THE PROCESS OF

SOFTWARE ARCHITECTING

PETER EELES
PETER CRIPPS

FOREWORD BY GRADY BOOCH

Praise for *The Process of Software Architecting*

"The role of software architect has really come into its own in the last few years and is now recognized as a key determinant of project success. However, even today there is little common understanding of how to do the job: analyze requirements, understand concerns, evaluate alternatives, and construct and document an architectural description that is fit for purpose. Eeles and Cripps have filled this gap in their useful and practical book. The material is clearly and simply laid out, and follows a logical progression from inception through delivery, with tasks and work products clearly explained and illustrated using a real-world case study. This is a vital handbook for everyone from brand new architects to seasoned professionals."

—Nick Rozanski, coauthor of Software Systems Architecture

"If you need a thorough and authoritative reference for a complete software architecture process, then look no further. Peter Eeles and Peter Cripps have produced a definitive guide and reference to just such a process. Being precisely defined using a metamodel, illustrated with a realistic case study, and clearly related to key standards such as UML, RUP, and IEEE 1471, the process presented in this book provides a valuable guide for those developing software architectures for large projects. I have no doubt that it will become a valued reference for many software architects."

—Eoin Woods, coauthor of Software Systems Architecture

"Eeles and Cripps distill years of experience into a single guide that helps the reader understand not just what architects produce, but *how* they produce it. *The Process of Software Architecting* is a very practical guide filled with lessons learned and pitfalls to avoid. Practicing architects will want to refer to it as they hone their skills, and aspiring architects will gain key insights that otherwise could take painful years of experience to acquire."

—Bob Kitzberger, program director, Strategy, IBM Software Group

"For most of my career in this field, software architecture has had the feel of being a black art that only a select few gurus and wizards had a talent for. This book follows on from industry best practice and a wealth of author experience to bring solutions architecture into the realms of being a true engineering discipline. Now I have a real guide I can pass on to new practitioners, a guide that embodies what used to require years of trial and error."

—Colin Renouf, enterprise architect and technology author,
vice chairman, WebSphere User Group, UK

work is frequently based on one or more patterns and is often documented in a similar manner.

The term *application framework* should not be confused with *architecture description framework* (discussed in Chapter 4, "Documenting a Software Architecture"), which is a standard for describing a software architecture based on a set of viewpoints.

Component Library/Component

In the most general sense, several asset types considered here represent *components* of a solution, including existing applications and packaged applications. In the context of reusable assets, however, we use the term *component* to represent something that has a complete implementation but is finer-grained than a complete application.

Even so, such components can vary widely in granularity. A component may represent a significant element in the architecture, such as a message broker. It also could represent a service in SOA or a technology-specific component such as an Enterprise JavaBean. A widely understood form of reuse is with respect to even finer-grained elements, such as a user interface widget representing a table.

Components also may be provided in the form of a component library, class library, or procedure library. An example of a component library is the Java class library.

Attributes of an Architecture Asset

As you will see, assets can be qualified with several attributes. As an example, Figure 5.7 shows a subset of the architecture assets discussed earlier in this chapter characterized by two attributes: granularity and articulation (the level of implementation). Consumers use these attributes when trying to locate an asset for any given situation.

As shown in this figure, assets can be considered in terms of their granularity (size) and their level of articulation (implementation). Granularity is related to both the number of elements that comprise the asset and the asset's impact on the overall architecture. Articulation is concerned with the extent to which the asset can be considered to be complete and may be one of the following:

- **Specification**. Such assets have no implementation and are represented in an abstract form, such as a model or document. Such assets include various types of pattern, architectural styles, and reference models.

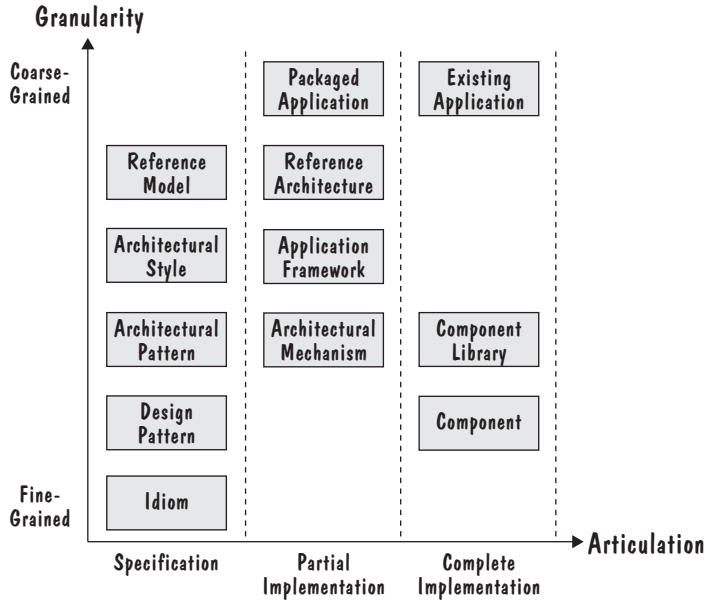

Figure 5.7 Using Attributes to Characterize Asset Types

- **Partial Implementation**. Such assets are considered to be partial implementations but require additional elements before they can be instantiated. Such assets include application frameworks and packaged applications.
- **Complete Implementation**. Such assets are considered to be complete implementations and can be instantiated as is, without modification. Such assets include components and existing applications.

This is all well and good, but granularity and articulation are just two attributes that may be of interest. If the purpose of assigning attributes to particular types of assets is to help a potential consumer locate the asset, you may want to consider many more attributes. A basic set of attributes that are relevant to any asset includes

- **Articulation**. The extent to which the asset provides a concrete implementation
- **Author**. The person who originally created the asset
- **Concerns addressed**. The concerns that the asset is aimed at addressing
- **Contained artifacts**. The physical files that comprise the asset

- **Granularity**. An indication of the size of the asset in relation to other assets
- **Name**. The name of the asset
- **Prerequisites**. The conditions that must apply before the asset can be used
- **Related assets**. Any assets that are related to this asset and the nature of these relationships
- **State**. The current state of the asset in terms of any asset life cycle (such as submitted or approved)
- **Type**. The asset type (such as a reference architecture or design pattern)
- **Use instructions**. The instructions for using the asset
- **Variability**. The extent to which the asset may be varied (such as none, limited, or extensive)
- **Version**. The asset version number

Attributes that result from the use of the asset include

- **Feedback**. Any feedback comments associated with the asset
- **Rating**. An indication of the rating associated with the asset
- **Usage count**. An indication of the number of times the asset has been requested

Attributes related to the context within which the asset is applied include

- **Business domain**. The domain to which the asset applies (such as telecoms or financial services)
- **Development discipline**. The discipline to which the asset applies (such as requirements or architecture)
- **Development method**. The method to which the asset applies (such as Scrum)
- **Development phase**. The development phase to which the asset applies (such as Inception)
- **Development scope**. The scope of the asset from a development perspective (such as software engineering or enterprise architecture)
- **Technical domain**. The technical (rather than business) domain to which the asset applies (such embedded systems)

In addition, a variety of non-functional properties may be associated with the asset: cost, performance, scalability, and the like.

Other Reuse Considerations

The past few years have seen renewed focus on reusable assets as a means of improving project performance and decreasing time to value. The seemingly widespread acceptance of SOA, for example, has led to renewed focus on how the associated reusable assets (*services*, in this case) are identified, specified, realized, deployed, managed, supported, maintained, upgraded, and (ultimately) retired. In more general terms, there are considerations related to a strategic approach to reuse. Unfortunately, a detailed discussion of such topics warrants a book of its own, although we do discuss a system-of-systems approach to the identification of large-scale reusable assets in Chapter 10, "Beyond the Basics."

Underpinning any reuse initiative is a means of describing reusable assets, which is where the Reusable Asset Specification (RAS) comes in (RAS 2004). This OMG standard fundamentally defines two aspects of a reuse strategy. The first aspect is a standard for describing a reusable asset. The second aspect is a standard that defines an interface to a RAS-compliant repository (referred to as a *RAS repository service*). Any organization that wants to be consistent in its handling of reusable assets would do well to look at the RAS and reuse repositories that implement this standard.

Achieving successful asset reuse involves more than just resolving the purely technical challenges, of course. Specifically, we've not gone into any detail regarding the roles that may exist in an organization in support of a strategic reuse initiative, the tasks that are performed (especially around the creation, use, and maintenance of assets), and any specific work products that specifically support a reuse initiative (such as an asset catalog). There are also broader implications at an organizational level, such as the embedding of a reuse culture and assessing what level of reuse culture is practical or beneficial to the organization.

Summary

Many types of reusable assets are at the disposal of software architects, and the use of such assets can improve project performance dramatically when the assets are applied correctly. This chapter provided an overview of commonly encountered assets available to architects in their work, as well as a discussion of asset attributes.

We consider the application of reusable assets in the case study–related chapters (Chapters 6 through 9).

Chapter 6

Introduction to the Case Study

The purpose of this chapter is to introduce the case study used as an example in Chapters 7 through 9. The detailed requirements and the corresponding solution, with an emphasis on the architectural elements, are progressively introduced and discussed in these chapters. This chapter sets the context and describes the high-level requirements for the solution. It contains information commonly found in work products, such as a business case, a vision document, or a business model. This information is input to several tasks undertaken during the definition of the detailed requirements, as you will see in Chapter 7, "Defining the Requirements."

Applying the Process

Before discussing the case study, we should first make some general comments regarding the application of the elements described in this chapter and in Chapters 7 through 9. As we mention at the very start of this book, what you have in your hands is an articulation of a series of practices that you can apply in many ways and on many types of projects. This section (and the equivalent section in Chapters 7 through 9) provides guidance on their application.

> A *practice* is an approach to solving one or several commonly occurring problems. Practices are intended as "chunks" of process for adoption, enablement, and configuration. (RUP 2008)

Given the discussion in Chapter 3, "Method Fundamentals," we could more correctly say that a practice is a chunk of *method*, because a method has two dimensions: method content and process, both of which are configurable. We said in Chapter 3 that *method content* describes the life cycle–independent elements, such as roles, tasks, and work products. A *process* then takes these elements and defines a sequence in which they are applied and considers concepts such as iterations.

We discuss each of these dimensions in Chapters 7 through 9. We can make some general statements, however, regarding the application of the various method elements described in these chapters (roles, tasks, work products, iterations, and the like). In particular, the content in these chapters represents our attempt to strike a balance between an extremely simple project that executes in weeks or months and a very large and complex project that executes in years. The various elements described in this book, therefore, need to be *right-sized* for your development project. Put another way, you need to consider the level of ceremony that needs to be applied. For the purposes of exemplification, the case study applies all the elements described in this book—no more and no less.

The first elements that need to be right-sized are those that comprise the *method content*: the roles, tasks, and work products. The obvious question that must be asked is "How much of each of these items do we need?" The answer, of course, is "It depends." A general comment is that *all elements should be considered* and then right-sized. Examples of right-sizing for two fictitious projects, small and large, are given in Table 6.1. In these examples, we consider the selection of viewpoints, which is discussed further in the sidebar "Best Practice: Select the Relevant Set of Viewpoints."

Similarly, the *process* elements within the method—phases, iterations, and the like—also need to be right-sized. Some of the items you need to consider include

- The definition of any phases in the process and any milestones associated with them.
- The emphasis applied to each of the architecting tasks within an iteration, based on where the iteration is in the overall life cycle and the phase within which the iteration occurs.
- The number of times a given task is performed within an iteration. It may make perfect sense to apply a task more than once.
- The sequencing of tasks within an iteration. (The sequencing suggested in this book is just that—a suggestion.)