CHAPTER 2

The Object-Oriented Development Process

This chapter describes the use case and the object-oriented development process, using the architecture presented in Chapter 1. It describes the characteristics of a robust software development process and the detail of how the use case integrates into all of the process activities.

2.1 High-level use case definition

A use case is a transaction that begins with a user stimulus and ends with a response. In between the program logic accepts and validates the input and then determines the required actions, which may include updates to business data. This definition of a use case is not the same as that suggested by [Jacobson93]. This use case definition is intended to be more granular and rigorous. This granularity is essential to the correctness issues discussed in the details of OODP.

Consider the use case as the definition of all possible responses to an external event. Events are those needs that motivate a user of the system to stimulate it by pressing a key, manipulating a mouse, or performing any other activity that the interface will recognize as a stimulus. Events are described
with phrases such as: “The customer wishes to transfer money,” “The customer wants to rent a video,” and “The manager needs to add a new item to the inventory.” The wants and needs phrases are intended to show the motivation of the user of the system, a need that the application must fulfill.

When identifying use cases, it is important to differentiate them from business processes. A single business process may be composed of several use cases. For example, the business process shown in Figure 2-1 is actually composed of several potential use cases.

To do effective requirements gathering, it is important to recognize that this business process involves, not one, but four use cases. Chapter 3 on requirements definition elaborates on the rules that help make the distinction between use case and business processes. Many analysts, in a rush to find use cases, will identify the business processes instead. Organizations that struggle with identifying use cases often do so because they do not see beyond the business processes. One client team was trying to find the use case in an inventory management application. The business process analysts were arguing over whether the count of items in inventory should be updated when the order was received on the dock, delivered to the storage facility, or placed in the bin. The software analysts were frustrated because they could not get agreement on business rules. When they finally realized that the use case was “Update inventory” and not “Receive inventory and update quantity in hand,” the use case and the business rules became simple to differentiate.

**Fig. 2-1** A business process for a loan application.
2.2 The rationale behind the use case model

The business process model could have been used as a vehicle for requirements gathering, but the use case as defined was chosen for OODP instead for the following reasons:

- Business processes are too complex and often involve steps that are not part of the application. For example, the banking application may be able to accept a loan request, obtain the credit report, and create the loan, but the process for approving the loan is a human activity.
- As use cases are combined into a single requirement, it multiplies their complexities. When they are treated separately, system complexity grows linearly.
- Business processes are more likely to change than the individual use cases, and often a business process change can be accommodated without ever adding or changing a use case.
- Each transaction can be validated as its own entity with its own specification, which makes it easier for the design and implementation to be validated. This concept of a use case, as an independent entity, is discussed further in the use case design chapter.

2.3 How use cases link the process

The following topics describe the role that use cases play in each OODP activity and demonstrate how the use case model integrates into the process architecture of Chapter 1.

2.3.1 Requirements

The application’s functionality is recorded as use cases. For each use case, the requirements capture its inputs and outcomes.

2.3.2 Analysis

The use case is partitioned into scenarios, each with its own pre-condition, post-condition, and output. This is the detail that defines the business rules for the use case.

2.3.3 Design

Each use case is designed by creating a series of messages that determine the scenario in effect at the time the use case is instantiated. This is followed by additional messages to update the business object attributes with the values defined in the scenario’s post-condition.
2.3.4 Implementation

The use case design messages are implemented as methods in the application objects. A special application class with a method for each use case is written to manage the control flow within an individual use case.

2.3.5 Test

One or more test cases are created for every scenario in a use case.

2.3.6 User interface

If the use case is triggered by a user interface event, it is the interface’s responsibility to gather the inputs required for that use case and send a message with those inputs to the appropriate method in the application object.

2.4 Criteria for a Robust Process

During the design of the object-oriented development process, the elements of several methodologies were reviewed. It became apparent that what was needed was not another methodology but a formal software process that would support object-oriented development. Methodologies tend to lack cohesion and structure. They often focus on only a few software activities or deliverables and ignore the process aspects. One of the results of this study was a set of criteria that could be used to evaluate any operational software process. These were not abstract properties like effective, usable, or modern but rather the following concrete and measurable attributes:

- Complete
- Consistent
- Verifiable
- Traceable
- Incrementable
- Testable

The definition of each of these attributes is discussed in the following topics. Subsequent chapters will refer to these attributes and measure the object-oriented development process against them.

2.4.1 Complete

A complete process is one that starts with basic minimal requirements as input and includes the details necessary to progress through all the required software development activities and to deliver a tested working product. This is an essential element often overlooked in many methodologies. The process is
incomplete when, for example, it provides complete details on the analysis activity but omits the details for design and implementation. When using a process that is not complete, the project team may not know whether it is doing analysis or design, and more importantly, it may not know when it is done. The process architecture described in Chapter 1 can be an aid to ensure that the process has not omitted critical steps. The end of this chapter contains a short description of the activities and deliverables for the object-oriented development process.

2.4.2 Consistent

Hand in hand with a complete process is one that is **consistent**. That means that, except for the first activity, every input to a process activity must be a deliverable from a previous one. When methods, attributes, objects, and associations seem to appear out of thin air, there is no way to know when all of them have been found or how many may still be missing. A robust process should have only one ill-defined input, that is the user’s requirements, and all other activity inputs should have been created by a prior step in the process. A corollary to this principle says that no deliverable should be created that is not either input to a subsequent process step or is considered one of the deliverables that was promised to the client as part of building the application. These include the machine-runnable code and documentation.

2.4.3 Verifiable

**Verifiable** processes are those whose activities contain deliverable correctness criteria. This is a checklist of objective questions that assert that the deliverable was correctly created from the input and that it will meet the entrance criteria of the input of subsequent activities. If the initial set of requirements is checked to see that they are what the customer requires, and the analysis deliverables are validated against those requirements, then we can reason that the analysis use cases are a correct refinement of the requirements. If design and implementation continue this practice for their inputs and outputs and the process is consistent, then the delivered code will also meet the user’s requirements. Often a process relies on testing to validate that the product has met requirements. Testing is an important activity, but it is inefficient and sometimes ineffective in terms of correctness. Testing to validate correctness incorrectly assumes the following:

- All defects can be discovered during test.
- There is time to redesign and retest when defects are found.
- The test cases will test the requirements.

By ensuring correctness of each deliverable with respect to its input verification is streamlined, errors are detected sooner, and testing becomes confirmation that the team was successful in building the right piece of software.
2.4.4 Traceable

A **traceable** process has two components:

- Every deliverable can be associated with a requirement.
- For each product requirement, the deliverables that implement it can be identified.

The first rule prevents overbuilding the software; the second makes maintenance as efficient as possible. If a product will never be changed and no new requirements will surface during development, perhaps the traceable attribute could be ignored; however, since this is almost never the case, there needs to be a way that, when requirements do change, each and every deliverable in the product need not be scrutinized to see if it is affected by the change. A traceable process allows swift identification of a few pieces of the application that together completely encapsulate the change. This attribute not only includes delivered code but design, analysis, and test deliverables as well. Use cases are designed and implemented in a way that makes them independent of each other. That is, a use case has its own design and a discrete set of methods and objects that implement it, which means that only they need to be considered for rework when that use case changes. This approach will normally involve a very small and manageable piece of the product. Of course, some new methods, attributes, or objects may be required as well, but even they can be localized. Because of these characteristics, traceable is sometimes referred to as maintainable.

2.4.5 Incrementable

The **incrementable** characteristic implies that the requirements can be partitioned such that the effort required to build the application in pieces is virtually the same as if it were built as a single unit. This approach allows for tremendous flexibility in work distribution and includes all the advantages of incremental development discussed in Chapter 1. The important part of the incremental property is that the cost of partitioning the problem into several smaller pieces approaches zero. It eliminates, as much as possible, rewriting or removing running code. Use cases make this approach possible because they are implemented as independent entities.

2.4.6 Testable

Finally, the process must be **testable**, which means that test cases can be developed from the requirements. This condition implies that requirements either do not change or are always maintained to reflect the software being built. It has been said about some development efforts, that when a delivered product’s functionality matches its requirement it is known as “coincidence.”
Testability requires that as the specification, design, or code changes, so must the associated requirement use case definition, and all the existing intermediate deliverables as well. The only thing worse than missing documentation is incorrect documentation.

Since all use cases will not be detailed during analysis, it must be possible to design some use cases directly from the requirements. If the design is also omitted, or later deleted, the code for this use case must be developed directly from the requirements. Whenever the decision is made to eliminate some of the process deliverables, the developer must develop the remainder of the solution from the lowest level documentation available. This implies that requirements can never be eliminated and must always accurately reflect the current application. This is the summation of the rules for a testable process. “The requirements use case definitions must always be complete and accurate.”

2.5 Overview of the activities of OODP

An abbreviated description of the activities in the object-oriented development process follows. It includes a short definition of the purpose of each activity, its inputs (requirements), and its outputs (deliverables). It does not contain all the process detail described in Chapter 1, but that information can be found in Appendix A where the process and its activities are fully explained. It is the intent here to highlight the flow among the activities of OODP while providing a brief definition of each one. The details of each activity are covered in Chapters 3 through 12.

Figure 2-2 shows the flow and deliverables of the activities that make up the object-oriented development process. The ovals represent the activities and the lines between the activities represent the deliverables of one activity that flow into the next. Additional details on these activities can be found in Appendix A.

One project may refer to an activity as “analysis,” whereas another calls the same activity “requirements gathering.” OODP’s activity naming is not intended to be definitive, but it is important to understand the terms used within the process definition. The remainder of the book will assume the names and activities as defined here.

2.5.1 Requirements gathering

The purpose of requirements is to bound the problem space and provide the team with enough information to grasp its scope without getting mired in the details of analysis. There should be sufficient information about the product to develop a project resource plan, estimate, and schedule and to begin the activities of analysis, test planning, and user interface design.
Input Textual statement of need
Deliverables Use case definitions
Initial list of objects, attributes, and actors
Incremental plan

2.5.2 Test planning

Once the scope and functionality of the product is captured the test team needs to demonstrate that they can test this application thoroughly. The test plan is provided so the client can review the tests and concur that the tested product will deliver all the required capabilities.

Input Use case definitions (for the current increment)
Deliverables Details on how each scenario will be tested
2.5.3 Analysis (and application specification)

If requirements define the product breadth then analysis defines its depth. Analysis captures, records, and validates the business rules for each scenario of every use case. That means supplementing the use case inputs and outcomes by adding a pre-condition and post-condition for each scenario. The last step is to get the client’s concurrence that the use case descriptions completely and correctly define the application’s response to every event.

**Input**
- Use case definitions (for current increment)
- Initial list of object and attributes

**Deliverables**
- Use case descriptions
- Data dictionary
- Class diagram
- Updated test plan

2.5.4 User interface design

Requirements record the application’s functionality, but it is still necessary to show the user how that functionality will look and feel in a production environment. User interface design, while considered a nonsoftware related activity within OODP, is still necessary. The screen designs created and approved by the client are input to the user interface implementation activity.

**Input**
- Use case definitions (from requirements)
- Client/user input on look and feel

**Deliverables**
- Screen designs and flow

2.5.5 User interface implementation and test

This step in the process addresses the creation of the user interface after it has been designed. It involves implementing the user interface independent of any application and testing to ensure that it meets the criteria specified by the requirements use case definitions.

**Input**
- (Screen designs)
- Use case definitions
- Scenario semantics (includes input types)

**Deliverables**
- A set of user interfaces that send the correct messages to the application for each use case
2.5.6 Test case design and development

When analysis is complete and the scenario pre-conditions and post-conditions are defined, the test plan must be reviewed for correctness and completeness, and, when necessary, updated. While the development team is doing use case design and implementation the test team will be writing the test cases based on the updated test plan.

- **Input**
  - Use case descriptions
  - Updated test plan

- **Deliverables**
  - Details of messages for each test case in this increment
  - List of attributes required for test case validation, for the current increment

2.5.7 Use case design

Use cases are designed by creating a sequence of messages that first determine the correct scenario and then modify the attributes of the business objects to achieve the required state of the use case scenario as specified in the post-condition.

- **Input**
  - Use case descriptions

- **Deliverables**
  - Design of each use case recorded with a sequence diagram
  - Design data model for the current increment

2.5.8 Class design

Use case design is followed by class design and method specification. The methods are those that the designers determined were required to support the design of the use case for this increment and those that the testers required to support test case validation.

- **Input**
  - Use case designs
  - Updated data model

- **Deliverables**
  - Class specifications
  - Updated design data model

2.5.9 Use case and class implementation

Each use case is implemented as a separate method in the application. Classes as defined in support of the design are created as well.


2.5 Overview of the activities of OODP

Input
Use case designs
Class specifications and design
Test case attribute references

Deliverables
Coded and compiled classes

2.5.10 Use case test

During test, every test case must run correctly, with no intervening failures. That means that if a failure occurs during a test run, then all the tests must be rerun once the defects are found and fixed. This requires a program capable of running all tests without intervention. As failures are discovered, the test and development teams work together to determine if the flaw is in the use case design, the implementation, or the test case itself.

Input
Test cases
Compiled classes

Deliverables
Test results

2.5.11 Interface and application integration

Since OODP tests the application without using the user interface and the user interface is developed without using the application, a final integration test is required when the user interface and application are joined. More traditional testing including performance, stress, integration, and system follows this.

Input
Tested use cases
Installed classes
User interface implementation

Deliverables
Integrated tested product

Do not expect to completely understand this process from the brief amount of information shown so far. This section was intended only to present the flow of OODP at a high level. The information required to implement and execute this process with confidence is contained in the details of the chapters that follow.