Chapter 8
Documenting a Software Architecture

8.1 Introduction

Architecture documentation is often a thorny issue in IT projects. It's common for there to be little or no documentation covering the architecture in many projects. Sometimes, if there is some, it's out-of-date, inappropriate and basically not very useful.

At the other extreme there are projects that have masses of architecture related information captured in various documents and design tools. Sometimes this is invaluable, but at times it's out-of-date, inappropriate and not very useful!

Clearly then, experience tells us that documenting architectures is not a simple task. But there are many good reasons why we want to document our architectures, for example:

- Others can understand and evaluate the design. This includes any of the application stakeholders, but most commonly other members of the design and development team.
- We can understand the design when we return to it after a period of time.
- Others in the project team and development organization can learn from the architecture by digesting the thinking behind the design.
- We can do analysis on the design, perhaps to assess its likely performance, or to generate standard metrics like coupling and cohesion.

Documenting architectures is problematic though, because:

- There's no universally accepted architecture documentation standard.
- An architecture can be complex, and documenting it in a comprehensible manner is time consuming and nontrivial.
- An architecture has many possible views. Documenting all the potentially useful ones is time consuming and expensive.
- An architecture often evolves as the system is incrementally developed and more insights into the problem domain are gained. Keeping the architecture documents current is often an overlooked activity, especially with time and schedule pressures in a project.

I. Gorton, *Essential Software Architecture*,
DOI 10.1007/978-3-642-19176-3_8,

I'm pretty certain the predominant tools used for architecture documentation are Microsoft Word, Visio and PowerPoint, along with their non-Microsoft equivalents. And the most widely used design notation is informal "block and arrow" diagrams, just like we've used in this book so far, in fact. Both these facts are a bit of an indictment on the state of architecture documentation practices at present. We should be able to do better.

This chapter examines some of the most useful architecture views to document, and shows how the latest incarnation of the *Unified Modeling Language*, UML v2.0, can help with generating these views. Using these techniques and supporting tools, it's not overly difficult or expensive to generate useful and valuable documentation.

8.2 What to Document

Probably the most crucial element of the "what to document" equation is the complexity of the architecture being designed. A two-tier client server application with complex business logic may actually be quite simple architecturally. It might require no more than an overall "marketeture" diagram describing the main components, and a perhaps a structural view of the major components (maybe it uses a model-view-controller architecture) and a description of the database schema, no doubt generated automatically by database tools. This level of documentation is quick to produce and routine to describe.

Another factor to consider is the likely longevity of the application. Will the system serve a long-term business function, or is it being built to handle a one-off need for integration, or is it just a stop-gap until a full ERP package is installed? Projects with little prospect of a long life probably don't need a lot of documentation. Still, never let this be an excuse to hack together some code and throw good design practices to the wind. Sometimes these stop-gap systems have a habit of living for a lot longer than initially anticipated, and someone (maybe even you) might pay for these hacks 1 day.

The next factor to consider is the needs of the various project stakeholders. The architecture documentation serves an important communications role between the various members of the project team, including architects, designers, developers, testers, project management, customers, partner organizations, and so on. In a small team, interpersonal communication is often good, so that the documentation can be minimal, and maybe even maintained on a whiteboard or two using agile development techniques. In larger teams, and especially when groups are not colocated in the same offices or building, the architecture documentation becomes of vital importance for describing design elements such as:

- Component interfaces
- Subsystems constraints
- Test scenarios

- Third party component purchasing decisions
- Team structure and schedule dependencies
- External services to be offered by the application

So, there's no simple answer here. Documentation takes time to develop, and costs money. It's therefore important to think carefully about what documentation is going to be most useful within the project context, and produce and maintain this as key reference documents for the project.

8.3 UML 2.0

There's also the issue of how to document an architecture. So far in this book we've used simple box-and-arrow diagrams, with an appropriate diagram key to give a clear meaning to the notation used. This has been done deliberately, as in my experience, informal diagrammatical notations are the most common vehicle used to document IT application architectures.

There are of course many ways to describe the various architecture views that might be useful in a project. Fortunately for all of us, there's an excellent book that describes many of these from Paul Clements et al. (see Further Reading), so no attempt here will be made to replicate that. But there's been one significant development since that book was published, and that's the emergence of the Unified Modeling Language (UML) 2.0.

For all its greatly debated strengths and weaknesses, the UML has become the predominant software description language used across the whole range of software development domains. It has wide and now quality and low-cost tool support, and hence is easily accessible and useable for software architects, designers, developers, students – everyone in fact.

UML 2.0 is a major upgrade of the modeling language. It adds several new features and, significantly, it formalizes many aspects of the language. This formalization helps in two ways. For designers, it eliminates ambiguity from the models, helping to increase comprehensibility. Second, it supports the goal of model-driven development, in which UML models are used for code generation. There's also a lot of debate about the usefulness of model-driven development, and this topic is specifically covered in a later chapter, so we won't delve into it now.

The UML 2.0 modeling notations cover both structural and behavioral aspects of software systems. The structure diagrams define the static architecture of a model, and specifically are:

- *Class diagrams*: Show the classes in the system and their relationships.
- *Component diagrams*: Describe the relationship between components with well-defined interfaces. Components typically comprise multiple classes.
- *Package diagrams*: Divide the model into groups of elements and describe the dependencies between them at a high level.

- *Deployment diagrams*: Show how components and other software artifacts like processes are distributed to physical hardware.
- *Object diagrams*: Depict how objects are related and used at run-time. These are often called instance diagrams.
- *Composite Structure diagrams*: Show the internal structure of classes or components in terms of their composed objects and their relationships.

In contrast, behavior diagrams show the interactions and state changes that occur as elements in the model execute:

- *Activity diagrams*: Similar to flow charts, and used for defining program logic and business processes.
- *Communication diagrams*: Called collaboration diagrams in UML 1.x, they depict the sequence of calls between objects at run-time.
- *Sequence diagrams*: Often called swim-lane diagrams after their vertical timelines, they show the sequence of messages exchanged between objects.
- *State Machine diagrams*: Describe the internals of an object, showing its states and events, and conditions that cause state transitions.
- *Interaction Overview diagrams*: These are similar to activity diagrams, but can include other UML interaction diagrams as well as activities. They are intended to show control flow across a number of simpler scenarios.
- *Timing diagrams*: These essentially combine sequence and state diagrams to describe an object's various states over time and the messages that alter the object's state.
- *Use Case diagrams*: These capture interactions between the system and its environment, including users and other systems.

Clearly then, UML 2.0 is a large technical area in itself, and some pointers to good sources of information are provided at the end of this chapter. In the following sections though, we'll describe some of the most useful UML 2.0 models for representing software architectures.

8.4 Architecture Views

Let's return to the order processing example introduced in the previous chapter. Figure 7.9 shows an informal description of the architecture using a box and arrow notation. In Fig. 8.1, a UML component diagram is used to represent an equivalent structural view of the order processing system architecture. Note though, based on the evaluation in the previous chapter, a queue has been added to communicate between the *OrderProcessing* and *OrderSystem* components.

Only two of the components in the architecture require substantial new code to be created. The internal structure of the most complex of these, *OrderProcessing*, is shown in the class diagram in Fig. 8.2. It includes three classes that encapsulate each interaction with an existing system. No doubt other classes will be

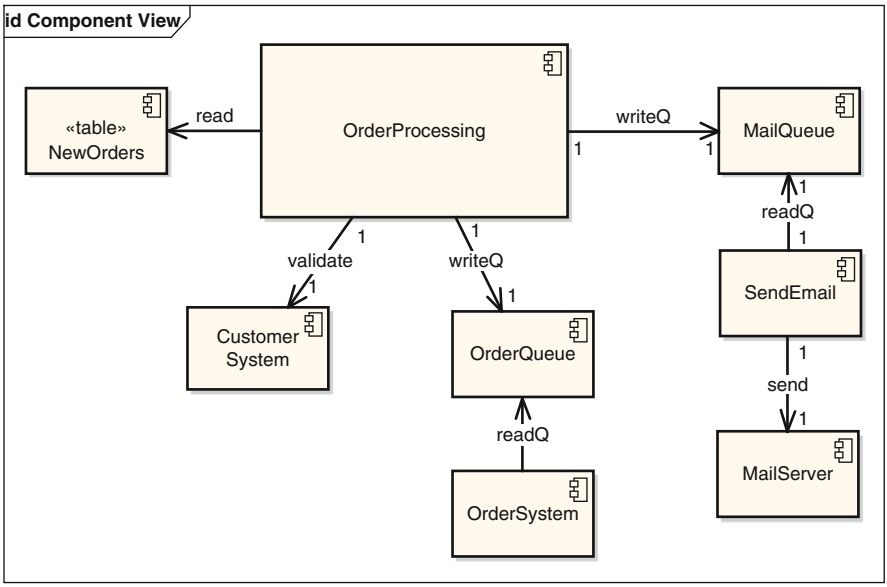

Fig. 8.1 A UML component diagram for the order processing example

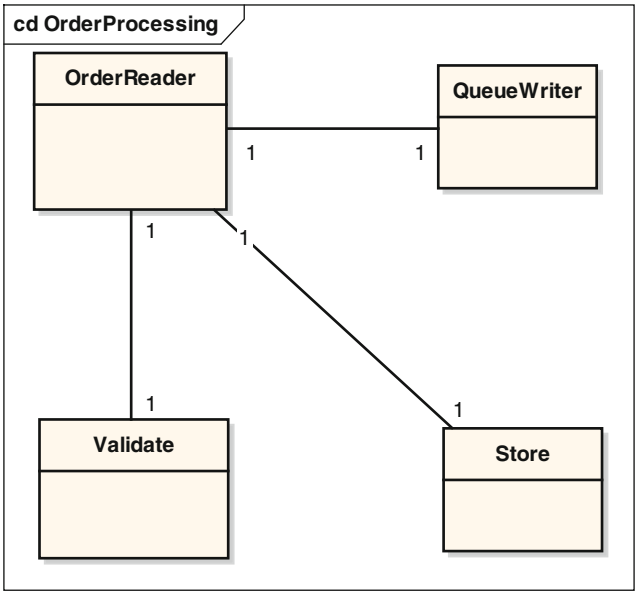

Fig. 8.2 Classes for the order processing component

introduced into the design as it is implemented, for example one to represent a new order, but these are not shown in the class diagram so that they do not clutter it with unnecessary detail. These are design details not necessary in an architecture description.

With this level of description, we can now create a sequence diagram showing the main interactions between the architectural elements. This is shown in Fig. 8.3, which uses the standard UML stereotypes for representing *Boundary* (*CustomerSystem*, *OrderQueue*, *MailQueue*) and *Entity* (*NewOrder*) components. This sequence diagram omits the behavior when a new order is invalid, and what happens once the messages have been placed on the *OrderQueue* and *MailQueue*. Again, this keeps the model uncluttered. Descriptions of this additional functionality could either be described in subsequent (very simple) sequence diagrams, or just in text accompanying the sequence diagram.

Sequence diagrams are probably the most useful technique in the UML for modeling the behavior of the components in an architecture. One of their strengths actually lies, somewhat ironically, in their inherent weakness in describing complex processing and logic. Although it is possible to represent loops and selection in sequence diagrams, they quickly become hard to understand and unwieldy to create. This encourages designers to keep them relatively simple, and focus on describing the major interactions between architecturally significant elements in the design.

Quite often in this style of business integration project, it's possible to create a UML deployment diagram showing where the various components will execute.

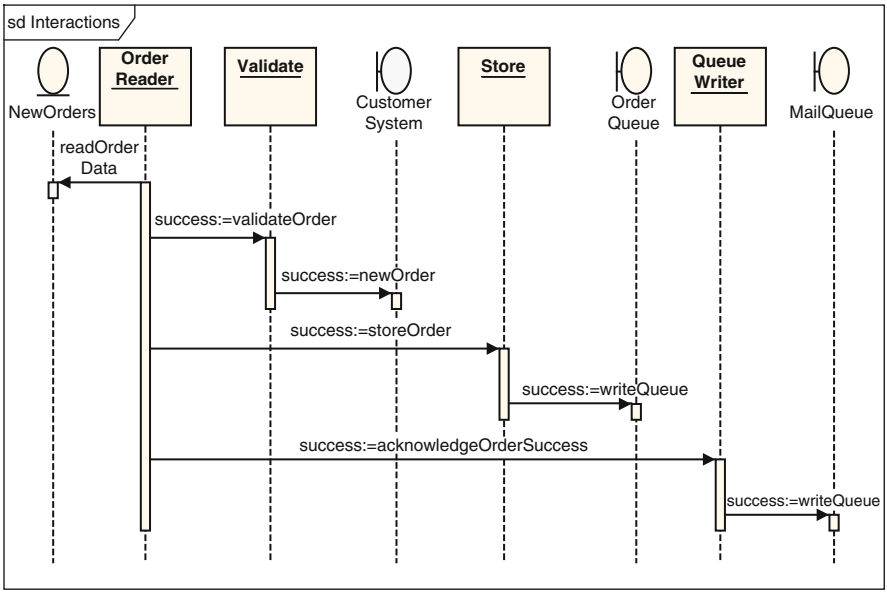

Fig. 8.3 Sequence diagram for the order processing system

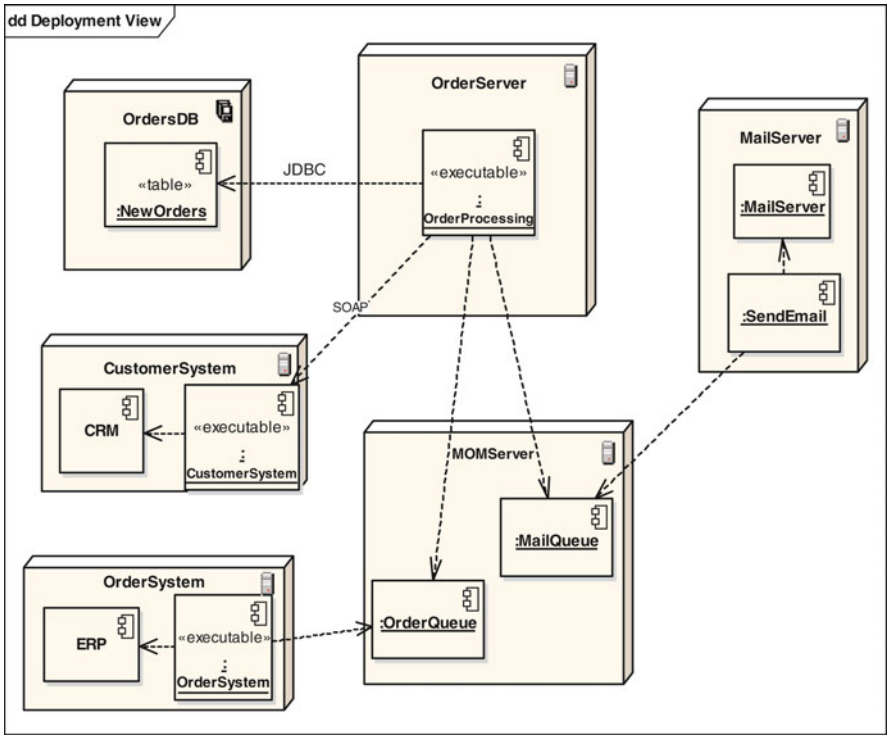

Fig. 8.4 UML Deployment diagram for the order processing system

This is because many of the components in the design already exist, and the architect must show how the new components interact with these in the deployment environment. An example of a UML deployment diagram for this example is given in Fig. 8.4. It allocates components to servers and shows the dependencies between the components. It's often useful to label the dependencies with a name that indicates the protocol that is used to communicate between the components. For example, the *OrderProcessing* executable component requires JDBC[1] to access the *NewOrders* table in the *OrdersDB* database.

8.5 More on Component Diagrams

Component diagrams are very useful for sketching out the structure of an application architecture. They clearly depict the major parts of the system, and can show which off-the-shelf technologies will be used as well as the new components that

[1]Java Database Connectivity.

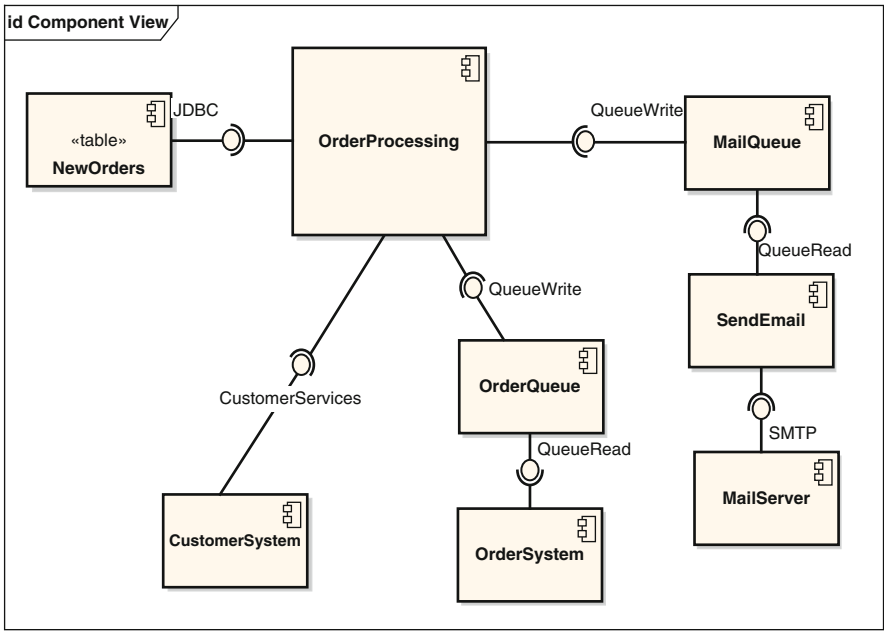

Fig. 8.5 Representing interfaces in the order processing example

need to be built. UML 2.0 has also introduced improved notations for representing component interfaces. An interface is a collection of methods that a component supports. In addition to the UML 1.x "lollipop" notation for representing an interface supported by a component (a "provided" interface), the "socket" notation can be used to specify that a component needs a particular interface to be supported by its environment (a "required" interface). These are illustrated in Fig. 8.5. Interface definition is particularly important in an architecture, as it allows independent teams of developers to design and build their components in isolation, ensuring that they support the contracts defined by their interfaces.

By connecting provided and required interfaces, components can be "plugged" or "wired" together, as shown in Fig. 8.5. The provided interfaces are named, and capture the dependencies between components. Interface names should correspond to those used by off-the-shelf applications in use, or existing home-grown component interfaces.

UML 2.0 makes it possible to refine interface definitions even further, and depict how they are supported within the context of a component. This is done by associating interfaces with "ports". Ports define a unique, optionally named interaction point between a component and its external environment. They are represented by small squares on the edge of the component, and have one or more provides or requires interfaces associated with them.

The order processing system architecture using ports for the *OrderProcessing* and *CustomerSystem* components is depicted in Fig. 8.6. All the ports in this design

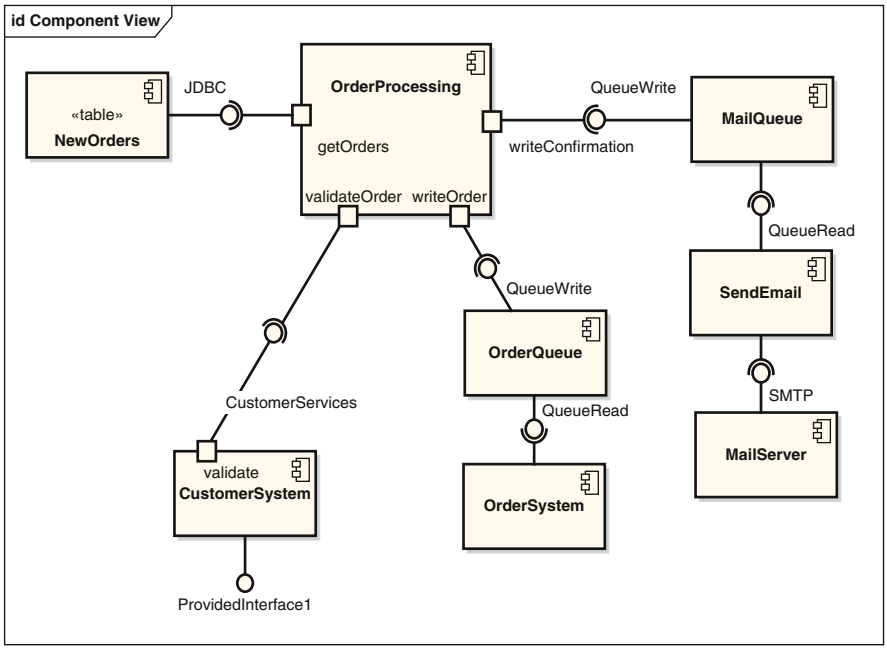

Fig. 8.6 Using ports in the order processing example

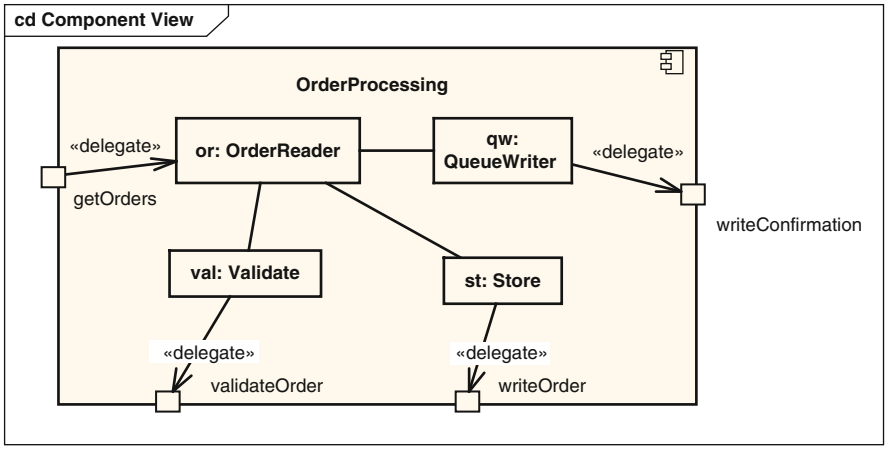

Fig. 8.7 Internal design of the *OrderProcessing* component

are unidirectional, but there is nothing stopping them from being bidirectional in terms of supporting one or more provides or requires interfaces. UML 2.0 composite diagrams enable us to show the internal structure of a design element such as a component. As shown in Fig. 8.7, we can explicitly depict which objects comprise

the component implementation, and how they are related to each other and to the ports the component supports. The internal objects are represented by UML 2.0 "parts". Parts are defined in UML 2.0 as run-time instances of classes that are owned by the containing class or component. Parts are linked by connectors and describe configurations of instances that are created within an instance of the containing component/class.

Composite diagrams are useful for describing the design of complex or important components in a design. For example, a layered architecture might describe each layer as a component that supports various ports/interfaces. Internally, a layer description can contain other components and parts that show how each port is supported. Components can also contain other components, so hierarchical architectures can be easily described. We'll see some of these design techniques in the case study in the next section.

8.6 Architecture Documentation Template

It's always useful for an organization to have a document template available for capturing project specific documentation. Templates reduce the start-up time for projects by providing ready-made document structures for project members to use.

Once the use of the templates becomes institutionalized, the familiarity gained with the document structure aids in the efficient capture of project design details.

Architecture Documentation Template
Project Name: XXX
1 Project Context
2 Architecture Requirements
- 2.1 Overview of Key Objectives
- 2.2 Architecture Use Cases
- 2.3 Stakeholder Architectural Requirements
- 2.4 Constraints
- 2.5 Non-functional Requirements
- 2.6 Risks

3 Solution
- 3.1 Relevant Architectural Patterns
- 3.2 Architecture Overview
- 3.3 Structural Views
- 3.4 Behavioral Views
- 3.5 Implementation Issues

4 Architecture Analysis
- 4.1 Scenario analysis
- 4.2 Risks

Fig. 8.8 Architecture documentation outline

Templates also help with the training of new staff as they tell developers what issues the organization requires them to consider and think about in the production of their system.

Figure 8.8 shows the headings structure for a documentation template that can be used for capturing an architecture design. To deploy this template in an organization, it should be accompanied by explanatory text and illustrations of what information is expected in each section. However, instead of doing that here, this template structure will be used to show the solution to the ICDE case study problem in the next chapter.

8.7 Summary and Further Reading

Generating architecture documentation is nearly always a good idea. The trick is to spend just enough effort to produce only documentation that will be useful for the project's various stakeholders. This takes some upfront planning and thinking. Once a documentation plan is established, team members should commit to keeping the documentation reasonably current, accurate and accessible.

I'm a bit of a supporter of using UML-based notations and tools for producing architecture documentation. The UML, especially with version 2.0, makes it pretty straightforward to document various structural and behavioral views of a design. Tools make creating the design quick and easy, and also make it possible to capture much of the design rationale, the design constraints, and other text based documentation within the tool repository. Once it's in the repository, generating design documentation becomes a simple task of selecting the correct menu item and opening up a browser or walking to the printer. Such automatic documentation production is a trick that is guaranteed to impress nontechnical stakeholders, and even sometimes the odd technical one!

In addition, it's possible to utilize UML 2.0 flexibly in a project. It can be used to sketch out an abstract architecture representation, purely for communication and documentation purposes. It can also be used to closely model the components and objects that will be realized in the actual implementation. This "closeness" can be reduced further in the extreme case to "exactness", in which elements in the model are used to generate executable code. If you're doing this, then you're doing so-called model-driven development (MDD).

There's all manner of debates raging about the worth and value of using the UML informally versus the precise usage required by MDD. Back in Chap. 1, the role of a software architecture as an abstract representation of the system was discussed. Abstraction is a powerful aid to understanding, and if our architecture representation is abstract, then it argues for a more informal usage of the UML in our design. On the other hand, if our UML models are a precise representation of our implementation, then they are hardly much of an abstraction. But such detailed models make code generation possible, and bridge the semantic gap between models and implementation. I personally think there's a place for both, it just

depends what you're building and why. Like many architecture decisions, there's no right or wrong answer, as solutions need to be evaluated in the context of their problem definition. Now there's a classic consultant's answer.

For in-depth discussions on architecture documentation approaches, the *Views & Beyond* book from the SEI is the current font of knowledge:

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, 2nd Edition, 2010

Good UML 2.0 books around. The one I find useful is:

S. W. Ambler. *The Object Primer 3rd Edition: Agile Model Driven Development with UML 2*. Cambridge University Press, 2004

This book also gives an excellent introduction into agile development methods, and how the UML can be used in lightweight and effective ways.

There's an IEEE standard, IEEE 1471-2000, for architecture documentation that is well worth a read if you're looking at defining architecture documentation standards for your organization. This can be found at:

http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html

An emerging area of research is the architecture knowledge management, aiming at capturing design rationale and "tribal knowledge" that is inevitably associated with any long-lived software system. Here's an excellent book that will give you pointers to the emerging technologies and practices in this area:

Ali Babar, M.; Dingsøyr, T.; Lago, P.; Vliet, H. van (Eds.), *Software Architecture Knowledge Management: Theory and Practice*, Springer-Verlag 2008