III

Refactoring Toward Deeper Insight

Part II of this book laid a foundation for maintaining the correspondence between model and implementation. Using a proven set of basic building blocks along with consistent language brings some sanity to the development effort.

Of course, the real challenge is to actually *find* an incisive model, one that captures subtle concerns of the domain experts and 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 part of the book 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 as well as the developers themselves.

Success developing useful models 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 to use effectively.

Levels of Refactoring

Refactoring is the redesign of software in ways that do not change its functionality. Rather than making elaborate up-front design decisions, developers take code 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 allows 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"[1]

1. Patterns as targets for refactoring were briefly mentioned in Gamma et al. (1995). Joshua Kerievsky has developed refactoring to patterns into a more mature and useful form (Kerievsky 2003).

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 a 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 type of refactoring 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 based on domain 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 goal is that not only can a developer understand what the code does; he or she can also understand *why* it does what it does and can relate that to the ongoing communication with the domain experts.

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 would be impossible to compile.

Modeling is as inherently unstructured as any exploration. Refactoring to deeper insight should follow wherever learning and deep thinking lead. Published collections of successful models can be helpful, as discussed in Chapter 11, but we shouldn't get sidetracked trying to reduce domain modeling to a cookbook or a toolkit. Modeling and design call for creativity. The next six 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 explanation is recognized as an oversimplification that can be useful for teaching object modeling to

beginners. The truth is, though, that initial models usually are naive and superficial, based on shallow knowledge.

For example, I once worked on a shipping application for which 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 to a layperson, but much more relevant to the experts. It was refocused on the business of delivering cargo.

The ships were still there, but 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 shipping container all but disappeared from the model. It did emerge in a cargo-handling application in a different, very complex form, but in the context of the original application, the container 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, such as 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.

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.

Versatility, simplicity, and explanatory power come from a model that is truly in tune with the domain. One feature such 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 for 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 10.

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 can actually feed insight into the model discovery process when it has the flexibility to let a developer experiment and the clarity to show a developer what is happening. 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 concepts 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 effort 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 depend 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." Such patterns are not ready-made solutions, but they feed your knowledge crunching process and narrow your search.

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. Although such breakthroughs don't come along every day, they are so valuable that when they do happen, 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. This experience is not something you can plan for. Nonetheless, it provides a good context for thinking about domain refactoring.