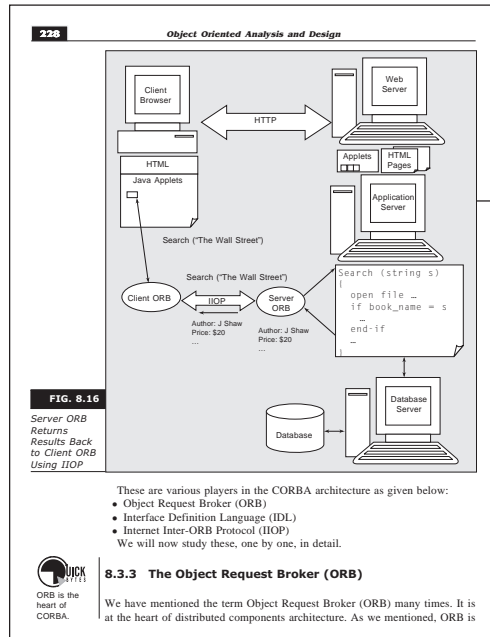


# Guided Tour



## ILLUSTRATIONS

More than 413 diagrams used to illustrate the concepts and methods described in the text.



Give additional snippets of information about the topics under discussion.

Unified Process Model 305

Actors send **messages** to the system for communication. They also receive messages from the system in response. By defining what actors do and what the system does, we can clearly identify and distinguish between the responsibilities of the actors and that of the system.

Use cases may also take upon themselves the job of defining **non-functional requirements**, such as the security, performance, accuracy, etc. For instance, there may be a (non-functional) use case specification stating that in the case of a (functional) *Funds transfer* use case, the system should perform the operation in 5 seconds.

We have discussed in detail how use cases get transformed into Analysis classes, Design classes; and therefore, in the Collaboration diagrams and Sequence diagrams. As such, we need not discuss it here again. The Unified Process specifies these steps in detail.

**10.4 UNIFIED PROCESS IS ARCHITECTURE-CENTRIC**

**10.4.1 Architecture Concepts**

Use cases drive the software development lifecycle in the Unified Process Model. However, use cases alone are not sufficient. We also need to concentrate on the architecture of the system. The architecture of the system is the uniform vision which is shared by all the people contributing the project, mainly the architects, designers, developers, and testers.

The most common examples of the model elements are subsystems, dependencies, interfaces, collaborations, nodes, classes, etc. They are the basis for the system to function properly at all times.

The concept of software architecture can be best understood by comparing it with that of the architecture of a building. When a building (say a house) is constructed, the architect prepares a layout of the proposed building. Why is this needed? This is basically to fit in the requirements specified by the prospective house owner in civil engineering terms. However, a more important use of this layout or blueprint is that all the concerned people (i.e. laborers, carpenters, bricklayers, plumbers, electricians and so on) understand what needs to be done clearly and in a uniform manner. This leaves very little scope for ambiguity. Everyone knows what the final goal is, and how best they can play their respective roles in order to achieve that goal. For instance, the electrician knows that wires must not be placed in an area where water pipes are present. This may not be possible, or at least obvious, without a layout in place.

**The system's architecture describes the model elements that are most crucial.**

Study how this discussion matches with modularity in software development.

its corresponding flow of control gets invoked. When the active object is destroyed, the corresponding flow of control is also terminated.

#### Use Case

A use case is a description of a set of actions that a system performs to produce certain output.

Examples of use case are:

- Insert an employee record
- Place an order
- Perform spell checking
- Determine examination result

A computer system cannot remain in isolation. It must interact with human or other actors that use the system for some purpose. The actors expect the system to show a particular behavior. A use case describes this behavior. As we have mentioned, it is a collection of the actions that are performed to show this behaviour.

A use case is shown as an oval containing the use case name. An example of this is shown in Fig. 9.14.

FIG. 9.14

Example of Use Case

Determine Examination Result

Use cases show what users of a system do.

Use cases are used to model the user requirements (i.e. the expected behavior of the system) during the analysis phase. During the analysis phase, the emphasis is on *what* is expected from a system, and not *how* that can be achieved. In this respect, use cases serve the purpose of describing what a system should do, regardless of the actual computer-based implementation of that requirement.

Use cases describe a how a system is expected to behave. For instance, in a funds-transfer application, you can specify the following high-level use case:

*The system should allow the user to transfer funds from one account to another successfully. If the operation fails, the system should go back to the original state (i.e. retain the original pre-transfer balances).*

*Note that without providing any more details regarding whether there should be two database tables involved, or whether a transaction is needed, etc. we have simply stated what a user wants. The actual implementation details would come later.*

Use cases enable the analysts, designers, developers and the intended users of the system to come to common conclusions regarding the requirements. Once the development begins, and the focus shifts from the analysis/design models to the actual program code, use cases also help the designers and the developers in ensuring that the requirements are getting correctly translated into program code. Use cases also play a major role in validating the system architecture.

### Note

A use case is a description of a set of

The text is interspersed with definitions and important notes related to the topics being discussed.

- *Partition testing* aims at reducing the number of test cases needed to test a class.
- Partition testing can be done by using three techniques: **State-based partitioning**, **Attribute-based partitioning**, and **Category-based partitioning**.
- The *state-based partitioning* technique divides the operations of a class depending on their capability to alter the state of the class.
- The *attribute-based partitioning* technique categorizes test cases based on the attributes of the class and the purposes for which they are used.
- The *category-based partitioning* technique partitions operations based on their purpose.
- **Interclass testing** means testing how classes work with each other to perform a given set of functions.

#### KEY TERMS AND CONCEPTS

• Attribute-based partitioning	• Basis path testing
• Black-box testing	• Category-based partitioning
• Class testing	• Class-Responsibility-Collaboration (CRC)
• Cluster testing	• Controllability
• Coverage	• Debugging
• Decomposability	• Dependent class
• Independent class	• Integration testing
• Interclass testing	• Observability
• Operability	• Partition testing
• Random testing	• Simplicity
• Software quality assurance	• Stability
• State-based partitioning	• System testing
• Test case	• Testability
• Tester	• Testing
• Thread	• Thread-based testing
• Understandability	• Unit testing
• Use-based testing	• White-box testing

#### PRACTICE EXERCISES

##### True/False Questions

1. Testing is the same as debugging.
2. Simplicity is one of the factors behind OO testing.
3. White-box testing means testing the internals of a class.
4. Black-box testing means testing the functionalities of a system.

#### KEY TERMS AND CONCEPTS

Gives the list of the important terms and concepts discussed in the chapter.

## CHAPTER SUMMARY

Gives the gist of the chapter in a few bulleted points.

### CHAPTER SUMMARY

- Traditionally, focus was on hardware. Over the last few years, it has changed to software.
- Software projects have become extremely complex and costly. Hence, older methodologies are no longer advisable.
- The earliest software approach was on the basis of **procedural programming**.
- **Modular programming** came later, where the programs were divided into smaller, more manageable modules.
- **Object technology** is a newer method of designing and developing software.
- Object technology introduces the concept of an **object**. An object contains both data (called as the **attributes** of an object) and its functionalities (called as **methods**).
- Objects interact with each other by passing **messages**.
- Generic form of objects, i.e. a template for similar kinds of objects is called as **class**.
- **Abstraction** is an important concept in object technology. It allows the designer to focus on the significant concepts and ignore unimportant details.
- **Encapsulation** protects the data in an object.
- The term **Object Oriented Analysis and Design (OOAD)** was coined in order to differentiate it from the classical analysis and design techniques, such as **Structured System Analysis and Design (SSAD)** and the Jackson Methodology.
- OOAD has evolved over several years.
- The early efforts in OOAD can be attributed to Grady Booch, Ivar Jacobson, and James Rumbaugh.
- Of late, a combination of these methodologies has become popular, called as the **Rational Unified Process (RUP)** model.
- The **fountain model** provides some insights into the working of OOAD.

### KEY TERMS AND CONCEPTS

- |                  |                       |
|------------------|-----------------------|
| ☞ Abstract       | ☞ Functional model    |
| ☞ Abstraction    | ☞ Jacobson method     |
| ☞ Attribute      | ☞ Logical model       |
| ☞ Booch method   | ☞ Message             |
| ☞ Class          | ☞ Method              |
| ☞ Dynamic model  | ☞ Modular programming |
| ☞ Encapsulation  | ☞ Model               |
| ☞ Fountain model | ☞ Object              |

### Detailed Questions

1. What is the meaning of analysis?
2. Why do we need the analysis phase?
3. What is modeling?
4. Discuss the idea of functional modeling with the help of DFDs.
5. What is a functional model?
6. D
7. W
8. E
9. D
10. H

### PRACTICE EXERCISES

#### True/False Questions

1. Analysis phase comes before the design phase.
2. Analysis phase is closely tied to programming languages.
3. It is not necessary to consider the surrounding environment when analysing a system or its requirements.
4. Modeling is a part of analysis.
5. Problem statement is important in the analysis phase.
6. Object model is dynamic in nature.
7. Functional model deals with the flow of data in a system.
8. Dynamic model represents the state of a system with respect to time.
9. Use cases represent the actions of users.
10. Actors deal with use cases.

#### Multiple Choice Questions

1. The first phase in software development is \_\_\_\_\_.  
(a) analysis (b) design (c) construction (d) testing
2. \_\_\_\_\_ are closely associated with the analysis phase.  
(a) Programmers (b) Implementers (c) Testers (d) Users
3. The next logical step after analysis is \_\_\_\_\_.  
(a) implementation (b) design  
(c) construction (d) testing
4. The mapping of user actions to something that can be later translated into software is called as \_\_\_\_\_.  
(a) detailed design (b) modeling  
(c) testing (d) debugging
5. In \_\_\_\_\_ model, we identify the classes and objects of a system.  
(a) object (b) dynamic (c) functional (d) class
6. In \_\_\_\_\_ model, we identify the time-related behaviour of a system.  
(a) object (b) dynamic (c) functional (d) class
7. In \_\_\_\_\_ model, we identify the data flow of a system.  
(a) object (b) dynamic (c) functional (d) class
8. Data flow diagrams help us in \_\_\_\_\_.  
(a) time related sequencing of events (b) modeling data  
(c) identifying classes (d) identifying methods
9. \_\_\_\_\_ classes represent the line between internal objects and external entities of a system.  
(a) Entity (b) Boundary (c) Control (d) Static
10. \_\_\_\_\_ classes control the interaction between classes.  
(a) Entity (b) Boundary (c) Control (d) Static

## PRACTICE EXERCISES

A question-answer section that contains True/False Questions, Multiple Choice Questions and Detailed Questions.

A  
P  
P  
E  
N  
D  
I  
X

## Algorithms and Flow Charts

**A**

**A.1** INTRODUCTION

The way in which we design algorithms and flow charts will be a good deal of help.

A  
P  
P  
E  
N  
D  
I  
X

## Software Engineering and Methodology

**B**

**B.1** SOFTWARE ENGINEERING

From many years of professional work, it has become clear that the development of software is a very complex task. It is not enough to write a program; one must also know how to design it, how to test it, and how to maintain it.

A  
P  
P  
E  
N  
D  
I  
X

## Programming Languages

**C**

**C.1** NEED FOR PROGRAMMING LANGUAGES

We shall study the concept of the level of a language in this appendix. For instance, *Make tea* is a very high-level instruction. It involves many smaller tasks and still more sub-tasks, in fact. We assume that the person who has been told to make tea need not have any further explanations such as taking a teacup, adding sugar, tea, milk and so on. Similarly, *Brush teeth* is a high-level instruction for a child. For a kid, it might be clear. If the child is very small, we initially instruct him to do the smallest operations such as *Left right hand*, *Walk a step*, *Turn left*, etc. After a while, we sequence the detailed instructions for him; for example, *Take a step*, *Turn right*, *Walk until the house*, *Left hand*, *Take drink*, etc. After he has learned the process, we start instructing him in a higher-level language: *Brush your teeth* etc. Thus, we first establish a particular level of language. A computer is built to perform very primitive tasks such as adding two numbers, comparing them, moving a few bytes of data from one memory location to another, and so on. In this sense, it is like a small child who knows how to do very basic operations such as moving one step ahead, picking up the brush and so on. Once a computer knows how to perform these operations, we can build a more complicated system on top of it.

A  
P  
P  
E  
N  
D  
I  
X

## Case Study on OO Systems

**D**

**D.1** INTRODUCTION

We shall summarize the key OO principles in this case study for the benefit of readers who want to get a quick understanding of the concepts behind OO and then work on a case study to illustrate some of these principles. The theory covered in this appendix is explained in detail in the main chapters of the book. However, the reason it is summarized once again is on two counts: (a) Provide a recap, and (b) Provide an easy way for the busy reader to understand the main concepts in OO quickly.

Of late, **object technology** has become the de facto standard for developing most of the modern software applications. The word *object technology* may create some fear and mystery. However, it is more *natural* than all the other methods. All the technologies for creating dynamic and active Web pages are based on the object technology. The object technology has become even more popular with the ever-increasing usage of the object-oriented programming language Java. Java is the basis for many dynamic Web page technologies such as servlets, JSP and EJB. Active documents began with Java.

*The world around us is made up of objects. An object could be natural, or humans can create it. Objects can be described, categorized.*

- ## APPENDICES
- **Appendix A** discusses the concepts in algorithms and flow charts.
  - **Appendix B** covers the topics in software engineering and methodology.
  - **Appendix C** deals with the basics of programming languages.
  - **Appendix D** details a case study on OO systems.