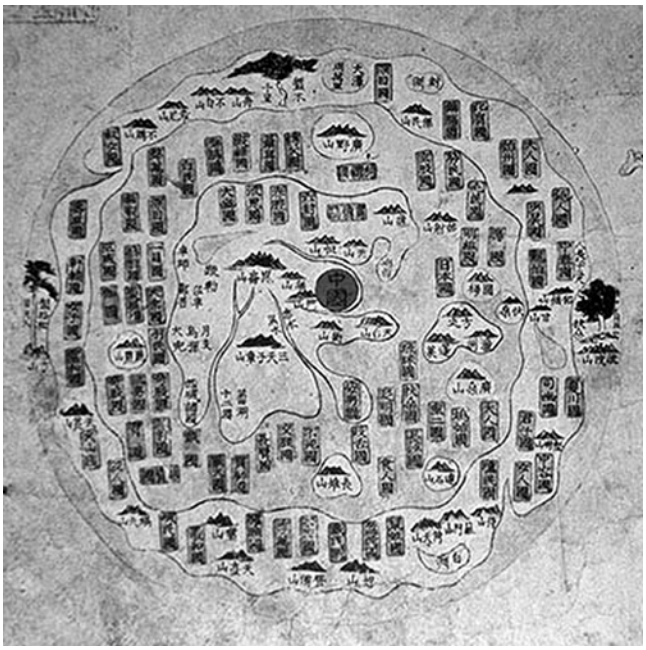
I

Putting the Domain Model to Work



This eighteenth-century Chinese map 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 an idea that is of interest. A model 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 software that is valuably involved in users' activities, a development team must bring to bear a body of knowledge related to those activities. The breadth of knowledge required can be daunting. The volume and complexity of information can be overwhelming. Models are tools for grappling with this overload. A model is a selectively simplified and consciously structured form of knowledge. An appropriate model makes sense of information and focuses it on a problem.

A 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 moviemaking, 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 modeler chooses a particular model for its utility.

The Utility of a Model in Domain-Driven Design

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

1. *The model and the heart of the design shape each other.* 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 during maintenance and continuing development, because the code can be interpreted based on understanding the model. (See 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. (See 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 developers and domain experts to collaborate effectively as they wrestle information into this form. The binding of model and implementation makes experience with early versions of the software applicable as feedback into the modeling process. (See 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.

The Heart of Software

The heart of software is its ability to solve domain-related problems for its 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 these are not the priorities on most software projects. Most talented developers do not have much interest in learning about the specific domain in which they are working, much less making a major commitment to expand their domain-modeling skills. Technical people enjoy quantifiable problems that exercise their technical skills. Domain work is messy and demands a lot of complicated new knowledge that doesn't seem to add to a computer scientist's capabilities.

Instead, the technical talent goes to work on elaborate frameworks, trying to solve domain problems with technology. Learning about and modeling the domain is left to others. Complexity in the heart of software has to be tackled head-on. To do otherwise is to risk irrelevance.

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, 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, Cleese 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. 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 focused on the precise execution of his own specialty. He was concerned that other film editors who saw the movie would judge his work based on its technical perfection. 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 development of a model that reflects deep understanding gets lost in the shuffle.

This book will show that domain development holds opportunities to cultivate very sophisticated design skills. The messiness of most

software domains is actually 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 developers 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 makes a developer much more valuable, even in an initially unfamiliar domain.