Chapter 2

# The Development Process

All software development projects follow two distinct processes at the same time. The **Management Process** schedules work, plans deliveries, allocates resources, and monitors progress. The **Development Process** creates working software from requirements. Assuming you have these processes written down, you would consult your management process guide if you wanted help with setting milestones and your development process guide if you wanted help with allocating operations to interfaces.

The software process literature contains examples of methods that include either or both of these processes. For example, Dynamic Systems Development Method (DSDM) is a management process [DSDM], Catalysis is mostly a development process [D'Souza99], and the Rational Unified Process (RUP) [Jacobson99] covers both.

Although it hasn't always been this way, today the development process has to be subservient to the management process. This is because the management process controls project risk, and risk control is rightly viewed as paramount, even if the development process is compromised as a result. The favored management process nowadays is one based on evolution, where the software is delivered over a number of development iterations, each refining and building on the one before [Gilb88, GilbWeb]. The development process has to fit with that, so it isn't possible to specify everything, then design everything, then code everything, and so on, even if you wanted to.

Nevertheless, when we describe the development process we do so without taking into account the constraints of the management process. We do this because we want the development process to be usable with a variety of management processes. It also helps to make the development process understandable. We revisit the effects of the management process after we have explained the development process.

As you might have guessed from the title of this chapter, we explain our preferred development process here and throughout the rest of the book, but we do not cover the details of the various possible management processes.

## 2.1 Workflows

Figure 2.1 shows the overall development process. The boxes represent **Workflows**, as found in RUP. In his book on RUP, Philippe Kruchten defines a workflow as "a sequence of activities that produces a result of observable value" [Kruchten99]. The thin arrows represent the flow of **Artifacts**—deliverables that carry information between workflows. Comparing the workflows of Figure 2.1 to those found in RUP, the requirements, test, and deployment workflows correspond directly to those with the same names in RUP and are not discussed further in this book, with the exception of some minor elaboration of the requirements workflow. The specification, provisioning, and assembly workflows replace RUP's analysis and design and implementation workflows, and they define the scope of this book (see Figure 2.2).

Our primary concern is the specification workflow. Chapter 4 covers just those aspects of the requirements workflow that we need to generate inputs for specification. The specification workflow is divided into three sections: component identification, component interaction, and component specification (Chapters 5, 6, and 7). Chapter 8 covers sufficient parts of the provisioning and assembly workflows to give you at least a taste of implementation.

The specification workflow takes as its input from requirements a use case model and a business concept model. It also uses information about existing software assets, such as legacy systems, packages, and databases,

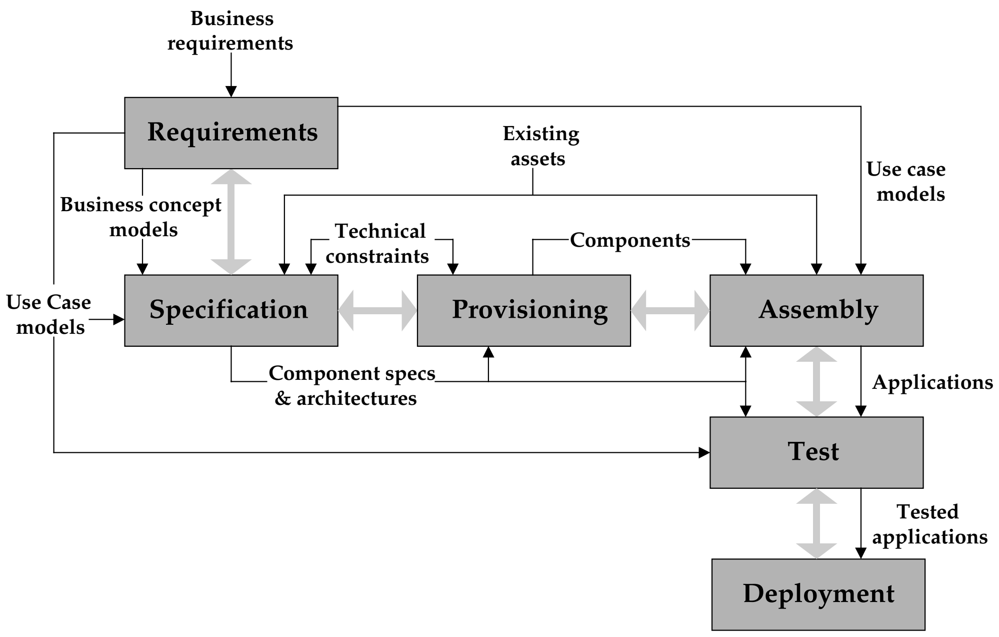

Figure 2.1 The workflows in the overall development process

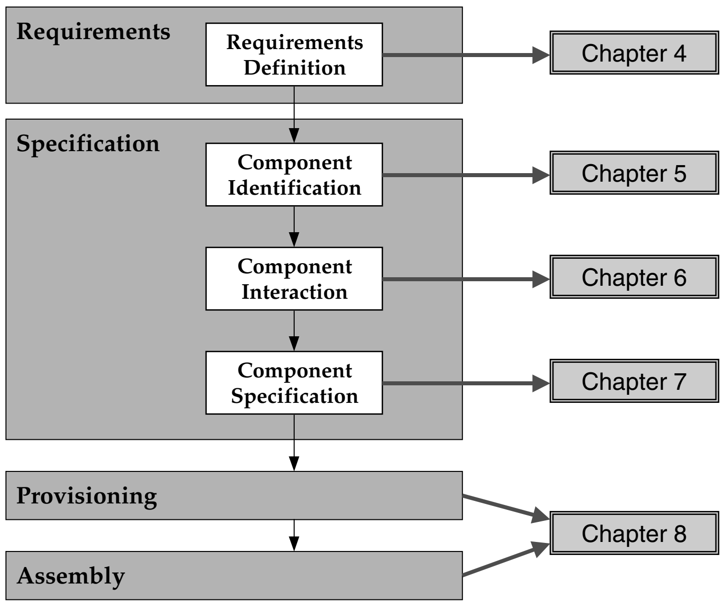

Figure 2.2 Workflow mappings to chapters

and technical constraints, such as use of particular architectures or tools. It generates a set of component specifications and a component architecture. The component specifications include the interface specifications they support or depend on, and the component architecture shows how the components interact with each other.

These outputs are used in the provisioning workflow to determine what components to build or buy, in the assembly workflow to guide the correct integration of components, and in the test workflow as an input to test scripts.

The provisioning workflow ensures that the necessary components are made available, either by building them from scratch, buying them from a third party, or reusing, integrating, mining, or otherwise modifying an existing component or other software. The provisioning workflow also includes unit testing the component prior to assembly.

The assembly workflow takes all the components and puts them together with existing software assets and a suitable user interface to form an application that meets the business need.

## 2.2 The Impact of the Management Process

### 2.2.1 The Evolution of Software Processes

The optimal development process—assuming an ideal, stable, and unchanging world—is one that gathers complete requirements, specifies completely a system to meet the requirements, designs all the software pieces, implements and integrates them, and tests the result. This is optimal, at least in theory, because doing all the design up front means you won't have to change the software once it has been written, and changing software has traditionally been expensive. But the risks inherent in this kind of waterfall process are well known, so the truly optimal management process, with risk management at its heart, takes a quite different tack. It insists that partial systems are delivered early and built incrementally. It assumes that requirements will change or, at best, are incompletely understood. It acknowledges that software might have to be changed as the truth behind the requirements emerges but accepts that as a price worth paying.

Early software methods often had a management process that followed, and was subservient to, the development process. Although such methods were fine for designing software, they were poor at controlling risks, with inevitable results. The reaction to the failure of these methods was the rise of methods that focused only or mainly on the management process, methods with a rapid application development (RAD) style. The predictable consequence was the creation of systems that were delivered on time but with very poor structure, making them difficult and expensive to change and extend. In turn, the reaction to RAD has been the rise of "architecture" as a rallying word for the reintroduction of sound design practices. Recent methods such as RUP attempt to achieve balance between the processes by setting out a full development process that lives within a rigid time-boxed management structure.

An interesting recent development has been the assertion by some leaders in the object technology community that by using modern software tools and techniques—especially planned refactoring of working software to improve its structure—the cost of changing software can be greatly reduced. They propose a method, called eXtreme Programming (XP), that adopts a highly iterative style and requires extensive changes to software already built and tested [Beck99]. XP relies on the belief that the cost of change can be made so low that an iterative and incremental development process, aligned completely with the iterative management process, can be as efficient as a linear one. If that is true, the two processes are no longer in tension.

### 2.2.2 Accommodating Change

Although the development process outlined here can be used with any management process, we assume you will be using a management process based on evolutionary delivery, and such processes require change to the software at each iteration.[1] As you will recall from Chapter 1, we think the major motivation for using a component approach is to manage change better. So there is a natural fit between components and evolutionary delivery.

1. Evolutionary delivery really combines iterative and incremental techniques: Each cycle of development both refines the existing artifacts and produces new ones.

Every iteration typically involves activities from all the workflows in the development process. The activities improve and extend the artifacts produced in the last iteration.

Figure 2.3 illustrates the evolution of artifacts during a project comprising five iterations. The first iteration concentrated on the requirements workflow, so the business concept model and use case model were substantially completed. But a small amount of work was done, even on the first iteration, in the specification, provisioning, and other workflows to demonstrate technical feasibility, as required for risk management. Each successive iteration has a different focus, but, even in the fourth iteration, for example, it may be necessary to revisit the requirements workflow use case model.

Actually, the whole notion of completion is highly suspect. Projects end when we have "done enough" —that is, when we decide that no more money should be spent on these artifacts, either because they meet the business need well enough or because it looks as though they never will. Of course, we hope that the artifacts produced by successful projects will have a long and glorious life. That life will inevitably involve changes, so it is important to design components to be amenable to both short- and long-term change.

But don't get carried away. It's extremely unlikely you will be able to predict the changes required for a component over its lifetime, and attempts explicitly to design-in multiple axes of change are likely to be expensive mistakes. However, the power of encapsulation means we can change pretty much anything about a component's implementation with only limited impact on the whole system. And factoring a component's capabilities into multiple interfaces reduces the impact of interface

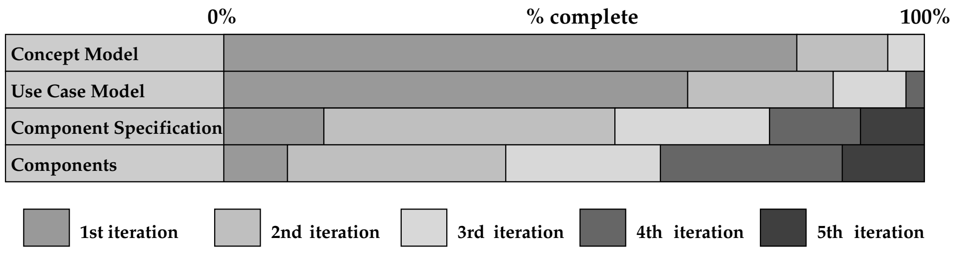

**Figure 2.3** Evolution of artifacts

changes. So simply by following component principles you'll get a powerful ability to react to change. When we say that components help with change we mean that their inherent characteristics limit the effects of change, not that they eliminate the need for change itself.

## 2.3 Workflow Artifacts

The requirements and specification workflows are responsible for producing a number of model artifacts. The main requirements artifacts we'll be dealing with are the **Business Concept Model** and the **Use Case Model**. In specification we'll be producing the **Business Type Model**, **Interface Specifications, Component Specifications,** and **Component Architecture.**

We've found it helpful to organize our model elements into a package structure that reflects these artifacts (see Figure 2.4).[2] Subsequent chapters elaborate on this structure as we explain how UML is applied to model each of the artifacts, and as we develop the case study.

The business concept model is a conceptual model of the business domain that needs to be understood and agreed. Its main purpose is to create a common vocabulary among the business people involved with the project. For example, if "customer" means three different things within the business, you need to get this cleared up as early as possible so that everyone is working to the same set of terms with agreed meanings.

A use case is a way of specifying certain aspects of the functional requirements of the system. It describes interactions between a user (or other external actor) and the system, and therefore helps to define what we call the system boundary. The use case model is the set of use cases you consider to be representative of the total functional requirements. You might start with the key ones, then add others later.

The business type model is an intermediate specification artifact and not an output of the specification workflow. Its purpose is to formalize the

2. As part of the management process you may wish vary from this, for example to keep multiple versions of certain artifacts and to retain intermediate refinement states, but these aspects are not shown.

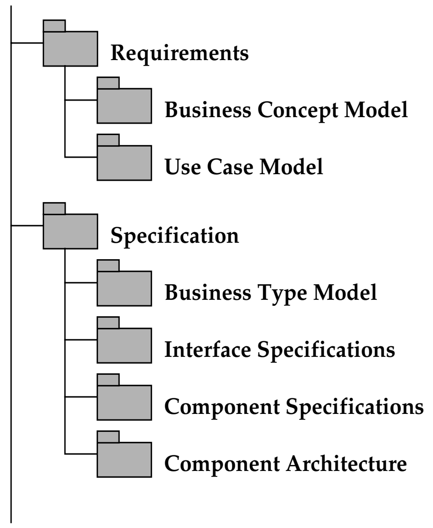

**Figure 2.4** Top-level organization of workflow artifacts

business concept model to define the system's knowledge of the outside world. This model is the basis for initial interface identification. While the business concept model describes the business domain as the business people understand it, the business type model captures exactly those aspects and rules of the business domain that the system knows about. The business concept model may be imprecise, but the business type model must be precise.

The interface specifications artifact is a set of individual interface specifications. Each interface specification is a contract with a client of a component object. Each interface specification defines the details of its contract in terms of the operations it provides, what their signatures are, what effects they have on the parameters of the operations and the state of the component object, and under what conditions these effects are guaranteed. This is where most of the detailed system behavior decisions are pinned down.

The component specifications artifact is a set of individual component specifications. Each component specification is defined in terms of interface specifications and constraints. A component specification defines the interfaces it supports and how their specifications correspond to each other. It also includes the interfaces it uses or consumes. While an interface specification represents the contract with the client, the component specification pulls these disparate client contracts together to define a single realization contract. This is where the building blocks of the system are defined.

The component architecture describes how the component specifications fit together in a given configuration. It binds the interface dependencies of the individual component specifications into component dependencies and describes how the component objects interact with each other. The architecture shows how the building blocks fit together to form a system that meets the requirements.

In Chapter 3 we explain how we model these artifacts in UML.

## 2.4 The Specification Workflow

The specification workflow is rather tricky to explain because explanations tend to be sequential, whereas the workflow tasks are highly iterative. The various workflow artifacts have clear dependencies, but their development is incremental, with additions and modifications at every stage. We have attempted to summarize the workflow tasks into the three stages: component identification, component interaction, and component specification (see Figure 2.5). For the most part you can take the word "component" here to be shorthand for "component and interface."

Note also that since we are staying management-process neutral, we haven't attempted to characterize the degree of completeness, or other quality criteria, of these workflow artifacts. The management process phases specify those.

### 2.4.1 Component Identification

The component identification stage takes as input the business concept model and the use case model from the requirements workflow. It assumes an application layering that includes a separation of system components and business components, as discussed in Chapter 1. Its goal is to identify an initial set of business interfaces for the business components and an initial set of system interfaces for the system components, and to pull these together into an initial component architecture. The business type model is an intermediate artifact from which the initial business interfaces are

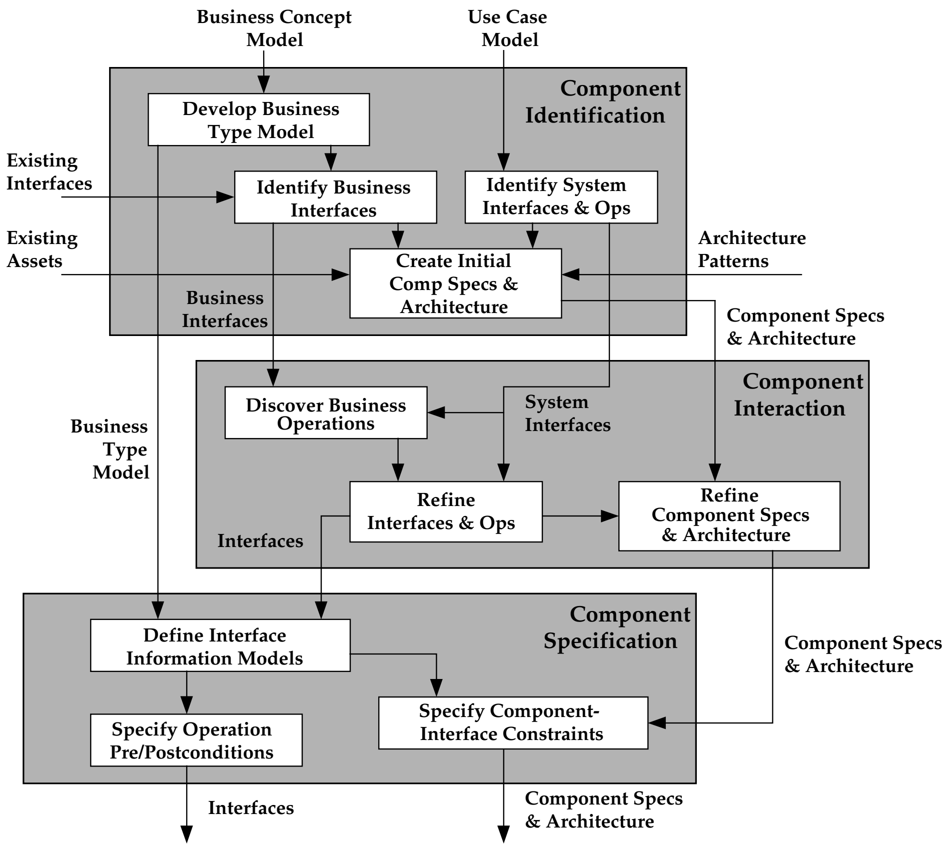
Business Concept Model
Use Case Model
Component Identification
Develop Business Type Model
Existing Interfaces
Identify Business Interfaces
Identify System Interfaces & Ops
Existing Assets
Create Initial Comp Specs & Architecture
Architecture Patterns
Business Interfaces
Component Specs & Architecture
Component Interaction
Discover Business Operations
System Interfaces
Business Type Model
Refine Interfaces & Ops
Refine Component Specs & Architecture
Interfaces
Component Specification
Define Interface Information Models
Component Specs & Architecture
Specify Operation Pre/Postconditions
Specify Component-Interface Constraints
Interfaces
Component Specs & Architecture

**Figure 2.5** The three stages of the specification workflow

formed. It is also used later, in the component specification stage, as the raw material for the development of interface information models.

Any existing components or other software assets need to be taken into account too, as well as any architecture patterns you plan to use. At this stage it's fairly broad-brush stuff, intended to set out the component and interface landscape for subsequent refinement.

In addition to identifying system interfaces, the identification stage also makes a first cut at the operations that need to be supported by the system. These are identified by name, but signatures and others details are added at a later stage. The system operations required are derived by examining the steps in the different use cases and deciding what the system's responsibilities are.

### 2.4.2 Component Interaction

The component interaction stage examines how each of the system operations will be achieved using the component architecture. It uses interaction models to discover operations on the business interfaces. As more interactions are considered, common operations and patterns of usage emerge that can be factored out and reused. Responsibility choices become clearer and operations are moved from one interface to another. Alternative groupings of interfaces into components can be investigated. This is the time to think through the management of references between component objects so that dependencies are minimized and referential integrity policies are accommodated.

The component interaction stage is where the full details of the system structure emerges, with a clear understanding of the dependencies between components, down to the individual operation level.

### 2.4.3 Component Specification

The final stage of specification is where the detailed specification of operations and constraints takes place. For a given interface it means defining the potential states of component objects in an **Interface Information Model**, and then specifying pre- and postconditions for operations, and capturing business rules as constraints. The pre- and postconditions and other constraints make reference to the types in the interface information model and the types of the parameters. In addition to these interface specification details, this stage also witnesses the specification of constraints that are specific to a particular component specification and independent of each interface. These component specification constraints determine how the type definitions in individual interfaces will correspond to each other in the context of that component.

The architecture should not materially change at this stage. This detailed specification task should be undertaken once the architecture is stable and all the operations of the interfaces have been identified. The act of writing the precise rules for each operation may help you discover missing parameters, or missing information, but the emphasis is on filling in detail onto a stable framework.

4504_CH02 Page 36 Friday, September 15, 2000 2:23 PM

## 2.5 Summary

Projects follow two processes. The management process schedules work, plans deliveries, allocates resources, assesses risk, and monitors progress. The development process creates working software from requirements.

The development process, which is the subject of this book, can be divided into a number of workflows that have dependencies but that can proceed iteratively.

The requirements workflow generates a business concept model and a use case model.

The specification workflow is divided into three stages:

1. The component identification stage produces an initial component architecture from the requirements.
2. The component interaction stage discovers the operations needed and allocates responsibilities.
3. The component specification stage creates precise specifications of operations, interfaces, and components.

The provisioning workflow is responsible for delivering software conforming to the component specification it is given. This may be through implementing it, buying it, and adapting it, or by integrating and adapting existing software.

The assembly workflow links components, user interface and application logic, and existing software together into a working application.