Chapter 7

Conceptual Model of Software Architecture

In this book's introduction, you read a story about a coach and a rookie watching the same game. They both saw the same things happening on the field, but despite the rookie's eyes being younger and sharper, the coach was better at understanding and evaluating the action. As a software developer, you would like to understand and evaluate software as effectively as the coach understands the game. This and subsequent chapters will help you build up a mental representation of how software architecture works so that when you see software you will understand it better and will design it better.

The idea of using models, however, is often wrongly conflated with the choice of software process (i.e., waterfall) and has been associated with analysis paralysis. This book is not advocating building lots of written models (i.e., documentation) up front, so it is best to knock down a few strawmen arguments or misunderstandings:

- **Every project should document its architecture: False**. You should make plans before going on a road trip, but do you plan your commute to work in the morning? Models help you solve problems and mitigate risks, but while some problems are best solved with models, others can be solved directly.

- **Architecture documents should be comprehensive: False**. You may decide to build a broad architecture document, or even a comprehensive one, but only in some circumstances — perhaps to communicate a design with others. Most often you can model just the parts that relate to your risks, so a project with scalability risks would build a narrow model focusing on scalability.

- **Design should always precede coding: False**. In one sense this is true, because code does not flow from your fingers until you have thought about what you will build. But it is false to believe that a design phase (in the software process sense) must precede coding. In fact, early coding may help you discover the hardest problems.

So you should set these strawmen ideas aside. The real reason to use software architecture models is because they help you perform like the coach, not the rookie. If you are not already at the coach level, you want to get there as soon as possible. The standard architecture models represent a condensed body of knowledge that enables you to efficiently learn about software architecture and design. Afterwards, you will find that having a standard model frees your mind to focus on the problem at hand rather than on inventing an new kind of model for each problem.

Conceptual models accelerate learning. If you want to become as effective as a coach, you could simply work on software and wait until you are old. Eventually, all software developers learn something about architecture, even if they sneak up on that knowledge indirectly. It just takes practice, practice, practice at building systems. There are several problems with that approach, however. First, not all old software developers are the most effective ones. Second, the approach takes decades. And third, your understanding of architecture will be idiosyncratic, so you will have a hard time communicating with others, and vice versa.

Consider another path, one where you see farther by standing on the shoulders of others. Perhaps we are still waiting for the Isaac Newton of software engineering, but there is plenty to learn from those who have built software before us. Not only have they given us tangible things like compilers and databases, they have given us a set of abstractions for thinking about programs. Some of these abstractions have been built into our programming languages — functions, classes, modules, etc. Others likely will be, such as components, ports, and connectors[1].

Some people are born brilliant, but for those of us who are not, how effective is standing on the shoulders of those who came before us? Consider this: you are probably a better mathematician than all but a handful of the people in the 17th century. Then, as now, math virtuosos had talent and practiced hard, but today you have the benefit of centuries of compacted understanding. By the time you leave high school, you solve math problems that required a virtuoso a few hundred years ago. And before that, the virtuosos of the 17th century had the benefit of someone else inventing the positional number system and the concept of zero. As you consider

[1]Research languages like ArchJava have already added these concepts to Java.

the two paths, remember that you can and should do both: learn the condensed understanding of architecture and then practice, practice, practice.

Conceptual models free the mind. A condensed understanding can take the form of a conceptual model. The coach's conceptual model includes things like offense and defense strategies, positions, and plays. When he watches the movement of players on the field, he is categorizing what he sees according to his conceptual model. He sees the motion of a player as more than that — it is an element of a play, which is part of a strategy. The rookie, with his limited conceptual model, sees less of this.

Conceptual models accelerate progress in many fields. If you ever took physics, you may have forgotten most of the equations you learned, but you will still conceive of forces acting on bodies. Your physics teacher's lessons were designed to instill that conceptual model. Similarly, if you have ever studied design patterns, you cannot help but recognize those patterns in programs you encounter.

A conceptual model can save you time through faster recognition and consistency, and amplify your reasoning. Alfred Whitehead, said "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race." (Whitehead, 1911) This applies equally to conceptual models. As mentioned in the introduction, Alan Kay has observed that a "point of view is worth 80 IQ points", continuing to say that the primary reason we are better engineers than in Roman times is because we have better problem representations (Kay, 1989).

There is a general consensus on the essential elements and techniques for architecture modeling, though different authors emphasize different parts. For example, the Software Engineering Institute (SEI) emphasizes techniques for quality attribute modeling (Bass, Clements and Kazman, 2003; Clements et al., 2010). The Unified Modeling Language (UML) camp emphasizes techniques for functional modeling (D'Souza and Wills, 1998; Cheesman and Daniels, 2000). The conceptual model in this book integrates both quality attribute and functional models.

Chapter goals and organization. The goal of this part of the book is to provide you with a conceptual model of software architecture, one that enables you to quickly make sense of the software you see and reason about the software you design. The conceptual model includes a set of abstractions, standard ways of organizing models, and know-how. You will never become good at anything without talent and practice, but you can accelerate your progress by building up a mental conceptual model.

This chapter shows you how to partition your architecture into three primary models: the domain, design, and code. It relates these models using designation and refinement relationships. Within each model, details are shown using views. The three chapters that follow this one examine the domain, design, and code models in more detail. An example system for a website called Yinzer runs throughout. A *Yinzer* is a slang term for someone from Pittsburgh, home of Carnegie Mellon University, and is derived from *yinz*, which is Pittsburgh dialect equivalent to *y'all*.

> Yinzer offers its members online business social networking and job advertisement services in the Pittsburgh area. Members can add other members as business contacts, post advertisements for jobs, recommend a contact for a job, and receive email notifications about matching jobs.

Subsequent chapters cover other details on modeling and give advice about how to use models effectively.

7.1 Canonical model structure

Once you start building models, there are lots of bits and pieces to keep track of. If you see a UML class diagram for the Yinzer system that shows a Job Advertisement associated with a Company, you want to know what it represents: is it things from the real world, your design, or perhaps even your database schema? You need an organization that helps you sort those bits into the right places and to make sense of the whole thing.

The *canonical model structure* presented here provides you with a standard way to organize and relate the facts you encounter and the models you build. You will not always build models that cover the whole canonical model structure, but most projects over time will have bits and pieces of models that follow the canonical structure.

Overview

The essence of the *canonical model structure* is simple: Its models range from abstract to concrete, and it uses views to drill down into the details of each model.

There are three primary models: the domain model, the design model, and the code model, as seen in Figure 7.1. The canonical model structure has the most abstract model (the domain) at the top and the most concrete (the code) at the bottom. The *designation* and *refinement* relationships ensure that the models correspond, yet enable them to differ in their level of abstraction.

Each of the three primary models (the domain, design, and code models) are like databases in that they are comprehensive, but are usually too large and detailed to work on directly. (More on this shortly, in Section 7.4). *Views* allow you to select just a subset of the details from a model. For example, you can select just the details about a single component or just the dependencies between modules. You have no doubt worked with views before, such as a data dictionary or a system context diagram. Views allow you to relate these lists and diagrams back to the canonical model structure. Organizing the models in the canonical structure aids categorization and simplification.

The canonical model structure categorizes different kinds of facts into different models. Facts about the domain, design, and code go into their own models. When you encounter a domain fact like “billing cycles are 30 days”, a design fact like “font resources must always be explicitly de-allocated”, or an implementation fact like “the customer address is stored in a varchar(80) field”, it is easy to sort these details into an existing mental model.

The canonical model structure shrinks the size of each problem. When you want to reason about a domain problem you are undistracted by code details, and vice versa, which makes each easier to reason about.

Let’s first take a look at the domain, design, and code models before turning our attention to the relationships between them.

7.2 Domain, design, and code models

The *domain model* describes enduring truths about the domain; the *design model* describes the system you will build; and the *code model* describes the system source code. If something is “just true” then it probably goes in the domain model; if something is a design decision or a mechanism you design then it probably goes in the design model; and if something is written in a programming language, or is a model at that same level of abstraction, then it goes in the code model. Figure 7.1 shows the three models graphically and summarizes the contents of each.

Domain model. The domain model expresses enduring truths about the world that are relevant to your system. For the Yinzer system, some relevant truths would include definitions of important concepts like Ads and Contacts, relationships between those concepts, and behaviors that describe how the concepts and relationships change over time. In general, the domain is not under your control, so you cannot decide that weeks have six days or that you have a birthday party every week.

Design model. In contrast, the design is largely under your control. The system to be built does not appear in the domain model, but it makes its appearance in the design model. The design model is a partial set of design commitments. That is, you leave undecided some (usually low-level) details about how the design will work, deferring them until the code model.

The design model is composed of recursively nested *boundary models* and *internals models*. A boundary model and an internals model describe the same thing (like a component or a module), but the boundary model only mentions the publicly visible interface, while the internals model also describes the internal design.

Code model. The code model is either the source code implementation of the system or a model that is equivalent. It could be the actual Java code or the result of

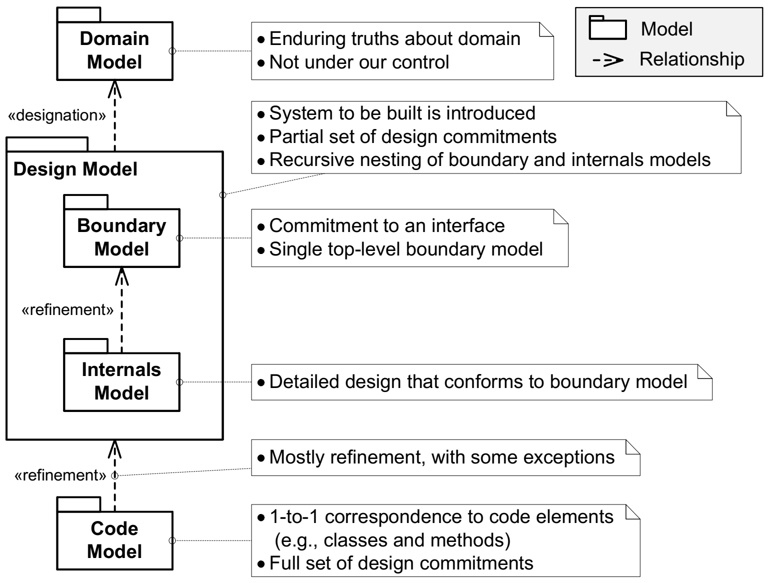
Domain Model
• Enduring truths about domain
• Not under our control
Model
Relationship
«designation»
Design Model
• System to be built is introduced
• Partial set of design commitments
• Recursive nesting of boundary and internals models
Boundary Model
• Commitment to an interface
• Single top-level boundary model
«refinement»
Internals Model
• Detailed design that conforms to boundary model
• Mostly refinement, with some exceptions
«refinement»
Code Model
• 1-to-1 correspondence to code elements (e.g., classes and methods)
• Full set of design commitments

Figure 7.1: The *canonical model structure* that organizes the domain, design, and code models. The design model contains a top-level boundary model and recursively nested internals models.

running a code-to-UML tool, but its important feature is that has a full set of design commitments.

Design models often omit descriptions of low-risk parts knowing that the design is sufficient so long as the developer understands the overall design and architecture. But where the design model has an incomplete set of design commitments, the code model has a complete set, or at least a sufficiently complete set to execute on a machine.

7.3 Designation and refinement relationships

You no doubt have an intuitive sense of how the domain relates to the design and how the design relates to the code. Because this chapter seeks to divide up models and relate them, it is a good idea to examine these relationships carefully so that you can fully understand them.

Designation. The *designation* relationship enables you to say that similar things in different models should correspond. Using the Yinzer example, the domain model describes domain truths, such as people building a network of contacts and companies posting ads. Using the designation relationship, these truths carry over into the design, as seen in Figure 7.2.

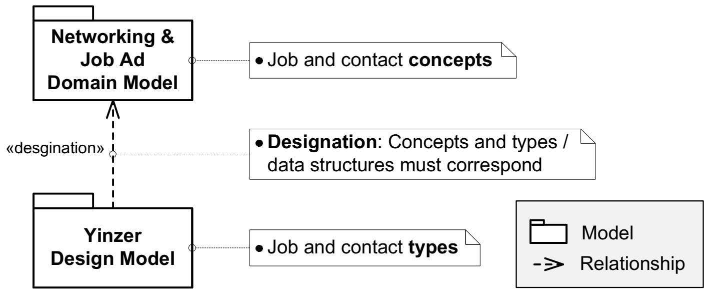
Networking & Job Ad Domain Model
• Job and contact **concepts**
«desgination»
• **Designation**: Concepts and types / data structures must correspond
Yinzer Design Model
• Job and contact **types**
Model
Relationship

Figure 7.2: The *designation relationship* ensures that concepts you choose from the domain correspond to types or data structures in your design.

You have leeway in your design but it should not violate domain truths. You can designate that selected concepts from the domain must correspond to types and data structures from the design. Things that you do not designate are unconstrained.

While in practice the designation relationship is rarely written down precisely, it would be a mapping that defined the correspondence between the domain elements (e.g., Advertisement and Job concepts) and the design elements (e.g., Advertisement and Job types and data structures).

Perhaps surprisingly, the design is rarely 100% consistent with the domain because systems often use a simplified or constrained version the domain concepts. For example, the system may not realize that the same person reads email at two different email addresses, and so might consider them two different people. Or the system may restrict domain concepts, such as limiting the number of contacts a person can have in the system. But when correspondence with the domain is broken, bugs often follow. The designation relationship is covered in more detail in Section 13.6.

Refinement. *Refinement* is a relationship between a low-detail and a high-detail model of the same thing. It is used to relate a boundary model with an internals model, since they are both models of the same thing, but vary in the details that they expose. Refinement is useful because it lets you decompose your design into smaller pieces. Perhaps the Yinzer system is made up of a client and a server piece, and the server is made up of several smaller pieces. Refinement can be used to assemble these parts into a whole, and vice versa. The mechanics of refinement are discussed in depth in Section 13.7.

Refinement is also used to relate the design model with the code model, but there it is not so straightforward. The structural elements in the design model map neatly to the structural elements in the code model. For example, a module in the design maps to packages in the code, and a component in the design maps to a set of classes in the code.

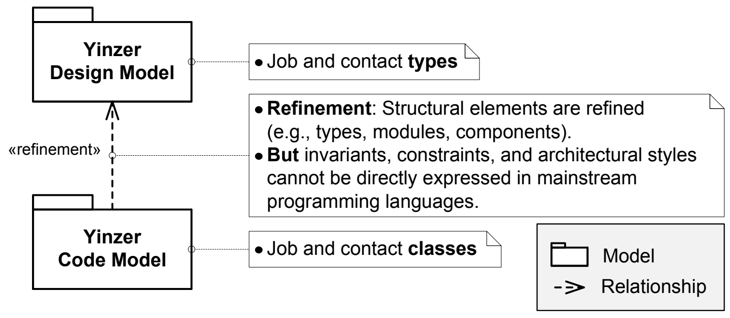
Yinzer Design Model
• Job and contact **types**
«refinement»
• **Refinement**: Structural elements are refined (e.g., types, modules, components).
• **But** invariants, constraints, and architectural styles cannot be directly expressed in mainstream programming languages.
Yinzer Code Model
• Job and contact **classes**
Model
Relationship

Figure 7.3: The *refinement relationship* ensures that concepts you choose from the domain correspond to types or data structures in your design. Be aware that there are elements in the design (invariants, constraints, styles) that cannot be expressed in programming languages.

However, as shown in Figure 7.3, other parts of the design model are absent in the code model: invariants, constraints, and architectural styles. Essentially no mainstream programming languages can directly express the constraints from the design model. It is true that constraints such as “all web requests must complete within 1 second”, or “adhere to the pipe-and-filter style” can be *respected* by the code but they cannot be directly *expressed*. This gap between design and code models is discussed in more depth in Section 10.1.

7.4 Views of a master model

In your head, you understand how any number of systems work and carry around models that describe them, such as models of your neighborhood or how you manage your household. From time to time, you sketch out excerpts of those models, such as a map for a friend showing him how to get to that great restaurant, or you write down a list of groceries. These excerpts are consistent with that comprehensive model from your head. For example, you could have written out a full map for your friend, but presumably the one you drew is accurate so far as it goes, and is sufficient to get him there. And your grocery list represents the difference between your eating plans and the contents of your refrigerator.

The domain, design, and code models are comprehensive models like these. They are jam-packed full of details since, conceptually at least, they contain everything that you know about those topics. It would be difficult or impossible to write down all those details, and even keeping them straight in your head is difficult. So, if you want to use a model to reason about security, scalability, or any other reason, you need to winnow down the details so that you can see the relevant factors clearly. This is done with *views*.

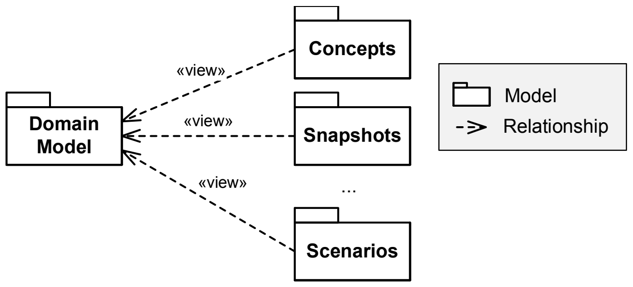

Figure 7.4: The *domain mode*l acts as a *master model* containing all details. Views show selected details from the master model. Because they are all views of the same master model, all of the views are *consistent* with each other.

Definition. A view, also called a *projection*, shows a defined subset of a model's details, possibly with a transformation. The domain, design, and code models each have many standard views. Views of the domain model include lists of concepts, lists of relationships between them, and scenarios that show how the concepts and relationships change over time (see Figure 7.4). Design views include the system context diagram and the deployment diagram. You can invent new views as appropriate.

Philippe Kruchten, in his paper on 4+1 views of architecture, showed that it is impractical to use a single diagram to express everything about your architecture and design (Kruchten, 1995). He explained that you need distinct architectural views because each has its own abstractions, notation, concerns, stakeholders, and patterns. Each view can use an appropriate notation and focus on a single concern, which makes it easy to understand. Together, the views comprise a full architecture model, and each view presents a subset of the details from that full model.

View consistency. Each view (or diagram) you create of a domain, design, or code model shows a single perspective on that model, exposing some details and hiding others. The diagrams are not isolated parts of the model, like drawers in a cabinet. Instead, they are live projections of the model and the views are consistent with each other. So if the model changes, the views do too. House blueprints are views of a house (or its design) so you expect them to be consistent with each other.

For example, imagine that you have two views of the domain model: a list of concepts in the networking and job advertisement domain (such as Ads, Jobs, and Contacts), and a scenario (a story) describing them. We will describe scenarios in more detail soon, but for now consider it a story told about how the domain concepts interact over time. If you were to revise the scenario to reference a new domain concept, like a declined invitation to join a contact network, you would expect to see that concept in the list of defined concepts. If it is not there, it is a bug in your domain model.

Master models. The domain, design, and code models are each conceptually a single *master model*. Every view you draw must be consistent with that master model. Think of it this way: when you revised the scenario to refer to a new concept, your understanding of the master model was revised. Since any other view is derived from the master model, it should reflect your new understanding. Disregarding pragmatics for a moment, all of the diagrams you build should be consistent at all times with each other because that is the way that you, in your mind, understand the domain to work. Pragmatically, however, while you are building models there will be times when they are inconsistent with each other, but you strive to eliminate those bugs.

To reinforce the idea of unified and consistent models, it may be helpful to imagine a programming environment where all of these elements fit together and are typechecked. In that programming environment, a scenario that tried to refer to a concept that was not defined in the master model would yield a type checking error.

Discussing views formally makes them sound difficult, but in reality people can use them with almost no effort. For example, you can imagine your bookcase as it is now, or imagine it with only the red books, or imagine it with the red books rotated so that you can see the cover instead of the spine. Each of these is a view of your master model of the bookcase. Notice that while you have never written down a model of your bookcase, you nonetheless have one in your head that you can manipulate. One of the challenges in software development is ensuring that developers, subject matter experts, and others all have the same master model in their heads.

Examples of master models. Master models are a helpful concept because they explain what your views refer to, but their abstractness can be confusing. The most straightforward example of a master model is an already existing system. You can create many views of that system, say your neighborhood. You do not have a complete model of your neighborhood written down anywhere, but you do have the neighborhood itself. Views of the neighborhood can be tested against the neighborhood to see if they are consistent with that master model.

Another example of a master model is a system that will be built. Unlike your neighborhood, this system does not yet exist, so it is a bit trickier to build views of it and ensure the views are consistent. Yet somehow things tend to work out OK. You might embark on a project to renovate a room in your house without writing down any explicit models, but you must have a master model in your head in some form. That model includes details about what should happen when (for example, demolition happens *before* painting) and cost estimates. That model in your head is likely incomplete, so views of it will necessarily be incomplete too.

Here are some concrete examples of master models of software systems. The master model may be the system you previously built or a system you plan to build. It can be a combination of the two, such as an existing system with planned additions. Or it could be even more complex, such as a model of the system as you expect it to look at three-month intervals over the next few years.

Limiting size and focusing attention. You use views in modeling to limit the size of the diagrams and to focus attention. Imagine how confusing a medium-sized domain model would be if you tried to show all the concepts, definitions, behaviors, etc., on the same diagram. You may have seen the giant printouts of corporate database schemas taped to a wall somewhere and seen people trying to use them by putting a finger in one place and tracing the lines to other parts of the diagram. Views avoid that.

7.5 Other ways to organize models

The canonical model structure from this chapter consists of a domain model, a design model, and a code model. This basic organization of models has a long history, visible in the Syntropy software development process (Cook and Daniels, 1994), though it probably traces back even further.

Other authors have proposed similar model structures, and while there are some differences in their organizations and nomenclature, there is a core similarity shared by all. With only a little bit of squinting, one can identify the domain, design (boundary and internals), and code models. Figure 7.5 is a summary that maps this book's model names to some of those found elsewhere.

Despite the broad strokes of similarity between authors, there are differences. The one concept that does not align well across authors is *requirements*, because it can mean different things to different people. Requirements models could overlap with business models, domain models, boundary models, or internals models.

7.6 Business modeling

There is a kind of model not found in this book's canonical model structure: *business models*. Business models describe what a business or organization does and why it does it. Different businesses in the same domain will have different strategies, capabilities, organizations, processes, and goals and therefore different business models.

Domain modeling is related to the field of *business modeling*, which includes not only facts but also decisions and goals that organizations must make. Someone at some point decides what the organization does and the processes it follows. Some of the processes are partly or fully automated with software. The goals and decisions of an organization can be changed and can be influenced by the software that you build and buy.

So why include domain models but not business models in this book? This book includes domain modeling because misunderstanding the domain is a common cause

	Business Model	**Domain Model**	**Design Model**		**Code Model**
			Boundary Model	**Internals Model**	
Bosch			System context	Component design	Code
Cheesman & Daniels		Business concept	Type specs	Component architecture	Code
D'Souza (MAp)	Business architec-ture	Domain	Blackbox	Whitebox	Code
Software Engineering Institute (SEI)			Requirements	Architecture	Code
Jackson		Domain	Domain + machine	Machine	
RUP	Business modeling	Business modeling	Requirements	Analysis & design	Code
Syntropy		Essential	Specification	Implementation	Code

Figure 7.5: A table summarizing the models proposed by various authors and how they map to the business, domain, design (boundary, internals), and code models found in this book.

of failure in IT projects. Misunderstanding business processes can also cause failures, but those are rarely engineering failures.

7.7 Use of UML

This book uses Unified Modeling Language (UML) notation because it is ubiquitous and its addition of architectural notation in UML 2.0 has brought it visually closer to special purpose architecture languages.

This book deviates from strict UML in a few places by augmenting UML diagram elements with line style and shading.

- In UML, connectors can be solid lines or ball-and-socket style. They are distinguished using stereotypes to indicate their types. In this book, connectors are shown using a variety of line styles, which is a more compact way to convey their types and can be less cluttered.

- In UML, a port's type is shown with a text label near it. This book uses that style, but it sometimes clutters the diagram, in which case ports are shaded and defined in a legend. Not all UML tools allow shading or coloring of ports.

Any remaining deviations from UML are inadvertent.

7.8 Conclusion

Once you begin to build models of your system, you realize that understanding and tracking lots of little models is hard, but building a single gigantic model is impractical. The strategy proposed in this chapter is to build small models that fit into a canonical model structure. If you understand the canonical structure then you will understand where each model fits in.

The first big idea was to use designation and refinement to create models that differ in their abstraction. The primary models are the domain model, design model, and code model, and they range from abstract to concrete. The second big idea was to use *views* to zoom in on the details of a model. Since the views are all projections of a single master model, their details are consistent (or are intended to be). In order to hierarchically nest design models, you use refinement to relate boundary and internals models.

Coaches see and understand more than rookies not because they have sharper eyes, but because they have a conceptual model that helps them categorize what they are seeing. This chapter describes the entire canonical model structure in detail, but do not let this alarm you. In practice you would rarely, if ever, create every possible model and view. Once you have internalized these ideas, they will help you to understand where a given detail, diagram, and model fits. As shown in the case study (Chapter 4) and the chapter on the risk-driven model (Chapter 3), following a risk-driven approach to architecture encourages you to build a subset of models, ones that help you reduce risks you have identified. This chapter, and subsequent ones, provides detailed descriptions to help you can internalize the models and thus be better at building software, not to encourage you towards analysis paralysis.

7.9 Further reading

This book is a synthesis of the architectural modeling approaches invented by other authors. It has three primary influences. The first is the work on modeling components in UML from D'Souza and Wills (1998) and Cheesman and Daniels (2000), which focus primarily on modeling functionality. The second is the quality attribute centric approach from the Software Engineering Institute (Bass, Clements and Kazman, 2003; Clements et al., 2010) and Carnegie Mellon University (Shaw and Garlan, 1996). The third is the agile software development community (Boehm and Turner, 2003; Ambler, 2002) which encourages efficient software development practices.

There are several good books that describe the general concepts of software architecture. Bass, Clements and Kazman (2003), describes a quality-attribute centric view of software architecture and provides case studies of applying their techniques. Taylor, Medvidović and Dashofy (2009) is a more modern treatment and is logically organized like a textbook. Shaw and Garlan (1996) is becoming dated but is the best book for understanding the promise of software architecture. Clements et al. (2010) is an excellent reference book for architecture concepts and notations (and also has a useful appendix on using UML as an architecture description language). These books rarely venture down into objects and design, but D'Souza and Wills (1998) and Cheesman and Daniels (2000) do, showing how architecture fits into object-oriented design.

Probably more than any other book, Bass, Clements and Kazman (2003) has shaped the way the field thinks about software architecture, shifting the focus away from functionality and towards quality attributes. It describes not only the theory but also processes for analyzing architectures and discovering quality attribute requirements. The book also contains a great discussion of the orthogonality of functionality and quality attributes.

Rozanski and Woods (2005) offer perhaps the most complete treatment of how to understand and use multiple views in software architecture. It also contains valuable checklists relating to several standard concerns.

The simplest pragmatic approach to component-based development is found in Cheesman and Daniels (2000). They lay out an organizational structure for models using UML and treat components as abstract data types with strict encapsulation boundaries. A similar approach, but with greater detail, is found in D'Souza and Wills (1998). Both emphasize detailed specifications, such as pre- and post-conditions, as a way to catch errors during design. This book de-emphasizes pre- and post-conditions because on most projects they are too expensive, but the mindset they encourage is excellent.

The best book at articulating a vision of software engineering that includes software architecture is probably Shaw and Garlan (1996). While reading it, it is difficult not to share their enthusiasm for how architecture can help our field.

The nuts and bolts of architectural modeling, including pitfalls, are well described by Clements et al. (2010). One of the book's goals is to teach readers how to document the models in a documentation package, which can be important on large projects.

To date, the most comprehensive treatment of software architecture is by Taylor, Medvidović and Dashofy (2009) in their textbook on software architecture. It covers real-world examples of software architecture as well as research developments on formalisms and analysis.

Developers working in the field of Information Technology (IT) will be well served by Ian Gorton's treatment of software architecture, as his book covers not only the basics of software architecture, but also the common technologies in IT, such as Enterprise Java Beans (EJB), Message-Oriented Middleware (MOM), and Service Oriented Architecture (SOA) (Gorton, 2006).

Using abstraction to organize a stack of models is an old technique. It is used in the Syntropy object oriented design method (Cook and Daniels, 1994) and is central to Cheesman and Daniels (2000), Fowler (2003a), and D'Souza and Wills (1998).

Many authors have suggested ways of organizing and relating architecture models. Jan Bosch models the system context, the archetypes, and the main components (Bosch, 2000). John Cheesman and John Daniels propose building a model of the requirements (a business concept model and a scenario model) and a model of the system specification (a business type model, interface specifications, component specifications, and the component architecture) (Cheesman and Daniels, 2000). Desmond D'Souza, in MAp, suggests modeling the business architecture, the domain, and the design as a blackbox and a whitebox (D'Souza, 2006). David Garlan conceives as architecture being a bridge between the requirements and the implementation (Garlan, 2003). Michael Jackson suggests modeling the domain, the domain with the machine, and the machine (Jackson, 1995). Jackson's primary focus is on system requirements engineering, not design, but his specifications overlap well with design. The Rational Unified Process (RUP) does not advocate specific models, but suggests activities for business modeling, requirements, and analysis & design (Kruchten, 2003).

Every developer should be familiar with the 4+1 architecture views paper (Kruchten, 1995), but also be aware that it is just one of many different sets of views that have been proposed for architecture, such as the Siemens Four Views (Hofmeister, Nord and Soni, 2000).

You should also be aware of the IEEE standard description of software architecture, IEEE 1471-2000 (Society, 2000). In it, you will find most of the same concepts as in this book. It has a few additions and differences worth noting. While it uses *views*, it treats them as requirements from the *viewpoint* of a *stakeholder* focused on a particular *concern*, rather than as projections of a consistent master model, what it would call an *architecture description*. It also describes the *environment* the system inhabits, its *mission*, and *library viewpoints* (which are reusable viewpoint definitions).

Authors are increasingly paying attention to business process modeling in addition to domain modeling. Martin Ould provides a practical process for modeling business processes (Ould, 1995). Desmond D'Souza describes how to connect business processes to software architecture by connecting business goals to system goals

(D'Souza, 2006).

The relationship between software architecture (specifically enterprise architecture) and business strategy is covered in Ross, Weill and Robertson (2006). As software developers, we perhaps assume that the natural future state should be that all systems can inter-operate. The surprising thesis of the book is that the level of integration should relate to the chosen business strategy.