Chapter

1

OBJECT-ORIENTED ANALYSIS AND DESIGN

Le temps est un grand professeur, mais malheureusement il tue tous ses élèves
(Time is a great teacher, but unfortunately it kills all its pupils.)

—Hector Berlioz

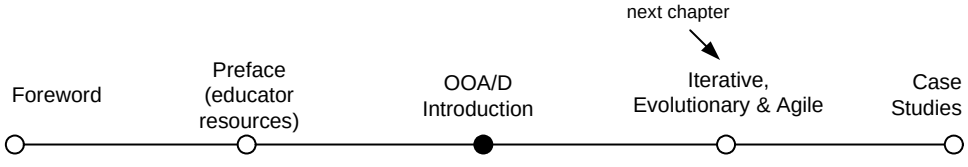
Objectives

- Describe the book goals and scope.
- Define object-oriented analysis and design (OOA/D).
- Illustrate a brief OOA/D example.
- Overview UML and visual agile modeling.

1.1 What Will You Learn? Is it Useful?

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 OO technologies and languages such as Java or C#.

What's Next? This chapter introduces the book goals and OOA/D. The next introduces iterative and evolutionary development, which shapes how OOA/D is presented in this book. The case studies are evolved across three iterations.



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 critical!

This is an introduction to OOA/D while applying the Unified Modeling Language (UML) and patterns. And, to iterative development, using an agile approach to the Unified Process as an example iterative 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 patterns. At the same time, mostly in later chapters, the material progresses to some intermediate-level topics, such as framework design and architectural analysis.

UML vs. Thinking in Objects

The book is not just about UML. The **UML** is a standard diagramming notation. Common notation is useful, but there are more important OO things to learn—especially, how to think in objects. The UML is not OOA/D or a method, it is just diagramming notation. It's useless to learn UML and perhaps a UML CASE tool, but not really know how to create an excellent OO design, or evaluate and improve an existing one. This is the hard and important skill. Consequently, this book is an introduction to object design.

Yet, we need a language for OOA/D and "software blueprints," both as a tool of thought and as a form of communication. Therefore, this book explores how to *apply* the UML in the service of doing OOA/D, and covers frequently used UML.

OOD: Principles and Patterns

How should **responsibilities** be allocated to classes of objects? How should objects collaborate? What classes should do what? These are critical questions in the design of a system, and this book teaches the classic OO design metaphor: **responsibility-driven design**. Also, 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 or principles, supports quicker learning and skillful use of these fundamental object design idioms.

Case Studies

This introduction to OOA/D is illustrated in some **ongoing case studies** that are 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

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

Iterative Development, Agile Modeling, and an Agile UP

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 and practiced in the context of some development process. In this case, an **agile** (light, flexible) approach to the well-known **Unified Process** (UP) 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 an agile UP does not invalidate their applicability to other methods, such as Scrum, Feature-Driven Development, Lean Development, Crystal Methods, and so on.

In conclusion, this book helps a student or developer:

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

It illustrates this in the context of long-running case studies that evolve over several iterations.

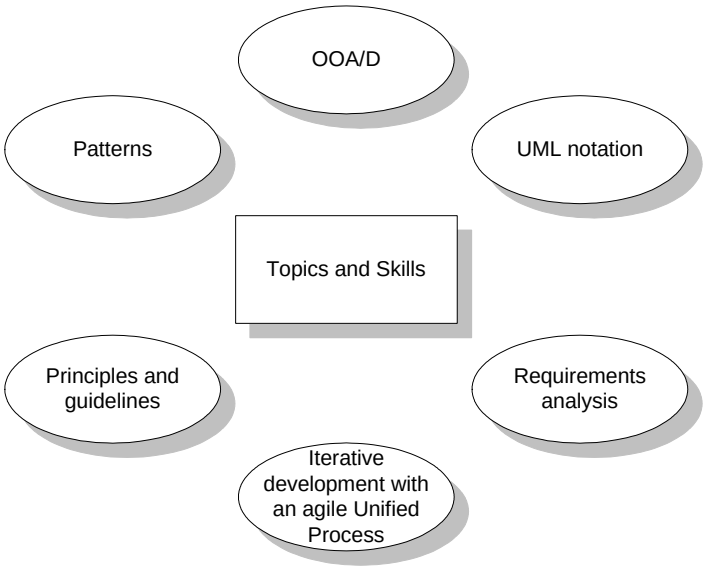


Figure 1.1 Topics and skills covered.

Many Other Skills Are Important!

This isn't the *Compleate Booke of Software*; it's primarily an introduction to OOA/D, UML, and iterative development, while touching on related subjects. Building software involves myriad other skills and steps; for example, usability engineering, user interface design, and database design are critical to success.

1.2 The Most Important Learning Goal?

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 ability in OO development is to
> skillfully assign responsibilities to software objects.

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 important skills in OOA/D, but *responsibility assignment* is emphasized in this introduction because it tends to be a challenging skill to master (with many "degrees of freedom" or alternatives), and yet is vitally important. On a real project, a developer might not have the opportunity to perform any other modeling 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 **GRASP** of principles with names such as *Information Expert* and *Creator*.

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 online trading system is desired, how will it be used? What are its functions?

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

Design emphasizes a *conceptual solution* (in software and hardware) that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Design ideas often exclude low-level or "obvious" details—obvious to the intended consumers. Ultimately, designs can be implemented, and the implementation (such as code) expresses the true and complete realized design.

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

Useful analysis and design have been summarized in the phrase *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 flight information system, some of the concepts include *Plane, Flight,* and *Pilot*.

During **object-oriented design** (or simply, object design) there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, a *Plane* software object may have a *tailNumber* attribute and a *getFlightHistory* method (see Figure 1.2).

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

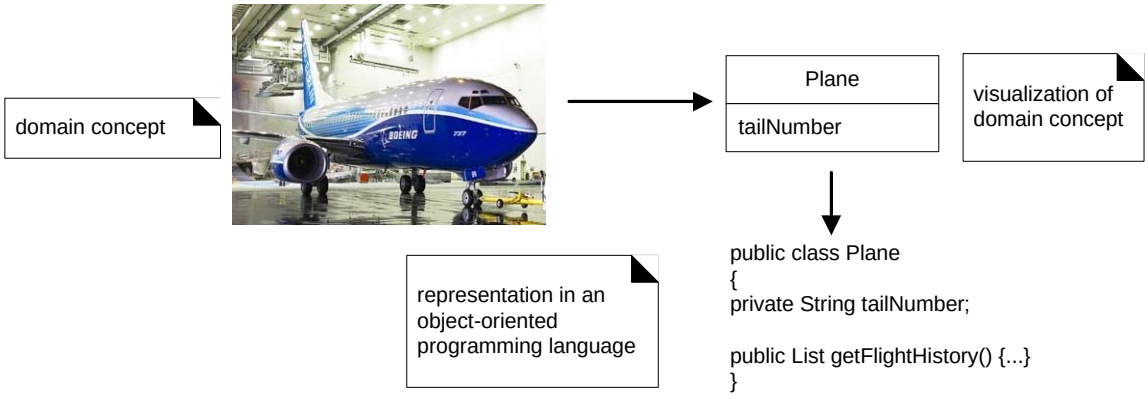


Figure 1.2 Object-orientation emphasizes representation of objects.

1.5 A Short Example

Before diving into the details of iterative development, requirements analysis, UML, and OOA/D, this section presents a bird's-eye view of a few key steps and diagrams, using a simple example—a "dice game" in which software simulates a player rolling two dice. If the total is seven, they win; otherwise, they lose.

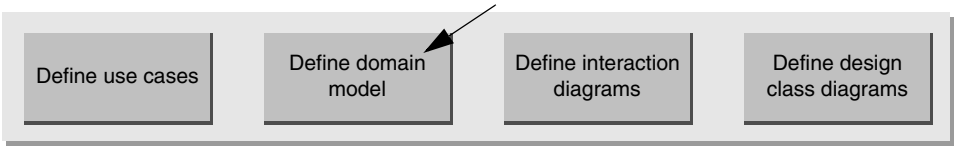
Define Use Cases

Requirements analysis may include stories or scenarios of how people use the application; 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. For example, here is a brief version of the *Play a Dice Game* use case:

> **Play a Dice Game**: Player requests to roll the dice. System presents results: If the dice face value totals seven, player wins; otherwise, player loses.

Define a Domain Model



Object-oriented analysis is concerned with creating a description of the domain from the perspective of objects. There is an identification of the concepts, attributes, and associations that are considered noteworthy.

The result can be expressed in a **domain model** that shows the *noteworthy* domain concepts or objects.

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

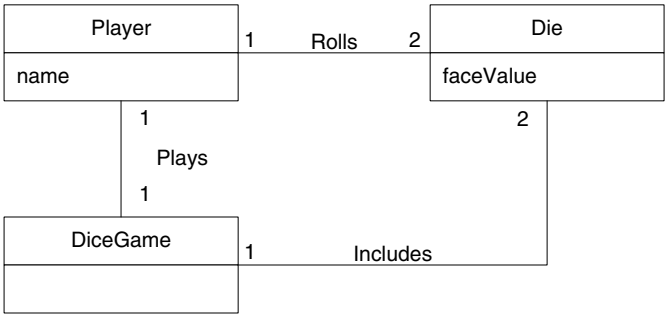


Figure 1.3 Partial domain model of the dice game.

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 the concepts or mental models of a real-world domain. Thus, it has also been called a **conceptual object model**.

Assign Object Responsibilities and Draw Interaction Diagrams

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

For example, the sequence diagram in Figure 1.4 illustrates an OO software design, by sending messages to instances of the *DiceGame* and *Die* classes. Note this illustrates a common real-world way the UML is applied: by sketching on a whiteboard.

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.

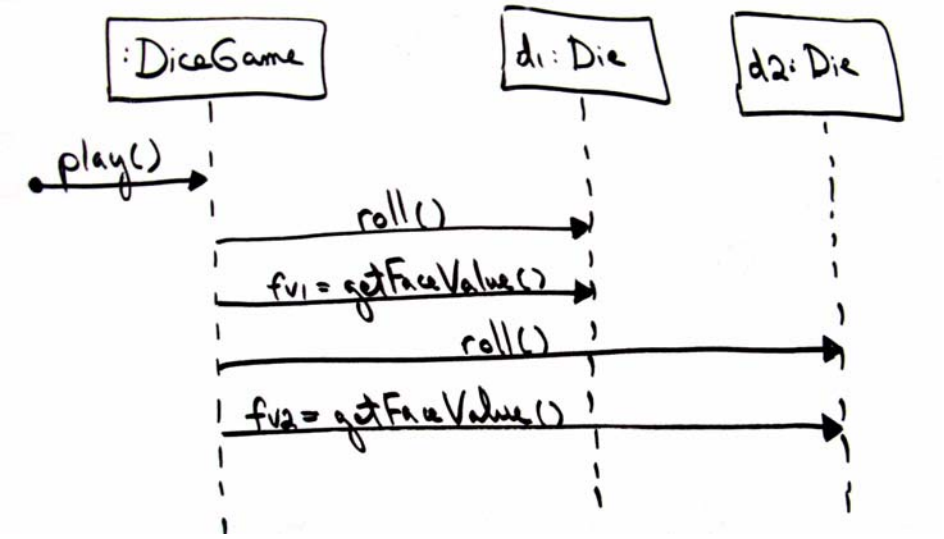


Figure 1.4 Sequence diagram illustrating messages between software objects.

Define Design Class Diagrams

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

For example, in the dice game, an inspection of the sequence 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 showing real-world classes, this diagram shows software classes.

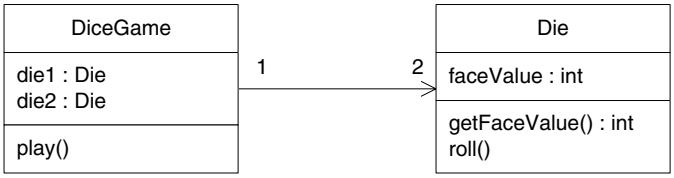


Figure 1.5 Partial design class diagram.

Notice that although this design class diagram is not the same as the domain model, some class names and content are similar. In this way, OO designs and languages can support a **lower representational gap** between the software components and our mental models of a domain. That improves comprehension.

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 What is the UML?

To quote:

> The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems [OMG03a].

The word *visual* in the definition is a key point—the UML is the de facto standard *diagramming notation* for drawing or presenting pictures (with some text) related to software—primarily OO software.

This book doesn't cover all minute aspects of the UML, a large body of notation. It focuses on frequently used diagrams, the most commonly used features within those, and core notation that is unlikely to change in future UML versions.

The UML defines various **UML profiles** that specialize subsets of the notation for common subject areas, such as diagramming Enterprise JavaBeans (with the *UML EJB profile*).

At a deeper level—primarily of interest to **Model Driven Architecture** (MDA) CASE tool vendors—underlying the UML notation is the **UML meta-model** that describes the semantics of the modeling elements. It isn't something a developer needs to learn.

Three Ways to Apply UML

In [Fowler03] three ways people apply UML are introduced:

- **UML as sketch**—Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.
- **UML as blueprint**—Relatively detailed design diagrams used either for 1) reverse engineering to visualize and better understanding existing code in UML diagrams, or for 2) code generation (forward engineering).

UML and "Silver Bullet" Thinking

There is a well-known paper from 1986 titled "No Silver Bullet" by Dr. Frederick Brooks, also published in his classic book *Mythical Man-Month* (20th anniversary edition). Recommended reading! An essential point is that it's a fundamental mistake (so far, endlessly repeated) to believe there is some special tool or technique in software that will make a dramatic order-of-magnitude difference in productivity, defect reduction, reliability, or simplicity. *And tools don't compensate for design ignorance*.

Yet, you will hear claims—usually from tool vendors—that drawing UML diagrams will make things much better; or, that Model Driven Architecture (MDA) tools based on UML will be the breakthrough silver bullet.

Reality-check time. The UML is simply a standard diagramming notation—boxes, lines, etc. Visual modeling with a common notation can be a great aid, but it is hardly as important as knowing how to design and think in objects. Such design knowledge is a very different and more important skill, and is not mastered by learning UML notation or using a CASE or MDA tool. A person not having good OO design and programming skills who draws UML is just drawing bad designs. I suggest the article *Death by UML Fever* [Bell04] (endorsed by the UML creator Grady Booch) for more on this subject, and also *What UML Is and Isn't* [Larman04].

Therefore, this book is an introduction to OOA/D and *applying* the UML to support skillful OO design.

- If reverse engineering, a UML tool reads the source or binaries and generates (typically) UML package, class, and sequence diagrams. These "blueprints" can help the reader understand the big-picture elements, structure, and collaborations.
- Before programming, some detailed diagrams can provide guidance for code generation (e.g., in Java), either manually or automatically with a tool. It's common that the diagrams are used for some code, and other code is filled in by a developer while coding (perhaps also applying UML sketching).

- **UML as programming language**—Complete executable specification of a software system in UML. Executable code will be automatically generated, but is not normally seen or modified by developers; one works only in the UML "programming language." This use of UML requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams), and is still under development in terms of theory, tool robustness and usability.

agile modeling p. 30

Agile modeling emphasizes *UML as sketch*; this is a common way to apply the UML, often with a high return on the investment of time (which is typically short). UML tools can be useful, but I encourage people to also consider an agile modeling approach to applying UML.

Three Perspectives to Apply UML

The UML describes raw diagram types, such as class diagrams and sequence diagrams. It does not superimpose a modeling perspective on these. For example, the same UML class diagram notation can be used to draw pictures of con-

cepts in the real world or software classes in Java.

This insight was emphasized in the Syntropy object-oriented method [CD94]. That is, the same notation may be used for three perspectives and types of models (Figure 1.6):

1. **Conceptual perspective**—the diagrams are interpreted as describing things in a situation of the real world or domain of interest.
2. **Specification (software) perspective**—the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).
3. **Implementation (software) perspective**—the diagrams describe software implementations in a particular technology (such as Java).

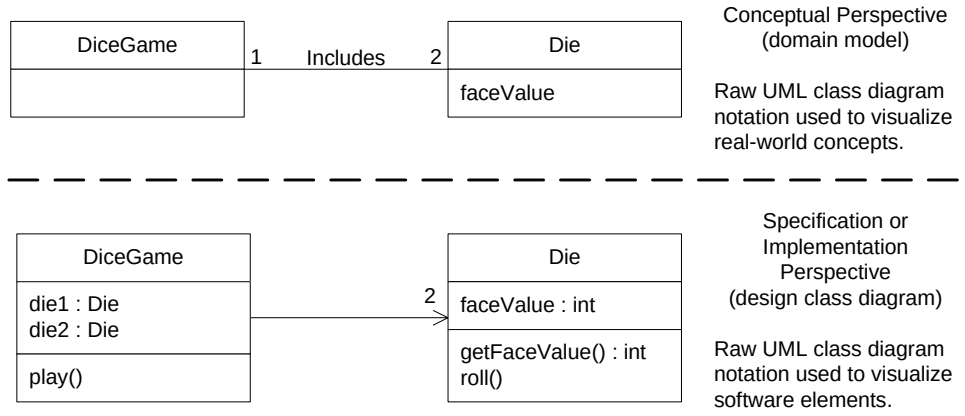


Figure 1.6 Different perspectives with UML.

We've already seen an example of this in Figure 1.3 and Figure 1.5, where the same UML class diagram notation is used to visualize a domain model and a design model.

In practice, the specification perspective (deferring the target technology, such as Java versus .NET) is seldom used for design; most software-oriented UML diagramming assumes an implementation perspective.

The Meaning of "Class" in Different Perspectives

In the raw UML, the rectangular boxes shown in Figure 1.6 are called **classes**, but this term encompasses a variety of phenomena—physical things, abstract concepts, software things, events, and so forth.[1]

A method superimposes alternative terminology on top of the raw UML. For example, in the UP, when the UML boxes are drawn in the Domain Model, they

are called **domain concepts** or **conceptual classes**; the Domain Model shows a conceptual perspective. In the UP, when UML boxes are drawn in the Design Model, they are called **design classes**; the Design Model shows a specification or implementation perspective, as desired by the modeler.

To keep things clear, this book will use class-related terms consistent with the UML and the UP, as follows:

- **Conceptual class**—real-world concept or thing. A conceptual or essential perspective. The UP Domain Model contains conceptual classes.
- **Software class**—a class representing a specification or implementation perspective of a software component, regardless of the process or method.
- **Implementation class**—a class implemented in a specific OO language such as Java.

UML 1 and UML 2

Towards the end of 2004 a major new release of the UML emerged, UML 2. This text is based on UML 2; indeed, the notation used here was carefully reviewed with key members of the UML 2 specification team.

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

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

1.7 Visual Modeling is a Good Thing

At the risk of stating the blindingly obvious, drawing or reading UML implies we are working more visually, exploiting our brain's strength to quickly grasp symbols, units, and relationships in (predominantly) 2D box-and-line notations.

This old, simple idea is often lost among all the UML details and tools. It shouldn't be! Diagrams help us see or explore more of the big picture and relationships between analysis or software elements, while allowing us to ignore or hide uninteresting details. That's the simple and essential value of the UML—or any diagramming language.

---

1. A UML class is a special case of the general UML model element **classifier**—something with structural features and/or behavior, including classes, actors, interfaces, and use cases.

1.8 History

The history of OOA/D has many branches, and this brief synopsis can't do justice to all the contributors. The 1960s and 1970s saw the emergence of OO programming languages, such as Simula and Smalltalk, with key contributors such as Kristen Nygaard and especially Alan Kay, the visionary computer scientist who founded Smalltalk. Kay coined the terms *object-oriented programming* and *personal computing,* and helped pull together the ideas of the modern PC while at Xerox PARC.[2]

But OOA/D was informal through that period, and it wasn't until 1982 that OOD emerged as a topic in its own right. This milestone came when Grady Booch (also a UML founder) wrote the first paper titled *Object-Oriented Design*, probably coining the term [Booch82]. Many other well-known OOA/D pioneers developed their ideas during the 1980s: Kent Beck, Peter Coad, Don Firesmith, Ivar Jacobson (a UML founder), Steve Mellor, Bertrand Meyer, Jim Rumbaugh (a UML founder), and Rebecca Wirfs-Brock, among others. Meyer published one of the early influential books, *Object-Oriented Software Construction,* in 1988. And Mellor and Schlaer published *Object-Oriented Systems Analysis*, coining the term *object-oriented analysis*, in the same year. Peter Coad created a complete OOA/D method in the late 1980s and published, in 1990 and 1991, the twin volumes *Object-Oriented Analysis* and *Object-Oriented Design*. Also in 1990, Wirfs-Brock and others described the responsibility-driven design approach to OOD in their popular *Designing Object-Oriented Software*. In 1991 two very popular OOA/D books were published. One described the OMT method, *Object-Oriented Modeling and Design*, by Rumbaugh et al. The other described the Booch method, *Object-Oriented Design with Applications*. In 1992, Jacobson published the popular *Object-Oriented Software Engineering*, which promoted not only OOA/D, but use cases for requirements.

The UML started as an effort by Booch and Rumbaugh in 1994 not only to create a common notation, but to combine their two methods—the Booch and OMT methods. Thus, the first public draft of what today is the UML was presented as the *Unified Method*. They were soon joined at Rational Corporation by Ivar Jacobson, the creator of the Objectory method, and as a group came to be known as the *three amigos*. It was at this point that they decided to reduce the scope of their effort, and focus on a common diagramming notation—the UML—rather than a common method. This was not only a de-scoping effort; the Object Management Group (OMG, an industry standards body for OO-related standards)

2. Kay started work on OO and the PC in the 1960s, while a graduate student. In December 1979—at the prompting of Apple's great Jef Raskin (the lead creator of the Mac)—Steve Jobs, co-founder and CEO of Apple, visited Alan Kay and research teams (including Dan Ingalls, the implementor of Kay's vision) at Xerox PARC for a demo of the Smalltalk personal computer. Stunned by what he saw—a graphical UI of bit-mapped overlapping windows, OO programming, and networked PCs—he returned to Apple with a new vision (the one Raskin hoped for), and the Apple Lisa and Macintosh were born.

was convinced by various tool vendors that an open standard was needed. Thus, the process opened up, and an OMG task force chaired by Mary Loomis and Jim Odell organized the initial effort leading to UML 1.0 in 1997. Many others contributed to the UML, perhaps most notably Cris Kobryn, a leader in its ongoing refinement.

The UML has emerged as the de facto and de jure standard diagramming notation for object-oriented modeling, and has continued to be refined in new OMG UML versions, available at www.omg.org or www.uml.org.

1.9 Recommended Resources

Various OOA/D texts are recommended in later chapters, in relation to specific subjects, such as OO design. The books in the history section are all worth study—and still applicable regarding their core advice.

A very readable and popular summary of essential UML notation is *UML Distilled* by Martin Fowler. Highly recommended; Fowler has written many useful books, with a practical and "agile" attitude.

For a detailed discussion of UML notation, *The Unified Modeling Language Reference Manual* by Rumbaugh is worthwhile. Note that this text isn't meant for learning how to do object modeling or OOA/D—it's a UML notation reference.

For the definitive description of the current version of the UML, see the on-line *UML Infrastructure Specification* and *UML Superstructure Specification* at www.uml.org or www.omg.org.

Visual UML modeling in an agile modeling spirit is described in *Agile Modeling* by Scott Ambler. See also www.agilemodeling.com.

There is a large collection of links to OOA/D methods at www.cetus-links.org and www.iturls.com (the large English "Software Engineering" subsection, rather than the Chinese section).

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 for those studying object design. However, it is not an introductory text and is best read after one is comfortable with the fundamentals of object design and programming. See also www.hillside.net and www.iturls.com (the English "Software Engineering" subsection) for links to many pattern sites.