ONE

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 having the PCB designers 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.

The 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 electrical 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.

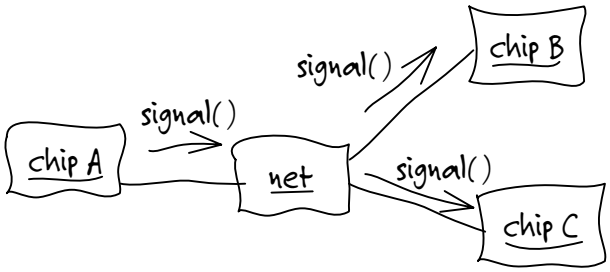


Figure 1.2

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?

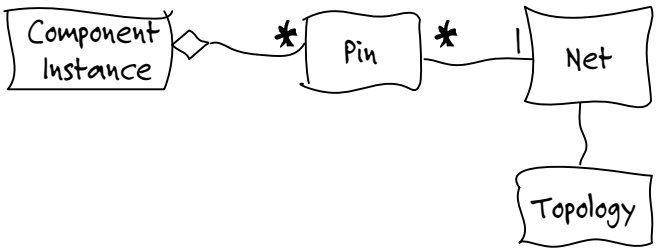


Figure 1.3

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.]

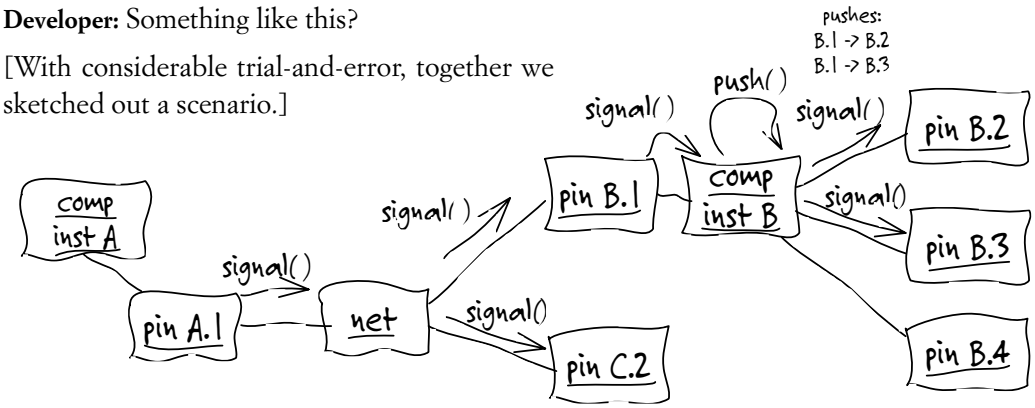


Figure 1.4

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 than 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.

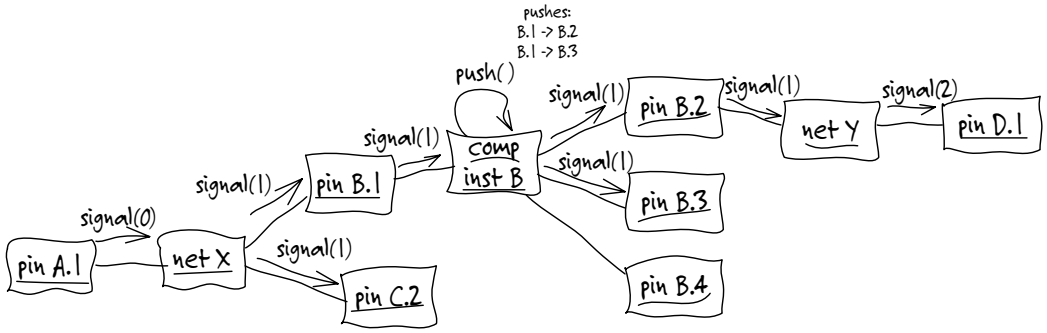


Figure 1.5

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?

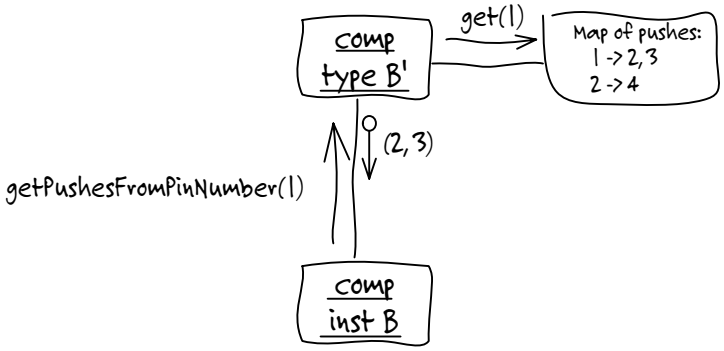


Figure 1.6

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. . . . 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 looks something like this.

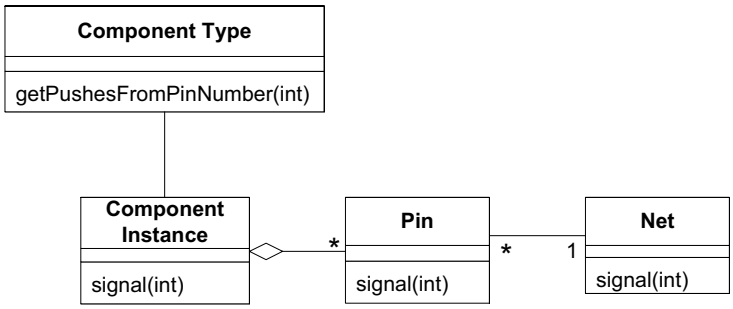


Figure 1.7

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 user interface (UI). 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 nonetheless 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 the domain experts 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 than 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 how the objects interacted. When the model objects couldn't carry us through an important scenario, we brainstormed new ones or changed old ones, crunching their knowledge. We refined the model; the code coevolved. A few months later the PCB engineers had a rich tool that exceeded their expectations.

Ingredients of Effective Modeling

Certain things we did led to the success I just described.

1. *Binding the model and the implementation.* That crude prototype forged the essential link early, and it was maintained through all subsequent iterations.

2. *Cultivating a language based on the model.* At first, the engineers had to explain elementary PCB issues to me, and I had to explain what a class diagram meant. But as the project proceeded, any of us could take terms straight out of the model, organize them into sentences consistent with the structure of the model, and be unambiguously understood without translation.

3. *Developing a 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. *Distilling the model.* 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 an 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. *Brainstorming and experimenting.* 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. As the team went through scenarios, the spoken expressions themselves provided a quick viability test of a proposed model, as the ear could quickly detect either the clarity and ease or the awkwardness of expression.

It is the creativity of brainstorming and massive experimentation, leveraged through a model-based language and disciplined by the feedback loop through implementation, that makes it possible to find a knowledge-rich model and distill it. This kind of *knowledge crunching* turns the knowledge of the team into valuable models.

Knowledge Crunching

Financial analysts crunch numbers. They sift through reams of detailed figures, combining and recombining them looking for the underlying meaning, searching 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 makes 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 interpretations.

In the old waterfall method, the business experts talk to the analysts, and analysts digest and abstract and pass the result along to the programmers, who code the software. This approach fails 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 of software. Knowledge trickles in one direction, but does not accumulate.

Other projects use an iterative process, but they fail to build up knowledge because they don't abstract. Developers get the experts to describe a desired feature and then they go build it. They show the experts the result and ask 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 learn only what the application should do, not the principles behind it. Useful software can be built that way, but the project will never arrive at a point 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 naive. 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 to produce functions mechanically. 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 that software projects require.

All this makes the team members 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, so it can provide leverage for the implementation. Because the domain experts are feeding into it, the model reflects deep knowledge of the business. The abstractions are true business principles.

As the model improves, it becomes a tool for organizing the information that continues to flow through the project. The model focuses requirements analysis. It intimately interacts with programming and design. And in a virtuous cycle, it deepens team members' 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 it's 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 ignorance 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 knowledge isn't. And with typical design approaches, the code and documents don't

express this hard-earned knowledge in a usable form, so when the oral tradition is interrupted for any reason, the knowledge is lost.

Highly productive teams grow their knowledge consciously, practicing *continuous learning* (Kerievsky 2003). 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. (For more on this, see Chapter 15.) The accumulated knowledge in the minds of this core team makes them more effective knowledge crunchers.

At this point, stop and ask yourself a question. Did you learn something about the PCB design process? Although this example has been a superficial treatment of that domain, 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, our team eventually discovered that the probe simulation was a low priority for development, and the feature 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 lie 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 discusses these issues in depth.)

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

Knowledge-Rich Design

The kind of knowledge captured in a model such as the PCB example goes beyond "find the nouns." Business activities and rules are as central to a domain as are the entities involved; 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, 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, reconcile contradictions, and fill 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 could be the basis of an application for booking cargos onto a voyage of a ship.

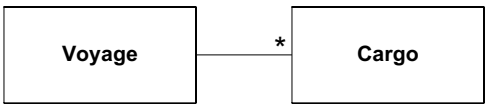


Figure 1.8

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;
}
```

Because there are always last-minute cancellations, standard practice in the shipping industry is to accept more cargo than a particular vessel can carry on a voyage. This is called "overbooking."

Sometimes a simple percentage of capacity is used, such as booking 110 percent of capacity. In other cases complex rules are applied, favoring major customers or certain kinds of cargo.

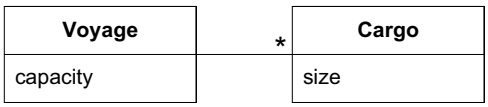
This is a basic strategy in the shipping domain that would be known to any businessperson in the shipping industry, but it might not be understood by all technical people on a software team.

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, in Chapter 4, we'll look at the principle of LAYERED ARCHITECTURE, which would guide us to move 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. This will bring us to a similar solution.

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

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

We can change the design to better capture this knowledge. The overbooking rule is a policy. *Policy* is another name for the design pattern known as STRATEGY (Gamma et al. 1995). It is usually moti-

vated 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 12, "Relating Design Patterns to the Model.")

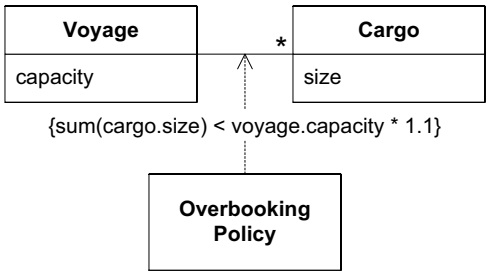


Figure 1.10

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 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 a domain model and corresponding design can be used to secure and share knowledge. The more explicit design has these advantages:

1. In order to bring the design to this stage, the programmers and everyone else 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 domain experts (with guidance), thereby closing the feedback loop.

Deep Models

Useful models seldom lie on the surface. As we come to understand the domain and the needs of the application, we usually discard superficial model elements that seemed important in the beginning, or we shift their perspective. Subtle abstractions emerge that would not have occurred to us at the outset but that pierce to the heart of the matter.

The preceding example is loosely based on one of the projects that I'll be drawing on for several examples throughout the book: a container shipping system. The examples in this book will be kept accessible to non-shipping experts. But on a real project, where continuous learning prepares the team members, models of utility and clarity often call for sophistication both in the domain and in modeling technique.

On that project, because a shipment begins with the act of 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, after months of knowledge crunching, we realized that the handling of cargo, the physical loading and unloading, the movements from place to place, was largely carried out by subcontractors or by operational people in the company. In the view of our shipping experts, there was a series of transfers of responsibility between parties. A process governed that transfer of legal and practical responsibility, from the shipper to some local carrier, from one carrier to another, and finally 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 the shipping company's business decisions. Rather than

the logistics of the itinerary, what came to the fore were legal documents such as the bill of lading, and processes leading to the release of payments.

This deeper view of the shipping business did not lead to the removal of the 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.