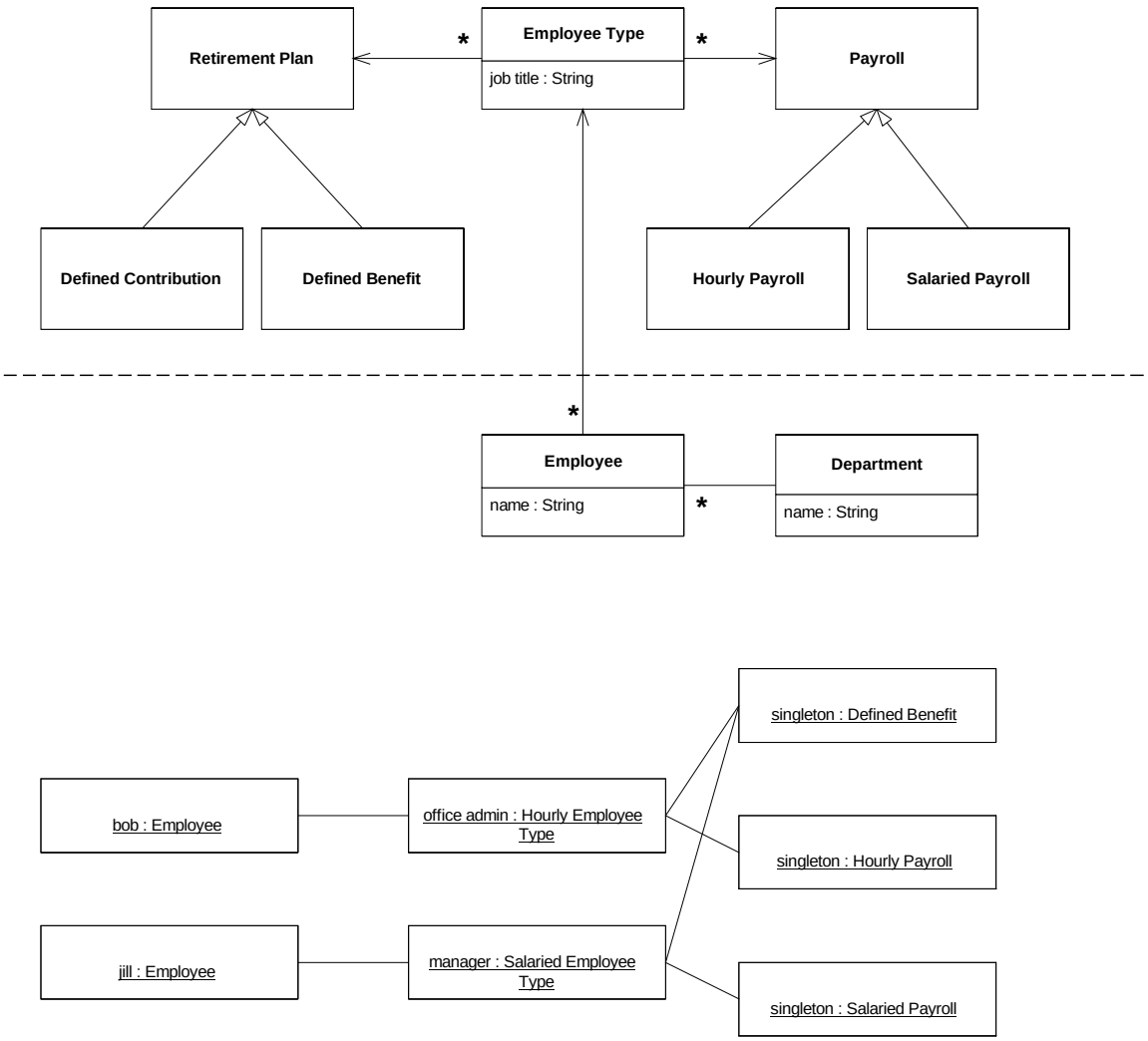
The restricted edits were in the KNOWLEDGE LEVEL, while the day-to-day edits were in the operational level. A nice fit. All the objects above the line described types or long standing policies. The Employee Type effectively imposed behavior on the Employee.

He was sharing his insight around with his colleagues, when one of the other developers had another insight. The clarity of seeing the model organized by knowledge level had let her spot what had been bothering *her* the previous day. Two distinct concepts were being combined in the same object. She had heard it in the language the previous day, but hadn't put her finger on it:

> An Employee Type is assigned to either Retirement Plan or either payroll.

But that was *not* really a statement in the UBIQUITOUS LANGUAGE. There was no "payroll" object in the model. They had spoken in the LANGUAGE they wanted, rather than the one they had. The concept of Employee Type was lumped together arbitrarily with payroll. It hadn't been so obvious before the KNOWLEDGE LEVEL was separated out, and the very elements in that key phrase all appeared in the same level together…except one.

Based on this insight, she refactored again to this model, which does support that statement.



The need for user control of the rules for associating objects drove the team to a model that had an implicit KNOWLEDGE LEVEL. KNOWLEDGE LEVEL was hinted at by the characteristic access restrictions and a thing-thing type relationship. Once it was in place, the clarity it afforded helped produce another insight that disentangled two important domain concepts.

KNOWLEDGE LEVEL, like other large-scale structures, isn't strictly necessary. The objects will still work without it, and the insight that separated Employee Type from Payroll will still be in place. There may come a time when it doesn't seem to be pulling its weight an can be dropped. But for now, it seems to tell a useful story about the system and help developers grapple with the model.

♦ ♦ ♦

At first glance, KNOWLEDGE LEVEL looks like a special case of RESPONSIBILITY LAYERS, especially the "policy" layer, but it is not. For one thing, dependencies run in both directions between the levels, while lower layers are independent of their upper layers. In fact, KNOWLEDGE LEVEL can coexist with most other large-scale structures, providing an additional dimension of organization.

PLUGGABLE COMPONENT FRAMEWORK

Opportunities arise in a very mature model that is deep and distilled. A PLUGGABLE COMPONENT FRAMEWORK usually comes into play when a few applications have already been implemented in the same domain.

♦ ♦ ♦

When a variety of applications have to interoperate, all based on the same abstractions but designed independently, translations between multiple BOUNDED CONTEXTS limit integration. A shared kernel is not be feasible for teams that do not work closely together. Duplication and fragmentation raise costs of development and installation, and interoperability becomes very difficult.

Some successful projects break down their design into components, each with responsibility for certain categories of functions. Usually there is a central hub that all the components plug into, which supports any protocols they need and knows how to talk to the interfaces they provide, although other patterns of connecting components are also possible. The design of these interfaces and the hub that connects them must be coordinated, while more independence is possible designing the interiors.

Several widely used technical frameworks support this pattern, but that is a secondary issue. The basic pattern is a conceptual organization of responsibilities. It can easily be applied within a single Java program. A technical framework is only needed if it solves some essential technical problem like distribution, or sharing a component among different applications.

Distill an ABSTRACT CORE of interfaces and interactions and create a framework that allows diverse implementations of those interfaces to be freely substituted. Likewise, allow any application to use those components, so long as it operates strictly through the interfaces of the ABSTRACT CORE.

High-level abstractions are identified and shared across a breadth of the system while specialization occurs in modules. The central hub of the application is an ABSTRACT CORE within a SHARED KERNEL. But multiple BOUNDED CONTEXTS can lie behind the encapsulated component interfaces, so that this structure can be especially convenient when much components are coming from many different sources, or preexisting software that is being integrated.

Not to say components must have divergent models. Multiple components can be developed within a single CONTEXT if the teams CONTINUOUSLY INTEGRATE, or they can define another SHARED KERNEL held in common by a closely related set of components. All these strategies can coexist easily within a large-scale structure of PLUGGABLE COMPONENTS.

There are a few downsides to a PLUGGABLE COMPONENT FRAMEWORK. The first is that this is a very difficult pattern to apply. It requires precision in the design of the interfaces and a deep enough model to capture the necessary behavior in the ABSTRACT CORE. The other major downside is that applications have limited options. If an application needs a very different approach to the domain, they can't easily do that. They can specialize the model, but they can't change it without changing the protocol of all the diverse components. As a result, the process of continuous refinement, refactoring toward deeper insight, is more or less frozen in its tracks.

[FJ2000] gives a good glimpse of ambitious attempts at PLUGGABLE COMPONENT FRAMEWORKS in several domains, including a discussion of SEMATECH CIM. The success of such frameworks is a mixed story. Probably the biggest obstacle is the maturity of understanding needed to design a useful framework. A PLUGGABLE COMPONENT FRAMEWORK should not be the first large-scale structure applied on a project; nor the second. The most successful examples have followed after the full development of multiple specialized applications.

Example: SEMATECH CIM Framework

In a factory producing computer chips, groups (called "lots") of silicon wafers are moved from one machine to another through hundreds of steps of processing until the microscopic

circuitry being printed and etched into them is complete. Software is needed that can track each individual lot, recording the exact processing that has been done to it, and direct either factory workers or automated equipment to take it to the next appropriate machine and apply the next appropriate process. Such software is called a “manufacturing execution system” (MES).

Hundreds of different machines from dozens of vendors are used, with carefully tailored recipes at each step of the way. Developing MES software that could deal with such a complex mix was daunting and prohibitively expensive. In response, an industry consortium, SEMATECH, developed the CIM Framework.

The CIM Framework is big and complicated and has many aspects, but two aspects are relevant here. First, the framework defines abstract interfaces for the basic concepts of the semiconductor MES domain, in other words, the CORE DOMAIN in the form of an ABSTRACT CORE. These interface definitions include behavior and semantics.

A highly simplified subset of the CIM interfaces

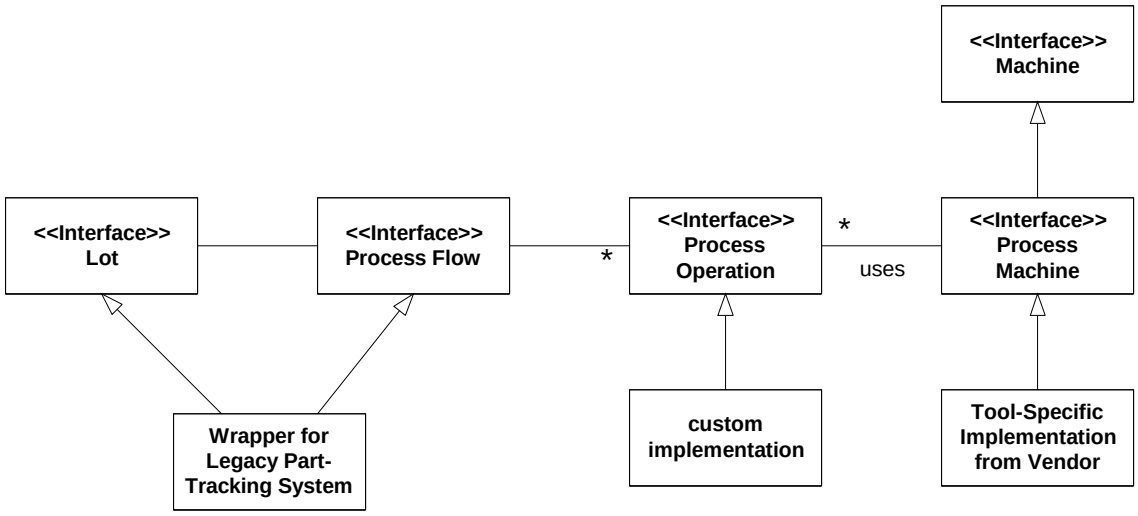


Figure 16. 17

If a vendor produced a new machine, they would have to develop a specialized implementation of the Process Machine interface. If they adhered to that interface, their machine control component should plug into any application based on the CIM Framework.

Having defined these interfaces, they defined the rules by which they could interact in an application. Any application based on the CIM Framework would have to implement a protocol that hosted objects implementing some subset of those interfaces. If this protocol were implemented, and the application strictly observed the abstract interfaces, then the application could count on the promised services of those interfaces regardless of implementation. The combination of those interfaces and the protocol for using them constitutes a tightly restrictive large-scale structure.

User places lot in next machine and logs move into computer

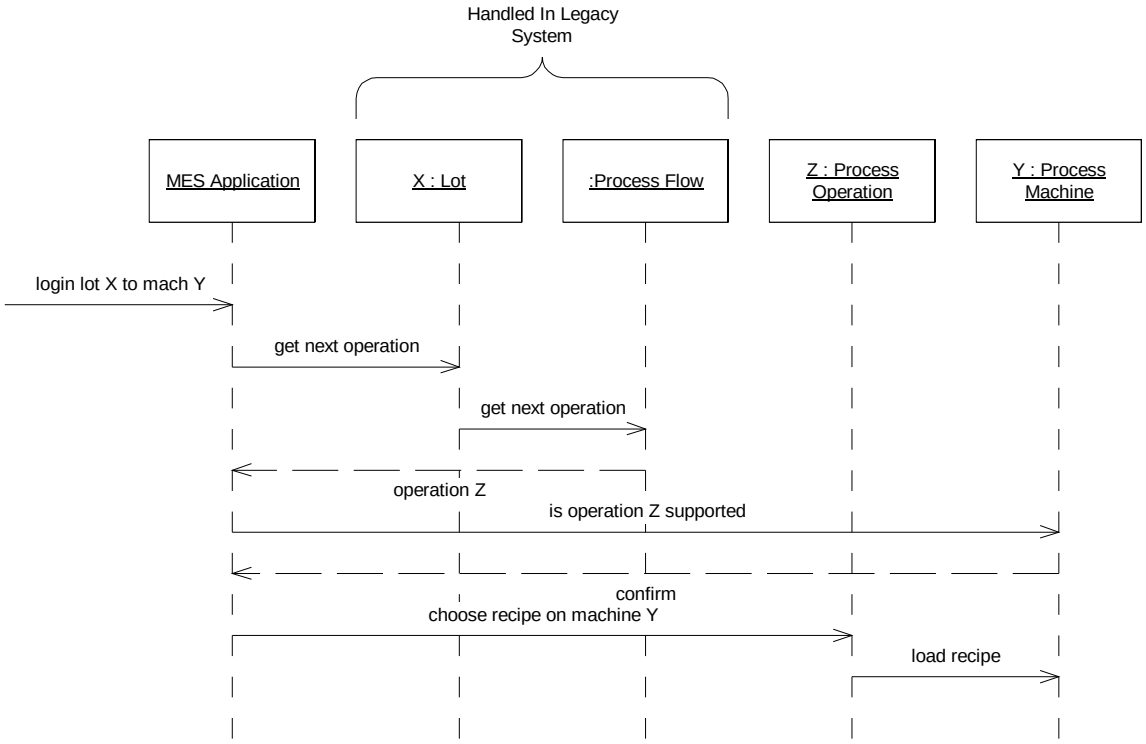


Figure 16. 18

The framework has very specific infrastructure requirements. It is tightly coupled to CORBA to provide persistence, transactions, events and other technical services. But the interesting thing about it is the definition of a PLUGGABLE COMPONENT FRAMEWORK that allows people to develop parts of these immense systems independently and smoothly integrate them into immense systems that no one knows all the details of, but everyone understands an overview of.

♦ ♦ ♦

The plug-in interface of hub might use a PUBLISHED LANGUAGE.

<<Side Bar>>

How can thousands of people work independently to create a quilt of more than 40,000 panels? A few simple rules provide a **large-scale structure** for the AIDS Memorial Quilt, leaving the details to individual contributors.

Here's how to create a panel for the Quilt:

Design the panel

Include the name of the person you are remembering. Feel free to include additional information such as the dates of birth and death, and a hometown. Please limit each panel to one individual.

Choose your materials

Remember that the Quilt is folded and unfolded many times, so durability is crucial. Since glue deteriorates with time, it is best to sew things to the panel. A medium-weight, non-stretch fabric such as a cotton duck or poplin works best.

Your design can be vertical or horizontal, but the finished, hemmed panel must be 3 feet by 6 feet (90 cm x 180 cm)--no more and no less! When you cut the fabric, leave an extra 2-3 inches on each side for a hem. If you can't hem it yourself, we'll do it for you. Batting for the panels is not necessary, but backing is recommended. Backing helps to keep panels clean when they are laid out on the ground. It also helps retain the shape of the fabric.

Create the panel

To construct your panel you might want to use some of the following techniques:

Appliqué: Sew fabric, letters and small mementos onto the background fabric. Do not rely on glue–it won't last.

Paint: Brush on textile paint or color-fast dye, or use an indelible ink pen. Please don't use "puffy" paint; it's too sticky.

Stencil: Trace your design onto the fabric with a pencil, lift the stencil, then use a brush to apply textile paint or indelible markers.

Collage: Make sure that whatever materials you add to the panel won't tear the fabric (avoid glass and sequins for this reason), and be sure to avoid very bulky objects.

Photos: The best way to include photos or letters is to photocopy them onto iron-on transfers, iron them onto 100% cotton fabric and sew that fabric to the panel. You may also put the photo in clear plastic vinyl and sew it to the panel (off-center so it avoids the fold).

--AIDS Memorial Quilt Project, www.aidsquilt.org

<<END Sidebar>>

How Restrictive Should a Structure Be?

The large-scale structure patterns discussed in this chapter range from the very loose SYSTEM METAPHOR to the restrictive PLUGGABLE COMPONENT FRAMEWORK. Other structures are possible, of course, and even within a general structural pattern, there is a lot of choice about how restrictive to make the rules.

For example, RESPONSIBILITY LAYERS dictate a kind of factoring of model concepts and their dependencies, but you could add rules that would specify communication patterns between the layers.

Consider a factory where software directs each part to a machine where it is processed according to some recipe. The correct process was ordered from a Policy layer and executed in an Operations layer. But inevitably there will be mistakes made on the factory floor. The actual situation will not be consistent with the rules of the software. Now, an Operations layer *must reflect the world as it is*, which means that when a part is occasionally put in the wrong machine, that information must be accepted unconditionally. Somehow, this exceptional condition needs to be communicated to a higher layer, a decision making layer that can use other policies to correct the situation, perhaps by rerouting the part to a repair process or by scrapping it. But Operations does not know anything about higher layers. The communication has to be done in a way that doesn't create two-way dependencies the lower layers to the higher ones.

Typically, this would be done through some kind of event mechanism. The Operations objects generate events whenever their state changes. Policy layer objects listen for events of interest from the lower layers. When an event occurs that violates a rule, the rule executes an action (which is part of the rule's definition) that makes the appropriate response, or it might generate an event for the benefit of some still higher layer.

In the banking example above, the value of assets change (operations), shifting the value of segments of our portfolio. When these exceed portfolio allocation limits (policy), perhaps a trader is alerted, who can buy or sell assets to redress the balance.

We could figure this out on a case-by-case basis, or we could decide on a consistent pattern for everyone to follow in interactions of objects of particular layers. A more restrictive structure increases uniformity,

making the design easier to interpret. If the structure fits, the rules are likely to push developers toward good designs. Disparate pieces are likely to fit together better.

On the other hand, the restrictions may take away flexibility developers need. Very particular communication paths might be impractical to apply across BOUNDED CONTEXTS, especially in different implementation technologies, in a heterogeneous system.

So, you have to fight the temptation to build frameworks and regiment the implementation of the large-scale structure. The most important contribution of the large-scale structure is conceptual coherence, and giving insight into the domain. Each structural rule should *make development easier*.

Refactoring Toward a Fitting Structure

In an era when the industry is shaking off excessive up-front design, some will see large-scale structure as a throwback to the bad old days of waterfall architecture. But, in fact, the only way a useful structure can be found is from a very deep understanding of the domain and the problem, and the practical way to that understanding is an iterative development process.

A team committed to EVOLVING ORDER must fearlessly rethink the large-scale structure throughout the project lifecycle. The team should not saddle itself with a structure conceived of early on when no one understands the domain or the requirements very well.

Unfortunately, that means that your final structure will not be available at the start, and that means that you will have to refactor to impose it as you go along. This can be expensive and difficult, but it is *necessary*. There are some general ways of controlling the cost and maximizing the gain.

Minimalism

One key to keeping the cost down is keep the structure simple and lightweight. Don't attempt to be comprehensive. Just address the most serious concerns and leave the rest to be handled on a case-by-case basis.

Early on, it can be helpful to choose a loose structure, such as a SYSTEM METAPHOR or RESPONSIBILITY LAYERS. A minimal, weak structure can provide lightweight guidelines that will nonetheless help prevent chaos.

Communication and Self-Discipline

First, it is necessary to follow the structure in new development and refactoring. **To do this, the structure must be understood by the entire team. The terminology and relationships must enter the UBIQUITOUS LANGUAGE.**

Large-scale structure can provide a vocabulary for the project to deal with the system broadly, and for different people to independently make harmonious decisions. But since most large-scale structures are loose conceptual guidelines, the teams must exercise self-discipline.

Without consistent adherence by the many people involved, structures have a tendency to decay. The relationship of the structure to detailed parts of the model or implementation is not usually explicit in the code, and functional tests do not rely on the structure. Plus, the structure tends to be abstract, so that consistency of application can be difficult to maintain across a large team (or multiple teams).

The kind of conversations that take place on most teams are not enough to maintain a consistent large-scale structure in a system. It is critical to incorporate it into the UBIQUITOUS LANGUAGE of the project, and for everyone to exercise that language relentlessly.

Restructuring Yields Supple Design

Second, any change to the structure may lead to a lot of refactoring. Since the structure is evolving (as system complexity increases and understanding deepens) **the entire system has to be changed to adhere to the new structure as it changes**. Obviously that is a lot of work, as the system grows.

One hopeful discovery that I've made is that a design with a large-scale structure is usually much easier to transform than one without. This seems to be true even when changing from one kind of structure to another, say METAPHOR to LAYERS. I can't entirely explain this. Part of the answer is that it is easier to rearrange something when you can understand its current arrangement, and the preexisting structure makes that easier. Partly it is that the discipline that it took to maintain the earlier structure filters through all aspects of the system. But there is something more, I think, because it is *even easier* to change a system that has had *two* previous structures.

A new leather jacket is stiff and uncomfortable, but after the first day of wear the elbows have been bent a few times and are easier to bend. After a few more wearings, the shoulders have loosened up, and the jacket is easier to put on. After months of wear, the leather becomes supple and is comfortable and easy to move in. So it seems to be with models that are transformed repeatedly with conceptually sound transformations. Ever increasing knowledge is embedded into them and **the principle axes of change have been identified and made flexible**, while stable aspects have been simplified.

Distillation Lightens the Load

Another crucial force that should be applied to the model is continuous distillation. This reduces the difficulty of changing the structure in various ways. First, by removing mechanisms, GENERIC SUBDOMAINS and other support structure from the CORE DOMAIN, there may simply be less to restructure.

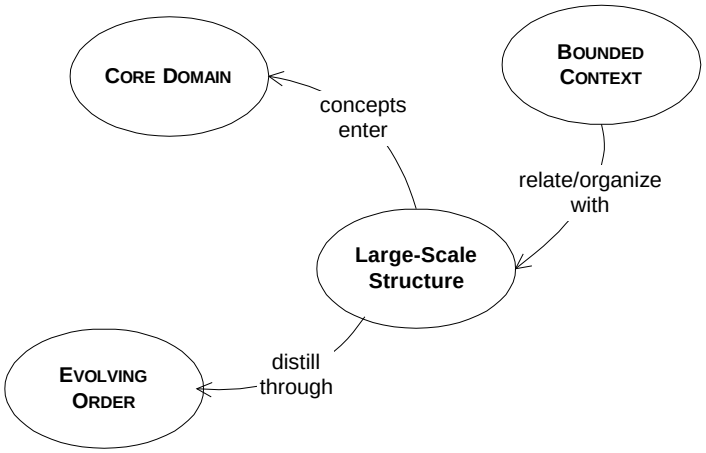
If possible, these supporting elements should be defined to fit into the large-scale structure in a simple way. For example, in a system of RESPONSIBILITY LAYERS, a GENERIC SUBDOMAIN could be defined that would fit within a single layer. With PLUGGABLE COMPONENTS, it could be a component, be owned entirely by a single component, or be a SHARED KERNEL between a set of related components. This means that these supporting elements may also have to be refactored to find their place in the structure, but they move independently of the CORE DOMAIN, and tend to be more narrowly focused. And ultimately they are less critical.

The principles of distillation and refactoring toward deeper insight apply even to the large-scale structure itself. For example, the layers may be initially chosen based on a superficial understanding of the domain, and are gradually replaced with deeper abstractions that express the fundamental responsibilities of the system. This sharp-edged clarity that lets people easily see deep into the design is the goal, and it also is part of the means, since it makes manipulation of the system on a large-scale easier and safer.

17.Bringing the Strategy Together

The preceding 3 presented a lot of principles and techniques for domain-driven strategic design. Now it is time to bring them all together. How does a large-scale structure coexist with a CONTEXT MAP? Where do the building blocks fit in? You have to devise a single strategic design for a project, probably applying several of the patterns. What do you do first? Second? Third? How do you go about devising your strategy?

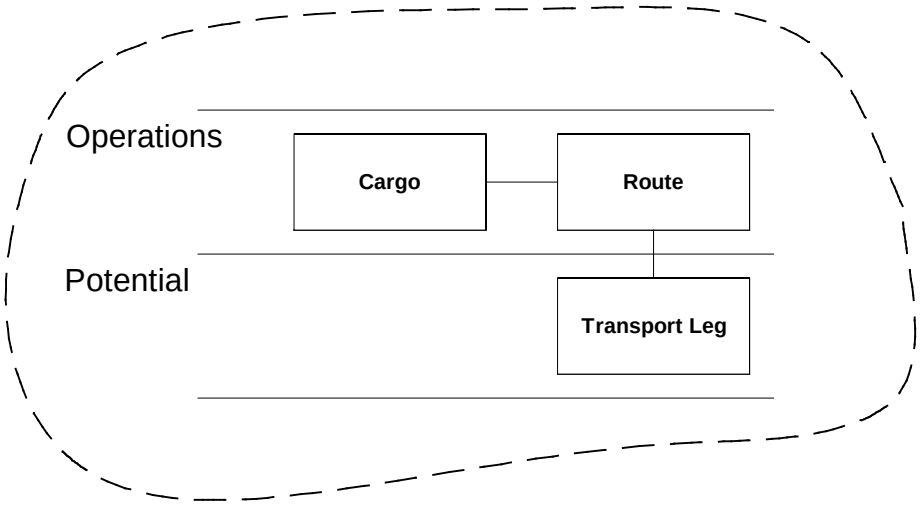
Bringing Together Large-Scale Structures and BOUNDED CONTEXTS



The three basic principles of strategic design (context, distillation, and large-scale structure) are not substitutes for each other, they are complementary and can interact in many ways. For example, a large-scale structure can exist within one BOUNDED CONTEXTS or it can cut across them and organize the CONTEXT MAP.

The previous examples of RESPONSIBILITY LAYERS were confined to one BOUNDED CONTEXT. This is easiest way to explain the idea, and a common use of the pattern. In this scenario, the meaning of layer names is restricted to that CONTEXT, just as the names of model elements or subsystem interfaces that exist within that CONTEXT.

Structuring an OBJECT MODEL within a single BOUNDED CONTEXT

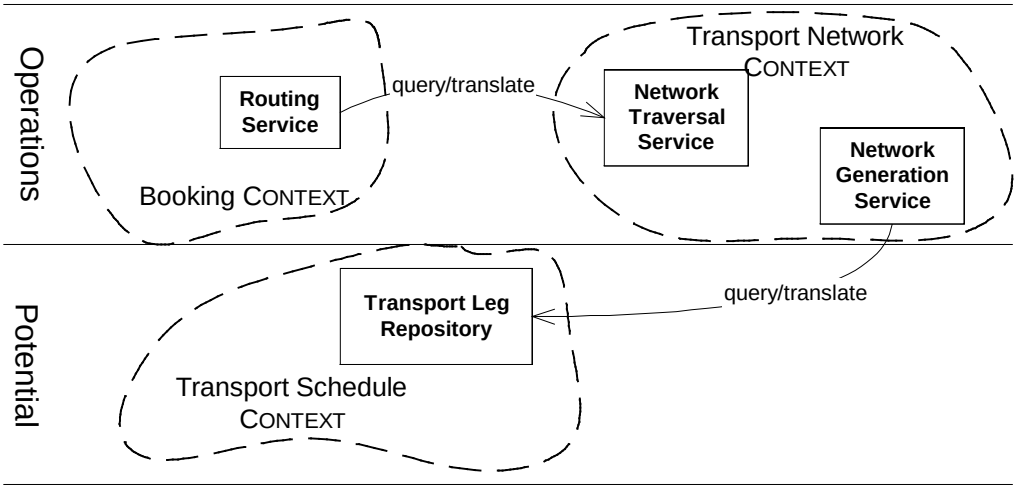


17. 1

This can be valuable in a very complicated model, raising the complexity ceiling of how much can be maintained in a single, unified BOUNDED CONTEXT.

But on many projects the greater challenge is understanding how disparate parts fit together. They may be partitioned into separate contexts, but what part does each play in the whole integrated system and how do they relate to each other? Then the large-scale structure can be used to organize the CONTEXT MAP. In this case, the terminology of the structure applies to the whole project (or at least some clearly bounded part of it).

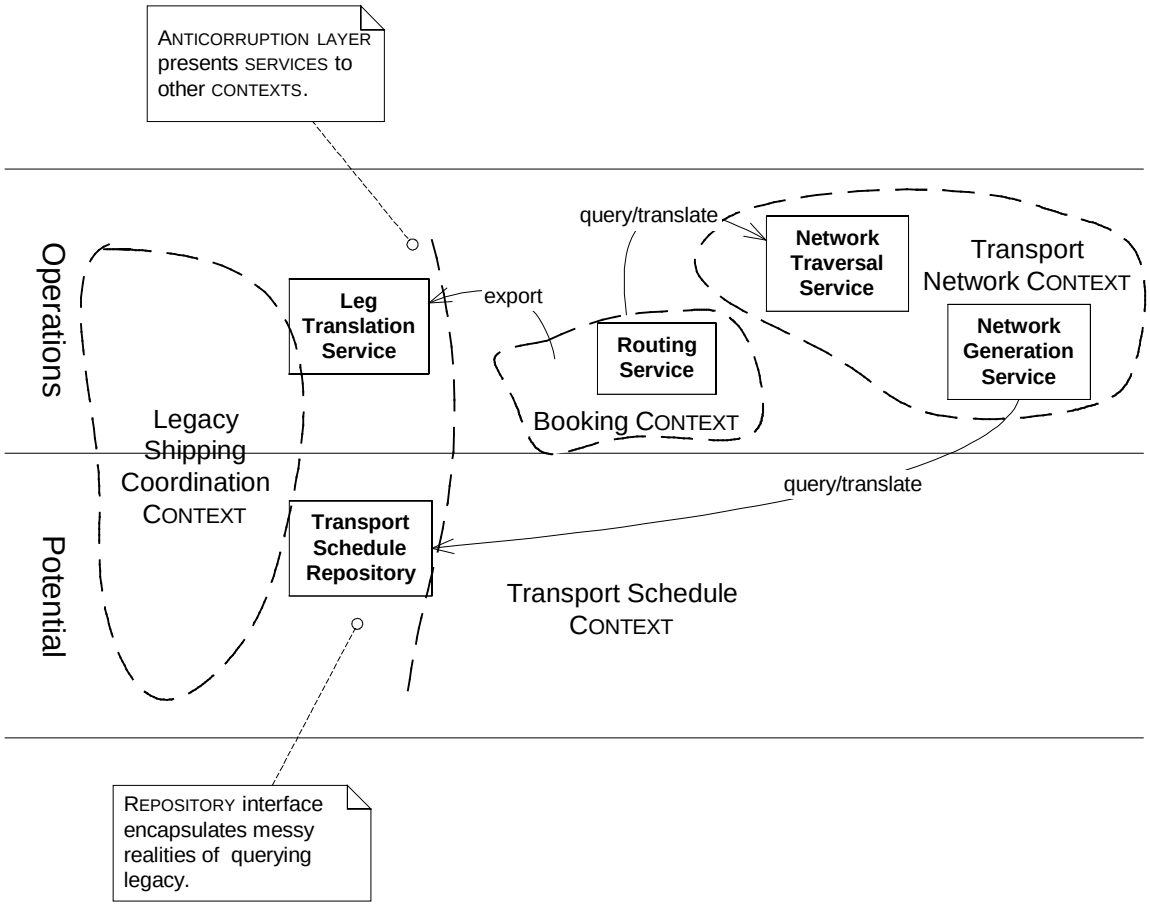
Structure imposed on the relationships of components with distinct BOUNDED CONTEXTS



17. 2

Suppose you want to adopt RESPONSIBILITY LAYERS, but you have a legacy system that wasn't organized in a way consistent with your desired large-scale structure. Do you have to give up your LAYERS? No, but you have to acknowledge the actual place the legacy has within the structure. In fact, it may help to characterize the legacy. The SERVICES the legacy provides may in fact be confined to only a few LAYERS. To be able to say that the legacy system fits within particular responsibility layers concisely describes a key aspect of its scope and role.

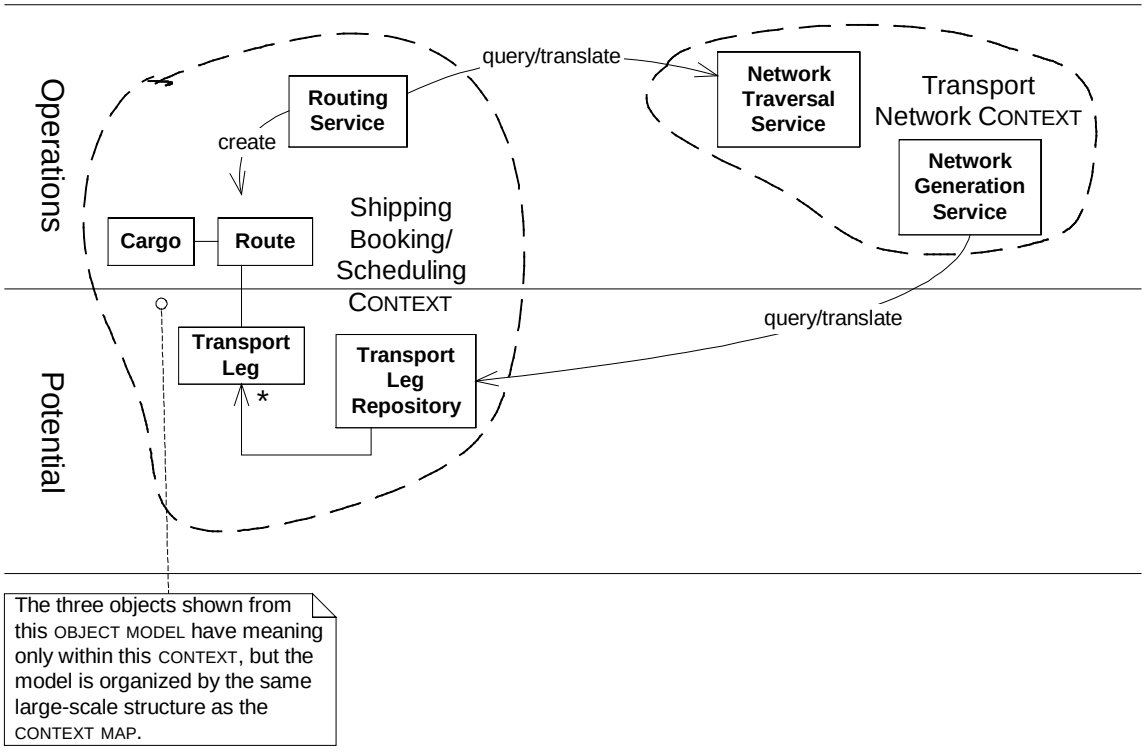
Structure that allows some components to span layers



17. 3

If the legacy subsystem's capabilities are being accessed through a FAÇADE, you may be able to design each SERVICE offered by the FAÇADE to fit within one layer.

The interior of the Shipping Coordination application, being a legacy in this example, is a presented as an undifferentiated mass. But on a team with a well-established large-scale structure could choose to build in a CONTEXT that spans multiple LAYERS using the same structure internally to structure the OBJECT MODEL.



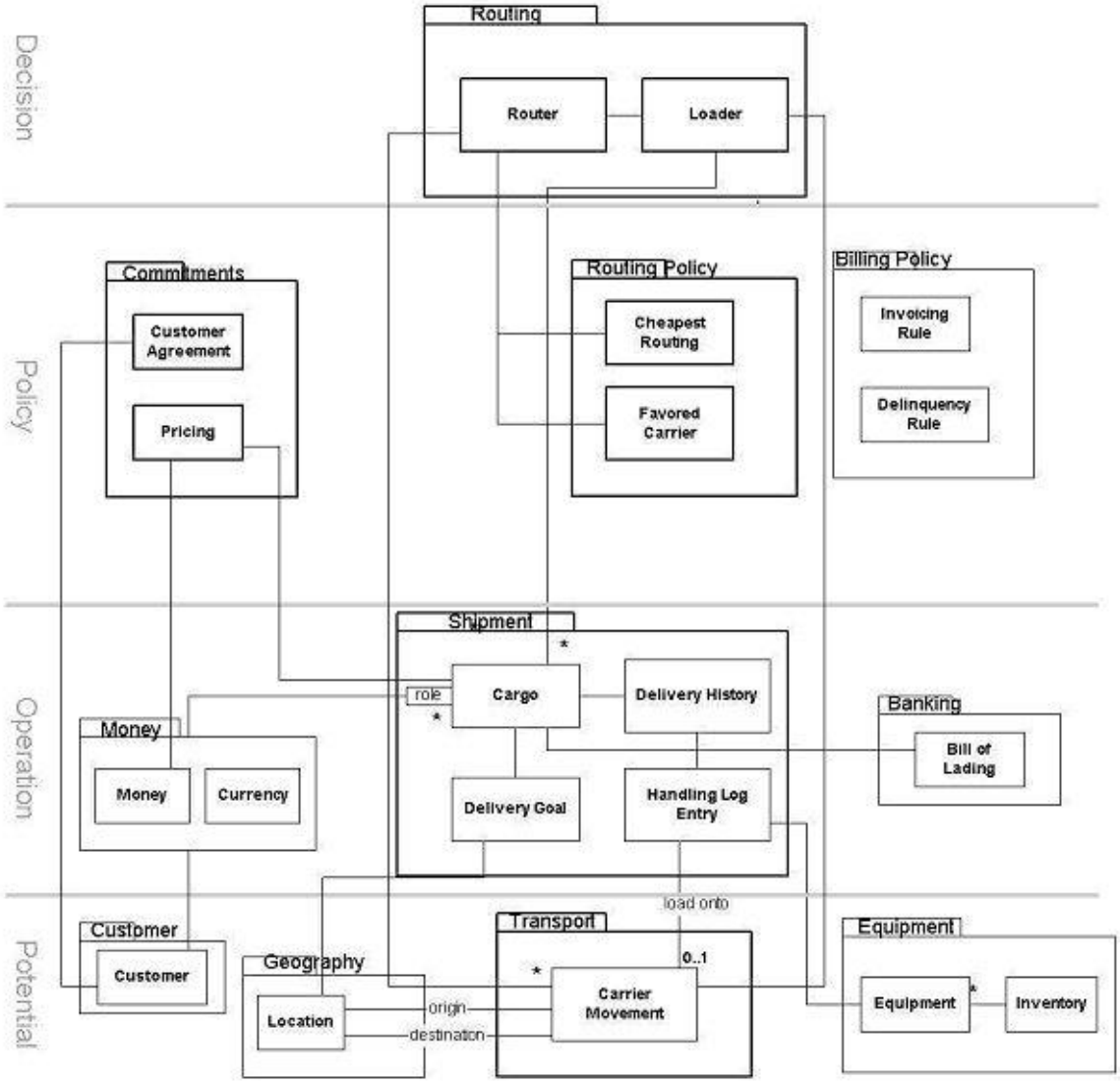
17. 4

Of course, since each BOUNDED CONTEXT is its own name-space, a structure could be used to organize the OBJECT MODEL within one CONTEXT, while another was used a neighboring CONTEXT and still another was used to organize the CONTEXT MAP. However, going too far down that path can erode the value of the large-scale structure as a unifying set of concepts for the project.

Bringing Together Large-Scale Structures and Distillation

Large-scale structure and distillation also complement each other. The large-scale structure can help explain the relationships within the CORE DOMAIN and between generic subdomains.

Modules of the CORE DOMAIN (in bold) and GENERIC SUBDOMAINS are clarified by the layers



17. 5

At the same time, **the large-scale structure itself may be an important part of the CORE DOMAIN**. For example, the importance of distinguishing the layering of potential, operations, policy and decision support distills an important insight very fundamental to the business problem addressed by the software. This is especially useful if a project is chopped up into many BOUNDED CONTEXTS, so that the objects of the CORE DOMAIN don't have meaning over much of the project.

Assessment First

When you are tackling the strategic design for the first time on a project, where do you start? Strategically, the most important thing is to start from a clear assessment of the current situation.

1. Draw a CONTEXT MAP. Can you draw a consistent one or are there ambiguous situations?

2. Attend to the use of language on the project. Is there a UBIQUITOUS LANGUAGE? Is it rich enough to help development?
3. Understand what is important. Is the CORE DOMAIN identified? Is there a DOMAIN VISION STATEMENT? Can you write one?
4. Does the technology of the project work for or against a MODEL-DRIVEN DESIGN?
5. Do the developers on the team have the necessary technical skills?
6. Are the developers knowledgeable about the domain? Are they *interested* in the domain?

You won't find perfect answers, of course. You know less about this project than you ever will in the future. But this gets you a solid starting point. By the time you have specific initial answers to these questions, you'll have start getting insight into what most urgently needs to be done. As time goes along, you can refine the answers, especially the CONTEXT MAP, DOMAIN VISION STATEMENT, and any other artifacts you've created, to reflect changed situations and new insights.

Who Sets the Strategy

Traditionally, architecture is handed down, developed before application development begins, from a team that has more power in the organization than the application development team. But it doesn't have to be that way. That way doesn't usually work very well

Strategic design must, by definition, apply across the project. There are many ways to organize a project, and I don't want to be too prescriptive. However, there are some requirements for any decision making process to be effective.

First, let's take a quick look at the styles I've seen provide some value in practice (thus ignoring the old "wisdom from on-high" style).

Emergent Structure from Application Development

A team capable of doing these things must be disciplined, it must have very good communication. Such a team can use EVOLVING ORDER to arrive at a shared set of principles to operate from so that order is organic rather than imposed by fiat.

This is the typical model for an Extreme Programming team. The structure may emerge completely spontaneously from the insight of any programming pair. More often, having an individual or a subset of the team with some oversight responsibility for large-scale structure helps keep the structure unified. This works particularly if this person is a hands-on developer on the team and is not considered the source of all such decisions so much as an arbiter and communicator. On the Extreme Programming teams I have seen, such strategic design leadership seems to have emerged spontaneously, often in the person of the coach. Whoever this natural leader is, he or she is still a member of the development team. It follows that the development team must have at least a few people of the caliber to make design decisions that will affect the whole project.

When a large-scale structure spans multiple teams, informal collaboration of members of closely affiliated teams may emerge. With a committee like this, the discoveries that lead to the idea for a large-scale structure still happen on an application team, but then the options are discussed with representatives of the various teams. After assessing the impact, they may decide to adopt it, or modify it, or leave it on the table. All the teams attempt to move together in this loose affiliation. This can work when there are relatively few teams, when they are all committed to coordinating with each other, when their design capabilities are comparable, and when their structural needs are similar enough to be met by a single large-scale structure.

Customer Focused Architecture Team

When the strategy will be shared among several teams some centralization of decision does seem attractive. The failed model of the ivory tower architect is not the only possibility. An architecture team can act as a peer with various application teams, helping to coordinate and harmonize their large-scale structures as well as BOUNDED CONTEXT boundaries and other cross-team technical issues. To be useful in this, they must have a mindset that emphasizes application development.

On an org chart, this may look just like the traditional architecture team, but they are actually distinct in every activity. This is a true collaboration with development, discovering patterns along with the developers, experimenting with various teams to reach distillations, and getting their hands dirty.

I have seen this a couple of times, when a project ends up with a lead architect who does most of the things on the following list.

What is essential for strategic design decisions to be effective?

- **Must reach entire team**

Obviously, if everyone doesn't know the strategy and follow it, it is irrelevant. This is one of the points that leads people to organize around centralized architecture teams with official "authority" -- so that the same rules will be applied everywhere. Ironically, the ivory tower architect often gets ignored or bypassed. Lack of hands-on contact and lack of feedback from attempts to apply the rules to real applications makes their rules impractical.

On a project with very good communication, a process where strategic design emerges from the application team may actually reach everyone more effectively. It will be relevant, and it will have the authority that attaches to intelligent community decisions.

Whatever the system, be less concerned with what authority is bestowed by management than the actual relationship the developers have with the strategy.

- **Must absorb feedback**

Creating an organizing principle, large-scale structure, or distillation of this subtlety requires a really deep understanding of the needs of the project and the concepts of the domain. The only people who have that depth of knowledge are the members of the application development team. This is why application architectures created by architecture teams are so seldom helpful, in spite of the undeniable talent of many of the architects.

Unlike technical infrastructure and architectures, strategic design does not itself involve writing a lot of code although it influences all development. What it does require is involvement with the application development teams. An experienced architect may be able to listen to ideas coming from various teams may be able to facilitate the development of a generalized solution.

One technical architecture team I worked with actually circulated its own members through the various application development teams that were attempting to use the framework. The pulled into the architecture team hands-on experience of the challenges facing the developers, while simultaneously transferring the knowledge of how to apply the subtle features of the framework. Strategic design has this same need of a tight feedback loop.

- **Must allow evolution**

Effective software development is a highly dynamic process. When the highest level of decisions is set in stone, the team has fewer options when it must respond to change. EVOLVING ORDER addresses this by emphasizing ongoing change to the large-scale structure in response to deepening insight. This still leaves the question of who makes these decisions.

When too many design decisions are preordained, the development team can be hobbled, without the flexibility to solve the problems they are charged with. So, while a harmonizing principle can be valuable, it must grow and change with the ongoing life of the development project, and it must not take too much power away from the application developers, whose job is hard enough as it is.

Strong feedback helps, since new innovations emerge from understanding the obstacles encountered in building applications and from the unexpected opportunities discovered along the way.

- **Must not siphon off all the best and brightest**

Design at this level calls for sophistication that is probably in short supply. Managers tend to move the most technically talented developers into architecture teams and infrastructure teams because they want to leverage the skills of these advanced designers. For their part, the developers are attracted to the opportunity to have a broader impact, or to work on what are considered by many to be more interesting problems. And there is prestige attached to being a member of one of these elite teams.

This pair of forces often leaves only the least technically sophisticated developers to actually build applications. Building good applications takes design skill, so this is a setup for failure. Even if the strategy team follows all the other points on this list and creates a great strategic design, the application team won't have the design sophistication to follow it.

Conversely, such teams almost never include the developer who has extensive experience in the domain, but maybe weaker design skill. Strategic design is not a purely technical task, and so cutting themselves off from these guys who know everything hobbles their efforts further.

It is essential to have strong designers on all application teams. It is essential to have domain knowledge on any team attempting strategic design. Maybe this means hiring more advanced designers. Maybe it means keeping the strategy team small or part-time. I'm sure there are many ways that work, but any effective strategy team has to have a partner of an effective application team.

- **Minimalism and Humility**

Distillation and minimalism is essential to any good design work, but minimalism is even more critical for strategic design, because even the slightest ill-fit has a terrible potential of getting in the way. Separate architecture teams have to be especially careful because have less feel for the obstacles they might be placing in front of application teams. At the same time, their enthusiasm for their primary responsibility makes them more likely to get carried away. I've seen this many times, and I've even done it. One good idea leads to another and we end up with an overbuilt architecture that is counter-productive.

Instead, we have to discipline ourselves to produce organizing principles and core models are pared down to contain nothing that does not significantly improve the clarity of the design. The truth is, almost everything gets in the way of something, so it had better be worth it. Realizing that your best idea is likely to get in somebody's way takes humility.

- **Objects are Specialists; Developers are Generalists**

The essence of good object design is to give each object a clear and narrow responsibility and reduce interdependence to an absolute minimum. Sometimes we try to make interactions on teams as tidy as they should be in our software. A good project has lots of people sticking their nose in other people's business. Developers play with frameworks. Architects write application code. Everyone talks to everyone. It is efficiently chaotic. Objects are specialists; developers are generalists.

Because I've made the distinction between "strategic design" and other kinds of design to help clarify the tasks, it seems incumbent on me to point out that two kinds of design activity does not mean two kinds of people. Creating a supple design based on a deep model is an advanced design activity, but the details are

so important that it has to be done by someone working with the code. Strategic design emerges out of application design, yet it requires a big picture view of activity possibly spanning multiple teams. People love to find ways to chop up tasks so that one person is a design experts don't have to know the business and domain experts don't have to understand technology. There is a limit to how much an individual can learn, but overspecialization takes the steam out of domain-driven design.

The Same Goes for the Technical Frameworks

Technical frameworks can greatly accelerate application development, including the DOMAIN LAYER, by providing an infrastructure layer that frees the application from implementing basic services, and by helping to isolate the domain from other concerns. But there is a risk that an architecture *can interfere with expressive implementations of the domain model and easy change*. This can happen even when the framework designers had no intention of venturing into the domain or application layers.

The same biases that limit the downside of strategic design can help with technical architecture. Evolution, minimalism, and involvement with the application development team can lead to a continuously refined set of services and rules that genuinely help application development without getting in the way. Architectures that don't follow this path will either stifle the creativity of application development or will find their architecture circumvented, leaving application development, for practical purposes, with no architecture at all.

There is one particular attitude that can ruin frameworks in particular:

- **Don't write frameworks for dummies to use**

Divisions that assume some people are not smart enough to design are likely to fail because they underestimate the difficulty of application design. If those people are not smart enough to design, they shouldn't be assigned to develop software. If they are smart enough, then the attempts to coddle them will only put up barriers between them and the tools they need.

This attitude also poisons the relationship between teams. I've ended up on arrogant teams like this and find myself apologizing to developers in every conversation, embarrassed by my association. (I've never managed to change such a team, I'm afraid.)

Now, *encapsulating irrelevant technical detail is completely different* from the kind of prepackaging I'm disparaging. It is hard to describe the difference in a generalized way. You can tell the difference most easily by asking what the expectation is of the person who will be using the tool/framework/components. If there seems to be a high respect for the user of the framework, then you are probably on the right track.

<<SIDE BAR>>

A group of architects (the kind who design physical buildings), led by Christopher Alexander, were advocates of piecemeal growth in the realm of architecture and city planning. They explained very nicely why master plans fail.

> Without a planning process of some kind, there is not a chance in the world that the University of Oregon will ever come to possess an order anywhere near as deep and harmonious as the order that underlies the University of Cambridge.
> The master plan has been the conventional way of approaching this difficulty. The master plan attempts to set down enough guidelines to provide for coherence in the environment as a whole – and still leave freedom for individual buildings and opens spaces to adapt to local needs.
> …and all the various parts of this future university will form a coherent whole, because they were simply plugged into the slots of the design.
> …in practice master plans fail – because they create totalitarian order, not organic order. They are too rigid; they cannot easily adapt to the natural and unpredictable

> changes that inevitably arise in the life of a community. As these changes occur… the master plan becomes obsolete, and is no longer followed. And even to the extent that master plans *are* followed…they do not specify enough about connections between buildings, human scale, balanced function, etc. to help each local act of building and design become well-related to the environment as a whole. …The attempt to steer such a course is rather like filling in the colors in a child's coloring book… At best, the order which results from such a process is banal. …Thus, as a source of organic order, a master plan is both too precise, and not precise enough. The totality is too precise: the details are not precise enough. …the existence of a master plan alienates the users [since, by definition] the members of the community can have little impact on the future shape of their community because most of the important decisions have already been made.
>
> *The Oregon Experiment*, 1975 (pp. 16-28), [ASAIA1975]

They advocate instead a *set of principles* that all community members apply to every act of piecemeal growth, so that "organic order" emerges, well adapted to circumstances.

<<END SIDE BAR>>

Part V. Conclusions

Epilogues

Although we all get satisfaction from the working on a cutting edge project and from experimenting with interesting ideas and tools, for me it is a hollow experience if the software does not find productive use. In fact, the true test of success is how the software serves over a period of time. I have been able to follow the story of some of my former projects over the years.

Here I'll follow up on a few projects that I worked on extensively, and that made a serious attempt at domain-driven design, though not systematically and not by that name, of course. All of these projects did deliver software, but some managed to carry through and produce a model-driven design while one slipped off that track. Some of the applications continued to grow and change for many years, while one stagnated and one died young.

Newly Planted Olive Grove

The PCB design software storied in Chapter 1 was a smash hit among beta users in the field. Unfortunately, the startup company that had initiated the project utterly failed in its marketing function, and was eventually euthanized. The software is now used by a handful of PCB engineers who have old copies they kept from the beta program. Like any orphan software, it will continue to work until there is a fatal change to some program it is integrated with.

The loan software discussed in Chapter 9 thrived and evolved on much the same track for three years after the breakthrough I wrote about. At that point, the project was spun off as an independent company. In the turmoil of this reorganization, the project manager who had led the project from the beginning was ejected and some of the core developers left with him. The new team had a somewhat different design philosophy, not as fully committed to object modeling, but retaining a distinct DOMAIN LAYER with complex behavior and valuing deep domain knowledge on the development team. Seven years after the spin-off, the software continues to be enhanced with new features. It is the leading application in its field and serves an increasing number of client institutions, as well as being the largest revenue stream for the company.

Seven Years Later

On one project I paired with another developer to write a utility the customer needed to produce their core product. The features were fairly complicated and combined in intricate ways. I enjoyed the project work and we produced a supple design with an ABSTRACT CORE. When this was handed off, that was the end of involvement of everyone who had initially developed it, and because it was such an abrupt transition I expected the design features that supported the combinable elements might be confusing and get replaced by more typical case logic. This did not initially happen. When we handed off, the package included a thorough test suite and a DISTILLATION DOCUMENT. They used that document to guide their explorations and, as they explored, they became excited by the possibilities the design presented to them. When I heard their comments a year later, I realized that the UBIQUITOUS LANGUAGE had sparked across and stayed alive, continuing to evolve.

Then, another year later, I heard a different story. The team had encountered new requirements that they didn't see any way to accomplish within the inherited design. They had been forced to change the design almost beyond recognition. As I probed for more details, I could see that aspects of our model would have made solving those problems awkward. It is precisely such moments when a breakthrough to a deeper model is often possible, especially when, as in this case, the developers had accumulated deep knowledge and experience in the domain. In fact, they had had a rush of new insights and ended up transforming the model and design based on those insights.

They told me this story carefully, diplomatically, expecting, I suppose, that I would be disappointed by their discarding of so much of my work. I am not that sentimental about my designs. The success of a design is not necessarily marked by its longevity. Make an essential system opaque and it will live forever as untouchable legacy. A deep model allows clear vision that can yield new insight, while a supple design facilitates ongoing change. The model they came up with was deeper, better aligned with the real concerns of the users. Their design solved real problems. It is the nature of software to change, and this program has continued to evolve in the hands of the team that owns it.

Until the domain-driven approach is more widespread, the interesting software on many projects will be built in a short, highly productive interval. Eventually the project will transform into something more conventional that may not be able to fully exploit, much less enhance, the power of the deep models that were distilled earlier. I could wish for more, but truly those are successes that deliver sustained value to users over many years.

The shipping examples scattered through the book are loosely based on a project for a major international container shipping company. Early on, the leadership of the project was deeply committed to a domain-driven approach, but never produced a development culture that could fully support it. Several teams with widely different levels of design skill and object experience set out to create modules, loosely coordinated by informal cooperation between team leaders and by a customer-focused architecture team. We did develop a reasonably deep model of the CORE DOMAIN, and there was a viable UBIQUITOUS LANGUAGE.

The company culture fiercely resisted iterative development, and we waited far too long to push out a working internal release. Therefore, problems were exposed at a late stage, when they were more risky and expensive to fix. Some aspects of the model led to performance problems in the database. A natural part of MODEL-DRIVEN DESIGN is the feedback from implementation problems to changes in the model, but by that time there was a perception that we were too far down the road to change the fundamental model. Instead, changes were made to the code to make it more efficient, and its connection to the model was weakened. The initial release also exposed scaling limitations in the technical infrastructure that threw a scare into management. Expertise was brought in to fix the infrastructure problems and the project bounced back. But the loop was never closed between implementation and domain model.

A few teams delivered fine software with complex capabilities and expressive models. Others delivered stiff software that reduced the model to data structures, though even they retained traces of the UBIQUITOUS LANGUAGE. Perhaps a CONTEXT MAP would have helped us as much as anything, as the relationship between the output of the various teams was haphazard. Yet that core model carried in the UBIQUITOUS LANGUAGE did help the teams ultimately to glue together a system.

Although reduced in scope, the project replaced several legacy systems. The whole was held together by a shared set of concepts, though most of the design was not very supple. It has itself largely fossilized into legacy now, five years later, but it still serves the global business 24 hours a day. Gradually, the more successful teams' influence spread, but time runs out eventually, even in the richest company. The culture of the project never really absorbed MODEL-DRIVEN DESIGN. New development is on different platforms and is only indirectly influenced by the work we did – as they CONFORM to their legacy.

In some circles, ambitious goals like those the shipping company started with have been discredited. Better, it seems, to make simple little applications we know how to deliver. Better to stick to the lowest common denominator of design to do simple things. This conservative approach has its place, and allows for neatly scoped, quick response projects. But integrated model-driven systems promise value that those patchworks

can't. There is a third way. Domain-driven design allows piecemeal growth of big systems with rich functionality, by building on a deep model and supple design.

I'll close this list with Evant, a company that develops inventory management software, where I played a secondary supporting role and contributed to an already strong design culture. Others have written about this project as a poster-child of Extreme Programming, but what is not usually remarked upon is that the project was intensely domain-driven. Ever deeper models were distilled and expressed in ever more supple designs. This project thrived until the "dot com" crash of 2001. Then, starved for investment funds, the company contracted, software development went mostly dormant, and it seemed that the end was near. But in Summer 2002, they were approached by one of the top 10 retailers in the world. This potential client liked the product, but needed design changes to allow the application to scale up for an enormous inventory planning operation. It was Evant's last chance.

Although reduced to four developers, the team had assets. They were highly skilled, with knowledge of the domain and one member with expertise in scaling issues. They had a very effective development culture. And they had a code-base with a supple design that facilitated change. That summer, those four made a heroic development effort resulting in the ability to handle billions of planning elements and hundreds of users. On the strength of these capabilities, Evant won the behemoth client and, soon after, was bought by another company that wanted to leverage their software and their proven ability to accommodate new demands.

The domain-driven design culture (as well as the Extreme Programming culture) survived the transition and was revitalized. Today, the model and design continue to evolve, far richer and more supple two years later than when I made my contribution. And rather than being assimilated into the purchasing company, the existing projects seem to be interested in following the lead of the Evant team. This story isn't over yet.

No project will ever employ every technique in this book. Even so, any project committed to domain-driven design will be recognizable in a few ways. The defining characteristic is a priority of understanding the target domain and incorporating that understanding into the software. Everything else flows from that premise. Team members are conscious of the use of language on the project and cultivate its refinement. They are hard to satisfy with the quality of the domain model, because they keep learning more about the domain. They see continuous refinement as an opportunity, and an ill-fitting model as a risk. They take design skill seriously because it isn't easy to develop production quality software that clearly reflects the domain model. They stumble over obstacles, but hold onto their principles as they pick themselves up and continue forward.

Looking Forward

Weather, ecosystems and biology used to be considered messy, “soft” fields in contrast to physics or chemistry. Recently, however, it has been recognized that the appearance of “messiness” in fact presents a profound technical challenge to discover and understand the order in these very complex phenomena. The field of “complexity” is the vanguard of the sciences. While purely technological tasks have generally seemed most interesting and challenging to talented software engineers, domain-driven design opens up a new area of challenge that is at least equal. Business software does not have to be a bolted together mess. Wrestling a complex domain into a comprehensible software design is an exciting challenge for strong technical people.

We are nowhere near the era of lay people creating complex software that works. Armies of programmers with rudimentary skills can produce certain kinds of software, but not the kind that saves a company in its eleventh hour. What is needed is for tool builders to put their minds to the task of extending the power and productivity of talented software developers. What is needed is sharper ways of exploring domain models and expressing them in working software. I look forward to experimenting with new tools and technologies devised for this purpose.

But while improved tools will be valuable, we mustn’t get distracted by them and loose sight of the core fact that creating good software is primarily a learning and thinking activity. Modeling requires imagination and self-discipline. Tools that help us think or avoid distraction are good. Efforts to automate what must be the product of thought are naïve and counterproductive.

With tools and technology we already have, we can build much more valuable systems than most projects today. We can write software that is a pleasure to use and a pleasure to work on; software that doesn’t box us in as it grows, but creates new opportunities and continues to add value for its owners.

Glossary

Here are brief definitions of selected terms, pattern names and otherwise, used in the book.

AGGREGATE, A cluster of associated objects that are treated as a unit for the purpose of data changes. External references are restricted to on member of the AGGREGATE, designated as the "root", a set of consistency rules applies within the AGGREGATE'S boundaries.

analysis pattern, A group of concepts that represent a common construction in business modeling. It may be relevant to only one domain or it may span many domains.

ASSERTION, A statement of what a design element must do, independently of how it does it. Ideally, it should be in a form that can be implemented as part of the program.

BOUNDED CONTEXT, delimited applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other models in other contexts

client, program element that is calling the element under design, using its capabilities.

command, (aka modifier), an operations that affects some change to the systems (e.g, by setting a variable). An operation that creates a side effect.

CONCEPTUAL CONTOUR, Scope of classes, operations and AGGREGATES is dictated by natural conceptual distinction, relying on the underlying consistency of the domain itself to make the design more naturally accommodate change.

context, The setting in which a word or statement appears that determines its meaning. See BOUNDED CONTEXT.

CONTEXT MAP, a representation of the BOUNDED CONTEXTS involved in a project and the actual relationships between them.

cohesion, logical agreement and dependence; as, the cohesion of ideas. --Locke. [definition 3 in Merriam-Webster Dictionary].

CORE DOMAIN, the distinctive part of the model, central to the user's goals, that differentiates the application and makes it valuable.

declarative design, A form of programming in which a precise description of properties actually controls the software. An executable specification.

deep model, An incisive expression of the primary concerns of the domain experts and their most relevant knowledge that sloughs off the superficial aspects of the domain and naïve interpretations.

distillation, a process of separating the components of a mixture to extract the essence in a form that makes it more valuable and useful.

domain, a sphere of knowledge, influence, or activity <the *domain* of art> [definition 4 in Merriam-Webster Dictionary].

domain expert, A member of a software project whose field is the domain of the application, rather than software development. Not just any user of the software, but someone with deep knowledge of the subject.

domain layer, The design and implementation of domain logic constitute the domain layer. The domain model is a set of concepts. In model-driven design, software elements directly correlate to those concepts. The domain layer where those domain design elements live.

design pattern, descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. [GHJV1995, p.3]

ENTITY, An object not fundamentally defined by its attributes, but by a thread of continuity and identity.

FACTORY, a mechanism for encapsulating complex creation logic and abstracting the type of the created object from the point of view of the client.

function, an operation that computes and returns a result without observable side effects.

immutable, never changing observable state after creation.

implicit concept, A concept necessary to understanding the meaning of a model or design, but which is never mentioned.

INTENTION-REVEALING INTERFACE, A design where names of classes, methods, and other elements convey the original developer's purpose in creating them, and their value to a client developer.

invariant, an assertion that must always be true, except during specifically transient situations like the middle of the execution of a method, or the middle of an uncommitted database transaction.

iteration, a process in which a program is repeatedly improved in small steps. One of those steps.

large-scale structure, a set of high-level concepts and/or rules that establish a pattern of design for an entire system. A language that allows the system to be discussed and understood in broad strokes.

LAYERED ARCHITECTURE, a system of separating the concerns of a system, isolating, among other things, a domain layer.

lifecycle, a sequence of states an object goes through between creation and deletion, typically with constraints to ensure integrity when going from one state to another.

model, a system of abstractions that describe selected aspects of a domain and can be used to solve problems related to that domain.

MODEL-DRIVEN DESIGN, a design in which some subset of software elements correspond as closely as possible elements of a model. A process of co-developing a model and implementation that stay aligned with each other.

modeling paradigm, a particular style of carving out concepts in a domain, combined with tools to create software analogs of those concepts. (e.g. object-oriented programming and logic programming)

paradigm, see modeling paradigm.

REPOSITORY, A globally accessible object responsible for finding and retrieving an object that has previously been created and stored.

responsibility, a mechanism for encapsulating storage, retrieval and search behavior which emulates a collection of objects.

SERVICE, an operation offered as an interface that stands alone in the model, without encapsulating state.

side effect, any observable change of state resulting from an operation, even a deliberate update.

SIDE-EFFECT-FREE FUNCTION, see function.

STANDALONE CLASS, a class that can be understood and tested without reference to any others, except system primitives and basic libraries.

stateless, any client can use any instance of a stateless element without regard to the instance's individual history. May use information that is accessible globally, and may even change that global information (have side-effects), but holds no private state that affects its behavior.

strategic design, modeling and design decisions that apply to large parts of the system. Strategic design helps manage large systems. Such decisions affect the entire project, and have to be decided at team level.

supple design, A supple design puts the power inherent in a deep model into the hands of a client developer, to enable natural, clear and flexible expressions that give expected results robustly. It leverages

that *same* deep model to make the design easy for the implementer to mold and reshape to accommodate new insight.

UBIQUITOUS LANGUAGE, a language structured around the domain model, and used by all team members to connect all the activities of the team with the software.

unification, internal consistency of a model such that each term is unambiguous and no rules contradict.

VALUE OBJECT, An object that describes some characteristic or attribute, but carries no concept of identity.

WHOLE VALUE, an object that models a single, complete concept.

References

[AIS1977] Alexander, C., Ishikawa, S., Silverstein, M. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

[ASAIA1975] Alexander, C., Silverstein, M., Angel, S., Ishikawa, S., Abrams, D. 1975. *The Oregon Experiment.* Oxford University Press.

[Beck 1997] Beck, Kent, 1997. *Smalltalk Best Practice Patterns*. Prentice Hall PTR.

[Beck 2000] Beck, Kent, 2000. *Extreme Programming Explained, Embrace Change*. Addison-Wesley.

[Beck 2002] Beck, Kent, 2002. *Test Driven Development: by Example*. Addison-Wesley.

[BMRSS1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. P. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley.

[Cockburn 1998] Cockburn, A., 1998. *Surviving Object-Oriented Projects.* Addison-Wesley.

[EF 1997] Evans, E., Fowler, M. 1997, "Specifications", Proceedings of PLoP 97 Conference.

[FJ 2000] Fayad, M., Johnson, R. 2000. *Domain-Specific Application Frameworks.* Wiley.

[Fowler 1996] Fowler, M., 1996. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

[Fowler 1999] Fowler, M., 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Fowler 2002] Fowler, M., 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley.

[GHJV 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns*. Addison-Wesley.

[Kerievsky 2001] Kerievsky, Joshua 2001, "Continuous Learning", Proceedings of XP 2001 Conference.

[Larman 1998] Larman, Craig 1998. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall.

[Meyer 1988] Meyer, Bertrand 1988. *Object-oriented Software Construction*. Prentice Hall.

[MRL1995] Murray-Rust, P., Rzepa, H., Leach, C. *Abstract 40. Presented as a poster at the 210th ACS Meeting in Chicago on 21 August 1995.* **http://www.ch.ic.ac.uk/cml/**

[WWW 1990] Wirfs-Brok, R., Wilkerson, B., Wiener, L. 1990. *Designing Object-Oriented Software*. Prentice Hall.

[WM 2003] Wirfs-Brok, R., McKean, A. 2003. *Object Design: Roles, Responsibilities, and Collaborations.* Addison-Wesley.

Appendix: Use of Patterns in This Book

My first “nice car”, shortly after college, was an eight-year-old Peugeot. Sometimes called the “French Mercedes”, this car was well crafted, a pleasure to drive, and had been very reliable. But by the time I got it, it was reaching the age where things start to go wrong, and more maintenance is required.

Peugeot is an old company and has followed its own evolution over many decades. They have their own mechanical terminology, and their designs are idiosyncratic; even the breakdown of functions into parts is sometimes nonstandard. The result is a car that only Peugeot specialists can work on, a potential problem on a grad student income.

On one typical occasion I took the car to a local mechanic to investigate a fluid leak. He examined the undercarriage and told me that oil was “leaking from a little box about two thirds of the way back that seems to have something to do with distributing braking power between front and rear”. He then refused to touch it and advised me to go to the dealership, 50 miles away. Anyone can work on a Ford or a Honda, which makes those cars more convenient and less expensive to own.

I did love that car, but I will never own a quirky car again.

Standard design elements are lacking for domain development, and so every domain model and corresponding implementation is quirky and hard to understand. Moreover, every team has to reinvent the wheel, (or the gear, or the windshield wiper). In the world of object-oriented design, everything is an object, a reference or a message, which, of course, is a useful abstraction. But that does not sufficiently constrain the range of domain design choices and does not to support an economical discussion of a domain model.

To stop with “everything is an object” would be like a carpenter or architect who referred to every room of a house as simply a “room”. There would be the big room with high voltage outlets and a sink, where you might cook. There would be the small room upstairs, where you might sleep. It would take me pages to describe an ordinary house. People who build or use houses realize that rooms follow patterns, patterns with special names, like “kitchen”. This enables economical discussion of house design.

Not only that, but not all combinations of functions turn out to be practical. Why not a room where you bathe and sleep? Wouldn’t that be convenient. But long experience has precipitated into custom and we separate our “bedrooms” from our “bathrooms”. After all, bathing facilities tend to be shared among more people than bedrooms are and require very high privacy even from the others who share the same bedroom. And bathrooms have specialized and expensive infrastructure requirements. Bathtubs and toilets typically end up in the same room because they both require the same infrastructure (water and drainage) and they both demand very high privacy.

Another room that has special infrastructure requirements is that room you might prepare meals in, also known as the “kitchen”. In contrast to the bathroom, a kitchen has no special privacy requirements. Because of its expense, there is typically only one, even in relatively large houses. This also facilitates our communal food preparation and eating customs.

When I say that I want a three bedroom, two bath house with an open-plan kitchen, I have packed a huge amount of information into a short sentence, and I’ve avoided a lot of silly mistakes that could have been made, such as putting a toilet next to the refrigerator.

In every area of design - houses, cars, rowboats, or software - we build on patterns that have been found to work in the past, improvising within establish themes. Sometimes we have to invent something completely new. But by basing standard elements on patterns, we avoid wasting our energy on problems with known solutions so that we can focus on our unusual needs. Also, building from conventional patterns helps us avoid a design so idiosyncratic it is difficult to communicate.

While software domain design is not as mature as other design fields, and in any case may be too diverse to accommodate patterns as specific as those used for car-parts or rooms, there is nonetheless a need to move beyond the "everything is an object" level to at least the equivalent of distinguishing bolts from springs.

A form for sharing and standardizing design insight was introduced in the 1970s by a group of architects led by Christopher Alexander [AIS1977]. Their "pattern language" wove together tried and true design solutions to common problems (much more subtle than my "kitchen" example above). The intent was that builders and users would communicate in this language and be guided by the patterns to produce beautiful buildings that worked well and felt good to the people who used them.

Whatever the architects may think of the idea, it has had a big impact on software design. In the 1990's software patterns were applied in many ways with some success, notably in detailed design [GHJV95] and technical architectures [BMRSS1996] More recently it has been used to document basic object-oriented design techniques [Larman 1998] and enterprise architectures [Fowler 2002]. It is now a main-stream technique for organizing software design ideas.

The format for patterns used in this book:

PATTERN NAME

[Illustration of concept]

Context. Brief explanation of how the concept relates to other patterns. In some cases, a brief overview of the pattern.

♦ ♦ ♦

Problem discussion.

Problem summary.

Discussion of resolution of problem forces into a solution.

[Therefore,]

Solution summary.

Solution consequences. Implementation considerations.

♦ ♦ ♦

[Optional: Discussion of implementation challenges. (In Alexander's original format, this would have been folded into the resolution of problem section, and it is often handled that way in this book. But some patterns demand lengthier discussions of implementation. To keep the core pattern discussion tight, I have moved these into this extension.

♦ ♦ ♦]

Resulting context: Brief explanation of how the pattern leads to later patterns.

The pattern names are meant to become terms in the language of the team. When a pattern name appears in a discussion, it will be FORMATTED to call it out.