1

Problem to Attack

Component-based development is a widely used approach to build complex systems. Basically, you allocate requirements to components of some kind—classes, packages, services, and so forth. Although many requirements can be effectively localized to individual components, you find many requirements that cannot be localized to an individual component and that sometimes even impact many components. In aspect-speak, these requirements cut across components and are called *crosscutting concerns*. The inability to keep such concerns separate during design and implementation makes a system difficult to understand and maintain. It inhibits parallel development and makes a system difficult to extend and results in many of the problems that plague so many projects today. A successful solution to this problem involves two things: an engineering technique to separate such concerns from requirements all the way to code and a composition mechanism to merge the design and implementation for each concern to result in the desired system. With aspect orientation under the guidance of an appropriate methodology, you do have such a solution today.

1.1 The Use of Components Today

Software systems are important to businesses today. Most, if not all, businesses today cannot run without the help of software to conduct business operations. As we all know, software systems are complex and when we design such complex systems our limited minds cannot possibly consider everything and solve everything at once.

Our natural inclination is to break the problem into smaller parts and solve them one by one. This is why we have components. Each component plays its specific role and has specific responsibilities and purposes. We assemble components of various kinds to form the complete system. This is basically how we develop any kind of product: electronic devices, cars, and more.

In a generic sense, components are elements that conform to well-defined interfaces, and if you invoke them through their interfaces, they produce some well-defined responses. For example, a computer chip is a component. It has pins through which you can send electric signals. Upon receiving the signal, the chip performs some actions and possibly returns some response through some other pins. Your video projector is also a component. If you plug in a video cable from your laptop to the projector, you can make images appear on the wall.

A component encapsulates its contents. Its internals are all hidden from you. As a user of the component, you do not need to know how it really works on the inside. All you need to know is that you send the correct signals to it through its interfaces in some acceptable sequence and you get your desired response. This characteristic of components is very attractive because as long as the interfaces do not change, you can replace them with other components that conform to those same interfaces. This substitutability is extremely useful if you want to extend the system with some new capabilities—all you need to do is replace an existing component with a better one that conforms to the same interface. Even if you have to change an interface, you may delimit the changes to a few components. It allows you to gracefully grow a complex system.

1.1.1 Building a System with Components

The usual approach to building systems in terms of components is as follows: You begin by first understanding what the system is supposed to do: What are the stakeholder concerns? What are the requirements? Next, you explore and identify the parts (i.e., components) that will constitute the system. You then map the world of requirements to the world of components. This is an M-to-N mapping and, normally, M is much larger than N. For example, you might have 1,000 requirements and maybe 50 components. The common approach to mapping is as follows: You identify a set of candidate components and check that each requirement will be implemented with these components. In this process, you may learn more about the requirements and, provided that the requirements are not too critical, change them so that they are easier to implement. Alternately, you might modify components to improve the fit. Once the required set of components is found, they are connected to form the desired system.

Figure 1-1 shows the components for a Hotel Management System. We use this system as an example here and throughout the rest of the book. Briefly, this system provides the functionalities to Reserve Room, Check In, and Check Out to be used by both customers and hotel staff.

Figure 1-1 shows components of various kinds. The Customer Screen and Staff Screen components deal with presenting information to the users and accepting and validating input from them. The Reserve Room, Check In, and Check Out components encapsulate the business and control logic for the respective functionalities. The reservation and the room components maintain information in a data store. This separation of roles and responsibilities across components is essential to a system that is resilient—one that will not easily break when changes are introduced.

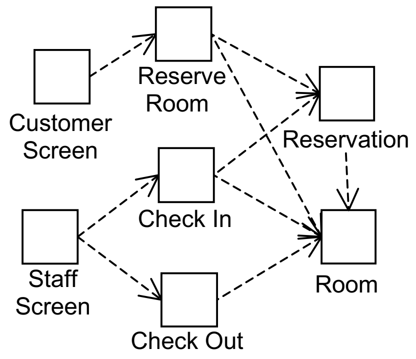

Figure 1-1 *Hotel Management System made up of interconnected components.*

1.1.2 Benefits of Components

Components are useful and important because they represent the static structure of a system such as that depicted in Figure 1-1. They are essential to understanding, designing, implementing, distributing, testing, and configuring the system. They are the most important asset for reuse in practice. Components contain things that change together. They keep concerns about a kind of object or an abstraction of a real-world phenomena separate.

For instance, a component (e.g., the Room and Reservation components in Figure 1-1) may encapsulate the manipulation of data structures capturing room and reservation information. If you change the data structure, you change the operations that touch these data. There are also components encapsulating the specifics of user interfaces. If you want to change the look and feel of the system, you simply change the screen components. Thus, you see that components make a system resilient to changes as you add new requirements to the system.

You can also meet customers' new demands by configuring the system using the components. New features or use-cases are usually provided by adding a new component and changing some already existing components.

In recent years, component frameworks such as J2EE and .Net have evolved and gained widespread popularity. Basically, all new software being developed is componentized.

1.2 Limitation of Components

To understand the limitations of components, we start with concerns. The goal of a system is to meet requirements or, more generally, concerns. A concern is anything that is of interest to a stakeholder, whether an end user, project sponsor, or developer. For example, a concern can be a functional requirement, a nonfunctional requirement, or a design constraint on the system. It can be more than a requirement of the system. It can even be a low-level concern such as caching or buffering.

Breaking down a problem into smaller parts is called *separation of concerns* in computer science. Ideally, we want to be able to cleanly separate the different concerns into modules of some kind and explore and develop each in isolation, one at a time. Thereafter, you compose these software modules to yield the complete system. Thus, the concept of separation of concerns and the concept of modularity are two sides of a coin—you separate concerns into modules, and each module solves or implements some distinct set of concerns.

Successful separation of concerns must start early. You begin software development by attempting to understand the stakeholder concerns. You explore and collect the requirements for the system according to stakeholder concerns. Although some concerns can be realized by distinct and separate components, in general, you find many concerns for which components are not adequate. These are known as *crosscutting concerns*—concerns that impact multiple components. There are different kinds of crosscutting concerns: *infrastructure* concerns are crosscutting concerns to meet nonfunctional requirements—for instance, logging, distribution, and transaction management. Some crosscutting concerns deal with functional requirements as well. You frequently find that the realization of functional requirements (which can be specified as use-cases) cut across multiple components. Thus, even use-cases are crosscutting concerns.

Sidebar 1-1 How Does Aspect Orientation Impact Object Orientation?

Aspect orientation is established precisely to overcome the limitation of object orientation. Conventional modularity such as classes and services suffer from their inability to keep crosscutting concerns separate. It does not matter whether or not you are implementing your system using object-oriented programming languages: they are all inadequate in dealing with crosscutting concerns.

As we write this book, we find that having to list all the conventional modules (components, classes, services, etc.) every time we talk about their limitations can be quite lengthy. So, for brevity, we simply use the term *components* as a representative of conventional modularity. So, when we say a "crosscutting concern can cut across classes," it applies to components as well.

1.2.1 Inability to Keep Peers Separate

We particularly want to highlight two kinds of crosscutting concerns. The first is what we call *peers*. These are concerns that are distinct from each other. No one peer is more important than another. If you consider the familiar ATM example, cash withdrawal, fund transfer, and cash deposit are all peers. In our hotel management example, Reserve Room, Check In Customer, and Check Out Customer are peers. These concerns do not need each other to exist. In fact, you can build separate systems for each one. However, when you start to implement peers in the same system, you find significant overlap between them. This is illustrated in Figure 1-2.

Figure 1-2 depicts concerns in different shades on the left-hand side. The right-hand side shows the components with multiple shades. Each shade represents the codes that implement the respective concerns. The limitation of components to keep peers separate is evident in Figure 1-2. It results in two effects, which in aspect-speak are known as *tangling* and *scattering*.

Tangling. You find that each component contains the implementation (i.e., code) to satisfy different concerns. For example, in Figure 1-2, you see that the Room component is involved in the realization of three different concerns. This means that as a developer/owner of a component, you need to understand a diverse set of concerns. The component, instead of single-mindedly fulfilling a particular concern, participates in many. This hinders understandability and makes the learning curve steeper for developers.

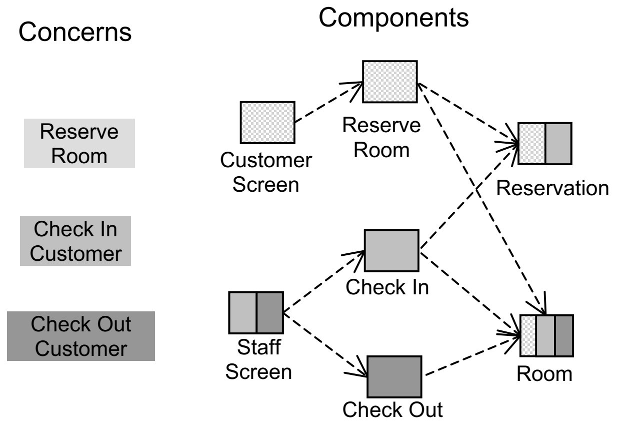

Figure 1-2 *Tangling and scattering when realizing peers.*

Do not confuse tangling with reuse. Reuse implies that the same code or behaviors are useable under different contexts. Definitely, some parts of the Room component will be reusable without changes. However, in many cases, as highlighted in Figure 1-2, each concern demands additional and distinct behaviors on the Room component not needed to realize other concerns. There is no reuse among them, and they result in tangling.

Scattering. You also find codes that realize a particular concern are spread across multiple components. For example, in Figure 1-2, you see that the realization of Check In Customer imposes additional behaviors on four components. So, if ever the requirements about that concern change, or if the design of that concern changes, you must update many components.

More importantly, scattering means that it is not easy to understand the internals of a system. For instance, it is not easy to uncover requirements by reading the source code of each component or a set of components. If the requirement for a particular concern changes, different classes need to be updated as well. Poor understandability leads to poor maintainability, and it is not easy to make enhancements, especially for large systems.

1.2.2 Inability to Keep Extensions Separate

The second kind of crosscutting concern is what we call *extensions*. Extensions are components that you define on top of a base. They represent additional service or features. For example, the Hotel Management System has a waiting list for room reservations. If there are no rooms, the system puts the customer on a waiting list. Thus, the provision of a waiting list is an extension of Reserve Room. Keeping extensions separate is a technique to make a complex problem understandable. You do not want to be entangled by too many issues, so you keep them separate as extensions.

Although it is natural to describe the base and extension separately, there is a problem when it comes to implementing the extension, as exemplified in Figure 1-3.

Figure 1-3 shows the Reserve Room component, which serves as the base. To incorporate the Waiting List extension, a corresponding component is added (shown in a darker shade). But in addition, you need to add some code fragments in the Reserve Room component at a particular location,

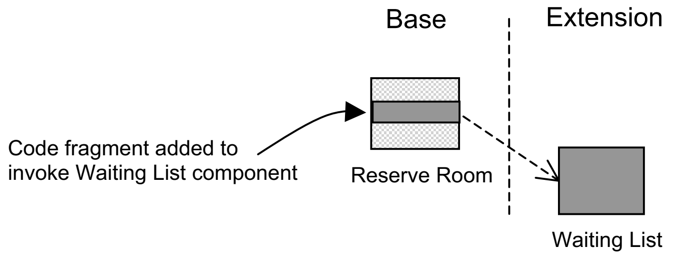

Figure 1-3 *Extensions inserted intrusively.*

which we call an extension point. The purpose of this code fragment is to connect or invoke the Waiting List component.

The problem is this: some code has been added to a place that didn't really need it before we added the new feature. It is there for the purpose of hooking the new component onto the existing component. This code fragment is affectionately known as *glue code*. In aspect-speak, such a change is known as *intrusive*.

No matter how good your design is, you still need glue code, and if you need to extend the system at another location, you must add the glue code there too. For example, if you need to support different payment methods for the Hotel Reservation System, you need additional glue code to open up an extension point in the system.

Adding all this glue code and making all these changes to existing code definitely makes the original classes harder to comprehend. But a greater problem exists: you cannot possibly identify all the extension points a priori. Thus, what is really needed is a way for you to designate extension points on demand at any time during the system's life cycle. Although this is a significant advantage, there is a limit to how far you can go. If a system is poorly designed, designating extension points is definitely not easy. In addition, after adding several enhancements, you have a better picture of the whole system and you might want to separate concerns differently. In this case, you might dispense some effort to refine the base.

Jacobson_Ch01.fm Page 11 Wednesday, December 8, 2004 3:16 PM

Sidebar 1-2 The Difference Between Concerns and Requirements

You might be wondering what the difference between a concern and a requirement is. They are not the same. Developing a system involves specifying requirements, which are then refined into design and subsequently to implementation. So, requirements are only part of the software development life cycle. A concern represents something of importance to some stakeholder, and it encompasses everything: you must specify concerns, design them, and implement them. So, requirements are simply for specifying concerns.

In general, for each concern, you will have many requirement statements to clarify what the concern is. For example, the Reserve Room functionality is a concern. There will be many requirement statements because the system deals with the Reservation of kinds of Rooms, different Reservation schemes, and so on.

In addition to specifying the concern, you must design and implement it. When we talk about separating concerns, we mean separating at requirements time and keeping the separation during design and implementation.

1.3 Approaching a Solution

So, you find that even though components are excellent tools to structure a complex system in some hierarchical fashion, they are nevertheless insufficient. Components cannot keep crosscutting concerns separate all the way down to code. Adding a new concern (a set of requirements) to the system becomes very painful.

The search is on for a new kind of modularity, one that can keep crosscutting concerns separate throughout the life cycle of the module—from requirements to analysis, design, code, and test. To achieve this modularity, you must also have a corresponding way to integrate or compose the new modularity into a coherent whole to get executable code. The new modularity must also help you collect all work on a concern, not just the code, but requirements, analysis, design, implementation, and test for that concern.

To achieve this new modularity, you need two things: a concern separation technique and a concern composition technique.

Concern Separation Technique. In order to keep concerns separate, you must model and structure concerns. The use-case technique is quite helpful in modeling concerns during requirements and analysis. We discuss use-cases in greater detail in Chapter 3, "Today with Use-cases," and in further depth in Part 2 of the book. Separating peer use-cases is easy (that is how use-cases are normally defined). Separating extension use-cases requires new language constructs. On top of that, you also need techniques to preserve the separation during design and implementation.

Concern Composition Mechanism. At some point in time, you need to compose the concerns. This can happen during compile time, post-compilation time, or even during execution. Composing normal extensions is relatively easy, since all that is needed is some automated way to monitor the execution of the base and execute the extension when required.

Composing peer use-cases is much harder—you must deal with overlapping behavior, conflicts, and other problems. Thus, the early efforts have been to keep extensions separate. In the next section, we discuss some of these early efforts to highlight that crosscutting concerns is not a new problem. Keeping extensions separate is also not a technique that is invented only recently. But certainly, aspect orientation (which we discuss in Chapter 2, "Attacking the Problem with Aspects") provides an elegant solution and hence a renewed interest in the problem of dealing with crosscutting concerns. Another reason we highlight earlier works is to show that aspect thinking is very much in line with use-case thinking and, hence, use-case–driven development is a strong candidate for conducting aspect-oriented software development.

1.3.1 Early Support for Extensions

The idea of keeping extensions separate dates back a long way and appeared in a 1986 paper discussing "Language Support for Changeable Large Real-Time Systems" [Jacobson 1986]. Jacobson introduced several terms in that paper; see Figure 1-4. The original program, that is, the base, is termed an *existion.* The new functionalities to be added to the existion are termed *extensions.* Extensions are inserted at designated execution points in the existion. These execution points are known as *extension points.*

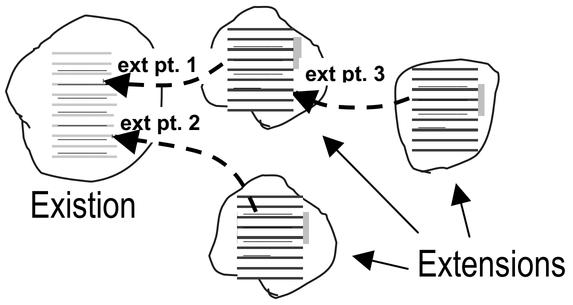

Figure 1-4 *Existions, extension points, and extensions.*

The key idea behind the approach is that you insert extensions into the existions during compilation or execution—not during coding. Thus, the source code of the existing system and even possibly its binaries remain clean and free of any extensions.

Structuring a system as shown in Figure 1-4 has several advantages. First, it makes extending an existing system a lot easier. When you want to introduce an extension, all you need to do is designate the extension point where the extension needs to be inserted. But this is no excuse for poor programming and design practices. Good development and programming practices make it easier for you to specify extension points.

Second, and even more fundamentally, structuring a system this way makes systems much more understandable. You structure the system from a base and then add more functionality in chunks that are not necessary to understanding the base. This allows our limited minds to focus on a particular concern at a time in isolation without the disturbance of other concerns. You can apply this approach to structure almost anything beyond codes. You can even apply it to requirement specifications and design.

How is support for extensions achieved? It can be achieved in several ways—during compilation time or runtime. One of the possible ways to do so during runtime is through a much earlier work by Jacobson. It is known as a sequence variator, and its operation is depicted in Figure 1-5 [Jacobson 1981].

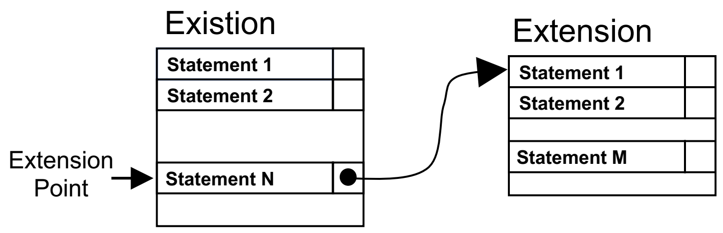

Figure 1-5 *Sequence variator.*

The sequence variator works at microprogram level. The program consists of a list of statements, and each statement has a bit flag to indicate whether an extension occurs at that point. In typical operation, the sequence variation takes a statement from memory and executes it, then takes and executes the next statement, then the next, and so on. If the extension bit flag is set, the sequence variator looks for an extension that references the current statement and proceeds to execute the statements in the extension. When all the statements in the extension have been executed, the sequence variator resumes with the statement at the existion and continues.

From the existion programmer's viewpoint, you only view the statements, not the extension bit flag. When an extension must be added later on, all that is needed is to code the extension statements and turn on the appropriate extension bit flags in the existion. The existion programmer does not have to worry about extensions that are added later. You can easily add extension after extension without breaking the modularity of the existion. There is no tangling or scattering.

Jacobson filed a patent for this approach in 1981, but the patent was not accepted. The idea was too close to a patented patching technique for which his proposal would have been an infringement, so Jacobson always had to apologize for this closeness before explaining the idea.

A common fear about adopting aspect orientation is that practitioners feel that it *is like patching*. Definitely, if used in an ad hoc manner, it indeed is like patching. But aspect orientation is not for patching. It is for you to achieve better separation of crosscutting concerns. It is for you to achieve better modularity. The goal of this book is to provide you with sound techniques and practical guidelines to achieve this.

1.3.2 Support for Extensions in UML

Even though the patent was not accepted, the concept of keeping extensions separate persists. It manifests as extension use-cases, which made it to the Unified Modeling Language. In fact, for those of you who have applied use-cases, you should be quite familiar with the use-case extension concept. We go into the details later, but what we want to say is that the idea of keeping extensions separate is not new. Briefly, use-case extensions permit us to describe additional behaviors that can be inserted into an existing use-case. For example, you have an existing use-case called Reserve Room, and you want to add some waiting list functionality, as exemplified earlier. With the use-case modeling technique, you simply add an extension use-case, which is modeled as the Handle Waiting List use-case in Figure 1-6. In use-case modeling terminology, the Reserve Room use-case would be a base use-case, and the Handle Waiting List would be an extension use-case.

The use-case technique provides the means to specify how behaviors described in the extension use-case are inserted at the extension points defined in the base use-case.

Nevertheless, the idea of keeping extensions separate remains a specification technique as opposed to an implementation technique. In *Object-Oriented Software Engineering* [Jacobson et al. 1992], there are techniques to keep extensions separate during analysis and during use-case modeling. However, there are no techniques to keep extensions separate during design, since there was no aspect-oriented programming language available when the book was written. In *Software Reuse* [Jacobson 1997], the authors generalize the concept of extension points into variation points, and many of these ideas have been carried over to the Reusable Asset Specification (RAS).

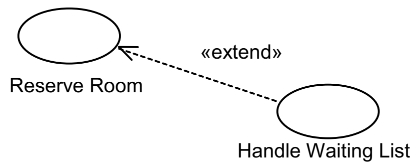

Figure 1-6 *Waiting list as an extension use-case.*

The first serious attempt to implement extensions was done in the early 1990s at Ericsson in the development of a new generation of switches. The people who adopted extensions took them into a new development environment called Delos, which supported extensions all the way down to code. Nevertheless, support for extensions in mainstream programming languages did not appear until the advent of aspect orientation technologies.

1.4 Keeping Concerns Separate

Being able to keep concerns separate is extremely important in software development. It helps you break down a complex problem into smaller parts and solve them individually. When it comes to large systems, it is the only way for you to build them. If you cannot keep concerns separate, the complexity of the system increases exponentially as you add enhancement after enhancement. By keeping concerns separate, on the other hand, the system is much easier to understand, maintain, and extend.

Existing modularity such as classes and components do help you keep concerns separate, at least to a certain extent. Each class keeps the specifics of a kind of object or real-world phenomenon separate; each component encapsulates the computation and data related for some functionality; and so on. However, when it comes to crosscutting concerns—concerns that cut across classes and components—you need another approach to modularity.

Moving ahead, in Chapter 2, we demonstrate how to maintain the separation of crosscutting concerns during implementation (i.e., code) using aspect orientation. In Chapter 3, we show how use-cases help us capture and model concerns. In Chapter 4, we show how, with use-cases and aspects, we can achieve separation of concerns from requirements to code, and we explain the steps necessary to get there.