Chapter

1

OBJECT-ORIENTED ANALYSIS AND DESIGN

The shift of focus (to patterns) will have a profound and enduring effect on the way we write programs.

—Ward Cunningham and Ralph Johnson

Objectives

- Compare and contrast analysis and design.
- Define object-oriented analysis and design (OOA/D).
- Illustrate a brief example.

1.1 Applying UML and Patterns in OOA/D

What does it mean to have a good object design? This book is a tool to help developers and students learn core skills in object-oriented analysis and design (OOA/D). These skills are essential for the creation of well-designed, robust, and maintainable software using object technologies and languages such as Java, C++, Smalltalk, and C#.

The proverb "owning a hammer doesn't make one an architect" is especially true with respect to object technology. Knowing an object-oriented language (such as Java) is a necessary but insufficient first step to create object systems. Knowing how to "think in objects" is also critical.

This is an introduction

This is an introduction to OOA/D while applying the Unified Modeling Language (UML), patterns, and the Unified Process. It is not meant as an advanced text; it emphasizes mastery of the fundamentals, such as how to assign responsibilities to objects, frequently used UML notation, and common design pat-

terns. At the same time, primarily in later chapters, the material progresses to a few intermediate-level topics, such as framework design.

Applying UML

The book is not just about the UML. The **UML** is a standard diagramming notation. As useful as it is to learn notation, there are more critical object-oriented things to learn; specifically, how to think in objects—how to design object-oriented systems. The UML is not OOA/D or a method, it is simply notation. It is not so helpful to learn syntactically correct UML diagramming and perhaps a UML CASE tool, but then not be able to create an excellent design, or evaluate and improve an existing one. This is the harder and more valuable skill. Consequently, this book is an introduction to object design.

Yet, one needs a language for OOA/D and "software blueprints," both as a tool of thought and as a form of communication with others. Therefore, this book explores how to *apply* the UML in the service of doing OOA/D, and covers frequently used UML notation. But the emphasis is on helping people learn the art and science of building object systems, rather than notation.

Applying patterns and assigning responsibilities

How should **responsibilities** be allocated to classes of objects? How should objects interact? What classes should do what? These are critical questions in the design of a system. Certain tried-and-true solutions to design problems can be (and have been) expressed as best-practice principles, heuristics, or **patterns**—named problem-solution formulas that codify exemplary design principles. This book, by teaching how to apply patterns, supports quicker learning and skillful use of these fundamental object design idioms.

One case study

This introduction to OOA/D is illustrated in a **single case study** that is followed throughout the book, going deep enough into the analysis and design so that some of the gory details of what must be considered and solved in a realistic problem are considered, and solved.

Use cases and requirements analysis

OOA/D (and all software design) is strongly related to the prerequisite activity of **requirements analysis**, which includes writing **use cases**. Therefore, the case study begins with an introduction to these topics, even though they are not specifically object-oriented.

An example iterative process—the Unified Process

Given many possible activities from requirements through to implementation, how should a developer or team proceed? Requirements analysis and OOA/D needs to be presented in the context of some development process. In this case, the well-known **Unified Process** is used as the *sample* **iterative development process** within which these topics are introduced. However, the analysis and design topics that are covered are common to many approaches, and learning them in the context of the Unified Process does not invalidate their applicability to other methods.

In conclusion, this book helps a student or developer:

- Apply principles and patterns to create better object designs.
- Follow a set of common activities in analysis and design, based on the Unified Process as an example.
- Create frequently used diagrams in the UML notation.

It illustrates this in the context of a single case study.

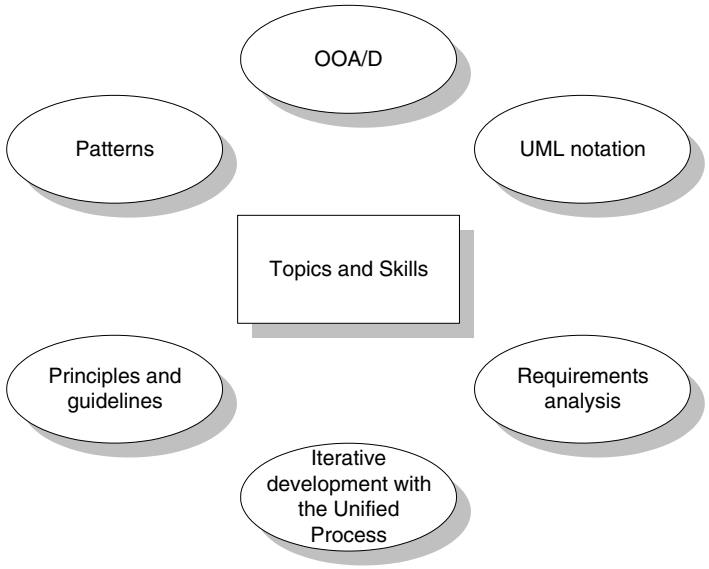


Figure 1.1 Topics and skills covered

Many Other Skills Are Important

Building software involves myriad skills and steps beyond requirements analysis, OOA/D, and object-oriented programming. For example, usability engineering and user interface design are critical to success; so is database design.

However, this introduction emphasizes OOA/D, and does not attempt to cover all topics in software development. It is one piece of a larger picture.

1.2 Assigning Responsibilities

There are many possible activities and artifacts in introductory OOA/D, and a wealth of principles and guidelines. Suppose we must choose a single practical skill from all the topics discussed here—a "desert island" skill. What would it be?

> A critical, fundamental ability in OOA/D is to skillfully assign responsibilities to software components.

Why? Because it is one activity that must be performed—either while drawing a UML diagram or programming—and it strongly influences the robustness, maintainability, and reusability of software components.

Of course, there are other necessary skills in OOA/D, but responsibility assignment is emphasized in this introduction because it tends to be a challenging skill to master, and yet vitally important. On a real project, a developer might not have the opportunity to perform any other analysis or design activities—the "rush to code" development process. Yet even in this situation, assigning responsibilities is inevitable.

Consequently, the design steps in this book emphasize principles of responsibility assignment.

> Nine fundamental principles in object design and responsibility assignment are presented and applied. They are organized in a learning aid called the GRASP patterns.

1.3 What Is Analysis and Design?

Analysis emphasizes an *investigation* of the problem and requirements, rather than a solution. For example, if a new computerized library information system is desired, how will it be used?

"Analysis" is a broad term, best qualified, as in *requirements analysis* (an investigation of the requirements) or *object analysis* (an investigation of the domain objects).

Design emphasizes a *conceptual solution* that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Ultimately, designs can be implemented.

As with analysis, the term is best qualified, as in *object design* or *database design*.

Analysis and design have been summarized in the phase *do the right thing (analysis), and do the thing right (design)*.

1.4 What Is Object-Oriented Analysis and Design?

During **object-oriented analysis**, there is an emphasis on finding and describing the objects—or concepts—in the problem domain. For example, in the case of the library information system, some of the concepts include *Book, Library,* and *Patron*.

During **object-oriented design**, there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a *Book* software object may have a *title* attribute and a *getChapter* method (see Figure 1.2).

Finally, during implementation or object-oriented programming, design objects are implemented, such as a *Book* class in Java.

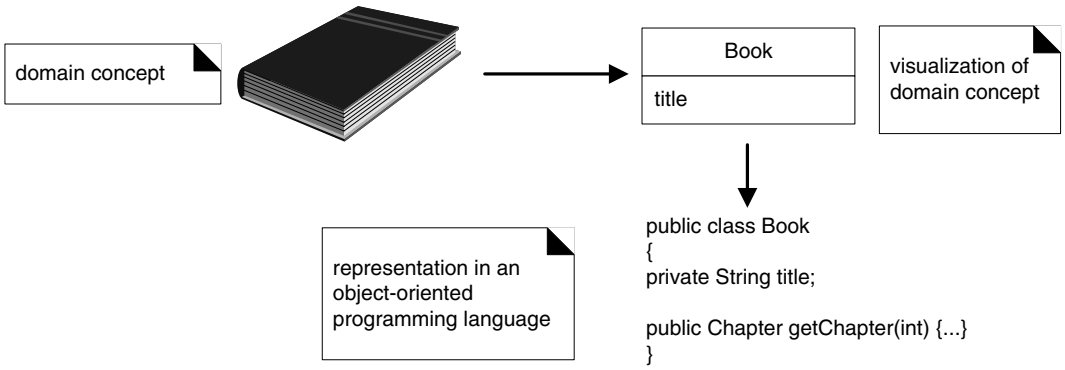


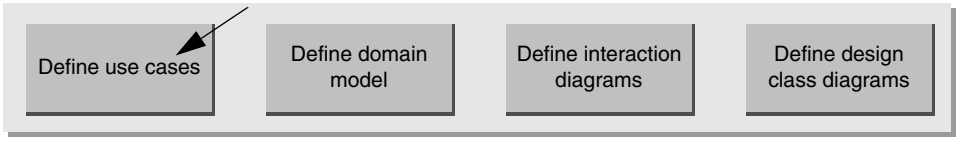
Figure 1.2 Object-orientation emphasizes representation of objects.

1.5 An Example

Before diving into the details of requirements analysis and OOA/D, this section presents a birds-eye view of a few key steps and diagrams, using a simple example—a "dice game" in which a player rolls two die. If the total is seven, they win; otherwise, they lose.

Define Use Cases

Requirements analysis may include a description of related domain processes; these can be written as **use cases**.

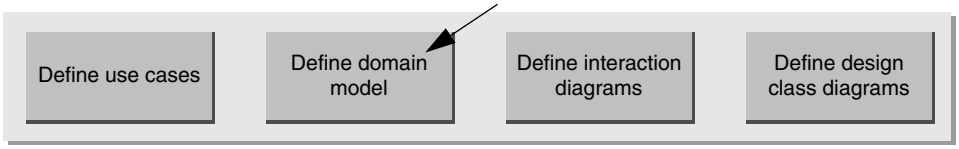


Use cases are not an object-oriented artifact—they are simply written stories. However, they are a popular tool in requirements analysis and are an important part of the Unified Process. For example, here is a brief version of the *Play a Dice Game* use case:

> **Play a Dice Game**: A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose.

Define a Domain Model

Object-oriented analysis is concerned with creating a description of the domain from the perspective of classification by objects. A decomposition of the domain involves an identification of the concepts, attributes, and associations that are considered noteworthy. The result can be expressed in a **domain model**, which is illustrated in a set of diagrams that show domain concepts or objects.



For example, a partial domain model is shown in Figure 1.3.

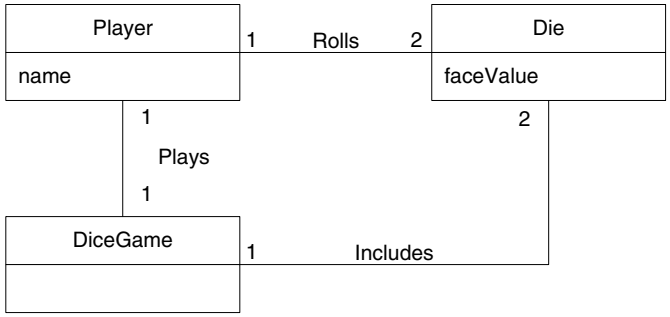


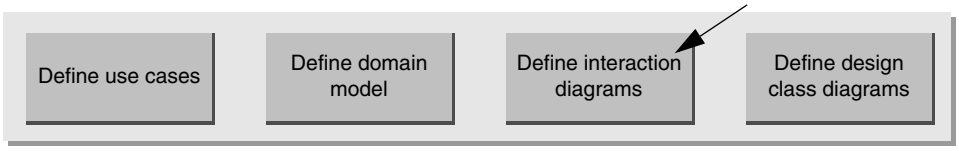
Figure 1.3 Partial domain model of the dice game.

This model illustrates the noteworthy concepts *Player*, *Die,* and *DiceGame*, with their associations and attributes.

Note that a domain model is not a description of software objects; it is a visualization of concepts in the real-world domain.

Define Interaction Diagrams

Object-oriented design is concerned with defining software objects and their collaborations. A common notation to illustrate these collaborations is the **interaction diagram**. It shows the flow of messages between software objects, and thus the invocation of methods.



For example, assume that a software implementation of the dice game is desired. The interaction diagram in Figure 1.4 illustrates the essential step of playing, by sending messages to instances of the *DiceGame* and *Die* classes.

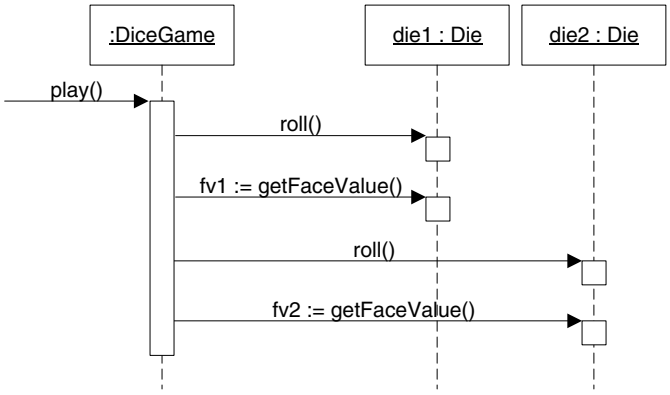


Figure 1.4 Interaction diagram illustrating messages between software objects.

Notice that although in the real world a *player* rolls the dice, in the software design the *DiceGame* object "rolls" the dice (that is, sends messages to *Die* objects). Software object designs and programs do take some inspiration from real-world domains, but they are *not* direct models or simulations of the real world.

Define Design Class Diagrams

In addition to a *dynamic* view of collaborating objects shown in interaction diagrams, it is useful to create a *static* view of the class definitions with a **design class diagram**. This illustrates the attributes and methods of the classes.

For example, in the dice game, an inspection of the interaction diagram leads to the partial design class diagram shown in Figure 1.5. Since a *play* message is sent to a *DiceGame* object, the *DiceGame* class requires a *play* method, while class *Die* requires a *roll* and *getFaceValue* method.

In contrast to the domain model, this diagram does not illustrate real-world concepts; rather, it shows software classes.

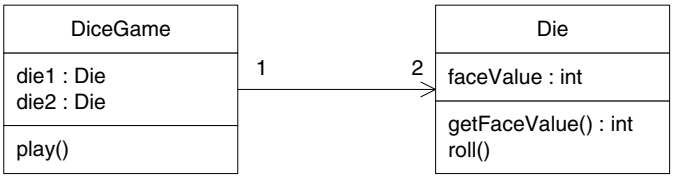


Figure 1.5 Partial design class diagram.

Summary

The dice game is a simple problem, presented to focus on a few steps and artifacts in analysis and design. To keep the introduction simple, not all the illustrated UML notation was explained. Future chapters explore analysis and design and these artifacts in closer detail.

1.6 The UML

To quote:

> The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [OMG01].

The UML has emerged as the de facto and de jure standard diagramming notation for object-oriented modeling. It started as an effort by Grady Booch and Jim Rumbaugh in 1994 to combine the diagramming notations from their two popu-

lar methods—the Booch and OMT (Object Modeling Technique) methods. They were later joined by Ivar Jacobson, the creator of the Objectory method, and as a group came to be known as the *three amigos*. Many others contributed to the UML, perhaps most notably Cris Kobryn, a leader in its ongoing refinement.

The UML was adopted in 1997 as a standard by the OMG (Object Management Group, an industry standards body), and has continued to be refined in new OMG UML versions.

This book does not cover every minute aspect of the UML, which is a large body of notation (some say, too large[1]). It focuses on diagrams which are frequently used, the most commonly used features within those diagrams, and core notation that is unlikely to change in future versions of the UML.

Why Won't We See Much UML for a Few Chapters?

This is not just a UML notation book, but one that explores the larger picture of applying the UML, patterns, and an iterative process in the context of software development. The UML is primarily applied during OOA/D, which is normally preceded by requirements analysis. Therefore, the initial chapters present an introduction to the important topics of use cases and requirements analysis, which are then followed by chapters on OOA/D and more UML details.

1.7 Further Readings

A very readable and popular summary of essential UML notation is *UML Distilled*, by Martin Fowler.

A succinct and popular introduction to the Unified Process (and its refinement in the Rational Unified Process) is *The Rational Unified Process—An Introduction* by Philippe Kruchten.

For a detailed discussion of UML (version 1.3) notation, *The Unified Modeling Language Reference Manual* and *The Unified Modeling Language User Guide,* by Booch, Jacobson, and Rumbaugh are worthwhile. Note that these texts were not meant for learning how to do object modeling or OOA/D—they are UML diagram notation references.

For a description of the current version of the UML, the on-line *OMG Unified Modeling Language Specification* at www.omg.org is necessary. UML revision work and soon-to-be released versions can be found at www.celigent.com/uml.

There are many books on software patterns, but the seminal classic is *Design Patterns*, by Gamma, Helm, Johnson, and Vlissides. It is truly required reading

1. The UML 2.0 effort includes exploration of the goal of simplifying and reducing the notation. This book presents high-use UML likely to survive future simplification.

for those studying object design. However, it is not an introductory text and is best read after developing comfort with the fundamentals of object design and programming.