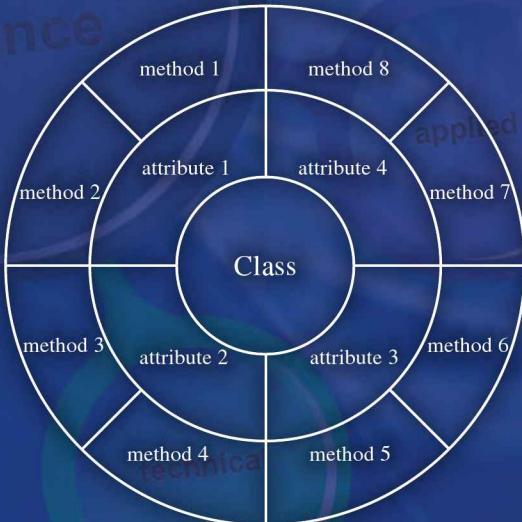


Eastern
Economy
Edition

MAGNIFYING

Object-Oriented Analysis and Design



Arpita Gopal • Netra Patil

Magnifying Object-Oriented Analysis and Design

Arpita Gopal

Director-MCA

*Sinhgad Institute of Business Administration and Research
Pune*

Netra Patil

Assistant Professor

*Sinhgad Institute of Business Administration and Research
Pune*

PHI Learning Private Limited

New Delhi - 110001
2011

MAGNIFYING OBJECT-ORIENTED ANALYSIS AND DESIGN
Arpita Gopal and Netra Patil

© 2011 by PHI Learning Private Limited, New Delhi. All rights reserved. No part of this book may be reproduced in any form, by mimeograph or any other means, without permission in writing from the publisher.

ISBN-978-81-203-4068-8

The export rights of this book are vested solely with the publisher.

Published by Asoke K. Ghosh, PHI Learning Private Limited, M-97, Connaught Circus, New Delhi-110001 and Printed by Baba Barkha Nath Printers, Bahadurgarh, Haryana-124507.

To
Shri M.N. Navale
The Visionary
Founder President
Sinhgad Technical Education Society
Pune, Maharashtra
India

Contents

<i>Preface</i>	xiii
CHAPTER 1: SYSTEM ANALYSIS AND DESIGN	1–14
1.1 Introduction	1
1.2 Software Development Models.....	2
1.2.1 Waterfall Model	2
1.2.2 The Incremental Model	3
1.2.3 Spiral Model	4
1.3 Software Development Techniques	5
1.3.1 Prototyping	5
1.3.2 Clean-room Technique.....	6
1.3.3 Object-Oriented Technique	6
1.4 Structured Systems Analysis and Design (SSAD)	8
1.4.1 Features of SSAD	9
1.4.2 The Traditional Way of Programming—Structured Programming	10
1.4.3 SSADM Disadvantage	12
1.4.4 Why is SSADM not Evacuated?	12
1.5 Object-Oriented Analysis and Design (OOAD)	12
1.5.1 The New Way of Programming—The Object-Oriented Approach	13
1.5.2 Why an OOAD over SSADM?	13
1.5.3 Contribution of Object-Oriented Methodology towards SDLC	14
<i>Exercises</i>	14
CHAPTER 2: OBJECT ORIENTED ANALYSIS AND DESIGN	15–24
2.1 Object-Oriented Modelling	15
2.1.1 Analysis Model	15
2.1.2 Architecture Model	15
2.1.3 Component (Design) Models	15
2.2 Object-Oriented Approach	16
2.2.1 Object Orientation	16
2.2.2 Object-Oriented Analysis.....	17
2.2.3 Object-Oriented Design	18

2.3	The Constituents of OOAD	18
2.3.1	Objects and Classes	18
2.3.2	Links and Association	19
2.3.3	Generalization and Specialization	19
2.3.4	Aggregation and Composition	20
2.4	Pillars of Object Oriented Analysis and Design	20
2.4.1	Abstraction	20
2.4.2	Encapsulation	20
2.4.3	Inheritance	21
2.4.4	Polymorphism	21
2.4.5	Coupling	21
2.4.6	Cohesion	22
2.4.7	Components	22
2.4.8	Interfaces	22
2.5	The Language of OOAD—Unified Modelling Language and BPMN	22
2.5.1	UML Diagrams	23
2.5.2	Introduction to BPMN	24
	<i>Exercises</i>	24

CHAPTER 3: BUSINESS PROCESS DIAGRAM AND USE CASE DIAGRAM25-47

3.1	Introduction to Business Process Diagram	25
3.2	Scope of Business Process Diagram	25
3.3	Elements of Business Process Diagram	26
3.3.1	Flow Objects	26
3.3.2	Connecting Objects	28
3.3.3	Swimlanes	28
3.3.4	Artifacts	30
3.4	Guidelines for Design of Business Process Diagram	31
3.4.1	Finding out Elementary (Atomic) Processes and Complex (Non-atomic) Processes	31
3.4.2	Example Explaining Elementary and Complex Processes	31
3.4.3	Represent Processes along the Pools and Lanes	32
3.4.4	Check for Consistency within the Diagram	32
3.4.5	Analyze the Level of Interaction	32
3.5	Case: Get-Well-Soon Hospital Pvt. Ltd.	32
3.5.1	Swimlanes	33
3.5.2	Flow Objects	34
3.5.3	Artifacts	35
3.5.4	Business Process Diagram	36
3.6	Use-Case Diagram	36
3.7	Scope of Use-Case Diagrams	36
3.8	Benefits of a Use-Case Diagrams	38

3.9	Elements of Use-Case Diagram	39
3.9.1	Actors	39
3.9.2	Use-Cases	39
3.9.3	Relationships between Actor and Use Case	40
3.9.4	Relationships between Use-Cases	40
3.9.5	Relationships between Actors	42
3.10	Guidelines for Design of Use-Case Diagrams	42
3.10.1	Actors	42
3.10.2	Use-Case	42
3.10.3	Relationships	42
3.10.4	System Boundary Box	43
3.11	Case Merchant National Bank	43
3.11.1	Identifying Actors and Use-Cases	43
3.11.2	Use-Case Diagram	43
	<i>Exercises</i>	45

CHAPTER 4: CLASS DIAGRAM AND OBJECT DIAGRAM 48–66

4.1	Class Diagrams	48
4.1.1	Analysis and Design Versions of a Class Diagram	48
4.2	Elements of Class Diagram	49
4.2.1	Class	49
4.2.2	Relationships	50
4.2.3	Association Class	55
4.2.4	Interface	56
4.2.5	Package	58
4.3	Guidelines for Design of a Class Diagram	58
4.4	Problem Statement: Merchant National Bank	58
4.4.1	Analysis of Merchant National Bank	59
4.4.2	Class Diagram—Merchant National Bank	61
4.5	Object Diagram	62
4.6	Elements of Object Diagram	62
4.6.1	Objects	62
4.6.2	Links	63
4.7	Guidelines for Design of Object Diagrams	63
4.8	Problem Statement: ABC India Pvt. Ltd.	63
4.8.1	Analysis of ABC India Pvt. Ltd.	63
4.8.2	Object Diagrams for ABC India Pvt. Ltd.	64
	<i>Exercises</i>	65

CHAPTER 5: SEQUENCE DIAGRAM AND COLLABORATION DIAGRAM 67–81

5.1	Introduction to Sequence Diagram	67
5.2	Elements of Sequence Diagram	68
5.2.1	Life Lines	68
5.2.2	Messages	68

5.2.3	Activation	69
5.2.4	Guards	69
5.2.5	Combined Fragments	70
5.2.6	Objects	72
5.3	Guidelines for Design of Sequence Diagrams	74
5.4	Problem Statement: Milton Jewels Pvt. Ltd.....	75
5.4.1	Analysis of Milton Jewels Pvt. Ltd.	75
5.4.2	Sequence Diagrams: Milton Jewels Pvt. Ltd.	76
5.5	Collaboration Diagram	77
5.6	Elements of Collaboration Diagram.....	78
5.6.1	Links	78
5.6.2	Messages.....	78
5.6.3	Objects	78
5.7	Guidelines for Design of Collaboration Diagrams	79
5.8	Problem Statement: Milton Jewels Pvt. Ltd.....	79
5.8.1	Analysis of Milton Jewels Pvt. Ltd.	79
5.8.2	Collaboration Diagram: Milton Jewels Pvt. Ltd.	79
	<i>Exercises</i>	80

CHAPTER 6: ACTIVITY DIAGRAM AND STATE CHART DIAGRAM 82–97

6.1	Introduction to Activity Diagram	82
6.2	Elements of Activity Diagram and their Notation	82
6.2.1	Initial State	82
6.2.2	Final State	83
6.2.3	Action/Activity	83
6.2.4	Transitions	83
6.2.5	Decisions	84
6.2.6	Synchronization, Fork and Join	84
6.2.7	Swimlanes	85
6.2.8	Objects and Object Flows	86
6.3	Guidelines for Design of Activity Diagram	86
6.4	Problem Statement: MCA Admission System	87
6.4.1	Analysis of MCA Admission System	87
6.4.2	Activity Diagrams: MCA Admission System	88
6.5	Introduction to State Chart Diagram	89
6.6	Elements of State Chart Diagram and their Notation	90
6.6.1	Initial State	90
6.6.2	Final State	90
6.6.3	State	90
6.6.4	Transitions	93
6.7	Guidelines for Design of Statechart Diagram	94
6.8	Problem Statement: Advertisement Campaign of ABC Pvt. Ltd.	95
6.8.1	Analysis of Advertisement Campaign of ABC Pvt. Ltd.	95
6.8.2	State Chart Diagram for Advertisement Campaign of ABC Pvt. Ltd.	96
	<i>Exercises</i>	97

CHAPTER 7: COMPONENT DIAGRAM AND DEPLOYMENT DIAGRAM 98–111

7.1	Introduction to Component Diagram	98
7.2	Elements of Component Diagram and their Notation	98
7.2.1	Components	98
7.2.2	Interfaces	100
7.2.3	Dependencies	102
7.2.4	Port	102
7.3	Guidelines for Design of Component Diagram	103
7.4	Problem Statement: GlobalTherm USA Pvt. Ltd.....	104
7.4.1	Analysis of GlobalTherm USA Pvt. Ltd.	104
7.4.2	Component Diagram for GlobalTherm USA Pvt. Ltd.	105
7.5	Introduction to Deployment Diagram	105
7.6	Elements of Deployment Diagram and their Notation	106
7.6.1	Node	106
7.6.2	Component.....	106
7.6.3	Artifacts	106
7.6.4	Link	107
7.6.5	Dependency	107
7.7	Guidelines for Design of Deployment Diagram	107
7.8	Deployment Diagram for Web-Based Application	108
7.8.1	Simplified Deployment Diagram	108
7.8.2	Detailed Deployment Diagram	108
7.9	Problem Statement: John and Tom's Online Beverages	108
7.9.1	Analysis of John and Tom's Online Beverages	110
7.9.2	Deployment Diagram for John and Tom's Online Beverages	110
7.10	Simplified Deployment Diagram for 2-Tier Architecture	110
	<i>Exercises</i>	111

CHAPTER 8: CASE STUDY—STUDENT LOAN SYSTEM 112–148

8.1	Problem Statement: Student Loan System	112
8.2	Analysis Phase Diagrams: Student Loan System	112
8.2.1	Business Process Diagram.....	113
8.2.2	Use-Case Diagram	116
8.2.3	Class Diagram	118
8.2.4	Object Diagram	120
8.3	Design Phase Diagrams: Student Loan System	120
8.3.1	Sequence Diagram	120
8.3.2	Collaboration Diagram.....	134
8.3.3	Statechart Diagram	141
8.3.4	Activity Diagrams	143
8.4	Implementation Phase Diagrams: Student Loan System	144
8.4.1	Component Diagram	144
8.4.2	Deployment Diagram	147

CHAPTER 9: CASE STUDY—ON LINE TRADING OF SECURITIES 149–181

9.1	Problem Statement: Online Trading of Securities	149
9.2	Analysis Phase Diagrams: Online Trading of Securities	150
9.2.1	Business Process Diagram.....	150
9.2.2	Use-Case Diagrams	152
9.2.3	Class Diagram: Analysis Phase	155
9.3	Design Phase Diagrams: Online Trading of Securities	158
9.3.1	Sequence Diagrams.....	158
9.3.2	Collaboration Diagram.....	168
9.3.3	Statechart Diagram.....	173
9.3.4	Activity Diagrams	174
9.4	Implementation Diagrams: Online Trading of Securities	178
9.4.1	Component Diagram	178
9.4.2	Deployment Diagram	178

CHAPTER 10: CREDIT CARD MANAGEMENT SYSTEM 182–228

10.1	Problem Statement: Credit Card Management System	182
10.1.1	Application for Credit Card	182
10.1.2	Produce Monthly Statement	182
10.1.3	Customer Information Maintenance	183
10.1.4	Credit Card Transaction Recording	183
10.1.5	Transaction and Statement Information Checking	183
10.2	Analysis Phase Diagrams: Credit Card Management System	184
10.2.1	Business Process Diagram.....	184
10.2.2	Use-Case Diagram	189
10.2.3	Class Diagram	192
10.3	Design Phase Diagrams: Credit Card Management System	194
10.3.1	Sequence Diagram	194
10.3.2	Collaboration Diagram.....	206
10.3.3	Statechart Diagram.....	212
10.3.4	Activity Diagram	213
10.4	Implementation Phase Diagrams: Credit Card Management System	225
10.4.1	Component Diagram	225
10.4.2	Deployment Diagram	226

CHAPTER 11: WAREHOUSE MANAGEMENT SYSTEM 229–263

11.1	Problem Statement: Warehouse Management System	229
11.2	Analysis Phase Diagrams: Warehouse Management System	230
11.2.1	Business Process Diagram.....	230
11.2.2	Use-Case Diagram	235
11.2.3	Class Diagram	238

11.3	Design Phase Diagrams: Warehouse Management System	239
11.3.1	Sequence Diagram	239
11.3.2	Collaboration Diagram.....	251
11.3.3	Statechart diagram	258
11.3.4	Activity Diagram.....	258
11.4	Implementation Phase Diagrams: Warehouse Management System	260
11.4.1	Component Diagram	260
11.4.2	Deployment Diagram	261
CHAPTER 12: EXISTING OBJECT ORIENTED METHODOLOGIES		264–282
12.1	Object Oriented Methodologies and UML	264
12.2	Different Models for Object Analysis and Design	265
12.3	Rumbaugh's Object Modelling Technique (OMT) Methodology	265
12.3.1	Object Model.....	265
12.3.2	Object Modelling Notations	265
12.3.3	Dynamic Model.....	266
12.3.4	Dynamic Modelling Notation	266
12.3.5	Functional Model	267
12.3.6	Functional Model Notation	267
12.4	Booch Methodology	267
12.4.1	Booch's Static Diagrams	269
12.4.2	Booch's Dynamic Diagrams	275
12.4.3	Booch Methodology Process.....	276
12.5	The Coad–Yourdon Methodology	277
12.5.1	Coad–Yourdon Diagrams	277
12.6	Jacobson's Object-Oriented Software Engineering	279
12.6.1	Requirements Model	280
12.6.2	Analysis Model	280
12.6.3	Design Model	281
12.6.4	Implementation Model	281
12.6.5	Testing Model	281
12.7	Comparison of Various Object-Oriented Methodologies.....	281
	<i>Exercises</i>	281
	<i>Index.....</i>	283–287

Preface

The art of building software is embodied in understanding how to abstract and model essential elements of the business and how to use this abstract to design software solutions.

Process modelling is an important part of business design, business redesign, or business reengineering. It is merely a tool that provides a means of communicating complex business functions in a form more easily understandable by people. Modelling provides for the formalization of processes which in turn allows the business to operate in a standardized manner. Effective design of business processes allows individuals to work together more efficiently. Modelling of lower level (elementary) processes provides consensus on business rules.

Process models are an aid to the understanding of the nature in which processes dynamically work with data. From a design perspective, modelling provides explicit guidelines for modularity, reusability, flexibility and integrity.

The object-oriented (OO) paradigm provides a powerful and effective environment for analyzing, designing, and implementing flexible and robust real-world systems, offering benefits such as encapsulation (information hiding), polymorphism, inheritance and reusability.

The OO methods provide their own representational notations for constructing a set of models during the development life cycle for a given system. They provide techniques and constructs to model an information processing system in terms of its data and the processes that act on those data. OO models focus on objects while the traditional system analysis and design methodology focus on processes. Moreover, the fundamental difference is that while OO models tend to focus on structure, system analysis and design models tend to emphasize behaviours or processes. One of the main benefits of the OO approach is that it provides a continuum of representation from analysis to design to implementation, thus engendering a seamless transition from one model to another.

Object-oriented modelling has come a long way and has changed the ways software is now-a-days developed. A firm grounding in the theory of object-oriented analysis and design and practical application is essential for understanding how to build good software.

The text provides practical guidance to the students by giving them a framework for building software following OOAD methodologies.

The case-oriented approach of the book is unique as it gives precise frameworks for object-oriented modelling using which students will be able to translate the complexities of the business accurately into design and implement the software requirement specifications of the systems into a working version.

Special features of Magnifying OOAD include:

1. Orientation towards realistic application of object-oriented analysis and design methodology.
2. Framework for analysis of business cases, followed by design of software solutions for them.
3. Implementation models along with its software components and deployment specifications.
4. Fifteen fully solved case studies encompassing a diverse selection of business domains, analysis and design models completely depicting the systems through the structured framework.
5. Introduction to the standard notation used in system and software development, the UML and BPMN.

Chapter 1 introduces various software development models and techniques. It further elaborates the traditional system analysis and design methodology of software development and then introduces the reader to object-oriented analysis and design.

Chapter 2 describes the object-oriented methodology in detail. It introduces the reader to the concepts of class, object etc. It further describes the various categories of models used in object-oriented methodology such as analysis models, design models and implementation models. The chapter also explains the basic principles of object-oriented modelling.

Chapters 3 and 4 explain analysis models with the help of business process diagrams, use-case diagram, class diagram and object diagrams. For each diagram, the purpose, the notations, and the guidelines are given for easy understanding. Every diagram is further explained by taking a business case as an example and modelling a solution for it.

Chapters 5 and 6 explain the design models with the help of sequence diagrams, collaboration diagrams, statechart diagrams and activity diagrams. The same approach of case-based explanation is followed to make it easy for the reader to understand.

Chapter 7 explains implementation models with the help component and deployment diagrams.

Chapters 8, 9, 10 and 11 elaborate case studies encompassing a diverse selection of business domains. Analysis of the business domain, developing analysis models, proposing design solutions using design models and further implementation models orient the reader towards realistic application of object-oriented analysis and design methodology.

The last chapter explains the most common existing methodologies of modelling using object-oriented analysis and design.

Though object-oriented way of software development is not new, many readers are still unaware of effective and correct usage of object-oriented modelling techniques. The following categories of readers may find this book of great value:

- Software developers and students who are having experience in object-oriented programming, but are relatively new to object-oriented analysis and design.
- Students of computer science or software engineering courses studying object-oriented technology.
- Experienced OOAD practitioners for a practical perspective of object-oriented modelling.

Acknowledgements

The most pleasurable aspect of writing a book is the opportunity it provides to thank God and for giving us the strength and fortitude to complete each and every project with utmost passion and dedication.

We would also like to thank our mentors and gurus Prof. M.N. Navale and Prof. S.N. Nawale for having facilitated our efforts by making available all the necessities that a writer could dream of.

Arpita Gopal expresses her heartfelt thanks to her husband Nirbhay Gopal and daughters Arushi and Ira for accommodating her very busy schedule and providing her with moral strength.

Netra Patil takes this opportunity to thank her husband Prashant and daughter Tejas for their continuous encouragement and cooperation throughout the completion of this book.

**ARPITA GOPAL
NETRA PATIL**

C|H|A|P|T|E|R

1

System Analysis and Design

1.1 INTRODUCTION

An information systems is about building a computer-based system to help the user operate a business, make decisions effectively and manage an enterprise successfully.

System analysis and design refers to the process of examining a business situation and suggesting improvements in it by the way of better procedures and methods.

Software system development consists of two major components: *system analysis* and *system design*. The system design includes planning an alternative system to improve the existing system. Before this planning can be done, the existing system must be thoroughly understood and how computers can best be used to improve the working of the existing system should be analyzed. The system analysis is thus the process of gathering and interpreting facts, diagnosing problems and using this information to recommend improvements to the system.

There are many kinds of software development methodologies which form the framework for planning and controlling the creation of an information system. By the term methodology we mean one of the software development models used for development of software along with one or more techniques used in its development, i.e.

$$\text{METHODOLOGY} = \text{MODEL} + \text{TECHNIQUE(S)}.$$

Table 1.1 categorizes different types of software development models and techniques.

TABLE 1.1 Software development models and techniques

Software development models

- Waterfall
- Incremental
- Spiral

Software development techniques

- Prototype
 - Clean-room
 - Object-oriented
-

Prototyping, clean-room, and object-oriented are ways and techniques for implementing information system development using the waterfall, incremental and spiral models. These

techniques may be mixed and matched as per our requirement even in a single project. Also, a technique can be used in parts, i.e. an individual technique may be used without using all aspects of that technique. This means that a project using the spiral model may combine prototyping with object-oriented analysis and design and also use clean-room testing techniques.

1.2 SOFTWARE DEVELOPMENT MODELS

Earlier in the process of software development, code was developed first and then debugged. There was no formal design or analysis approach. This code and debug approach was not found optimal as complexity of software systems increased. As the time progressed the approach to developing complex hardware systems was well understood, and gradually different models for developing software evolved. Discussed below are a few models used for software development.

1.2.1 Waterfall Model

The waterfall approach to information system development emphasizes completing a phase of the development before proceeding to the next phase. After completion of the certain phases, a baseline is established that “freezes” the products of the development at that point. If need of a change is identified, a formal change process is followed to make the change. The graphic representation of these phases in software development resembles the downward flow of a waterfall. Figure 1.1 depicts the phases of the waterfall model.

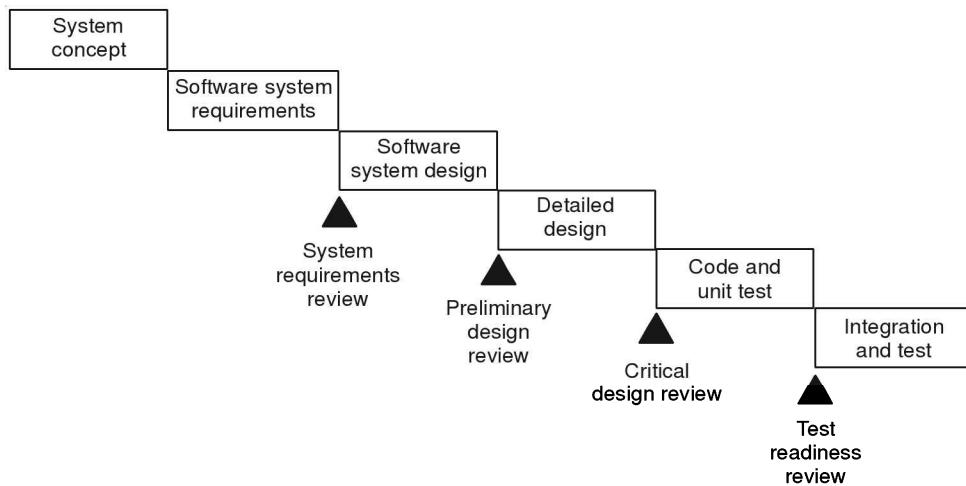


FIGURE 1.1 Waterfall model.

Each box in Figure 1.1 represents a phase. Output from each phase includes documentation. The phases below the detailed design phase include software as part of their output. Transition from phase to phase is accomplished by holding formal reviews which provide insight into the progress. Baselines are established at critical points on the waterfall model, the last of which is the product baseline. The final baseline is accompanied by audits.

The application most suitable for the use of the waterfall model should be limited to situations where the requirements and the implementation of those requirements are very well understood.

For example, if a company has experience in building accounting systems, I/O controllers, or compilers, then building another such product based on the existing designs is best managed with the waterfall model.

1.2.2 The Incremental Model

The incremental model performs the waterfall in overlapping sections (see Figure 1.2), attempting to compensate for the length of waterfall model projects by producing usable functionality earlier. The project using the incremental model may start with general objectives, later some portion of these objectives are defined as requirements and are implemented, followed by the next portion of the objectives until all objectives are implemented.

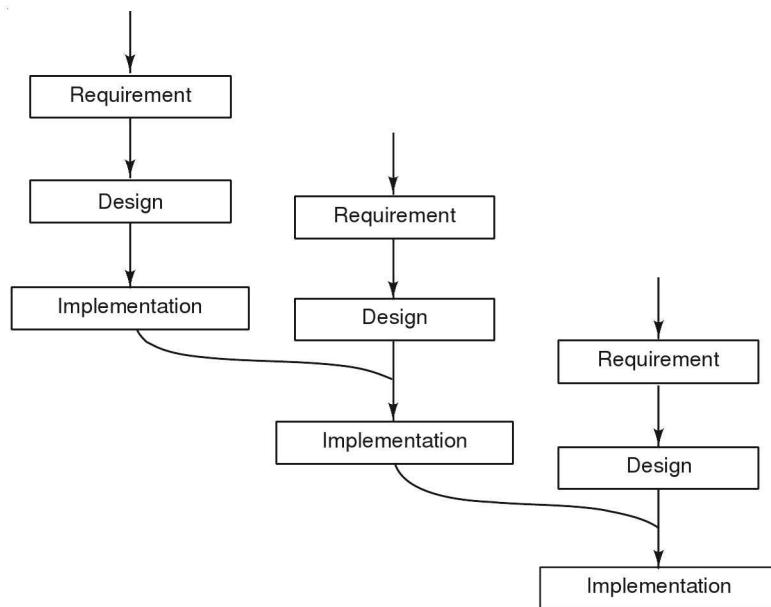


FIGURE 1.2 Incremental model.

But this approach of use of general objectives initially, rather than complete requirements can be uncomfortable for management as some modules will be completed long before the others. Another difficulty is that formal reviews and audits are more difficult to implement on

increments than on a complete system. Additionally, there can be a tendency to push difficult problems to the future to demonstrate early success to management. To overcome these problems, well-defined interfaces are required.

Use of the incremental model is appropriate when, it is too risky to develop the whole system at once.

1.2.3 Spiral Model

The spiral model (Figure 1.3) is proposed by Barry Boehm in which prototyping is used to control cost. The Boehm spiral model has become quite popular among ADE (Aerospace, Defense and Engineering) specialists, and is not so familiar among business developers. It is particularly useful in projects, that are risky in nature. Business projects are more conservative. They tend to use mature technology and to work well-known problems.

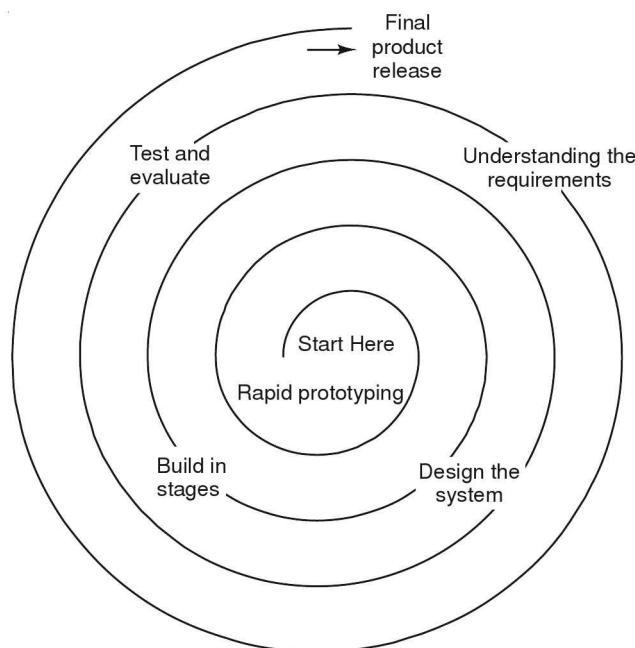


FIGURE 1.3 Spiral model.

Table 1.2 summarizes the strengths and weaknesses of the waterfall, incremental, and Boehm spiral models.

TABLE 1.2 Strengths and weaknesses of models

	<i>Waterfall model</i>	<i>Incremental model</i>	<i>Boehm spiral model</i>
Strengths			
Allows for work force specialization	×	×	×
Orderliness appeals to management	×	×	×
Can be reported about	×	×	×
Facilitates allocation of resources	×	×	×
Early functionality		×	×
Does not require a complete set of requirements at the onset		×	×
Resources can be held constant		×	
Control costs and risk through prototyping			×
Weaknesses			
Requires a complete set of requirements at the onset	×		
Enforcement of non-implementation attitude hampers analyst/designer communications	×		
Beginning with less defined general objectives may be uncomfortable for management		×	×
Requires clean interfaces between modules		×	
Incompatibility with a formal review and audit procedure		×	×
Tendency for difficult problems to be pushed to the future so that the initial promise of the first increment is not met by subsequent products		×	×

The incremental model may be used with a complete set of requirements or with less defined general objectives.

1.3 SOFTWARE DEVELOPMENT TECHNIQUES

1.3.1 Prototyping

Prototyping is the process of building a working replica of a system. The prototype is the equivalent of a small-scale model of the real world. Prototyping may be used with the waterfall; it can be useful to demonstrate technical feasibility when the technical risk is high or projects where requirements are not clear in the beginning, to better understand and extract user requirements. In either case, the goal is to limit cost by understanding the problem bit by bit before committing more resources.

Table 1.3 presents the strengths and weaknesses of prototyping.

TABLE 1.3 Strengths and weaknesses of prototyping

Strengths of prototyping
<ul style="list-style-type: none"> • Offers early functionality. • Gives a process to perfect the requirements definition. • Provides risk control facility. • Documentation focuses on the end product not the evolution of the product. • Provides a formal specification embodied in an operating replica.
Weaknesses of prototyping
<ul style="list-style-type: none"> • Is less applicable to the systems already exist than to new, original development. • Bad reputation among conservatives as a ‘quick and dirty’ method. • Suffers from bad documentation. • Sometimes produces a system with poor performance. • Difficult problems to be pushed to the future so that the initial promise of the prototype is not met by subsequent products.

1.3.2 Clean-room Technique

Clean-room technology gives management an engineering approach to release reliable products. The clean-room technique attempts to keep contaminants (software bugs) out of the product. It thus controls cost by detecting bugs as early as possible, when they are less costly to remove. Formal notations are preferred over using natural languages like English to produce specifications on which all software design and requirements validation is based. Developing stable specifications early establishes clear accountability (Figure 1.4).

Off-line review techniques are used to develop understanding of the software before it is executed. Software is intended to execute properly the first time. Programmers are not allowed to perform trial- and-error executions, though automation checks syntax, data flow and variable types.

The clean-room technique can be used in conjunction with waterfall, incremental, or spiral models to produce software of arbitrary size and complexity. The technique has been successful as it provides high quality software, still its widespread acceptance has not materialized due to its radical non-intuitive approach. Table 1.4 gives the strengths and weaknesses of the clean-room technique.

1.3.3 Object-Oriented Technique

The object-oriented approach to software development focuses on real-world objects. It is based on the fact that human beings have a fundamental limitation to manage more than seven different objects or concepts at one time. Grady Booch suggests that the principles of software engineering that can help us decompose systems so that we never simultaneously deal with more than seven entities.

Object orientation includes object-oriented analysis (OOA), design (OOD) and programming (OOP), and its popularity is increasing with the increasing complexity of software systems.

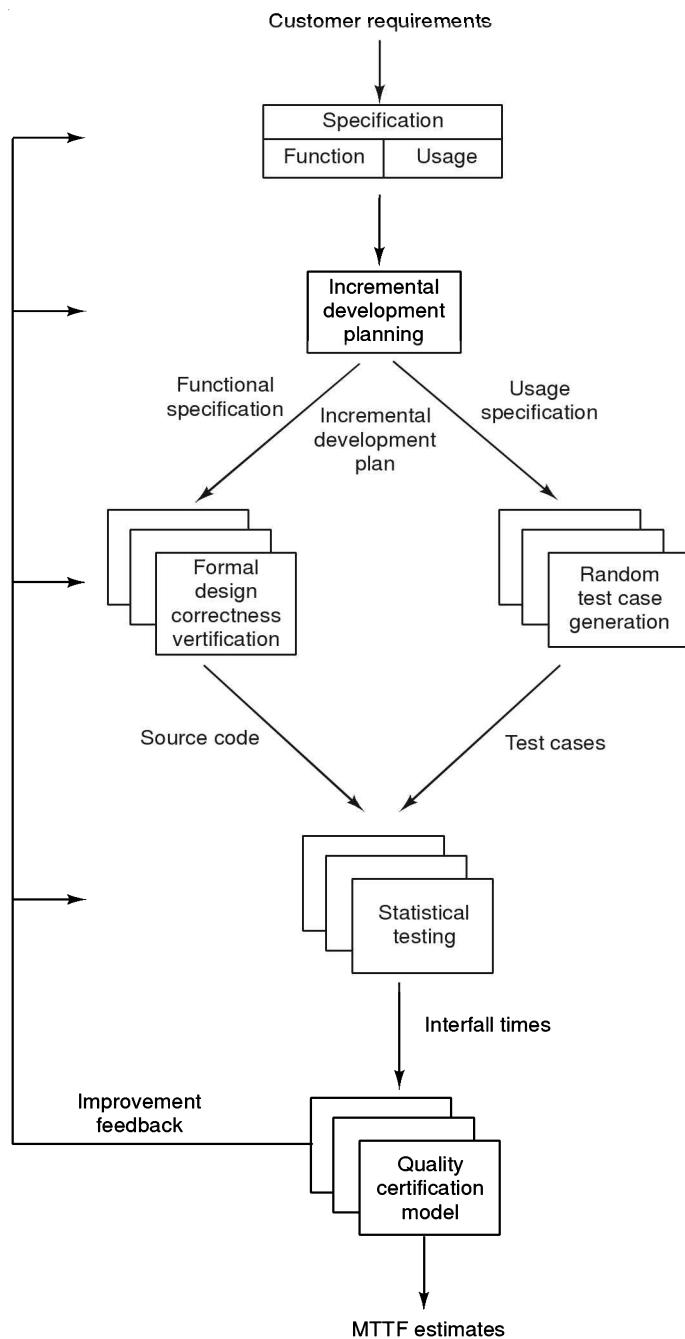


FIGURE 1.4 Clean-room technique.

TABLE 1.4 Strengths and weaknesses of the clean-room technique

Strengths of clean-room technique
<ul style="list-style-type: none"> • Production of reliable software. • Clean-room may be gradually introduced to an organization.
Weaknesses of Clean-room technique
<ul style="list-style-type: none"> • Needs a complete set of requirements. • Disciplined style may stifle creativity.

Object-oriented methodologies are normally used in projects where

1. The system is data-oriented and functional complexity is of lesser concern.
2. Good object-oriented implementation technology is available, and the organization provides adequate tools for its practitioners to effectively use object-oriented techniques.
3. The organization is sophisticated enough to successfully change its development methods.
4. The systems and applications developed by the organization are the kind that will most effectively use the object-oriented paradigm. Note that object-oriented analysis is not very useful for systems with very limited responsibilities.

Table 1.5 shows the strengths and weaknesses of object-oriented approach.

TABLE 1.5 Strengths and weaknesses of object-oriented approach

Strengths of object-oriented approach
<ul style="list-style-type: none"> • Owners of a problem can participate in finding the solution. • Lowers maintenance costs because the object-oriented analysis encourages a complete solution. • Expresses the user's view of reality.
Weaknesses of object-oriented approach
<ul style="list-style-type: none"> • Can be difficult for those with a structured analysis background. • Can be difficult to reconcile with.

It is important to be aware of how object-oriented analysis and design can offer advantages towards software development in comparison to any other technique used.

The structured systems analysis and design method (SSADM) was the most commonly used systems approach for development of information systems. An understanding of SSADM and a comparative analysis of SSADM and object-oriented approach will help us proceeding further.

1.4 STRUCTURED SYSTEMS ANALYSIS AND DESIGN (SSAD)

The structured systems analysis and design method is a set of standards developed in the early 1980s for analysis of systems and subsequently software application development. The SSADM uses a combination of text and diagrams for analysis and design of a system, from the initial design idea to the actual physical design of the application.

Application development projects using structured system analysis and design methodology are divided into five modules that are further broken down into a hierarchy of stages, steps and tasks. These main five modules are as follows:

1. **Feasibility study:** The concerned business domain is analyzed to determine whether an alternate system can cost effectively support the business requirements. Other feasibility such as technological feasibility or organizational feasibility, etc. is also determined before taking a decision to go in favour of an alternative system.
2. **Requirements analysis:** The existing system is studied and the requirements for an improved system which can be developed are identified and the current business environment is modelled in terms of the processes carried out and the data structures involved.
3. **Requirements specification:** The findings of detailed analysis are converted into detailed functional and non-functional requirements, and new techniques are contemplated to improve on the existing system requirement in a better manner.
4. **Logical system specification:** The required processing and data structures are thought about, technical systems options are produced, and the logical design of the system is finalized.
5. **Physical design:** A physical database design and a set of program specifications are created using the logical system specification and technical system specification.

SSADM uses a combination of three techniques such as:

1. **Logical data modelling:** Logical data modelling is the process of identifying the data requirements of the system and, modelling and documenting those data requirements of the system being designed. The data is separated into entities (things about which a business needs to record information) and relationships (the associations between the entities).
2. **Data flow modelling:** Data flow modelling is the process of identifying, modelling and documenting the movement of data in and around the information system. It examines processes (activities that transform data from one form to another), data stores (the holding areas for data), external entities (what sends data into a system or receives data from a system), and data flows (routes by which data can flow).
3. **Entity behaviour modelling:** The process of identifying, modelling and documenting the events that affect each entity and the sequence in which these events occur.

Each of these three-system models provides a different viewpoint of the same system, and each viewpoint is required to form a complete model of the system being designed. The three techniques are cross-referenced against each other to ensure the completeness and accuracy of the whole application.

1.4.1 Features of SSAD

SSAD supports a top-down functional decomposition approach where the system is viewed as being composed of two components, i.e. functionality and data, each of which is modelled more or less separately.

Functionalities are modelled using a technique called *process modelling*. Data is modelled using a technique called *data modelling*.

Since SSAD models, functionality and data separately, we say that it follows a *function/data methodology*. It is more suitable for data intensive projects.

The analysis stage produces a software requirements specification (SRS) consisting of the following deliverables:

- A *process model* consisting of a set of data flow diagrams
- A set of *process specifications*
- A *data model* consisting of an entity-relationship diagram
- A *data-dictionary*

The design stage in SSAD produces:

Database design: The database design is arrived at by applying data modelling techniques to the ER diagram produced during analysis.

Structure chart: The structure chart shows the overall architecture of the system, in terms of the various components and their calling structure and parameters.

Program specifications: The program specifications contain the algorithms to generate the required outputs from the given inputs as specified in the analysis document.

1.4.2 The Traditional Way of Programming—Structured Programming

In structured programming, the general method was to look at the problem, and then design a collection of *functions* that can carry out the required tasks. If these functions are too large, then the functions are broken down until they are small enough to handle and understand. This process is known as *functional decomposition*.

The data in a functional system was usually held in some kind of database or in memory as *global variables*.

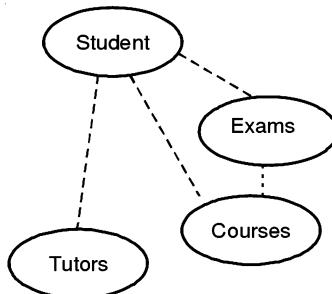
As a simple example, consider a college management system [Figure 1.5(a)]. This system holds the details of every student and faculty in the college. In addition, the system also stores information about the courses available at the college, and tracks which student is following which courses. A possible structured design would be to write the following functions:

- add_student
- appear_for_exam
- issue_marksheet
- expel_student

The dotted lines in Figure 1.5 indicate where one set of data is dependent on another. For example, each student is taught by several tutors.

We would also need a data model to support these functions. We need to hold information about Students, Tutors, Exams and Courses, so we would design a database schema to hold this data.

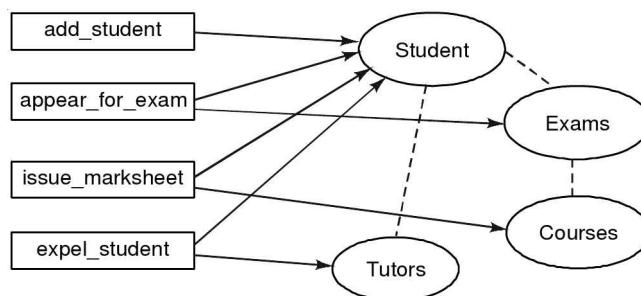
In continuation with the above problem, the functions we defined earlier are clearly going to be dependent on this set of data. For example, the add_student function will need to

**FIGURE 1.5(a)** Simple database schema.

modify the contents of Student. The issue_marksheet function will need to access the Student data to get details of the student, and the function will also need to access the Exam data.

The problem with this approach is that if the problem we are tackling becomes too complex, the system becomes harder and harder to maintain.

Figure 1.5(b) is a sketch of all the functions, together with the data and lines have been drawn where a dependency exists.

**FIGURE 1.5(b)** Functions, data, and the dependencies—The SSADM approach.

If a requirement changes that leads to an alteration in the way which Student data is handled? As an example, imagine our system is running perfectly well, but we realize that storing the Student's date of birth with a two-digit year was a bad idea. The obvious solution is to change the "Date of Birth" field in the Student table, from a two-digit year to a four-digit year.

The serious problem with this change is that we might have caused unexpected side effects to occur. The Exam data, the Course data and the Tutors data all depend on the Student data in some way, so we might have broken some functionality due to our simple change. In addition, we might well have broken the add_student, appear_for_exam, issue_marksheet and expel_student functions.

For example, add_student will certainly not work any more, as it will be expecting a two-digit year for "Date of Birth" rather than four. So we have a large degree of potential secondary problems raised due to slight changes.

1.4.3 SSADM Disadvantage

The disadvantages of SSAD are as follows:

1. Since SSADM is process-oriented, it ignores the non-functional requirements.
2. Since SSADM is non-iterative like the waterfall model, requirements changes would mean restarting the entire process.
3. Except for the logical design and the DFD's, SSADM provides no other tools for communication with users, and therefore, it is more difficult for users to measure progress.
4. In SSADM it is more difficult to decide when to stop functional decomposition and to start building the system.

1.4.4 Why is SSADM not Evacuated?

The main reason why we still follow the structured approach is because of its industry demand. Even if most of the programming languages used in the industry are object oriented, most companies have already established documentation standards based on the structured analysis approach.

1.5 OBJECT-ORIENTED ANALYSIS AND DESIGN (OOAD)

In the last few years, a new methodology has emerged which is a combination of process-oriented and data-oriented methods called an *object-oriented methodology*. This methodology is more suitable for projects with complex business logic.

The object-oriented approach uses a set of diagramming techniques known as unified modelling language. It focuses on three architectural view of a system: functional, static and dynamic. The functional view describes the external behaviour of the system from the perspective of the user. Use-cases and use-case diagrams are used to depict the functional view. The static view is described in terms of attributes, methods, classes, relationship and messages. Class diagrams and object diagrams are used to portray the static view.

The dynamic view is represented by the sequence diagrams, collaboration diagrams and statecharts. All diagrams are refined iteratively until the requirement of the information system is fully understood.

OOAD has several obvious advantages over SSAD as OOAD is more close to the real world, because it is based upon the object. Where SSAD fails after a certain level of complexity, OOAD comes out with flying colours and handles the complexity in a robust and dependable manner. OOAD is supported by various principles like *Maintainability, Extensibility, Reusability, Interoperability and Quality*.

The following are the features of object oriented methodology:

1. Follows bottom-up approach.
2. Programs are divided into objects.
3. It puts emphasis on data rather than procedure.

4. Data is hidden and cannot be accessed by external functions.
5. Objects can communicate with each other through functions.
6. New data and functions can be easily added whenever necessary.

1.5.1 The New Way of Programming—The Object-Oriented Approach

Object-oriented approach tries to simplify the problem by simply combining related data and functions into the same module. Looking at Figure 1.6, it is apparent that the data and functions are related. For example, the `add_student` and `expel_student` functions are clearly very closely related to the `Student` data.

Figure 1.6 shows the full grouping of the related data and functions, in the form of modules:

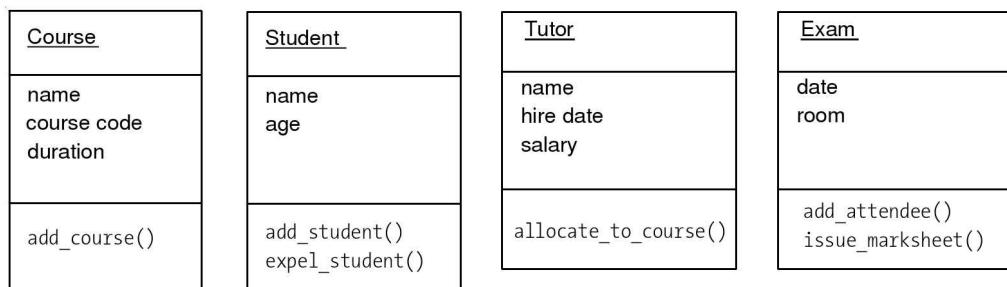


FIGURE 1.6 Related data and functions placed in modules.

1.5.2 Why an OOAD over SSADM?

Object-oriented methods allow us to create a set of objects that works together to produce software that better model the problem domain as in the real world than similar systems produced by traditional techniques SSADM.

The systems are easier to adapt to changing requirements, easier to maintain, more robust, and promote greater design and code reuse. Once objects are defined along with its data and behaviour, it is sure that they will perform their desired functions and your attention as a programmer can be shifted to what they do rather than how they do it.

The object-oriented approach supports abstraction at the object level. The development can proceed at the object level and ignore the rest of the system for as long as necessary, which makes designing, coding, testing, and maintaining the system much simpler.

In the object-oriented approach, moving from one phase to another does not require different styles and methodologies as it uses the same language to talk about analysis, design, programming, and database design. This approach reduces the complexity and makes clearer, more robust system development.

In the object-oriented system, a class carefully separates its interface (specifications of what the class can do) and the implementation of that interface (how the class does it). In a well designed system, the classes will be grouped into subsystems but remain independent, therefore changing one class has no impact on other classes and so the impact is minimized.

Objects are reusable because they are modelled directly from a real-world problem domain. Classes are designed generically with reuse as a constant background goal. As the object orientation adds inheritance, a powerful technique which allows classes to be built from each other and therefore only differences and enhancements between the classes need to be designed and coded, other functionality remains and can be reused without change.

1.5.3 Contribution of Object-Oriented Methodology towards SDLC

Object oriented methodology chains software development cycle. The main objective of object-oriented programming is to develop reusable program routines and data structures.

The steps of developing an information system do not change with object-oriented methodology but differ with the implementation steps.

The object-oriented approach supports the iterative and incremental approach due to which with every phase developer moves through gathering information, analysis, documenting developing and testing.

EXERCISES

1. Write about the traditional development models and techniques for software.
2. Compare the waterfall model, the incremental model and the spiral model.
3. Explain the prototyping technique for software development.
4. What are the features of SSAD?
5. What are the deliverables during structured system analysis and design.
6. What are the disadvantages of SSAD?
7. What is object-oriented analysis and design?
8. What are the feature of OOAD?
9. What are the advantages of OOAD over SSAD?
10. Briefly explain the object-oriented Approach.

C H A P T E R

2

Object Oriented Analysis and Design

2.1 OBJECT-ORIENTED MODELLING

Object-oriented analysis and design (OOAD) focuses on the three most important concepts it encompasses—*objects*, *analysis* and *design*. Object-oriented analysis and design (OOAD) is a software engineering approach that *models* a system as a group of interacting objects. Each object represents some entity of interest in the system being modelled, and is characterized by its class, its state (data elements), and its behaviour. Various models are used in object-oriented modelling to show the static structure, dynamic behaviour, and run-time deployment of these collaborating objects.

The core artifact of the OOAD process is the modelling around objects. There can be various models for different phases, which are explained as follows:

2.1.1 Analysis Model

The analysis model is a model of the existing system, the end user's requirements from the system, and a high-level understanding of a possible solution to those requirements.

2.1.2 Architecture Model

The architecture model is an evolving model of the architectural structure of the solution to the requirements defined in the analysis model. Its primary focus is on the architecture: the components, interfaces, and structure of the solution; the deployment of that structure across nodes; and the trade-offs and decisions that lead up to that structure.

2.1.3 Component (Design) Models

This is a number of models (roughly, one per component) that depict the internal structure of the pieces of the architecture model. Each of the component model focuses on the detailed class structure of its component, and facilitates the design team to precisely specify the attributes, operations, dependencies, and behaviours of its classes.

Depending on the development process, we may have even more models such as a business model, a domain model, and possibly others. The benefit of modelling is that we can

make model changes far earlier in the development cycle in comparison to the code changes. Modelling let us catch costly bugs early, and early detection and correction can save a lot on the cost and schedule of a bug fix.

Object-oriented analysis (OOA) applies object-modelling techniques to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on what the system does, whereas OOD focuses on how the system does it.

2.2 OBJECT-ORIENTED APPROACH

In the first step in object-oriented methodology is concerned with understanding the domain and modelling the real world application for problem statement formulation.

The analysis stage produces a software requirements specification (SRS) consisting of the following deliverables:

- Business process diagrams
- Use-case diagrams
- Class and object diagram.

The *design* stage in OOAD produces:

Design phase: The database design is arrived at by applying data modelling techniques to the class diagram produced during analysis. The design involves:

- Sequence diagram
- Collaboration diagram
- Activity diagram
- State-chart diagram

Implementation phase: Object-oriented methodology provides a clear modular structure for programs where implementation details are hidden. The implementation phase produces:

- Component diagrams
- Deployment diagram

OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones. OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer.

2.2.1 Object Orientation

The term object-oriented means that we organize software as a collection of discrete objects that incorporate both data structure and behaviour. One reason why objects are a powerful programming technique is because they map naturally to real-world objects.

The object-oriented approach supports abstraction at the object level. The development can proceed at the object level and ignore the rest of the system for as long as necessary, which makes designing, coding, testing, and maintaining the system much simpler. In object-oriented approach, moving from one phase to another does not require different styles and methodologies as it uses the same language to talk about analysis, design, programming and database design. This approach reduces the complexity and makes clearer system development.

Suppose, for example, that a company has to deal with orders. These orders would possibly have an ID number and contain information on products. Therefore, you could create Order objects, which would map to these real-world objects, and which would have properties like ID and ProductList. You would probably want to be able to add a product to the order and to submit the order, so you could write AddProduct and SubmitOrder methods. This mapping between objects in the real world and more abstract code objects encourages programmers to think in the problem domain, rather than in computer science terms. These “real-world” objects form just the surface of your system. The complexity of your design lies underneath that surface, in code that reflects business rules, resource allocation, algorithms and other computer science concerns.

Many modern programming languages depend largely or exclusively on the concept of objects: a close syntactic binding of data to the operations that can be performed upon that data. In these object-oriented languages—C++, C#, Java, Eiffel, Smalltalk, Visual Basic .NET, Perl, and many others—programmers create classes, each of which defines the behaviour and structure of a number of similar objects. Then they write code that creates and manipulates objects that are instances of those classes.

A more important benefit of classes and objects is that they form a nice syntactic mechanism for achieving some classic aspects of well-designed code.

2.2.2 Object-Oriented Analysis

Object-oriented analysis (OOA) look examines at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being examined. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt with during object-oriented design (OOD). Analysis is done before the design.

The sources for the analysis can be either a written statement or a formal vision document, and interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which is analyzed separately. The result of object-oriented analysis is a description of what the system is functionally required to do, in the form of a conceptual model. This model is typically represented as a set of use-cases, one or more UML class diagrams, and a number of interaction diagrams.

Object-oriented analysis (OOA), is the process of defining the problem in terms of objects: real-world objects with which the system must interact, and candidate software objects used to explore various solution alternatives. The natural fit of programming objects to real-world objects has a big impact here as one can define all of the real-world objects in terms of their classes, attributes and operations.

2.2.3 Object-Oriented Design

If analysis means defining the problem, then design is the process of defining the solution. It involves defining the ways in which the system satisfies each of the requirements identified during analysis.

Object-oriented design (OOD) is the process of defining the components, interfaces, objects, classes, attributes, and operations that satisfy the systems functional requirements. We typically start with the candidate objects defined during analysis, but add much more rigour to their definitions. Then we add or change objects as needed to refine a solution.

In large systems, design usually occurs at two scales: *architectural design* which involves defining the components from which the system is composed; and *component design* which involves defining the classes and interfaces within a component.

Object-oriented design (OOD) transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional (technological or environmental) constraints, like transaction throughput, response time, run-time platform, development environment, or programming language.

The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of how the system is to be built.

2.3 THE CONSTITUENTS OF OOAD

2.3.1 Objects and Classes

An *object* is the foundation of an object oriented programming. They are the basic run-time entities in an object-oriented system. In this approach the problems are analyzed in terms of objects. When a program is executed, objects interact with each other by sending messages. Different objects can also interact with each other without knowing the details of their data or code. Objects which improve program reliability, simplify software maintenance and management of libraries.

Any object is identified/characterized by its identity that distinguishes it from other objects, and its behaviour.

In object-oriented programming one might say that *class* is a blue print or factory that defines the abstract characteristics of an object including its attribute, fields or properties and its abstract behaviour like its methods, operations or features.

A class is the definition of the behaviour and properties of one or more objects within the system. A class binds the data (attributes) of an object to the behaviour (operations) that it can perform.

A *class* is a collection of objects of similar type. For example, a television you see is one of the millions of televisions that exist in the world. Once a class is defined, any number of objects can be created which belong to that class (Figure 2.1).

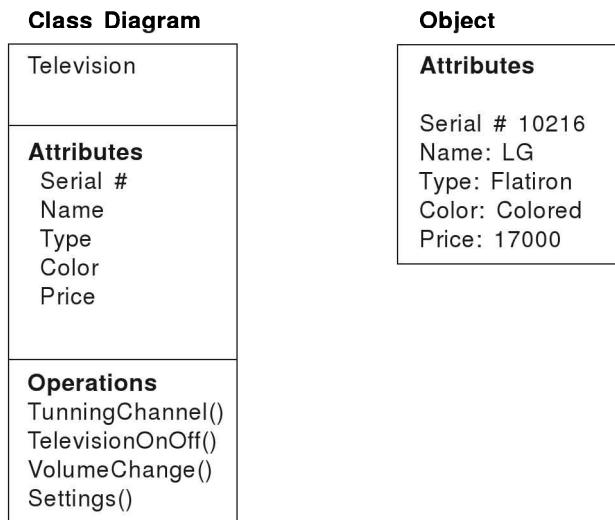


FIGURE 2.1 A sample class and object.

An *attribute* is a data value or state that describes an object and helps you tell one object from another of the same class. It seems that every new OO language author feels the need to distinguish their language by coming up with new terminology. In some OO languages, these data values are called properties or member variables or member data. In UML terminology, the proper term is attributes.

An *operation* is a behaviour or function that an object can perform. Depending on the OO language, these might be called methods or member functions or even messages. The last term, messages, comes from Smalltalk, one of the earliest OO languages, in which all objects communicated by sending messages to each other.

2.3.2 Links and Association

A link is a relationship among instance of classes. In short, the link is used to represent the relationship between the two objects. In contrast an *association* is used to represent the relationship between the two classes.

Let us take an example of a student who selects a course. Here a connection exists between two classes Student and Course. So this is the example of association but if we take one instance of each of the above classes as Tom and MCA, then we can be specific Tom selects an MCA course, which is an example of a link. Also, links and associations cannot be interchangeably used, i.e. we cannot specify that a student selects an MCA course.

2.3.3 Generalization and Specialization

A classification in object-oriented modelling is act of identifying and categorizing similar objects into classes. In real world you can find many objects that are specialized versions of

other more general objects. For example, if you consider the class of cars, you can instantly think of all the models of cars that is a generalization. But if you are deciding on a realistic car to buy you would ask yourself a number of questions like which manufacturer, its average, warranty period, price, etc. in this case you go from the general idea of a car to the specialization of a specific car. The generalization and specialization are represented by “is—a” relationship.

2.3.4 Aggregation and Composition

Both aggregation and composition are special kinds of associations. Aggregation is used to represent ownership or a whole/part relationship, and composition is used to represent an even stronger form of ownership. With composition, we get coincident lifetime of part with the whole. The composite object has sole responsibility for the disposition of its parts in terms of creation and destruction. In implementation terms, the composite is responsible for memory allocation and de-allocation.

2.4 PILLARS OF OBJECT ORIENTED ANALYSIS AND DESIGN

2.4.1 Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes. It is a mental ability that permits people to view real-world problem domains with varying degrees of detail depending upon the current context of the problem.

2.4.2 Encapsulation

Encapsulation is like a black box, where data is sent to a method, processing of specific task takes place using the data, of which you do not know or care about. An output is returned to the user. Encapsulation is a technique in which data are packaged together in a single unit (class) with their corresponding procedures. Data cannot be accessible to the outside world and only those functions which are stored in the class can access it. While tuning a channel the user does not care about what is happening inside the television, what the user cares about is right output.

The goal of encapsulation is to expose only enough of a module or subsystem to allow other modules to make use of it. Object-oriented programming allows you to specify the degree of visibility of elements of your code, so that the client code is restricted in what it can access. Thus, you can syntactically seal off implementation details, leading to more flexibility and maintainability in your system.

2.4.3 Inheritance

It is the process by which objects can acquire the properties of objects of other class. In OOP, inheritance provides reusability, like adding additional features to an existing class without modifying it. This is achieved by deriving a new class from the existing one. The new class will have combined features of both the classes. This concept indicates that one class (the superclass) provides some common or general behaviour inherited by one or more specific classes (the subclasses). The subclasses then provide more or different behaviours beyond that defined in the superclass.

For example, Dog class can have objects like Tommy, Tuffy, etc. Then we can have Cat objects and Horse objects that live on one property. Each class has unique behaviours: Dogs must be walked, Cats use the litter box and Horses drop manure that must be scooped up and thrown in the manure pile. Yet all classes have some common behaviour: they must be fed, and they must have vet visits. So we can define a superclass, Pet, and have many subclasses such as Dog, Cat and Horse, derive their shared behaviour from the Pet class.

In UML (section 2.5), this concept is known under the slightly different term of generalization, in which a superclass provides the generalized behaviour of the subclasses. It is really the same concept, but just looking at it the other way up.

2.4.4 Polymorphism

Polymorphism means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends on the data types used in the operation. Polymorphism is extensively used in implementing inheritance.

A television has common properties like size, price, colour, type, etc. which are specific to all make/model of a television. The additional features like CD changer slot, home theatre attachment, dolby speakers, etc. are specific with the specific make/model. In addition, certain behaviours like television-On-Off, tuning channel, settings, etc. are basic functionalities of all televisions but some of the methods like CD changer, playing games, etc. are specific with certain make/models. These certain makes/models inherits the properties and behaviour of the basic model of televisions and ad-on its new properties and behaviour.

2.4.5 Coupling

Coupling refers to the ways, in which and degrees, to which one part of the system relies on the details of another part. The tighter the coupling, the more changes in one part of the system will ripple throughout the system. With loose coupling, the interfaces between subsystems are well defined and restricted. What lies beyond those interfaces can change without any changes needed in the client subsystems. Object-oriented programming supports loose coupling by allowing us to define and publish a class's methods without publishing how those methods are carried out. This principle goes even further in OO languages that support interfaces (described later in section 2.4.8).

2.4.6 Cohesion

Cohesion refers to the degree in which elements within a subsystem form a single, unified concept, with no excess elements. Where there is high cohesion, there is easier comprehension and thus more reliable code. Object-oriented programming supports strong cohesion by allowing one to design classes in which the data and the functions that operate on them are tightly bound together.

2.4.7 Components

A component is a collection of related classes that together provide a larger set of services. Components in a system might include applications, libraries, ActiveX controls, JavaBeans, daemons, and services. In the .NET environment, most of the projects will require component development.

2.4.8 Interfaces

An interface is a definition of a set of services provided by a component or by a class. This allows further encapsulation: the author of a component can publish just the interfaces to the component, completely hiding any implementation details. Each of these concepts will be explored in more detail as we discuss the UML diagrams that represent them.

2.5 THE LANGUAGE OF OOAD—UNIFIED MODELLING LANGUAGE AND BPMN

The unified modelling language (UML) is a graphical language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things like business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

It provides a comprehensive notation for communicating the requirements, behaviour, architecture, and realization of an object-oriented design. UML provides us a way to create and document a model of a system.

The purpose of UML, is communication; to be specific, it is to provide a comprehensive notation for communicating the requirements, architecture, implementation, deployment, and states of a system.

UML communicates these aspects from the particular perspective of object-orientation (OO), in which everything is described in terms of objects: the actions that objects take, the relationships between the objects, the deployment of the objects, and the way the states of the objects change in response to external events.

In the case of OOAD with UML, the models consist primarily of diagrams: static diagrams that depict the structure of the system, and dynamic diagrams that depict the behavior of the system.

With the dynamic diagrams, we trace through the behaviour and analyze how various scenarios play out. With the static diagrams, we ensure that each component or class has access to the interfaces and information that it needs to carry out its responsibilities. It is very easy to make changes in these models: adding or moving or deleting a line takes moments; and reviewing the change in a diagram takes minutes. A few important concepts are detailed as follows before we get into the details of UML diagrams.

2.5.1 UML Diagrams

UML consists of nine different diagram types, each focused on a different way to analyze and define the system. These diagrams are summarized briefly here:

- (i) **Class diagrams** show class definition and relations. A class diagram depicts the classes and interfaces within the system, as well as the relations between them. It is useful for defining the internal, object-oriented structure of the software.
- (ii) **Object diagrams** show a set of objects and their relationship. Object diagrams are helpful for documenting test cases and discussing examples. One class diagram corresponds to an infinite set of object diagrams.
- (iii) **Use-case diagrams** show the externally visible behavior of the system. A use-case diagram depicts actions by people and systems outside your system, along with what the system does in response. It is useful for depicting the functional requirements of your system.
- (iv) **Sequence diagrams** show object interactions over time. A sequence diagram depicts the detailed behaviour over time within one path or scenario of a single functional requirement. It is useful for understanding the flow of messages between elements of the system.
- (v) **Collaboration diagrams** show object interactions with emphasis on relations between objects.
- (vi) **Statechart diagrams** show state changes in response to events. A statechart diagram depicts how the state of the system changes in response to internal and external events. It is useful for ensuring that each event is handled properly no matter what state the system may be in.
- (vii) **Activity diagrams** show an elaboration of the behaviour of the system. An activity diagram depicts the detailed behaviour inside a single functional requirement, including a primary scenario and a number of alternative scenarios. It is useful for thorough understanding of a given functionality.
- (viii) **Component diagrams** show the architecture of the system. A component diagram depicts the deployable units of a system—executables, components, data stores, among others—and the interfaces through which they interact. It is useful for exploring the architecture of the system.
- (ix) **Deployment diagrams** show physical architecture of the system. A deployment diagram depicts how the deployable units of the system—applications, components, data stores, etc.—are assigned to various nodes, as well as how the nodes communicate with each other and with devices. It is useful both as a map of the system and as a means for studying the load across the system.

Making good UML diagrams takes some skill and some practice; but reading well-crafted diagrams is a very different matter. Most of us already know how to read UML diagrams, and we do not even realize that. UML diagrams are straightforward and one need not be an expert to understand the information contained in the diagrams. It is because, UML is all about communication.

2.5.2 Introduction to BPMN

The Business Process Management Initiative (BPMI) has developed a standard Business Process Modelling Notation (BPMN). The BPMN 1.0 specification was released to the public in May, 2004. This specification represents more than two years of effort by the BPMI Notation Working Group. The primary goal of the BPMN effort was to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.

BPMN is an agreement between multiple modelling tools vendors, who had their own notations, to use a single notation for the benefit of end-user's understanding and training. BPMN defines a Business Process Diagram (BPD), which is based on a flowcharting technique tailored for creating graphical models of business process operations. A business process model, then, is a network of graphical objects, which are activities (i.e. work) and the flow controls that define their order of performance.

EXERCISES

1. What is object-oriented analysis?
2. What is object-oriented design?
3. Briefly explain the concept of an object and a class.
4. Compare link and association.
5. Differentiate between generalization and specialization.
6. Explain aggregation and composition with an example.
7. What is meant by inheritance? Explain with an example.
8. What is encapsulation? Briefly explain.
9. Explain the concept of interface and component.
10. What is the purpose of UML and BPMN?

CHAPTER Business Process Diagram

3 and Use Case Diagram

3.1 INTRODUCTION TO BUSINESS PROCESS DIAGRAM

Process modelling is an important part of business design, business redesign, or business reengineering. It is merely a tool that provides a means of communicating complex business functions in a form more easily understandable by people. Modelling provides for the formalization of processes which in turn allows the business to operate in a standardized manner. The effective design of business processes allows individuals to work together more efficiently. Modelling of lower level (elementary) processes provides consensus on business rules.

Process models are an aid to the understanding of the nature in which processes dynamically work with data. From a design perspective, modelling provides explicit guidelines for modularity, reusability, flexibility and integrity.

The business process diagram (BPD) is a standardized graphical flowchart for defining business processes in a workflow. The primary goal of the business process diagram is to provide a standard notation that is readily understandable by all business stakeholders. These business stakeholders include the business analysts who create and refine the processes, the technical developers responsible for implementing the processes, and the business managers who monitor and manage the processes.

One of the drivers for the development of the business process diagram is to create a simple mechanism for creating business process models, while at the same time being able to handle the complexity inherent to business processes.

Taking an example of a typical business process diagram shows a series of inputs and converts them into a specific output, such as *Create Sales Forecast*, which rolls up individual sales representative forecasts into a company-wide forecast.

3.2 SCOPE OF BUSINESS PROCESS DIAGRAM

The objective of the business process diagram is to support business process management for both technical users and business users by providing a notation that is intuitive to business users yet able to represent complex process semantics. Consequently, BPD is intended to serve as a common language to bridge the communication gap that frequently occurs between business process design and implementation. BPD limits its scope to support only the concepts

of modelling that are applicable to business processes. Other types of modelling done by organizations for non-business purposes is out of scope for BPD. For example, modelling of organizational structures, functional breakdowns, data models, etc., are out of scope of BPD.

In addition, while BPMN will show the flow of data (messages), and the association of data artifacts to activities, it is not a data flow diagram.

3.3 ELEMENTS OF BUSINESS PROCESS DIAGRAM

The BPD modelling uses simple diagrams with a small set of graphical elements. It should make it easy for business users as well as developers to understand the flow and the process. The four basic categories of elements are as follows:

1. Flow objects
2. Connecting objects
3. Swimlanes
4. Artifacts

These four categories of elements give us the opportunity to make a simple business process diagram (BPD). It is also allowed in BPD to make your own type of a flow object or an artifact to make the diagram more understandable.

3.3.1 Flow Objects

Flow objects are the main describing elements within BPMN, and consist of three core elements: events, activities, and gateways.

Event

An *event* is something that happens during the course of a business process. These events affect the flow of the process and usually have a cause (trigger) or an impact (result).

Events are circles with open centres to allow internal markers to differentiate different triggers or results. There are three types of events, based on when they affect the flow: Start, intermediate and end as shown in Figure 3.1.

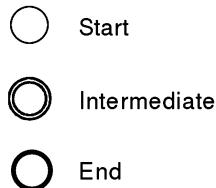


FIGURE 3.1 Events in a BPD.

Activity/Process

An *activity* shows us the kind of work which must be done (Figure 3.2). An Activity can be atomic or non-atomic (compound). The types of activities are: *task* and *sub-process*. The sub-process is distinguished by a small plus sign in the bottom center of the shape.



Figure 3.2 Activity/Process in a BPD.

Gateway

A *gateway* is used to control the divergence and convergence of sequence flow (Figure 3.3). Thus, it will determine traditional decisions, as well as the forking, merging, and joining of paths. Internal markers will indicate the type of behaviour control.

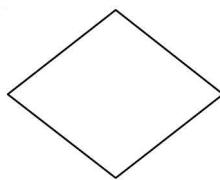


FIGURE 3.3 Gateway in a BPD.

Gateways

Gateways are modelling elements that are used to control how sequence flows interact as they converge and diverge within a process.

Gateways are inserted in the business process model where there is decision making or converging and diverging. Based on behaviour of flows, gateways are classified as Exclusive gateway, Inclusive gateway, Complex gateways and Parallel gateway.

Exclusive gateway

- Exclusive gateways (Decisions) are locations within a business process where the sequence flow can take two or more alternative paths. This is basically the “fork in the road” for a process.
- Only one of the possible outgoing paths can be taken when the process is performed.
- The decision may be based on data or event.
- They are also used to merge sequence flow.

Inclusive gateway

- Inclusive gateways are decisions where there is more than one possible outcome.
- At least one sequence flow must be selected.
- They are usually followed by a corresponding merging inclusive gateway.

Complex gateways

- A complex gateway is used when the other three gateway types are not appropriate for the process.

- In order to continue processing, at least one sequence flow must be selected.
- Complex behaviour can be defined for both the merging and splitting behaviour.

Parallel gateways

- Parallel gateways are places in the process where multiple parallel paths are defined.
- The gateway is also used to synchronize parallel paths.

In the above example, there is no decision making, converging or diverging of sequence flow appear during any of the lane or pool activities. Hence, no gateway is shown.

3.3.2 Connecting Objects

The *flow objects* are connected together in a diagram to create the basic skeletal structure of a business process with connecting objects. There are three *connecting objects* that provide this function. These connectors are as follows:

Sequence flow

A *sequence* is used to show the order in which the activities will be performed in a process (Figure 3.4).



FIGURE 3.4 Sequence flow in a BPD.

Message flow

A *message flow* is used to show the flow of messages between two separate process participants (business entities or business roles) that send and receive them. In BPMN, two separate pools in the diagram will represent the two participants (Figure 3.5).



FIGURE 3.5 Message flow in a BPD.

Association

An *association* is used to associate data, text and other artifacts with flow objects. Associations are used to show the inputs and outputs of activities Figure (3.6).

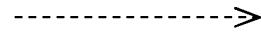


FIGURE 3.6 Association flow in a BPD.

3.3.3 Swimlanes

A *swimlane* is a visual mechanism of organizing different activities into categories of the same functionality. There are two different swimlanes, and they are described as follows:

Pool

A *pool* represents a participant in a process. It also acts as a graphical container for partitioning a set of activities from other pools usually in the context of B2B situations. A pool contains many flow objects, connecting objects and artifacts (Figures 3.7 and 3.8).

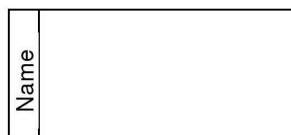


FIGURE 3.7 Pool in a BPD.

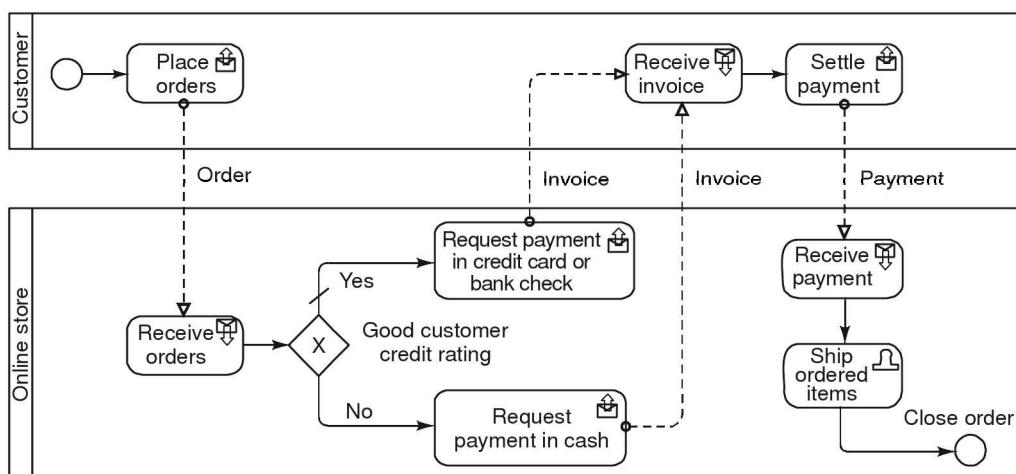


FIGURE 3.8 Different pools—Online store shopping.

Pools are used when the diagram involves two separate business entities or participants and are physically separated in the diagram. The activities within separate pools are considered self-contained processes.

Thus, the sequence flow should not cross the boundary of a pool. The message flow is used to show the communication between two participants, and thus, always connects between two pools. For example, in Figure 3.9 there are two different pools shown in the case of online store shopping.

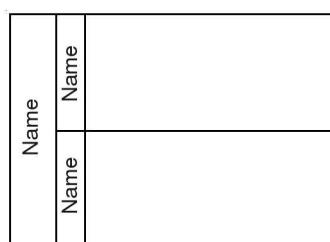


FIGURE 3.9(a) Lane in a BPD.

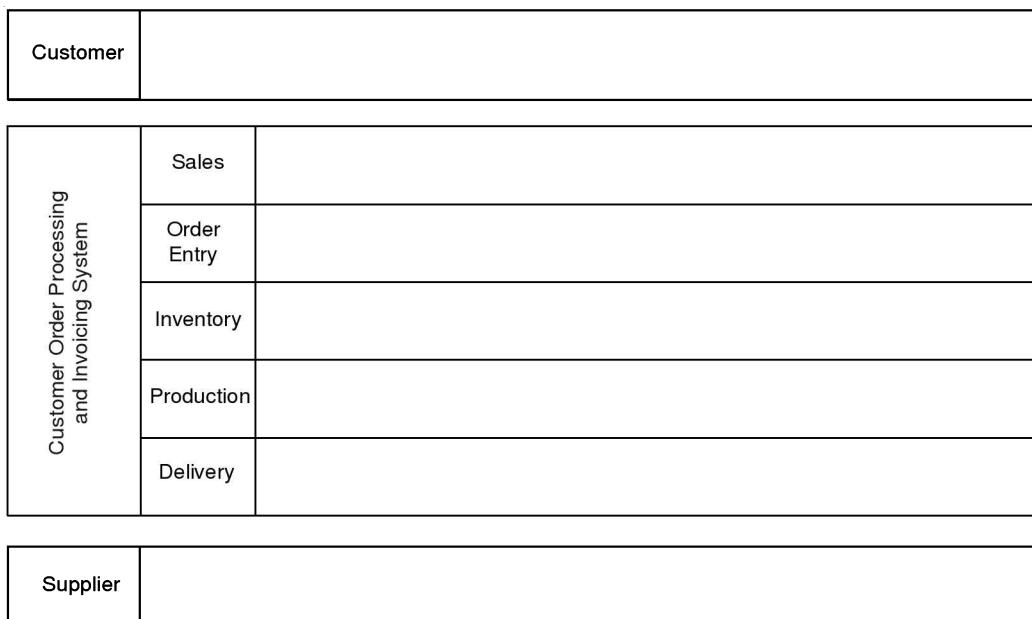


FIGURE 3.9(b) Different lanes in a pool.

Lane

A *lane* is a sub-partition within a pool and will extend the entire length of the pool, either vertically or horizontally. Lanes are used to organize and categorize activities.

The complete business diagram shown later in Figure 3.12.

3.3.4 Artifacts

Artifacts allow developers to bring some more information into the diagram. It allows modellers and modelling tools some flexibility in extending the basic notation and in providing the ability to additional context appropriate to a specific modelling situation. In this way the diagram becomes more readable.

There are three pre-defined artifacts and they are *data objects*, *group* and *annotations*. All these artifacts display some additional information in the diagram such as input to activities, output from activities (data objects), grouping of related process objects (groups), textual comments about events, activities and gateways (annotations).

Data objects

Data objects are a mechanism to show how data is required or produced by activities. They are connected to activities through associations [Figure 3.10(a)].

Group

A *group* is represented by a rounded corner rectangle drawn with a dashed line. The grouping can be used for documentation or analysis purposes, but does not affect the sequence flow [Figure 3.10(b)].

Annotation

Annotations are a mechanism for a modeller to provide additional text information for the reader of a diagram [Figure 3.10(c)].

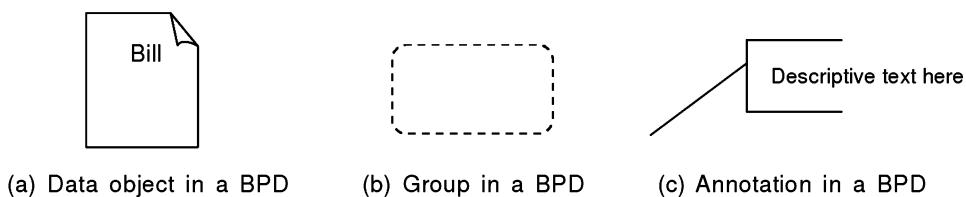


FIGURE 3.10 Different artifacts in a BPD.

3.4 GUIDELINES FOR DESIGN OF BUSINESS PROCESS DIAGRAM

A process is a series of repetitive actions with a definable start and end, and well-defined in terms of input, transformation and output of values.

An elementary process is the smallest unit of activity of the business and which, when complete, leaves the data in a consistent state.

3.4.1 Finding out Elementary (Atomic) Processes and Complex (Non-atomic) Processes

Processes which are identified during business modelling may be complex, non-atomic which can be broken down into a finer level of details. These complex high level processes may require analysis through a number of levels of diagrams in order to decompose into functional components which can be easily implemented. The processes which are having sufficient scope to analyze it further are clearly non-atomic processes. The process which cannot be further decomposed into sub-processes is atomic or elementary process.

3.4.2 Example Explaining Elementary and Complex Processes

In the example shown in Figure 3.11, all the level 2 processes would almost certainly be ‘elementary’, as they have simple and specific names. This level 2 diagram has been developed from the level 1 process ‘Order Delivery Process’ as shown in Figure 3.11(a). Elementary processes can occur at many levels within a business model as shown in Figure 3.12(b).



FIGURE 3.11(a) Complex process.



FIGURE 3.11(b) Elementary processes.

3.4.3 Represent Processes along the Pools and Lanes

All the identified processes must be represented along the appropriate pools and lanes connected with a sequence flow across the lanes within the pool and/or message flow across the pools. The start and end events must be specified in the right lane in the pool. If there is any decision point, specify it with a right gate. Each process, event, lane and pool must have a simple and meaningful name. Also check that each process has at least one incoming data flow and at least one outgoing data flow. Make sure that all data objects have logical names, indicating their content.

3.4.4 Check for Consistency within the Diagram

All data flows ending in a process on a high level diagram must end in at least one of the processes on the associated lower level diagram. Otherwise there may be an inconsistency which should be rectified rechecking both levels of diagrams.

3.4.5 Analyze the Level of Interaction

Remember, there are some basic ground rules to draw a business process diagram, but the scope depends on the level of interaction between two entities. The closer analysis is required if there are a large number of flows between any two items on the diagram. If there are no flows between two areas of a diagram, this may indicate that two separate diagrams exist. A simple business process diagram is shown in Figure 3.12.

3.5 CASE: GET-WELL-SOON HOSPITAL PVT. LTD.

Get-well-soon Hospital Pvt. Ltd. is a medical care unit to attend patients. These patients are given a patient identification number to avail facilities at the hospital. The hospital is attached with several pathological laboratories associated where different medical tests are performed. The patients are given expert medical consultation by the doctors who are associated with the hospital on different contracts, for example, on a regular basis or contract-on-hourly basis or as adjunct consultants and are paid accordingly. The doctors also suggest medical-tests for the

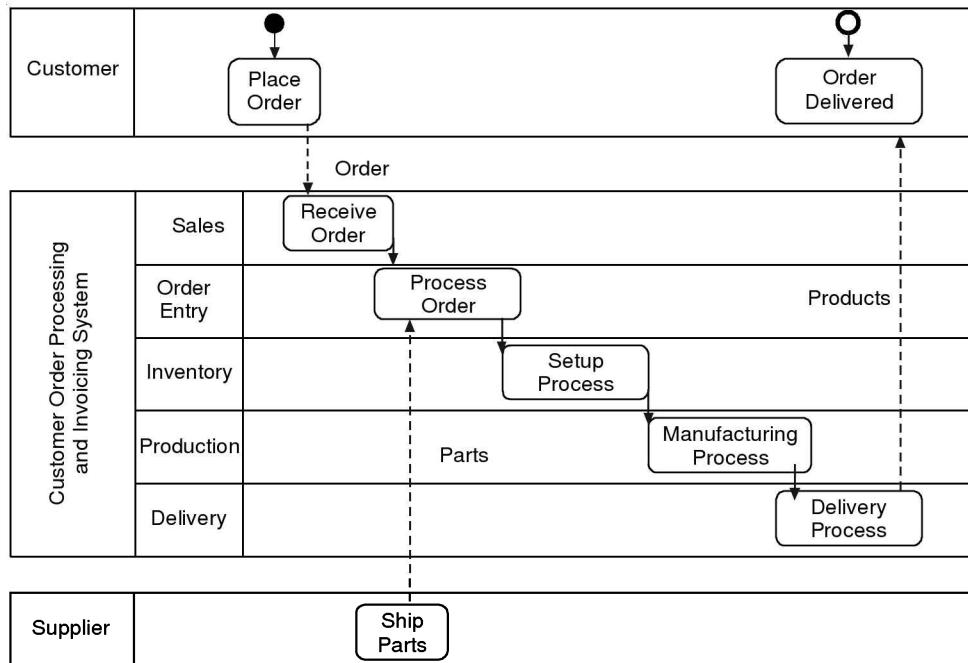


FIGURE 3.12 A simple business process diagram.

patients. The patients then get the medical tests done from any of the laboratory where the facility is available. There are several wards in the hospital of different categories, namely, general ward, semi-private ward, private ward, deluxe ward, operation-theater, intensive care unit, etc. The type of ward also determines the charges, the facilities and the number of beds in that ward.

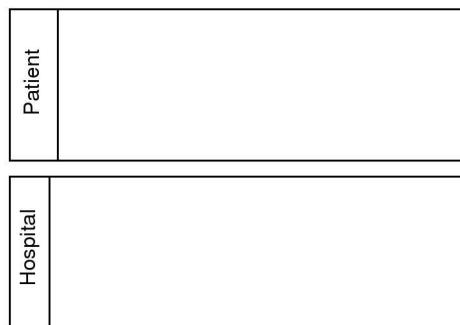
Miscellaneous services for the patients are also available such as attendant, special nurse care, specific diet, etc. which are charged separately. The patients are charged for their ward bill, for expert consultation, for laboratory tests and miscellaneous services. Below we try to identify the swimlanes, pools flow objects, artifacts and finally the BPD.

3.5.1 Swimlanes

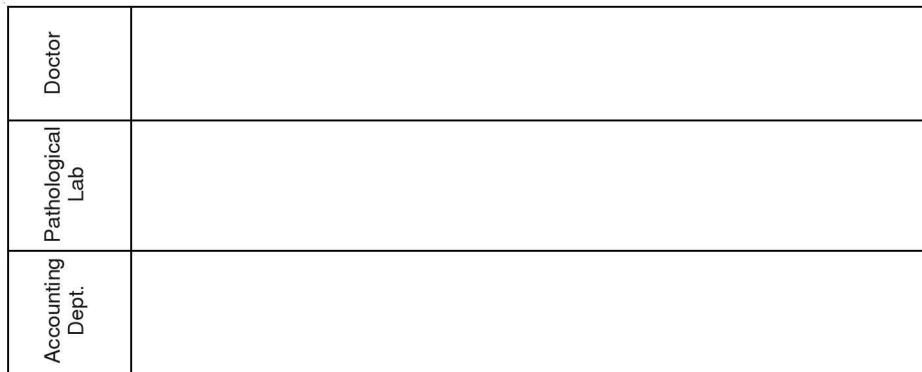
The pool represents participants in an interactive business process diagram. A participant may be a business role (e.g. buyer or seller) or may a business entity (e.g. IBM or Microsoft). Identify if there is only one or more than one participants performing the activities. All the participants will be the pools.

Here, *Hospital* and *Patient* are the participants involved in all the activities. So, pools in this system are Hospital, Patient as shown in Figure 3.13.

Lanes represent subpartitions for the objects within a pool. They often represent organization roles (e.g. Manager, Faculty). Now within each pool, find out if there is one or more organizational

**FIGURE 3.13** Pools in Get-Well-Soon Hospital.

roles within a pool. Those will be the *lanes*. Here, the patient is the pool which will not have any lane within it. Doctor, Pathological laboratory, Accounting department, etc. perform the organizational roles within the Hospital pool. So, the lanes in the Hospital pool are Doctor, Pathological laboratory and Accounting department, as shown in Figure 3.14.

**FIGURE 3.14** Lanes in Get-Well-Soon Hospital.

3.5.2 Flow Objects

Activities

An activity is work that is performed within a business process. An activity can be *atomic* or *non-atomic* (compound).

The types of activities that are a part of a process model are: *sub-process* and *task*. A *task* is an atomic activity that is included within a process. A task is used when the work in the process is not broken down to a finer level. There are specialized types of tasks for sending and receiving, etc. Markers or icons can be added to tasks to help identify the type of task. Tasks performed in the patient pool are illustrated in Figure 3.15.

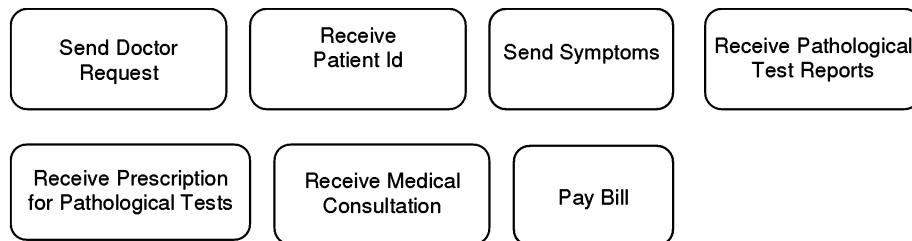


FIGURE 3.15 Tasks in patient pool.

Tasks performed within the hospital pool are shown in Figure 3.16.

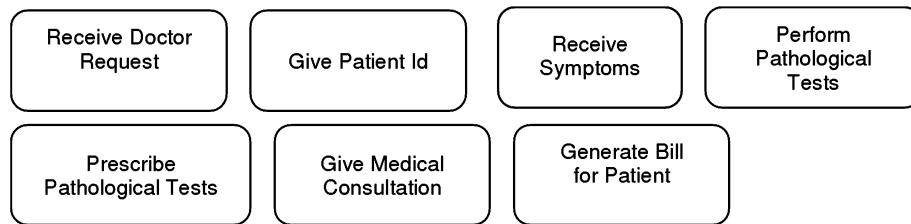


FIGURE 3.16 Tasks in hospital pool.

There are no sub-processes identified in this example as every activity identified as above is an atomic process.

Events

An event is something that happens during the course of a business process. There are only start and end events identified in this example. Only one intermediate event (Message) for sending and receiving messages occur in this example (Figure 3.17).

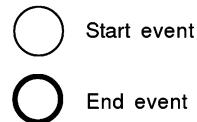


FIGURE 3.17 Events in Get-Well-Soon hospital.

3.5.3 Artifacts

All these artifacts display some additional information in the diagram such as input to activities, output from activities (data objects), and grouping of related process objects (groups), textual comments about events, activities and gateways (annotations). In this example, the only artifacts are data objects which are shown in Figure 3.18.



FIGURE 3.18 Data objects (Artifacts) in Get-Well-Soon hospital.

3.5.4 Business Process Diagram

Now a complete BPD with all the objects is drawn as shown in Figure 3.19.

Important points

1. There can be sequence flow within activities, events and gateways only within the same pool to show the sequence.
2. Sequence flow can cross the lanes, but not the pools.
3. To show the interaction across pools, use message flow and **NOT** sequence flow.
4. Within each pool, activities get initiated with *start* event and ended with *end* event.
5. If some activities execute on message trigger and timer trigger and rule trigger, it can be shown by a specific marker inside the event.

3.6 USE-CASE DIAGRAM

Use-cases diagrams provide a simple, fast means to decide and describe the purpose of a project. They are successfully employed by many software engineers as a way to capture the high-level objectives of an application during the initial phase of development.

The use-case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed *actors* and the processes are called *use-cases*. The use-case diagram shows which actors interact with each use case. Use-case diagrams are often used to:

1. Provide an overview of all or part of the usage requirements for a system or organization in the form of an essential model or a business model.
2. Communicate the scope of a development project.
3. Model your analysis of your usage requirements in the form of a system use case model.

3.7 SCOPE OF USE-CASE DIAGRAMS

A use-case diagram captures the functional requirements of a system. The main purpose of the use-case diagram is to present a graphical view of the functionality provided by a system in terms of actors, their goals (represented as use-cases) and any dependencies between those use-cases.

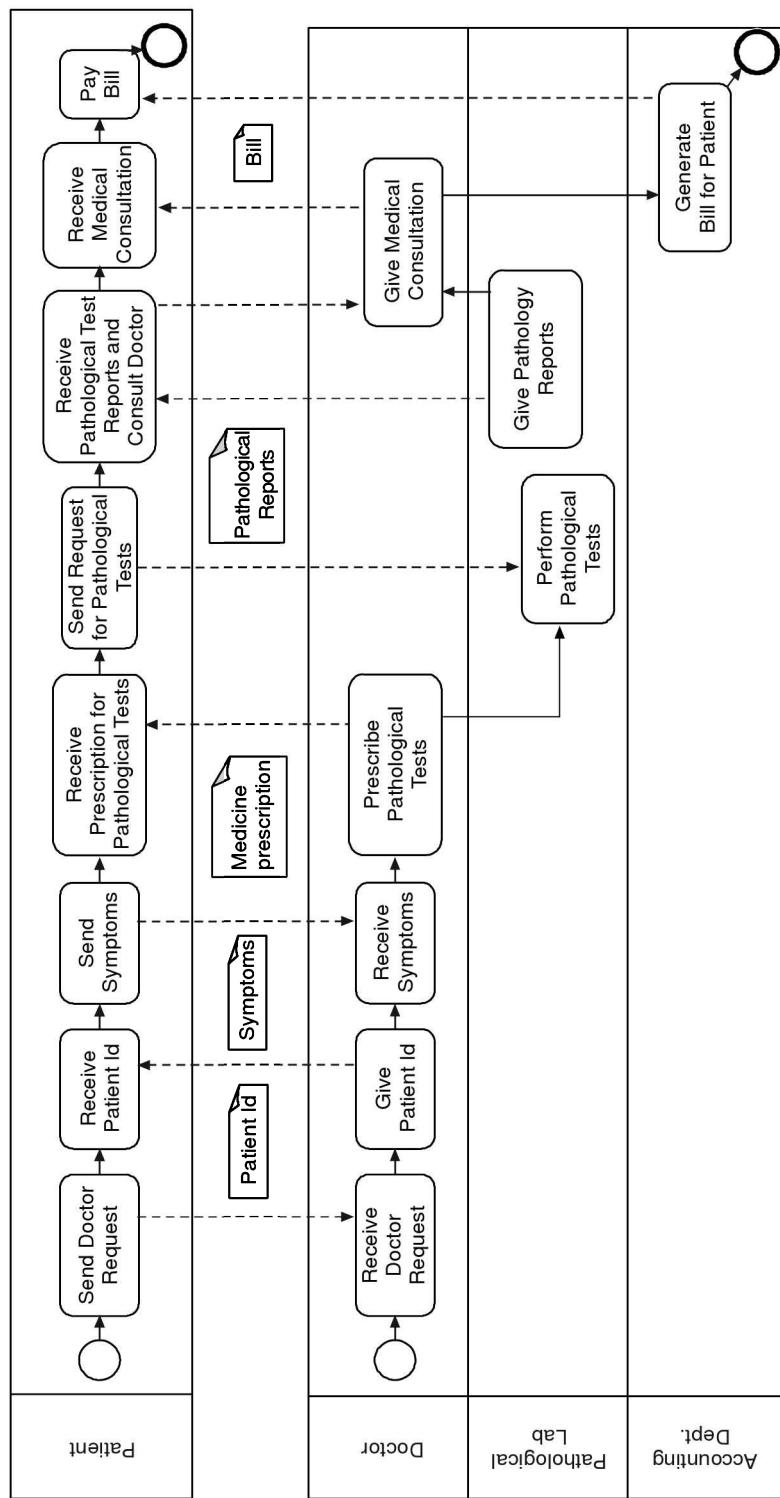


FIGURE 3.19 Business process diagram for Get-Well-Soon hospital.

Use-case diagrams define the requirements of the system being modelled and hence are used to write test scripts for the modelled system. Due to the simplicity of use-case diagrams, they are a great storyboard tool for user meetings.

The use-case model can be a powerful tool for controlling scope throughout a project's lifecycle. Because a simplified use-case model can be understood by all project participants, it can also serve as a framework for ongoing collaboration as well as a visual map of all agreed-upon functionality. It can, therefore, be a precious reference during later negotiations that might affect the project's scope.

Normally, domain experts and business analysts should be involved in writing use-cases for a given system. Use-cases are created when the requirements of a system need to be captured. Because, at this point no design or development activities are involved, technical experts should not be a part of the team responsible for creating use-cases. Their expertise comes in use later in the software lifecycle.

3.8 BENEFITS OF A USE-CASE DIAGRAMS

The crucial benefit of use-cases is the way they encourage a directed method of considering project requirements. From the very beginning, we are designing a product by concentrating upon the needs and wants of those who will use it.

As with any foundation, the better our understanding of the use-cases, the easier, more focused, and more appropriate will be the work that follows. Use-cases are the context that allows us to easily picture where, within a project's life, a particular element will fit, thus promoting clearer decision-making throughout design and development. Other benefits of use-cases are listed as follows:

1. If we simply consider the roles played by the actors and their goals, the use-case model can very rapidly emerge.
2. Use-case diagrams can distill a complex project into a more easily comprehensible picture.
3. A well-constructed use-case model can be understood by all the stakeholders in a project: developers, managers and clients. It is a powerful aid to collaborative development.
4. Use-cases ensure that scope is under control from the outset. The identification of use-cases and their dependencies make it easy to distinguish between core goals that must be satisfied and subsidiary enhancements that may be postponed. Scoping in this manner allows for better planning and prioritization.
5. It is an implementation-neutral picture of a project. No assumptions about tools and technologies are made, nor should they be.
6. It is transportable. No special tools are required—sticky notes, a whiteboard, pencil and paper, or your favourite graphics application can all be used to document your vision.
7. Use-case driven development is a mindset, as much as it is a technique. By emphasizing the actors and what they wish to achieve, project teams can advance with greater confidence and clarity. A solid early foundation of understanding amongst all concerned allows more rapid decision-making later on and encourages a continual focus on the project's true purpose.

3.9 ELEMENTS OF USE-CASE DIAGRAM

To define a project's use cases, we need to consider two concepts, i.e. the actors and their goals, and how they relate. Use-case diagrams make it easier to think about the relationships, or dependencies, between use-cases and actors. Actors are everyone and everything that will use the system. Goals are what one, some, or all of the actors want to achieve. To be complete, every use-case must describe a specific goal and the actors that will perform tasks to achieve that goal.

3.9.1 Actors

An actor [Figure 3.20(a)] represents a coherent set of roles that users of use-cases play when interacting with these use-cases. Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system.

Identify actors in the systems by asking the following questions:

1. Who uses the main functionality of the system?
2. Which hardware devices does the system need to handle?
3. Which other systems does the system need to interact with?
4. What *nouns/subjects* are used to describe the system? For example, *the reservation clerk makes a booking using the system.*

A user must login in order to check his account status.

They have the following characteristics:

1. Actors are external to a system.
2. Actors interact with the system. Actors may use the functionality provided by the system, including application functionality and maintenance functionality. Actors may provide functionality to the system. Actors may receive information provided by the system. Actors may provide information to the system.
3. Actor classes have actors instances or objects that represent specific actors.

3.9.2 Use-Cases

A use-case [Figure 3.20(b)] is a description of a set of sequences of actions, including variants, that a system performs to produce an observable result of value to an actor.

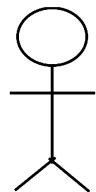


FIGURE 3.20(a) Actor in a use-case.



FIGURE 3.20(b) Use-case.

A use-case describes what a system (or a subsystem, class or interface) does but it does not specify how it does it.

Identify use-cases in the systems by asking the following questions:

1. What functions do the system perform?
2. What functions do the actors require?
3. What input/output do the system need?
4. What *verbs* are used to describe the system? For example, *The reservation clerk makes a booking using the system.*

The airport manager can make an entry for a new flight.

3.9.3 Relationships between Actor and Use Case

The participation of an actor in a use-case, i.e. instances of the actor and instances of the use-case communicate with each other. This is the only relationship between actors and use-cases. In Figure 3.21, a relationship between an actor (customer) and a use-case (withdrawal of cash) is shown.

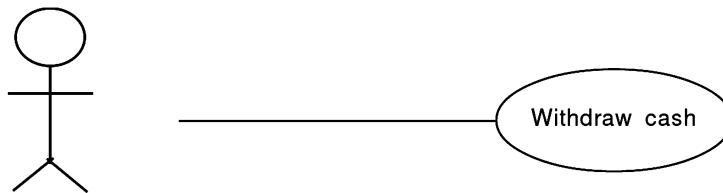


FIGURE 3.21 Customer withdraws cash.

3.9.4 Relationships between Use-Cases

Uses/Includes: An *include* relationship between two use-cases means that the sequence of behaviours described in the included (or *sub*) use-case is included in the sequence of the base (including) use-case. Including a use-case is thus analogous to the notion of calling a subroutine.

Use-case A *uses* use-case B when use-case B is a behaviour/functionality that is required by use-case A. That behaviour has been factored out into a separate use-case because it is required across several use-cases [Figure 3.22(a)].

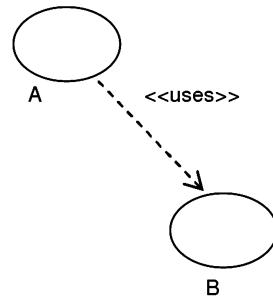


FIGURE 3.22(a) Relationship between use-cases: <<uses>>/<<includes>>.

Common behaviour in several use-cases can be factored out into a single use-case that is used by the other use-cases. The relationship arrowhead is pointing towards the included use-case from the base use-case [Figure 3.22(a)].

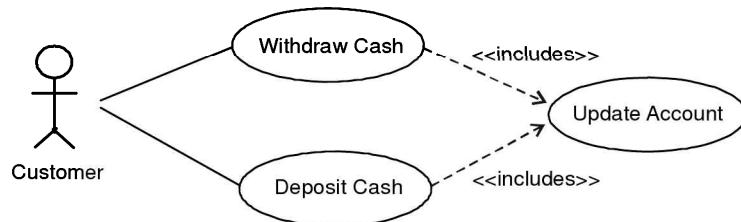


FIGURE 3.22(b) Example: <<uses>>/<<includes>>.

Extends: The *extends* relationship provides a way of capturing a variant to a use-case. Extensions are not true use-cases but changes to steps in an existing use-case. Typically extensions are used to specify the changes in steps that occur in order to accommodate an assumption that is false. The extends relationship includes the condition that must be satisfied if the extension is to take place, and references to the extension points which define the locations in the base use case where the additions are to be made.

Use-case B **extends** use-case A when use-case B describes the behaviour of use-case A under a particular condition. An extending use-case is used to describe variations in the normal flow of events described by a general use-case [Figure 3.23(a)]. The relationship arrowhead is pointing towards the base use-case from the extended use case [Figure 3.23(b)].

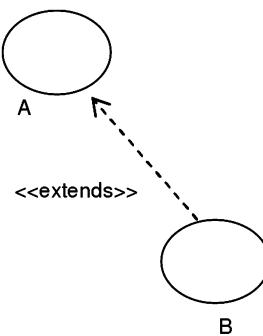


FIGURE 3.23(a) The <<extends>> relationship.

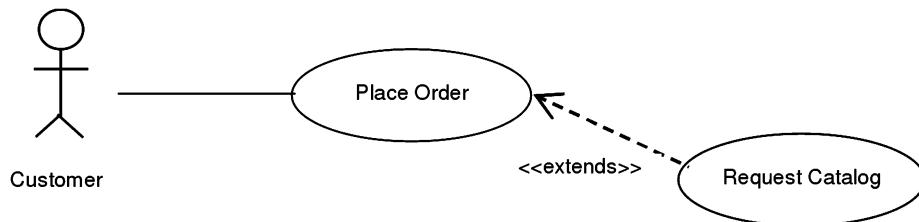


FIGURE 3.23(b) Example: <<extends>>.

3.9.5 Relationships between Actors

The only relationship allowed between actors on a use-case diagram is *generalization*. This is useful in defining overlapping roles between actors (Figure 3.24).

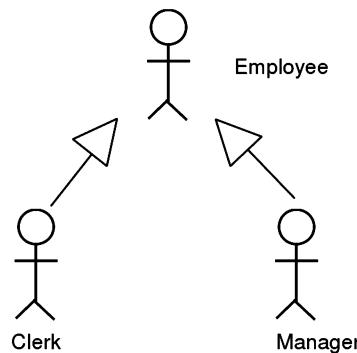


FIGURE 3.24 Generalization.

3.10 GUIDELINES FOR DESIGN OF USE-CASE DIAGRAMS

3.10.1 Actors

1. Primary actors are placed in the top-left corner of the diagram.
2. Actors are external objects; hence actors are always shown outside of the system boundary box of a use-case diagram.
3. Actors are always named with singular, business-relevant nouns.
4. One actor can be related with one or more use-cases.
5. External system as an actor is shown with notation <<system>>.
6. Actors never interact with each other directly, but they interact through the system.

3.10.2 Use-Case

1. Minor steps of functionality should not be considered as a use-case. Functionality as a whole must be considered as a valid use-case.
2. Use-case names must start with a verb.
3. Use-case name should be identified from business terminology and not the technical term.

3.10.3 Relationships

1. If an actor is involved in the use-case logic, a relationship is indicated between the actor and the use-case.
2. Actor-use case relationship is not shown with an arrowhead.

3. Use-case relationships like <<extends>> or <<includes>> should be used only when necessary.
4. An included use-case is placed to the right of the including use-case.
5. An extending use-case is placed below the base use-case.

3.10.4 System Boundary Box

The rectangle around the use-cases is called the system boundary box and as the name suggests it indicates the scope of your system—the use cases inside the rectangle represent the functionality that you intend to implement.

3.11 CASE MERCHANT NATIONAL BANK

Merchant National Bank has its head office in Mumbai and its branches in several parts of India. The bank has several types of accounts like saving a/c, current a/c, PPF a/c, fixed-deposit a/c, locker a/c, NRI a/c and many others. Every account type may or may not be offered at a particular branch. The customers can open any number of joint or single accounts in any branch of the bank. They will be allowed to access their account from any of the branch. The customers do various transactions on their account such as debiting, crediting, depositing and withdrawing local and non-local cheques, making and depositing drafts, operating the locker, making and withdrawing fixed deposits, etc. Periodically, the bank calculates interest as per the current rates of interest and credits it to the accounts of the account holders. From time to time, the head office requires management reports from the branches. The management of the branch offices also requires daily, weekly and monthly, quarterly and annual reports of the operations.

3.11.1 Identifying Actors and Use-Cases

While analyzing the problem statement, you can choose any of the approaches—either find out all the functionalities (Use-cases) occurring in the system, then find out the objects (Actors) who perform those functionalities and relate each use-case with actor or vice versa.

After identifying all use-cases and actors, try to identify relationships among use-cases such as <<extends>>, <<includes>> if any and show generalization relationship among actors if their role overlaps (Table 3.1).

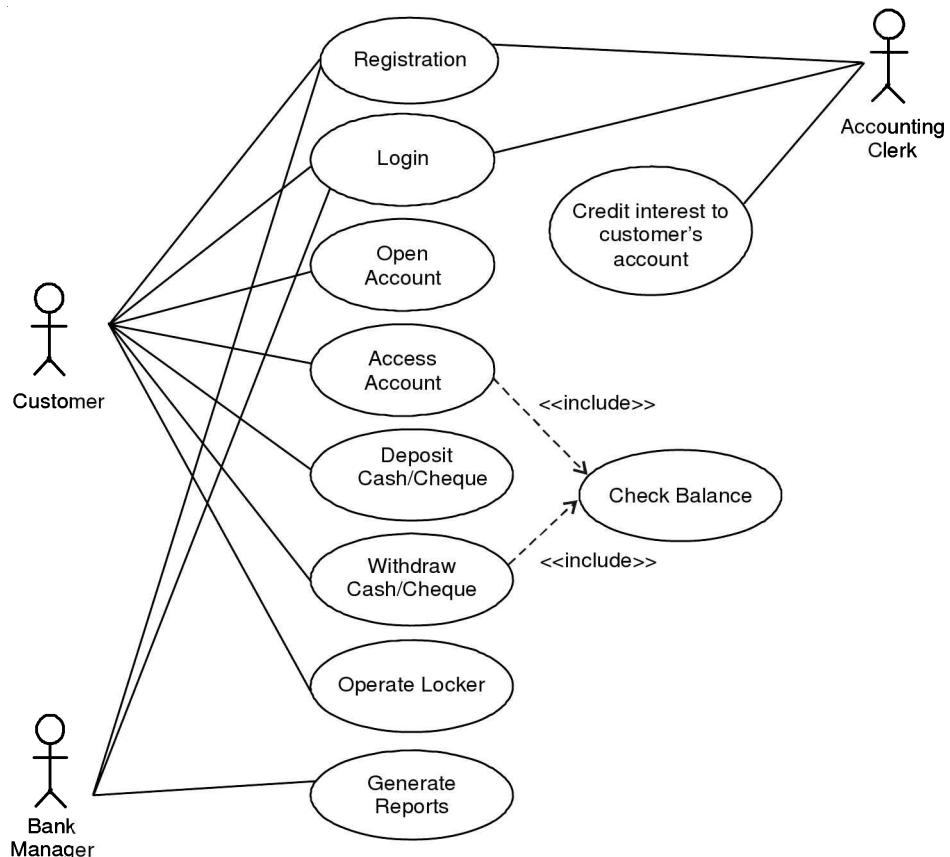
3.11.2 Use-Case Diagram

Figure 3.25 shows the use-case diagram for Merchant National Bank.

Points to ponder: In the use-case diagram shown in Figure 3.25, apart from use-cases obvious from the problem statement as found out in Table 3.1, we have added 2 additional use cases—*Registration* and *Login* which are not mentioned in the problem statement, but we have to consider it as every body has different roles in the system.

TABLE 3.1 Actors and corresponding use-cases for Merchant National Bank

Sr. No.	Actors	Use Cases		
		Base	Include	Extends
1	Customer	Open Account Access Account Operate locker		Check Balance
2	Customer	Deposit Cash/Cheque	Withdraw Money Cash/Cheque	Check Balance
3	Accounting Clerk	Credit interest to Customers' account		
4	Bank Manager	Generate Reports		

**FIGURE 3.25** Use-Cases for Merchant National Bank.

EXERCISES

1. What is BPMN? Explain with example elements of BPMN.
2. Explain the role of the use-case model in software development?
3. Compare the business process diagram and the use-case diagram.
4. Create a list of actors and use-cases in a library system.
5. Customers can perform any action on their accounts such as: withdrawal, deposit, transfer-money, etc. from inside the bank, we assume the existence of account managers that follow these operations.

Construct first a use-case diagram for the following description.

Customers can be golden and in-house (those working-at-the-bank) ones. Golden customers can withdraw with a credit, while in-house ones can deposit with interest. Besides, golden customers can have saving accounts, where they can shift from current account to that saving account as well as increasing their interest.

Extend the previous use-case diagram to cope with additional information.

With the presence of ATM facility, the withdrawal can be direct (at the office) or via ATM. Using the ATM, a customer should enter his/her card, enter pin, enter amount and then get money and the card back.

What change will occur in the use-case model?

6. A student contacts a college for admission. He/she submits his/her details on a preprinted form or from a web page (online system). The college verifies the student details. The management takes the decision (Admission granted or rejected) and conveys the status to the student.

Draw the business process diagram and the use-case diagram for the above case.

7. A Financial Trading System (FTS) is used by an accounting system for updating accounts. It allows analysis of trading risks by traders. The trading manager uses it to set limits on price deals. A trader arrives at price deals after checking with the valuation service of FTS. This valuation service is also used in the analysis of trading risks. A salesperson uses the price deal to capture a deal with a trader. Capturing a deal may sometime need previously set limits to be exceeded.

Draw the business process diagram and the use-case diagram for the above case.

8. Develop a new student registration system that students could use to register for courses at the beginning of each semester and view grade report cards for courses at the end of each semester. The system would also let professors indicate courses they would be teaching in a semester and see the students who have signed up for their course offerings. Professors would also be able to enter student grades into the system.

The school has an existing system used to maintain course catalog information, such as course titles, prerequisites, weekly meeting times, and locations. The registration system would use the course catalogue data for its purposes. Also, after the registration period's completion, the new registration system must send a notification to the school's billing system so that each student can be appropriately billed. At the semester's

beginning, the system lets a student select a maximum of 18 credits—six primary courses.

Further, each student can indicate two alternative three-credit courses that could be substituted for primary course selections. If a course fills up while a student is registering, the system must notify the student of the course's unavailability before it accepts the schedule for processing. A student could access the system to add or delete courses before the add/delete period expires. Once the system completes a student's registration, it notifies the school's billing system. The registrar uses the system to maintain student information, close registration, and cancel course offerings with fewer than five students.

Draw the business process diagram and the use-case diagram for the above case.

9. A game of chess that can be played across a network has to be implemented. The game consists of a chess server and multiple chess clients. To participate, a player has to use the chess client connect to the chess server. The chess server allows only registered users to participate in a game session. The chess server can serve multiple game sessions. While connecting to the server, the player can either join an existing session, or create a new session. When two players join a session, the actual game begins. A valid user of the system can also join an ongoing match session as a viewer. The player creating a new session can make it a private board. In that case, other players cannot join the session unless they know the key to the session. Using the chess client, the player can interact with the other player in the session. For example, a player may wish to propose a draw. If the other player agrees, the session ends in a draw. The chess server maintains the status of all the session boards and the ratings of all the registered players. The chess server validates all moves made by players. The server can enforce the timing constraints if indicated by both players at the start of a game. The server also maintains the archives of all the games played. The clients can play back these archived games for registered users. Each client program can be customized by individual players to remember their identity and preferences. Draw the business process diagram and the use-case diagram for the above case.
10. Sam requires a new point of sale and stock control system for their many stores throughout the UK to replace their ageing mini-based systems. A sales assistant will be able to process an order by entering product numbers and required quantities into the system. The system will display a description, price and available stock. In-stock products will normally be collected immediately by the customer from the store but may be selected for delivery to the customer's home address for which there will be a charge.

If stock is not available the sales assistant will be able to create a back-order for the product from a regional warehouse. The products will then either be delivered direct from the regional warehouse to the customer's home address, or to the store for collection by the customer. The system will allow products to be paid for by cash or credit card. Credit card transactions will be validated via an online card transaction system. The system will produce a receipt. Order details for in-stock products will be printed in the warehouse including the bin reference, quantity, product number and

description. These will be collected by the sales assistant and given to the customer. The sales assistant will be able to make refunds, provided a valid receipt is produced. The sales assistant will also be able to check stock and pricing without creating an order and progress orders that have been created for delivery.

The store manager will be able at any time to print a summary report of sales in the store for a given period, including assignment of sales to sales assistants in order to calculate weekly sales bonuses.

The stock manager will be able to monitor stock levels and weekly run-rates in order to set minimum stock levels and requisition products which fall below the minimum stock levels or for which demand is anticipated. When the stock arrives it will be booked in by the warehouse person. Stock that has been backordered for collection from the store is held in a separate area and the store manager advised of its arrival.

The catalogue of available products will be maintained remotely by marketing from the head office. Marketing will also be able to access sales information from each store system.

Draw the business process diagram and the use-case diagram for the above case.

CHAPTER

4

Class Diagram and Object Diagram

4.1 CLASS DIAGRAMS

The purpose of the class diagram is to show the static structure of the system being modelled. The class diagram captures the static structural aspect of the system. It represents the classes and relationship in the system. The class diagram serves mainly two purposes—understanding requirements and describing the detailed design. Class diagrams are typically used to explore the business concepts in the form of a domain model.

4.1.1 Analysis and Design Versions of a Class Diagram

Depending upon the purpose, the class diagram can be of two types—analysis class diagram representing domain analysis model and design class diagram representing detailed design model. Figure 4.1 specifically shows the difference between the analysis and the design class diagram. Figure 4.1 shows the entities in the system along with each entity's internal structure and their interrelationships, operations and attributes. During the analysis phase the class diagram is used to represent the conceptual model which depicts the detailed understanding of the problem space for the system.

Analysis	Design								
<table border="1"><thead><tr><th>Order</th></tr></thead><tbody><tr><td>Placement Date</td></tr><tr><td>Delivery Date</td></tr><tr><td>Order Number</td></tr><tr><td>Calculate Total</td></tr><tr><td>Calculate Taxes</td></tr></tbody></table>	Order	Placement Date	Delivery Date	Order Number	Calculate Total	Calculate Taxes	<table border="1"><thead><tr><th>Order</th></tr></thead><tbody><tr><td><ul style="list-style-type: none">- deliveryData: Date- orderNumber: int- placementDate: Date- taxes: Currency- total: Currency<p>#calculateTaxes(Country, State): Currency #calculateTotal(): Currency getTaxEngine() {visibility=implementation}</p></td></tr></tbody></table>	Order	<ul style="list-style-type: none">- deliveryData: Date- orderNumber: int- placementDate: Date- taxes: Currency- total: Currency <p>#calculateTaxes(Country, State): Currency #calculateTotal(): Currency getTaxEngine() {visibility=implementation}</p>
Order									
Placement Date									
Delivery Date									
Order Number									
Calculate Total									
Calculate Taxes									
Order									
<ul style="list-style-type: none">- deliveryData: Date- orderNumber: int- placementDate: Date- taxes: Currency- total: Currency <p>#calculateTaxes(Country, State): Currency #calculateTotal(): Currency getTaxEngine() {visibility=implementation}</p>									

FIGURE 4.1 Analysis and design version of class diagram.

As class diagrams only model the static structure of a system, only types of entities are shown on a class diagram, not the specific instances of those entities. Instances of those entities would be shown in the object diagram discussed in the next session. For example, a

class diagram would show a Student class, but would not show actual student instances such as John, Peter or Jimmy.

4.2 ELEMENTS OF CLASS DIAGRAM

4.2.1 Class

In an object-oriented application, classes have attributes (member variables), operations (member functions) and relationships with other classes. The UML class diagram can depict all these things quite easily. The fundamental element of the class diagram is an icon that represents a class. This icon is shown in Figure 4.2.

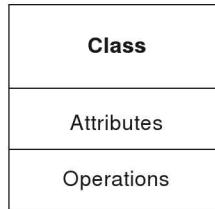


FIGURE 4.2 Class icon in UML class diagram.

A class notation is simply shown as a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables), and the bottom compartment contains a list of operations (member functions). In many diagrams, the bottom two compartments are omitted. Even when they are present, they typically do not show every attribute and operations. The goal is to show only those attributes and operations that are useful for the particular diagram. There is typically never a need to show every attribute and operation of a class on any diagram. For example, refer to Figure 4.3.

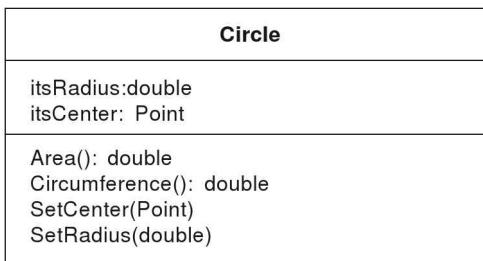


FIGURE 4.3 Example class circle.

Guidelines for identifying classes: The following are the guidelines for identifying classes:

1. Look for *nouns* and *noun phrases* in the problem statement.
2. Some classes are implicit or taken from general knowledge.

3. All classes must make sense in the application domain; avoid computer implementation classes—defer them to the design stage.
4. Carefully choose and define the class names.
5. Finding classes is an incremental and iterative process.
6. Remove redundant, irrelevant, adjective and attribute classes from the list of classes identified.

Guidelines for identifying attributes: The following are the guidelines for identifying attributes:

1. Look for the nouns followed by prepositional phrases such as *price* of the soup.
2. Look for the adjectives or adverbs.
3. Attributes may not be fully described in the problem statement. Draw your knowledge of the application domain and the real world to find them.
4. Do not discover excess attributes, as one can add more attributes in subsequent iterations.

Guidelines for identifying operations: The following are the guidelines for identifying operations:

1. Operations correspond to the methods responsible for managing the value of attributes such as query, updating, reading and writing. For example, *getBalance()*, *setBalance()* methods for the *Account* class.
2. Operations also sometimes correspond to queries about association of the objects. For example, *deposit()*, *withdraw()* methods for the *Account* class associated with the *Transaction* class.

4.2.2 Relationships

The types of relationships in a class diagram are listed as follows:

Association

Association represents a physical or conceptual connection between two or more classes. A reference from one class to another is an association. Some associations are implicit or taken from general knowledge. Associations are the “glue” that ties a system together. Figure 4.4(a) shows the graphic way of representing association.

FIGURE 4.4(a) Association relationship.

Association often corresponds to a **verb** or **prepositional phrase** such as *work for*, *talk to*, *order to*, *next to*.

Association may be *bidirectional*, *unidirectional* or *reflexive association*.

Bidirectional association: Associations are always assumed to be bidirectional. Bidirectional association means that both classes are aware of each other and their relationship. A bidirectional association is indicated by a solid line between the two classes as shown in Figure 4.4(b).

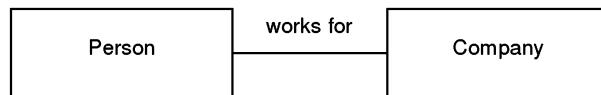


FIGURE 4.4(b) Bidirectional association.

Bidirectional association *works for* between the class *Person* and the class *Company* indicates that the person knows about the company, and the company knows about the person. When implemented, it will be easy to know for which company a person is working, and which person a company is hiring.

Unidirectional association: In a unidirectional association, two classes are related, but only one class knows that the relationship exists. Also second class need not pass messages to the first class at run-time. A unidirectional association is drawn as a solid line with an arrowhead at one end as shown in Figure 4.4(c).



FIGURE 4.4(c) Unidirectional association.

Given a polygon it is possible to query all points that make it up. But given a specific point, it is not possible to find which polygons the point is part of.

Reflexive association: All the examples above have shown a relationship between two different classes. However, a class can also be associated with itself, using a reflexive association. This may not make sense at first, but remember that classes are abstractions. Figure 4.4(d) shows how an Employee class could be related to itself through the manager/manages role. When a class is associated to itself, this does not mean that a class's instance is related to itself, but that an instance of the class is related to another instance of the class.

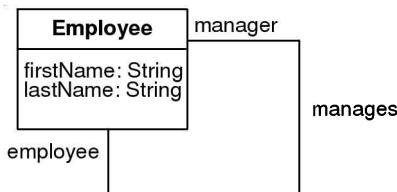


FIGURE 4.4(d) Réflexive association.

Each connection of an association to a class is called an *association end*. The association end names appear as nouns in the problem statement. The use of association end names is optional, but it is often easier and less confusing to assign association end names as shown in Figure 4.4(e).

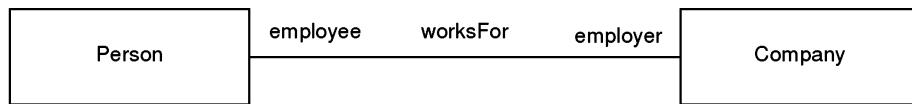


FIGURE 4.4(e) Use of association end name.

As shown in the above figure, a name appears next to the association end. In the figure, the person and the company participate in association *works for*. A person is an *employee* with respect to a company and a company is an *employer* with respect to a person.

Association end names are necessary for associations between two objects of the same class. Association end names can also distinguish multiple associations between the same pair of classes [Figure 4.4(f)].

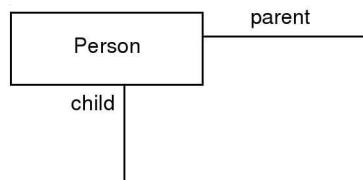


FIGURE 4.4(f) Example of multiple association.

Generalization

Generalization is a relationship between a class (the superclass) and one or more variations of the class (the subclass).

The instance of a subclass is fully consistent with the instance of a super class and contains additional information. A generalization is used to indicate inheritance. For generalization we look for classes with similar attributes or methods and group them by moving the common attributes and methods to an abstract class. We move attributes and methods as high as possible in the hierarchy. The rule is not to create very specialized classes at the top of the hierarchy. Generalization is sometimes called the “is-a” relationship, because each instance of a subclass is an instance of the superclass as well. Figure 4.5(a) shows the way to represent generalization.

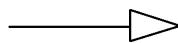


FIGURE 4.5(a) Generalization relationship.

In Figure 4.5(b), *Contact* is a superclass with two subclasses *Client* and *Company*. In each of *Client* and *Company* all of the attributes in *Contact* (address, city, etc.) exist, but with more information added. When using a generalization link, the child classes have the option to override the operations in the parent class. That is, they can include an operation that is defined in the superclass, but define a new implementation for it.

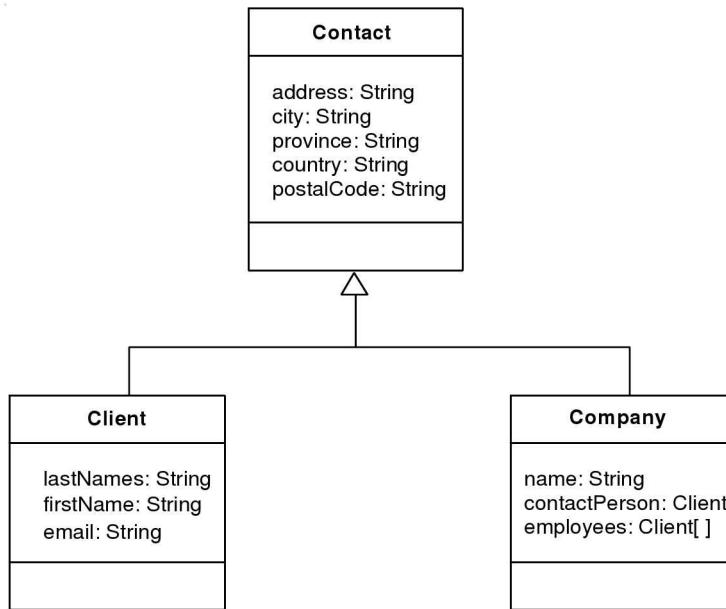


FIGURE 4.5(b) Example of generalization relationship.

Aggregation

Aggregation is a kind of association in which an aggregate object is made up of constituent parts. When a class is formed as a collection of other classes, it is called an aggregation relationship between these classes. It is also called a “has a” relationship [Figure 4.6(a)]. A class that is composed of other classes does not behave like its parts, it behaves very differently. Identify *a-part-of relation* between objects from the prepositional phrases like *part of*, *contained in*. Aggregation is drawn like association except a small hollow diamond indicates the assembly end.

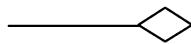


FIGURE 4.6(a) Aggregation relationship.

In Figure 4.6(b) a lawn mower consists of blade, engine, wheels and deck. The Lawn Mower is an aggregate object and blade, engine, wheels and deck are its constituent parts.

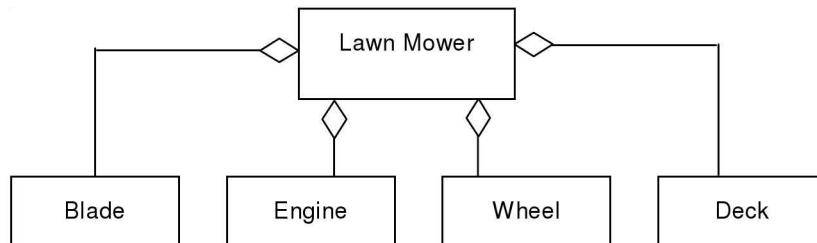


FIGURE 4.6(b) Representing aggregation.

Composition

Composition is a variation of the aggregation relationship. It indicates that a strong life cycle is associated between the classes [Figure 4.7(a)]. Composition is a form of aggregation with two additional constraints. A constituent part can belong to at the most one assembly. Once a constituent part has been assigned, an assembly its life is associated with the assembly.

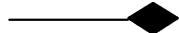


FIGURE 4.7(a) Composition relationship.

The black diamond in Figure 4.7(b) represents composition. It is placed on the *Circle* class, because it is the circle that is composed of a point. The arrowhead on the other end of the relationship denotes that the relationship is navigable in only one direction. That is, the point does not know about the circle. In UML relationships are presumed to be bidirectional unless the arrowhead is present to restrict them.

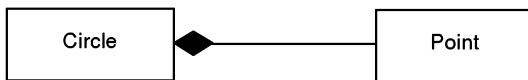


FIGURE 4.7(b) Representing composition.

Composition relationships are a strong form of containment or aggregation. Aggregation is a whole/part relationship. In this case, the circle is the whole, and the point is part of the circle. However, composition is more than just aggregation. Composition also indicates that the lifetime of point is dependent upon the circle. This means that if the circle is destroyed, the point will be destroyed with it.

Dependency

A dependency is a weak relationship between two classes and is represented by a dotted line with an open arrowhead [Figure 4.8(a)]. Dependency relationship indicates that one class depends on other class. If class A depends on class B, then change in class B affects class A.

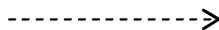


FIGURE 4.8(a) Dependency icon.

In the example shown in Figure 4.8(b), there is a dependency between the class *Point* and the class *Circle*, because *Circle*'s *draw()* operation uses the *Point* class. It indicates that the *Circle* class depends on the *Point* class. If there is a change in the point, it will affect the circle. Also, the circle has to know about the point, even if it has no attributes of that type.

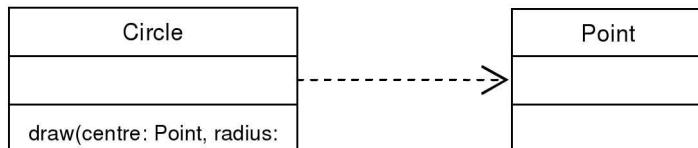


FIGURE 4.8(b) Representing dependency.

Multiplicity

Place multiplicity notations (Table 4.1) near the ends of an association. These symbols indicate how the instances of one class are linked instances of the other class.

TABLE 4.1 Multiplicity indicators

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
3	Three only
0..5	Zero to Five
5..15	Five to Fifteen
2,4	Two or Four

For example, one company will have one or more employees, but each employee works for one company only [Figure 4.9 (a)].

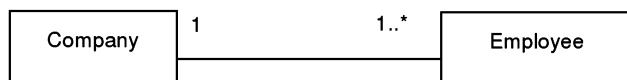


FIGURE 4.9(a) Representing multiplicity.

In other example, as shown in Figure 4.9(b), the number of polygons will have 3 or more points as vertices.

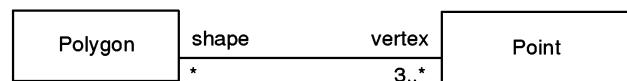


FIGURE 4.9(b) Representing multiplicity.

4.2.3 Association Class

As you describe a class with attributes, similar way you can also describe an association with attributes. UML represents such information with an association class. An association class is a construct that allows an association connection to have operations and attributes and participate in associations. Association classes are typically modelled during analysis. Association classes can be identified by looking for adverbs in a problem statement. Association classes are depicted as class attached via a dashed line to the association line.

The class and the dashed line are considered one symbol in UML, [refer to Figure 4.10(a)].

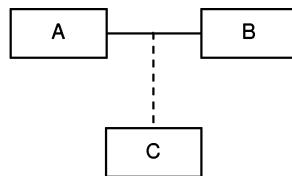


FIGURE 4.10(a) Association class.

Here, C is an association class.

The example depicted in Figure 4.10(b) shows that there is an association between the *Employee* class and the *Project* class which represents the employee is allocated to a project. The role the employee takes up on the project is a complex entity and contains detail that does not belong in the employee or project class. An employee may be working on several projects at the same time and have different job titles and security levels on each. Here *Role* is an association class.

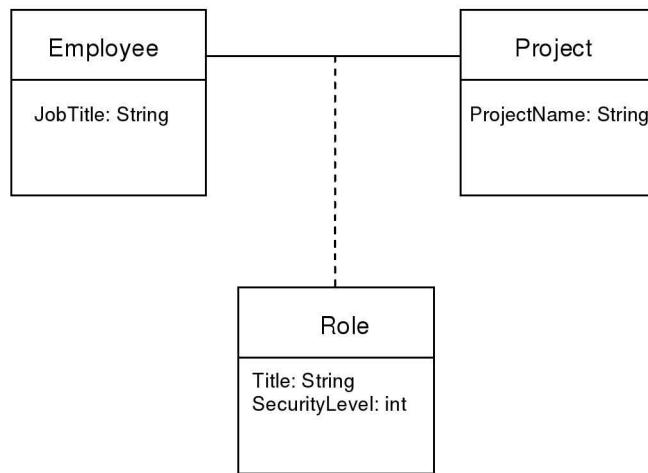


FIGURE 4.10(b) Representing association class.

4.2.4 Interface

An interface is a variation of a class. An interface provides only a definition of business functionality of a system. A separate class implements the actual business functionality. An interface shares the same features as a class, i.e. it contains attributes and methods. The only difference is that the methods are only declared in the interface and will be implemented by the class implementing the interface.

To elaborate these are classes that have nothing but pure virtual functions. The primary icon for an interface is just like a class except that it has a special denotation called a *stereotype*. Figure 4.11(a) shows this icon. The two surrounding characters “«»” are called *guillemets* (pronounced *Gee-may*).

The «type» stereotype [Figure 4.11(b)] indicates that the class is an interface. This means that it has no member variables, and that all of its member functions are pure virtual. UML supplies a shortcut for «type» classes.

Figure 4.11(c) shows how the “lollipop” notation can be used to represent an interface. Notice that the dependency between *Shape* and *DrawingContext* is shown as usual. The class *WindowsDC* is derived from, or conforms to, the *DrawingContext* interface. This is a shorthand notation for an inheritance relationship between the *WindowsDC* and *DrawingContext*.

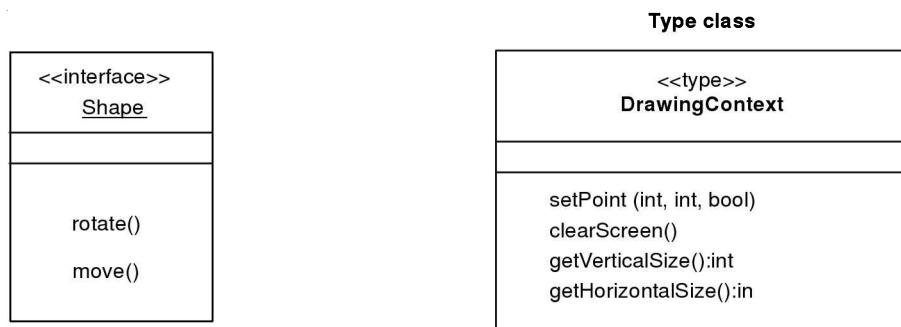


FIGURE 4.11(a) Interface icon.

FIGURE 4.11(b) Interface example.

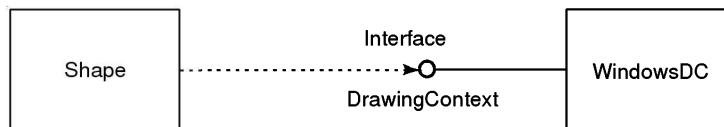


FIGURE 4.11(c) Lollypop notation to represent an interface.

In Figure 4.12, the *OutputDevice* interface is implemented by the *Printer* and *Monitor* classes.

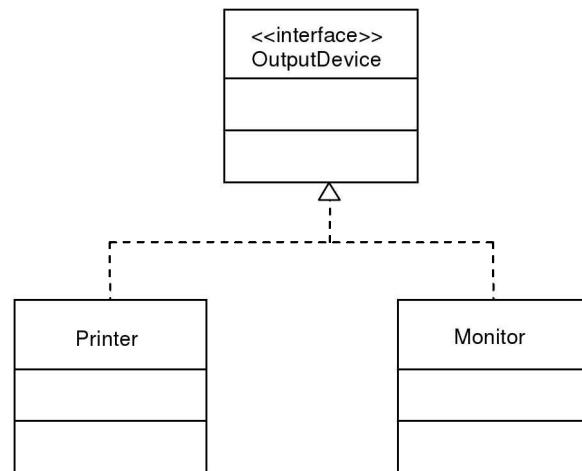


FIGURE 4.12 Representing interface output design.

4.2.5 Package

A package provides the ability to group together classes and/or interfaces that are either similar in nature or related. Grouping these design elements in a package element provides for better readability of class diagrams, especially complex class diagrams. A package can also have relationships with other packages similar to relationships between classes and interfaces (Figure 4.13).



FIGURE 4.13 Package icon.

4.3 GUIDELINES FOR DESIGN OF A CLASS DIAGRAM

1. As class diagrams are used for a variety of purposes—from understanding requirements to describing a detailed design—it is required to apply a different approach in each circumstance.
2. During analysis, a domain analysis class diagram will be drawn first.
3. Identify responsibilities of each domain class in domain class diagrams.
4. Visibility (private, protected and public) is indicated only on design models.
5. Design class diagrams should follow language naming conventions.
6. An association class is modelled on the analysis diagram and the association name is not specified in that case.
7. Draw a dashed line from the centre of association joining two classes to the association class.
8. The class name should be singular.
9. The class names, with which the client is comfortable, should be selected, rather than semantically accurate terminology.

4.4 PROBLEM STATEMENT: MERCHANT NATIONAL BANK

Merchant National Bank has its head office in Mumbai and its branches in several parts of India. The bank has several types of accounts like saving a/c, current a/c, PPF a/c, fixed-deposit a/c, locker a/c, NRI a/c and many others. Every account type may or may not be offered at a particular branch. The customers can open any number of joint or single accounts in any branch of the bank. They will be allowed to access his account from any of the branch. The customers do various transactions on their account such as debiting, crediting, depositing and withdrawing local and non-local cheques, making and depositing drafts, operating the locker, making and withdrawing fixed deposits etc. Periodically, the bank calculates interest

as per the current rates of interest and credits it to the accounts of the account holders. From time to time, the head office requires management reports from the branches. The management of the branch offices also requires daily, weekly and monthly, quarterly and annual reports of the operations.

4.4.1 Analysis of Merchant National Bank

1. The bank has its head office.

Class 1	Relationship Name	Relationship Type	Class 2
Bank	Has	Association	Head-office

2. The bank has branches.

Class 1	Relationship Name	Relationship Type	Class 2
Bank	Has	Association	Branches

Both the above examples are simple associations with 1:1 and 1: $*$ multiplicity respectively as shown in Figure 4.14(a).

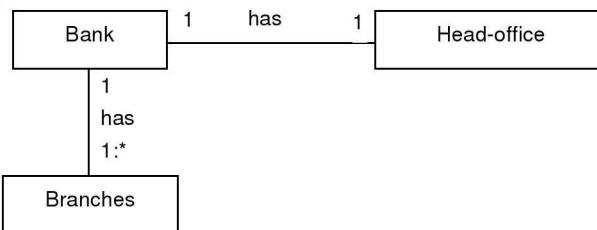


FIGURE 4.14(a) Part of class diagram—Merchant National Bank.

3. Several types of accounts like saving account, current account, PPF account, fixed deposit account, locker account may be offered at branches.

Class 1	Relationship Name	Relationship Type	Class 2
Branches	Offer	Association	Accounts
Account	Is-of	Generalization	Account type

Here, the first part, several account types offered at various branches is a simple association. In the second part, the *Account* class is a generalized class for rest of the classes based on account types. Hence Figure 4.14(b) shows generalization relationship among the *Account* and other account types.

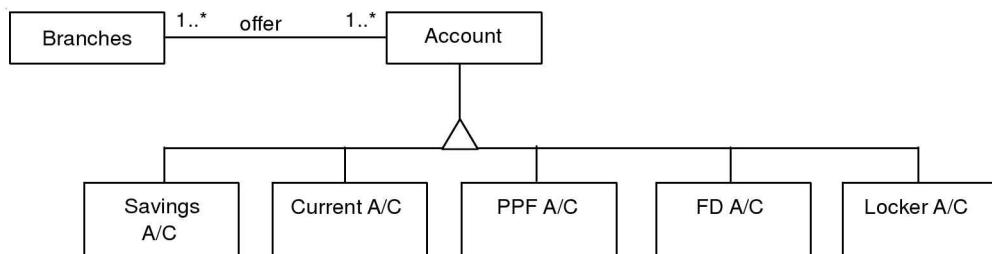


FIGURE 4.14(b) Part of class diagram—Merchant National Bank.

4. The customer opens any number of joint or single accounts [refer to Figure 4.14(c)].

Class 1	Relationship Name	Relationship Type	Class 2
Branches	Offer	Association	Accounts

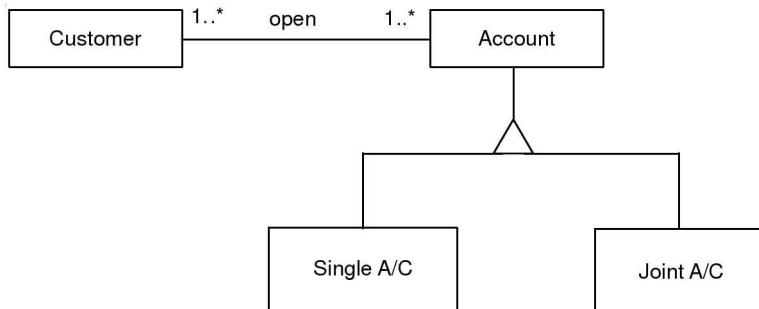


FIGURE 4.14(c) Part of class diagram—Merchant National Bank.

5. The customer performs transactions on his/her the accounts [refer to Figure 4.14(d)].

Class 1	Relationship Name	Relationship Type	Class 2
Customer	Performs	Association	Transaction

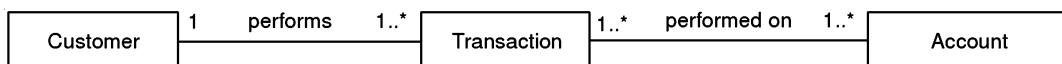


FIGURE 4.14(d) Part of class diagram—Merchant National Bank.

6. The head office checks management reports [Figure 4.14(e)].

Class 1	Relationship Name	Relationship Type	Class 2
Branches	Offer	Association	Accounts

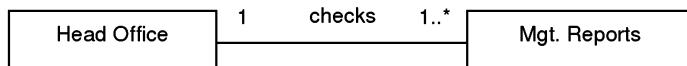


FIGURE 4.14(e) Part of class diagram—Merchant National Bank.

Combining all the segments of the class diagram, we will get a complete class diagram representing the static structure of the system.

4.4.2 Class Diagram—Merchant National Bank

Classes and relationships (at a glance): A *class* is any real-world entity (identified by extracting all nouns from problem statement). A *relationship* binds classes (identified by extracting all verbs from problem statement, which relate classes.) (refer to Table 4.2).

TABLE 4.2 Classes and their relationships—Merchant National Bank

Sr. No.	Class 1	Relationship Name	Relationship Type	Class 2
1	Bank	Has	Association	Head-office
2	Bank	Has	Association	Branches
3	Branches	Offer	Association	Accounts
4	Account	Is-of	Generalization	Account type
5	Customer	Opens	Association	Account
6	Customer	Performs	Association	Transaction
7	Head office	Checks	Association	Management Reports

Figure 4.15 shows the complete class diagram of the bank.

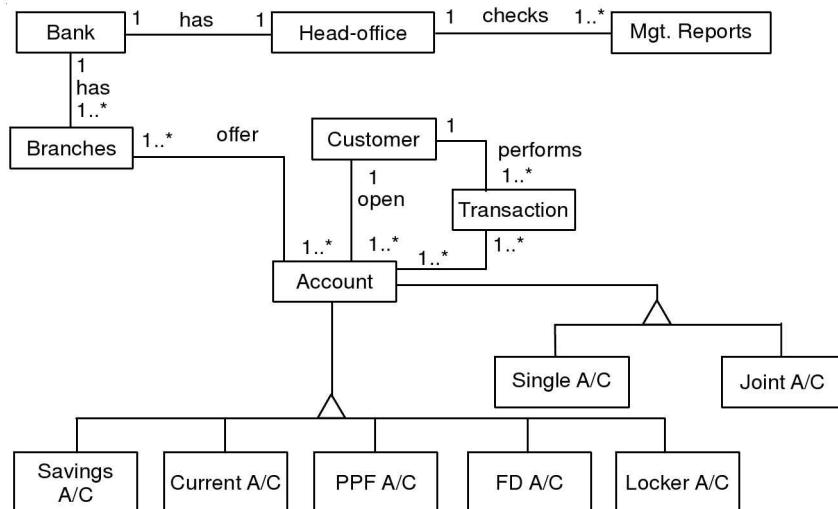


FIGURE 4.15 Complete class diagram—Merchant National Bank.

4.5 OBJECT DIAGRAM

An object diagram in the Unified Modelling Language (UML) is a diagram that shows a complete or partial view of the structure of a modelled system at a specific time. Object diagrams, sometimes referred to as *instance diagrams*, are useful for exploring real-world examples of objects and the relationships between them. Although UML class diagrams are very good at describing this very information, some people find them too abstract—a UML object diagram can be a good option for explaining complex relationships between classes. Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams. Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system, but this static view is a snapshot of the system at a particular moment.

Object diagrams are used to render a set of objects and their relationships as an instance hence it is more close to the actual system behaviour. The purpose is to capture the static view of a system at a particular moment. An object diagram may be considered as a special case of a class diagram.

Object diagrams are useful in understanding class diagrams. They do not show anything architecturally different to class diagrams, but reflect multiplicity and roles. Object diagrams are useful for showing examples of objects connected together. The object diagram is very useful when the possible connections between objects are complicated.

They are useful for explaining small pieces of information with complicated relationships, especially recursive relationships and are more concrete than class diagrams, often used to provide examples, or act as test cases for the class diagrams. Only those aspects of a model that are of current interest need to be shown on an object diagram.

4.6 ELEMENTS OF OBJECT DIAGRAM

UML object diagrams use a notation similar to class diagrams and are used to show an instance of a class at a particular point in time.

4.6.1 Objects

The object is an instance of a class. Instantiate a class with certain attribute values at a particular instance. Look for the proper nouns in the problem statement to identify objects. The UML symbol for an object is a box with an object name followed by a colon and the class name. The object name and class name are both underlined (Figure 4.16).

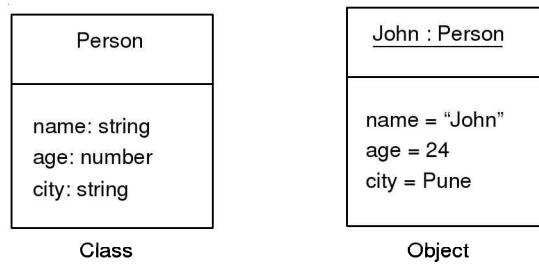


FIGURE 4.16 Representing object *John* for class *Person*.

4.6.2 Links

A link is a physical or conceptual connection among objects. Relationship among classes is instantiated as a link. The UML notation for a link is a line between objects. If the link has a name, it is underlined (Figure 4.17).

FIGURE 4.17 Link to connect object.

For example, India is an instance of the class *Country*, Delhi is an instance of the class *Capital* and a link is connecting these two objects as shown in Figure 4.18.

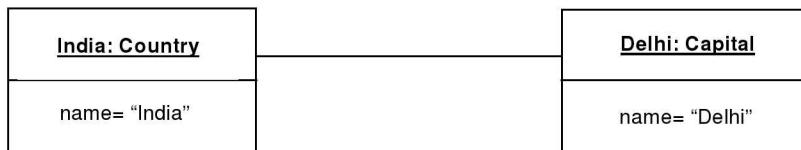


FIGURE 4.18 Representing link.

4.7 GUIDELINES FOR DESIGN OF OBJECT DIAGRAMS

The following are the guidelines for the design of object diagrams.

1. The most important objects are to be identified.
2. The link among objects should be clarified.
3. The values of attributes of objects need to be captured to include in the object diagram.
4. Proper notes should be added at points where more clarity is required.
5. You can create object diagrams by instantiating the classes that exist in a class diagram.
6. In which phase of the project you are, depending on that, class instances are created.
An object diagram is created for some portion of the class diagram and not for the whole class diagram.

4.8 PROBLEM STATEMENT: ABC INDIA PVT. LTD.

ABC India Pvt. Ltd. has many departments. Departments have managers.

4.8.1 Analysis of ABC India Pvt. Ltd.

1. A company has many departments [Figure 4.19(a)].

Class 1	Relationship	Class 2
Company	Has	Departments

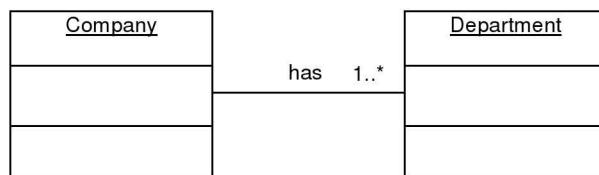


FIGURE 4.19(a) Part of class diagram—ABC India Pvt. Ltd.

2. Departments have managers [Figure 4.19(b)]. That is, one department has one manager.

Class 1	Relationship	Class 2
Department	Have	Manager

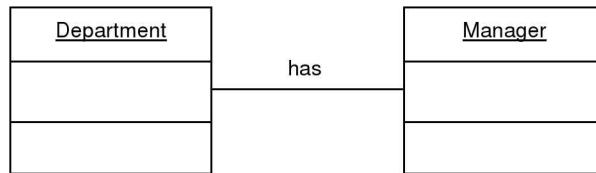


FIGURE 4.19(b) Part of class diagram—ABC India Pvt. Ltd.

3. A complete class diagram for the above statement is shown in Figure 4.19(c).

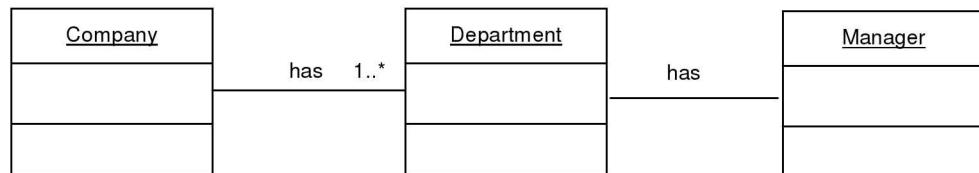


FIGURE 4.19(c) Complete class diagram—ABC India Pvt. Ltd.

4.8.2 Object Diagrams for ABC India Pvt. Ltd.

Instantiate classes *Company*, *Department* and *Manager* with specific attribute values as shown in Table 4.3.

TABLE 4.3 Classes, objects and their links—ABC India Pvt. Ltd.

Class	Objects	Links
Company	c	Company—Department
Department	d1 (name= “Sales”) d2 (name= “R&D”) d3 (name= “US Sales”)	Department—Department
Manager	m (name= “Erin”, employeeID = 4362, title= “VP of Sales”)	Department—Manager

In the object diagram (shown in Figure 4.20), object c (instance of class *Company*) links object d1 and d2 (instances of class *Department*). Object d1 linked with object d3 as d3 is a sub-department of d1 and object m is linked with object d3 as manager of “US Sales” department d3.

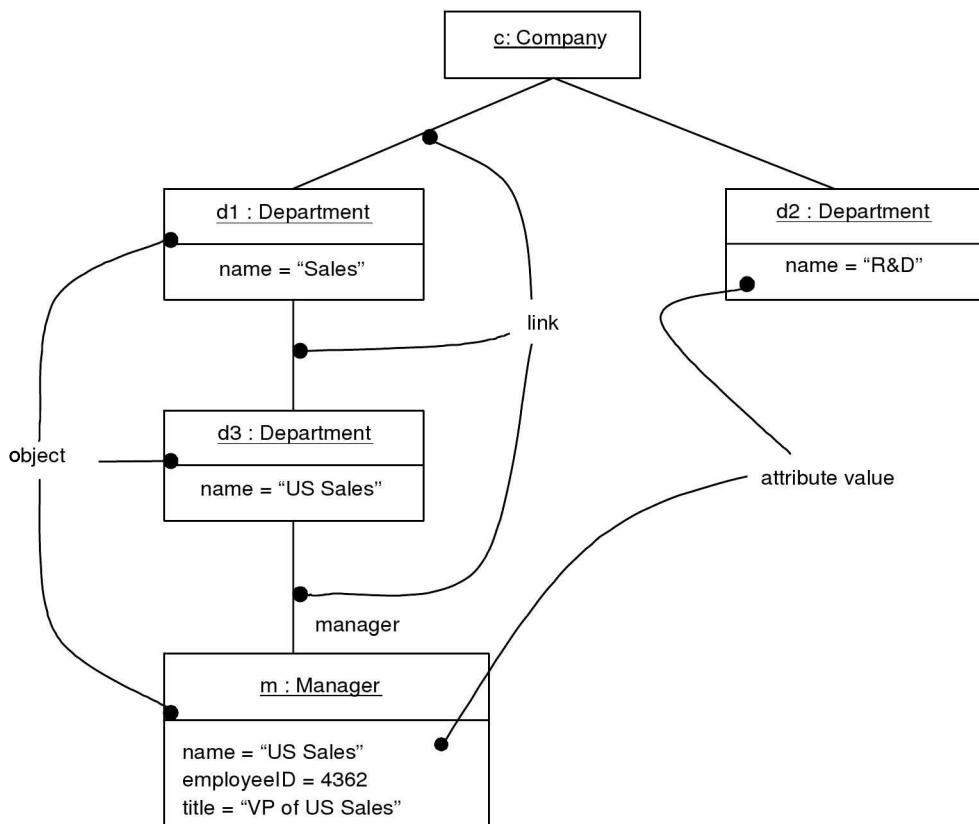


FIGURE 4.20 Object diagram—ABC India Pvt. Ltd.

Drawing the object diagram for the whole case is nearly impossible and very elaborate. Hence object diagrams are drawn only for specific instances which are of importance and represent the behaviour of the system at a particular instance.

EXERCISES

1. Differentiate between the class and the object with an example.
2. Explain the different types of relationships in a class diagram.
3. Compare links and associations.
4. Compare aggregation and composition giving an example.

5. What is an association class? Give an example.
6. Explain interface and package with an example.
7. What is the difference between a analysis class diagram and a design class diagram?
8. A publisher publishes different books. An author can write different books but for the same publisher. A contract is signed between the publisher and the author. Reports such as the number of books sold, number of complimentary copies given, royalty amount to be paid to the author, etc., are generated from the system. Draw a class diagram and an object diagram for the above case.
9. Draw a class diagram and an object diagram for a car rental application. The car rental agency has multiple offices/branches. The customer visits the agency for enquiry and takes a test ride, then selects the car by signing the “terms and conditions” form. The customer can also book the car through telephone, email and SMS. The agency checks the availability of the car and gives status to the customer. The customer can also avail the driver facility if required, by paying additional charges. The billing is done based on the type of vehicle and distance travelled.
10. An organization offers a variety of courses in a variety of areas such as learning management techniques and understanding different software languages and technologies. Each course is made up of a set of topics. Tutors in the organization are assigned courses to teach according to the area that they specialize in and their availability. The organization publishes and maintains a calendar of the different courses and the assigned tutors every year. There is a group of course administrators in the organization who manage the course including course content, assign courses to tutors and define the course schedule. The training organization aims to use the Courseware Management System to get a better control and visibility to the management of courses as also to streamline the process of generating and managing the schedule of different courses. Draw a class diagram and an object diagram for the above case.

CHAPTER 5 Sequence Diagram and Collaboration Diagram

5.1 INTRODUCTION TO SEQUENCE DIAGRAM

A sequence diagram is used to express use-case realizations, i.e. to show how objects interact to perform the behaviour of all or part of a use-case. A sequence diagram is an interaction diagram that emphasizes the time ordering of messages. The sequence diagram is used primarily to show the interactions between the different objects of a system, in the sequential order of the occurrence of interactions.

A sequence diagram depicts the sequence of actions that occur in a system and captures the invocation of methods in each object, and the order in which these invocations occur. Sequence diagrams are used to model the flow of logic within the system enabling both documentation and validation of logic.

The main purpose of a sequence diagram is to define event sequences that result in some desired outcome. The focus is less on messages and more on the time order in which messages occur. However, most sequence diagrams communicate what messages are sent between objects as well as the order in which they are sent.

The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top-down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to. A sequence diagram is the dynamic modelling diagram focusing on identifying the behaviour of related objects within the system. Sequence diagrams are used to model the usage scenario, the logic of methods and services.

Sequence diagrams are used to give a picture of the design because they provide a way to visually step through invocation of the operations defined by classes. One can easily understand the need to change the design and to re-distribute the load within the system by just having a glance at what messages are being sent to an object, and by looking at approximately how long it takes to run the invoked method.

Sequence diagrams are used to display the interaction between users, screens, objects and entities within the system. They provide a sequential map of message passing between objects over time. Frequently these diagrams are placed under use-cases in the model to illustrate the use-case scenario—how a user will interact with the system and what happens internally to get the work done.

5.2 ELEMENTS OF SEQUENCE DIAGRAM

A sequence diagram is two-dimensional in nature. On the vertical axis, it shows the life of the object that it represents, while on the horizontal axis, it shows the sequence of the creation or invocation of these objects. A sequence diagram is made up of objects and messages. Basic elements of sequence diagram are as follows:

5.2.1 Life Lines

Lifelines represent either roles or instances that participate in the sequence of interaction being modelled. Lifeline notation elements are placed at the top of the diagram.

Lifelines are drawn as a rectangle with a dashed line descending from the centre of the bottom edge of the rectangle and the lifeline's name is placed inside the rectangle. Figure 5.1(a) is one of the ways to represent lifelines. Here a named object "John" belonging to class "Person" is placed inside the box showing the lifeline.

Another way of representing the life line is shown in Figure 5.1(b). The lifeline can represent an anonymous or unnamed instance also. When modelling an unnamed instance on a sequence diagram, representing an anonymous instance of the Person class, the lifeline would be ":Person" as shown in Figure 5.1(b).

As the sequence diagrams are used during the design phase of projects, it is possible to have an object whose type is unspecified. This third option to draw a lifeline is shown in Figure 5.1(c) where the lifeline shows "John" as unspecified class.

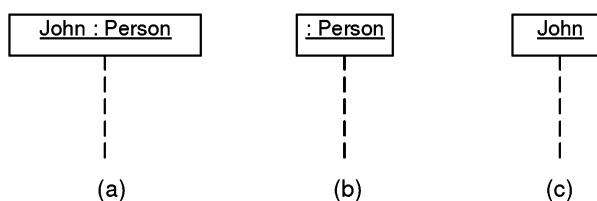


FIGURE 5.1 Lifelines.

5.2.2 Messages

A message defines a specific kind of communication between instances in an interaction. It represents a method call and invocation between objects. A message specifies the sender and the receiver, and defines the kind of communication that occurs between lifelines. For example, a communication can invoke, or call, an operation using a `synchCall` or `asynchCall`, and create or destroy a participant.

As shown in Figure 5.2 a line with a solid arrowhead that points towards the receiving lifeline represents a synchronous call operation in which the system waits for the flow of control to complete before continuing with the outer flow.

A line with an open arrowhead represents an asynchronous call in which the source object sends the message and immediately continues with the next step.

A dashed line with a solid arrowhead that points towards the originating lifeline represents a return message from a call to a procedure (Figure 5.2).

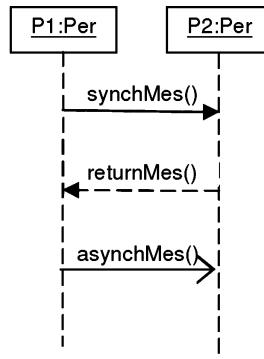


FIGURE 5.2 Message.

5.2.3 Activation

Activation boxes represent the time an object needs to complete a task. Activation is created automatically when we create a synchronous or an asynchronous message. Each activation represents an execution in a behaviour. Activation boxes also called, *focus of control* shown as a tall, thin rectangle on a lifeline, represents the period during which an element is performing an operation. The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively. These are represented using rectangular boxes in Figure 5.3.

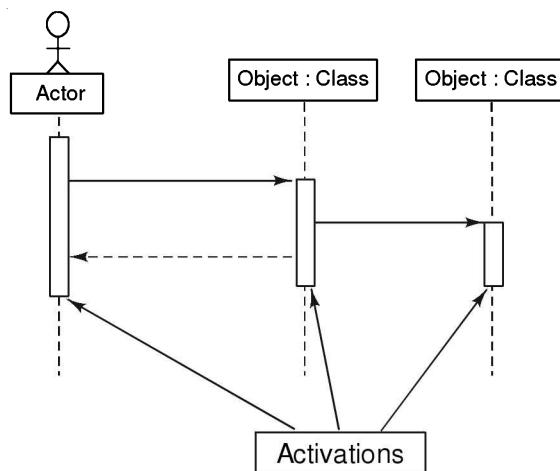


FIGURE 5.3 Activations.

5.2.4 Guards

Guards are the conditions used throughout UML diagrams to control flow. When modelling object interactions, there will be times when a condition must be met for a message to be sent to the object.

To display a guard on a sequence diagram, we place the guard element above the message line being guarded within square brackets and in front of the message name, e.g. [balance < 100] debitCharges() will invoke the method debitCharges() if balance<100 as shown in Figure 5.4.

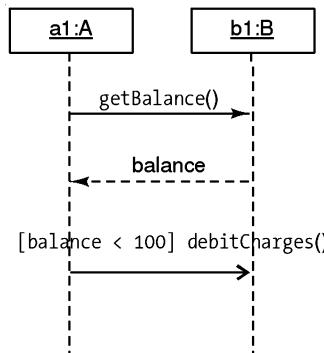


FIGURE 5.4 Guards.

5.2.5 Combined Fragments

Initially, before UML 2, sequence diagrams were lacking in notation to show complex procedural logic. In UML 2 combined fragments notation was introduced for adding procedural logic to sequence diagrams.

A combined fragment is used to group sets of messages together to show a conditional flow in a sequence diagram.

There are three main combined fragments in sequence diagrams—(a) alternatives, (b) options, and (c) loops.

(a) Alternatives: They are used to make only one choice between two or more message sequences. Alternatives model the typical “if then else” logic (e.g., if I purchase above Rs. 10000, then I will get 20% discount; else I will get 10% discount).

An alternative combination fragment element is drawn using a frame. The word “alt” is placed inside the frame’s name box. The larger rectangle is then divided into interaction operands. Interaction operands are containers that group interaction fragments in a combined fragment and are separated by a dashed line. Each operand is given a guard to test against, and this guard is placed towards the top left section of the operand. If an operand’s guard condition is “true”, then that operand sequence is to be executed.

Apart from only “if then else” condition, alternative combination fragments can also have many alternative paths as required. To have one more alternative, an operand with that sequence’s guard and messages must be added. Alternatives are shown in Figure 5.5(a).

(b) Options: The option combination fragment models a sequence that will occur for a certain condition, otherwise, the sequence does not occur.

An option fragment models a simple “if then” statement (i.e. if a customer purchases above Rs. 5000, give 10% discount, otherwise no discount). The option combination fragment

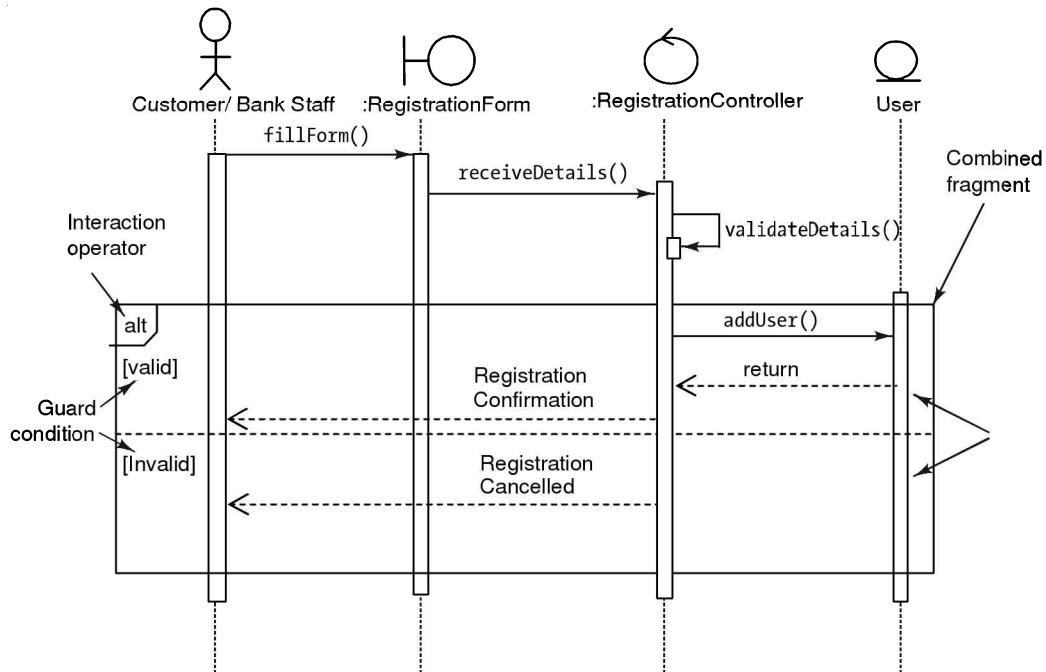


FIGURE 5.5(a) Representing alternatives.

notation is similar to the alternative combination fragment having only one operand and no other condition.

For an option combination, a frame is drawn and the text “opt” is placed inside the frame’s name box, and the option’s guard is placed towards the top left corner of the frame. The option’s sequence of messages is placed in the remainder of the frame’s content area. Figure 5.5(b) represents options.

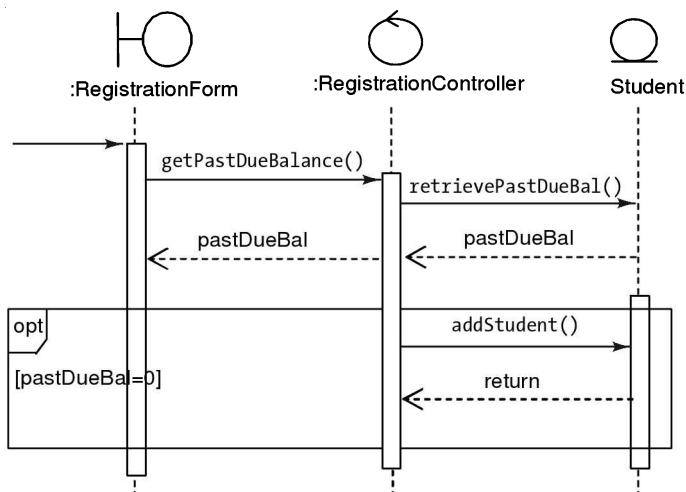


FIGURE 5.5(b) Representing options.

(c) Loops: The loop combination fragment is used to model an iterative sequence. This is the same as the option combination fragment except the guard condition which indicates the basis of iteration for the message sequence.

For the loop combination fragment, a frame is drawn and the text “loop” is placed inside the frame’s namebox, the loop’s guard is placed towards the top left corner in the frame Figure 5.5(c) represents the loop.

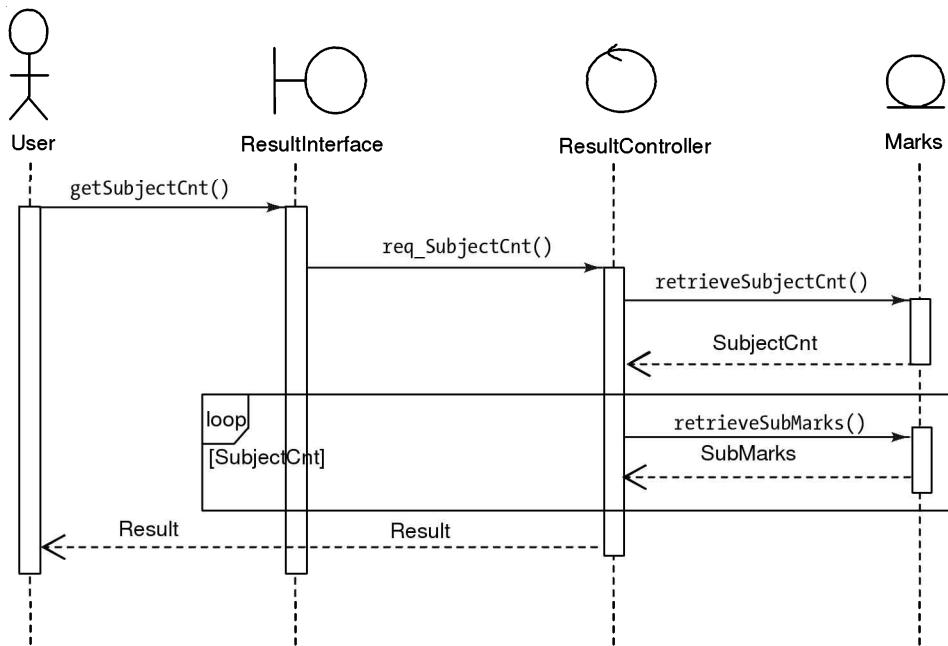


FIGURE 5.5(c) Representing loop.

In a loop, a guard can have two special conditions tested against, in addition to the standard Boolean test. The special guard conditions are minimum iterations written as “`minint=[the number]`” and maximum iterations written as “`maxint=[the number]`”.

With a minimum iterations guard, the loop must execute at least the number of times indicated, whereas with a maximum iterations guard the number of loop executions cannot exceed the number.

5.2.6 Objects

There are four types of objects which interact with each other in a sequence diagram.

- (a) Actor object
- (b) Boundary object
- (c) Controller object
- (d) Entity object

If any other type of objects is to be shown, then the notation used, is a rectangle with *object name : class name* as shown in Figure 5.6.

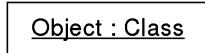


FIGURE 5.6 Object icon.

(a) Actor object: Actor object is an instance of a class which initiates the task. It represents an external person or entity that interacts with the system. The actor in the sequence diagram is the same actor interacting with that specific use-case in the use-case diagram (Figure 5.7).

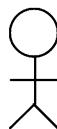


FIGURE 5.7 Customer—actor object.

(b) Boundary object: Boundary objects, instances of boundary classes, are used to model the interaction between a system and its actors. The boundary class may be a user interface class, system interface class or device interface class. A user interface class is used for communication of human users with the system.

A system interface class is used for communication of other systems with the system. A device interface class provides the interface to device.

e.g. In a web application, the user interface boundary classes represent the web pages such as in ASP.NET, *.aspx pages (Figure 5.8).

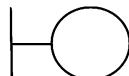


FIGURE 5.8 Application form—boundary object.

(c) Controller object: Controller objects are used to model the control behaviour specific to one or a few use-cases. Controller objects represent the use-case logic and coordinates the other classes. It separates the interface classes from business logic classes.

In ASP.NET the code-behind classes can be regarded as controller classes. Each controller object is closely related to the realization of a specific use-case. Some controller objects can participate in more than one use-case realization if the use-case tasks are strongly related.

Several controller objects can participate in one use-case. Not all use-cases require a control object. Some use-cases may be without controller objects (just using entity and boundary objects)—particularly use-cases that involve only the simple manipulation of stored information.

More complex use-cases generally require one or more control classes to coordinate the behaviour of other objects in the system. Examples of control objects include programs, such as transaction managers, resource coordinators and error handlers (Figure 5.9).

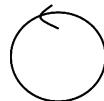


FIGURE 5.9 Login controller—controller object.

(d) Entity object: An entity object is used to model information and associated behaviour that must be stored. Entity objects (Figure 5.10) are used to hold and update information about some phenomenon, such as an event, a person or some real-life object.

They are usually persistent, having attributes and relationships needed for a long period, sometimes for the life of the system and can be used to generate database schema directly. An actor often provides the values of its attributes and relationships. Entity objects represent the key concepts of the system being developed. Typical examples of entity classes in a banking system are, the Account and Customer.

Entity objects represent stores of information in the system. Entity objects are frequently passive and persistent. Their main responsibilities are to store and manage information in the system (Figure 5.10).

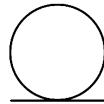


FIGURE 5.10 User—entity object.

5.3 GUIDELINES FOR DESIGN OF SEQUENCE DIAGRAMS

The following are guidelines for the design of sequence diagrams.

1. For each use-case, one sequence diagram is drawn.
 - For each use-case, identify the object (person, external system, or device) which initiates the task and that will be the actor for a sequence diagram.
 - Find out the interface (form or screen) through which an actor interacts with the system and that will be the boundary object for a sequence diagram.
 - Find out the object which does not have the interface and controls the use-case and that will be the controller object for sequence diagram.
 - Find out the object which stores and manages information, usually the database table and that will be the entity object for sequence diagram.
2. Always have one boundary class per actor/use case pair, one control class per use-case.
3. Ordering of message sequence is always shown from left to right.
4. An actor name must be the same as specified in a use-case diagram.
5. An actor can have the same name as a class.
6. Primary actors must be specified on the left-most side of the diagram.
7. Reactive system actors must be specified on the right-most side of your diagram.

8. Proactive system actors must be specified on the left-most side of your diagram.
9. An object can call itself recursively. An arrow commencing and ending at itself denotes this.

5.4 PROBLEM STATEMENT: MILTON JEWELS PVT. LTD.

Milton Jewels has specialized in online jewellery retail since 1998, selling wonderful ranges of both children's and women's jewellery. A customer can register online so that he/she can check the status of the placed order. A customer can purchase any jewellery item online either by using his/her existing account or as an anonymous user specifying shipping address and contact information. Customer can only check the status of his/her order if he/she creates an account. The customer will pay online through credit card or debit card and the order will be delivered on the shipping address within one week.

5.4.1 Analysis of Milton Jewels Pvt. Ltd.

A sequence diagram needs to be drawn during the design phase of the system.

During analysis, we have got:

- Use-case diagram
- Class diagram

We will make use of both these diagrams for design.

- For each use-case in the use-case diagram, we will draw one sequence diagram at least. If the use-case is much complex, then there can be more than one sequence diagram.
- While drawing the sequence diagram from the use-case diagram, we will draw the sequence diagram only for the base use-cases embedding the flow for <<include>> and <<extend>> use-cases in the same.

In the above case, actors and use-cases identified are summarized in Table 5.1.

TABLE 5.1 Actors and their corresponding use-cases—Milton Jewels Pvt. Ltd.

Sr. No.	Actors	Use-cases
1	Customer	Registration Login Purchase jewellery Check order status Search jewellery View jewellery
2	Store Manager	Registration Login Add/Edit/Remove jewellery Add/Edit/Remove Categories Specify discounts

- During design, identify four types of objects interacting with each other to perform the use-case such as:
 1. Actor objects—Person, external system or device that initiates the use case, e.g. Customer, Store Manager.
 2. Boundary objects—All the interfaces through which the actor interacts with the system, e.g. Login Screen, Application Form, etc.
 3. Controller objects—All the objects which co-ordinate the task, e.g. Security Manager, etc.
 4. Entity objects—All domain entities identified during analysis of the class diagram, e.g. Users, products etc.

5.4.2 Sequence Diagrams: Milton Jewels Pvt. Ltd.

From the identified use-cases and actors as above, consider use-case Login performed by both actors, the Customer and Stores Manager, which is the most common functionality in nearly every application. For the Login use-case, ask the following four questions and answers to those questions will be the objects interacting with each other for login process.

1. Who will login? (Actor)
Customer/Store Manager
2. Through which web page (interface)? (Boundary)
Login Screen
3. Who will validate the user? (Control)
Security Manager
4. Which object holds valid user details? (Entity)
Users

After identifying interacting objects, it is required to find out what sequences of message communication happen among them. Message communication occurs through method calls, also listed in Table 5.2.

TABLE 5.2 Classes their types and methods—Milton Jewels Pvt. Ltd.

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Customer	<i>Not required to specify in this context.</i>
Boundary	LoginScreen	login()
Control	SecurityManager	validateUser()
Entity	Users	retrieveUserDetails

Figure 5.11 shows one sequence diagram corresponding to the use-case for login utility. There will be many more such sequence diagrams corresponding to the total number of use-cases and depending upon the complexity of the use-cases. The other sequence diagrams are not shown here and left to the reader as an exercise.

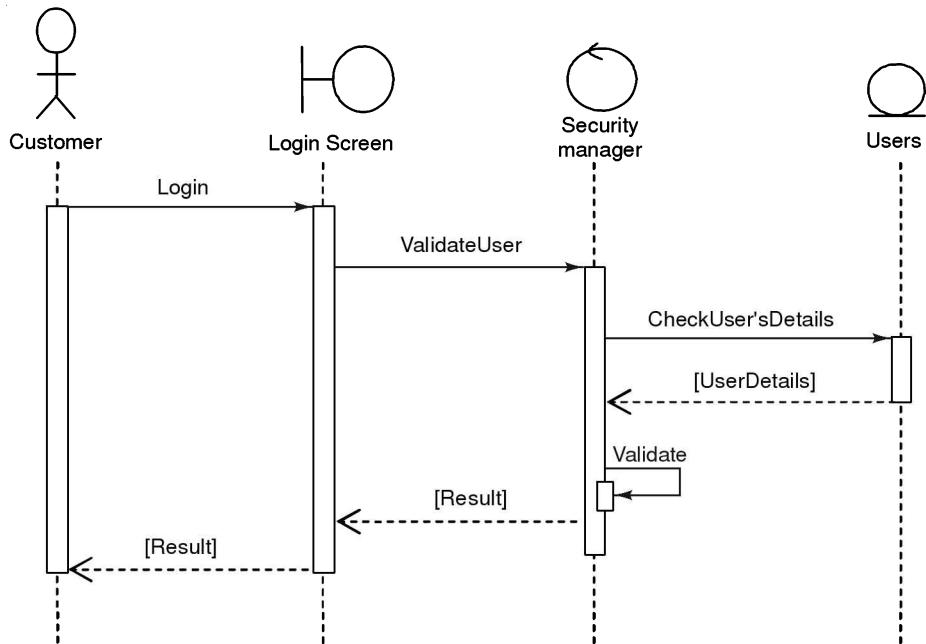


FIGURE 5.11 Sequence diagram of Milton Jewelers Pvt. Ltd. for Login Use-Case.

5.5 COLLABORATION DIAGRAM

A Collaboration diagram is a kind of interaction diagram, called as Communication diagram in UML 2.0. They describe the interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use-case diagrams describing both the static structure and dynamic behaviour of a system.

Like UML sequence diagrams, they are used to explore the dynamic nature of the system which emphasizes the data links between the various participants in the interaction. Instead of drawing each participant as a lifeline and showing the sequence of messages by the vertical direction as the sequence diagram does, it allows free placement of participants, allowing drawing links to show how the participants connect, and use numbering to show the sequence of messages.

Unlike a sequence diagram, a collaboration diagram shows the relationships among the objects. A collaboration diagram does not show time as a separate dimension, so sequence numbers determine the sequence of messages and the concurrent threads.

Collaboration diagrams show the message flow as well as relationships between objects and are often used to provide a bird's-eye view of a collection of collaborating objects. These diagrams are used to model the logic of the implementation of a complex operation, particularly one that interacts with a large number of other objects. They are better to draw when you want to emphasize the links between the objects, whereas sequence diagrams are better when you want to emphasize the sequence of calls.

5.6 ELEMENTS OF COLLABORATION DIAGRAM

The elements used to draw a collaboration diagram are almost similar as used in sequence diagrams. The lists of elements, their notation and examples are given as follows.

5.6.1 Links

The links connecting objects in a collaboration diagram show the various paths available for messages; they do not provide detailed information about the links. For example, the links in a collaboration diagram do not show the multiplicity of the links (Figure 5.12).

FIGURE 5.12 Link in a collaboration diagram.

5.6.2 Messages

A message defines a specific kind of communication between instances in an interaction. A message specifies the sender and receiver, and defines the kind of communication that occurs between lifelines.

A message flow carries a message from one object to another along the link. Each message flow in a collaboration diagram is characterized by direction and is numbered starting with 1. The number with the message represents the order/sequence of this interaction (Figure 5.13).

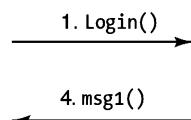


FIGURE 5.13 Messages in a collaboration diagram.

5.6.3 Objects

There are four types of objects which interact with each other in a collaboration diagram very similar to sequence diagram and already explained in section 5.2.6.

- (a) Actor objects
- (b) Boundary objects
- (c) Controller objects and
- (d) Entity objects

If any other type of objects is to be shown, then the notation used is a rectangle as shown in Figure 5.6.

5.7 GUIDELINES FOR DESIGN OF COLLABORATION DIAGRAMS

The guidelines for the design of a collaboration diagram are as follows:

1. Draw one collaboration diagram for each use-case.
2. Derive the collaboration diagrams from the sequence diagrams or create the object message trace in the same way that a sequence diagram is created, but represent it with the collaboration diagram notation.
3. To derive a collaboration diagram from a sequence diagram,
 - Draw each object (actor, boundary, control, entity) from the sequence diagram.
 - If the sequence diagram shows a message between the objects, draw a line connecting the objects on the collaboration diagram.
 - Label the line with the message name and a number identifying the sequence in which the message appears.
4. Name the actors consistently with the use case diagrams.

5.8 PROBLEM STATEMENT: MILTON JEWELS PVT. LTD.

Milton Jewels has specialized in online jewellery retail since 1998, selling wonderful ranges of both children's and women's jewellery. A customer can register online so that he/she can check the status of the placed order. A customer can purchase any jewellery item online either by using his/her existing account or as an anonymous user specifying shipping address and contact information. The customer can only check the status of his/her order if he/she creates an account. The customer will pay online through credit card or debit card and order will be delivered on the shipping address within one week.

5.8.1 Analysis of Milton Jewels Pvt. Ltd.

Since the sequence diagram of Milton Jewellers Pvt. Ltd. is drawn the collaboration diagram is easy to derive. Only one sample collaboration diagram is shown as follows. The rest of the collaboration diagrams are left for the reader as an exercise.

5.8.2 Collaboration Diagram: Milton Jewels Pvt. Ltd.

In the first part of this chapter, we have a sequence diagram ready for one of the several use-cases performed in the system, i.e., sequence diagram for Login use-case. We will derive the collaboration diagram from the sequence diagram (Figure 5.14). The objects, links and related methods are shown in Table 5.3.

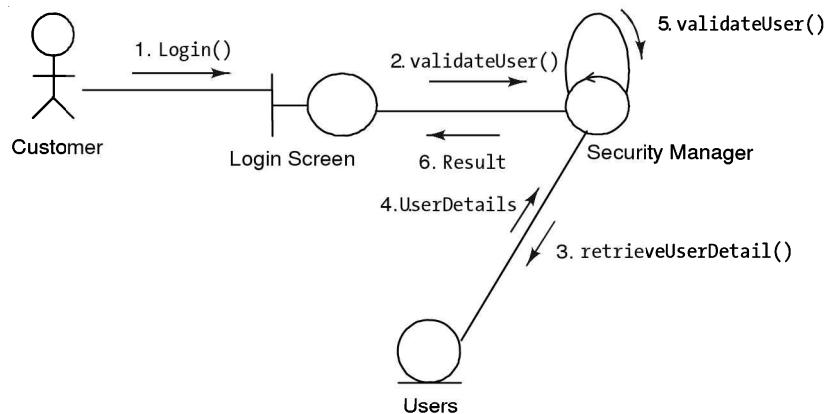


FIGURE 5.14 Collaboration diagram—Milton Jewels Pvt. Ltd. for Login use-case.

TABLE 5.3 Objects, links and methods—Milton Jewels Pvt. Ltd.

Objects	Links	Methods (with sequence number)
Customer	Customer—LoginScreen	1. Login()
LoginScreen	LoginScreen—SecurityManger	2. validateUser() 6. Result
SecurityManager	SecurityManager—SecurityManager	5. validatUser()
Users	SecurityManager—Users	3. retrieveUserDetail() 4. UserDetails

EXERCISES

1. Differentiate between a sequence diagrams and a collaboration diagram.
2. Explain the different types of objects in a sequence diagram.
3. Refer to section 5.4 and 5.8, and draw the sequence diagrams and collaboration diagrams for the remaining use-cases.
4. Draw the sequence and collaboration diagrams to withdraw money from a bank.
5. Draw the sequence and collaboration diagrams to deposit money into a bank.
6. Draw the sequence and collaboration diagrams for shopping cart application.
7. Draw the sequence and collaboration diagrams for an online Air Ticket Reservation System.
8. Draw the sequence and collaboration diagrams for a Hotel Reservation System.

9. The main function of a self-service machine is to allow a customer to buy a product from the machine (candy, chocolate, juice ...).

The customer wants to buy some of the products offered by the self-service machine. First of all he/she inserts money into the machine, selects one or more products, and the machine presents a selected product to the customer. It is possible that the self-service machine is out of one or more products, or the machine has not the exact amount of money to return to the customer.

When the customer inserts money into the machine and enters his or her selection, if the machine is out of brand, in this case it's preferable to present a message to the customer that the machine is out of brand and allow him or her to make another selection or return money back. If incorrect-amount-of-money scenario has happened, then the self-service machine is supposed to return original amount of money to the customer. A supplier has to restock the machine, and a collector has to collect the accumulated money from the machine. Draw the sequence diagram.

10. A new Student Registration System is to be developed which students could use to register for courses at the beginning of each semester and view grade report cards for courses at the end of each semester. The system would also let professors indicate courses they would be teaching in a semester and see students who have signed up for their course offerings. Professors would also be able to enter student grades into the system.

The school has an existing system used to maintain course catalogue information, such as course titles, prerequisites, weekly meeting times and locations. The registration system would use the course catalogue data for its purposes. Also, after the registration period's completion, the new registration system must send a notification to the school's billing system so that each student can be appropriately billed.

At the semester's beginning, the system lets a student select a maximum of 18 credits—six primary courses. Further, each student can indicate two alternative three-credit courses that could be substituted for primary course selections. If a course fills up while a student is registering, the system must notify the student of the course's unavailability before it accepts the schedule for processing.

A student could access the system to add or delete courses before the add/delete period expires. Once the system completes a student's registration, it notifies the school's billing system. The registrar uses the system to maintain student information, close registration, and cancel course offerings with fewer than five students. Draw the sequence and collaboration diagrams for the system.

C|H|A|P|T|R 6 Activity Diagram and State Chart Diagram

6.1 INTRODUCTION TO ACTIVITY DIAGRAM

In object-oriented analysis & design, activity diagrams, in many ways, are the equivalent of flow charts and data flow diagrams (DFDs) from structured development showing flow of control from activity to activity. An activity diagram is typically used for modelling the logic captured by a single use-case or usage scenario. The diagram can also be used to model a specific actor's workflow within the entire system. Activity diagrams can also be used independent of use-cases for other purposes such as to model business process of a system, to model detailed logic of business rules, etc.

An activity diagram shows all potential sequence flows in an activity. Activity diagrams are used for simple and perceptive illustration of what happens in a workflow, what activities can be done in parallel, and whether there are alternative paths through the workflow.

An activity diagram is a dynamic diagram that shows the activity and the event that causes the object to be in the particular state. It shows the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity. Activity diagrams may be used to detail situations where parallel processing may occur in the execution of some activities.

Activity diagrams are useful for business modelling where they are used for detailing the processes involved in business activities and business and operational workflows of a system.

6.2 ELEMENTS OF ACTIVITY DIAGRAM AND THEIR NOTATION

6.2.1 Initial State

An initial state is an element that explicitly shows the beginning of a workflow on an activity diagram. It is the point at which reading of the activity diagram begins. Because an activity diagram shows a sequence of actions, it must indicate the starting point of the sequence using the initial state element. The initial state is drawn as a solid circle with an optional name or label as shown in Figure 6.1.



FIGURE 6.1 Begin admission process—initial state.

6.2.2 Final State

A final state is an element that explicitly shows the end of a workflow on an activity diagram. Unlike initial state, there can be multiple final states in an activity diagram to indicate termination of specific branches of the workflow. The final state is drawn as a filled circle inside a larger unfilled circle called *bull's-eye*, with an optional name or label as shown in Figure 6.2.



FIGURE 6.2 End of admission process—final state.

6.2.3 Action/Activity

Action states represent the non-interruptible actions of objects which cannot be further decomposed and takes insignificant execution time. The activity states can be further decomposed, their activity being represented by other activity diagrams. Also activity states are not atomic, so they may be interrupted and takes some time to complete.

There is no notational distinction between action and activity states, except that an activity state may have additional parts, such as entry and exit action. Action/Activity is drawn as a rounded rectangle with a name or description of action/activity as shown in Figure 6.3.

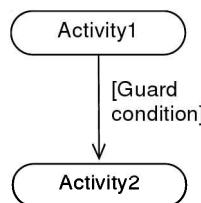


FIGURE 6.3 Activity in admission process.

6.2.4 Transitions

A transition element connects the various elements of the activity diagram. Typically the transition element represents the workflow between two or more actions/activities or other elements of the activity diagram. It is drawn as a solid line with an open arrowhead.

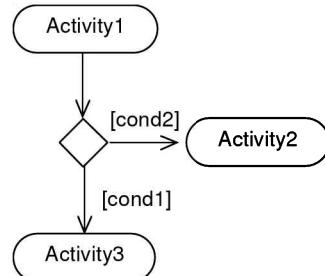
The transition element can have an optional label enclosed in square brackets called *guard condition* or simply *guard*. Guard conditions are used to define the conditional logic that controls the flow of control, and the specific criteria that must be met for a transition to occur as shown in Figure 6.4.

**FIGURE 6.4** Transitions.

6.2.5 Decisions

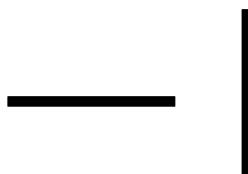
A decision element typically has one incoming transition and two or more outgoing transitions based upon the outcome of guard conditions from the previous element.

A decision box is drawn as a diamond shape usually without a name or label (no need for name/label because the guard conditions usually imply the reason for the decision). Transitions that leave a decision model element often have guard conditions as shown in Figure 6.5.

**FIGURE 6.5** Decision.

6.2.6 Synchronization, Fork and Join

A synchronization element allows modelling of simultaneous workflows in an activity diagram, i.e. parallel activities. Synchronizations visually define forks and joins that represent a parallel workflow or execution and is drawn as a horizontal or vertical bar with no name or label as illustrated in Figure 6.6(a).

**FIGURE 6.6(a)** Synchronization.

A fork is a kind of synchronization element that facilitates the modelling of simultaneous workflows in an activity. A fork identifies where a single flow of control divides into two or

more separate, but simultaneous flows. It is drawn as a bar with one transition going into it and two or more transitions leaving it as in Figure 6.6(b).

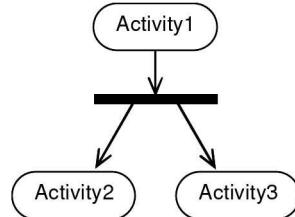


FIGURE 6.6(b) Fork.

A join is another kind of synchronization element that facilitates the modelling of simultaneous workflows in an activity. It identifies where two or more simultaneous flows of control unite into a single flow of control. A join is drawn as a bar with two or more transitions going into it and one transition leaving it as in Figure 6.6(c).

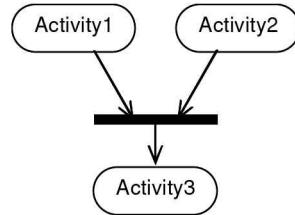


FIGURE 6.6(c) Join.

6.2.7 Swimlanes

A swimlane is an element that can represent a user, an organizational unit, or a role in an activity diagram. Swimlanes depict who or what is responsible for carrying out specific activities. They enable grouping of actions on the activity diagram performed by the same actor or by a single thread. An action transition can take place between two swimlanes. Swimlanes are drawn as vertical columns or horizontal rows with a name associated with each swimlane as shown in Figure 6.7.

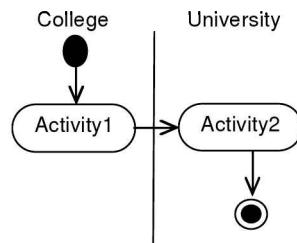


FIGURE 6.7 Swimlanes.

6.2.8 Objects and Object Flows

An object flow is a path along which objects or data can pass. An object is shown as a rectangle. Object flow refers to the creation and modification of objects by activities.

- An object flow arrow from an action/activity to an object means that the action/activity creates or influences the object.
- An object flow arrow from an object to an action/activity indicates that the action/activity state uses the object.

An object flow must have an object on at least one of its ends as shown in Figure 6.8.

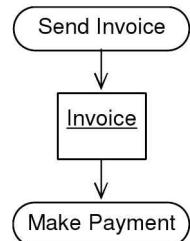


FIGURE 6.8 Object and Object Flows.

6.3 GUIDELINES FOR DESIGN OF ACTIVITY DIAGRAM

The following are the guidelines for the design of an activity diagram.

1. One activity diagram should be drawn for each use-case. No two use-cases flow should be mixed into a single activity diagram. An activity diagram can also be drawn for a system or an actor as a whole.
2. Always an activity diagram should reflect business flow rather than system flow.
3. Only one initial state element should be drawn in an activity diagram.
4. In case of swimlanes, the initial state should be placed in the first swimlane.
5. As a thumb rule, it should be avoided to have more than five swimlanes in a single activity diagram.
6. An initial state should be connected directly to “Action/Activity” element of the activity diagram and not to any other element.
7. The initial state must be connected to only one action/activity element and not to multiple action/activity elements.
8. On every transition, leaving a decision end, a guard condition must be specified.
9. Every fork must have a corresponding join.
10. A fork must have only one entering transition and two or more leaving transitions.
11. A join must have two or more entering transitions and only one leaving transition.
12. It is always better to provide a name/label for the initial and final states.

6.4 PROBLEM STATEMENT: MCA ADMISSION SYSTEM

MCA admission procedure as controlled by Directorate of Technical Education (DTE) is as follows:

1. DTE advertises the date of MCA entrance examination.
2. Student has to apply for the entrance examination.
3. Results are declared by DTE.
4. Student has to fill up the option form to select the college of his/her choice.
5. DTE displays the allotment list in the web site and intimation to all colleges.
6. Students should report the allotted colleges and complete the admission procedure.

6.4.1 Analysis of MCA Admission System

For drawing an activity diagram for the whole system we,

1. Find out swimlanes if any. To find swimlanes, see if we can span some activities over different organizational units/places.
2. Find out in which swimlane the admission process begins and where it ends. Those will become the initial and final states.
3. Then, identify activities occurring in each swimlane. Arrange activities in sequence flow spanning over all the swimlanes.
4. Identify conditional flow or parallel flow of activities. Parallel flow of activities must converge at a single point using join bar.
5. During the activities are performed, if any document is generated or used, take it as an object and show the object flow.

Swimlanes identified for admission process are shown in Figure 6.9.

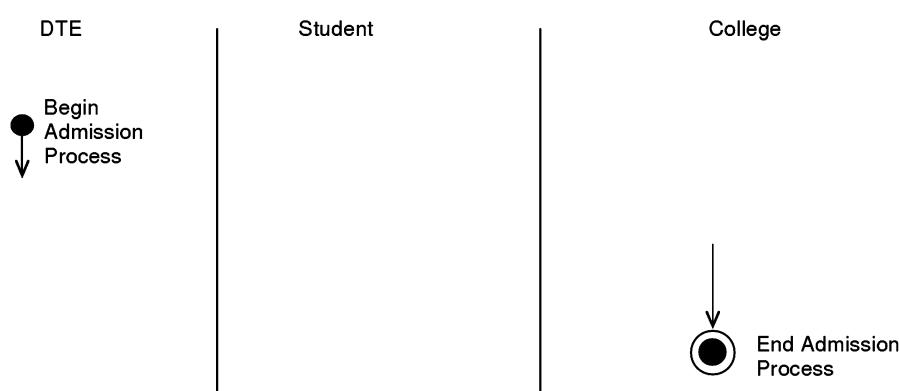


FIGURE 6.9 Swimlanes in admission process.

Activities identified in admission process are shown in Figure 6.10.

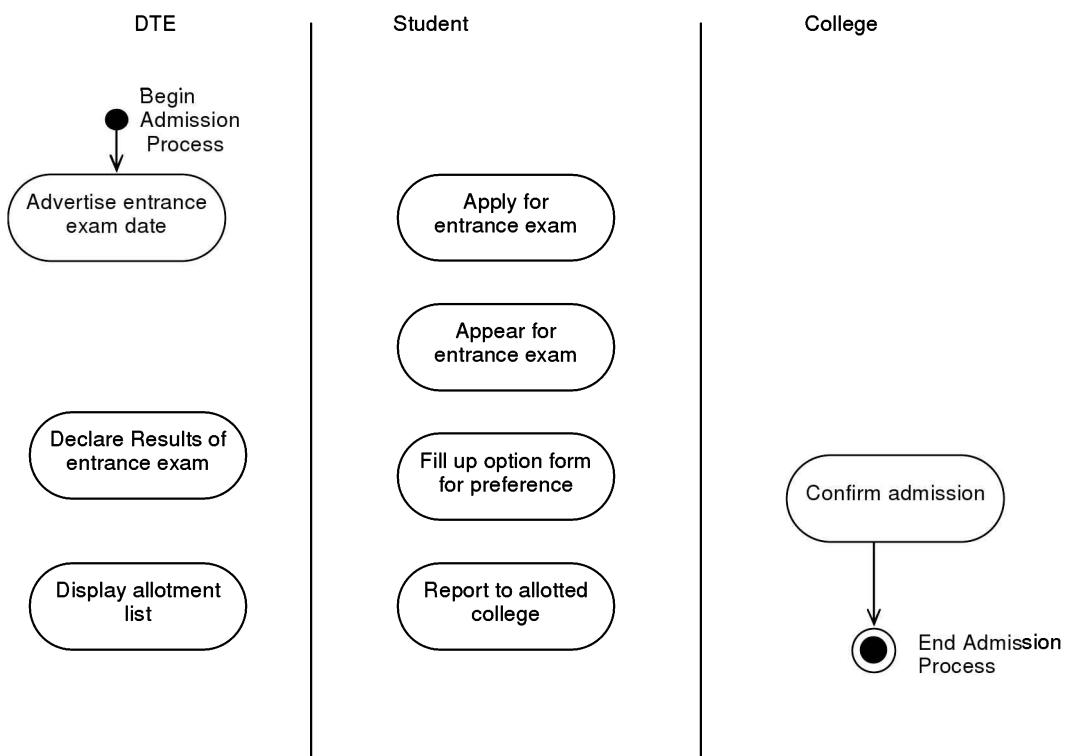


FIGURE 6.10 Activities in admission process.

As stated in the problem statement, no conditional flow and parallel flow is identified. *Fill up option form for preference* activity generates object Option Form. Hence the object flow is shown in Figure 6.11.

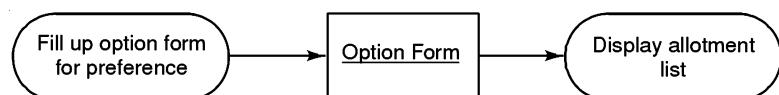


FIGURE 6.11 Activities in admission process.

6.4.2 Activity Diagrams: MCA Admission System

The complete activity diagram with the transitions is as given in Figure 6.12.

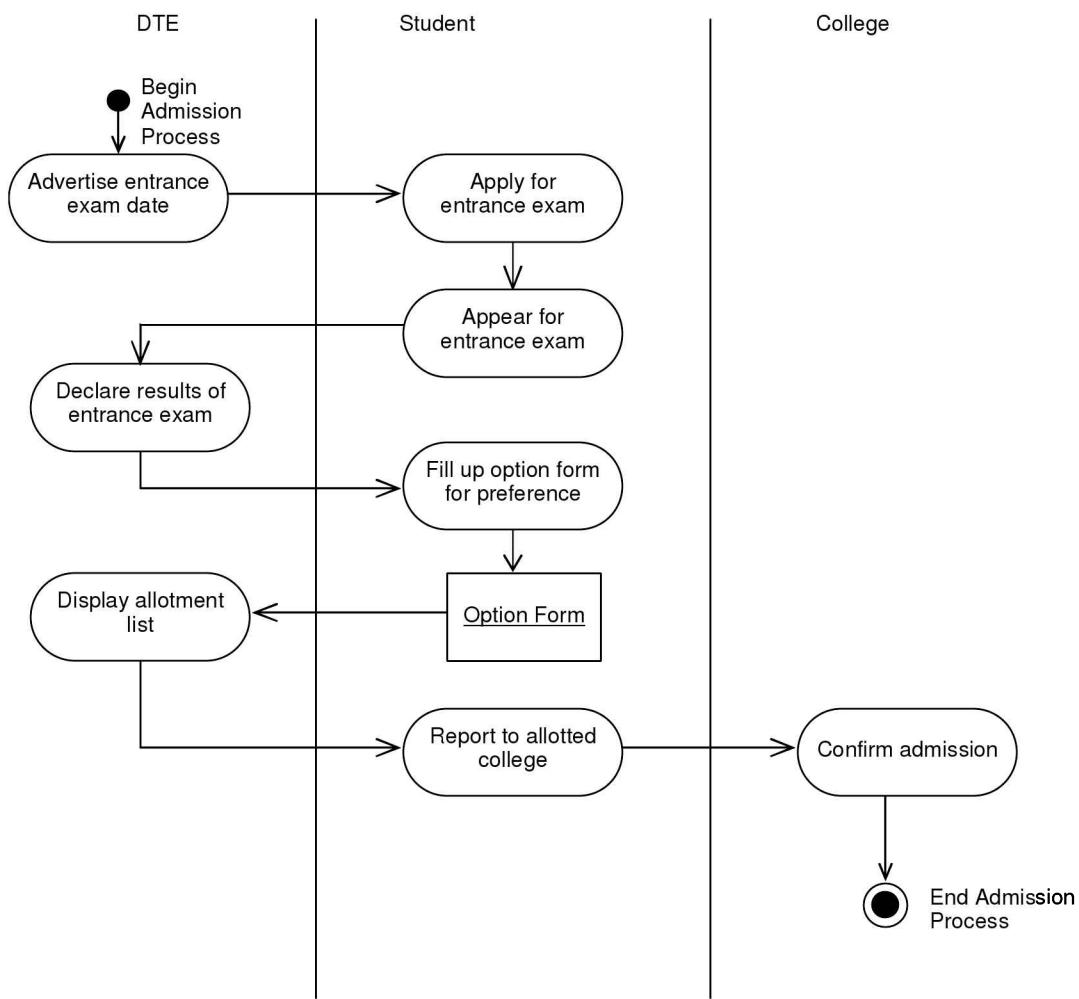


FIGURE 6.12 Complete activity diagram of admission process.

6.5 INTRODUCTION TO STATE CHART DIAGRAM

A state chart diagram describes the states of an object or system, as well as the transitions between states. It can also be referred to as a state diagram, state machine diagram, or a state-transition diagram.

State chart diagrams depict the dynamic behaviour of an object based on its response to events, showing how the object reacts to various events depending on the current state that it is in. A state chart diagram is drawn to explore the complex behaviour of a class, actor, subsystem, or component.

A state represents a stage in the behaviour pattern of an object, and like UML activity diagrams it is possible to have initial states and final states. An initial state, also called a

creation state, is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object. State diagrams are good at describing the behaviour of an object across several use-cases.

We should use state diagrams only for those classes that exhibit interesting behaviour, where building the state diagram helps understand what is going on. It is important to note that having a state diagram for your system is not a compulsion, but must be defined only on a need basis.

6.6 ELEMENTS OF STATE CHART DIAGRAM AND THEIR NOTATION

State chart diagrams have a very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicating the transition to the next state.

6.6.1 Initial State

The initial state is the first state of behaviour of an object after its creation. This shows the starting point or first activity of the flow. The initial state is denoted by a solid circle. This is also called a *pseudo state*, where the state has no variables describing it further and no activities as given in Figure 6.13.



FIGURE 6.13 Initial state.

6.6.2 Final State

The final state is the last state of behaviour of an object before it expires or is destroyed. It indicates the end of the state diagram as shown by a bull's-eye symbol. A final state is another example of a pseudo state, because it does not have any variable or action described as shown in Figure 6.14.



FIGURE 6.14 Final state.

6.6.3 State

A state represents a visible mode of behaviour of an object that persists for a period of time. Activities can run within a state. Transformations occur between states rather than within a state. For representing the condition of an object at an instant of time we use state, denoted

by a rectangle with rounded corners and compartments. Each state on a state chart diagram can contain multiple internal actions. An action is best described as a task that takes place within a state such as *On entry*, *On exit*, *Do*.

Types of states: The following are different types of states:

Simple state: A state that contains no *substates* [Figure 6.15(a)].

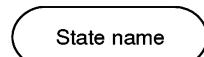


FIGURE 6.15(a) Simple state.

For example, a simple state diagram for a Door object with states *Opened*, *Closed* and *Locked* is shown in Figure 6.15(b). All the three states of the door are simple states without any substates.

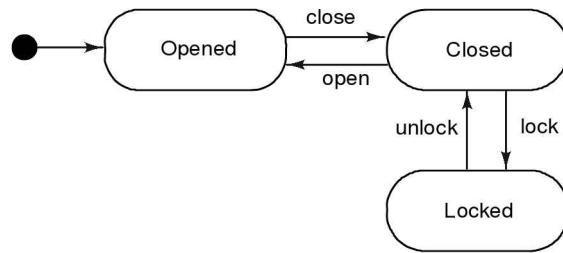


FIGURE 6.15(b) State diagram for a door showing simple states.

Super state or Composite state: A state contains *substates*. Sometimes, an object with complex behaviour can be modelled at different levels of abstraction (details). It is often difficult to model and analyze an object having many states using a single statechart diagram. In that case, we may draw a high level statechart diagram consisting of composite states or super states and other diagrams to further elaborate the internal states or substates inside individual composite states. For each super state or composite state, we can draw its nested states (internal states or substates) and their transitions between them.

States and transitions inside the super state are connected to the outside via the super state only. A super state is used when many transitions lead to a certain state. Transition arrows can be attached to the super state from the inside and the outside. By nesting a relatively autonomous part of a diagram in a super state, you can reduce the number of transitions and make the diagram easier to read.

A composite state or super state is composed of more than one sequential or concurrent substate and is called a *concurrent composite state* or a *sequential composite state* depending upon the kind of substate it has.

Concurrent composite state: The concurrent composite state is useful when a given object has sets of independent behaviours. However, there should not be too many concurrent

sets of behaviour occurring in a single object. If an object has several complicated concurrent statechart diagrams, the object should be split into separate objects. If a concurrent composite state is active, then one of the nested states from each concurrent state is also active. A state may be divided into regions containing substates that exist and execute concurrently by a dashed line [Figure 6.16(a)].

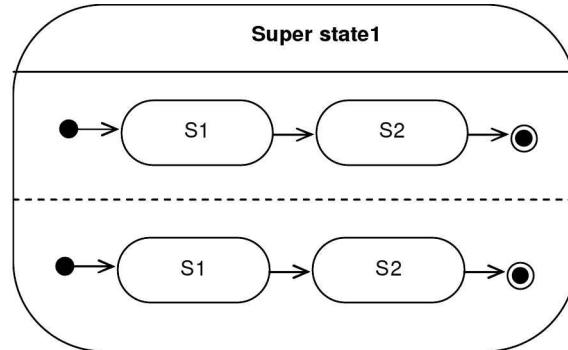


FIGURE 6.16(a) Representing super state concurrent composite state.

For example, Figure 6.16(b) shows a statechart diagram for a gas station where on arrival for filling gas, attendants can perform two tasks in parallel, filling gas as well as washing windshield. InService is the concurrent composite state having Filling Gas Tank and Washing Windshield substates in parallel.

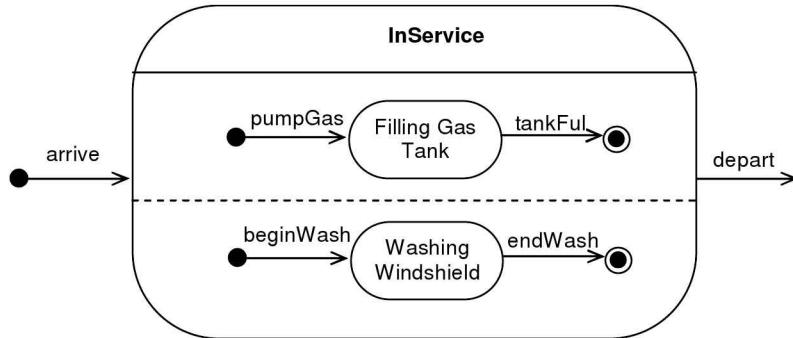


FIGURE 6.16(b) Example of concurrent composite state—InService.

Sequential composite state: If a sequential composite state is active, then exactly one of its substates is active [Figure 6.16(c)].

For example, Figure 6.16(d) shows the state chart diagram for car transmission. Transmission has three states: *Neutral*, *Reverse* and *Forward*. The forward state is a sequential composite state showing substates *First*, *Second*, *Third* and *Fourth*.

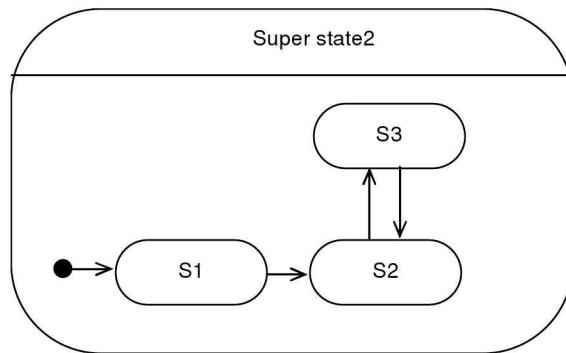


FIGURE 6.16(c) Representing superstate—sequential composite state.

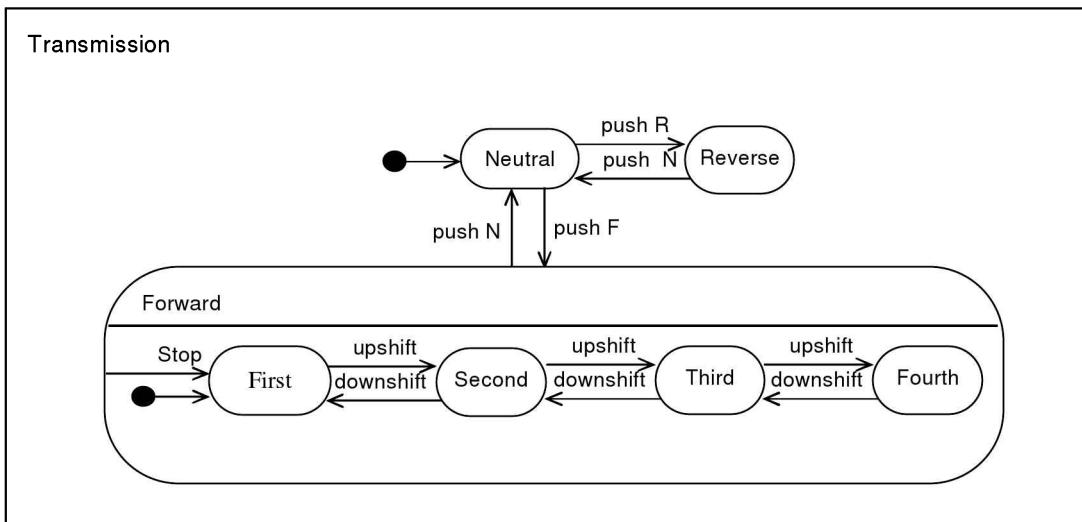


FIGURE 6.16(d) Example of sequential composite state transmission.

Substate: A state that is nested inside another state is called a substate. Substates allow state diagrams to show different levels of abstraction. Substates may be sequential or concurrent. Substates may be nested to any level. In Figure 6.16(d) *First*, *Second*, *Third*, *Fourth*, *Neutral* and *Reverse* are representing substates.

6.6.4 Transitions

A state transition is a relationship between two states that indicates when an object transit from one state to other state once certain conditions are met.

A transition to the same state is recursive transition which goes into the same state. An arrow (optional) indicates the transition of an object from one state to the other as given in Figure 6.17(a).



FIGURE 6.17(a) Representing transition.

A state can source many different transitions and can be the target of many transitions. Instead of going to a different state, a transition may have the same source and target state. Such transitions represent situations where a message is received but does not result in a change of state. These transitions are called *self-transitions* [Figure 6.17(b)].

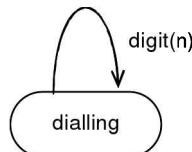


FIGURE 6.17(b) Representing self-transition.

Each transition has a label that has three parts: *event* [*guard*]/*activity*. All the three parts are optional. *Event* triggers a potential change of state, *guard* is a condition that must be true for the transition to take place and the *activity* is some behaviour that is executed during the transition. The actual trigger event and action causing the transition are written beside the arrow, separated by a slash. Transitions that occur because the state completed an activity are called *triggerless* transitions.

If an event has to occur after the completion of some event or action, the event or action is called the *guard condition*. The transition takes place after the guard condition occurs. This guard condition/event/action is depicted by square brackets around the description of the event/action as shown in Figure 6.17(c).

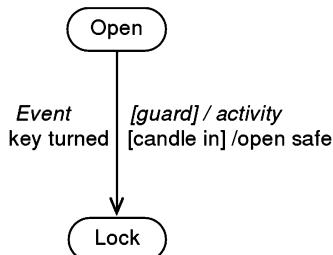


FIGURE 6.17(c) Transition with a guard condition.

6.7 GUIDELINES FOR DESIGN OF STATECHART DIAGRAM

The guidelines for the design of a state chart diagram are as follows:

1. A statechart diagram may be either one shot life cycle type or continuous loop type.
2. Objects for which one shot life-cycle statechart diagram is drawn have fixed life—from START state till END state, e.g. a chess game which has the START state presenting a new game opens till the END state presenting a win/loss of game.

3. Objects for which a continuous loop statechart diagram is drawn, do not have an end state.
4. A separate statechart diagram must be drawn for each object, showing various states of that object throughout its life cycle within the context of the system, e.g. a switch object in any electrical/electronic system can be in either ON or OFF state.
5. A statechart diagram is drawn for only those classes showing interesting or complex internal behaviour.
6. The class whose behaviour does not differ based on the state, the statechart diagram for those classes is trivial and of no much use.
7. While drawing a statechart diagram, first find out the initial state and final state. After this, the intermediate states during the life of an object are identified. The states of an object can be identified by looking at the boundary values of its attributes, e.g. in case of Railway reservation, when all the seats are reserved, reservation becomes full. *Full* is one of the valid states as various rules apply, when a person tries to make the reservation, he/she is put on a waiting list.
8. After identifying all the states, start looking for transitions. Transitions can be identified by asking the question at each state of the object as what makes that object to get out of that state. This also helps identify a new state of the object. Also check for the recursive transitions. A recursive transition is one that has the same state for both the starting and the end points.
9. Always place the initial state in the top-left corner and the final state in the bottom-right corner.
10. The state name should be simple and written in present tense.
11. If the state is very complex, it is better to exhibit its substates to resolve complexity.

6.8 PROBLEM STATEMENT: ADVERTISEMENT CAMPAIGN OF ABC PVT. LTD.

ABC Pvt. Ltd. company wants to start its advertisement campaign. The advertisement will be prepared. After the approval of the advertisement, the advertisement will be scheduled for publication. The advertisement will be published after scheduling is done.

6.8.1 Analysis of Advertisement Campaign of ABC Pvt. Ltd.

In this problem statement, the object for which a statechart diagram should be drawn is:

Advertisement campaign

Since the *Advertisement campaign* has fixed states from start till end, the statechart diagram will be of type one shot life cycle statechart diagram. Hence there will be one initial state and one or more final state.

Initial state: Figure 6.18(a) shows the initial state of the advertisement campaign process.



FIGURE 6.18(a) Begin advertisement campaign process.

Final state: Figure 6.18(b) shows the final state of the campaign process.



FIGURE 6.18(b) End advertisement campaign process.

Intermediate states: Figure 6.19(a) shows the intermediate states of the process.



FIGURE 6.19(a) Intermediate states—advertisement campaign.

Figure 6.19(b) shows all the transitions of the advertisement campaign process.

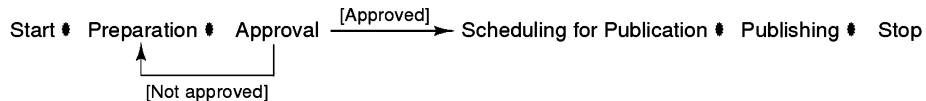


FIGURE 6.19(b) Transition—advertisement campaign: scheduling for publication and publishing.

- All the transitions in the above process are triggerless transitions which occur on completion of activity during the previous state.
- The guard condition to enter from *Approval* state into *Scheduling for Publication* state is Advertisement is approved/Not approved. No trigger is required to transit from the state.

6.8.2 State Chart Diagram for Advertisement Campaign of ABC Pvt. Ltd.

The complete state transition diagram for the *advertisement campaign* object is shown in Figure 6.20.

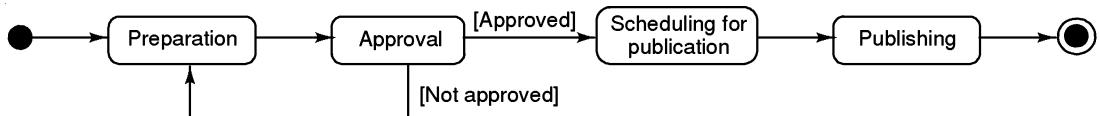


FIGURE 6.20 State chart diagram—advertisement campaign.

EXERCISES

1. What does a state-chart diagram represent?
2. What is the purpose of an activity diagram?
3. What are forks and joins in an activity diagram?
4. What is the difference between Action and Activity states?
5. Explain the super state in a statechart diagram with an example.
6. Propose a statechart diagram for a library book states (i.e., available, borrowed, to-be-returned, etc.).
7. Draw a statechart diagram capturing the state-transitions in a vending machine (i.e. waiting, inserting, choose-item, get-item, get-changes, etc.).
8. Draw a statechart diagram capturing the state-transitions in an ATM machine.
9. Draw a statechart diagram for a DVD player.
10. Develop an activity diagram based on the following narrative.

The purpose of the Open Access Insurance System is to provide automotive insurance to car owners. Initially, prospective customers fill out an insurance application, which provides information about the customer and his or her vehicles. This information is sent to an agent, who sends it to various insurance companies to get quotes for insurance. When the responses return, the agent then determines the best policy for the type and level of coverage desired and gives the customer a copy of the insurance policy proposal and quote.

11. Create an activity diagram based on the following narrative.

The purchasing department handles purchase requests from other departments in the company. People in the company who initiate the original purchase request are the *customers* of the purchasing department. A case worker within the purchasing department receives that request and monitors it until it is ordered and received. Case workers process the requests for purchasing products under \$1,500, write a purchase order, and then send it to the approved vendor. Purchase requests over \$1,500 must first be sent out for a bid from the vendor that supplies the product. When the bids return, the case worker selects one bid. Then, the case worker writes a purchase order and sends it to the approved vendor.

C H A P T E R Component Diagram and **7 Deployment Diagram**

7.1 INTRODUCTION TO COMPONENT DIAGRAM

A component diagram models the physical implementation of the software. Component diagrams are useful communication tools for larger teams. Their main purpose is to illustrate the structural relationships between the components of a system. Components are considered autonomous, encapsulated units within a system or subsystem.

The diagrams can be presented to key project stakeholders and implementation staff which give them an early understanding of the overall system that is being built. Component diagrams enable to model the high level software components and interfaces to those components which make it much easier to organize the development effort between subteams.

A component diagram is useful for the developers because it provides them with a high-level, architectural view of the system that they will be building, which helps them begin formalizing a roadmap for the implementation, and make decisions about task assignments and/or needed skill enhancements. The purpose of the component diagram is to define software modules and their relationships to one another.

System administrators also find component diagrams useful because they get an early view of the logical software components that will be running on their systems. Component diagrams can be used to show the dependencies among software components, including the implementation classes and the artifacts that implement them like source code files, binary code files, executables files, scripts and tables.

7.2 ELEMENTS OF COMPONENT DIAGRAM AND THEIR NOTATION

7.2.1 Components

A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Components are modelled graphically as rectangles with textual stereotype of <<component>> [Figure 7.1(a)]. Another notation for a component is a rectangle with a tabbed rectangle at the right-top corner [Figure 7.1(b)]. Components realize one or more interfaces and may have dependencies on other components.

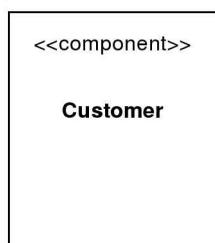


FIGURE 7.1(a) Component using <<component>> stereotype.

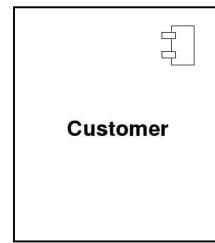


FIGURE 7.1(b) Component using tabbed rectangle symbol in right-top corner.

Components can be labelled with a stereotype such as <<document>>, <<library>>. Some standard stereotypes for component are as follows:

- <<application>>: A “front-end” of your system, such as the collection of HTML pages and ASP/JSPs that work with them for a browser-based system or the collection of screens and controller classes for a GUI-based system.
- <<database>>: A hierarchical, relational, object-relational, network, or object-oriented database.
- <<document>>: A document.
- <<executable>>: A software component that can be executed on a node.
- <<file>>: A data file.
- <<infrastructure>>: A technical component within your system such as a persistence service or an audit logger.
- <<library>>: An object or function library.
- <<source code>>: A source code file, such as a .java file or a .cpp file.
- <<table>>: A data table within a database.
- <<web service>>: One or more web services.
- <<XML DTD>>: An XML DTD.

Some examples of components are shown in Figure 7.1(c).

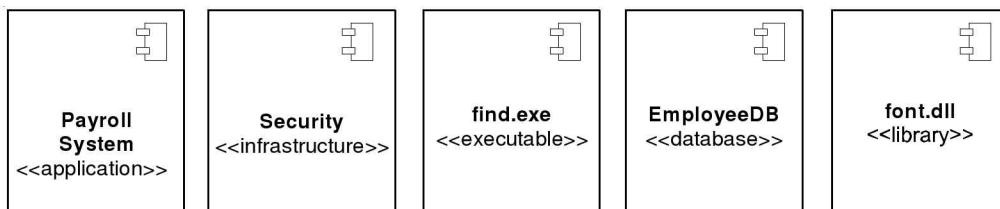


FIGURE 7.1(c) Examples of components.

Each component is a chunk of code that resides in memory on a piece of hardware. Each component must define an interface which allows other component to communicate with that component. The interface and the internal implementation of the component are encapsulated in the class that makes up the component.

The UML groups components into three broad categories:

- Deployment components, which are required to run the system.
- Work product components including models, source code, and data files used to create deployment components.
- Execution components which are components created while running the application.

Components may depend on one another. They may be dependent on classes. For example, to compile an executable file, you may need to supply the source code.

7.2.2 Interfaces

An interface is a collection of operations that are used to specify a service of a class or a component. The interface provides only the operations, but not the implementation. Method implementation is normally provided by a class or component. In complex systems, the physical implementation is provided by a group of classes rather than a single class. The relationship between a component and an interface is important. All the most common component-based operating system facilities (such as COM+, CORBA, and Enterprise Java Beans) use interfaces as the glue that binds components together.

The primary icon for an interface is just like a class except that it has the stereotype <>interface>> specified at the top of the rectangle [Figure 7.2(a)].

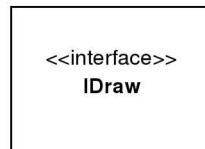


FIGURE 7.2(a) Representing interface using class icon.

Another shortcut notation for the interface is a lollipop notation [Figure 7.2(b)].

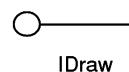


FIGURE 7.2(b) Representing interface using lollipop notation.

A component defines its behaviour in terms of provided and required interfaces.

Provided interface

A provided interface is an interface that declares the services that a class or component offers to provide to anonymous requestors. It is a relationship between an interface and a class or component that declares the class can be invoked to provide the services described in the interface.

A provided interface is modelled using a ball, labelled with the name, attached by a solid line to the classifier also called *lollipop notation* [Figure 7.2(c)].



FIGURE 7.2(c) Provided interface.

Required interface

A required interface is a relationship between an interface and a class or component that declares that the class or component requires the services described in the interface. The services must be made available by another class or component, usually as one of its provided interfaces.

A required interface is modelled using a socket, labelled with the name, attached by a solid line to the class or component [Figure 7.2(d)].



FIGURE 7.2(d) Required interface.

Required and provided interfaces are complementary. If one class declares a provided interface and the other class requires the same interface, the classes can interact over the interface with no additional structure. An example of an order component is shown in Figure 7.2(e).

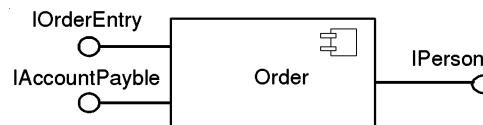


FIGURE 7.2(e) Order component.

Here the *order* component is shown with two provided interfaces *IOrderEntry*, *IAccountPayable* and one required interface *IPerson*. *IOrderEntry* and *IAccountPayable* are providing the services which are offered by the *Order* component, whereas the *Order* component is requiring the services described in *IPerson* interface.

When two components/classes provide and require the same interface, the two notations ball and socket, may be combined as shown in Figure 7.2(f).

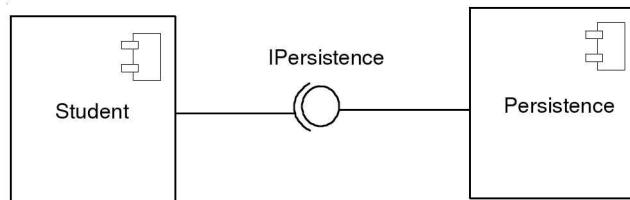


FIGURE 7.2(f) Component interface notation.

If an interface is shown using the rectangle symbol, we can use an alternative notation, using dependency arrows as shown in Figure 7.2(g).

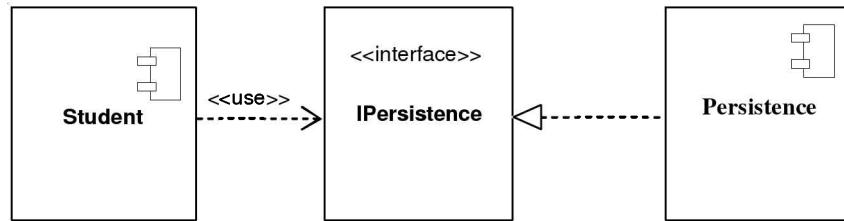


FIGURE 7.2(g) Explicit interface notation.

A component is connected to a provided interface by a dashed line with a triangular arrowhead. A component is connected to a required interface by a dashed line with an open arrowhead and the keyword <<use>>.

Here *Student* component requires the services described in *IPersistence* interface, whereas the interface *IPersistence* provides those services offered by the *Persistence* component to the *Student* component.

7.2.3 Dependencies

Dependency is a relationship between two elements in which a change to one element may affect or supply information by the other element. Components can be connected by usage dependencies. Usage dependency is relationship in which one element requires another element for its full implementation. It is a dependency in which the client requires the presence of the supplier. Dependency is shown in Figure 7.2(h) as dashed arrow with a <<use>> keyword. The arrowhead point from the dependent component (client) to the one of which it is dependent (supplier).

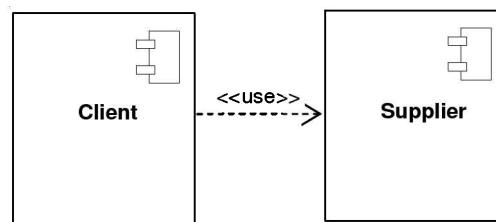


FIGURE 7.2(h) Representing dependency.

7.2.4 Port

A port is a connection point between a component and its environment. The port specifies a distinct interaction point between the component and its internal parts. If there are multiple interfaces associated with a port, these interfaces may be listed with the interface icon, separated

by commas. All interactions of a component with its environment are achieved through a port. The port isolates the internals of a component fully from the environment. This allows such a component to be used in any context that satisfies the constraints specified by its ports. Ports can support unidirectional communication or bidirectional communication.

The port is shown as a small square symbol superimposed over the boundary of a component rectangle [Figure 7.2(i)]. Ports can be named, and the name is placed near the square symbol.

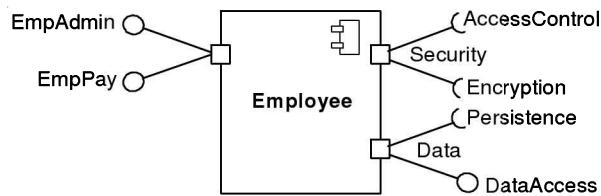


FIGURE 7.2(i) Representing port.

The *Employee* component implements three ports, two unidirectional ports and one bidirectional ports. The left-most port is an input port, the *Security* port is an output port, and the *Data* port is a bidirectional port.

7.3 GUIDELINES FOR DESIGN OF COMPONENT DIAGRAM

The following are the guidelines for the design of a component diagram:

1. The system for which a component diagram is drawn may be a subsystem or a part of a subsystem.
2. Based on the level of detail, different types of components in the system are identified, such as components representing functional elements of system, e.g. loan, components representing interface whose services the system is using, e.g. persistence interface, components representing interfaces with people, e.g. user interface.
3. Relationship among components can be represented with each other, to external systems, to system services, to people or to other component libraries.
4. Show the appropriate relationships among the components, among components and interfaces to show the dependency.
5. Avoid modelling data and user interface as components.
6. Only important interfaces should be shown.
7. To develop a component model, there are two approaches—top-down or bottom-up approaches.
 - The top-down approach is good for identifying software prototype early in the project which is important when a developer team is comprised of many subteams as everybody works towards the same goal. This approach has disadvantages of over-designing of the system.

- The bottom-up approach is better when we already have an existing collection of classes developed. Components are obtained from reusable functionality out of an existing application so that it can be easily distributed among the subteams.
- 8. A component should implement a single, related set of services.
- 9. Interface classes should be assigned to application components.
- 10. Technical classes should be considered as infrastructure components.
- 11. The classes which are collaborating frequently, they should be in the same domain component to reduce the network traffic between them.

7.4 PROBLEM STATEMENT: GLOBALTHERM USA PVT. LTD.

The GlobalTherm is the leading supplier of thermal reflective product in the USA for more than 25 years and provides creative solutions based on thermal reflective technology.

The company wanted to strengthen their business using online selling for their thermal reflective products. The requirement was taking online customized orders from the customers, accepting online payment, keeping their customers updated with the online information for orders until the order was delivered. The intention was to facilitate order tracking utility to the customers. The online web ordering system should facilitate customers to quickly and easily place orders directly to GlobalTherm. Upon receipt of the order, the order was logged, shipping booked and the customer emailed, almost immediately, with details of delivery day and time.

The web-based ordering systems had a number of benefits for GlobalTherm's customers like:

- (i) Increased efficiency: Because of web ordering customers can quickly and easily place orders via the GlobalTherm website.
- (ii) Less errors: With online ordering, it is far less likely that an incorrect order is shipped to a customer.
- (iii) Last minute order changes: Once an order is received and confirmed, a delivery date is supplied. If the customer realizes they need another item prior to that delivery date, they can simply call and have it added to the order.

7.4.1 Analysis of GlobalTherm USA Pvt. Ltd.

The system is all about receiving an order for products online and shipping those products. Depending on functionalities, the system is broadly divided into two major application components running on Apache Struts framework, namely online ordering and shipping.

The online ordering application component depends on components identified are Customer and Order, which are implementing corresponding interfaces ICustomer, IOrder.

Shipping application component depends on the delivery component implementing IDelivery interface. All these components need to implement IXML interface.

Persistence and security components represent functional components for storing data into database represented by the application DB component which is the data store component and managing the access control.

7.4.2 Component Diagram for GlobalTherm USA Pvt. Ltd.

Figure 7.3 shows the component diagram for GlobalTherm.

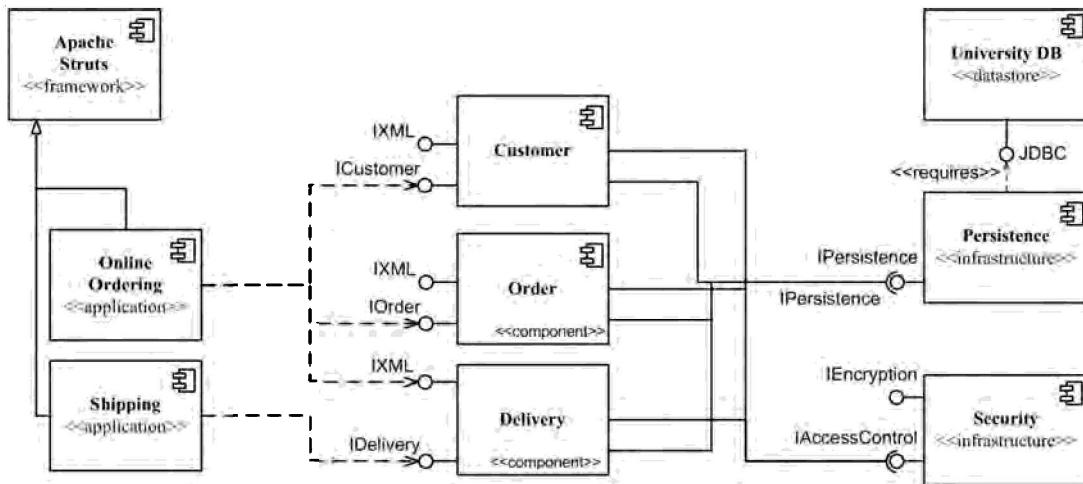


FIGURE 7.3 Component diagram—GlobalTherm USA Pvt. Ltd.

7.5 INTRODUCTION TO DEPLOYMENT DIAGRAM

A deployment diagram models the physical architecture of the hardware. It shows the hardware for the system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another. Deployment diagrams are used basically to depict the hardware/network infrastructure of an organization. Combining component and deployment diagrams, model the integration and distribution of your application software across the hardware implementation.

Deployment diagrams represent the physical relationships among software and hardware components as realized in a running system. A UML deployment diagram depicts a static view of the run-time configuration of hardware nodes and the software components that run on those nodes.

Deployment diagrams are used to explore the issues involved with installing the system into production. Deployment diagrams explore the dependencies that the system has with other systems that are currently in, or planned for, the production environment.

Deployment diagrams depict a major deployment configuration of a business application as shown in Figure 7.4. The purpose of the diagram is to design the hardware and software configuration of an embedded system. Deployment diagrams are useful for showing how components and objects move about in distributed systems. In deployment diagrams nodes represent computational elements (an intelligent device, processor, server, etc.).

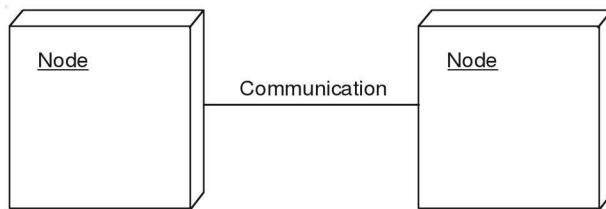


FIGURE 7.4 Example deployment diagram.

7.6 ELEMENTS OF DEPLOYMENT DIAGRAM AND THEIR NOTATION

7.6.1 Node

Nodes represent the hardware elements of the deployment system, anything that performs work in the system (Figure 7.5).

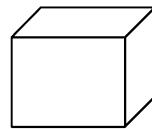


FIGURE 7.5 Node icon.

7.6.2 Component

Components define the requirements for software elements that are deployed to the hardware system. Refer to Figure 7.6.

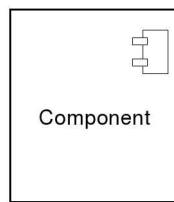


FIGURE 7.6 Component.

7.6.3 Artifacts

Artifacts represent a physical piece of information such as files, models, or tables, and the notation for it is as shown in Figure 7.7.

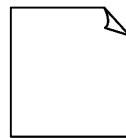


FIGURE 7.7 Sample artifact.

7.6.4 Link

Links are used to display communication relations between nodes using a straight line, (Figure 7.8).

—
FIGURE 7.8 Representing link.

7.6.5 Dependency

Dependency exists between components and can be specified by utilizing predefined or user-defined stereotypes using dotted arrow, as shown in Figure 7.9.

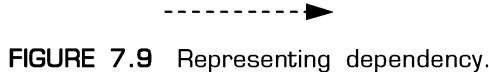


FIGURE 7.9 Representing dependency.

7.7 GUIDELINES FOR DESIGN OF DEPLOYMENT DIAGRAM

The following are the guidelines for the design of a deployment diagram.

1. A node represents a computational unit, typically a single piece of hardware, such as a computer, network router, mainframe, sensor.
2. Nodes can also be used to represent software artifacts such as file, framework, or domain component.
3. Nodes should be named with descriptive terms.
4. Appropriate relationships must be shown among the nodes.
5. While determining the deployment architecture of system, decide whether a single application or an application integrated with other application within the organization.
6. Technical issues like operating system, communication protocols, hardware and software with which the user directly interact must be considered.
7. If the system is to be deployed to an existing technical environment, one must decide about the distribution architecture strategy in advance. Also determine whether applications have two tiers, three tiers or more.

7.8 DEPLOYMENT DIAGRAM FOR WEB-BASED APPLICATION

7.8.1 Simplified Deployment Diagram

For any web-based application based on 3-tier architecture, a simplified deployment diagram is shown in Figure 7.10. Here, **Node A** represents the database server machine where there will be two things to consider: (1) any database server software, e.g. SQL Server, Oracle Server, MySQL etc. would be installed and (2) the database of your system, e.g. insurance database, product database would be deployed.

Node B represents the web server/application server machine where there will be two things to consider: (1) any web server/application server software, e.g. WebSphere, WebLogic, JBoss, Apache Geronimo, etc. would be installed and (2) business application consisting of all application files with a deployment descriptor file.

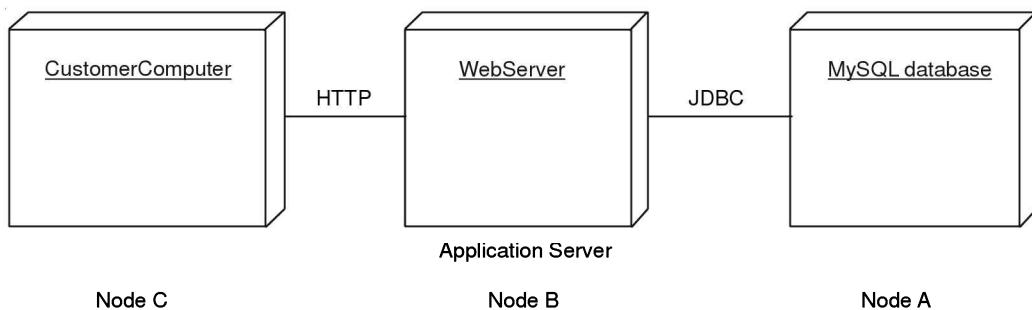


FIGURE 7.10 Basic deployment diagram.

Node C represents client machines from where application will be accessed through web URL. The client machine node only requires browser on it.

Communication between Node A and Node B occur through TCP/IP protocol, via JDBC, whereas communication between Node B and Node C occur through HTTP under which is lying TCP/IP.

7.8.2 Detailed Deployment Diagram

Figure 7.11 shows a more detailed deployment diagram representing the same web-based 3-tier application in the form of Java Servlets deployed on Web server Apache http server, MySQL database installed on database server and clients access the application through web browser through HTTP protocol.

7.9 PROBLEM STATEMENT: JOHN AND TOM'S ONLINE BEVERAGES

John and Tom's Online Beverages is an online beverages retailer. The beverages store is open to the public: anonymous users have limited access to the system, and users can make purchases

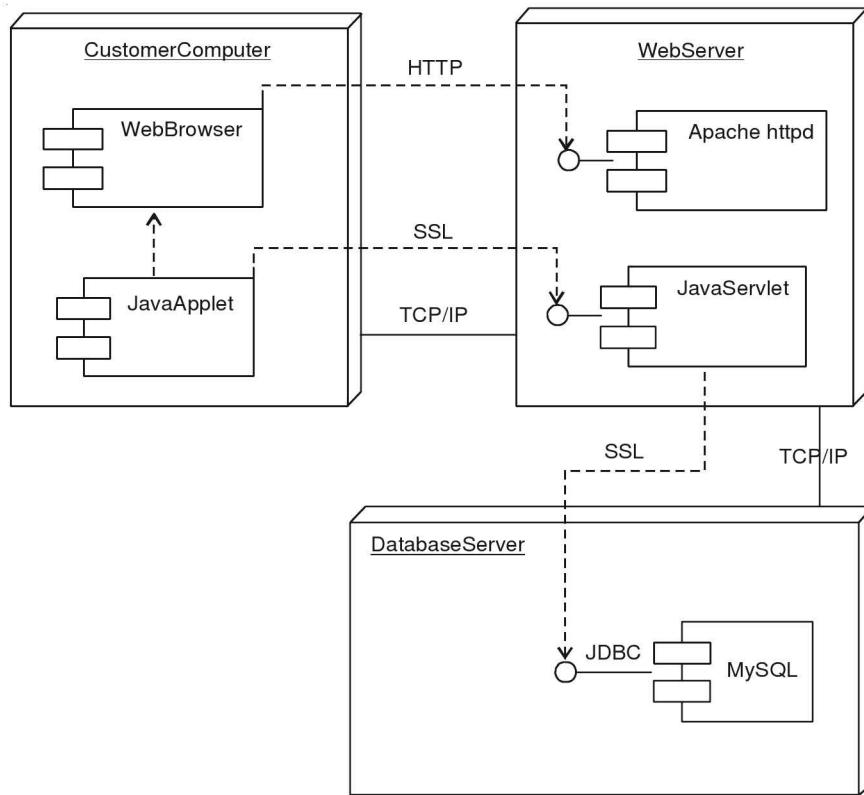


FIGURE 7.11 Detailed deployment diagram.

if they become members. The site should be attractive, simple, and usable. The store manages over 1000 beverages, stock information, and a database of around 1000 customers and their orders.

Any user with a web browser can access the site, browse or search for beverages that are in stock, and view the details. The details of beverages include the name, year of release, type and, in some cases, an expert review of the beverage. Anonymous users can add selected wines to a shopping cart. Users can also be members, and the membership application process collects details about the customer in the same way as at most online sites.

To purchase beverages, users must log in using their membership details. If a user has just joined as a member, he/she is logged in automatically. After selecting beverages for purchase, the user can place an order. An order is shipped immediately and a confirmation sent by email.

Behind the scenes, the system also allows the stock managers of the beverages store to add new shipments of beverages to the database. The web site manager can also add new beverages, beverage companies, and other information to the beverages store. Various types of reports should be available.

7.9.1 Analysis of John and Tom's Online Beverages

John and Tom's Online Beverages system is based on 3-tier architecture. Hence there will be database tier, application tier and client tier.

SERVER_DB will represent the database layer on which the database server—MS-SQL server software and Beverages database will be installed.

SERVER_APP will represent the application tier on which application server software—WebSphere application server and John and Tom's Online Beverages system (all source code pages, dynamic link libraries, etc.) will be deployed.

Clients will access the application through the browser. Hence on the client tier, only the browser is required.

7.9.2 Deployment Diagram for John and Tom's Online Beverages

The deployment diagram for this three-tier architecture will be as shown in Figure 7.12.

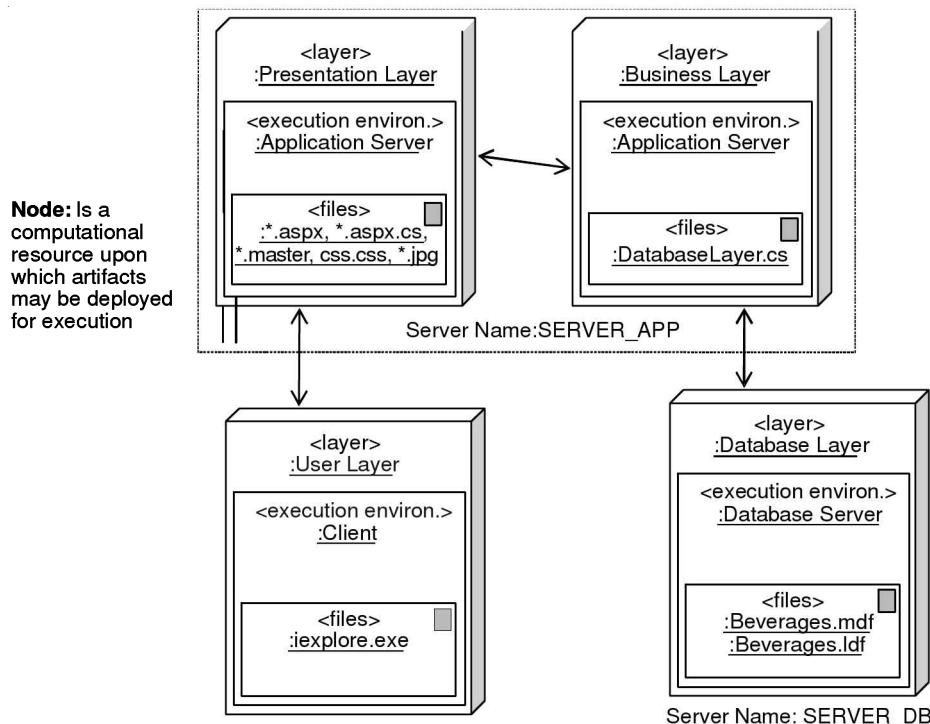


FIGURE 7.12 Deployment diagram for John and Tom's online beverages.

7.10 SIMPLIFIED DEPLOYMENT DIAGRAM FOR 2-TIER ARCHITECTURE

For any client-server application based on 2-tier architecture, a simplified deployment diagram will be as shown in Figure 7.13.

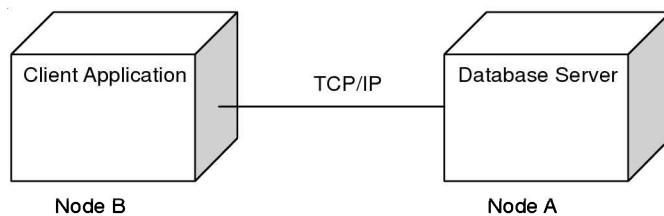


FIGURE 7.13 Basic deployment diagram—2-tier architecture.

Node A represents the database server on which the database server, e.g. Oracle/SQL server software and the database of application will be installed.

Node B represents the client application node where client application having business logic will be installed. Depending on the type of client, thick client or thin client, either business processing happen at the client side and only the database is installed on the database server, or only input-output is handled on the client node and business logic along with the database is installed on the server.

Communication between Node A and Node B occurs through TCP/IP protocol.

EXERCISES

1. What are components and interfaces?
2. Explain the terms *port* and *dependency*.
3. Explain provided and required interfaces.
4. What is the purpose of component and deployment diagrams?
5. What are the elements of a deployment diagram?
6. Draw a deployment diagram for a full-fledged website that the college wishes to host on the Internet.
7. Draw a deployment diagram online flight ticket reservation system.
8. Draw a component diagram payroll application processing management system.
9. Draw a component diagram for an application collecting students' feedback about courses. A student can read, insert, update and save data about the courses permanent. A professor can only see statistic elaboration of the data. The student application must be installed on the client PC sc1, sc2, etc. The manager application must be installed on the client PC in the manager's office. The database resides on the server.
10. Draw a deployment diagram for the case presented in Q9.

8.1 PROBLEM STATEMENT: STUDENT LOAN SYSTEM

A university gives loans to students. Before getting a loan, there is an evaluation process after which if the loan is approved, agreement is reached. A transaction records each step of the evaluation process, and another transaction records the overall loan agreement. A student can take any number of loans, but only one can be active at any time. Each loan is initiated by a separate transaction. Then, the student repays the loan with a series of repayments. Each repayment transaction is recorded. After the complete settlement, finally the loan account is closed.

Two output functions are desired: (1) an inquiry function that prints out the loan balance for any student, and (2) a repayment acknowledgement sent to each student after payment is received by the university.

The university loan office decides to implement the student loans on a single processor. Inquiries should be processed as soon as they are received. However, repayment acknowledgements need only be processed at the end of each day. For the above application, we need to build an Analysis Model, Design Model and Implementation Model with the following diagrams:

1. Business Process Diagram
2. Use-case Diagram
3. Class Diagram
4. Object Diagram
5. Sequence Diagram
6. Collaboration Diagram
7. Statechart Diagram
8. Activity Diagram
9. Component Diagram
10. Deployment Diagram

8.2 ANALYSIS PHASE DIAGRAMS: STUDENT LOAN SYSTEM

Notice that each student who receives one or more loans generates a stream of data over a long period of time, perhaps many years.

8.2.1 Business Process Diagram

Pools and lanes

Here, University and Student are the participants involved in all the activities. So, pools in this system are University and Student as shown in Figure 8.1.

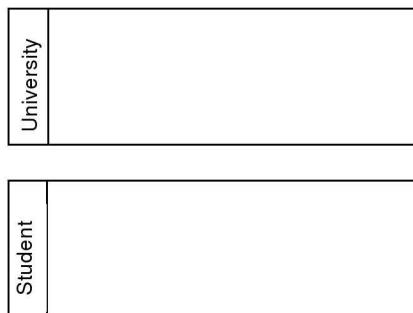


FIGURE 8.1 Pools for student loan system.

Lanes represent sub-partitions for the objects within a pool. Now within each pool, we try to find out if there is one or more organizational roles within a pool. Here in this case, both the pools, university and student, will not have any lane within it.

Activities: The work/task performed within the business process of a university pool is shown in Figure 8.2.

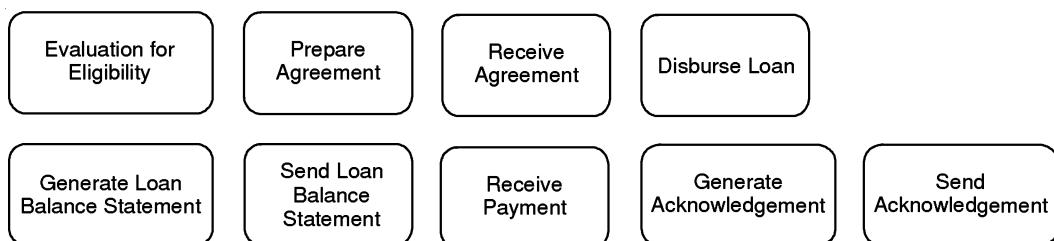


FIGURE 8.2 Activities/tasks for university pool.

Tasks performed within the student pool are as shown in Figure 8.3.

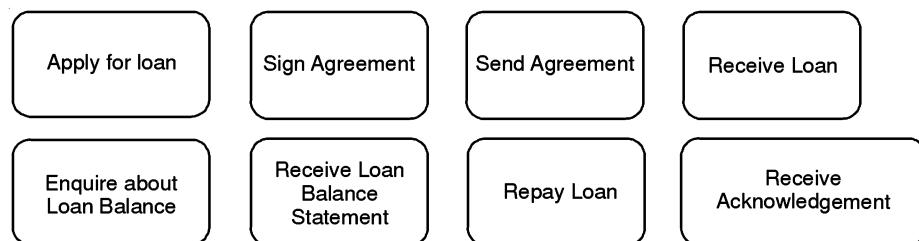


FIGURE 8.3 Activities/tasks within student pool.

Events: An event is something that “happens” during the course of a business process. There are only start and end events identified in this example as shown in Figure 8.4.

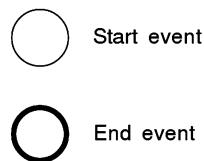


FIGURE 8.4 Events for student loan system.

Gateways: Gateways are modelling elements that are used to control the sequence flows interaction as they converge and diverge within a process. Only one gateway of type exclusive gateway will be included in the flow to decide the eligibility of student for loan as illustrated in Figure 8.5.

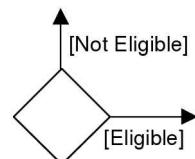


FIGURE 8.5 Gateway representing eligibility of student for loan process.

Artifacts: Artifacts display some additional information in the diagram such as input to activities, output from activities (Data objects), and grouping of related process objects (groups), textual comments about events, activities and gateways (comments). In this example, the only artifacts are data objects which are as shown in Figure 8.6.

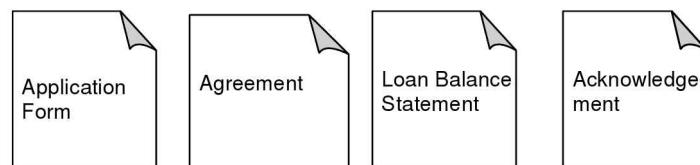


FIGURE 8.6 Data objects for student loan system.

Now a complete business process diagram for the student loan system will be drawn in two sub-diagrams as follows:

- Business process diagram for getting loan by the student (Figure 8.7)
- Business process diagram for repayment of loan by the student (Figure 8.8)

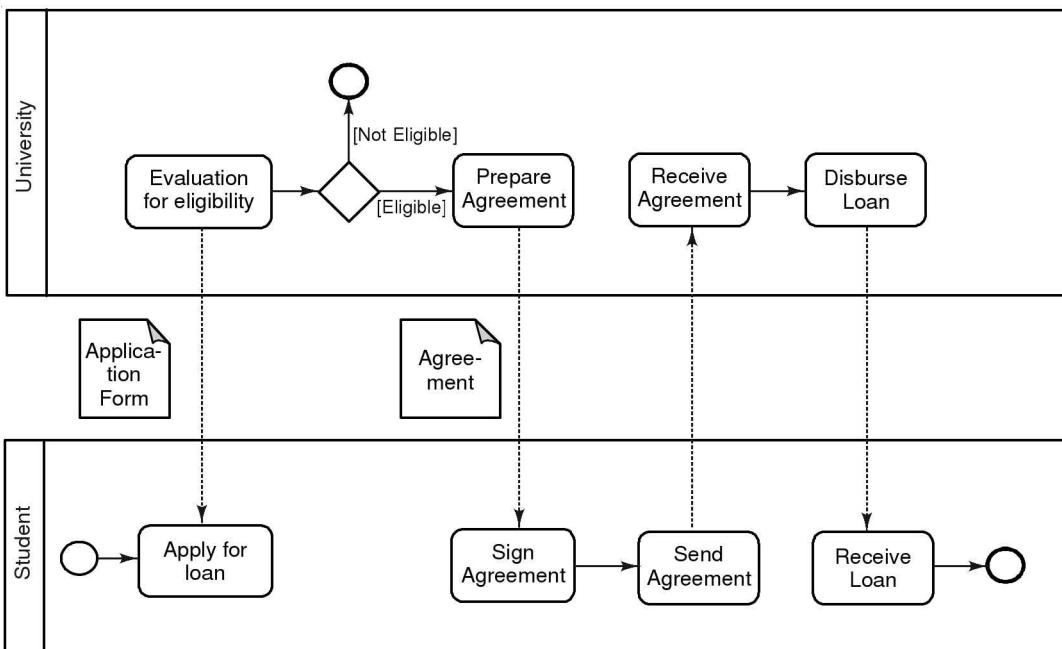


FIGURE 8.7 Business process diagram for getting loan by the student.

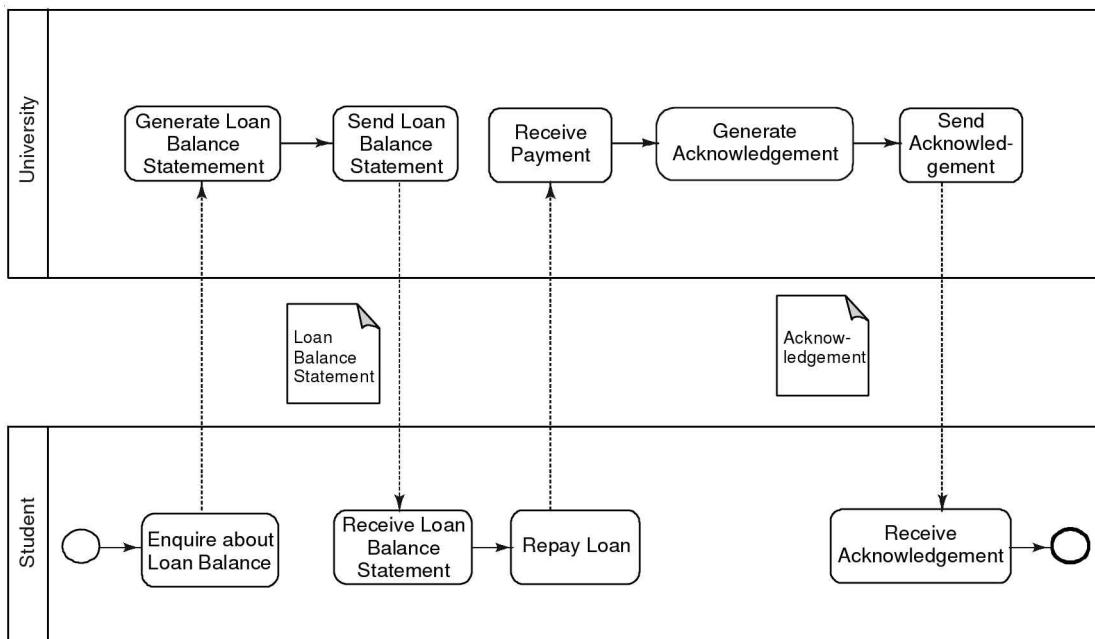


FIGURE 8.8 Business process diagram for repayment of loan by the student.

8.2.2 Use-Case Diagram

First, we will identify actors and their corresponding use-cases as listed in Table 8.1. There are two actors in the system namely

- Student (Person)
- Loan officer at university (Person)

TABLE 8.1 Actors and corresponding use-cases for student loan system

Actors	Use Cases		
	Base	Include	Extends
Student	Registration Login Loan enquiry Apply for loan Loan balance enquiry Repay loan		
Loan officer at University	Registration Login Sanction loan Prepare loan agreement Generate loan balance statement Generate payment acknowledgement		

Now let us try and draw actor-wise use-case diagrams. Finally, a combined use-case diagram representing the whole system with all actors and use-cases performed by them will be drawn.

Use-case diagram for student: Figure 8.9 shows the use-case diagram for the student.

Use-case diagram for loan officer at university: Figure 8.10 shows the use-case diagram for the loan officer at University.

Complete use-case diagram representing the complete student loan system: Figure 8.11 depicts the complete use-case diagram representing the complete student loan system.

Note: In the above use-case diagram, apart from use-cases obvious from the problem statement as found out in the table, we have added two additional use cases—Registration and Login which are not mentioned in the problem statement, but we have to consider it as everybody has a different role in the system.

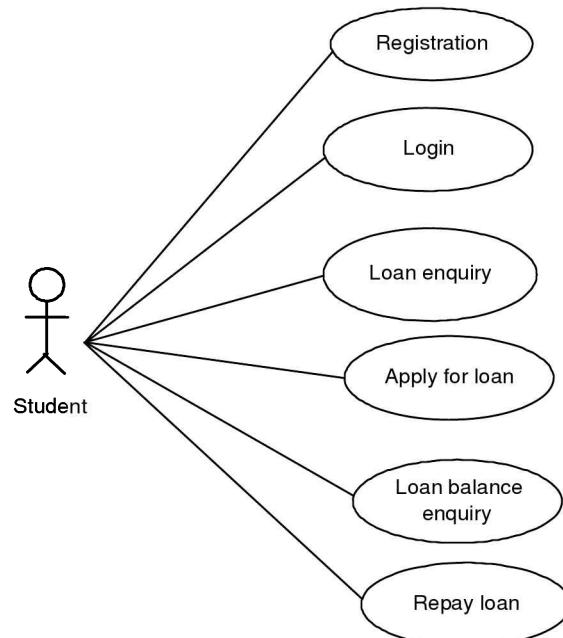


FIGURE 8.9 Use-case diagram for the student.

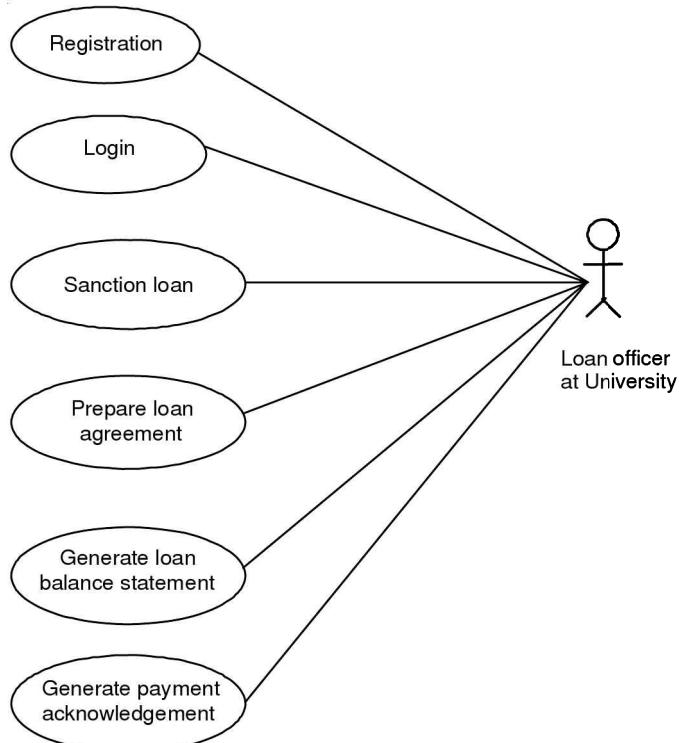


FIGURE 8.10 Use-case diagram for the loan officer at university.

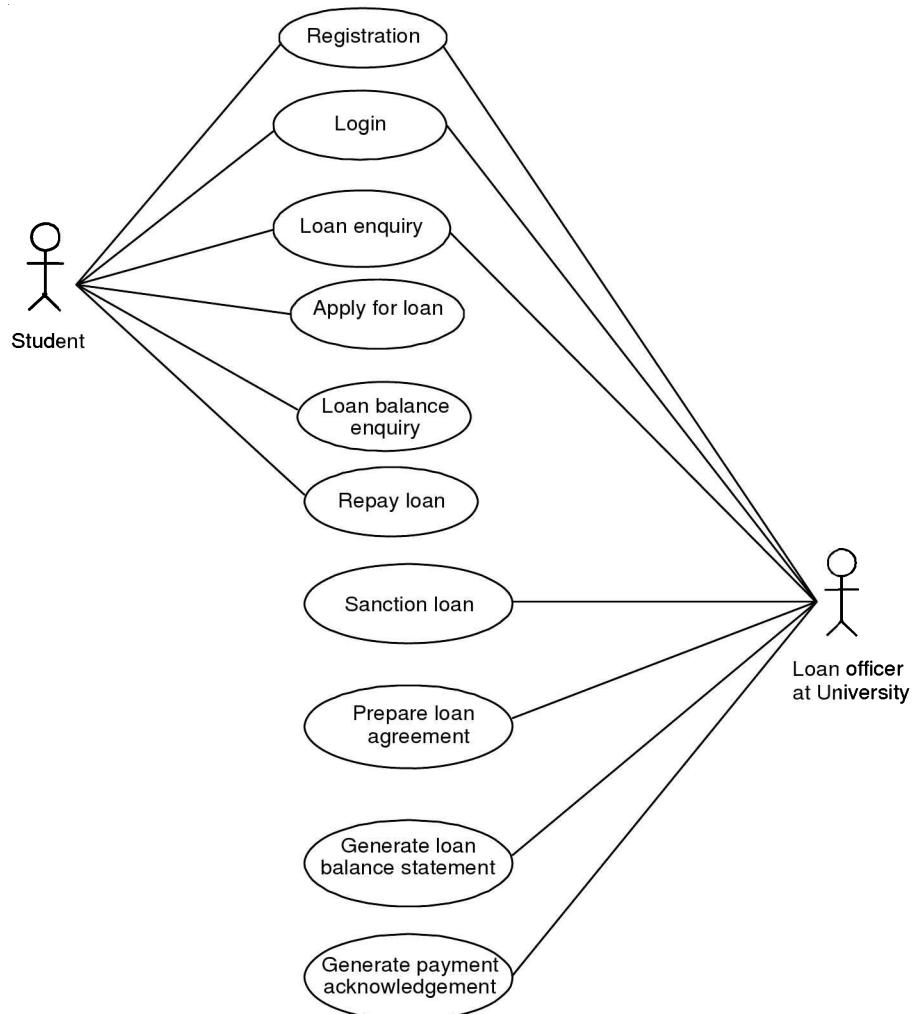


FIGURE 8.11 Use-case diagram for complete student loan system.

8.2.3 Class Diagram

After analyzing the case, we interpret the following:

1. University gives loans to students.
2. University generates loan balance statement.
3. University generates loan repayment acknowledgement.

The classes, their relationships and relationship types are given in Table 8.2. Figure 8.12 and 8.13 show the class and the object diagrams.

TABLE 8.2 Classes and their relationships for student loan system

Sr. No.	Class 1 Name	Relationship Name	Relationship Type	Class 2
1	University (Loan Office)	gives	Association	loans
2	loans	given to	Association	student
3	University (Loan Office)	generates	Association	Loan Repayment Acknowledgement
4	University (Loan Office)	generates	Association	Loan balance statement

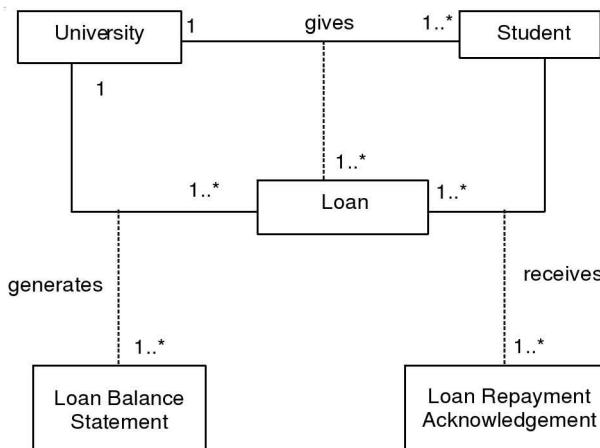


FIGURE 8.12 Class diagram for student loan system.

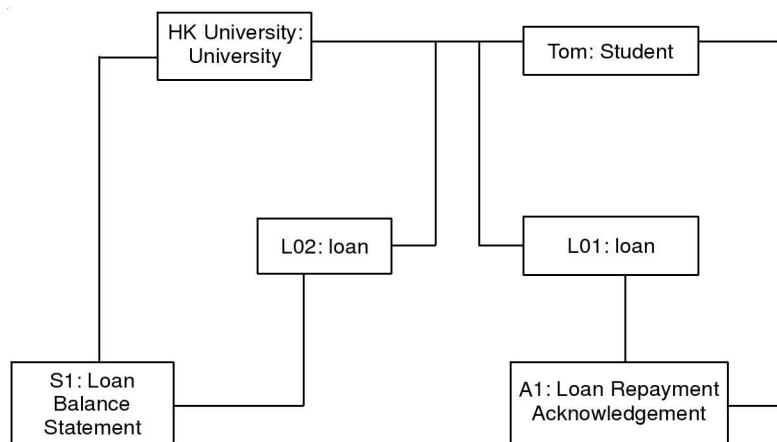


FIGURE 8.13 Object diagram for student loan system.

8.2.4 Object Diagram

Figure 8.13 shows a object diagram for the student loan system.

8.3 DESIGN PHASE DIAGRAMS: STUDENT LOAN SYSTEM

8.3.1 Sequence Diagram

A sequence diagram needs to be drawn during the design phase of the system.

During analysis, we have got:

- Use-case diagram
- Class diagram

We will make use of both these diagrams for design.

1. For each use-case in the use-case diagram, draw one sequence diagram at least. If use-case is much complex, then there can be more than one sequence diagram.
2. While drawing a sequence diagram from the use-case diagram, draw the sequence diagram only for the base use-cases embedding the flow for <<include>> and <<extend>> use-cases in the same.

In the above case, actors and use-cases identified can be summarized as in Table 8.3.

TABLE 8.3 Actor and their use-cases for student loan system

<i>Actors</i>	<i>Use-cases</i>
Student	Registration Login Loan enquiry Apply for loan Loan balance enquiry Repay loan
Loan officer at University	Registration Login Sanction loan Prepare loan aggrement Generate loan balance statement Generate payment acknowledgement

During design, identify four types of objects interacting with each other to perform the use-case as:

1. **Actor objects:** Person, External system or device that initiates the use-case, e.g. Student, Loan Officer at University.

2. **Boundary objects:** All the interfaces through which the actor interacts with the system, e.g. Login Screen, Loan Application Form, Balance Enquiry Form, etc.
3. **Controller objects:** All the objects which coordinate the task, e.g. Loan Controller, etc.
4. **Entity objects:** All domain entities identified during the analysis class diagram, e.g. User, Loan, Repayment etc.

Sequence diagram for registration use-case

The use-case *Registration* is performed by both actors, student and loan officer at university for registering as authorized users of the student loan system.

For registration use-case, ask four questions and answers to those questions will be the objects interacting with each other for registration process.

1. Who will register? (Actor)
Student/Loan Officer at University.
2. Through which web page (interface)? (Boundary)
Registration Form.
3. Who will add the user? (Control)
Registration Controller.
4. Which object holds valid user details? (Entity)
Users.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls. Table 8.4 shows the classes, their types and methods for registration use-case.

TABLE 8.4 Classes, their types and methods for registration use-case

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Student/Loan Officer	<i>Not required to specify in this context.</i>
Boundary	RegistrationForm	fillForm()
Control	RegistrationController	receiveDetails() validateDetails()
Entity	User	addUser()

For online registration, the sequence flow will be as follows:

- Step 1.* Student/Loan Officer at university will open the Registration Form.
- Step 2.* Fill up the registration details on the form.(Name, Address, DOB, etc.)
- Step 3.* Click on the Submit button to submit the details.
- Step 4.* Before submitting the details to the server, validation for all the fields on the form (Valid e-mail, phone no., etc.) will be carried out.
- Step 5.* After form validation, the registration controller object will validate the user, i.e. it will check if the user with the same details already exists or not or any other validation.

- Step 6.* After user validation, if the user is valid, it will be added to the database in the user table and registration confirmation message will be sent to the user.
- Step 7.* After user validation, if the user is not valid, the user will not be added to the database and registration cancellation message will be sent to the user.

For steps 6 and 7, the combined fragment is used, where there are two alternatives—Valid User and Invalid User. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used. The sequence diagram for the registration use-case is as shown in Figure 8.14.

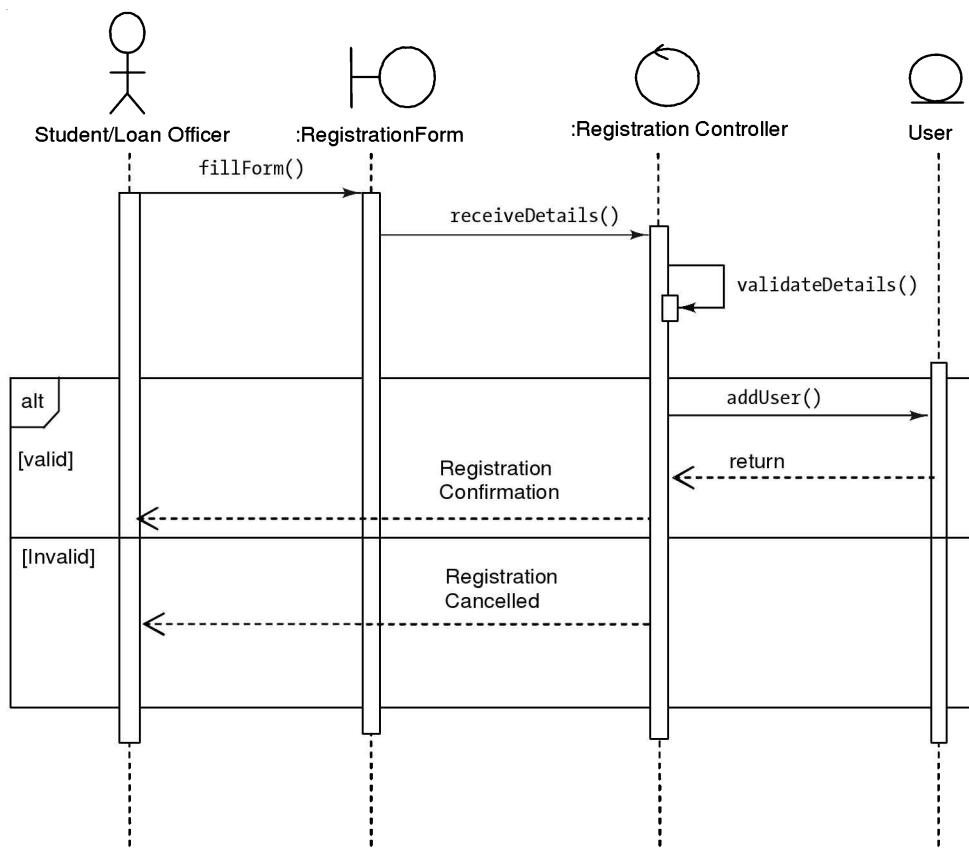


FIGURE 8.14 Sequence diagram for registration use-case.

Sequence diagram for login use-case

The use-case *login* is performed by both actors Student and Loan Officer at university for an authorized access to the student loan system. For the login use-case, ask four questions and answers to those questions will be the objects interacting with each other for login process.

1. Who will login? (Actor)
Student/ Loan Officer at university.

2. Through which web page (interface)? (Boundary)
Login Form.
3. Who will validate the user? (Control)
Login Controller.
4. Which object holds valid user details? (Entity)
Users.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls and for the login use-case the sequence diagram is as shown in Figure 8.15. The classes, their types and methods are explained in Table 8.5.

TABLE 8.5 Classes their types and methods for login use-case

Class Type	Classes	Methods
Actor	Student/ Loan Officer	<i>Not required to specify in this context</i>
Boundary	LoginForm	setUserDtls()
Control	LoginController	getUserDtls() validateUser()
Entity	User	retrieveUserDetails

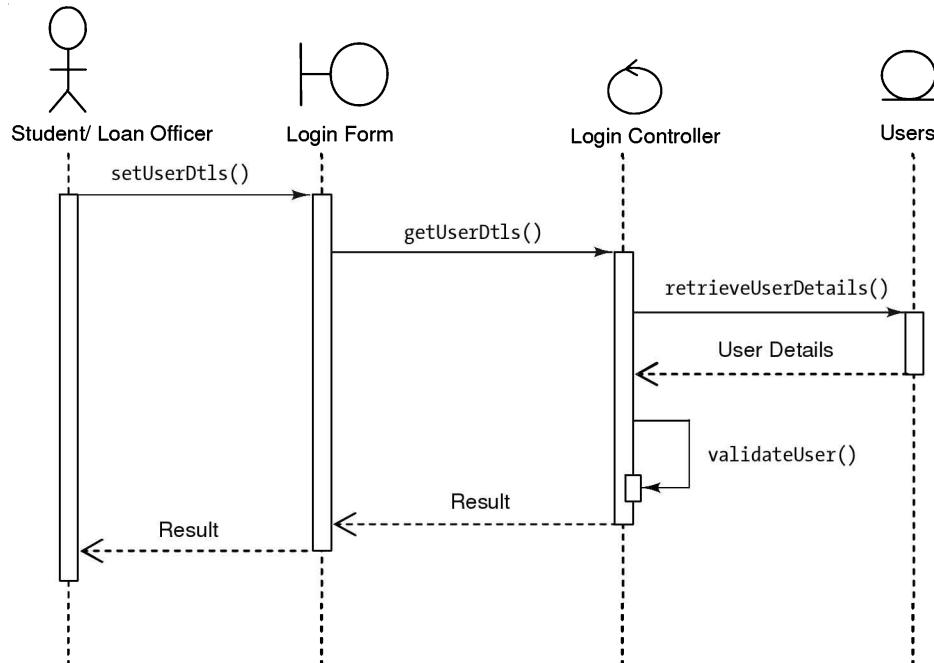


FIGURE 8.15 Sequence diagram for login use-case.

Sequence diagram for loan enquiry use-case

The use-case *Loan enquiry* is performed by Student for enquiring about education loan in the university through the student loan system. For the loan enquiry use-case, ask four questions and answers to those questions will be the objects interacting with each other for performing the process.

1. Who will enquire? (Actor)
Student.
2. Through which web page (interface)? (Boundary)
EnquiryForm.
3. Who will handle the enquiry? (Control)
EnquiryController, Loan Officer.
4. Which object holds valid user details? (Entity)
EnquiryDB.

After identifying the interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls. For the loan enquiry use-case the classes, their types and related methods are elaborated in Table 8.6 and the sequence diagram is as shown in Figure 8.16.

TABLE 8.6 Classes, their types and methods for loan enquiry use-case

Class Type	Classes	Methods
Actor	Student	<i>Not required to specify in this context.</i>
Boundary	EnquiryForm	fillForm()
Control	EnquiryController	receiveEnquiry(), getEnquiry(), giveResponse()
Entity	EnquiryDB	addEnquiry(), retrieveEnquiry()
Actor	Loan Officer	—
Boundary	EnquiryRespForm	SelectEnquiry()

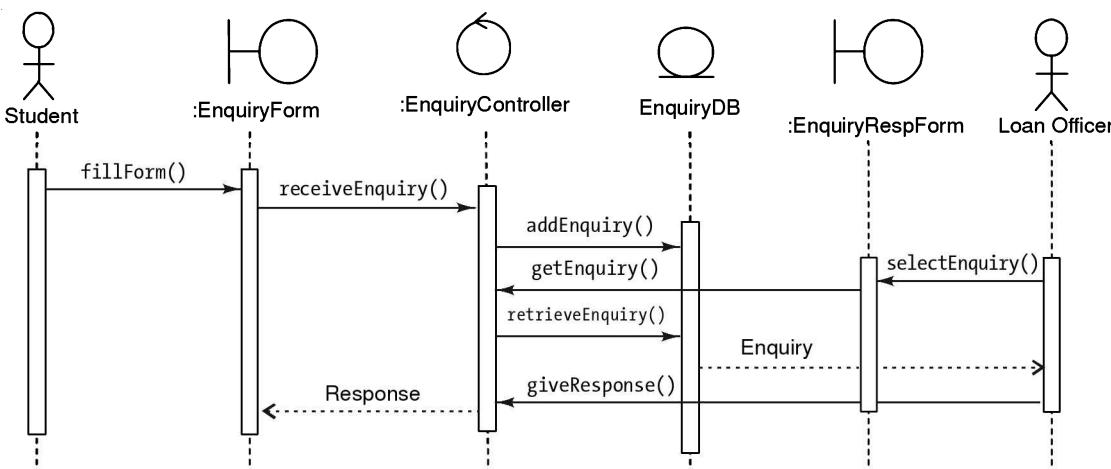


FIGURE 8.16 Sequence diagram for loan enquiry use-case.

For online registration, the sequence flow will be as follows:

1. Student/Loan Officer at university will open the Enquiry Form and fill it up.
2. After submitting the enquiry form, the EnquiryController will add the enquiry to the database in the EnquiryDB table.
3. The Loan Officer will retrieve the enquiries from the EnquiryDB through the EnquiryController and send response to the Student immediately on the same day.

Sequence diagram for apply for loan use-case

The use-case *apply for loan* is performed by the Student actor in order to apply for educational loan through university's student loan system.

For the apply for loan use-case, ask four questions and answers to those questions will be the objects interacting with each other for performing the process.

1. Who will apply for loan? (Actor)
Student at university.
2. Through which web page (interface)? (Boundary)
ApplicationForm.
3. Who will handle the Loan Application? (Control)
ApplicationController, Loan Oficer.
4. Which object holds valid Applications and Loan Account information? (Entity)
ApplicationDB, LoanDB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls. All these are detailed in Table 8.7. The corresponding sequence diagram is given in Figure 8.17.

TABLE 8.7 Classes their types and methods for apply for loan use-case

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Student	<i>Not required to specify in this context.</i>
Boundary	ApplicationForm	fillForm()
Control	ApplicationController	receiveAppl() validateForm() getAppl()
Entity	ApplicationDB	addLoanApplication(),retrieveAppl()
Actor	Loan Officer	verifyAppl()
Boundary	VerifyApplForm	selectApplication()
Entity	LoanDB	addLoan()

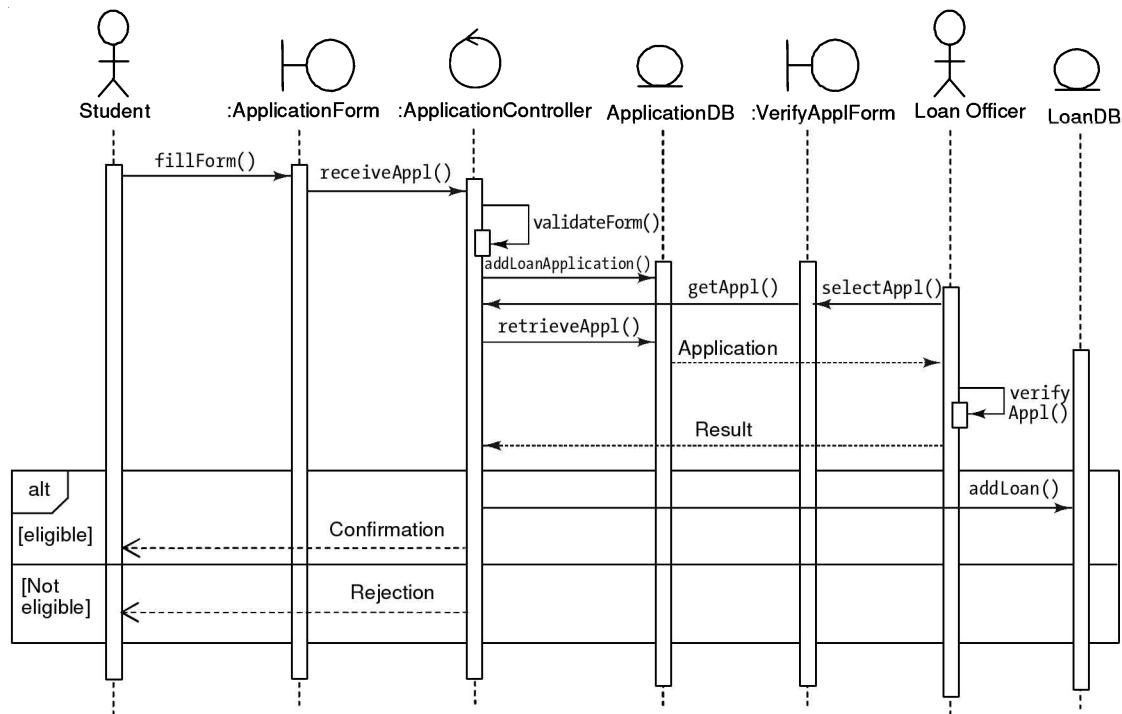


FIGURE 8.17 Sequence diagram for Apply for Loan use-case.

For the apply for loan use-case, the sequence flow will be as follows:

- Step 1. Student at university will open the Loan Application Form and fill up all the necessary information on the form and submit the application.
- Step 2. Validation for all the fields on the form (Valid e-mail, phone no., etc.) will be carried out.
- Step 3. After form validation, valid Applications will be added to the ApplicationDB table.
- Step 4. Loan Officer will then retrieve the Applications from the ApplicationDB table through ApplicationController for the verification purpose.
- Step 5. After verification of all the documents and Application, if the student is eligible, a new Loan account will be created and confirmation will be sent to the student by the Loan Officer.
- Step 6. After verification, if the student is not eligible, rejection will be sent to him by the Loan Officer.

For steps 5 and 6, the combined fragment is used, where there are two alternatives—eligible and not eligible student. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used.

Sequence diagram for loan balance enquiry use-case

The use-case *loan balance enquiry* is performed by Student at university for enquiring the loan balance amount through the student loan system.

For the loan balance enquiry use-case, ask four questions and answers to those questions will be the objects interacting with each other for performing the process.

1. Who will enquire for loan balance? (Actor)
Student/Loan Officer at University.
2. Through which web page (interface)? (Boundary)
BalanceEnqForm.
3. Who will handle the loan balance enquiry? (Control)
BalanceEnqController.
4. Which object holds loan information? (Entity)
LoanDB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls as explained in Table 8.8.

TABLE 8.8 Classes their types and methods for loan balance use-case

Class Type	Classes	Methods
Actor	Student	<i>Not required to specify in this context.</i>
Boundary	BalanceEnqForm	setLoanAcNo()
Control	BalanceEnqController	getLoanAcNo(), validateLoan()
Entity	LoanDB	retrieveLoanBal()

For the loan balance enquiry, the sequence flow will be as follows:

- Step 1.* Student at university will open the BalanceEnqForm and specify the loan account details and submit it.
- Step 2.* After submitting the balance enquiry form, the BalanceEnqController will retrieve loan details for the specified loan account no and validate the loan.
- Step 3.* After loan validation, if the loan account is valid, the loan balance will be retrieved from the Loan DB and the outstanding loan amount will be sent to the student.
- Step 4.* After loan validation, if the loan account is not valid, the message “Invalid Loan Account” will be sent to the student.

For steps 3, 4, the combined fragment is used, where there are two alternatives—valid LoanAc and invalid LoanAc. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 8.18.

Sequence diagram for repayment of loan use-case

The use-case *repayment of loan* is performed by Student at university for repaying the loan amount on a regular basis through the Student loan system.

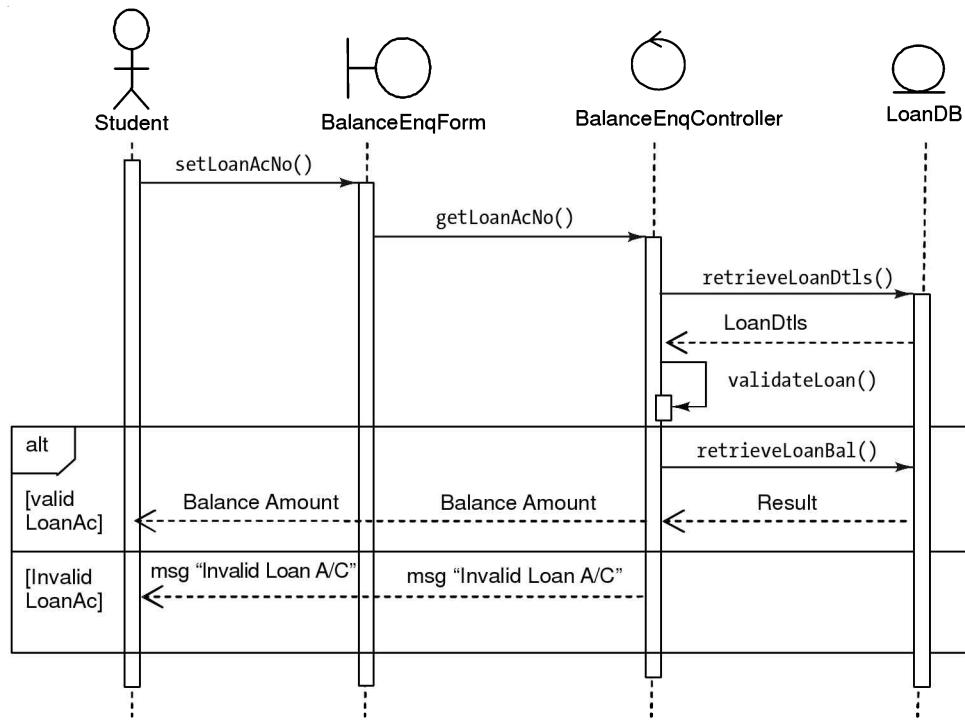


FIGURE 8.18 Sequence diagram for loan balance enquiry use-case.

For repayment of loan use-case, ask four questions and answers to those questions will be the objects interacting with each other for performing the process.

1. Who will repay the loan amount? (Actor)
Student at University
2. Through which web page (interface)? (Boundary)
Payment Form.
3. Who will handle the repayment process? (Control)
Payment Controller.
4. Which object holds the payment transaction data? (Entity)
Repayment DB.

After identifying interacting objects, it is required to find out what sequence of message communication happen among them and message communication occurs through method calls illustrated in Table 8.9.

TABLE 8.9 Classes, their types and methods for repayment of loan use-case

Class Type	Classes	Methods
Actor	Student	<i>Not required to specify in this context.</i>
Boundary	PaymentForm	setPaymentDtls()
Control	PaymentController	receivePaymentDtls() validatePaymentDtls()
Entity	RepaymentDB	addPayment()

For online Repayment of loan, the sequence flow will be as follows:

- Step 1. Student at University will open the PaymentForm and fill up the payment details on the form.(Loan AcNo., DD/Cheque No., Amount, etc.) and Submit form.
- Step 2. After submitting the form, the PaymentController object will validate the payment details (Loan AcNo, DD/Cheque no., etc.).
- Step 3. After payment validation, if the Payment Details are valid, it will be added to the database in the Repayment DB table and Payment confirmation message will be sent to the Student.
- Step 4. After payment validation, if the Payment Details are not valid, the Payment rejection message will be sent to the Student.

For steps 3 and 4, the combined fragment is used, where there are two alternatives—Valid PayDtls and Invalid PayDtls. To demonstrate the sequence flow for both alternatives, in the same sequence diagram the combined fragment is used as shown in Figure 8.19.

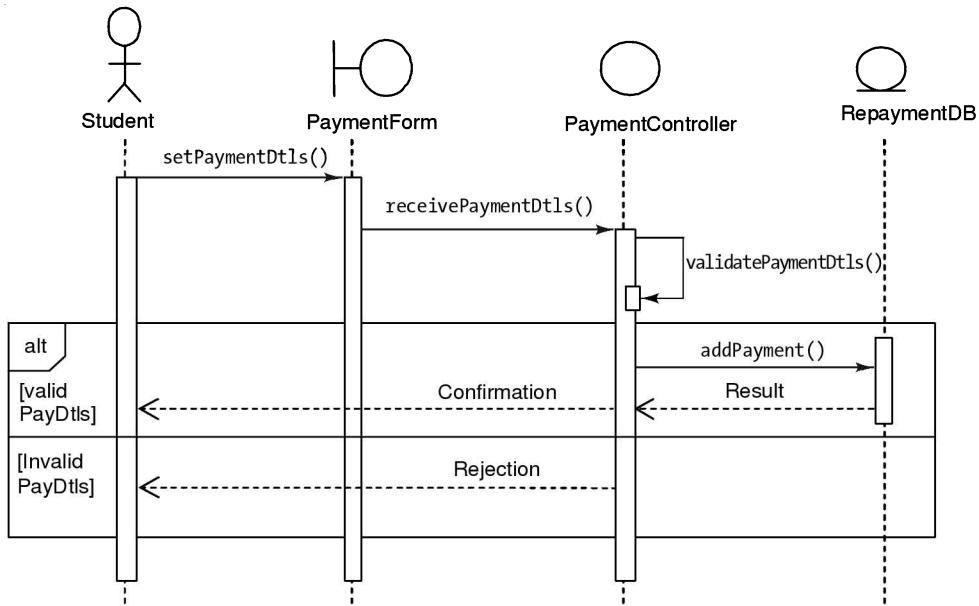


FIGURE 8.19 Sequence diagram for the repayment of loan use-case.

Sequence diagram for sanction loan and prepare loan agreement use-case

The use-case *sanction loan and prepare loan agreement* is performed by Loan Officer at university for sanctioning the loan for the student and preparing loan agreement for it using the student loan system.

For both use-cases, ask four questions and answers to those questions will be the objects interacting with each other for performing the process.

1. Who will sanction the loan and prepare loan agreement? (Actor)
Loan Officer at University
2. Through which web page (interface)? (Boundary)
LoanSanctionForm

3. Who will generate loan agreement? (Control)
SanctionController.
4. Which object holds valid loan data? (Entity)
LoanDB

After identifying interacting objects, it is required to find out what sequence of message communication happen among them. Message communication occurs through method calls as illustrated in Table 8.10.

TABLE 8.10 Classes, their types and methods for sanction loan and prepare loan agreement use-case.

Class Type	Classes	Methods
Actor	Loan Officer	<i>Not required to specify in this context.</i>
Boundary	LoanSanctionForm	selectLoanAppl()
Control	SanctionController	getLoanAppl() verifyAppl() generateLoanAgmt()
Entity	LoanDB	addLoan()

The sequence diagram for the sanction loan and prepare loan agreement use-case is shown in Figure 8.20.

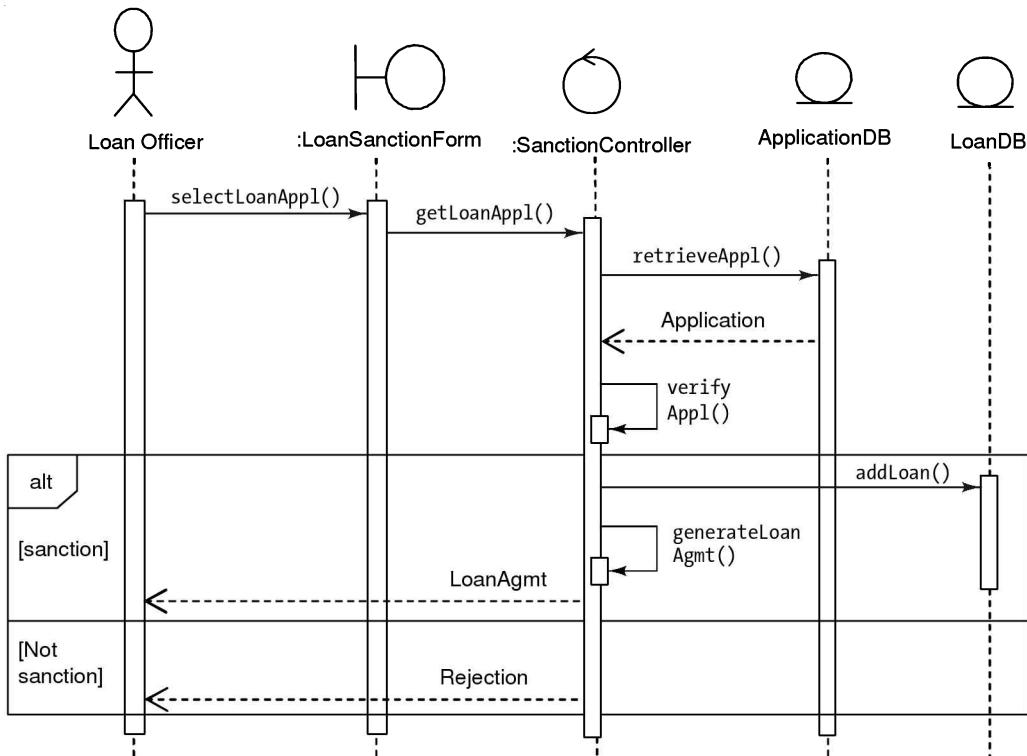


FIGURE 8.20 Sequence diagram for sanction loan and prepare loan agreement use-case.

For sanctioning the loan and preparing loan agreement, the sequence flow will be as follows:

- Step 1.* Loan Officer at the university will open the LoanSanctionForm and selects the loan applications to be sanctioned and submit those.
- Step 2.* After submitting the selected loan application for sanctioning, the SanctionController will verify those applications.
- Step 3.* After application verification if loan is sanctioned, a new loan account is created in LoanDB and Loan Agreement is generated by the SanctionController.
- Step 4.* After application verification, if loan is not sanctioned, the rejection message is send.

For steps 3 and 4, the combined fragment is used, where there are two alternatives—sanction and not sanction. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 8.20.

Sequence diagram for generate loan balance statement use-case

The use-case *generate loan balance statement* is performed by Loan Officer at university for generating the loan balance statement using Student Loan system.

For the *generate loan balance statement* use-case, ask four questions and answers to those questions will be the objects interacting with each other for performing the process.

1. Who will generate loan balance statement? (Actor)
Loan Officer at university
2. Through which web page (interface)? (Boundary)
BalStatmtForm
3. Who will co-ordinate the process? (Control)
BalStatmtController
4. Which object holds loan balance amount? (Entity)
LoanDB

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls as in Table 8.11.

TABLE 8.11 Classes their types and methods for generate loan balance statement use-case

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Loan Officer	<i>Not required to specify in this context.</i>
Boundary	BalStatmtForm	setLoanAcNo()
Control	BalStatmtController	getLoanAcNo() validateLoanAc() generateLoanBalStmt()
Entity	LoanDB	retrieveLoanAc(), retrieveLoanBal()

For generating a loan balance statement, the sequence flow will be as follows:

- Step 1. Loan Officer at university will open the BalStatmtForm and specify the loan account number of which the loan balance statement is to be generated and submit it.
- Step 2. After submitting the loan account number, BalStatmtController will retrieve the loan account details and validate the loan account.
- Step 3. After the loan account validation, if the loan account is valid, retrieve the loan balance amount and generate the loan balance statement.
- Step 4. After the loan account validation, if the loan account is not valid send the message "Invalid Loan Account".

For steps 3 and 4, the combined fragment is used, where there are two alternatives—valid Loan Ac and Invalid Loan Ac. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 8.21.

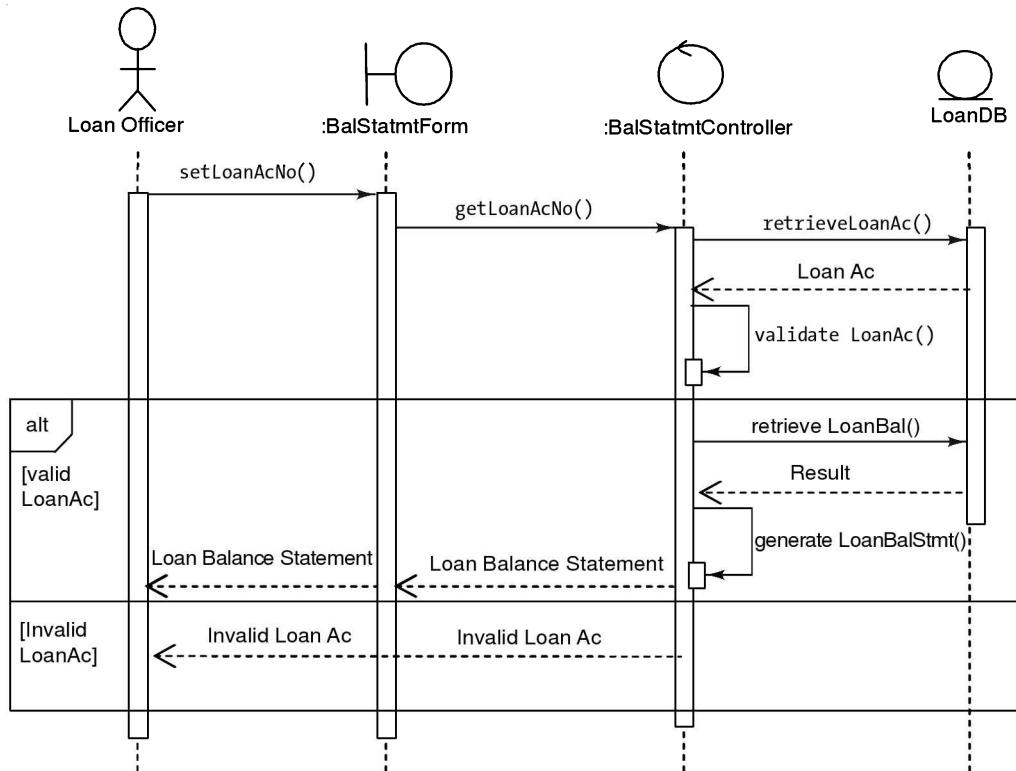


FIGURE 8.21 Sequence diagram for generate loan balance statement use-case.

Sequence diagram for generate payment acknowledgement use-case

The use-case *generate payment acknowledgement* is performed by Loan Officer at university for generating the repayment acknowledgement using the student loan system.

For the generate payment acknowledgement use-case, ask four questions and answers to those questions will be the objects interacting with each other for the process.

1. Who will generate payment acknowledgement? (Actor)
Loan Officer at university.
2. Through which web page (interface)? (Boundary)
AcknowledgeForm.
3. Who will handle the process? (Control)
AcknowledgeController.
4. Which object holds loan repayment details? (Entity)
RepaymentDB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method call as shown in Table 8.12.

TABLE 8.12 Classes their types and methods for generate payment acknowledgement use-case

Class Type	Classes	Methods
Actor	Loan Officer	<i>Not required to specify in this context.</i>
Boundary	Acknowledge Form	setLoanAcNo()
Control	Acknowledge Controller	getLoanAcNo() validateLoanAc() generateAcknowledge()
Entity	RepaymentDB	retrieveLoanAc() retrieveRepayDtls()

For generating the loan balance statement, the sequence flow will be as follows:

- Step 1.* Loan Officer at the university will open the Acknowledge Form and specify the loan account number of which the payment acknowledgement is to be generated and submit it.
- Step 2.* After submitting the loan account number, the AcknowledgeController will retrieve the loan account details and validate the loan account.
- Step 3.* After the loan account validation, if the loan account is valid retrieve the repayment details and generate the payment acknowledgement.
- Step 4.* After the loan account validation, if the loan account is not valid send the message “Invalid Loan Account”.

For steps 3 and 4, the combined fragment is used, where there are two alternatives—valid Loan Ac and Invalid Loan Ac. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 8.22.

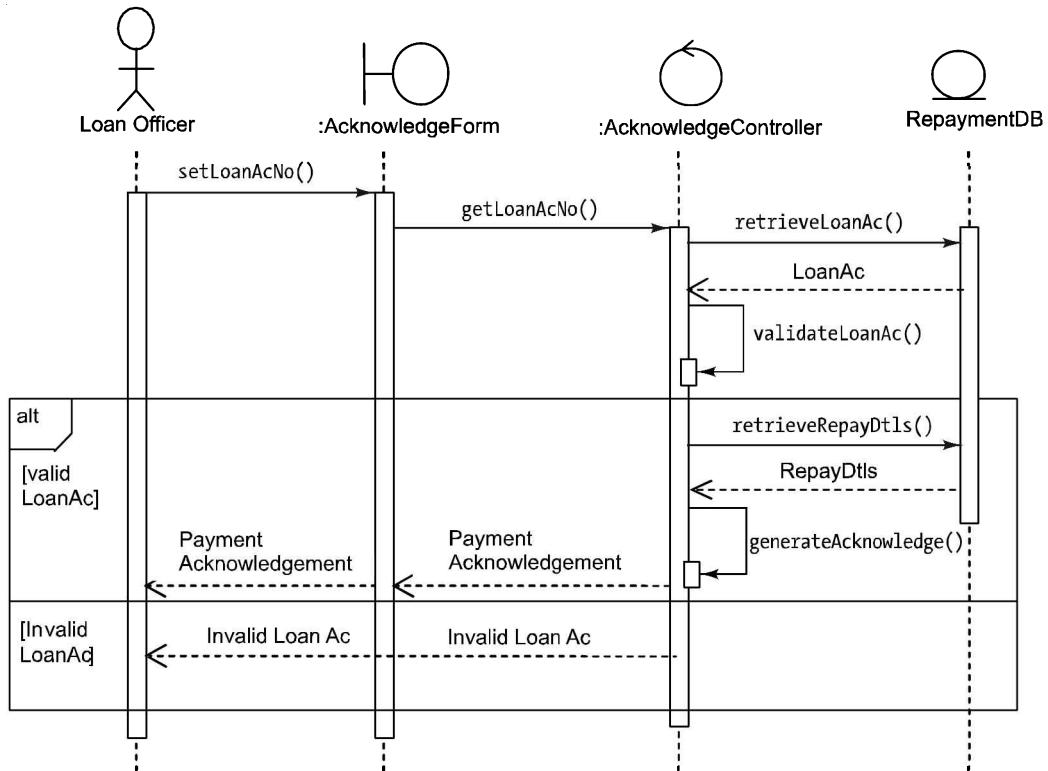


FIGURE 8.22 Sequence diagram for generate payment acknowledgement use-case.

8.3.2 Collaboration Diagram

Collaboration diagram for registration use-case

Table 8.13 lists objects, links and methods for Registration use-case. In Figure 8.23, the collaboration diagram for registration use-case is depicted.

TABLE 8.13 Objects, links and methods for Registration use-case

<i>Objects</i>	<i>Links</i>	<i>Methods (with sequence number)</i>
Student/Loan Officer	Student/Loan Officer— RegistrationForm	1. fillForm() 2. receiveDetails()
Registration Form	RegistrationForm— RegistrationController	3. validateDetails()
Registration Controller	RegistrationController—User	4. addUser()
User		5. Registration Confirmation/Cancelled

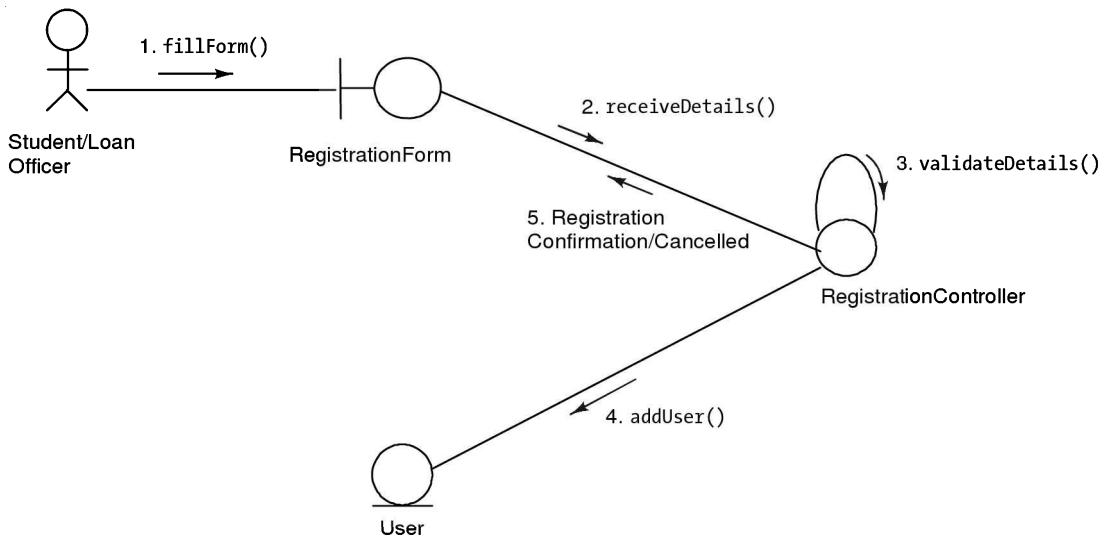


FIGURE 8.23 Collaboration diagram for registration use-case.

Collaboration diagram for login use-case

Table 8.14 lists objects, links and methods for the login use-case. Figure 8.24 shows the collaboration diagram for the login use-case.

TABLE 8.14 Objects, links and methods for login use-case

Objects	Links	Methods (with sequence number)
Student/Loan Officer	Student/Loan Officer—Login Form	1. getUserDetails()
LoginForm	LoginForm—LoginController	2. validateUser()
Login Controller	LoginController—User	3. retrieveUserDetails 4. UserDetails 5. validateUser() 6. Result
User		

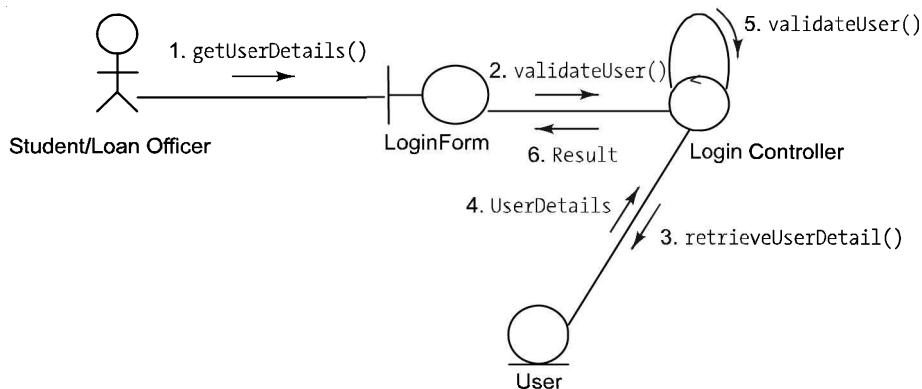


FIGURE 8.24 Collaboration diagram for login use-case.

Collaboration diagram for loan enquiry use-case

Object, links and methods for the loan enquiry use-case are given in Table 8.15. In figure 8.25, the collaboration diagram for the loan enquiry use-case is shown in Figure 8.25.

TABLE 8.15 Objects, links and methods for loan enquiry use-case

<i>Objects</i>	<i>Links</i>	<i>Methods (with sequence number)</i>
Student	Student—EnquiryForm	1. fillForm()
EnquiryForm	EnquiryForm—EnquiryController	2. receiveEnquiry()
EnquiryController	EnquiryController—User	3. addEnquiry()
EnquiryDB	EnquiryController—EnquiryDB	4. selectEnquiry()
Loan Officer	LoanOfficer—EnquiryRespForm	5. getEnquiry()
EnquiryRespForm	EnquiryRespForm—EnquiryController	6. retrieveEnquiry() 7. giveResponse() 8. Response

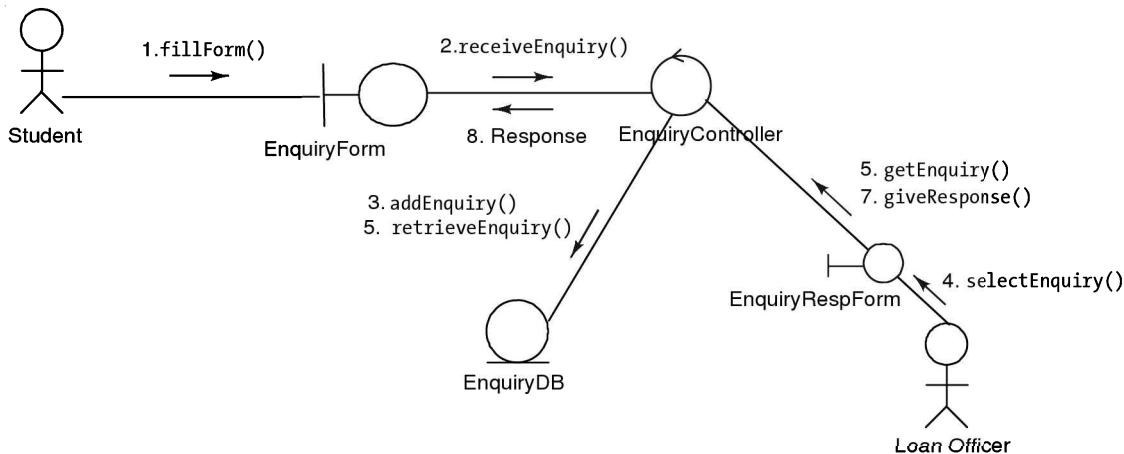


Figure 8.25 Collaboration diagram for loan enquiry use-case.

Collaboration diagram for apply for loan use-case

Table 8.16 lists the objects, links and methods for the apply for loan use-case. Figure 8.26 shows the collaboration diagram for the apply for loan use-case.

TABLE 8.16 Objects, links and methods for apply for loan use-case

Objects	Links	Methods (with sequence number)
Student	Student—ApplicationForm	1. fillForm()
ApplicationForm	ApplicationForm—ApplicationController	2. receiveAppl()
ApplicationController	ApplicationController—ApplicationDB	3. validateForm()
LoanDB	ApplicationController—LoanDB	4. addLoanApplication
ApplicationDB	ApplicationController—LoanOfficer	5. selectAppl()
Loan Officer	Loan Officer—VerifyApplForm()	6. getAppl()
VerifyApplForm	VerifyApplForm—ApplicationController	7. retrieveAppl()
		8. verifyAppl()
		9. addLoan()
		10. Confirmation/Rejection

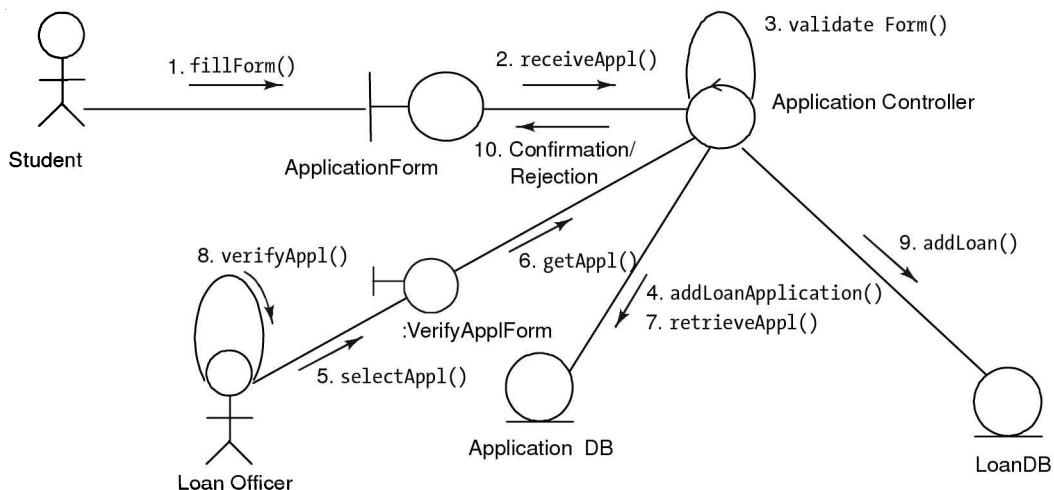


FIGURE 8.26 Collaboration diagram for apply for loan use-case.

Collaboration diagram for loan balance enquiry use-case

Table 8.17 lists the objects, links and methods for loan balance enquiry use-case. In Figure 8.27, the collaboration diagram for the loan balance enquiry use-case.

TABLE 8.17 Objects, links and methods for loan balance enquiry use-case

Objects	Links	Methods (with sequence number)
Student	Student—BalanceEnqForm	1. setLoanAcNo()
BalanceEnqForm	BalanceEnqForm— BalanceEnqController	2. getLoanAcNo()
BalanceEnqController	BalanceEnqController—LoanDB	3. retrieveLoanDtls() 4. LoanDtls 5. validateLoan() 6. retrieveLoanBal() 7. Result
Loan DB		8. BalanceAmt/ErrMsg

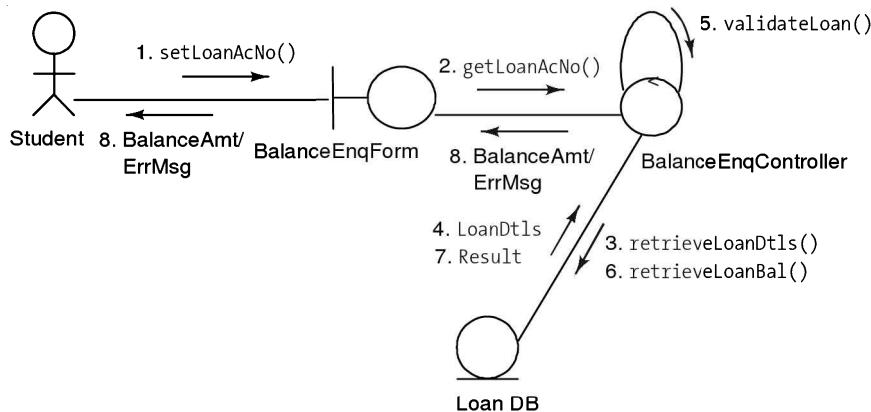


FIGURE 8.27 Collaboration diagram for loan balance enquiry use-case.

Collaboration diagram for repayment of loan use-case

Table 8.18 lists the objects, links and methods for the repayment of loan use-case. Figure 8.28 shows the collaboration diagram for the loan balance enquiry use-case.

TABLE 8.18 Objects, links and methods for repayment of loan use-case

Objects	Links	Methods (with sequence number)
Student	Student—Payment Form	1. setPaymentDtls()
PaymentForm	PaymentForm—Payment Controller	2. getPaymentDtls()
PaymentController	PaymentController—Repayment DB	3. validatePaymentDtls()
RepaymentDB		4. addPayment() 5. Result 6. Confirmation/Rejection

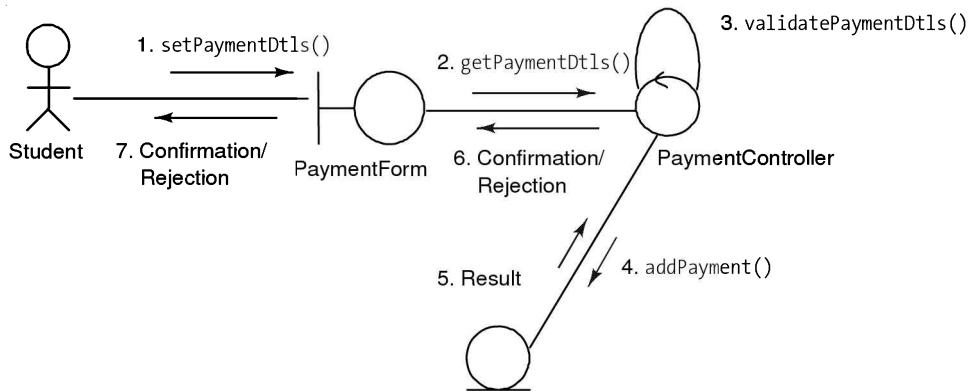


FIGURE 8.28 Collaboration diagram for repayment of loan use-case.

Collaboration diagram for sanction loan and prepare loan agreement use-case

In Table 8.19 objects, links and methods for the sanction loan and prepare loan agreement use-case. Figure 8.29 shows the collaboration diagram for the sanction loan and prepare loan agreement use-case.

TABLE 8.19 Objects, links and methods for sanction loan and prepare loan agreement use-case

Objects	Links	Methods (with sequence number)
Loan Officer	Loan Officer—LoanSanctionForm	1. selectLoanAppl()
LoanSanction Form	LoanSanctionForm—SanctionController	2. getLoanAppl()
SanctionController	SanctionController—ApplicationDB	3. retrieveAppl()
Loan DB	SanctionController—LoanDB	4. Application
Application DB	Loan Officer—ApplicationDB	5. verifyAppl()
	Loan Officer—Student	6. addLoan()
		7. generateLoanAggmt()
		8. LoanAgmt/Rejection

Collaboration diagram for generate loan balance statement use-case

Table 8.20 lists the objects, links and methods for the generate loan balance statement use-case. Figure 8.30 shows the collaboration diagram for the generate loan balance statement use-case.

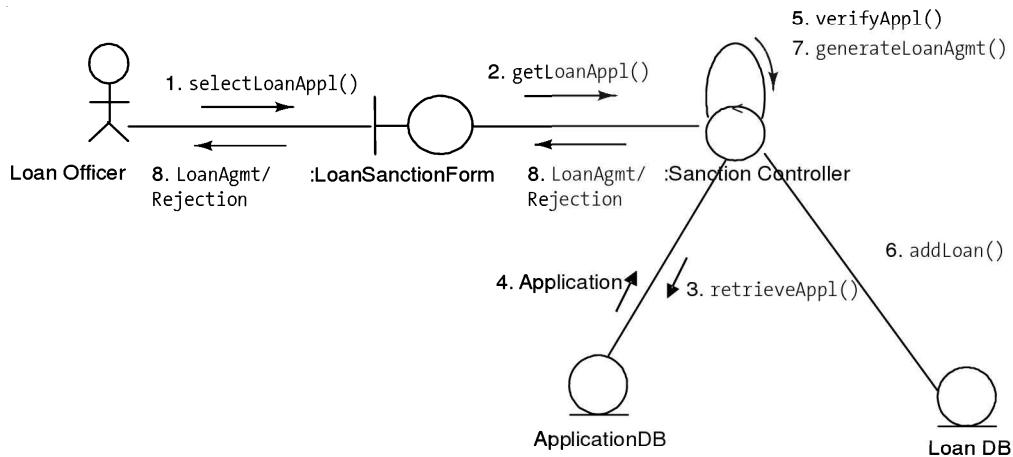


FIGURE 8.29 Collaboration diagram for sanction loan and prepare loan agreement use-case.

Table 8.20 Objects links and methods for generate loan balance statement use-case

Objects	Links	Methods (with sequence number)
Loan Officer	Loan Officer—BalStatmtForm	1. setLoanAcNo()
BalStatmtForm	BalStatmtForm BalStatmtController	2. getLoanAcNo()
BalStatmtController	BalStatmtController—LoanDB	3. retrieveLoanAc()
LoanDB		4. LoanAc 5. validateLoanAc() 6. retrieveLoanBal 7. Result 8. generateLoanBalStmt() 9. LoanBalStmt/ErrMsg

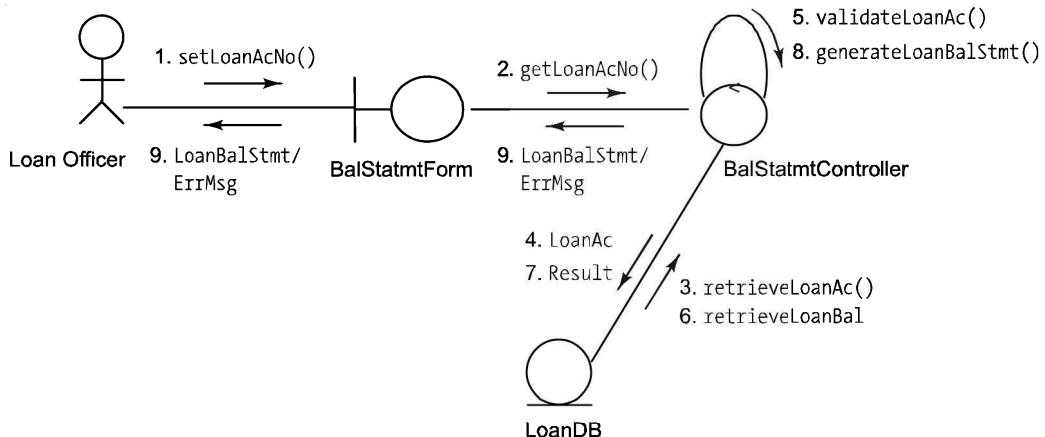


FIGURE 8.30 Collaboration diagram for generate loan balance statement use-case.

Collaboration diagram for generate payment acknowledgement use-case

Table 8.21 lists the objects, links and methods for the generate payment acknowledgement use-case. Figure 8.31 shows the collaboration diagram for the generate payment statement use-case.

TABLE 8.21 Objects, links and methods for generate payment acknowledgement use-case

Objects	Links	Methods (with sequence number)
Loan Officer	Loan Officer—AcknowledgeForm	1. setLoanAcNo()
Acknowledge Form	AcknowledgeForm—AcknowledgeController	2. getLoanAcNo()
Acknowledge Controller	AcknowledgeController—Repayment DB	3. retrieveLoanAc()
Repayment DB		4. LoanAc 5. validateLoanAc() 6. retrieveLoanBal() 7. Result 8. generateAcknowledge() 9. Stmt/ErrMsg

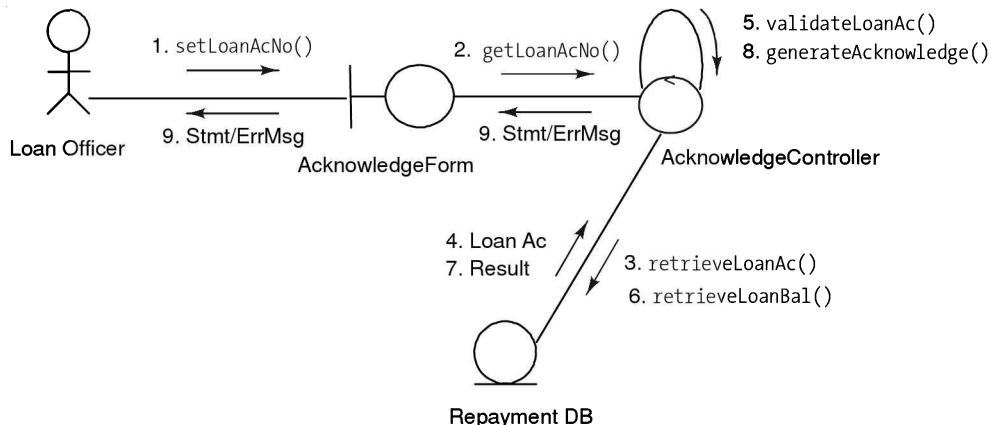


FIGURE 8.31 Collaboration diagram for generate payment statement use-case.

8.3.3 Statechart Diagram

In this problem statement, the object for which state chart diagram is analyzed is *Student Loan*.

The *student loan* object has fixed states throughout its life cycle and there maybe some abnormal exits also. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as the complete transaction.

The first state is receiving state from where the process starts. The next state is arrived for event *send application*. This event is responsible for state change of the student loan object.

As the *student loan* object has finite states throughout its life cycle, the statechart diagram will be one shot life cycle statechart diagram.

Hence there will be one initial state and one or more final state as in Figures 8.32(a) and (b). The intermediate states are shown in Figure 8.32(c).

All the transitions shown in Figure 8.32(d) are trigger-less transitions which occurs on completion of activity during the previous state. The guard condition to enter from *Evaluation* state into *Disbursement* or *Rejection state* is *eligible/Not eligible*. No trigger is required to transit from the state. The complete state transition diagram for the *student loan* object is as shown in Figure 8.32(e).



FIGURE 8.32(a) Initial state: Begin loan.



FIGURE 8.32(b) Final state: End loan.

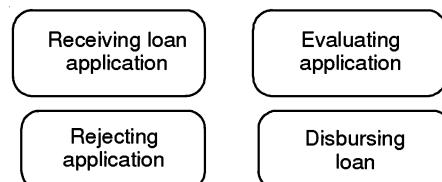


FIGURE 8.32(c) Intermediate states.

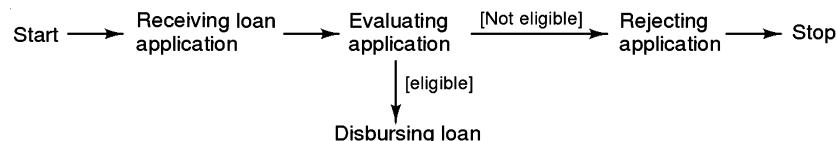


FIGURE 8.32(d) Transitions.

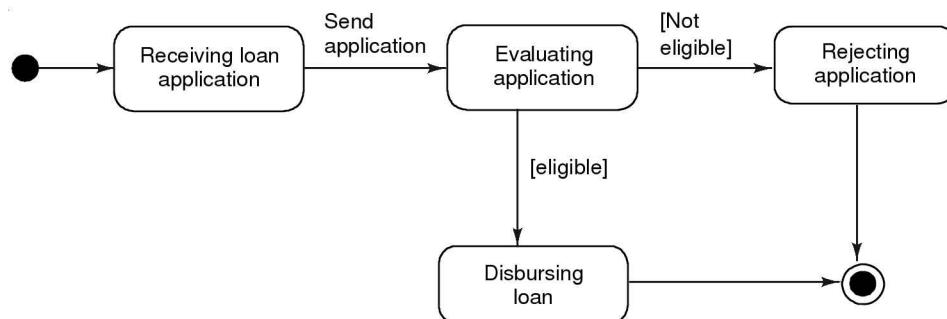


FIGURE 8.32(e) State transition diagram for student loan system.

8.3.4 Activity Diagrams

There are two main processes in the system for which an activity diagram should be drawn:

- Getting loan
- Repayment of loan

For drawing an activity diagram for any process:

1. Find out swimlanes if any. To find swimlanes, see if you can span some activities over different organizational units/places.
2. Find out in which swimlane the loan process begins and where it ends. Those will be the initial and final states.
3. Then, identify the activities occurring in each swimlane. Arrange activities in sequence flow spanning over all the swimlanes.
4. Identify the conditional flow or parallel flow of activities. The parallel flow of activities must converge at a single point using a join bar.
5. During the activities are performed, if any document is generated or used, take it as an object and show the object flow.

Activity diagram for getting loan

Swimlanes: As the Student interacts with the University Loan Officer for getting an educational loan from the University, there are only two swimlanes—Student and University Loan Officer [Figure 8.33(a)]. Activities are shown in Figure 8.33(b).

As stated in the problem statement, no parallel flow is identified. One conditional flow occurs after the *Evaluation for eligibility* activity which proceeds to prepare loan agreement if the student is eligible for getting loan or exits otherwise. So the complete activity diagram for the process with the transitions is as shown in Figure 8.33(c):

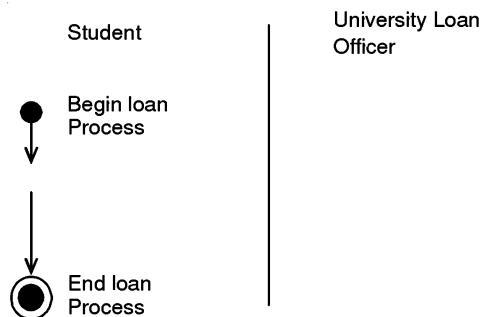


FIGURE 8.33(a) Swimlanes.

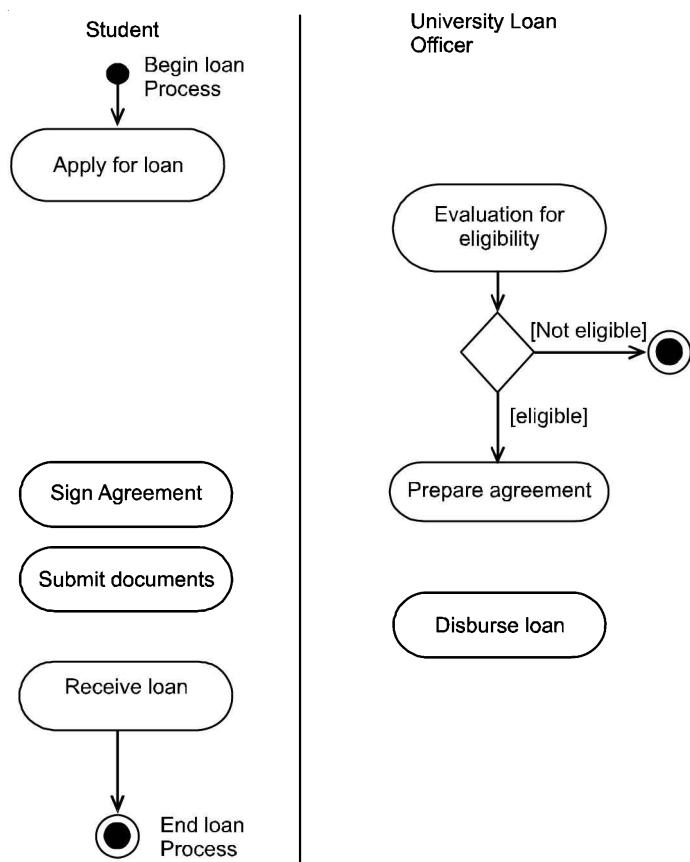


FIGURE 8.33(b) Activities.

Activity diagram for repayment of loan

Swimlanes: As the Student interacts with the University's Loan Officer for repaying the outstanding loan, there are two swimlanes, identified as Student and University Loan Officer [Figure 8.34(a)] and the activities involved are shown in Figure 8.34(b).

As stated in the problem statement, no parallel flow is identified. No conditional flow during this activity is identified.

So the complete activity diagram for the process with the transitions is as shown in Figure 8.34(c).

8.4 IMPLEMENTATION PHASE DIAGRAMS: STUDENT LOAN SYSTEM

8.4.1 Component Diagram

Component diagrams illustrate the pieces of software that will make up a system. So, major components identified making up this system are as follows:

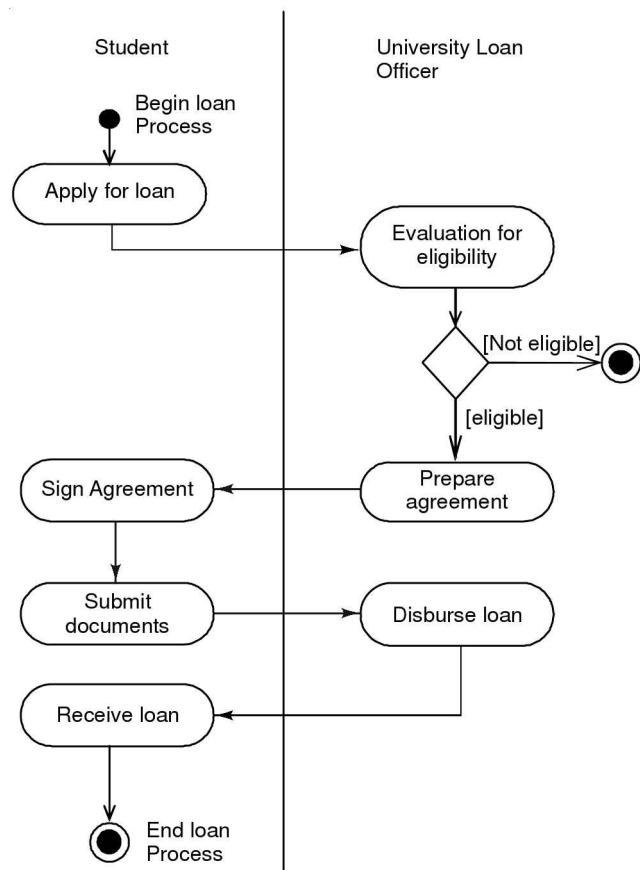


FIGURE 8.33(c) Activity diagram for getting loan.

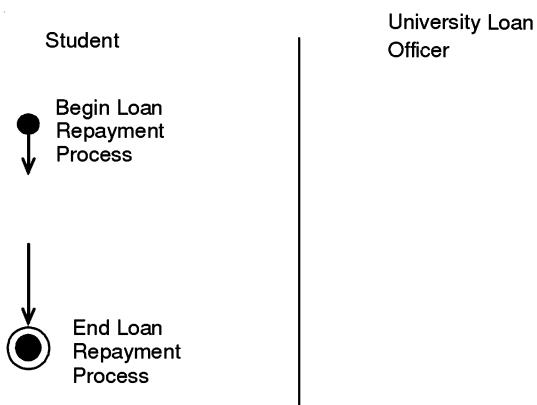


FIGURE 8.34(a) Swimlanes.

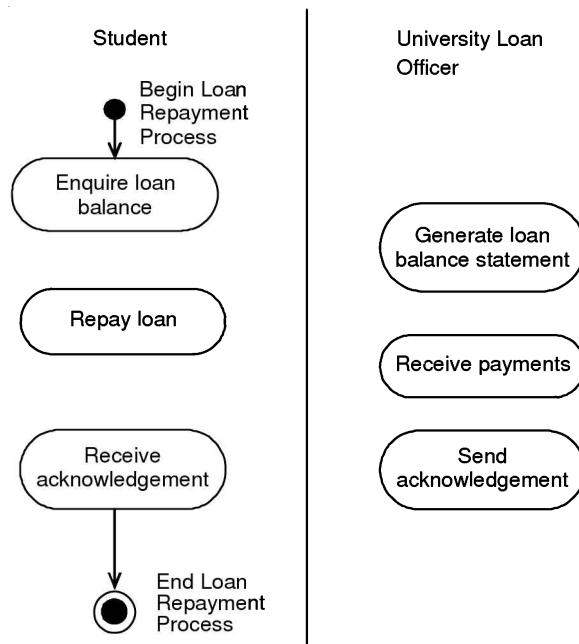


FIGURE 8.34(b) Activities.

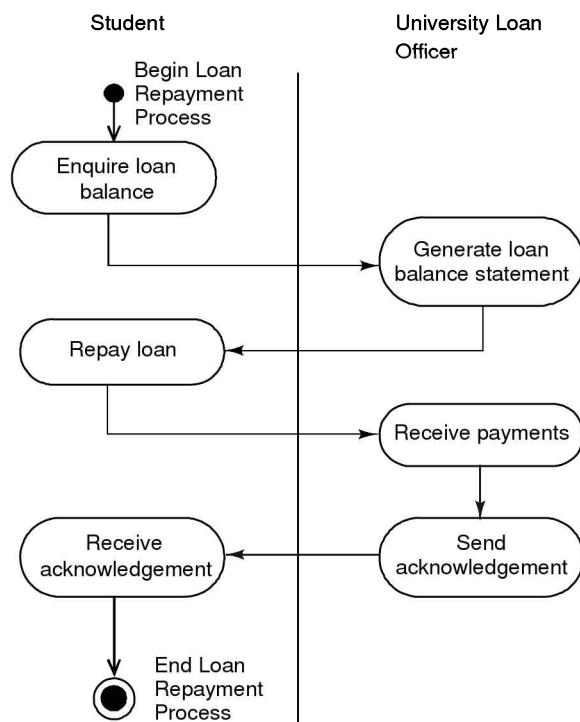


FIGURE 8.34(c) Activity diagram for repayment of loan.

1. Student Loan System
2. Student
3. Loan
4. Loan Database
5. Security and Persistence

The student loan system is for the students who want to apply for the education loan and further loan management as shown in Figure 8.35. The main application component in the system is *University's Student Loan system* which depends on components identified are Student and Loan, which are implementing corresponding interfaces *IStudent*, *ILoan*. *Persistence* and *Security* components represent functional components for storing data into database represented by the *loan database* component which is the data store component and managing the access control to the system.

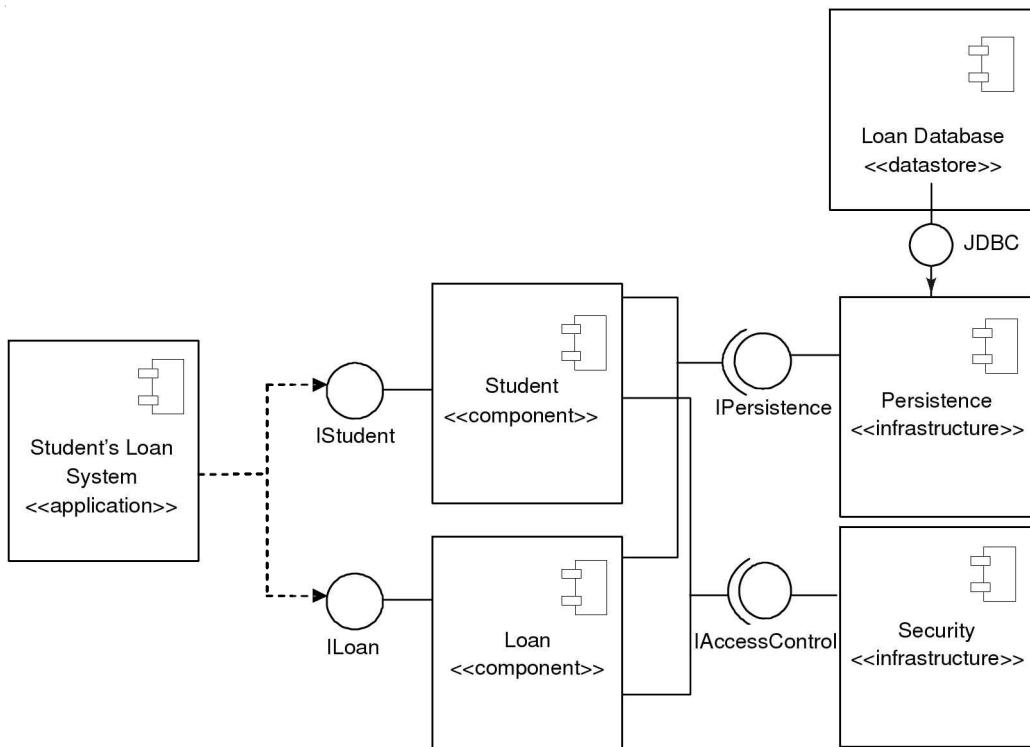


FIGURE 8.35 Component diagram for student loan system.

8.4.2 Deployment Diagram

The student loan system is an Internet-based application. It is based on multi-tier architecture. Hence there will be database tier holding loan database, application tier holding *university's student loan system*, client tier holding client workstation with browser.

Network used is WAN. Data communication among all the nodes is through TCP/IP protocol.

In Figure 8.36, the database server is representing the database layer on which the database server, e.g. MS-SQL Server software and loan database will be installed.

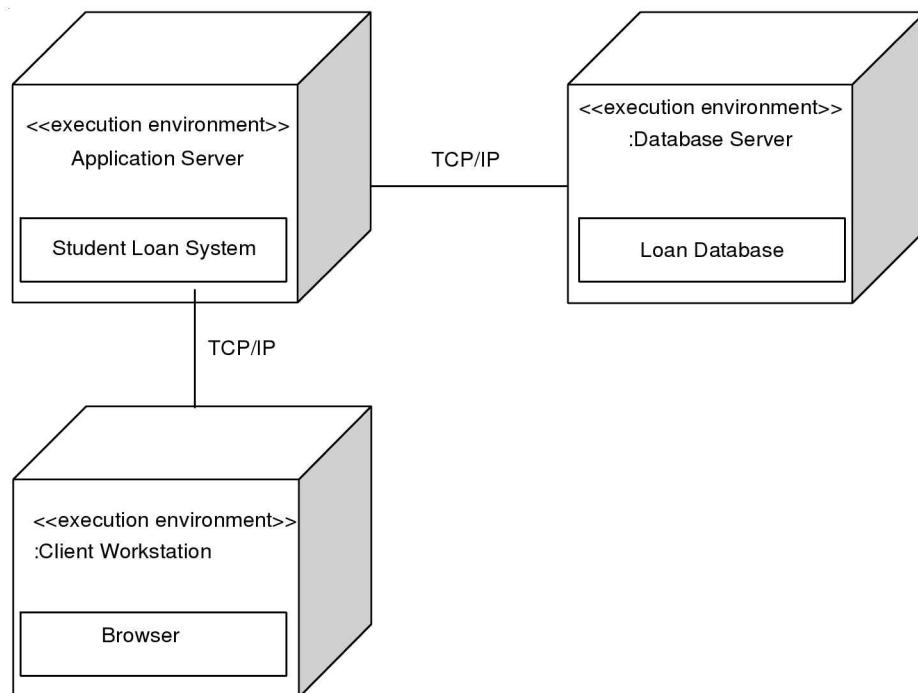


FIGURE 8.36 Deployment diagram for student loan system.

Application server is representing the application tier on which application server software, e.g. WebSphere application server and student loan system (all source code pages, dynamic link libraries etc.) will be deployed.

Clients (i.e. Students) access the application through the browser. Hence on the client tier only a browser is required.

CHAPTER

9

Case Study—On Line Trading of Securities

9.1 PROBLEM STATEMENT: ONLINE TRADING OF SECURITIES

CPL Bank is a local commercial bank based in Hong Kong. In addition to traditional banking functions (i.e. savings and current account), CPL also operates a securities department to provide securities trading services to its customers.

Before a customer can trade via CPL, he/she needs to open a securities account. This can be done by filling in a Securities Account Application Form. There is also a prerequisite that the customer must have got either a current or saving account with CPL, for settlement of the securities transaction amounts.

Once a securities account is opened, the customer can place order to buy and sell any stocks listed in the Hong Kong Exchange (HKEx), via telephone or Internet or through the counters in the branches. Fax instructions can also be accepted if the customer has completed a properly executed Fax Indemnity, either during or subsequent to the account open time.

The buy/sell orders received from customers should be placed to HKEx's trading system HTStock. However, CPL needs to ensure that the customer has got sufficient credit or stock holding (short-selling is not allowed) in his/her account before placing the order. For buy orders, CPL will place a hold on the customer's savings/current account for the buying amount to ensure that he/she will be able to pay for the order.

Order submitted to HTStock will either be executed immediately (if there are pending orders in the market which meet the price of the submitted order), or queued for matching price. An order will be transmitted back to CPL once it is executed. CPL will then need to advise the customer immediately about the execution price and volume (sometimes partial execution of an order will result if the customer allows it) via telephone call, except in the case of Internet order which the confirmation will be via Internet. At the end of the day, all the unexecuted orders will be dropped.

A monthly statement will be sent to each customer with details of the transactions in the month and the balance at the end of the month. CPL will also charge an annual service fee to all the securities account holders. The charge is done on September of each year. For the above application, build an Analysis Model, Design Model and Implementation Model with the following diagrams:

1. Business Process Diagram
2. Use-case Diagram

3. Class Diagram
4. Object Diagram
5. Sequence Diagram
6. Collaboration Diagram
7. Statechart Diagram
8. Activity Diagram
9. Component Diagram
10. Deployment Diagram

9.2 ANALYSIS PHASE DIAGRAMS: ONLINE TRADING OF SECURITIES

9.2.1 Business Process Diagram

The business process diagram shows the various flow objects, connecting objects, swimlanes and artifacts which are analyzed from the problem statement. This understanding mainly comes from business domain analysis and is crucial for capturing business intelligence and representing it into computer systems.

Pools and lanes

There are two identifiable pools as shown in Figure 9.1 within which various activities for online trading of securities are performed.

1. CPL Bank
2. Customer

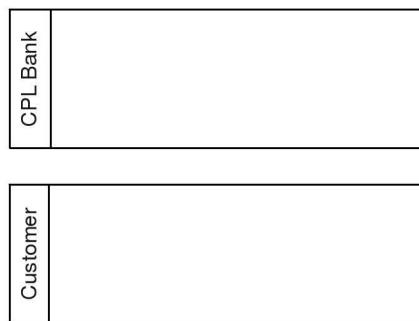


FIGURE 9.1 Pools for online trading of securities.

Lanes represent subpartitions for the objects within a pool. Now within each pool, we try to find out if there is one or more organizational roles within a pool. Here in this case, the customer pool has no lane whereas, CPL Bank pool has two lanes, namely Securities Department and HTStock as shown in Figure 9.2.

CPL Bank	HT Stock	
Securities Department		

FIGURE 9.2 Lanes within CPL pool.

Activities: Tasks performed in CPL Bank pool under various lanes are illustrated in Figure 9.3.

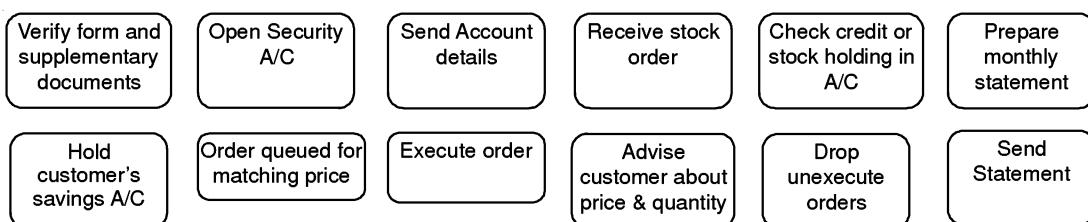


FIGURE 9.3 Tasks performed under various lanes within CPL Bank pool.

Tasks performed within the Customer pool are as shown in Figure 9.4.

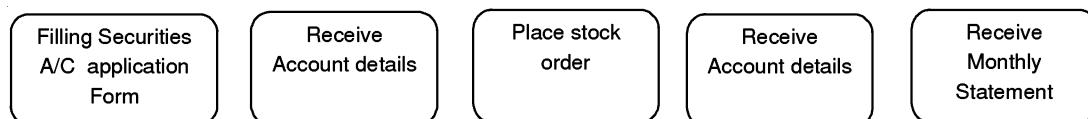


FIGURE 9.4 Tasks performed under customer pool.

Events: There are only start and end events identified in this process as shown in Figure 9.5.



Start event



End event

FIGURE 9.5 Events for Online trading of securities.

Gateways: Gateways are modelling elements that are used to control the sequence flows interaction as they converge and diverge within a process. Gateways of type Exclusive Gateway will be included in the flow to check if the customer has savings account with the bank or not,

to check if order is for buy or sell of stock, to check for pending orders and to check for matching price for trading as shown in Figure 9.6.

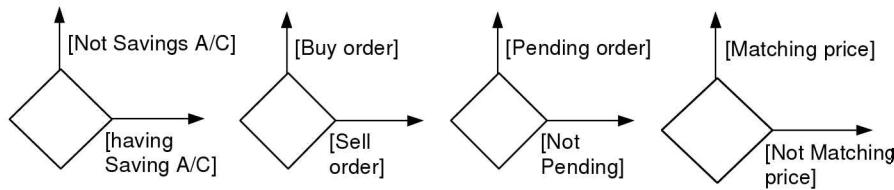


FIGURE 9.6 Gateways representing various conditions in online trading of securities system.

Artifacts: Artifacts display some additional information in the diagram such as input to activities, output from activities (Data objects), and grouping of related process objects (groups), textual comments about events, activities and gateways (comments). In this example, the only artifacts are data objects as shown in Figure 9.7.

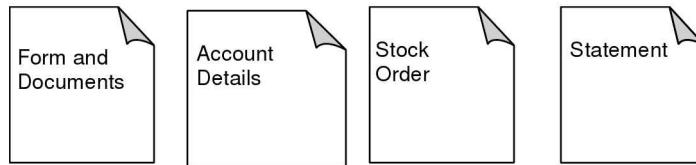


FIGURE 9.7 Data objects for online trading of securities.

A complete business process diagram for online trading of securities will be drawn as shown in Figures 9.8(a) and 9.8(b).

9.2.2 Use-Case Diagrams

There are three actors in the system as:

- Customer (Person)
- HTStock (External System)
- Bank's Securities Accounting Clerk (Person)

Identified actors and their use-cases are given in Table 9.1.

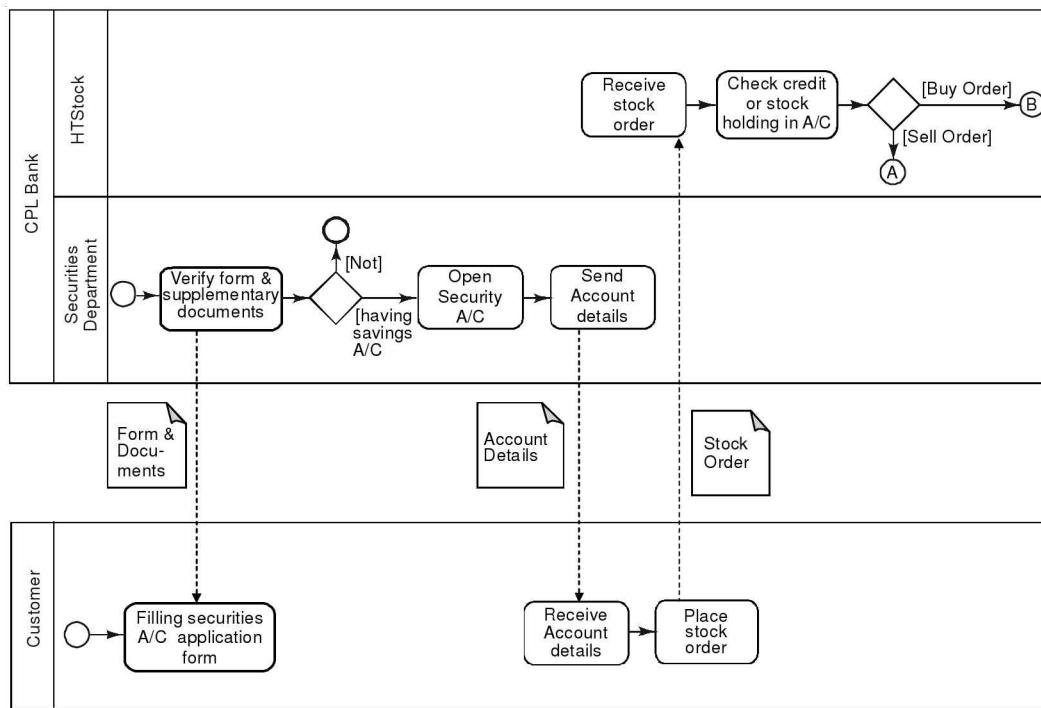


FIGURE 9.8(a) Business process diagram for online trading of securities.

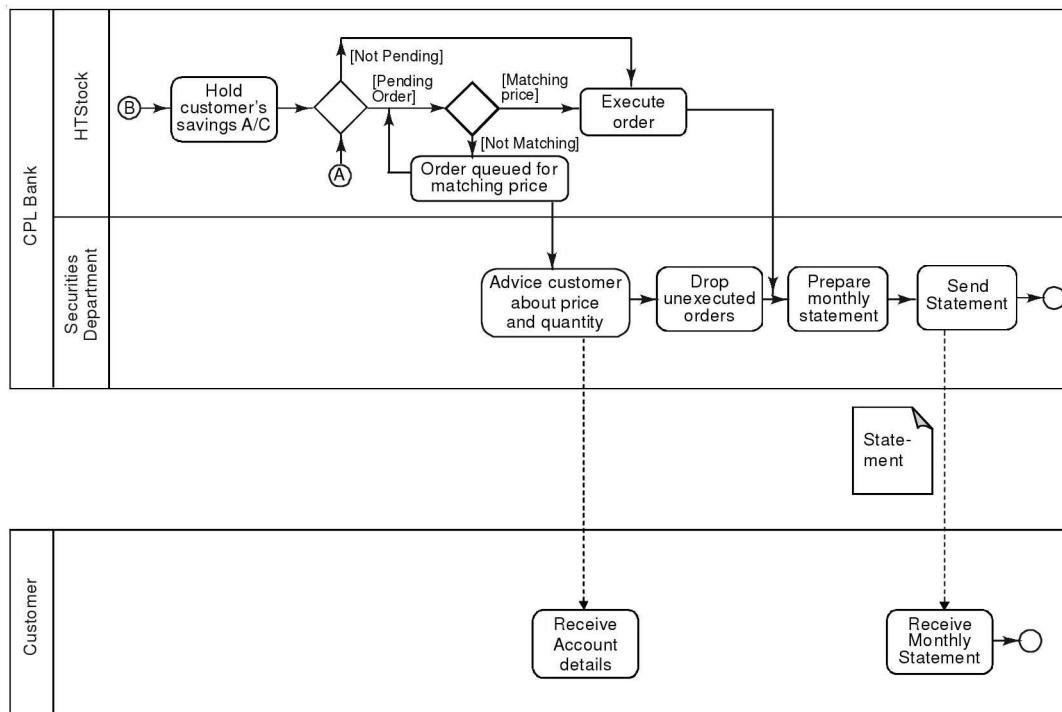


FIGURE 9.8(b) Business process diagram for online trading of securities.

TABLE 9.1 Actors and corresponding use-cases: Online trading of securities

Actors	Use Cases		
	Base	Include	Extends
Customer	Registration Login Open securities account Place stock order	Check credit or stock holding	Sell stock Buy stock
HTStock	Process placed orders		Execute placed order Drop unexecuted orders
Bank's securities Accounting clerk	Registration Login Generate monthly statement of transactions		

First, we will draw an actor-wise use-case diagram and finally a combined use-case diagram representing the whole system with all actors and use-cases performed by them.

Use-case diagram for customer: Figure 9.9(a) shows a use-case diagram for the customer.

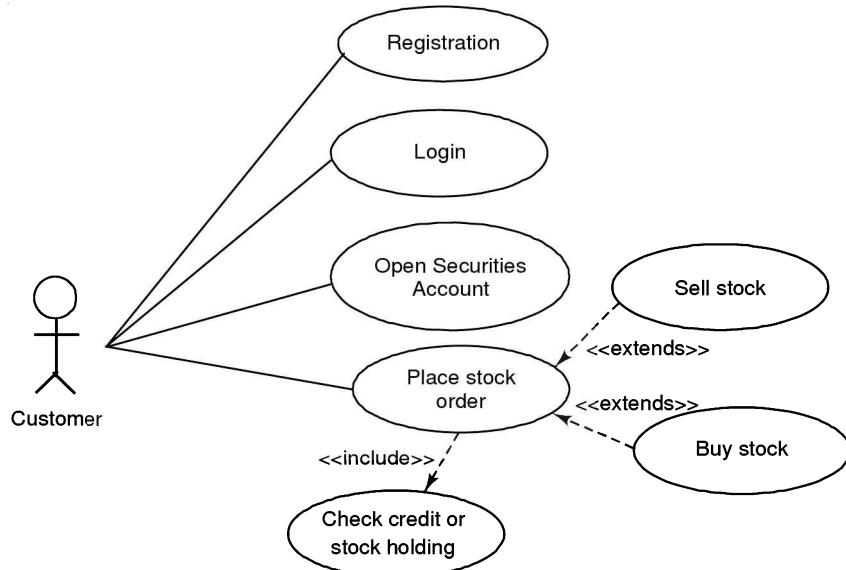


FIGURE 9.9(a) Use-case diagram for customer: Online trading of securities.

Use-case diagram for HTstock: Figure 9.9(b) shows a use-case diagram for HTstock.

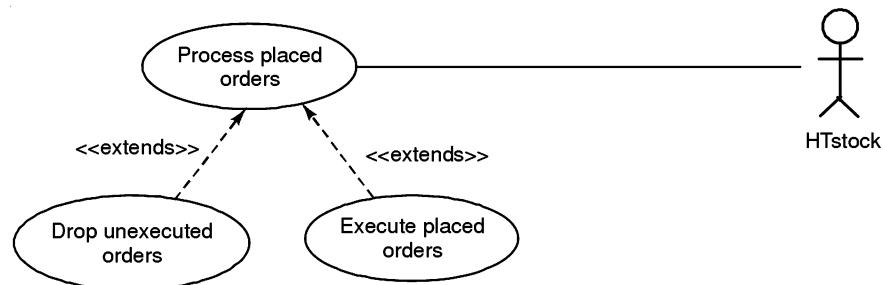


FIGURE 9.9(b) Use-case diagram for HTstock: Online trading of securities.

Use-case diagram for bank's securities accounting clerk: Figure 9.9(c) depicts a use-case diagram for the accounting clerk.

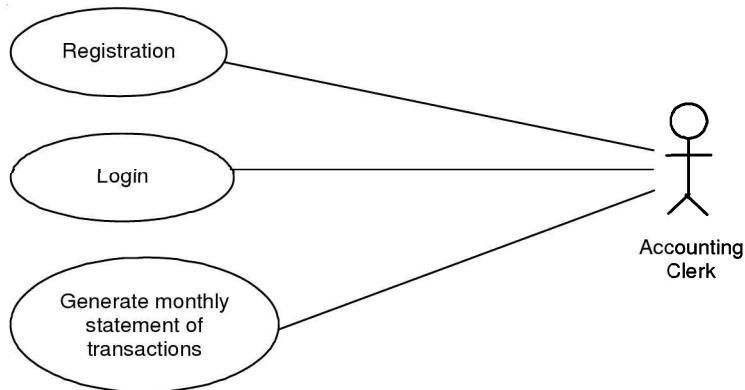


FIGURE 9.9(c) Use-case diagram for accounting clerk: Online trading of securities.

Complete use-case diagram online security trading system: Figure 9.9(d) gives the complete use-case diagram for the online trading of securities.

Important points: In the above use-case diagram, apart from use cases obvious from the problem statement as found out in Table 9.1, we have added two additional use-cases—registration and login which are not mentioned in the problem statement, but we have to consider it as everybody has different role in the system.

9.2.3 Class Diagram: Analysis Phase

Analyzing the case gives us the following interpretations:

1. CPL Bank operates the Securities Department.
2. CPL Bank has customers.

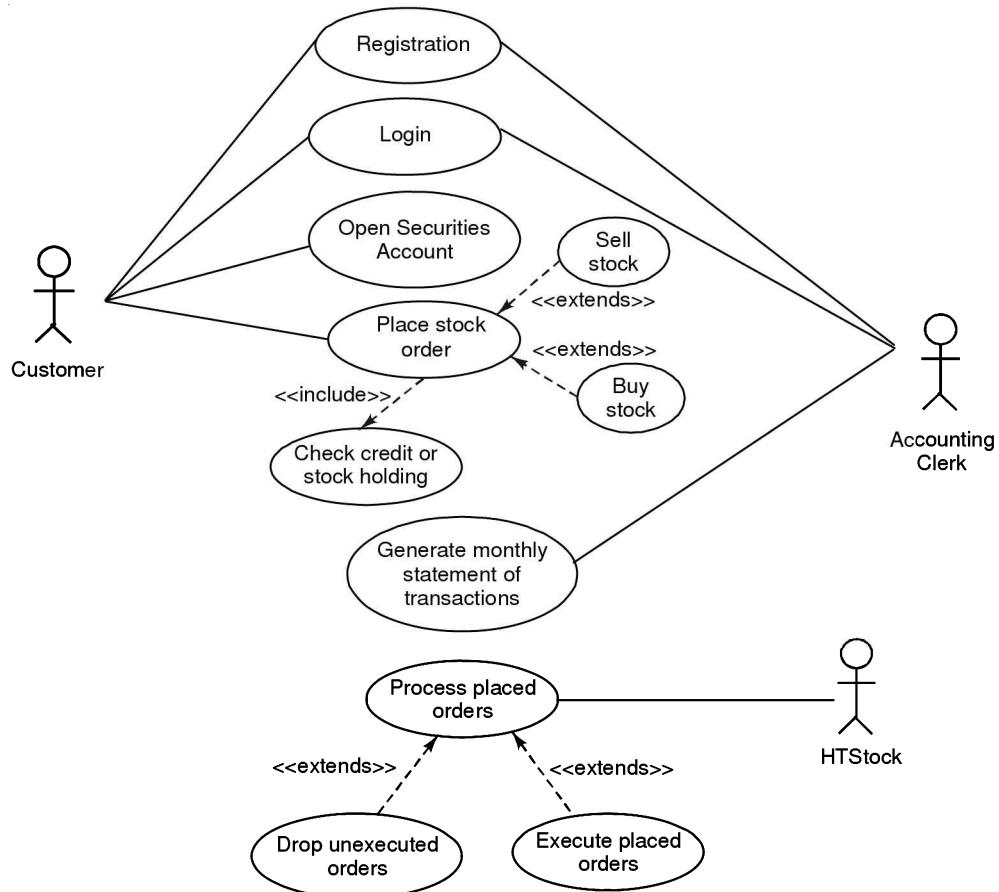


FIGURE 9.9(d) Complete use-case diagram for online trading of securities.

3. The customer has an account in the bank.
4. The account is of type Savings Account or Current Account.
5. The customer opens Securities Account.
6. The customer places stock trade order.
7. Stock trade orders are executed by stock exchange.
8. The Securities Department generates statement of transactions.
9. The customer receives the statement of transactions.

Using the interpretation the classes, their relationships and relationship types are as shown in Table 9.2.

TABLE 9.2 Class and their relationships for online trading of securities

Sr. No.	Class 1	Relationship Name	Relationship Type	Class 2
1	Bank	operates	Association	Securities department
2	Bank	has	Association	Customer
3	Customer	has	Association	Account
4	Account	is-of	Generalization	Account type (Savings A/C, Current A/C)
5	Customer	opens	Association	Securities account
6	Customer	places	Association	Stock trade order
7	Stock trade order	executed by	Association	Stock exchange
8	Securities department	generates	Association	Statement of transactions
9	Customer	receives	Association	Statement of transactions

Based on the above analyses and identification of classes and their relationships, we draw the class diagrams as shown in Figure 9.10(a) and the object diagram shown in Figure 9.10(b).

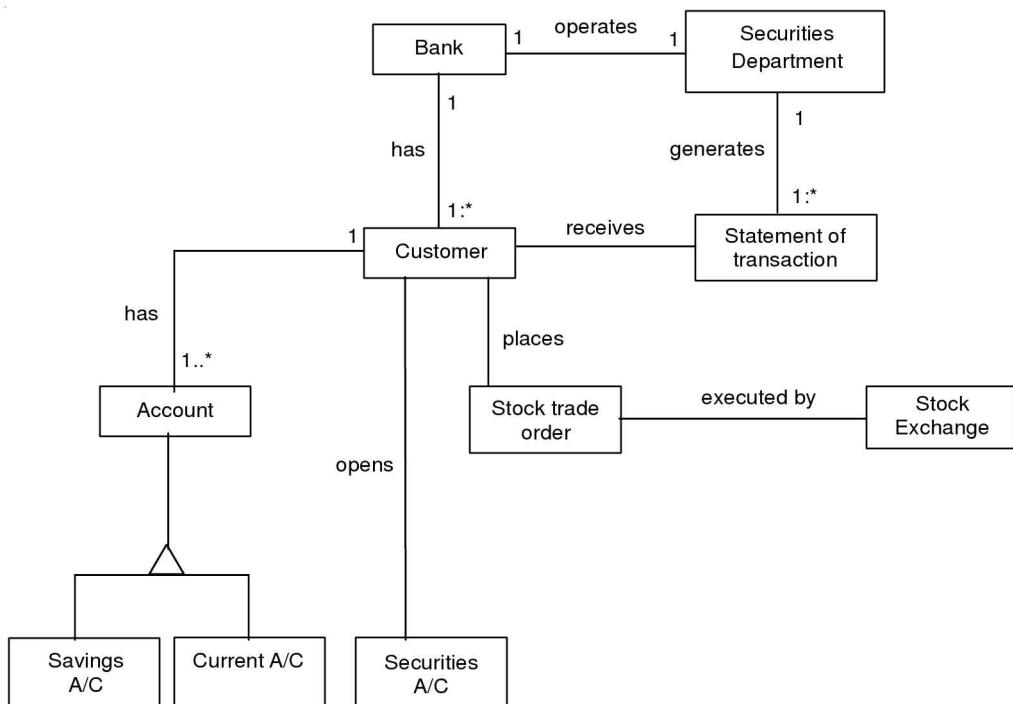


FIGURE 9.10(a) Class diagram for online trading of securities.

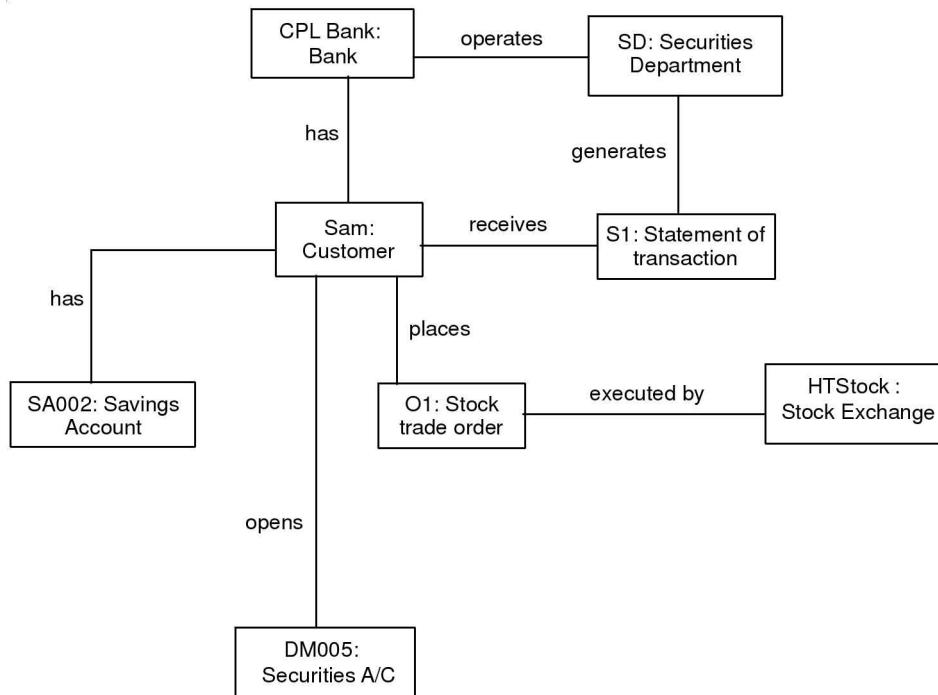


FIGURE 9.10(b) Object diagram for online trading of securities.

9.3 DESIGN PHASE DIAGRAMS: ONLINE TRADING OF SECURITIES

9.3.1 Sequence Diagrams

A sequence diagram needs to be drawn during the design phase of the system.

During analysis, we have got

- Use-case diagram
- Class diagram

We will make use of both these diagrams for design.

- For each use-case in the use-case diagram, we will draw one sequence diagram at least. If the use-case is much complex, then there can be more than one sequence diagram.
- While drawing a sequence diagram from the use-case diagram, we will draw the sequence diagram only for the base use-cases embedding the flow for <<include>> and <<extend>> use-cases.

In the above case, actors and their corresponding use-cases identified can be summarized as in Table 9.3.

TABLE 9.3 Actors and related use-cases for online trading of securities

<i>Actors</i>	<i>Use cases</i>
Customer	Registration Login Open Securities account Place stock order
HTStock	Process placed orders
Bank's Securities Accounting Clerk	Registration Login Generate monthly statement of transactions Charge annual service fees for service accounts

During the design, we have identified four types of objects interacting with each other to perform the use-case. They are as follows:

1. Actor objects—Person, external system or device that initiates the use case, e.g. Customer, HTStock, Bank's Securities Accounting Clerk.
2. Boundary objects—All the interfaces through which actor interact with system, e.g. Login Screen, Application Form, etc.
3. Controller objects—All the objects which coordinates the task, e.g. Security Manager, etc.
4. Entity objects—All domain entities identified during the analysis class diagram, e.g. Users, Account, etc.

Using these objects we attempt to design the sequence diagram as follows.

Sequence diagram for registration use-case

The use-case *registration* is performed by both actors Customer and Bank's Securities Accounting Clerk for registering as authorized users of an online trading system. For the registration use-case, we ask the basic four questions and answers to those questions will enable us to identify the objects interacting with each other for the registration process.

1. Who will register? (Actor)
Customer/Bank's Securities Accounting Clerk.
2. Through which web page (interface)? (Boundary)
Registration Form.
3. Who will add the user? (Control)
Registration Controller.
4. Which object hold valid user details? (Entity)
Users.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls and are listed in Table 9.4.

TABLE 9.4 Classes and methods for registration use-case

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Customer/Bank's Securities Accounting Clerk	<i>Not required to specify in this context.</i>
Boundary	RegistrationForm	fillForm()
Control	RegistrationController	receiveDetails() validateDetails()
Entity	Users	addUser()

For online registration, the sequence flow will be as follows:

- Step 1.* Customer/Clerk will open the Registration Form.
- Step 2.* Fill up the registration details on the form. (Name, Address, DOB, etc.)
- Step 3.* Click on the Submit button to submit the details.
- Step 4.* Before submitting the details to the server, validation for all the fields on the form (Valid e-mail, phone no., etc.) will be carried out.
- Step 5.* After form validation, the Registration controller object will validate the user, i.e. it will check if the user with the same details already exists or not or any other validation.
- Step 6.* After user validation, if the user is valid, it will be added to the database in User table and registration confirmation message will be sent to the user.
- Step 7.* After user validation, if the user is not valid, the user will not be added to the database and registration cancellation message will be sent to the user.

For steps 6 and 7, the combined fragment is used, where there are two alternatives—Valid User and Invalid User. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used. Hence the sequence diagram for the registration use-case is as shown in Figure 9.11.

Sequence diagram for login use-case

The use-case *login* is performed by both actors, namely Customer and Bank's Securities Accounting Clerk for an authorized access to the trading system. For the login use-case, we again ask the basic four questions and the answers of which identify objects interacting with each other for the login process.

1. Who will login? (Actor)
Customer/Bank's Securities Accounting Clerk.
2. Through which web page (interface)? (Boundary)
LoginForm.
3. Who will validate the user? (Control)
Login Controller.
4. Which object holds valid user details? (Entity)
Users.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls

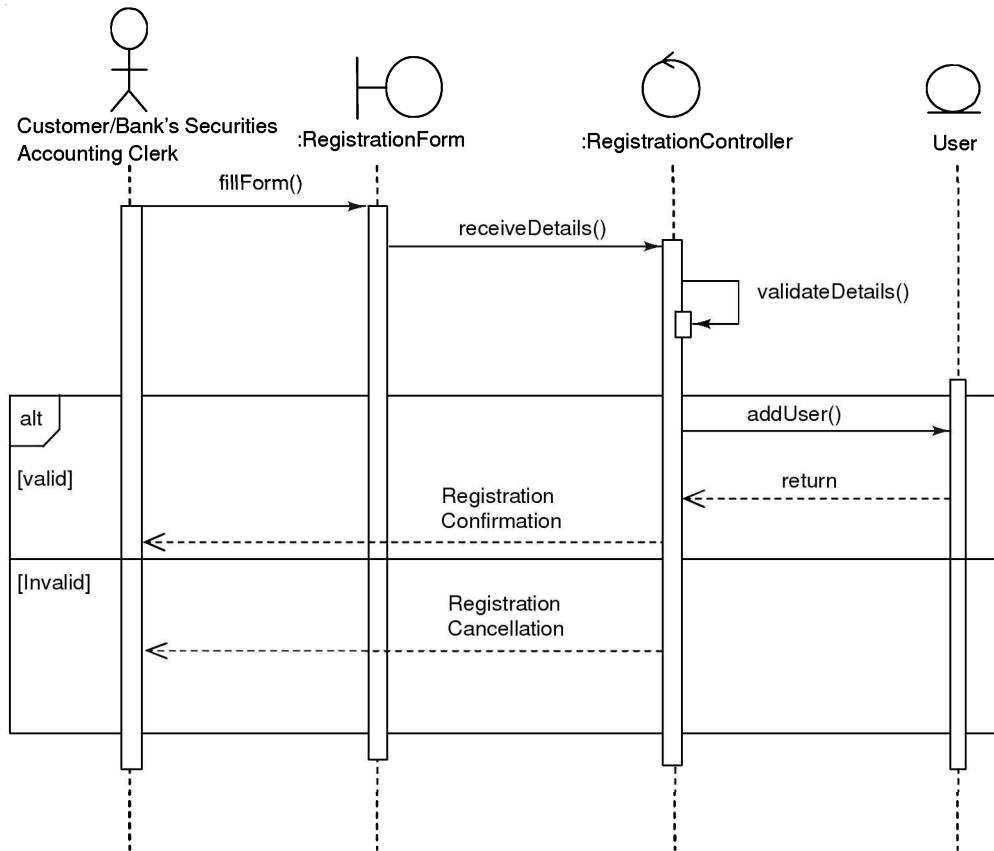


FIGURE 9.11 Sequence diagram for registration use-case.

as illustrated in Table 9.5. The sequence diagram for the login use-case is as shown in Figure 9.12.

TABLE 9.5 Classes and methods for login use-case

Class Type	Classes	Methods
Actor	Customer/Bank's Securities Accounting Clerk	<i>Not required to specify in this context</i>
Boundary	LoginForm	setUserDtls()
Control	LoginController	getUserDtls() validateUser()
Entity	Users	retrieveUserDetails()

Sequence diagram for opening securities account use-case

The use-case *opening securities account* is performed by Customer for opening securities account in the CPL Bank through the online trading of securities system.

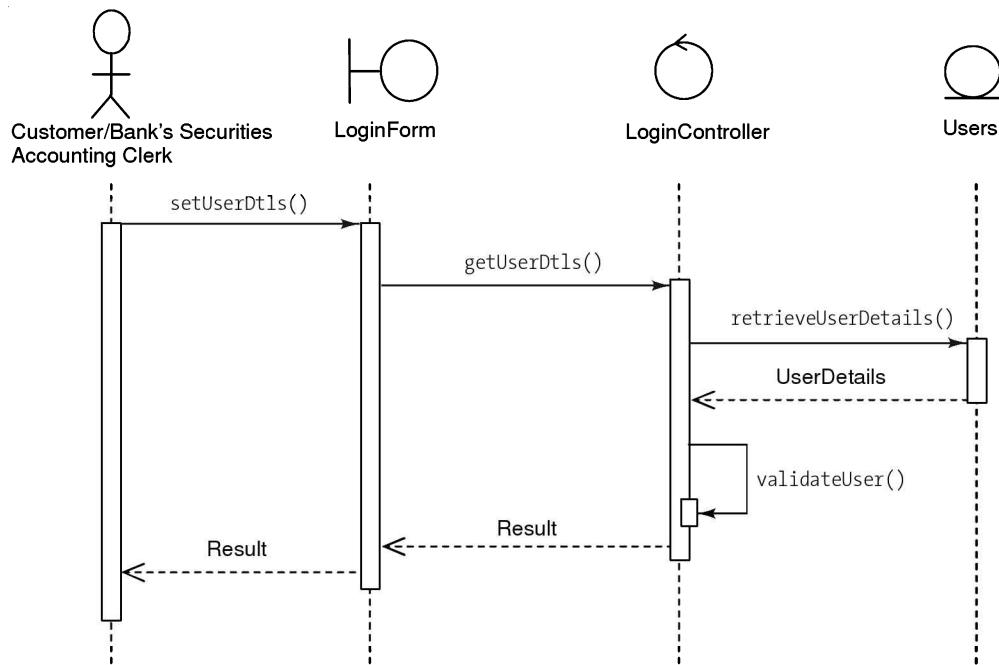


FIGURE 9.12 Sequence diagram for login use-case.

For performing the use-case, ask four questions and answers to those questions will be the objects interacting with each other for opening securities account in the bank.

1. Who will open securities account? (Actor)
Customer.
2. Through which web page (interface)? (Boundary)
ApplicationForm.
3. Who will coordinate credit card application process? (Control)
ApplicationController.
4. Which object holds credit card data? (Entity)
Application, CreditCard.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls as given in Table 9.6.

For the *opening securities account* use-case, the sequence flow will be as follows:

- Step 1.* Customer will open the Securities Account Application Form.
- Step 2.* Fill up the personal details (Contact info, Savings/Current A/C No., etc.) and submit it.
- Step 3.* Before sending the application to the database, validation for all the fields on the form (valid e-mail, phone no. etc.) will be carried out.
- Step 4.* After form validation, the Application Controller object will add the valid application into the Application DB.

TABLE 9.6 Classes and methods for opening securities account use-case

Class Type	Classes	Methods
Actor	Customer	<i>Not required to specify in this context.</i>
Boundary	ApplicationForm	fillForm()
Control	ApplicationController	receiveAppl() validateForm()
Entity	ApplicationDB	addApplication()
Entity	AccountDB	havingAccount()
Entity	SecuritiesAcDB	createsAccount()

Step 5. After adding the application into the database, processing starts. First it will check if the customer is having savings or current account in the bank or not.

Step 6. If not, the customer cannot open the securities account in the bank.

Step 7. If yes, a new securities account will be opened for the customer and it will be linked with his/her savings or current account in the bank.

For steps 6 and 7, the combined fragment is used, where there are two alternatives—having savings/current account and not having savings/current account in the bank. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 9.13.

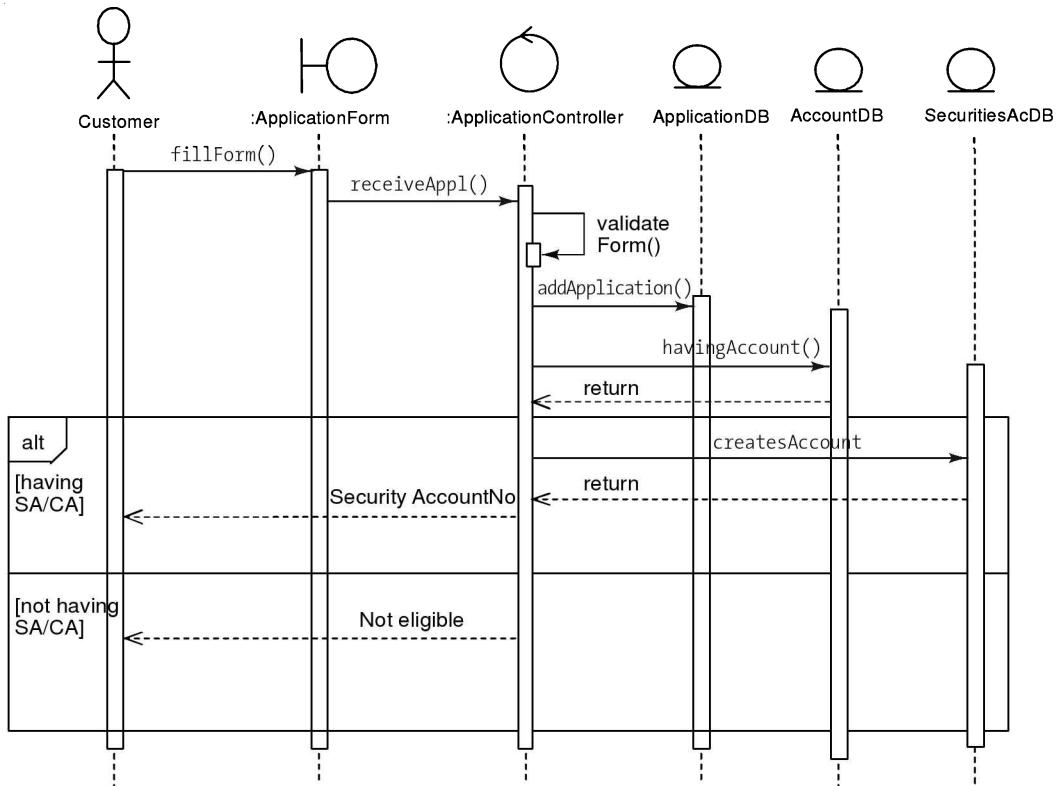


FIGURE 9.13 Sequence diagram for opening securities account use-case.

Sequence diagram for place stock order

The use-case *place stock order* is performed by actor Customer for placing stock trade order online for buying stock or selling stock using the system.

For the place stock order use-case, ask four questions and answers to those questions will be the minimum objects interacting with each other for the process.

1. Who will place stock order? (Actor)
Customer.
2. Through which web page (interface)? (Boundary)
Order Form.
3. Who will co-ordinate order? (Control)
Order Controller.
4. Which object holds order details? (Entity)
Order DB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls as illustrated in Table 9.7.

TABLE 9.7 Classes and methods for place stock order use-case

Class Type	Classes	Methods
Actor	Customer	<i>Not required to specify in this context.</i>
Boundary	OrderForm	fillOrderForm()
Control	OrderController	ReceiveOrder(), validateOrder()
Entity	OrderDB	addOrder()
Entity	AccountDB	retrieveBal(), holdAc()

For online placing stock order, the sequence flow will be as follows:

Precondition for this use-case is that, the *User must login*.

- Step 1.* Customer will open the Order Entry Form.
- Step 2.* Fill up the order details on the form (Cust Id, Order type, qty, price, etc.) and submit the form.
- Step 3.* After submitting the order, the Order Controller will check if the customer has enough credit or stock balance in his/her account.
- Step 4.* If the balance is less/nil, stock trading order will be cancelled.
- Step 5.* If the balance is sufficient and if the order is for buying the stock, then hold the account till the order is executed.
- Step 6.* All the buying and selling orders will be added into the Order DB table for further processing and order confirmation is sent to the customer.
- Step 7.* After the order validation, if the order is valid, it will be added to the database in the Order table and order confirmation message will be sent to the customer.

The sequence diagram is as shown in Figure 9.14.

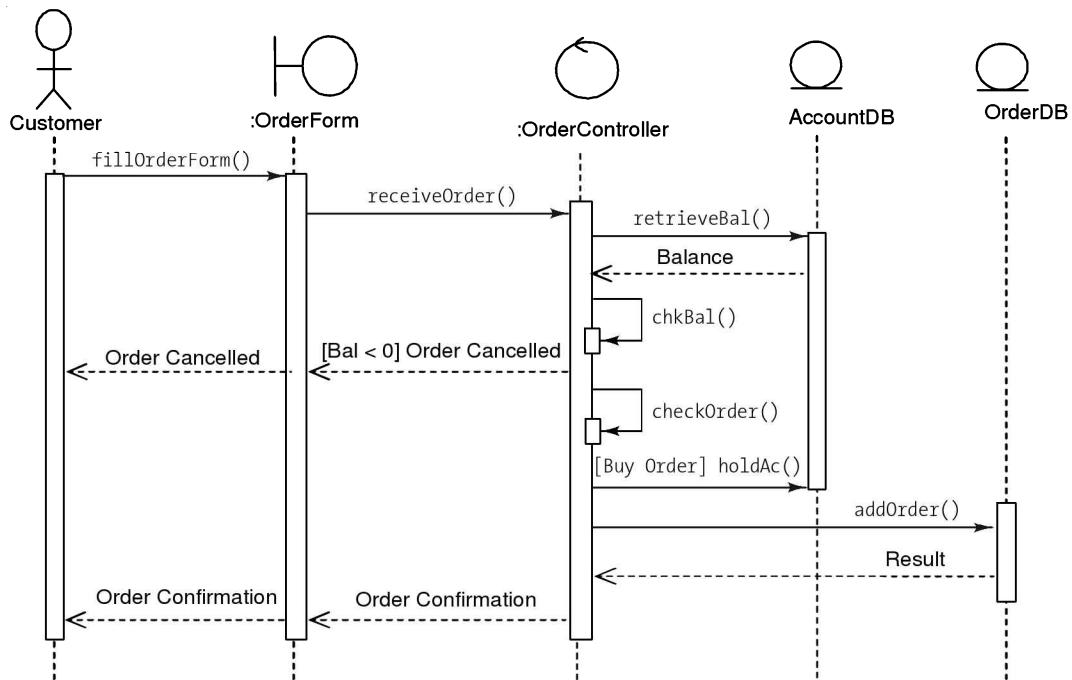


FIGURE 9.14 Sequence diagram for place stock order use-case.

Sequence diagram for process placed orders

The use-case *process placed orders* is performed by HTStock for processing the placed orders through the CPL Bank's on-line trading of securities system.

For performing the use-case, ask four questions and answers to those questions will be the objects interacting with each other for processing the placed orders.

1. Who will process the order? (Actor)
HTStock.
2. Through which interface? (Boundary)
HTStock Interface.
3. Who will handle the actor's request? (Control)
Stock Exchange Control.
4. Which object holds customer's orders? (Entity)
Order DB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls as in Table 9.8.

TABLE 9.8 Classes and methods for process placed orders use-case

Class Type	Classes	Methods
Actor	HTStock	<i>Not required to specify in this context.</i>
Boundary	HTStockInterface	getOrders()
Control	StockExchangeControl	receiveOrders() placeInQueue() executeOrder()
Entity	OrderDB	updateOrder() dropOrder()

For transferring transaction data, the sequence flow will be as follows:

1. HTStock is stock exchange system which actually process the stock trade orders.
2. Through the HTStockInterface, buy/sell stock orders will be received by the stock exchange.
3. As the orders are received they are put in a queue for execution.
4. If the price specified by the customer and the market price is matched, the order in queue is executed.
5. Till the market is open for the day, the order queue will be maintained.
6. At the end of the day all unexecuted orders will be dropped and executed orders will be transferred to the bank.

The sequence diagram is as given in Figure 9.15.

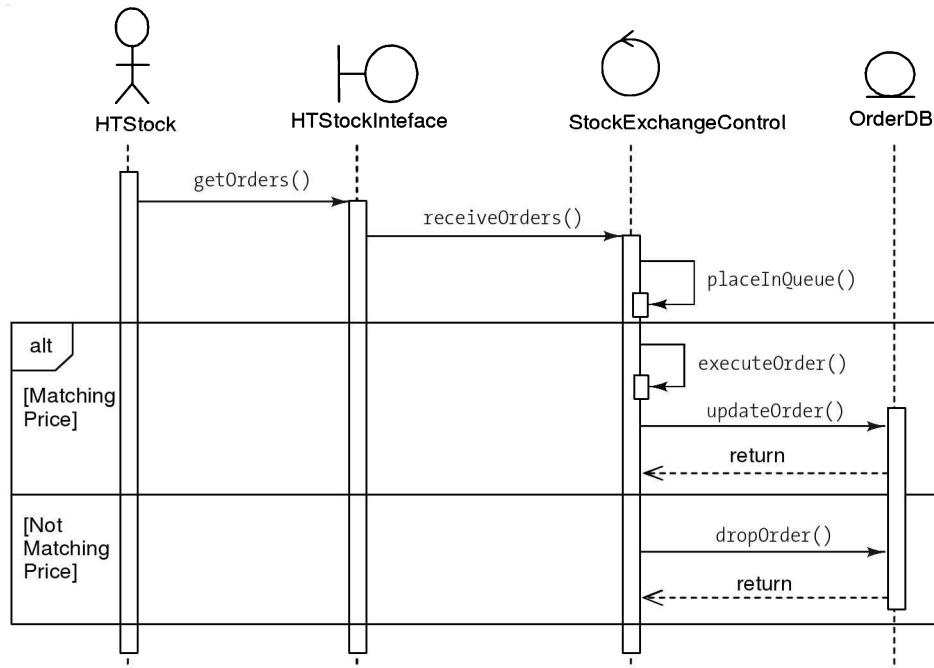


FIGURE 9.15 Sequence diagram for process placed orders use-case.

Sequence diagram for generate monthly statement of transactions use-case

The use-case *generate monthly statement of transactions* is performed by Bank's Securities Account Clerk at the bank for generating the monthly statement of trading transactions using online trading of securities system.

For the generate monthly statement of transactions use-case, ask four questions and answers to those questions will be the objects interacting with each other for the process.

1. Who will generate monthly statement of transactions? (Actor)
Bank's Securities Accounting Clerk.
2. Through which web page (interface)? (Boundary)
StmtGenerationForm.
3. Who will coordinate the process? (Control)
StmtGenerationController.
4. Which object holds loan balance amount? (Entity)
TransactionDB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls as in Table 9.9.

TABLE 9.9 Classes and methods for generate monthly statement of transactions use-case

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Bank's Securities Accounting Clerk	<i>Not required to specify in this context.</i>
Boundary	StmtGenerationForm	setSecuritiesAcNo()
Control	StmtGenerationController	getSecuritiesAcNo() validateSecuritiesAc() generateTransactStmt()
Entity	TransactionDB	retrieveSecuritiesAc() retrieveTransactions()

For generating the loan balance statement, the sequence flow will be as follows:

- Step 1.* Bank's Securities Accounting Clerk will open the StmtGenerationForm and specify the Securities Account No. of which the monthly statement is to be generated and submit it.
- Step 2.* After submitting the Securities Account number, the StmtGenerationController will retrieve the Securities account details and validate the Securities account.
- Step 3.* After the Securities account validation, if the Securities account is valid, retrieve the trade transactions and generate the monthly statement of transactions.
- Step 4.* After the Securities account validation, if the Securities account is not valid, send the message “Invalid Securities Account”.

For steps 3 and 4, the combined fragment is used, where there are two alternatives—Valid SecuritiesAc and Invalid SecuritiesAc. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used in Figure 9.16.

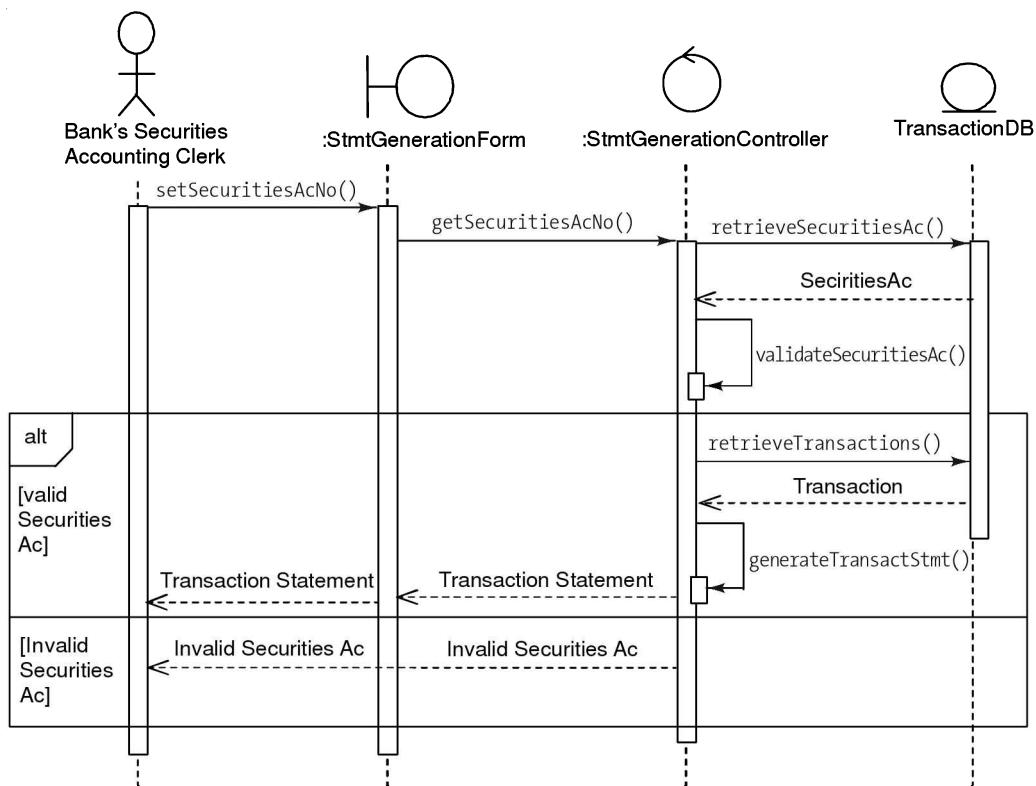


FIGURE 9.16 Sequence diagram for generate monthly statement of transactions use-case.

9.3.2 Collaboration Diagram

Collaboration diagram for registration use-case

The objects, links and methods for the registration use-case are listed in Table 9.10. Figure 9.17 shows the collaboration diagram for the registration use-case.

TABLE 9.10 Objects, links and methods for registration use-case

Objects	Links	Methods (with sequence number)
Customer/Bank Staff	Customer/Bank Staff—RegistrationForm	<code>fillForm()</code> <code>receiveDetails()</code>
RegistrationForm	RegistrationForm—RegistrationController	<code>validateDetails()</code>
RegistrationController	RegistrationController—User	<code>addUser()</code> Registration Confirmation/Cancelled

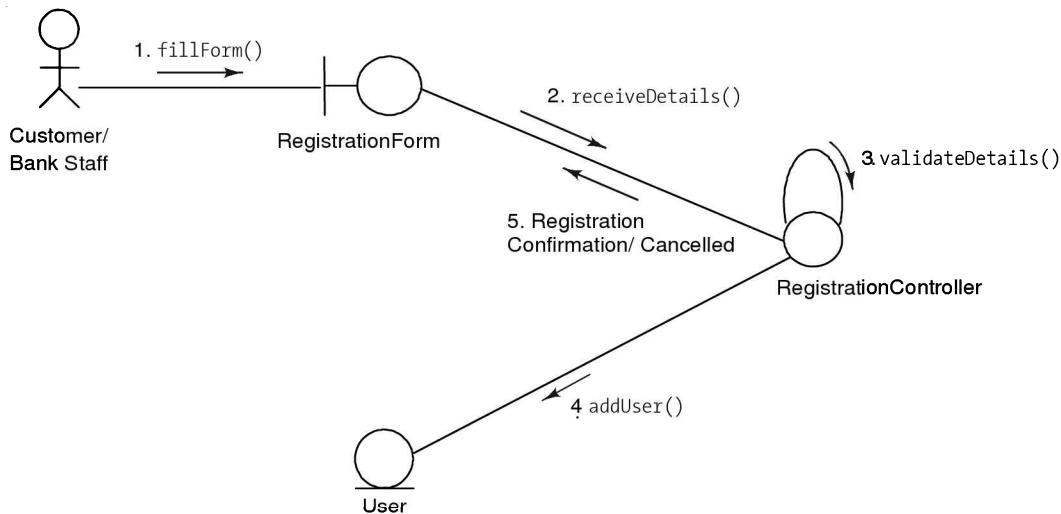


FIGURE 9.17 Collaboration diagram for registration use-case.

Collaboration diagram for login use-case

The objects, links and methods for the login use-case are listed in Table 9.11. The collaboration diagram for the login use-case is shown in Figure 9.18.

TABLE 9.11 Objects, links and methods for login use-case

Objects	Links	Methods (with sequence number)
Customer/Bank Staff	Customer/Bank Staff—LoginForm	1. getUserDetails()
LoginForm	LoginForm—LoginController	2. validateUser()
LoginController	LoginController—User	3. retrieveUserDetails
User		4. UserDetails
		5. validateUser()
		6. Result

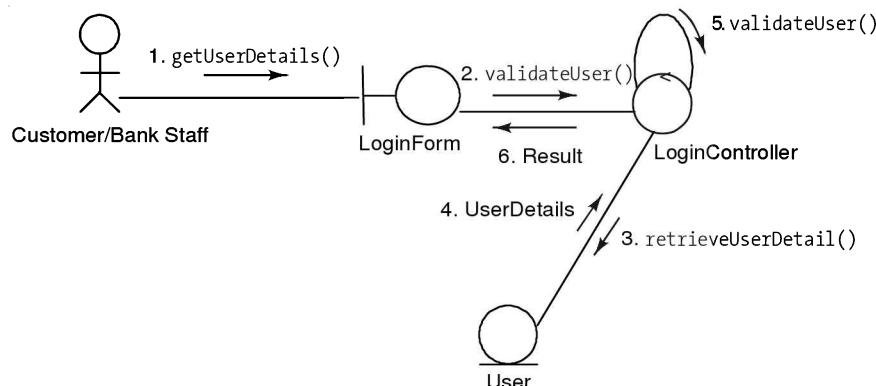


FIGURE 9.18 Collaboration diagram for login use-case.

Collaboration diagram for opening securities amount use-case

Table 9.12 gives objects, links and methods for opening securities account use-case. The collaboration diagram for opening securities account use-case.

TABLE 9.12 Objects links and methods for opening securities account use-case

Objects	Links	Methods (with sequence number)
Customer	Customer—ApplicationForm	1. fillForm()
ApplicationForm	ApplicationForm—Application Controller	2. receiveAppl()
ApplicationController	ApplicationController—Application DB	3. validateForm()
ApplicationDB	ApplicationController—Account DB	4. addApplication()
AccountDB	ApplicationController—SecuritiesAc DB	5. havingAccount()
SecuritiesAcDB		6. createsAccount()
		7. Security AcNo

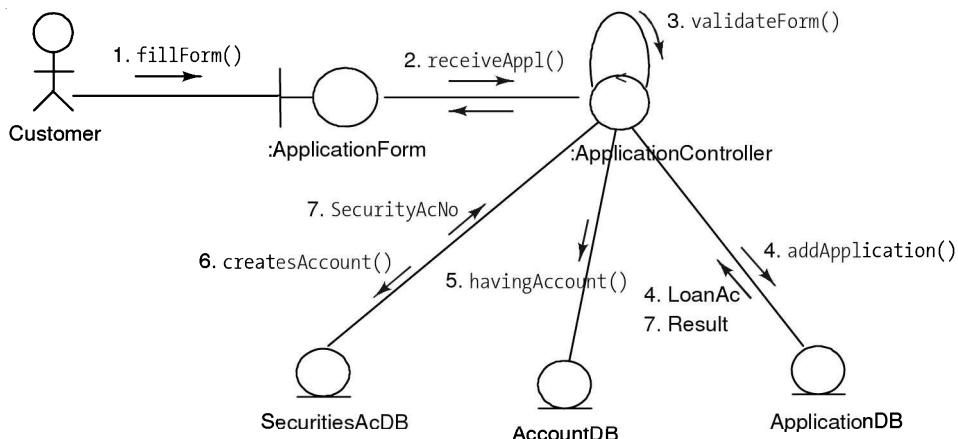


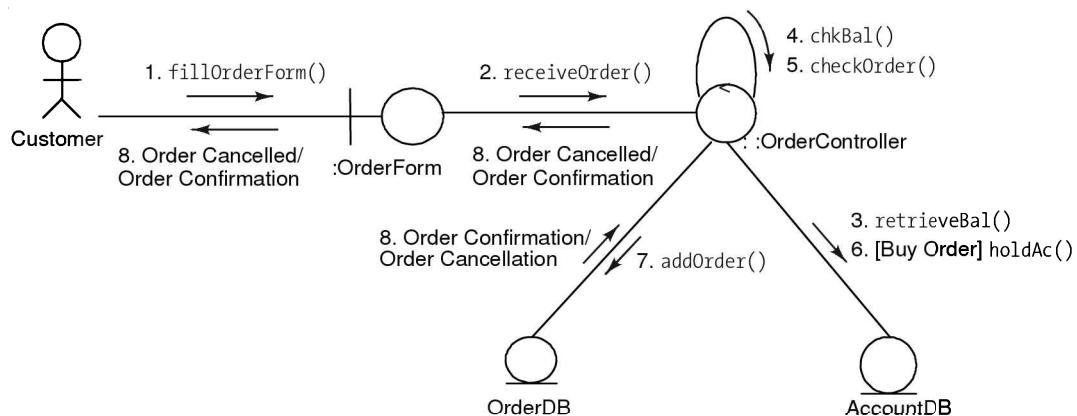
FIGURE 9.19 Collaboration diagram for opening securities account use-case.

Collaboration diagram for place stock order use-case

The objects, links and methods for place stock order use-case are given in Table 9.13. Figure 9.20 shows the collaboration diagram of the place order stock use-case.

Table 9.13 Objects links and methods for place stock order use-case

<i>Objects</i>	<i>Links</i>	<i>Methods (with sequence number)</i>
Customer	Customer—OrderForm	1. fillOrderForm()
OrderForm	OrderForm—OrderController	2. receiveOrder()
OrderController	OrderController—AccountDB	3. retrieveBal()
AccountDB	OrderController—OrderDB	4. chkBal()
OrderDB		5. checkOrder()
		6. holdAc()
		7. addOrder()
		8. Order Confirmation/Order Cancellation

**FIGURE 9.20** Collaboration diagram for place order stock use-case.**Collaboration diagram for process placed orders use-case**

The objects, links and methods for the process placed order use-case are given in Table 9.14. Figure 9.21 shows the collaboration diagram for the process placed order use-case.

TABLE 9.14 Objects, links and methods for process placed order use-case

<i>Objects</i>	<i>Links</i>	<i>Methods (with sequence number)</i>
HTStock	HTStock—HTStockInterface	1. getOrders()
HTStockInterface	HTStockInterface—StockExchangeControl	2. receiveOrders()
StockExchangeControl	StockExchangeControl— OrderDB	3. placeInQueue()
OrderDB		4. executeOrder()
		5. updateOrder()
		6. dropOrder()
		6. return

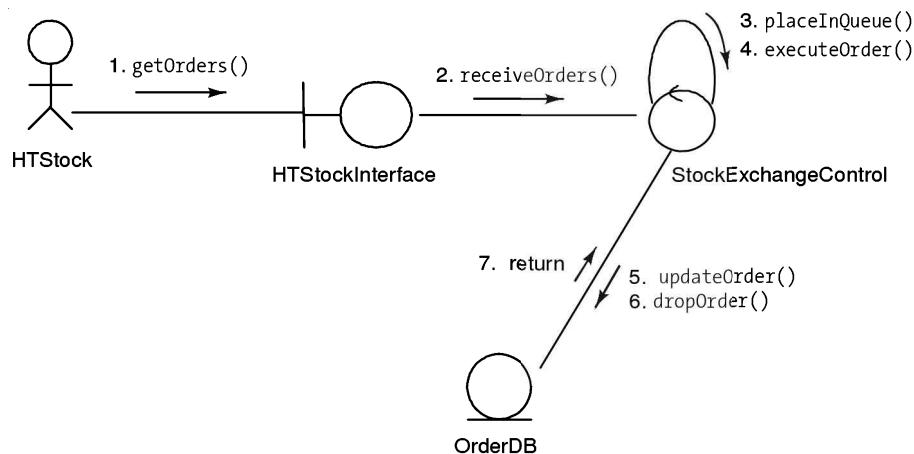


FIGURE 9.21 Collaboration diagram for process placed order use-case.

Collaboration diagram for generate monthly statement of transactions use-case

The objects, links and methods for the generate monthly statement of transaction use-case are given in Table 9.15. Figure 9.22 shows the collaboration diagram for the generate monthly statement of transaction use-case.

TABLE 9.15 Objects, links and methods for generate monthly statement of transactions use-case

Objects	Links	Methods (with sequence number)
Bank's Securities Accounting Clerk	Bank's Securities Accounting Clerk—StmtGenerationForm	1. setSecuritiesAcNo()
StmtGenerationForm	StmtGenerationForm—StmtGenerationController	2. getSecuritiesAcNo()
StmtGenerationController	StmtGenerationController—TransactionDB	3. retrieveSecuritiesAc() 4. SecuritiesAc
TransactionDB		5. validateSecuritiesAc() 6. retrieveTransactions() 7. Transactions 8. generateTransactStmt() 9. Transaction Statement/ Error Msg

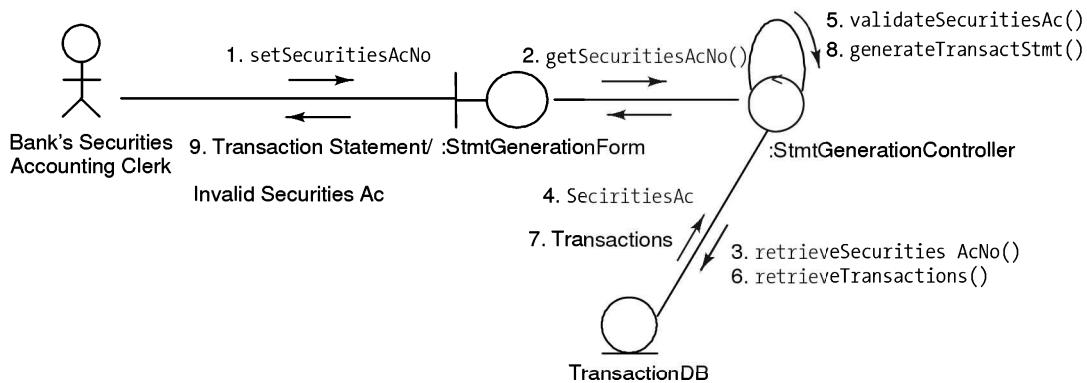


FIGURE 9.22 Collaboration diagram for generate monthly statement of transactions use-case.

9.3.3 Statechart Diagram

In this problem statement, the object for which the statechart diagram is analyzed is

Stock Trade Order

The *Stock Trade Order* object has fixed states throughout its life cycle and there may be some abnormal exits also. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as the complete transaction.

The first state is the waiting state from where the stock trading process starts. As the stock trade order object has finite states throughout its life cycle, the statechart diagram will be one shot life cycle statechart diagram.

Hence there will be one initial state and one or more final state as in Figures 9.23(a) and (b). The intermediate states are shown in Figure 9.23(c). Figure 9.23(d) shows the transitions.



FIGURE 9.23(a) Initial state: Begin order.



FIGURE 9.23(b) Final state: End order.

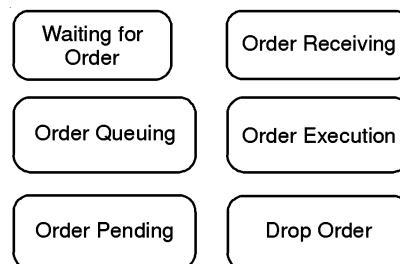


FIGURE 9.23(c) Intermediate states.

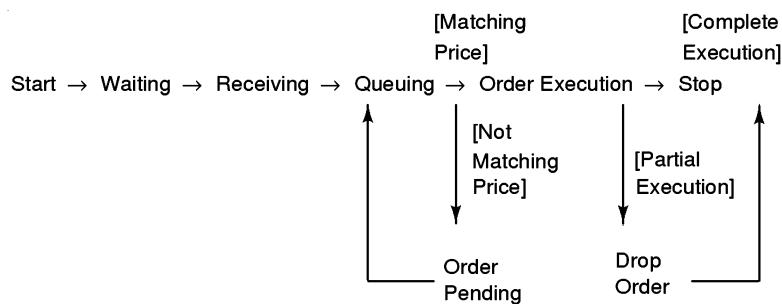


FIGURE 9.23(d) Transitions.

All the transitions in the above process are triggerless transitions which occur on completion of activity during the previous state. The guard condition to enter from the *Receiving* state into *Processing for Completion* state is *confirmed/Not confirmed*. No trigger is required to transit from the state. The complete state transition diagram for the *Stock Trade Order* object is as shown in Figure 9.24.

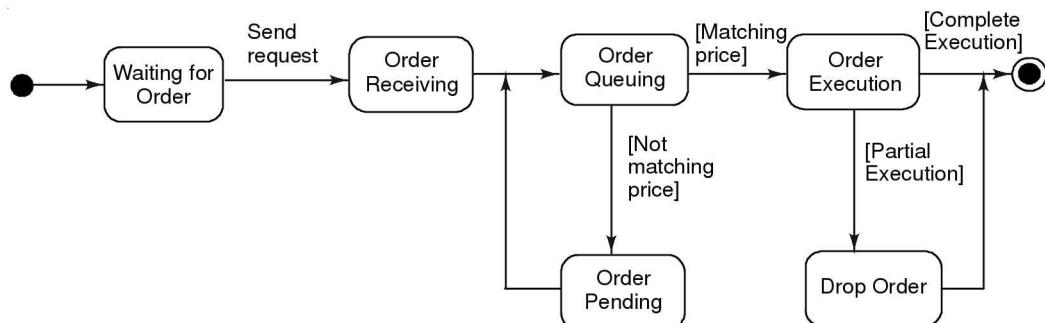


FIGURE 9.24 State transition diagram for stock trade order object.

9.3.4 Activity Diagrams

There are two main processes in the system for which an activity diagram should be drawn, namely

- Opening Securities A/C in Bank
- Trading securities

For drawing an activity diagram for any process, we

1. Find out swimlanes if any. To find swimlanes, see if you can span some activities over different organizational units/places.
2. Find out in which swimlane the online securities trading process begins and where it ends. Those will be the initial and final states.

3. Then, identify activities occurring in each swimlane. Arrange activities in the sequence flow spanning over all the swimlanes.
4. Identify the conditional flow or parallel flow of activities. The parallel flow of activities must converge at a single point using a join bar.
5. During the activities are performed, if any document is generated or used, take it as an object and show the object flow.

Activity diagram for opening securities account in bank

Swimlanes: As the customer interacts with bank's Securities Department to open Securities account with the Bank for trading securities through that account, there are only two swimlanes *Customer* and *Bank's Securities Department* (Figure 9.25). The related activities are illustrated in Figure 9.26.

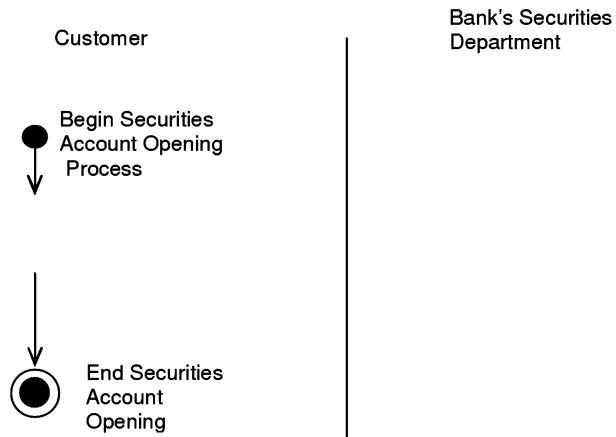


FIGURE 9.25 Swim lanes for opening securities account in bank.

As stated in the problem statement, no parallel flow is identified. One conditional flow occurs after the *Verify form and supplementary documents* activity which proceed to open the security account if the customer already has a savings account in the bank or exits if the customer does not have a savings account in the bank. So the complete activity diagram for the process with the transitions is as shown in Figure 9.27.

Activity diagram for trading securities

Swimlanes: As the customer interacts with bank's Securities Department to buy/sell securities through his/her securities account with the bank and in turn bank's Securities Department interacts with Hong Kong's Stock Exchange HTStock for executing stock order for buying or selling, there are three swimlanes identified as *Customer*, *Bank's Securities Department* and *Stock Exchange HTStock* (Figure 9.28). The related activities are shown in Figure 9.29.

As stated in the problem statement, no parallel flow is identified. One conditional flow occurs after the *Verify username and password* activity which proceeds to place an order for

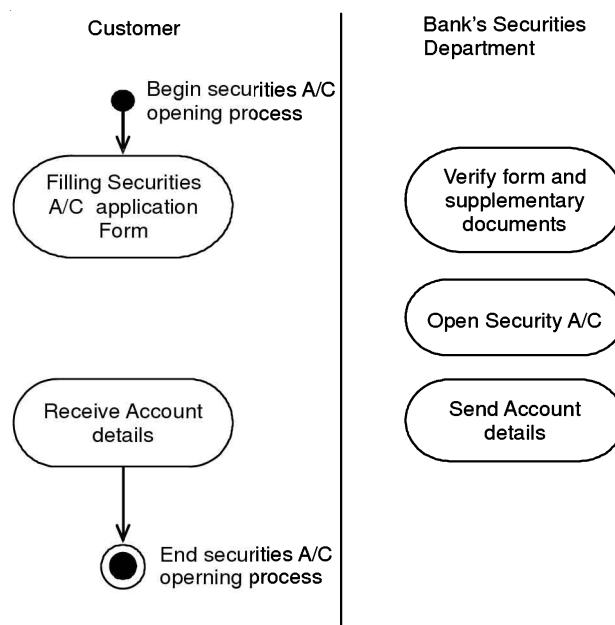


FIGURE 9.26 Activities for opening securities account in bank.

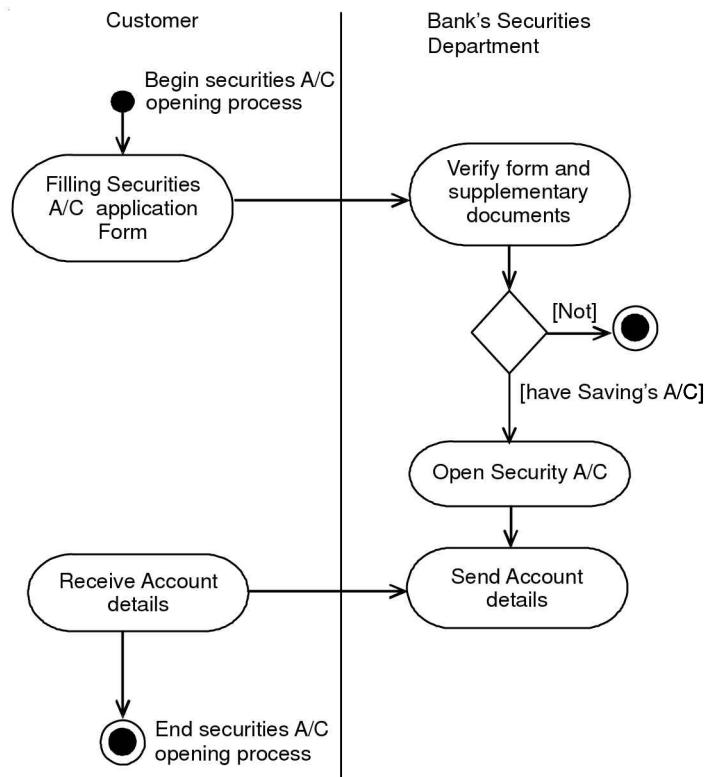


FIGURE 9.27 Activity diagram for opening securities account in bank.

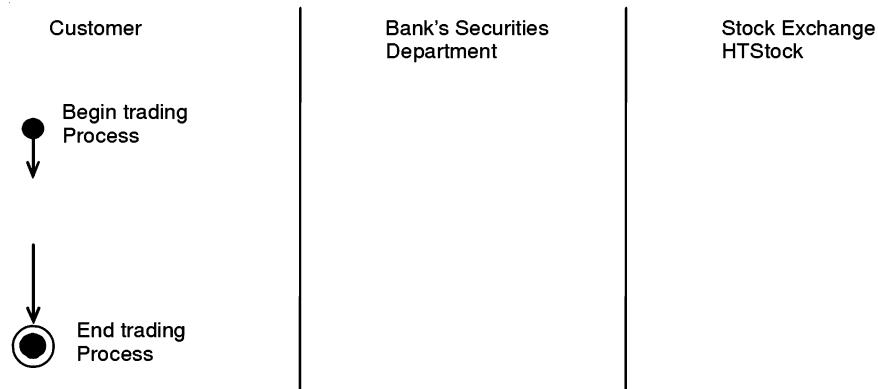


FIGURE 9.28 Swimlanes for trading securities.

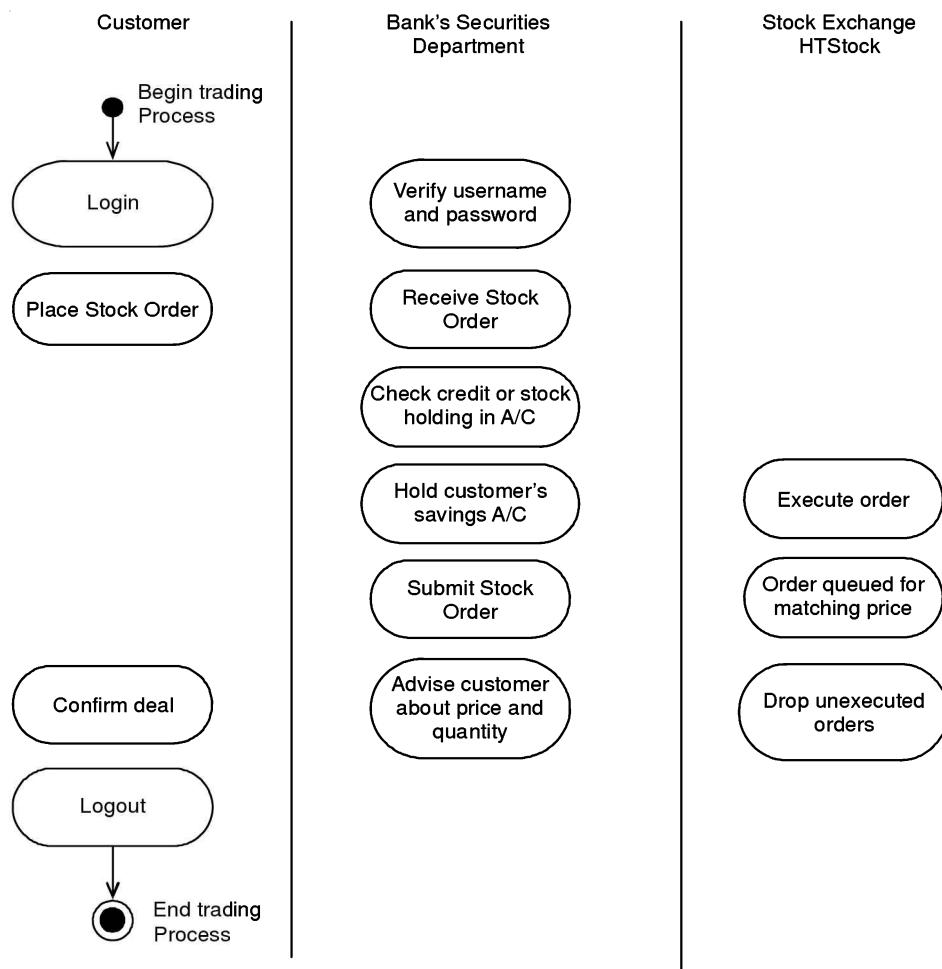


FIGURE 9.29 Activities for trading securities.

buying or selling if the customer is a valid user or else again ask for the valid details to enter if the customer is not a valid user. Another conditional flow occurs after the *Check credit or stock holding in A/C* activity to check if the order is for buying or selling stock.

If the placed order is for buying stock another conditional flow occurs to get hold on credit or stock account of the customer. If the placed order is for selling or after buying and holding account, check if the order is pending or not. If the order is not pending, it gets executed otherwise it will wait for matching price for trade. So the complete activity diagram for the process with the transitions is shown in Figure 9.30.

9.4 IMPLEMENTATION DIAGRAMS: ONLINE TRADING OF SECURITIES

9.4.1 Component Diagram

Component diagrams illustrate the pieces of software that will make up a system. So, major components identified making up this system are as follows:

1. Online Securities Trading Application
2. HTStock system
3. Customer
4. Stock Order
5. Trading Database
6. Security and Persistence

As shown in Figure 9.31, the system is all about performing buying and selling orders of securities online. The main application component in the system is CPL Bank's *Online Securities Trading Application* which depends on the components identified are *Customer* and *Stock Order*, which are implementing corresponding interfaces *ICustomer*, *IStockOrder*.

Persistence and *security* components represent functional components for storing data into the database represented by the *Trading Database* component which is the data store component and managing the access control to the system.

9.4.2 Deployment Diagram

The following observations are made before finalizing the deployment diagram.

- Online Securities Trading System is Internet-based application.
- Online Securities Trading System is based on a multi-tier architecture.
- Hence there will be a database tier holding the trading database, an application tier holding the bank's securities trading application, a client tier holding the client workstation with a browser and one more tier connected to an application tier holding the stock exchange system HTStock.
- Network used is WAN.
- Data communication among all the nodes is through TCP/IP protocol.

In Figure 9.32 Database Server is representing the database layer on which the database server, e.g. MS-SQL Server software and a trading database will be installed.

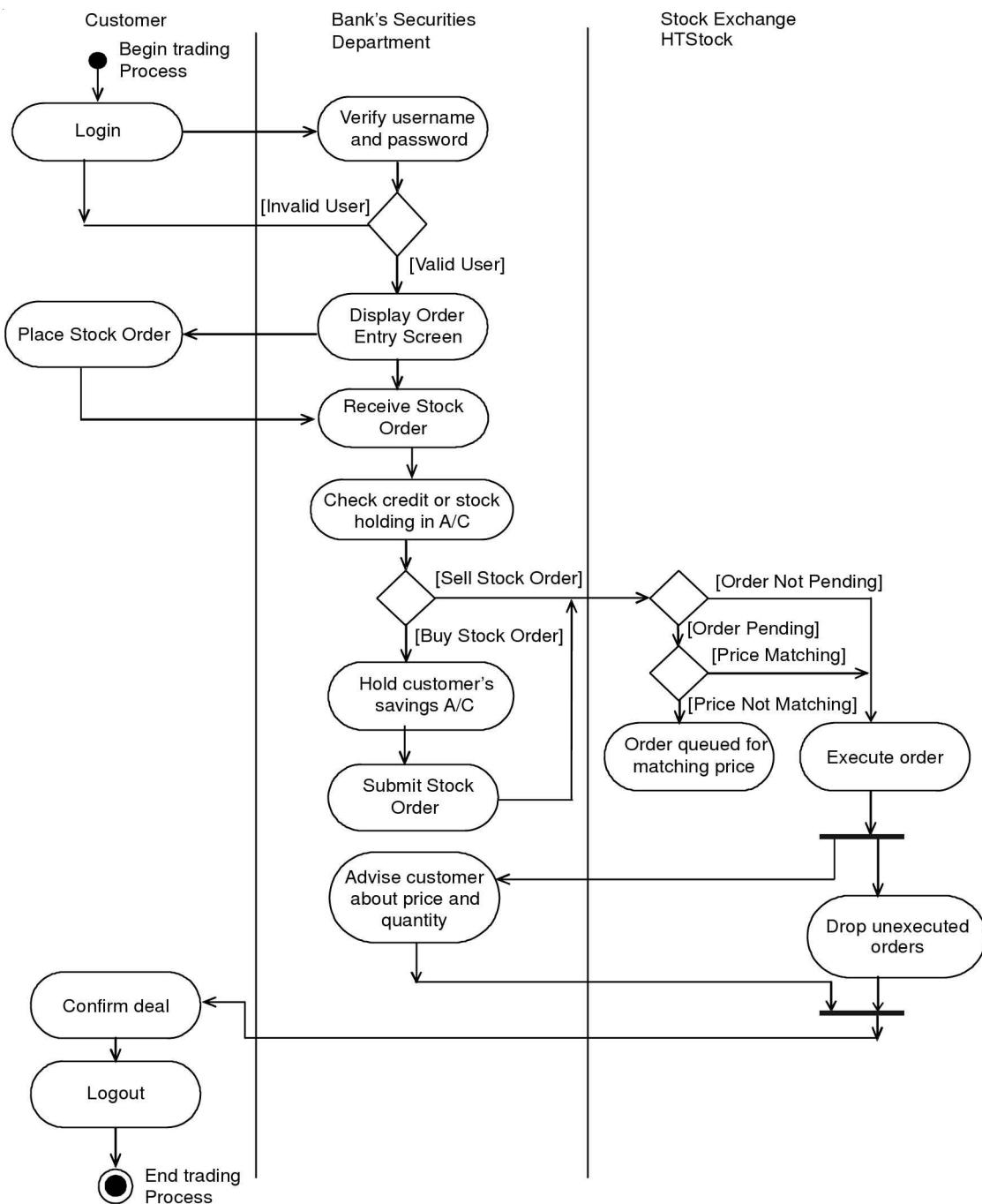


FIGURE 9.30 Activity diagram for trading securities.

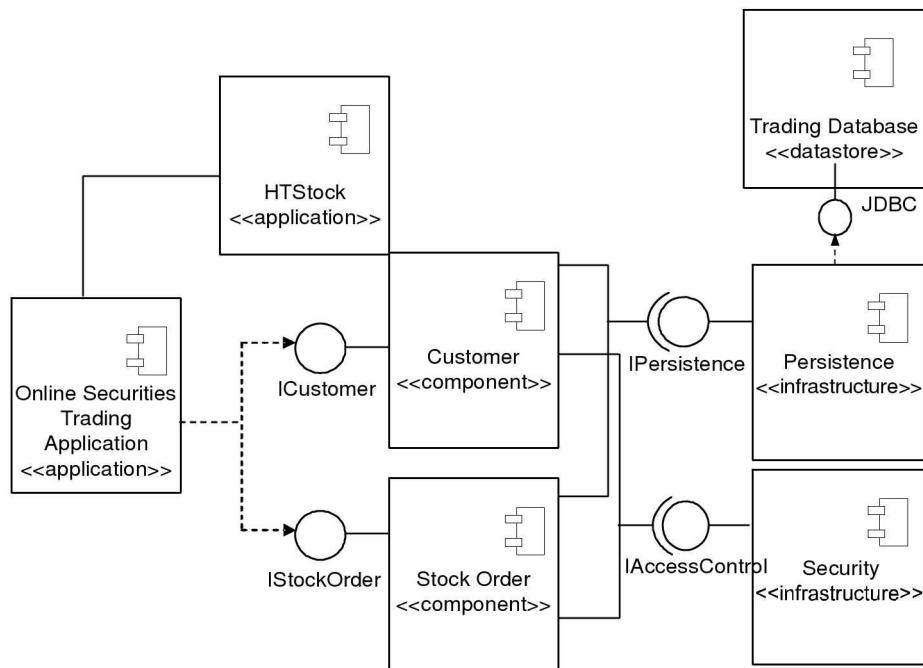


FIGURE 9.31 Component diagram for on-line trading of securities.

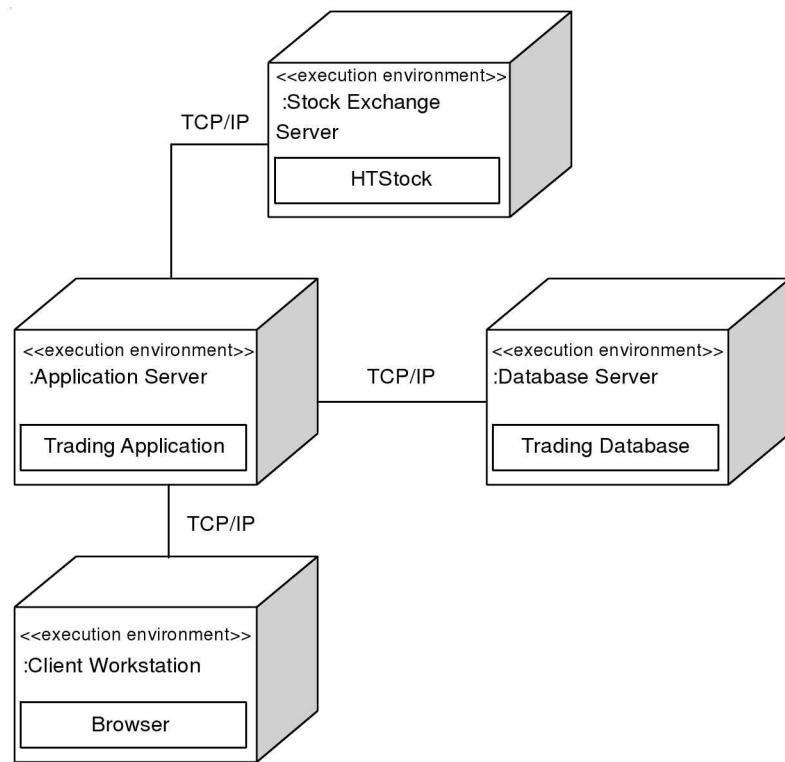


FIGURE 9.32 Deployment diagram for on-line trading of securities.

The application server is representing the application tier on which application server software, e.g. WebSphere application server and online securities trading application (all source code pages, dynamic link libraries etc.) will be deployed. The stock exchange server is representing another tier connected to the application tier on which the stock exchange system—HTStock (HKEx's trading system) will be deployed.

Clients access the application through the browser. Hence on the client tier only the browser is required.

C H A P T E R

10

Credit Card Management System

10.1 PROBLEM STATEMENT: CREDIT CARD MANAGEMENT SYSTEM

The credit card payment service is one of the core businesses in the banking industry. The usage of credit cards is very popular all over the world, even on the virtual community—Internet. ICBI Bank is a global commercial bank. It plans to build an online credit card management system in order to handle a large amount of information in an efficient way and provide better service to their customers. Most commercial banks provide the following major online services to facilitate their credit card business:

- Credit card application;
- On-line credit card payment;
- Special offers to customers, e.g. Cash Dollars; Credit card transaction checking.

With the advance of information technology, ICBI is planning to extend the above services through the Internet. The information systems development team of ICBI is going to conduct the feasibility analysis, system analysis and design of the online Credit Card Management System (CCMS).

10.1.1 Application for Credit Card

In order to obtain an ICBI credit card, customers may make the application by filling up an online application form through the CCMS. In the application, they need to specify the type of credit card that they want to apply, e.g. platinum, gold, or standard. The differences among them are the requirement on the minimum personal annual income, credit limit and annual fee. Information, such as personal contact information, current employment, and financial status, is also required in the application. Once ICBI accepts the application, it will send a confirmation mail to the applicants and state the available date, expiration date and credit limit of the credit card. When the applicants receive the mail with the credit card, they need to call the CCMS to activate the card.

10.1.2 Produce Monthly Statement

At the midnight of statement date in each month, CCMS generates monthly electronic statements for each customer. Paper-based statements are also produced for customers who prefer to

receive them. For each credit card and transactions within the month, CCMS computes the amount of debts and minimum return. The minimum amount of return should be 4% of the total debts. Late payment will be charged by 0.05% of the total debts plus extra \$100. If the payment is returned by auto-pay, the minimum amount of payment will be cleared automatically. If a paper-based statement is used, an extra \$10 service fee will be charged out. Finally, a monthly statement is recorded and posted to the cardholders who selected the paper-based monthly statement.

10.1.3 Customer Information Maintenance

The personal information of individual customers can be maintained online. The maintenance includes check and modification on customer's personal information via CCMS. Cardholder clicks on the "maintain personal information" button on computer screen, the related information will be displayed for the user viewing. Only specific data items are modifiable. The user must login to the system before he/she can begin to maintain the corresponding information.

10.1.4 Credit Card Transaction Recording

The transaction data with respect to customer's payments and purchases taking place randomly are transferred from the Card Brand Corp. to CCMS in a daily basis at each midnight (12:00 p.m.). For each incoming transaction record, the system will calculate the amount of debts and cash dollars. For every \$100 payment, ICBI will reward \$1 cash dollar to the customer. If the payment is specified to use cash dollars during purchase, the amount of its spending will be deducted.

10.1.5 Transaction and Statement Information Checking

CCMS allows customers to check their credit card transactions and monthly statement information online after login to their accounts. The user must specify a period (1 month ago, 2 months ago and 3 months ago) and the system displays the transaction and monthly statement information within that month.

For the above application, build an Analysis Model, Design Model and Implementation Model with the following diagrams:

1. Business Process Diagram
2. Use-case Diagram
3. Class Diagram
4. Object Diagram
5. Sequence Diagram
6. Collaboration Diagram
7. Statechart Diagram
8. Activity Diagram
9. Component Diagram
10. Deployment Diagram

10.2 ANALYSIS PHASE DIAGRAMS: CREDIT CARD MANAGEMENT SYSTEM

10.2.1 Business Process Diagram

The business process diagram shows the various flow objects, connecting objects, swimlanes and artifacts which are analyzed from the problem statement. This understanding mainly comes from business domain analysis and is crucial for capturing business intelligence and representing it into computer systems.

Pool and Lanes

There are three identifiable pools as shown in Figure 10.1 within which various activities for the credit card management system are performed.

1. ICBI Bank
2. Customer
3. Card Brand Corp.

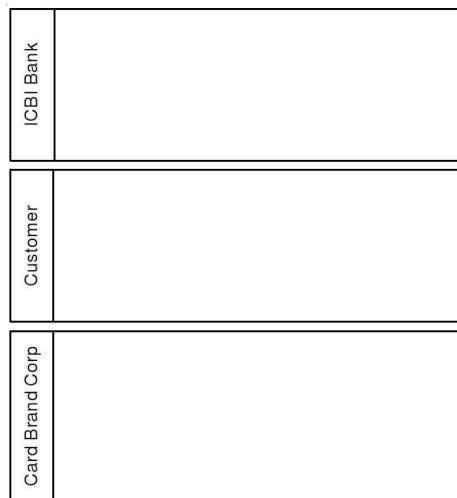


FIGURE 10.1 Pools for credit card management system.

Lanes represent sub-partitions for the objects within a pool. Now within each pool, we try to find out if there is one or more organizational roles within a pool. Here in this case, no pool will have any lane within it.

Also there are five distinct business processes:

1. Application for credit card and activation of credit card
2. Producing monthly statement
3. Customer information maintenance
4. Credit card transaction recording
5. Transaction and statement information checking.

Activities: The activities/tasks performed within the business process of ICBI Bank pool are as shown in Figure 10.2.

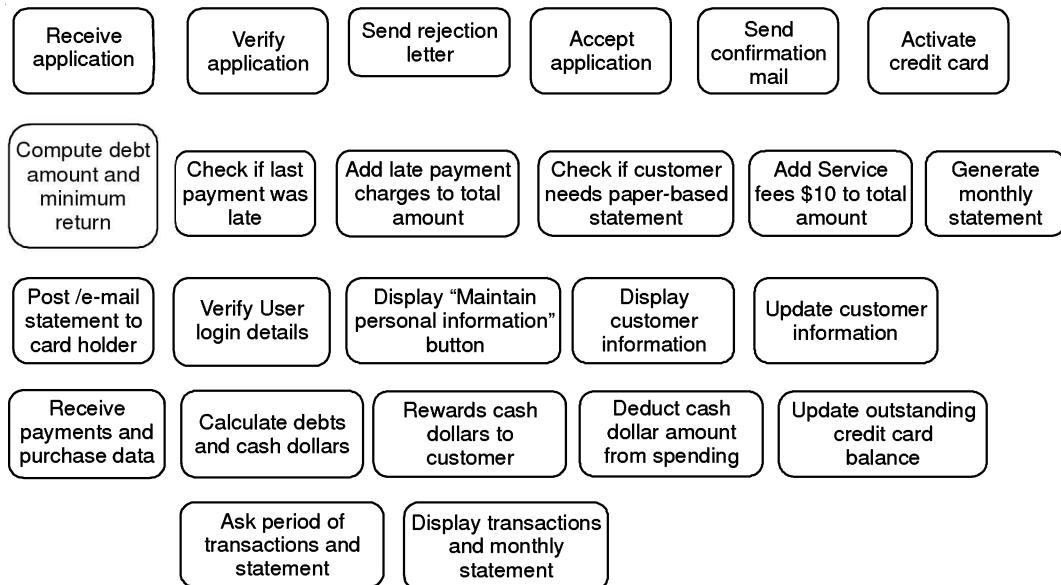


FIGURE 10.2 Activities/tasks for ICBI Bank pool.

The activities/tasks performed within the business process of the customer pool are as shown in Figure 10.3.

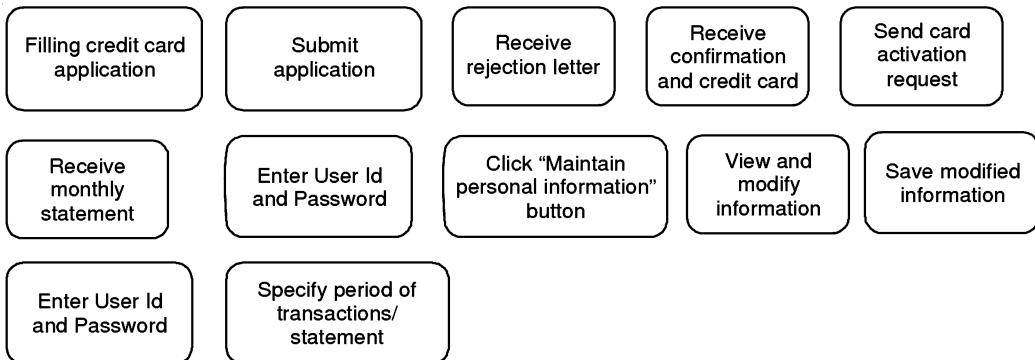


FIGURE 10.3 Activities/tasks for customer pool.

The activities/tasks performed within the business process of the Card Brand Corp pool are in Figure 10.4.

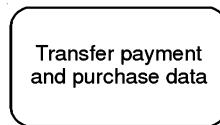


FIGURE 10.4 Activities/tasks for Card Brand Corp pool.

Events: An event is something that *happens* during the course of a business process. There are only start and end events identified in this example as depicted in Figure 10.5.

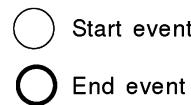


FIGURE 10.5 Events for credit card management system.

Gateways: Gateways are modelling elements that are used to control the sequence flows interaction as they converge and diverge within a process. Gateways of type Exclusive Gateway will be included in the flow to decide the eligibility of the customer for a credit card, to check if the customer is a valid user or not, to check if last payment is late or not, to check if the customer wants paper-based statement or e-statement, to check if transaction is payment or purchase and to check if the customer wants to use cash dollars or not. All these gateways are shown in Figure 10.6.

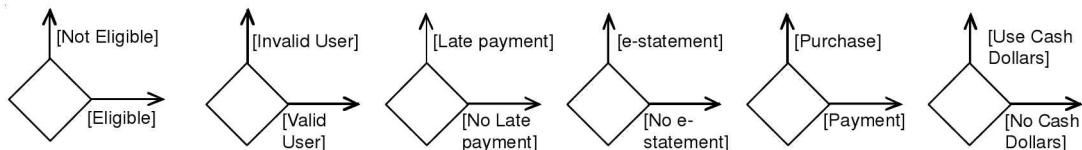


FIGURE 10.6 Gateways representing various conditions in credit card management system.

Artifacts: Artifacts display some additional information in the diagram such as input to activities, output from activities (data objects), and grouping of related process objects (groups), textual comments about events, activities and gateways (annotations). In this example, the only artifacts are data objects which are as shown in Figure 10.7.

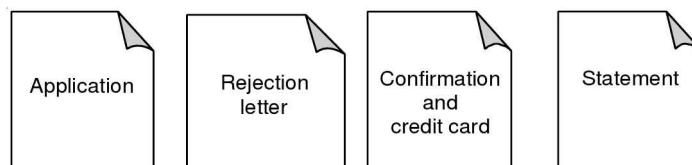


FIGURE 10.7 Data objects for credit card management system.

The business process diagrams are shown in Figures 10.8–10.12.

The business process diagram for application for credit card and activation of credit card is shown in Figure 10.8.

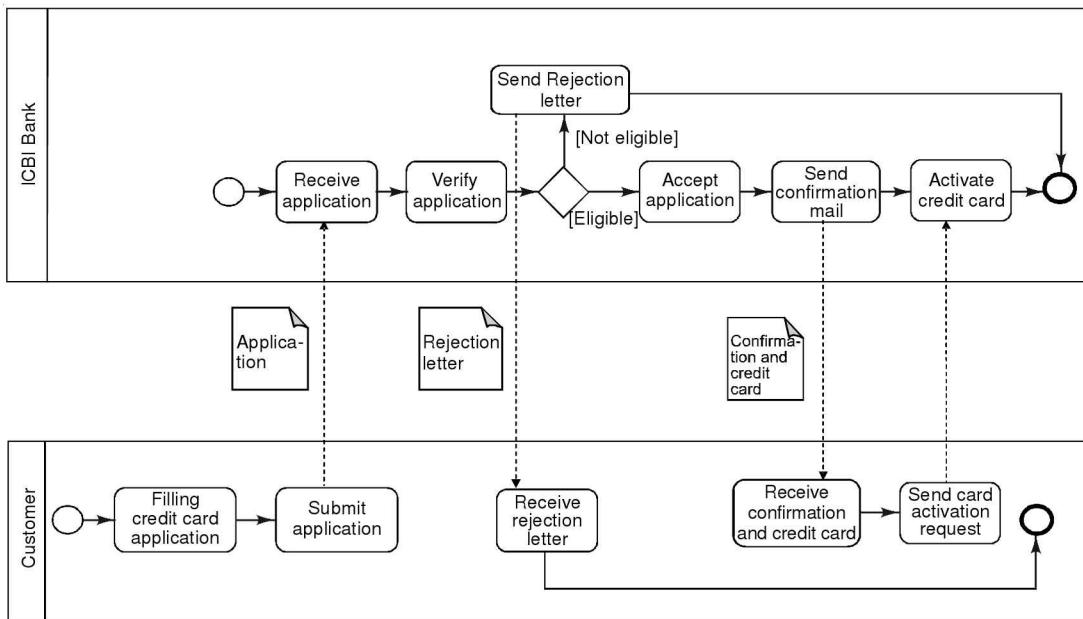


FIGURE 10.8 Business process diagram for application and activation of credit card.

The business process diagram for producing monthly statement is shown in Figure 10.9.

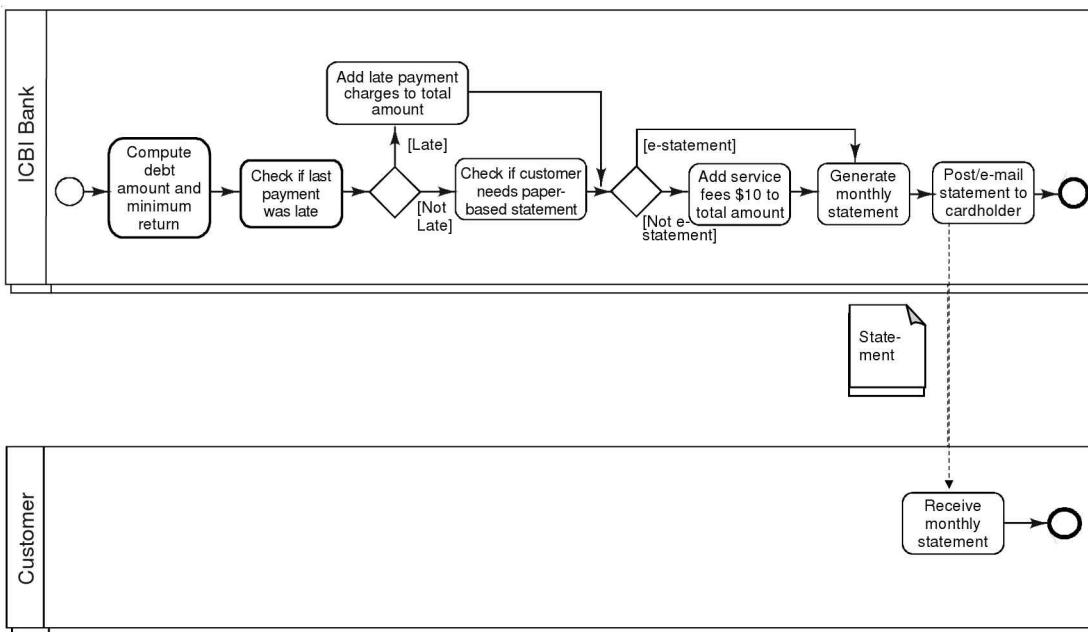


FIGURE 10.9 Business process diagram for producing monthly statement.

The business process diagram for customer information maintenance is given in Figure 10.10.

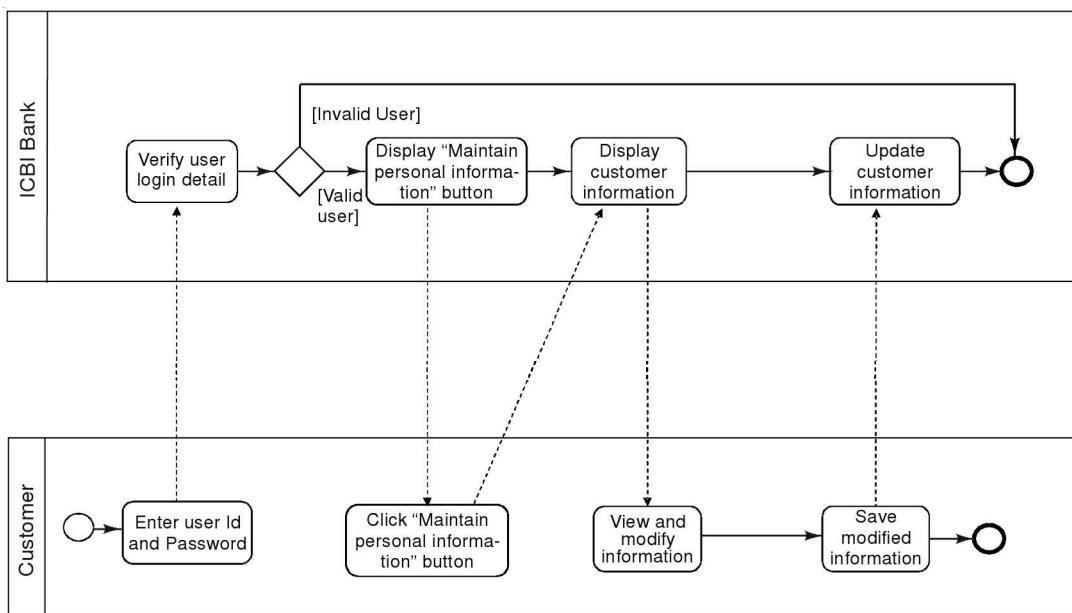


FIGURE 10.10 Business process diagram for customer information maintenance.

The business process diagram for credit card transaction recording is shown in Figure 10.11.

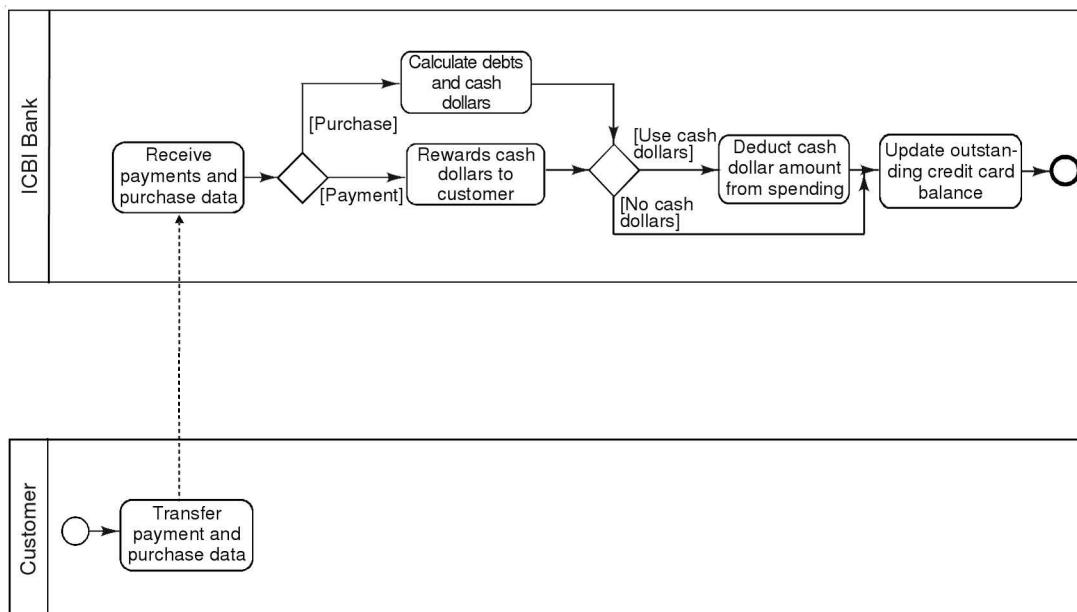


FIGURE 10.11 Business process diagram for credit card transaction recording.

The business process diagram for transaction and statement information checking is shown in Figure 10.12.

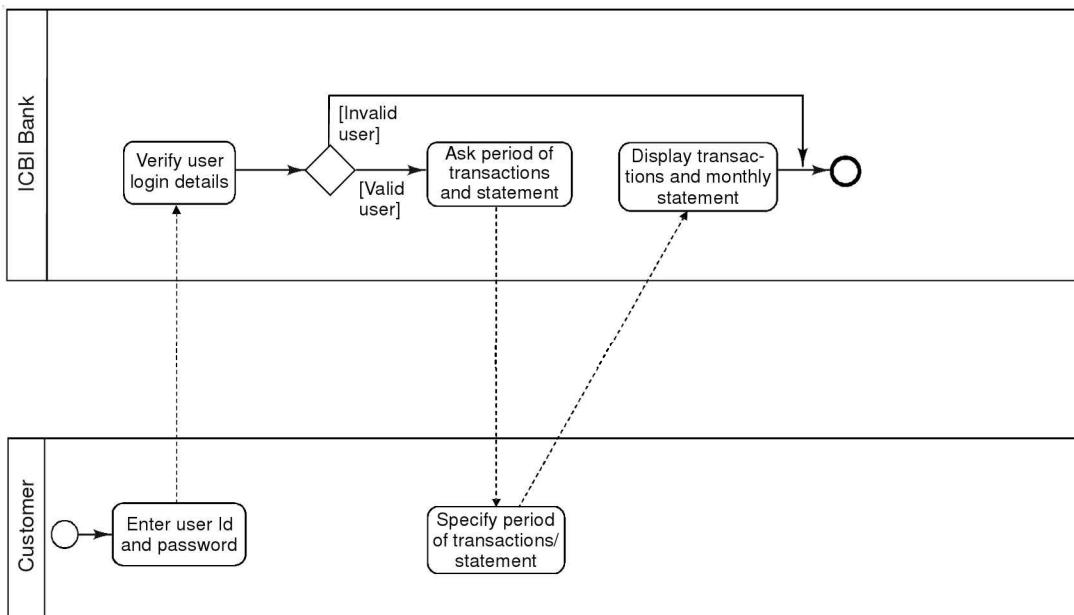


FIGURE 10.12 Business process diagram for transaction and statement information checking.

10.2.2 Use-Case Diagram

There are three actors in the system:

- Customer (Person)
- Card Brand Corp (External System)
- Bank Staff (Person)

Identified actors and their corresponding use-cases are listed in Table 10.1.

First, we will draw an actor-wise use-case diagram and finally a combined use-case diagram representing the whole system with all actors and use-cases performed by them.

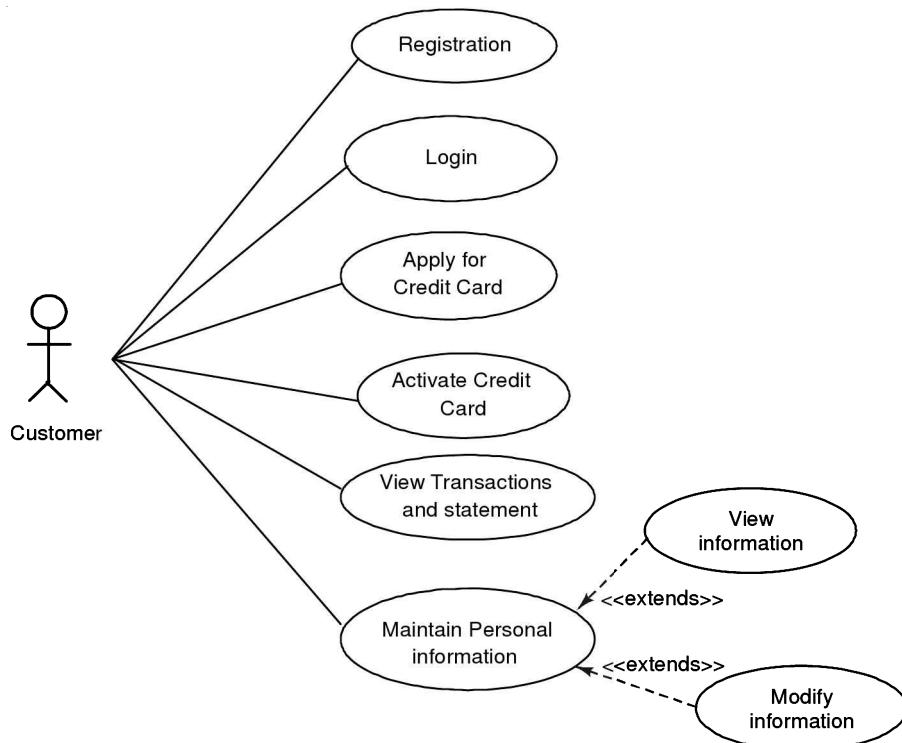
The use-case diagrams are shown in Figures 10.13(a)–(d).

The complete use-case diagram representing the whole credit card management system is shown in Figure 10.13(d).

Important points: In the above use-case diagram, apart from use-cases obvious from the problem statement as found out in Table 10.1, we have added two additional use-cases, *Registration* and *Login*, which are not mentioned in the problem statement, but we have to consider it as everybody has a different role in the system.

TABLE 10.1 Actors and their use-cases for credit card management system

Actors	Use-Cases		
	Base	Include	Extends
Customer	Registration Login Apply for Credit Card Activate Credit Card View Transactions and statement Maintain Personal information		
Card Brand Corp. Bank Staff	Transfer Transaction data Registration Login Verify Application Activate Credit Card		View information Modify information

**FIGURE 10.13(a)** Use-case diagram for customer.

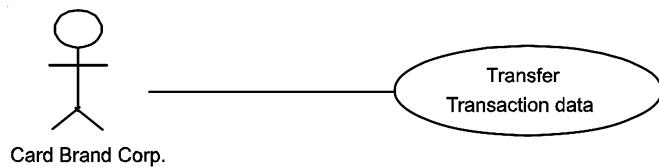


FIGURE 10.13(b) Use-case diagram for card brand corp.

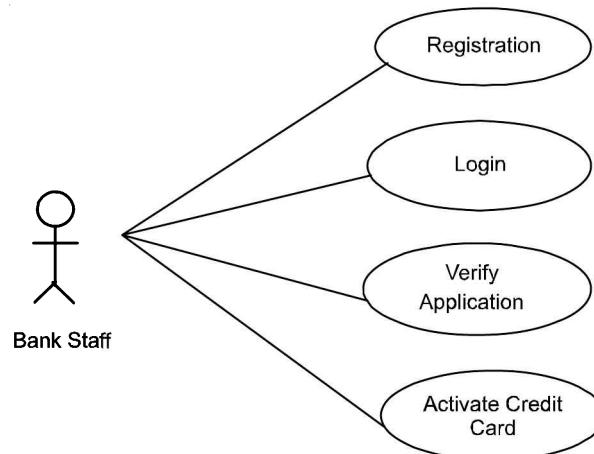


FIGURE 10.13(c) Use-case diagram for bank staff.

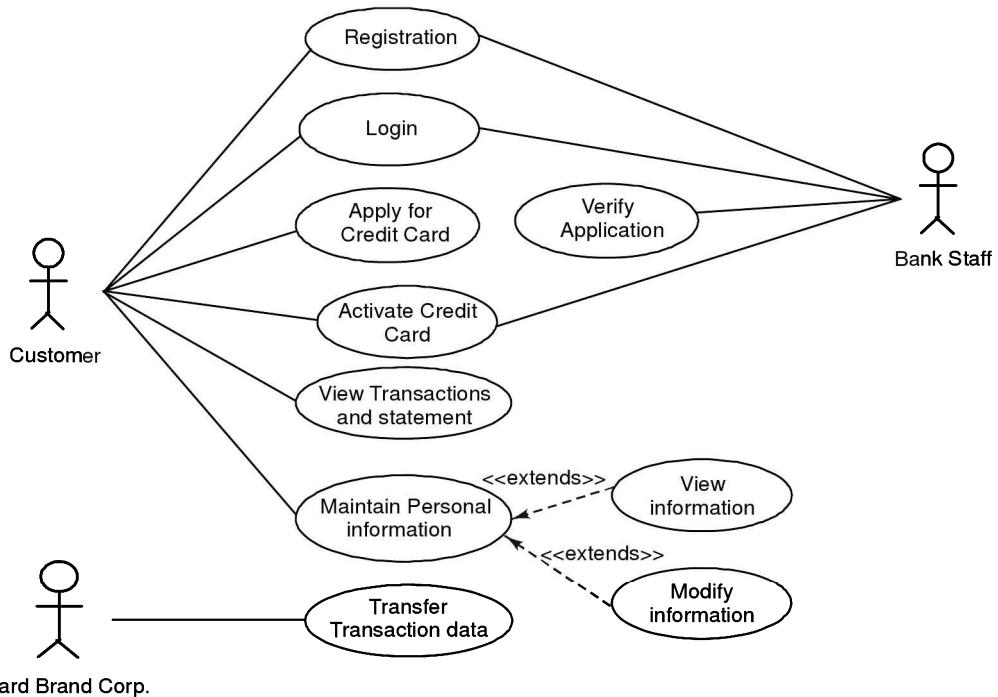


FIGURE 10.13(d) Complete use-case diagram for credit card management system.

10.2.3 Class Diagram

Analyzing the case gives us the following interpretation:

1. Customer fills up a credit card application form.
2. Credit card application form is submitted to the bank.
3. After verification, confirmation along with credit card is sent by bank.
4. Confirmation is received by the customer.
5. Customer gets the credit card.
6. Credit card may be of type platinum, gold or standard.
7. Customer performs transaction through the credit card.
8. Transaction can be either payment or purchase transaction.
9. CardBrandCorp transfers all the transactions in ICBI Bank on a daily basis.
10. Every month the bank generates credit card statement.
11. Customer receives the credit card statement.

Using the interpretation the classes, their relationships and relationship types are as shown in Table 10.2.

TABLE 10.2 Classes and their relationships for credit card management system

Sr. No.	Class 1	Relationship Name	Relationship Type	Class 2
1	Customer	fill	Association	Credit Card Application
2	Credit Card Application	submitted to	Association	Bank
3	Confirmation	sent by	Association	Bank
4	Confirmation	received by	Association	Customer
5	Customer	gets	Association	Credit Card
6	Credit Card	is-of	Generalization	Credit Card Type (Platinum, Gold or Standard)
7	Customer	performs	Association	Transaction
8	Transaction	is-of	Generalization	Transaction Type (Payment or Purchase)
9	CardBrandCorp	transfers	Association	Transaction
10	Bank	generates	Association	Credit Card Statement
11	Customer	receives	Association	Credit Card Statement

During analysis, for statement (7), an association class *Transaction* is found out, as it is interpreted as “Customer uses credit card for performing transaction”. There is an association between *Customer* and *Credit Card* classes. *Transaction* class is representing the association with some additional attributes. The class diagram is shown in Figure 10.14 and the object diagram in Figure 10.15.

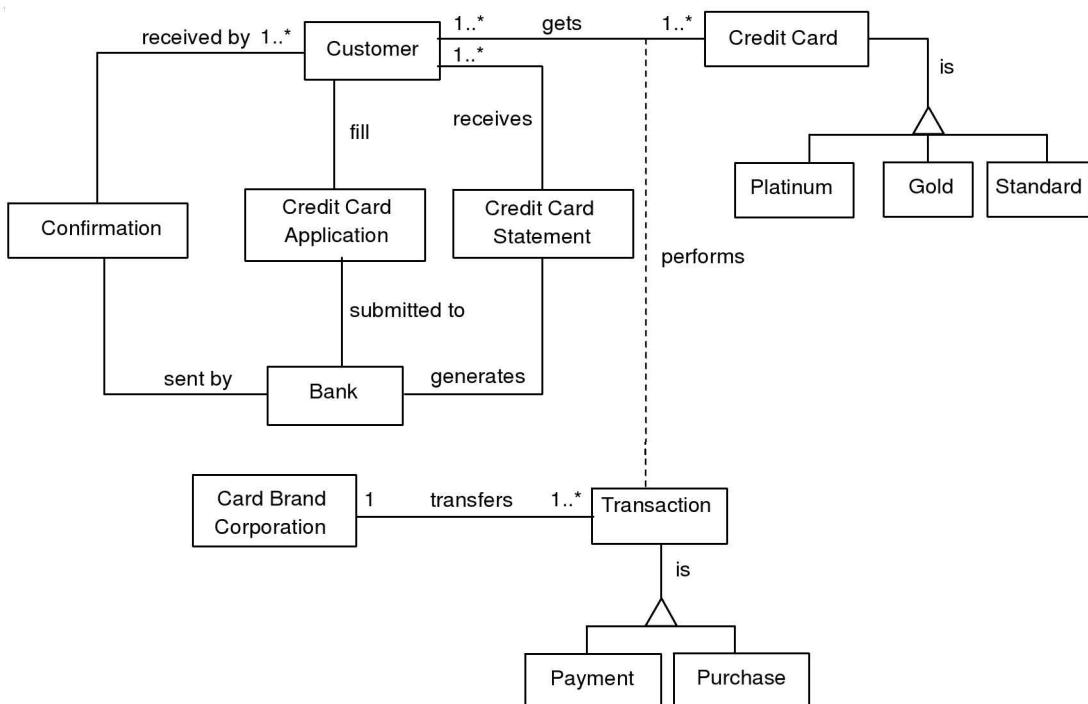


FIGURE 10.14 Class diagram for credit card management system.

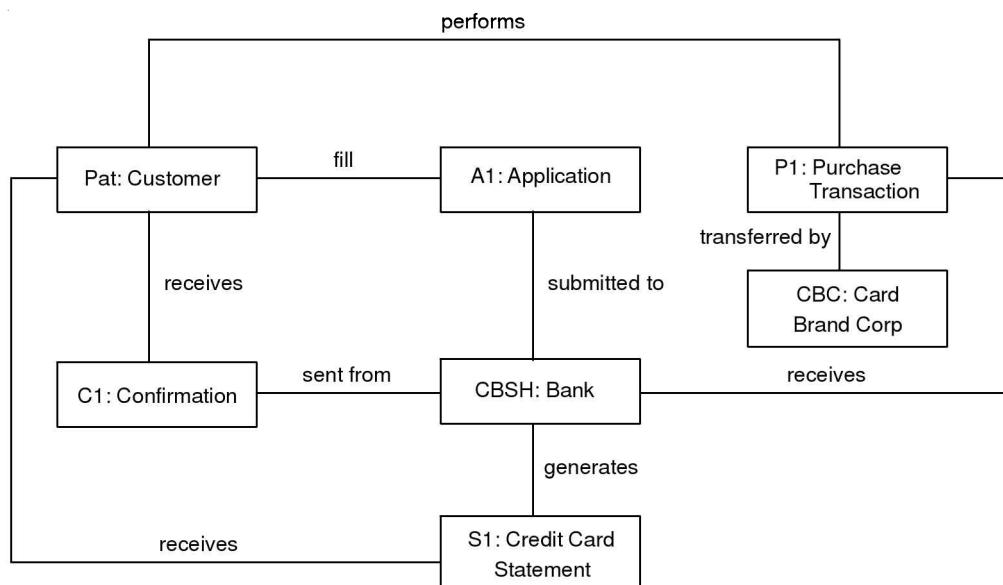


FIGURE 10.15 Object diagram for credit card management system.

10.3 DESIGN PHASE DIAGRAMS: CREDIT CARD MANAGEMENT SYSTEM

10.3.1 Sequence Diagram

A sequence diagram needs to be drawn during the design phase of the system.

During analysis, we have got

- Use-case Diagram
- Class Diagram

Will make use of both these diagrams for design.

1. For each use-case in a use-case diagram, draw one sequence diagram at least. If the use-case is much complex, then there can be more than one sequence diagram.
2. While drawing a sequence diagram from the use-case diagram, draw the sequence diagram only for the base use-cases embedding the flow for <<include>> and <<extend>> use-cases in the same.

In the above case, actors and use-cases identified can be summarized in Table 10.3:

TABLE 10.3 Actors and related use-cases for credit card management system

Sr. No.	Actors	Use Cases		
		Base	Include	Extends
1	Customer	Registration Login Apply for Credit Card Activate Credit Card View Transactions and statement Maintain Personal information		
2	Card Brand Corp.	Transfer Transaction data		
3	Bank Staff	Registration Login Verify Application Activate Credit Card		View information Modify information

During the design, identify four types of objects interacting with each other to perform the use-case as:

1. *Actor objects*: Person, External system or device that initiates the use case, e.g. Customer, Bank Staff, CardBrandCorp.
2. *Boundary objects*: All the interfaces through which actor interact with system, e.g. Login Screen, Application Form, etc.

3. *Controller objects:* All the objects which co-ordinates the task, e.g. Security Manager, etc.
4. *Entity objects:* All domain entities identified during the analysis class diagram, e.g. Users, Credit Card, etc.

Sequence diagram for Registration use-case

The use-case *Registration* is performed by both actors Customer and Bank Staff for registering as authorized users of an online credit card management system.

For the registration use-case, ask four questions and answers to those questions will be the objects interacting with each other for registration process:

1. Who will register? (Actor)
Customer/ Bank Staff.
2. Through which web page (interface)? (Boundary)
RegistrationForm.
3. Who will add the user? (Control)
RegistrationController.
4. Which object holds valid user details? (Entity)
Users.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls (Table 10.4).

TABLE 10.4 Classes and methods for registration use-case

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Customer/Bank Staff	<i>Not required to specify in this context.</i>
Boundary	RegistrationForm	fillForm()
Control	RegistrationController	receiveDetails(), validateDetails()
Entity	Users	addUser()

For online registration, the sequence flow will be as follows:

- Step 1.* Customer/Bank Staff will open the Registration Form.
- Step 2.* Fill up the registration details on the form. (Name, Address, DOB, etc.)
- Step 3.* Click on the Submit button to submit the details.
- Step 4.* Before submitting the details to the server, validation for all the fields on the form (Valid e-mail, phone No., etc.) will be carried out.
- Step 5.* After form validation, the Registration controller object will validate the user, i.e. it will check if the user with the same details already exists or not or any other validation.
- Step 6.* After user validation, if the user is valid, it will be added to the database in the user table and registration confirmation message will be sent to the user.
- Step 7.* After user validation, if the user is not valid, the user will not be added to the database and registration cancellation message will be sent to the user.

For steps 6 and 7, combined fragment is used, where there are two alternatives—Valid User and Invalid User. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 10.16.

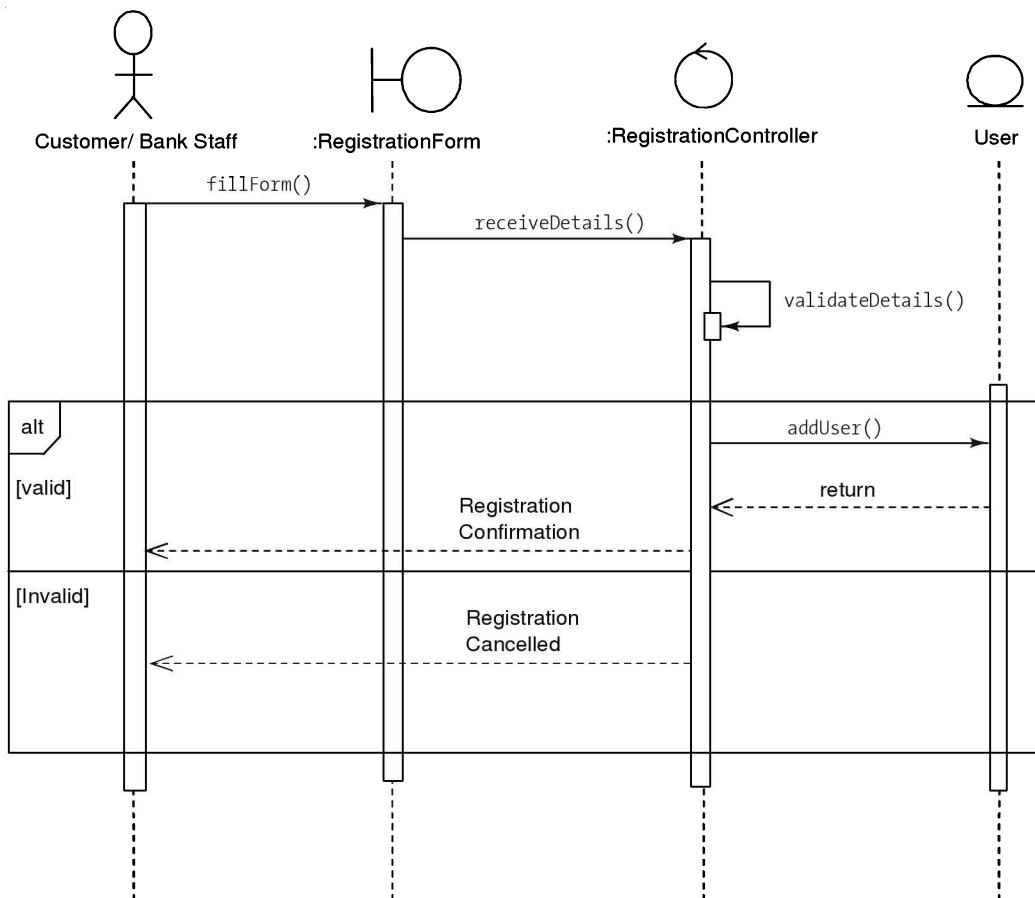


FIGURE 10.16 Sequence diagram for registration use-case.

Sequence diagram for Login

The use-case *Login* is performed by both actors Customer and Bank Staff for an authorized access to the online credit card management system.

For the login use-case, ask four questions and answers to those questions will be the objects interacting with each other for the login process.

1. Who will login? (Actor)
Customer/Bank Staff.
2. Through which web page (interface)? (Boundary)
LoginForm.

3. Who will validate the user? (Control)
LoginController.
4. Which object hold valid user details? (Entity)
Users.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them.

Message communication occurs through method calls (Table 10.5).

TABLE 10.5 Classes and methods for login use-case

Class Type	Classes	Methods
Actor	Customer/Bank Staff	<i>Not required to specify in this context.</i>
Boundary	LoginForm	getUserDetails()
Control	LoginController	validateUser()
Entity	User	retrieveUserDetails

The sequence diagram is as shown in Figure 10.17.

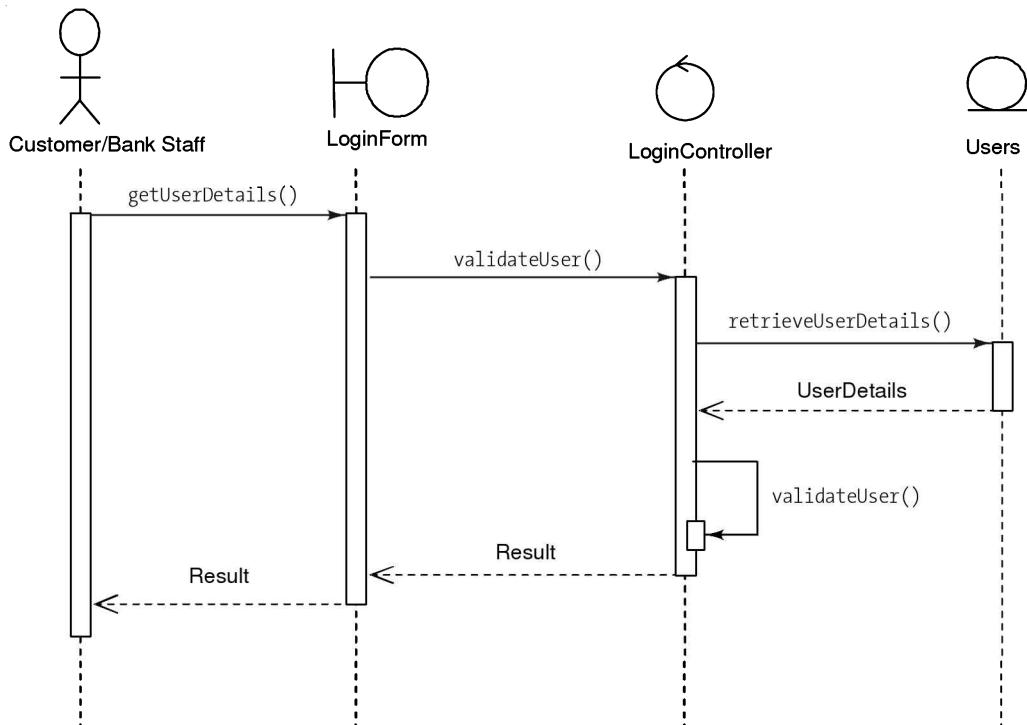


FIGURE 10.17 Sequence diagram for login use-case.

Sequence diagram for applying for credit card and verifying application

The use-case *applying for credit card* is performed by customer for getting credit card from the bank through the online credit card management system.

The use-case *verify application* is performed by Bank Staff after the application is received from the customer by the Bank for the verification of the application form and submitted documents.

A combined sequence diagram for both the use-cases is drawn as both the use-cases are interrelated.

For performing the use-case, ask four questions and answers to those questions will be the objects interacting with each other for applying for credit card and verification process.

1. Who will apply for a credit card and verify application? (Actor)
Customer will apply for a credit card and Bank Staff will verify the application.
2. Through which web page (interface)? (Boundary)
ApplicationForm.
3. Who will coordinate credit card application process? (Control)
CreditCardController.
4. Which object hold credit card data? (Entity)
Application, Credit Card.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them.

Message communication occurs through method calls (Table 10.6).

TABLE 10.6 Classes and methods for applying for credit card and verifying application use-case

Class Type	Classes	Methods
Actor	Customer	<i>Not required to specify in this context.</i>
Boundary	ApplicationForm	fillForm()
Control	CreditCardController	receiveAppl() validateForm() getAppl()
Entity	ApplicationDB	addCCApplication retrieveAppl()
Actor	Bank Staff	verifyAppl()
Entity	CreditCardDB	addCCard()

For *online applying for credit card and verify application* use-cases, the sequence flow will be as follows:

For activating the credit card, the sequence flow will be as follows:

- Step 1. Customer will open the Credit Card Application Form.
- Step 2. Fill up the personal details (Contact info, current employment, income, etc.) and specify the type of credit card (Platinum, Gold, or Standard) on the form and submit it.

- Step 3.* Before sending the application to the database, validation for all the fields on the form (valid e-mail, phone No. etc.) will be carried out.
- Step 4.* After form validation, the Credit Card Controller object will add the valid application into ApplicationDB.
- Step 5.* Bank Staff will then retrieve the applications one by one from the ApplicationDB for the verification purpose.
- Step 6.* After the verification, if the customer is eligible, a new credit card is created and added into the CreditCardDB.
- Step 7.* Then the Bank Staff will send the confirmation along with the new credit card to the Customer.
- Step 8.* If the customer is not eligible, rejection will be sent by the Bank Staff.

For steps 6 and 8, a combined fragment is used, where there are two alternatives—eligible and not eligible customer. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 10.18.

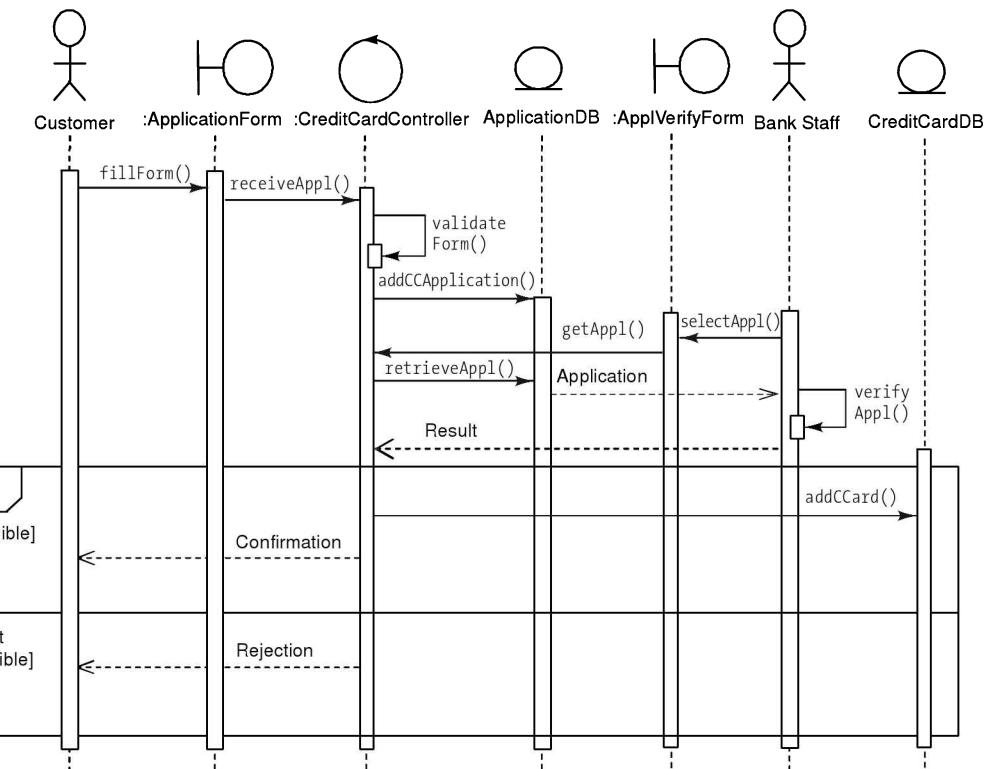


FIGURE 10.18 Sequence diagram for applying for credit card and verifying application use-case.

Sequence diagram for activating credit card

The use-case *activating Credit Card* is performed by Bank Staff for getting the credit card activated through the online credit card management system.

For performing the use-case, ask four questions and answers to those questions will be the objects interacting with each other for activating the credit card process.

1. Who will activate the credit card? (Actor)
Bank Staff.
2. Through which web page (interface)? (Boundary)
ActivationForm.
3. Who will update the credit card status? (Control)
ActivationController.
4. Which object holds valid credit card details? (Entity)
CreditCard.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them.

Message communication occurs through method calls (Table 10.7).

TABLE 10.7 Classes and methods for activating credit card use-case

Class Type	Classes	Methods
Actor	Bank Staff	<i>Not required to specify in this context.</i>
Boundary	ActivationForm	setCreditCardNo() changeStatus()
Control	ActivationController	validateCCardNo() getStatus() validateCard()
Entity	CreditCardDB	retrieveCreditCardDtls() updateStatus()

For activating the credit card, the sequence flow will be as follows:

- Step 1.* Bank Staff will open the ActivationForm and specify the Credit Card Number which is to be activated.
- Step 2.* ActivationController will validate that credit card against that in CreditCardDB.
- Step 3.* If the credit card is a valid credit card, the credit card details will be retrieved from the CreditCardDB for changing its status.
- Step 4.* Bank Staff will then change the status of the credit card as “Activated” and the ActivationController will update the status in database.

The sequence diagram is as shown in Figure 10.19.

Sequence diagram for viewing transactions and statement

The use-case *viewing transactions and statement* is performed by Customer for checking the transactions and monthly statement for the specified period through the online credit card management system.

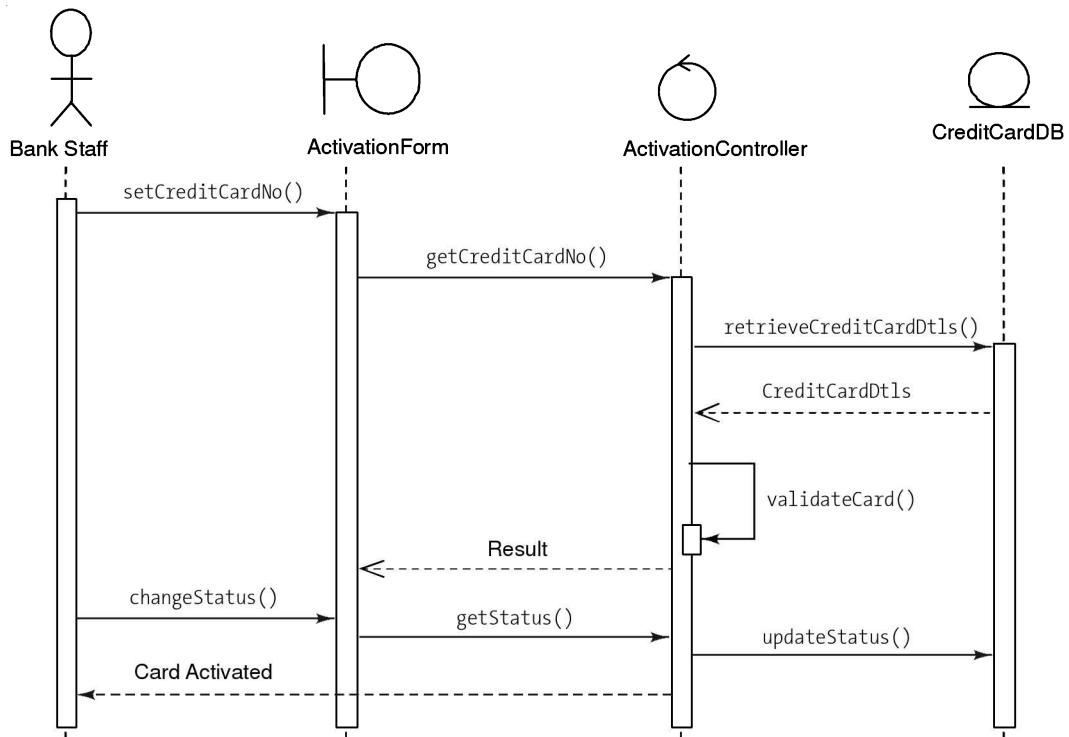


FIGURE 10.19 Sequence diagram for activating credit card use-case.

For performing the use-case, ask four questions and answers to those questions will be the objects interacting with each other for viewing transactions and statement.

1. Who will view transactions and statement? (Actor)
Customer.
1. Through which web page (interface)? (Boundary)
CheckTransactionForm.
2. Who will fetch transaction details for specified period? (Control)
TransactionController.
3. Which object holds Credit Card transactions? (Entity)
TransactionDB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them.

Message communication occurs through method calls (Table 10.8).

TABLE 10.8 Classes and methods for viewing transactions and statement use-case

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Customer	<i>Not required to specify in this context.</i>
Boundary	CheckTransactionForm	setCreditCardNo() setPeriod()
Control	TransactionController	getCreditCardNo() validateCard() getPeriod()
Entity	CreditCardDB	retrieveCreditCardDtls()
Entity	TransactionDB	retrieveTransactions()

For viewing transactions and statement, the sequence flow will be as follows:

- Step 1.* Customer will open the CheckTransactionForm to view the transactions and statement for a particular period.
- Step 2.* He/she will enter the Credit Card No. and the period for which he want to view the transaction.
- Step 3.* TransactionController will check if the credit card is valid or not using the CreditCardDB.
- Step 4.* If the card is valid, it will retrieve the transactions between the specified period from the TransactionDB.
- Step 5.* If the card is invalid, it will send “Invalid Card” message to the Customer.

For steps 4 and 5, a combined fragment is used, where there are two alternatives—Valid Credit Card and Invalid Credit Card. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 10.20.

Sequence diagram for maintaining personal information

The use-case *maintaining personal information* is performed by Customer for modifying personal information and updating in the database through the online credit card management system.

For performing the use-case, ask four questions and answers to those questions will be the objects interacting with each other for maintaining personal information.

1. Who will edit the personal information? (Actor)
Customer.
2. Through which web page (interface)? (Boundary)
editPersonalForm.
3. Who will update the information in database? (Control)
editPersonalController.
4. Which object holds customer's personal information? (Entity)
CustomerDB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them.

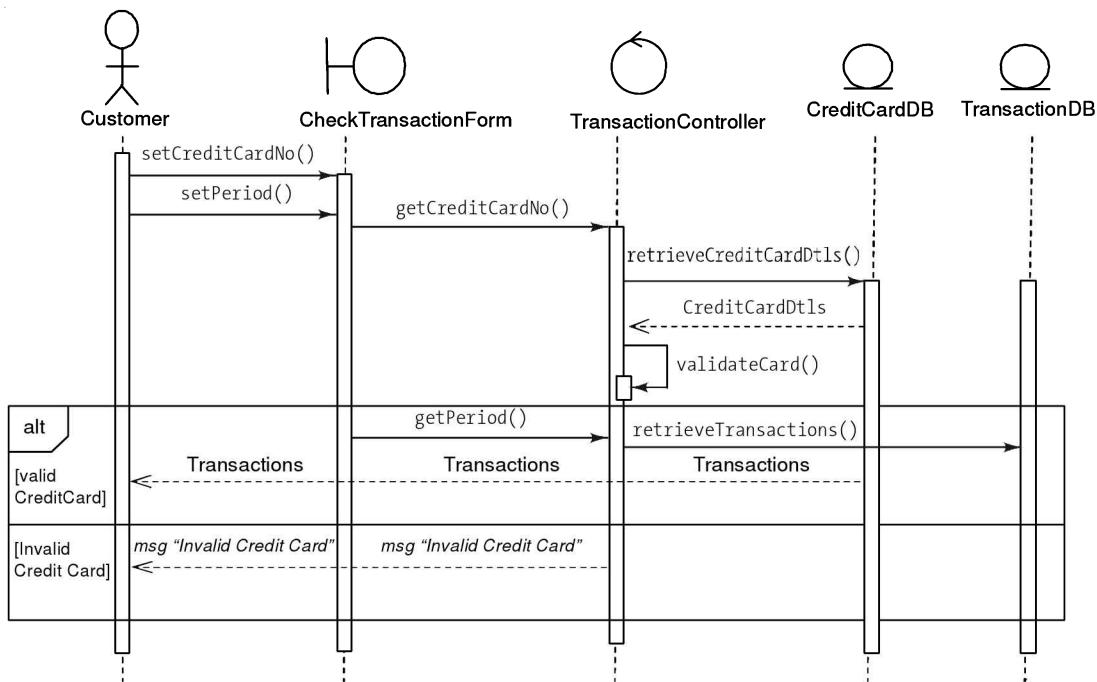


FIGURE 10.20 Sequence diagram for viewing transactions and statement use-case.

Message communication occurs through method calls (Table 10.9).

TABLE 10.9 Classes and methods for maintaining personal information use-case

Class Type	Classes	Methods
Actor	Customer	<i>Not required to specify in this context.</i>
Boundary	editPersonalForm	setCustomerId() modifyCustDtls()
Control	editPersonalController	getCustId() validateCustId() getCustDtls()
Entity	CustomerDB	retrieveCustId() retrieveCustDtls() updateCustDtls()

For maintaining personal information, the sequence flow will be as follows:

- Step 1. Customer will enter his CustomerId to open the editPersonalForm for modifying his/her personal information.
- Step 2. The CustomerId will be validated by the editPersonalController to check if that CustomerId exists and valid or not.

- Step 3.* If the CustomerId is valid, Customer details will be retrieved.
Step 4. If the CustomerId is Invalid, a message “Invalid Customer” will be sent to the Customer.
Step 5. Customer will modify the details retrieved and the CustomerDB will be updated.

For steps 3 and 4, a combined fragment is used, where there are two alternatives—Valid CreditCard and Invalid CreditCard. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 10.21.

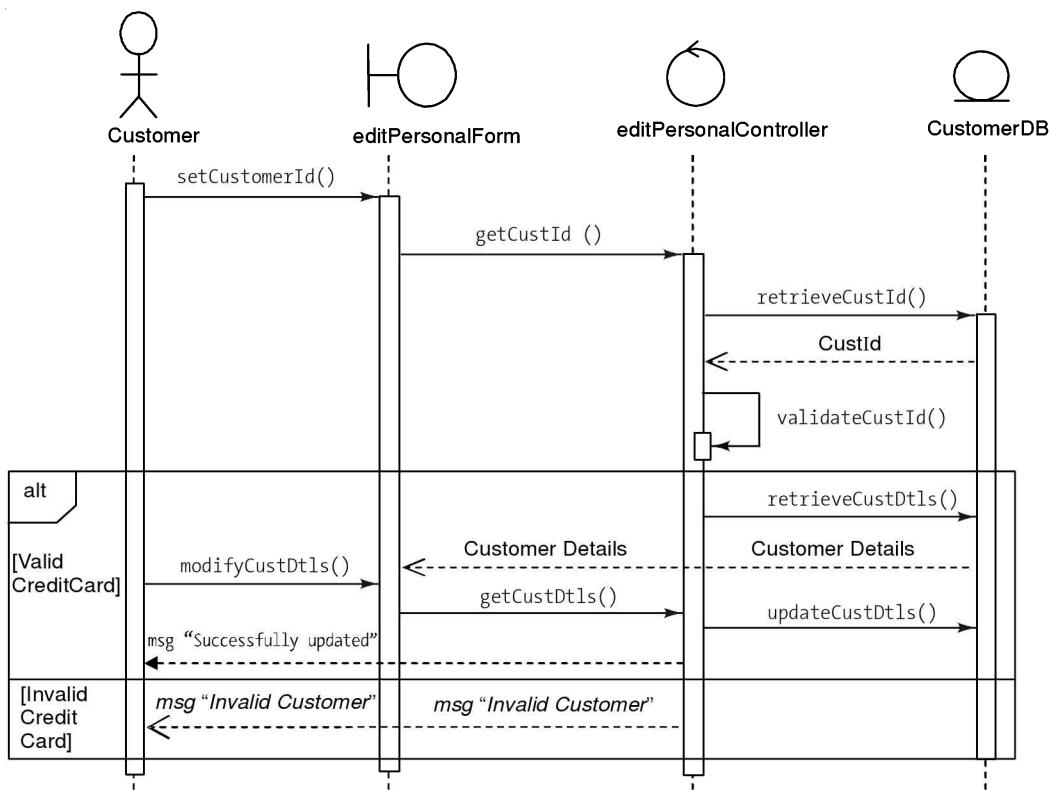


FIGURE 10.21 Sequence diagram for maintaining personal information use-case.

Sequence diagram for transfer transaction data

The use-case *transfer transaction data* is performed by CardBrandCorp for transferring the transaction data of purchases and payments made by customer on credit card, to ICBI Bank on a daily basis.

For performing the use-case, ask four questions and answers to those questions will be the objects interacting with each other for maintaining personal information.

1. Who will transfer the transaction data? (Actor)
CardBrandCorp.

2. Through which interface? (Boundary)
CardBrandTransferInterface.
3. Who will handle the actor's request? (Control)
TransferController.
4. Which object holds customer's transactions information? (Entity)
TransactionDB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them.

Message communication occurs through method calls (Table 10.10).

TABLE 10.10 Classes and methods for transfer transaction data use-case

Class Type	Classes	Methods
Actor	CardBrandCorp	<i>Not required to specify in this context.</i>
Boundary	TransferInterface	setTransactions()
Control	TransferController	receiveTransactions()
Entity	TransactionDB	addTransactions()

For transferring transaction data, the sequence flow will be as follows:

1. CardBrandCorp will automatically transfer the daily transactions at midnight 12.00.
2. TransactionDB will be updated daily.

The sequence diagram is as shown in Figure 10.22.

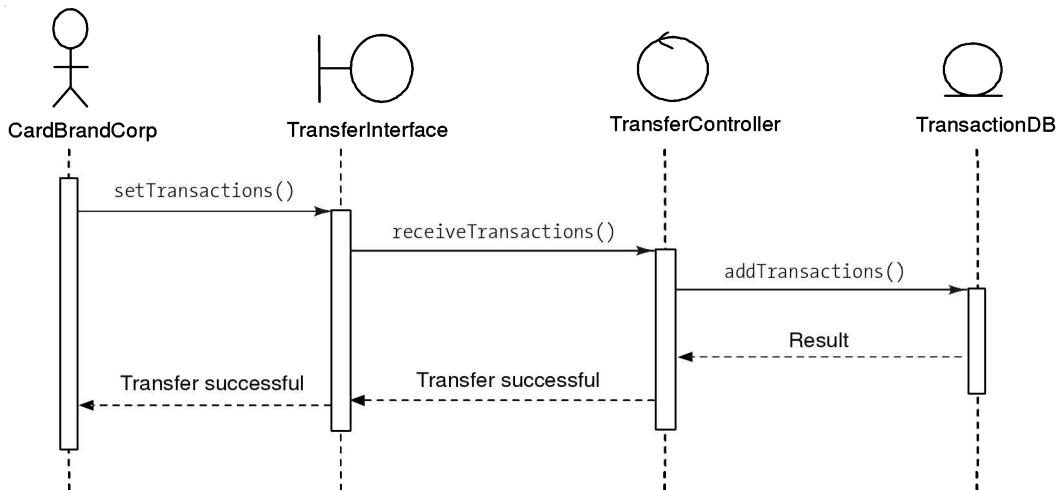


FIGURE 10.22 Sequence diagram for transfer transaction data use-case.

10.3.2 Collaboration Diagram

Collaboration diagram for registration

The objects, links and methods for the registration use-case are listed in Table 10.11. Figure 10.23 shows the collaboration diagram for the registration use-case.

TABLE 10.11 Objects, links and methods for registration use-case

Objects	Links	Methods (with sequence number)
Customer/Bank Staff	Customer/Bank Staff—RegistrationForm	1. fillForm()
RegistrationForm	RegistrationForm—RegistrationController	2. receiveDetails() 3. validateDetails()
RegistrationController	RegistrationController—User	4. addUser()
User		5. Registration Confirmation/Cancelled

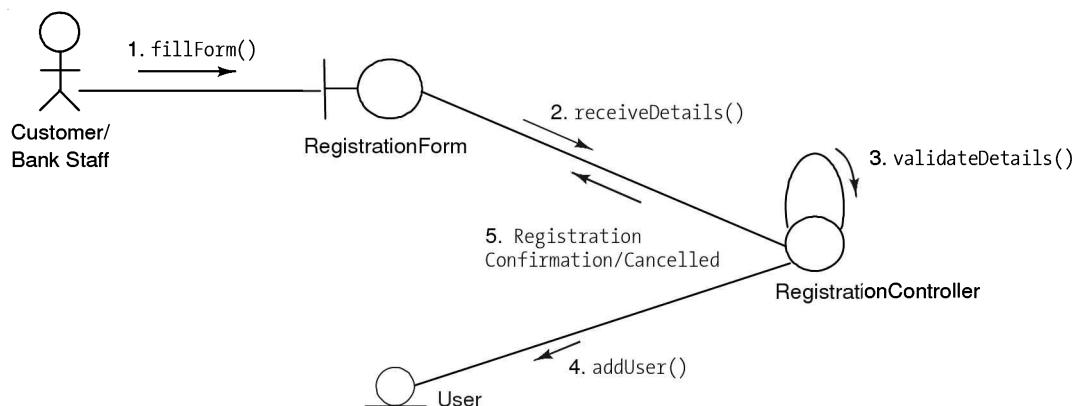


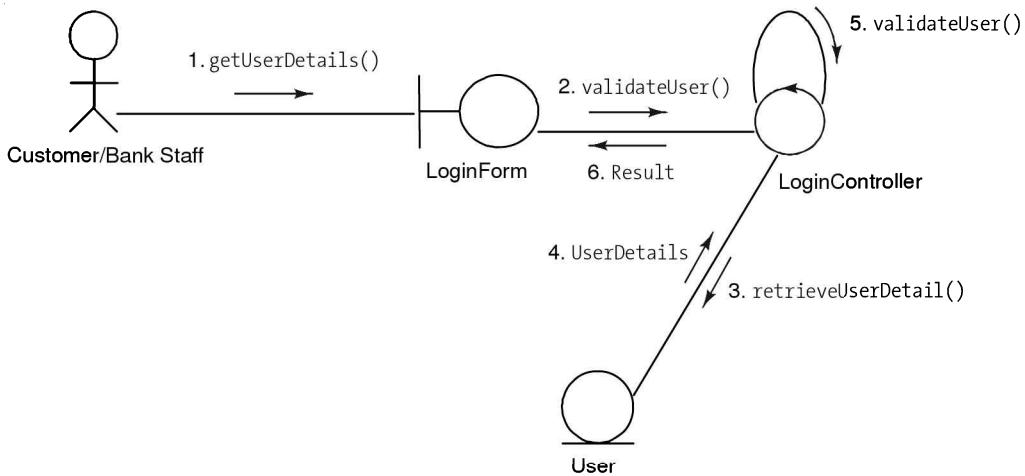
FIGURE 10.23 Collaboration diagram for registration use-case.

Collaboration diagram for login

Table 10.12 lists the objects, links and methods for the login use-case.

TABLE 10.12 Objects, links and methods for login use-case

Objects	Links	Methods (with sequence number)
Customer/ Bank Staff	Customer/Bank Staff—LoginForm	1. getUserDetails()
LoginForm	LoginForm—LoginController	2. validateUser()
LoginController	LoginController—User	3. retrieveUserDetails 4. UserDetails 5. validateUser() 6. Result
User		

**FIGURE 10.24** Collaboration diagram for login use-case.

Collaboration diagram for applying for credit card and verifying application

Table 10.13 lists the objects, links and methods for applying for a credit card and verifying application use-case is listed in Table 10.13. The collaboration diagram for applying for a credit card and verifying application use-case is shown in Figure 10.25.

TABLE 10.13 Objects, links and methods for applying for credit card and verifying application use-case

Objects	Links	Methods (with sequence number)
Customer	Customer—ApplicationForm	1. fillForm()
ApplicationForm	ApplicationForm—CreditCardController	2. receiveAppl()
CreditCardController	CreditCardController—ApplicationDB	3. validateForm()
ApplicationDB	CreditCardController—ApplVerifyForm	4. addCCApplication 5. selectAppl()
Bank Staff	ApplVerifyForm—CreditCardController	6. getAppl()
ApplVerifyForm	CreditCardController—CreditCardDB	7. retrieveAppl() 8. verifyAppl() 9. addCCard() 10. Confirmation/Rejection
CreditCardDB		

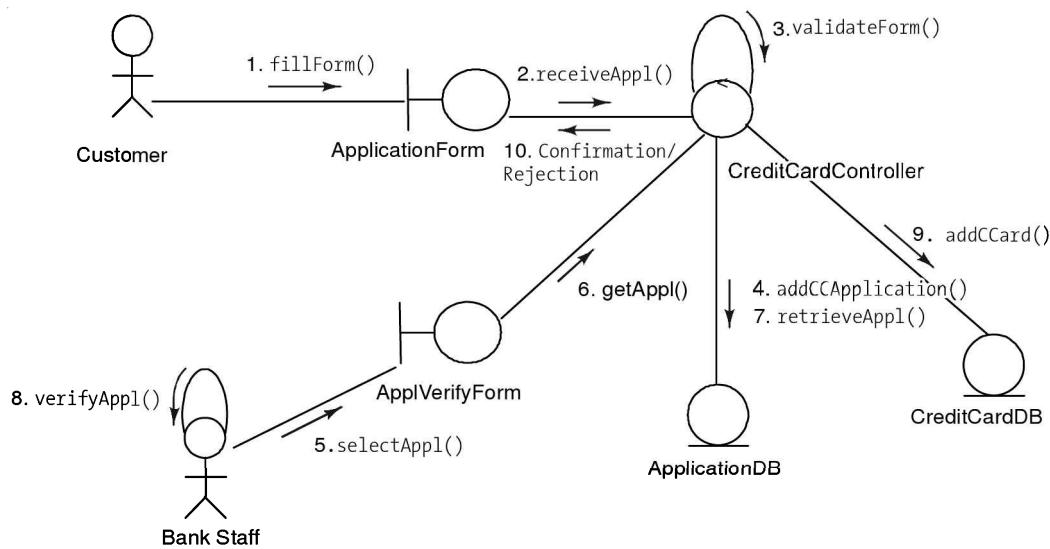


FIGURE 10.25 Collaboration diagram for applying for credit card and verifying application use-case.

Collaboration diagram for activating credit card

The objects, links and methods for activating a credit card are listed in Table 10.14. Figure 10.26 shows the collaboration diagram for activating a credit card use-case.

TABLE 10.14 Objects, links and methods for applying for activating credit card use-case

Objects	Links	Methods (with sequence number)
Bank Staff	Bank Staff—ActivationForm	1. setCreditCardNo()
ActivationForm	ActivationForm—ActivationController	2. validateCCardNo()
ActivationController	ActivationController—CreditCardDB	3. retrieveCreditCardDtls()
CreditCardDB		4. CreditCardDtls
		5. validateCard()
		6. changeStatus()
		7. getStatus()
		8. updateStatus()
		9. Card Activated

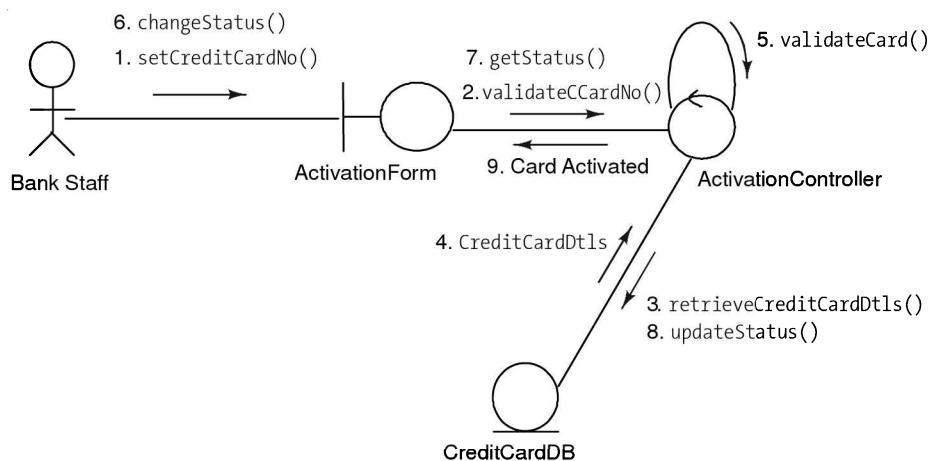


FIGURE 10.26 Collaboration diagram for activating credit card use-case.

Collaboration diagram for viewing transactions and statement

The objects, links and methods for viewing the transactions and statement use-case are given in Table 10.15. Figure 10.27 shows the collaboration diagram for viewing transactions and statement use-case.

TABLE 10.15 Objects, links and methods for viewing transactions and statement use-case

Objects	Links	Methods (with sequence number)
Customer	Customer—CheckTransactionForm	1. setCreditCardNo()
CheckTransactionForm	CheckTransactionForm— TransactionController	2. setPeriod()
TransactionController	TransactionController—CreditCardDB	3. getCreditCardNo()
CreditCardDB	TransactionController—TransactionDB	4. retrieveCreditCardDtls() 5. CreditCardDtls 6. validateCard() 7. getPeriod() 8. retrieveTransactions() 9. Transactions
TransactionDB		

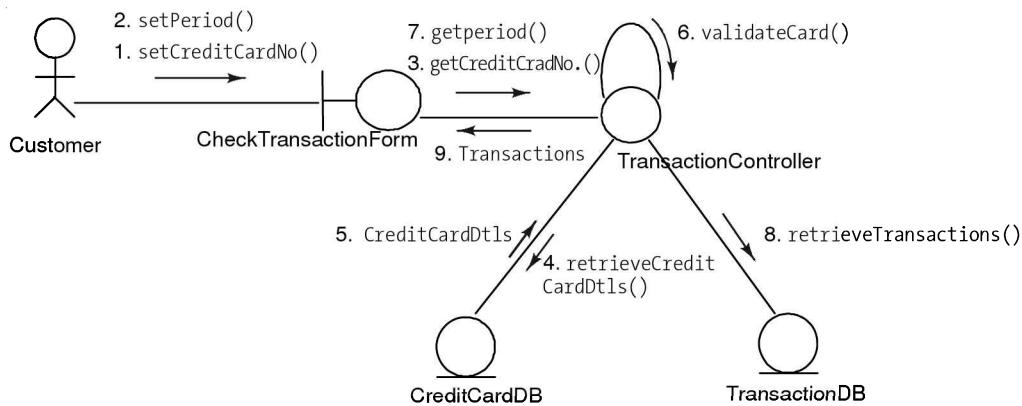


FIGURE 10.27 Collaboration diagram for viewing transactions and statement use-case.

Collaboration diagram for maintaining personal information

Table 10.16 lists the objects, links and methods for maintaining the personal information use-case. Figure 10.28 shows the collaboration diagram for maintaining the personal information use-case.

TABLE 10.16 Objects, links and methods for maintaining personal information use-case.

Objects	Links	Methods (with sequence number)
Customer	Customer—editPersonalForm	1. setCustomerId()
editPersonalForm	editPersonalForm—editPersonalController	2. getCustId()
editPersonalController	editPersonalController—CustomerDB	3. retrieveCustId()
CustomerDB		4. CustId
		5. validateCustId()
		6. retrieveCustDtls()
		7. CustDtls
		8. modifyCustDtls()
		9. getCustDtls()
		10. updateCustDtls()

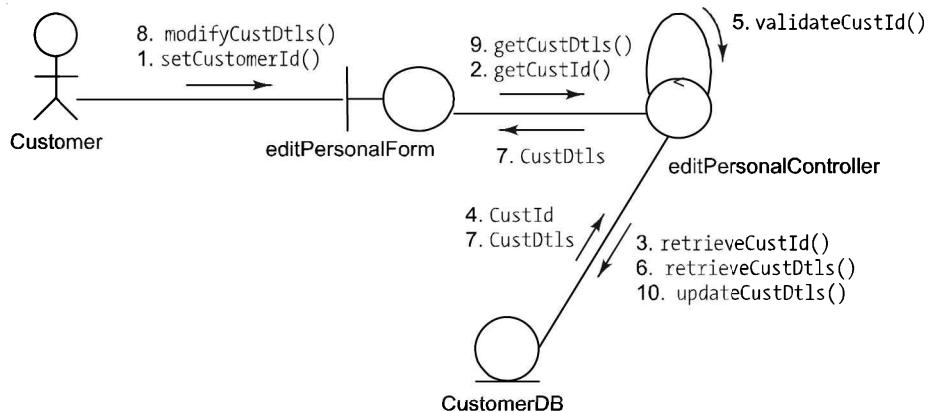


FIGURE 10.28 Collaboration diagram for maintaining personal information use-case.

Collaboration diagram for transfer transaction data

Table 10.17 lists the objects, links and methods for the transfer transaction data use-case. The collaboration diagram for the transfer transaction data use-case is shown in Figure 10.29.

TABLE 10.17 Objects, links and methods for transfer transaction data use-case

Objects	Links	Methods (with sequence number)
CardBrandCorp	CardBrandCorp—TransferInterface	1. setTransactions()
TransferInterface	TransferInterface—TransferController	2. receiveTransactions()
TransferController	TransferController—Transaction	3. addTransactions()
TransactionDB	—	4. Transfer Successful

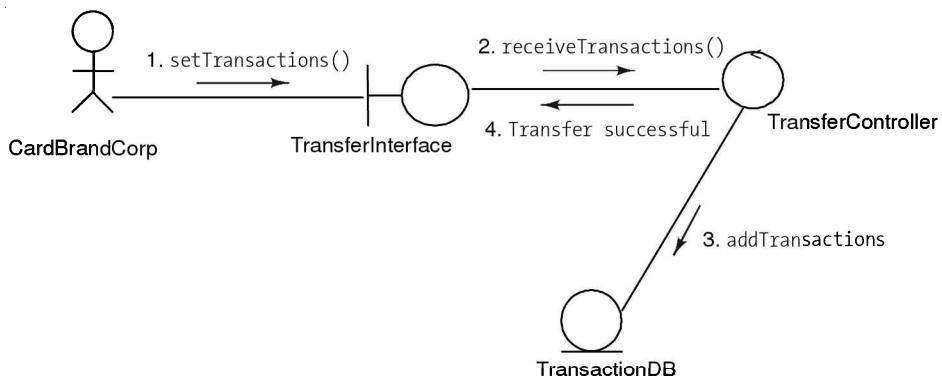


FIGURE 10.29 Collaboration diagram for transfer transaction data use-case.

10.3.3 Statechart Diagram

In this problem statement, the statechart diagram is drawn for *Credit Card Management System*.

The credit card management system has fixed states throughout its life cycle and there may be some abnormal exits also. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as the complete transaction.

The first state is the *receiving application* state from where the process starts.

As the credit card management system has finite states throughout its life cycle, the state chart diagram will be one shot life cycle statechart diagram.

Hence there will be one initial state and one or more final states as in Figures 10.30(a) and (b). The intermediate states are shown in Figure 10.30(c). Figure 10.31 shows the transitions.



FIGURE 10.30(a) Initial State: Begin credit card process.



FIGURE 10.30(b) Final State: End credit card process.

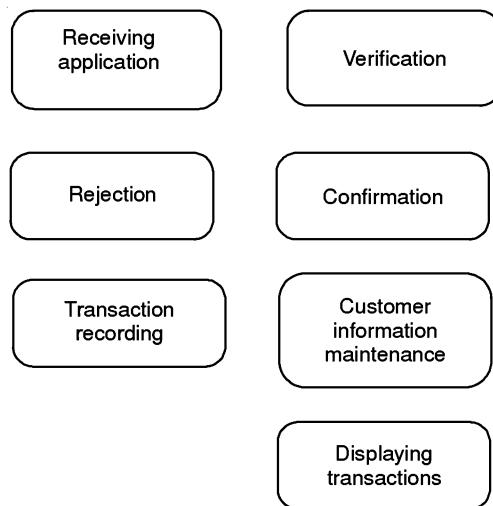


FIGURE 10.30(c) Intermediate states.

After the initial state *Receiving Application*, the next state *Verification* is arrived on the completion of activity during the *Receiving Application* state, hence the transition is a triggerless transition.

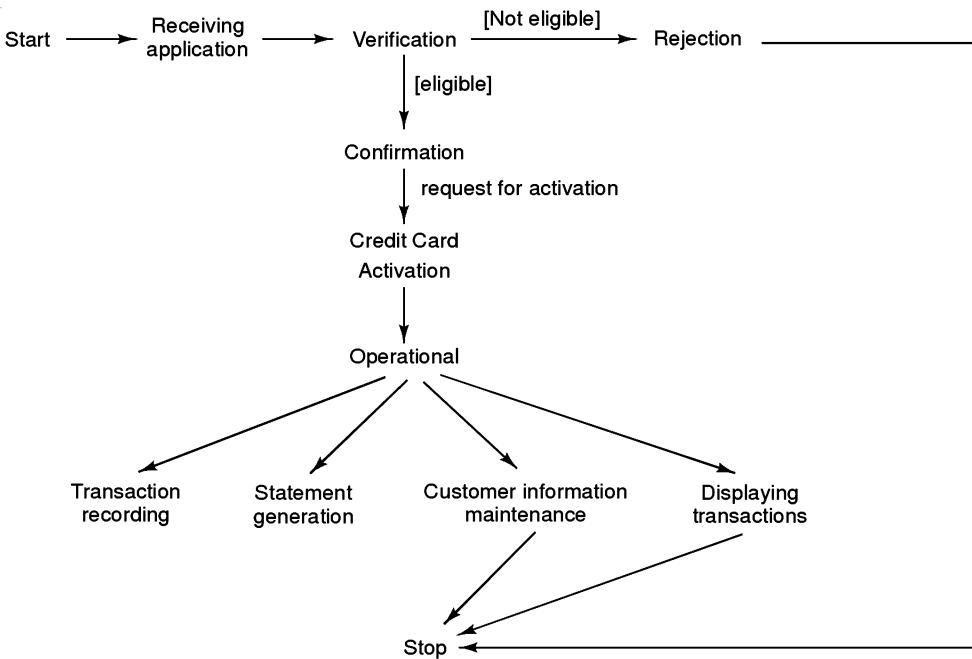


FIGURE 10.31 Transitions.

The guard condition to enter from the *Verification* state into the *Confirmation* or *Rejection* state is *eligible/Not eligible*. No trigger is required to transit from the state.

After *Confirmation*, on request for activation of the credit card, the system enters into the *Credit Card Activation* state.

From the *Credit Card Activation* state, it enters into *Transaction Recording* state on a daily basis at each midnight and into the *Statement Generation* state on a monthly basis automatically.

From the same state on selection of the operation, it can enter into operational state which may be either the *Customer information maintenance* or *Displaying transactions* state.

Complete the state chart diagram for the credit card management system is as in Figure 10.32.

10.3.4 Activity Diagram

There are five main processes in the system for which an activity diagram should be drawn:

- Application for a credit card and activation of a credit card
- Producing monthly statement
- Customer information maintenance
- Credit card transaction recording
- Transaction and statement information checking

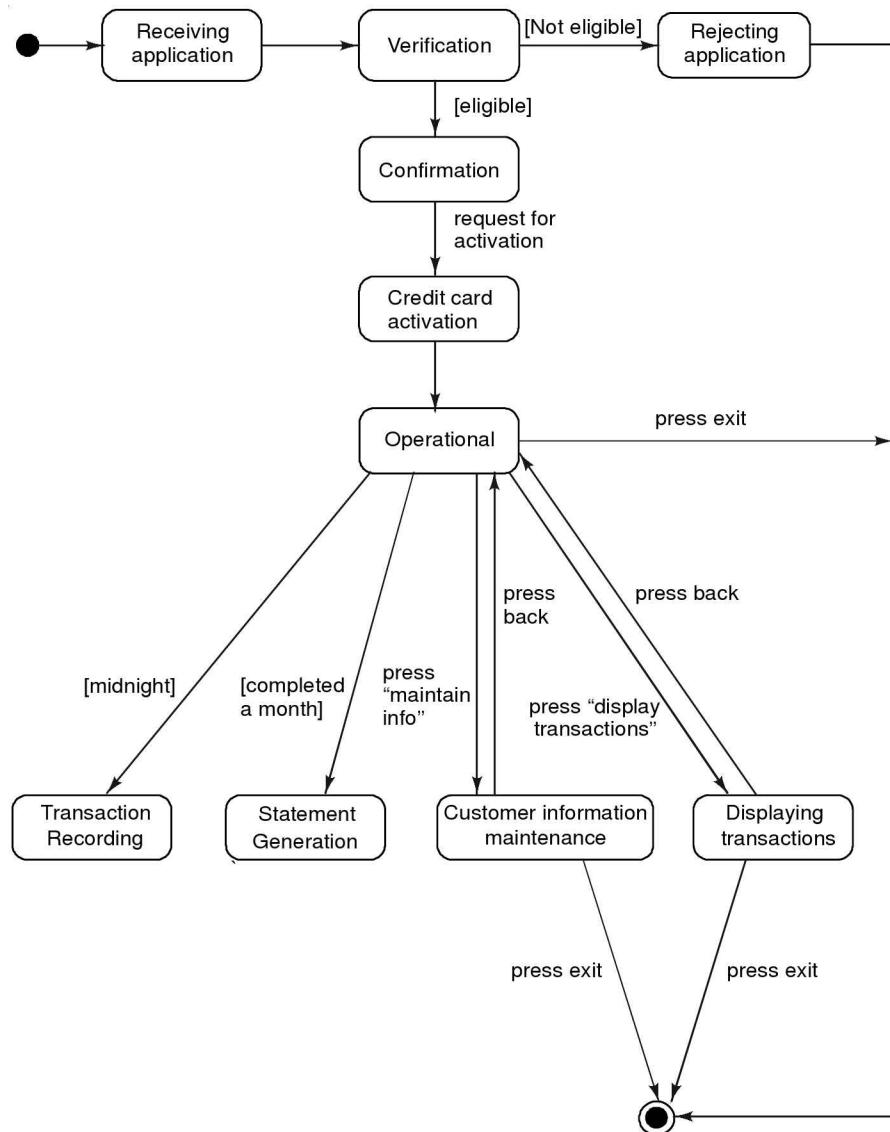


FIGURE 10.32 State chart diagram for credit management system.

For drawing an activity diagram for any process:

1. Find out swimlanes if any. To find swimlanes, see if you can span some activities over different organizational units/places.
2. Find out in which swimlane the online creditcard management process begins and where it ends. Those will be the initial and final states.
3. Then, identify activities occurring in each swimlane. Arrange activities in the sequence flow spanning over all the swimlanes.

4. Identify the conditional flow or parallel flow of activities. The parallel flow of activities must converge at a single point using a join bar.
5. During the activities are performed, if any document is generated or used, take it as an object and show the object flow.

Activity diagram for application for credit card and activation of credit card

Swimlanes: As the customer interacts with ICBI Bank to apply for the credit card and for getting the credit card activated, there are only two swimlanes, Customer and ICBI Bank (Figure 10.33). This process begins from the client side and ends at the bank. The related activities are illustrated in Figure 10.34.

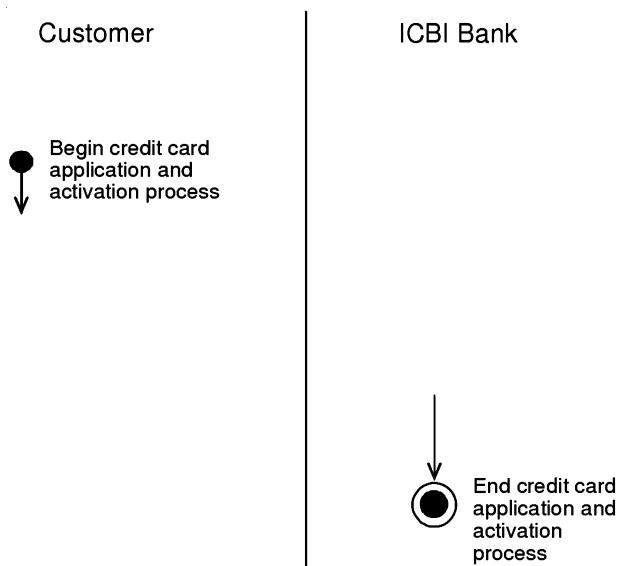


FIGURE 10.33 Swimlanes for application and activation of credit card.

As stated in the problem statement, no parallel flow is identified.

One conditional flow occurs after the *Verify form and supplementary documents* activity which proceeds towards accepting and activating the credit card depending upon the requirement of minimum personal annual income, current employment and financial status or exits if the customer is not fitting in the criteria required.

So the complete activity diagram for the process with the transitions is as shown in Figure 10.35.

Activity diagram for producing monthly statement

Swimlanes: In each month at the midnight of the statement date, ICBI generates monthly statements for each customer. As there is interaction between Customer and ICBI Bank to get a monthly statement, there are only two swimlanes, i.e. Customer and ICBI Bank

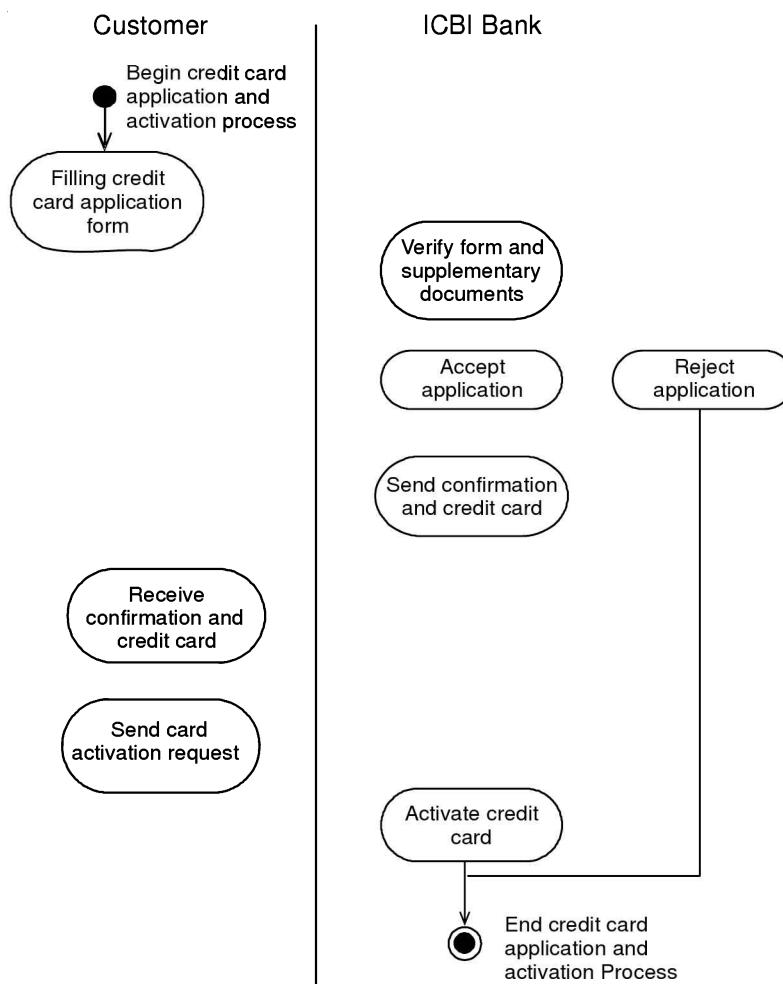


FIGURE 10.34 Activities for application and activation of credit card.

(Figure 10.36). This process begins in the bank and ends at the customer end. The related activities are illustrated in Figure 10.37.

As stated in the problem statement, no parallel flow is identified.

One conditional flow occurs after the *Check if last payment is late* activity which proceeds to generate the monthly statement including late payment charges. If the last payment done is within the date specified, without adding late payment charges, further activity for statement generation is taking place.

Another conditional flow occurs after the *Check if customer needs paper-based statement* activity to check if the customer needs paper-based or electronic statement. If the customer needs paper-based statement, the bank will add service fees for it to the total amount and then generate the monthly statement.

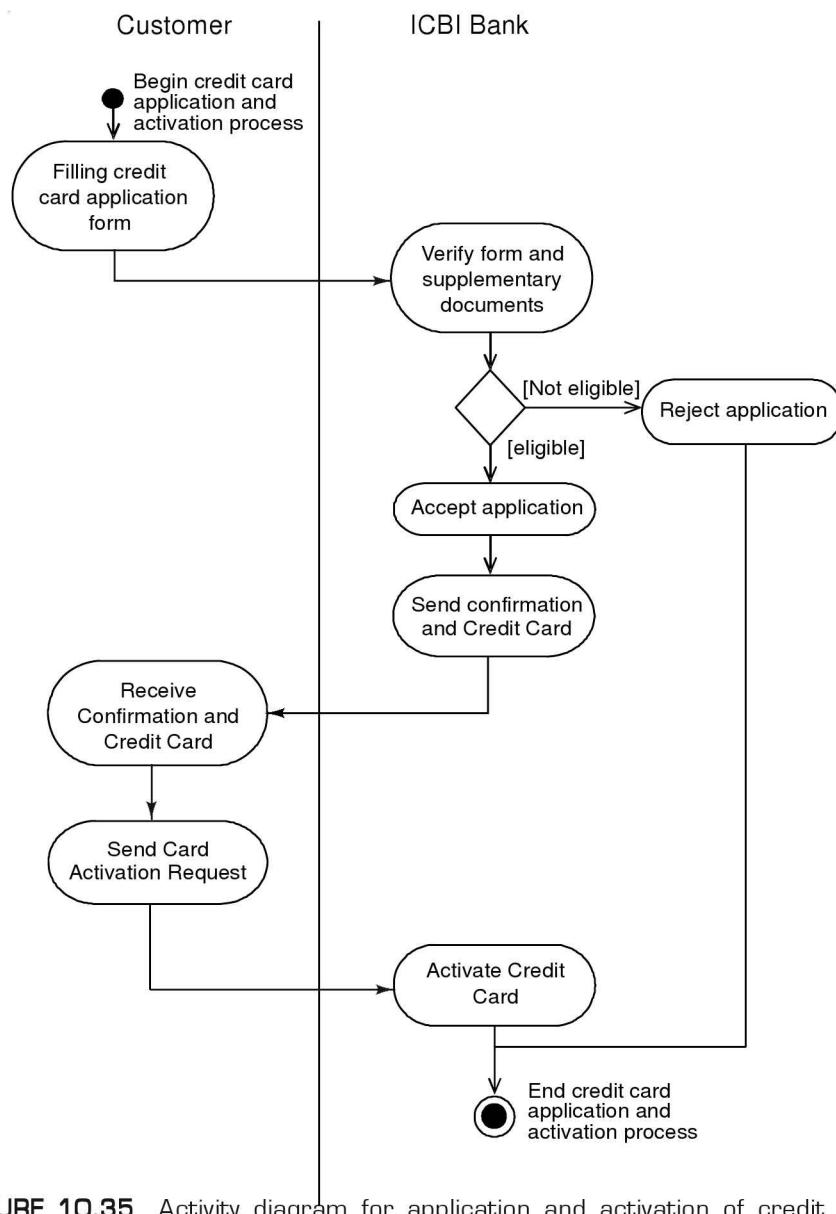


FIGURE 10.35 Activity diagram for application and activation of credit card.

If the customer does not need paper-based statement, i.e. he/she prefers an electronic statement, without adding any service fees, the bank generates the monthly statement. So the complete activity diagram for the process with the transitions is as shown in Figure 10.38.

Customer

ICBI Bank

- Begin getting monthly statement process



ICBI Bank

- Begin getting monthly statement process



FIGURE 10.36 Swimlanes for producing monthly statement.

Customer

ICBI Bank

- Begin getting monthly statement process

Compute debt amount and minimum return

Check if last payment was late

Check if customer needs paper-based statement

Add late payment charges to total amount

Add Service fees to total amount

Generate Monthly statement

Post /E-mail Statement to cardholder

Receive Monthly statement

End getting monthly statement process

FIGURE 10.37 Activities for producing monthly statement.

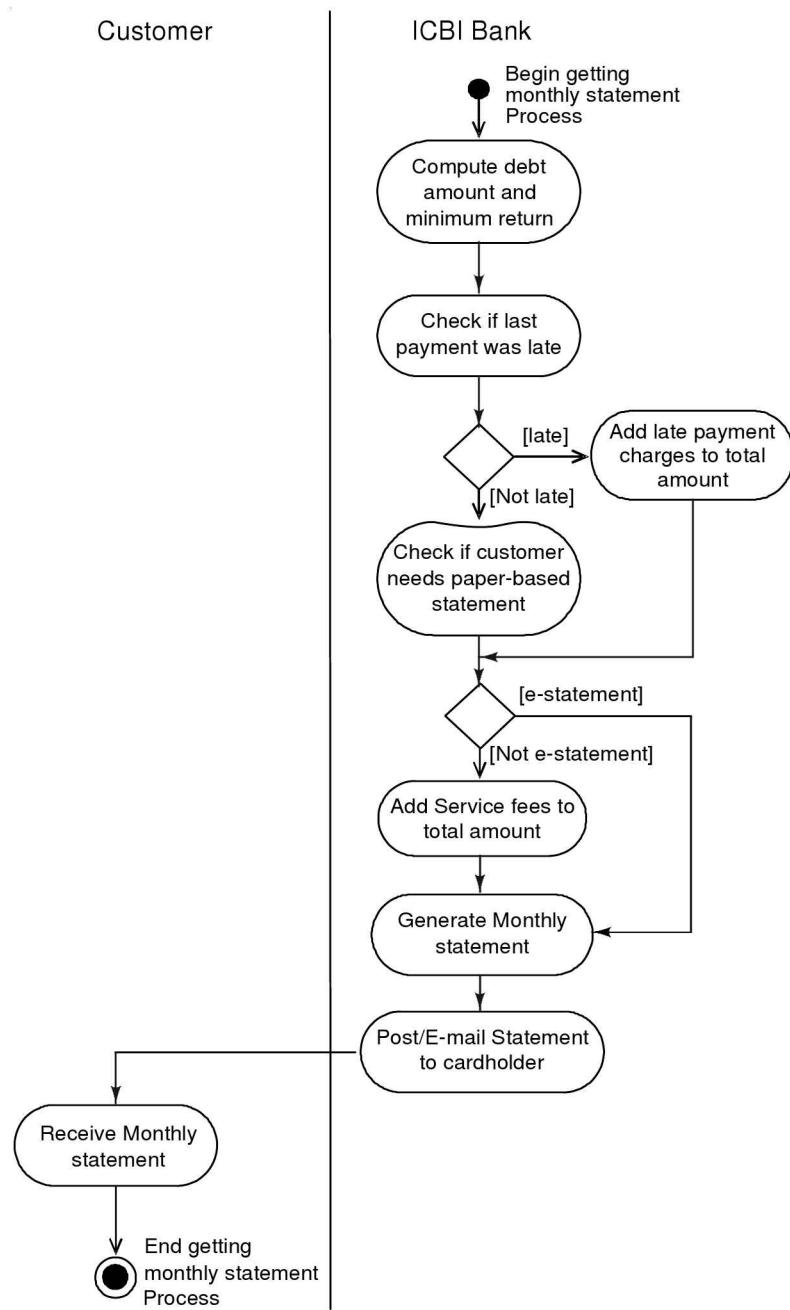


FIGURE 10.38 Activity diagram for producing monthly statement.

Activity diagram for customer information maintenance

Swimlanes: The personal information of individual customers can be maintained online and the process is initiated by the customer. After checking and editing the information by the customer, the customer information gets updated in the bank. There are only two swimlanes, i.e. Customer and ICBI Bank (Figure 10.39). This process begins from the customer side and ends in the bank. The related activities are illustrated in Figure 10.40.

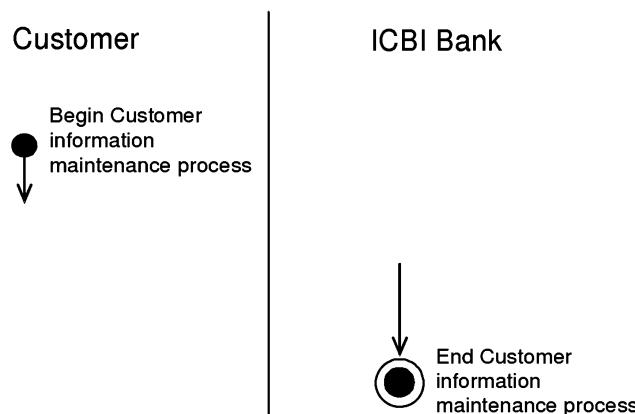


FIGURE 10.39 Swimlanes for customer information maintenance.

As stated in the problem statement, no parallel flow is identified. One conditional flow occurs after the *Verify user login details* which proceeds to allow the customer to modify his/her personal information if he/she is the valid user else exits.

So the complete activity diagram for the process with the transitions is as shown in Figure 10.41.

Activity diagram for credit card transaction recording

Swimlanes: The transaction data regarding customer's payments and purchases taking place randomly are transferred from CardBrandCorp to ICBI bank on a daily basis at each midnight (12:00 pm).

There are only two swimlanes, namely CardBrandCorp and ICBI Bank (Figure 10.42). This process of recording credit card transactions is initiated by CardBrandCorp and ends after recording the transactions in the bank. The related activities are illustrated in Figure 10.43.

As stated in the problem statement, no parallel flow is identified.

One conditional flow occurs after the *Transfer payment and purchase data* activity which proceeds towards rewarding cash dollars to customers if the transaction is payment transaction or calculation of debt amount if the transaction is purchase transaction.

Another conditional flow occurs after the *Rewards Cash Dollars to Customers* activity to check if the accumulated cash dollars should be used during purchase or not. If the payment is specified to use cash dollars during purchase, then deduct cash dollar amount from spending, otherwise just update the debt amount by adding new spending amount.

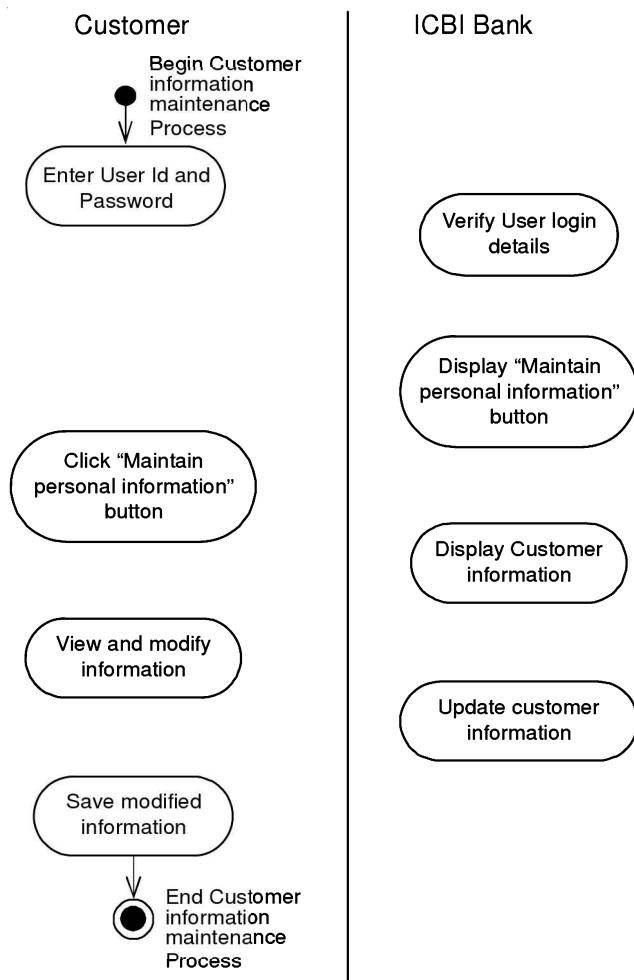


FIGURE 10.40 Activities for customer information maintenance.

The transaction recording process ends by updating the outstanding credit card balance in the bank.

So the complete activity diagram for the process with the transitions is as shown in Figure 10.44.

Activity diagram for transaction and statement information checking

Swimlanes: Customers check their credit card transactions and monthly statement information online after login to their accounts. There are only two swimlanes, namely Customer and ICBI Bank (Figure 10.45). This process of transaction and statement information checking is initiated by the customer and ends after displaying the transactions and monthly statement by the bank. The related activities are illustrated in Figure 10.46.

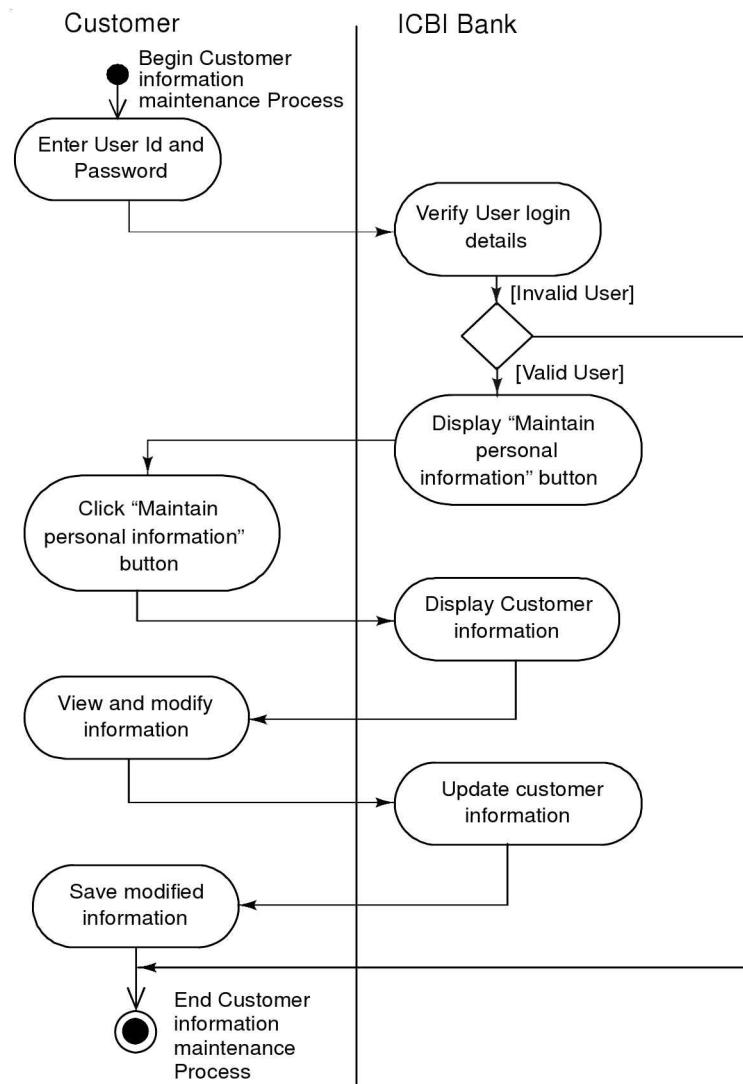


FIGURE 10.41 Activity diagram for customer information maintenance.

As stated in the problem statement, no parallel flow is identified.

One conditional flow occurs after the *Verify user login details* which proceeds towards displaying transactions and monthly statement information within the specified period if he/she is the valid user else exits.

So the complete activity diagram for the process with the transitions is as shown in Figure 10.47.

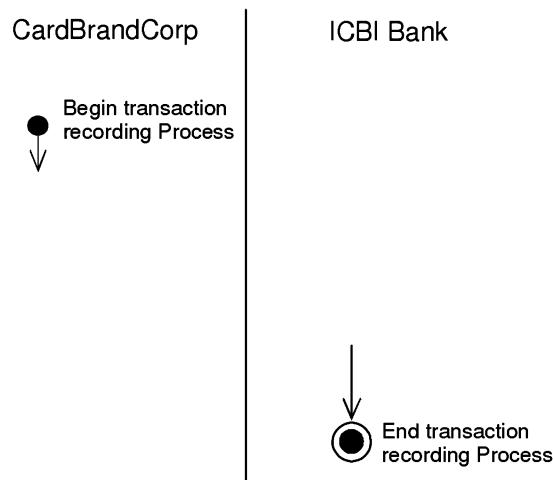


FIGURE 10.42 Swimlanes for credit card transaction recording.

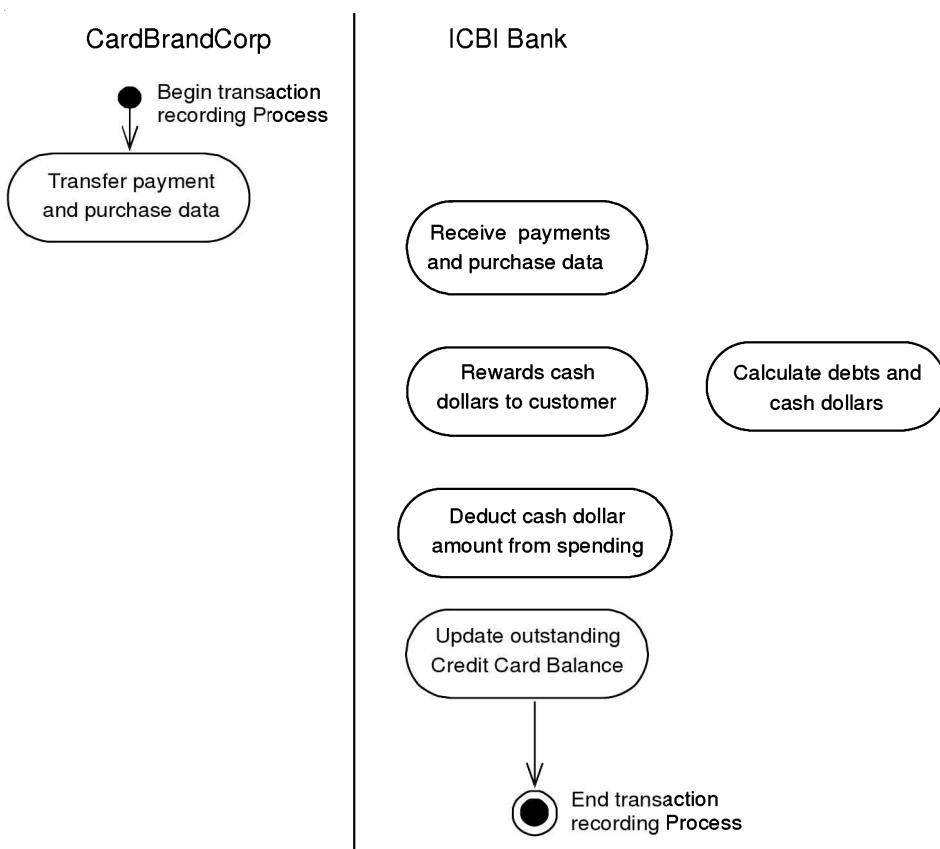


FIGURE 10.43 Activities for credit card transaction recording.

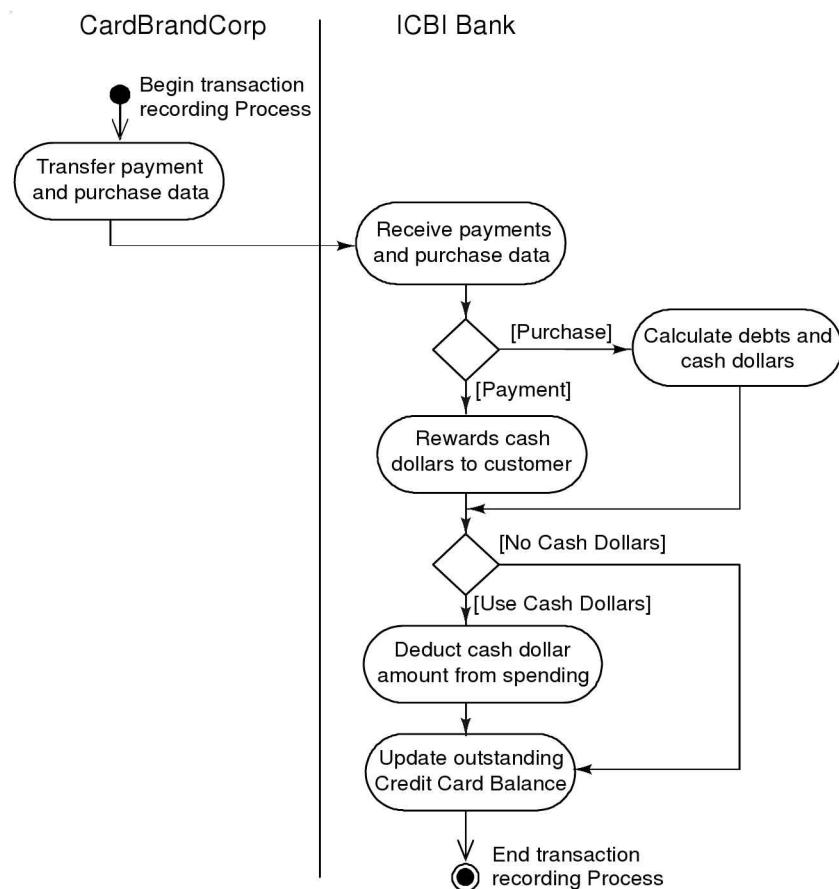


FIGURE 10.44 Activity diagram for credit card transaction recording.

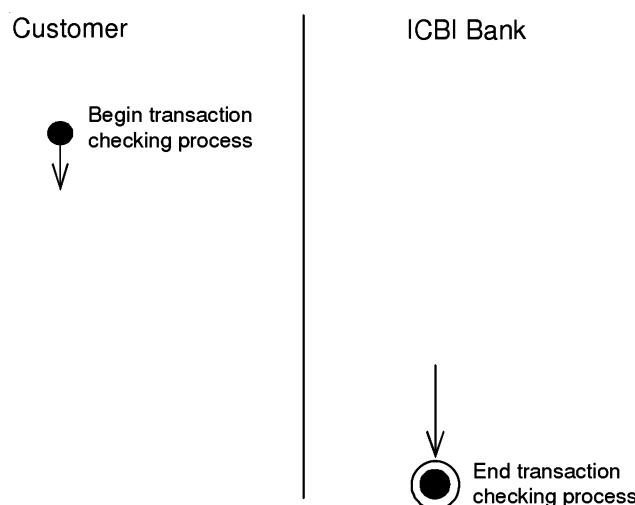


FIGURE 10.45 Swimlanes for transaction and statement information checking.

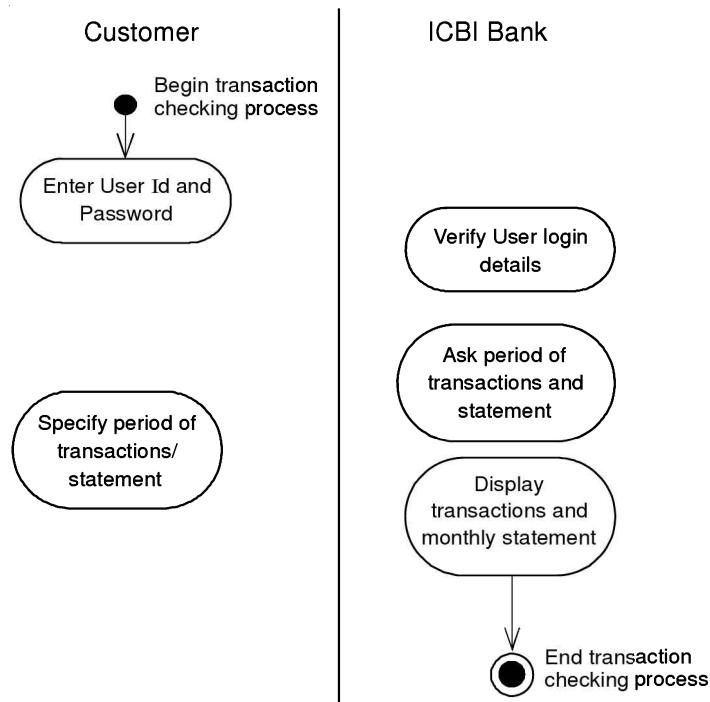


FIGURE 10.46 Activities for transaction and statement information checking.

10.4 IMPLEMENTATION PHASE DIAGRAMS: CREDIT CARD MANAGEMENT SYSTEM

10.4.1 Component Diagram

Component diagrams illustrate the pieces of software that will make up a system.

So, major components identified making up this system are as follows:

1. Online credit card management system
2. CardBrandCorp
3. Customer
4. Credit card
5. Credit card database
6. Security and persistence.

As shown in Figure 10.48, the system is for a bank which provides online services, to facilitate its credit card business, such as applying online for the credit card, online credit card payment, produce monthly statement, customer information maintenance, recording credit card transactions, checking the transaction and information online.

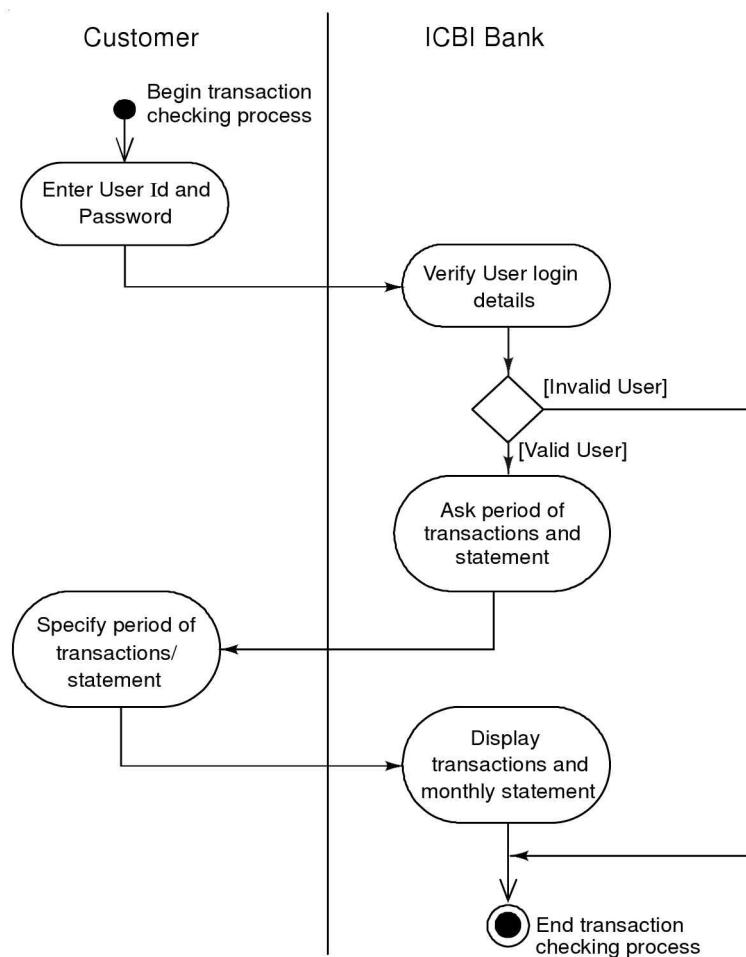


FIGURE 10.47 Activity diagram for transaction and statement information checking.

The main application component in the system is ICBI Bank's online credit card management system which depends on components identified, are *customer* and *credit card*, which are implementing corresponding interfaces *ICustomer*, *ICreditCard*.

Persistence and *Security* components represent functional components for storing data into the database represented by credit card database component which is the data store component and managing the access control to the system.

10.4.2 Deployment Diagram

- Online credit card management system is an internet-based application.
- Online credit card management system is based on multi-tier architecture.

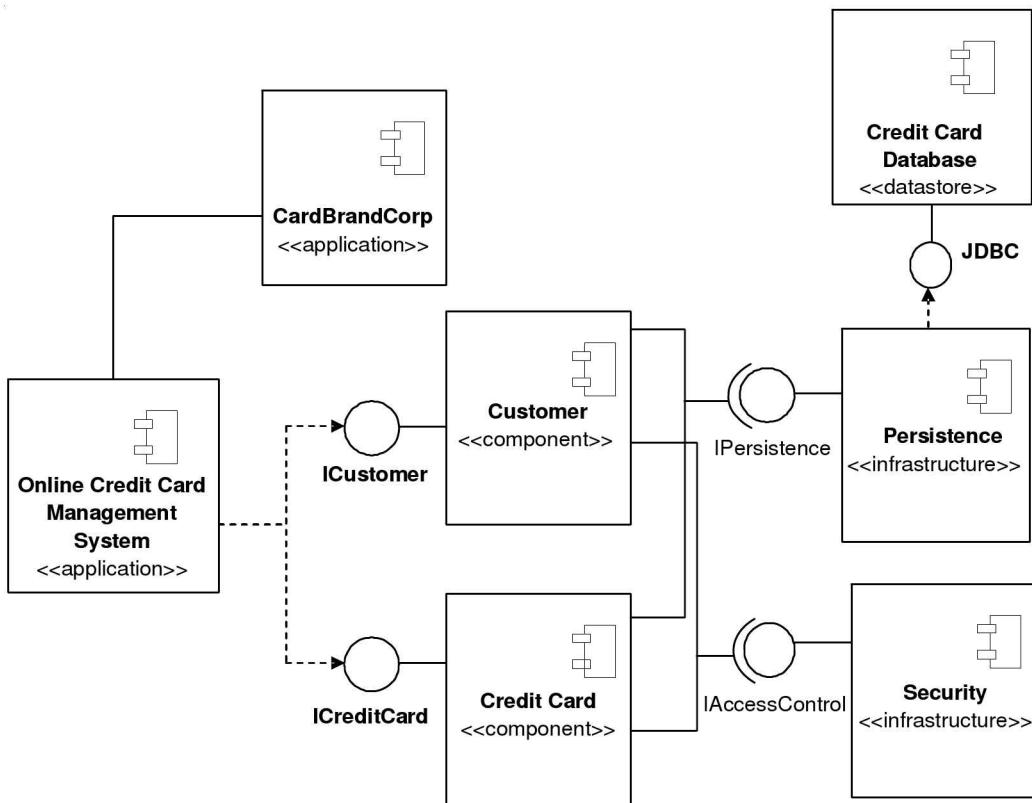


FIGURE 10.48 Component diagram for online credit card management system.

- Hence there will be a database tier holding the credit card database, application tier holding bank's credit card management system, client tier holding client workstation with a browser and one more tier connected to application tier holding the card brand corporation system CardBrandCorp.
- Network used is WAN.
- Data communication among all the nodes is through TCP/IP protocol.

In Figure 10.49, the database server is representing the database layer on which the database server, e.g. Oracle/MS-SQL Server software and Credit Card database, will be installed.

The application server is representing the application tier on which application server software, e.g. WebSphere application server and online credit card management system (all source code pages, dynamic link libraries, etc.) will be deployed.

The card brand corporation server is representing another tier connected to the application tier on which the system CardBrandCorp will be deployed.

Clients access the application through the browser. Hence, on the client tier, only browser is required.

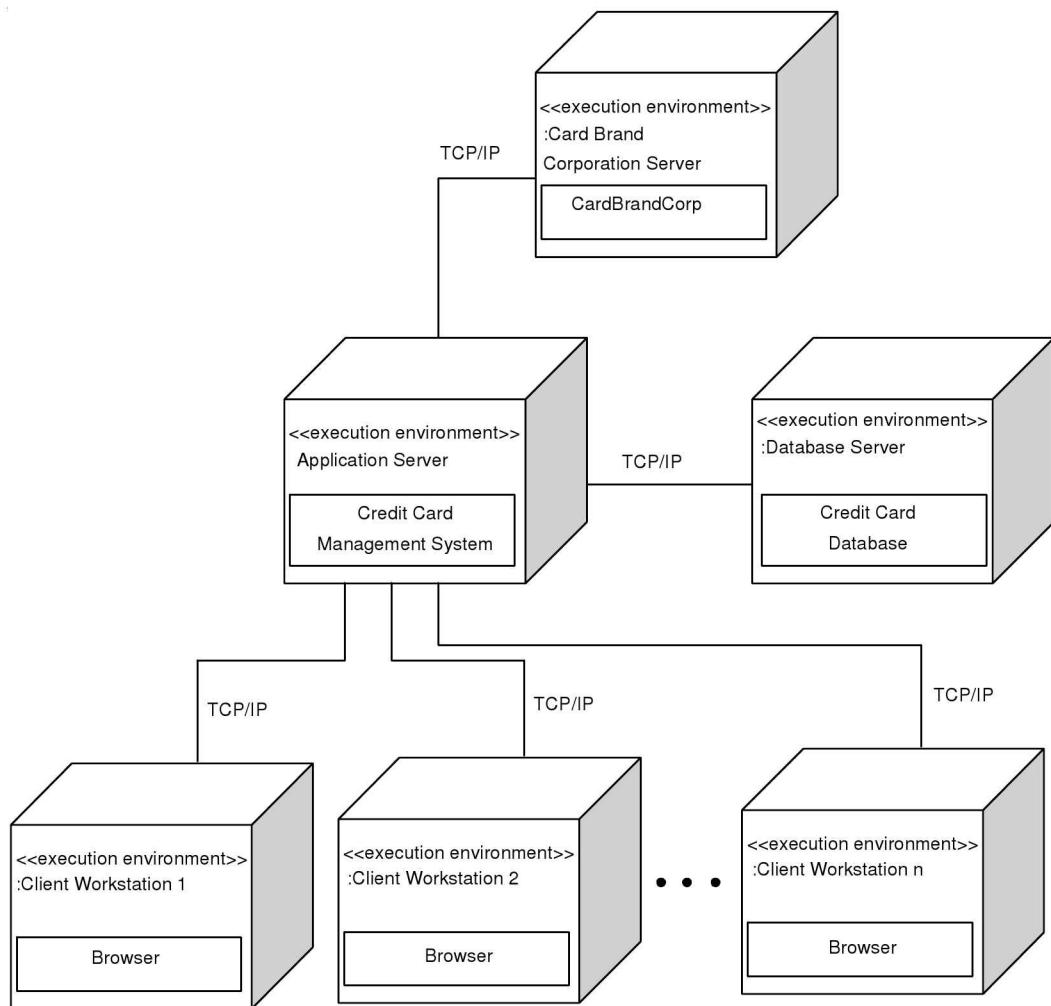


FIGURE 10.49 Deployment diagram for online credit card management system.

C H A P T E R
11

Warehouse Management System

11.1 PROBLEM STATEMENT: WAREHOUSE MANAGEMENT SYSTEM

The system will support warehouse management. The company ordering the system, APCL Warehouse Management Co., specializes in supporting its customers with warehouse spaces all over the nation. Examples of customers are companies that need space to store their products before they are shipped or companies that need local warehouses without having local offices. APCL is already a specialist in storing different kinds of items and in the use of trucks to redistribute the items. APCL plans to grow and now needs an information system with which they can grow. The idea is to offer the customers warehouse space and redistribution services between different warehouses with full computer support. The service includes redistribution both within a warehouse and between warehouses, all dictated by customer needs. All kinds of items may be stored in the warehouses, which means that it is important to differentiate between certain kinds of items; for example some items must not come into contact with other items (such as industrial chemicals and foods).

The following people will be using the system in some way or another:

- Foreman — responsible for one warehouse.
- Warehouse worker — works in a warehouse, loading and unloading.
- Truck driver — drives a truck between different warehouses.
- Forklift operator — drives a forklift in one warehouse.
- Office personnel — receives orders and requests from customers.
- Customers — own the items in the warehouses and give instructions as to where and when they want the items.

It is fundamental to the APCL system that it should be as decentralized as possible and that all persons involved should be reachable at all times. Therefore, the truck drivers should have communication devices for getting their orders and they must be able to communicate with the foreman or the office. This means that we also need a radio communication network, a system that should not be developed by us but bought separately. The warehouse worker's loading and unloading should use a barcode reader when handling the items in order to be as efficient as possible. This means that all items must be marked when inserted in the warehouse system by a warehouse worker; this marking must at the same time give information about the item to the information system. The foreman should be able to work with several items at the same time, so they will probably need a window-based terminal. They are responsible for effecting the redistribution orders from the office.

When a customer wants to do something with his items, he will contact the office, which in turn submits redistribution orders to the system. Eventually, if the system is to work well, APCL plans to give their customers terminals so that they can interact directly with the system.

For the above application, build an Analysis Model, Design Model and Implementation Model with the following diagrams:

1. Business process diagram
2. Use case diagram
3. Class diagram
4. Object diagram
5. Sequence diagram
6. Collaboration diagram
7. Statechart diagram
8. Activity diagram
9. Component diagram
10. Deployment diagram

11.2 ANALYSIS PHASE DIAGRAMS: WAREHOUSE MANAGEMENT SYSTEM

11.2.1 Business Process Diagram

There are two main processes identified in the above case study:

- (a) Storing goods in warehouses
- (b) Redistribution of goods from warehouse to warehouse

Hence, there will be two business process diagrams, one drawn for each process.

Business process diagram for storing goods in warehouse

Pool and lanes: Here, APCL Warehouse Management Company and Customer are the participants involved in all the activities. So, pools in this system are APCL Warehouse Management Company and Customer as shown in Figure 11.1.

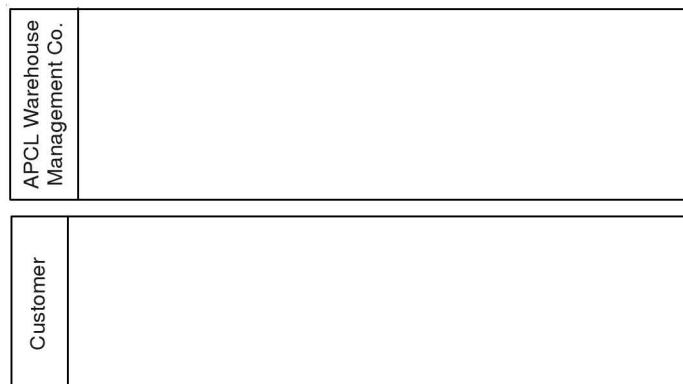


FIGURE 11.1 Pools for storing goods.

Here, the *customer* is the pool which will not have any lane within it. An *office personnel*, *foreman*, *warehouse workers* and *truck driver* perform the organizational roles within the APCL Warehouse Management Company pool. So, lanes in APCL Warehouse Management Company pool are the office personnel, foreman and warehouse workers as shown in Figure 11.2.

APCL Warehouse Management Co.	Warehouse workers	
	Foreman	
	Office personnel	

FIGURE 11.2 Lanes within APCL Warehouse Management Company pool.

Activities: Tasks performed in the APCL Warehouse Management Company pool under various lanes are shown in Figure 11.3.

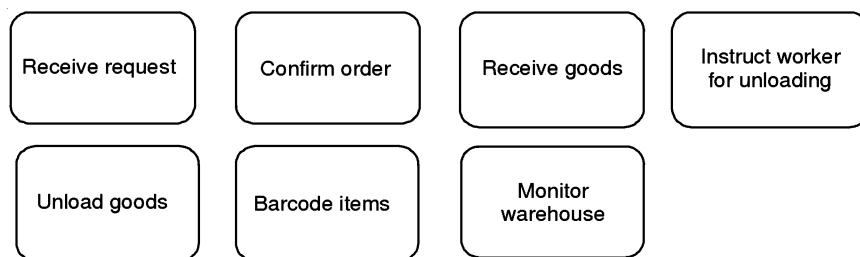


FIGURE 11.3 Task performed under various lanes within APCL Warehouse Management Company pool.

Tasks performed within the customer pool are shown in Figure 11.4.



FIGURE 11.4 Task performed within customer pool.

Events: There are only start and end events identified in this process as shown in Figure 11.5.

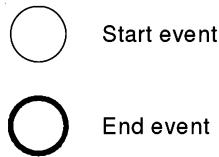


FIGURE 11.5 Events for warehouse management.

Gateways: There will not be any gateway element in the diagram as the whole process is sequential without any convergence or divergence in it.

Artifacts: In this example, artifacts are as shown in Figure 11.6.

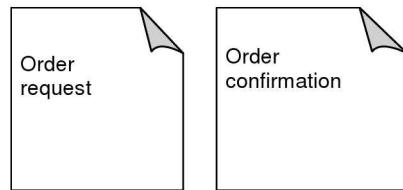


FIGURE 11.6 Artifacts for warehouse management.

Now a complete Business Process Diagram for storing goods in a warehouse will be drawn as shown in Figure 11.7.

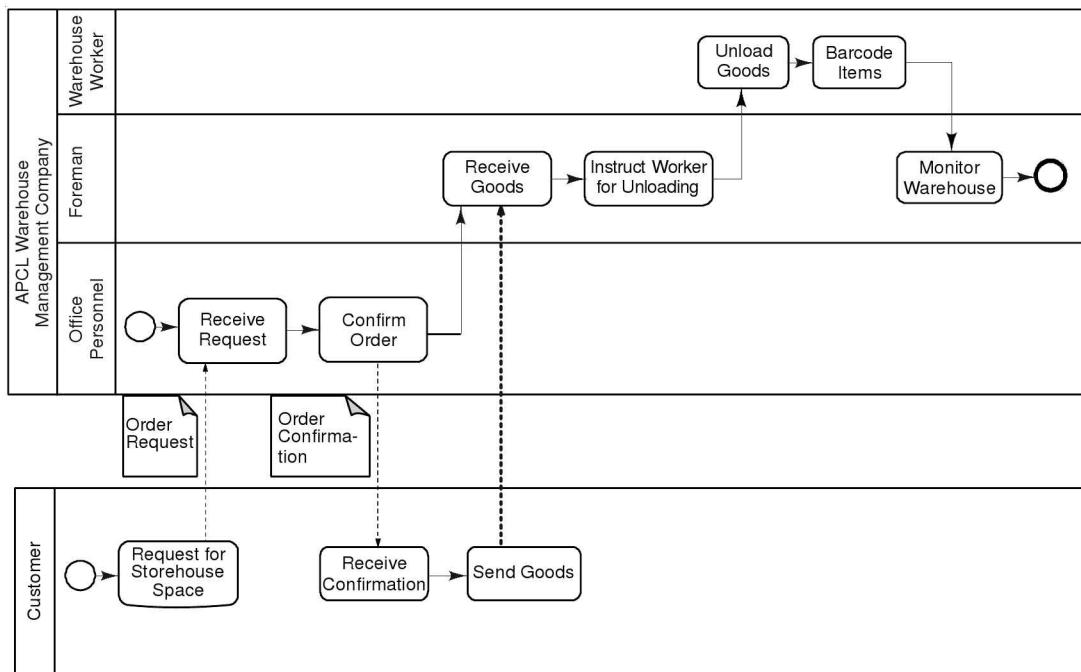


FIGURE 11.7 Business process diagram for storing goods in the warehouse.

Business process diagram for redistribution of goods from warehouse to warehouse

Here, *APCL Warehouse Management Company* and *Customer* are the participants involved in all the activities. So, pools in this system are APCL Warehouse Management Company and Customer as illustrated in Figure 11.8.

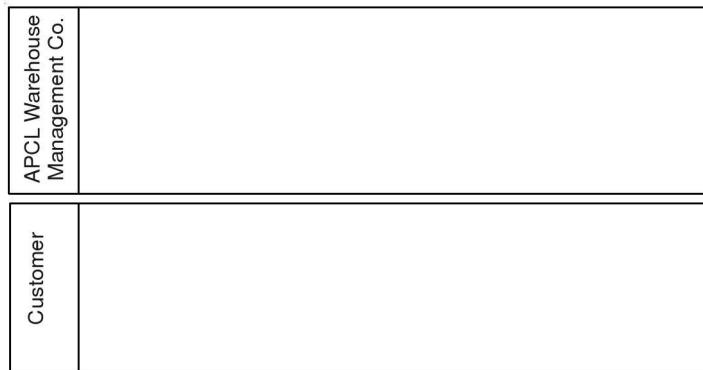


FIGURE 11.8 Pools for redistribution of goods.

Here, the customer is the pool which will not have any lane within it. An office personnel, foreman, warehouse workers and truck driver perform the organizational roles within the APCL Warehouse Management Company pool. So, lanes in the APCL Warehouse Management Company pool are an office personnel, foreman, warehouse workers and truck driver.

Activities: Tasks performed in the APCL Warehouse Management Company pool under various lanes are illustrated in Figure 11.9.

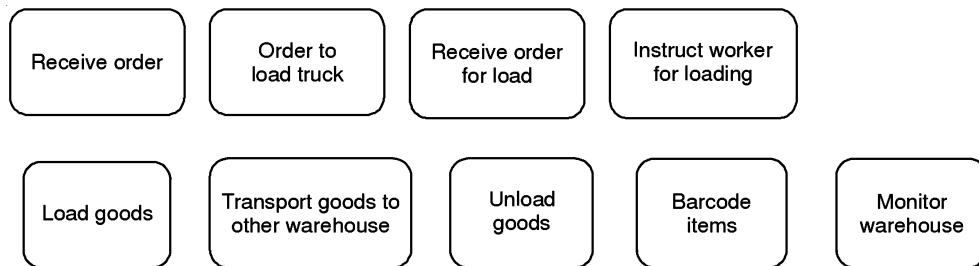


FIGURE 11.9 Tasks performed within APCL Warehouse Management Company pool.

Tasks performed within the customer pool are shown in Figure 11.10.



FIGURE 11.10 Tasks performed within customer pool.

Events: There are only start and end events identified in this process (Figure 11.11).

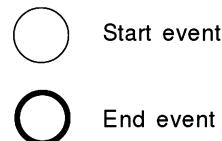


FIGURE 11.11 Events for redistribution of goods.

Gateways: There will not be any gateway element in the diagram as the whole process is sequential without any convergence or divergence in it.

Artifacts: In this example, the only artifact is shown in Figure 11.12.

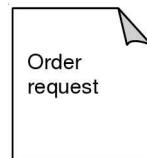


FIGURE 11.12 Artifacts for redistribution of goods.

Now a complete business process diagram for redistribution of goods from warehouse to warehouse will be drawn as shown in Figure 11.13.

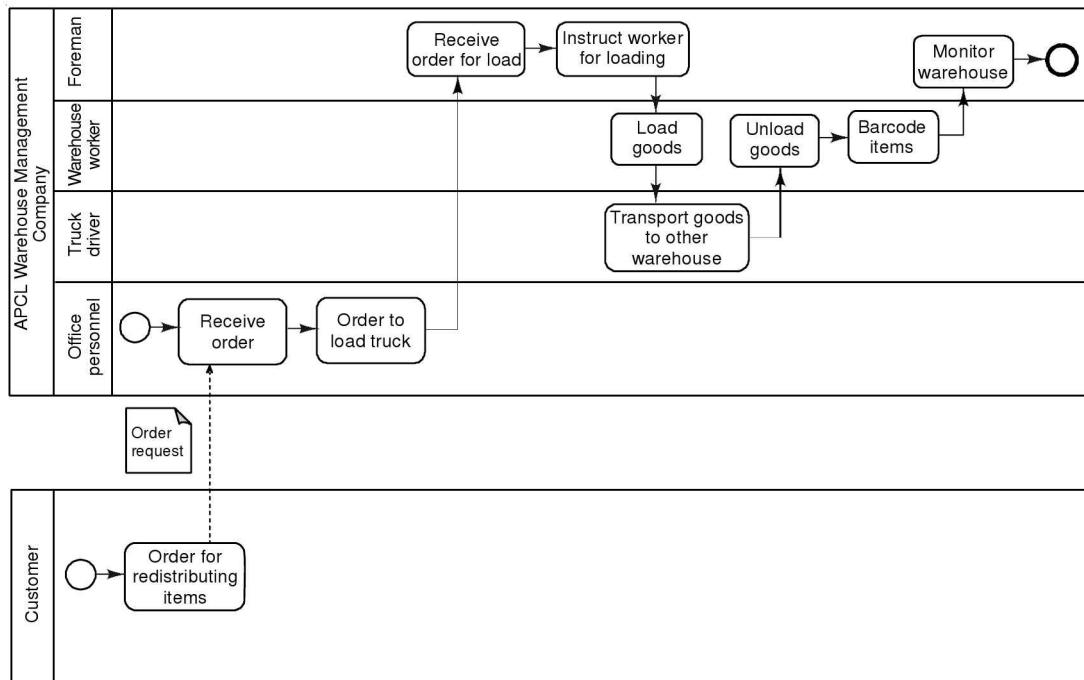


FIGURE 11.13 Business process diagram for re-distribution of goods.

11.2.2 Use-Case Diagram

Actors and use-cases for warehouse management are shown in Table 11.1.

TABLE 11.1 Actors and their use-cases for warehouse management

Actors	Use-Cases		
	Base	Include	Extends
Customer	Registration Login Place order		Order for storage Order for redistribution of goods
Office personnel	Registration Login Manage customer orders		
Foreman	Registration Login Monitor warehouse Handle orders		
Warehouse worker	Registration Login Handle goods		Load goods Unload goods

As specified in the problem statement, along with all the above actors the warehouse management system also involves two more actors having some responsibilities as understood from the problem statement and shown in the Table 11.2. But, these are the physical processes performed by the actors.

To carry out the functionalities, the actors do not interact with the system directly. Hence, these actors and functionalities could not be modelled in the use-case diagram. Note that, only those actors and functionalities involving interaction with the system should be modelled in the use-case diagram.

TABLE 11.2 Additional actors and their use-cases for warehouse management.

1	Forklift operator	Drive forklift
2	Truck driver	Get orders for pickup and redistribution of goods Transfer goods

First, we will draw a actor-wise use-case diagram and finally a combined use-case diagram representing the whole system with all actors and use-cases performed by them. There are four actors in the system: customer (Person), office personnel (Person), foreman (Person) and warehouse worker (Person).

Use-case diagram for customer: The use-case diagram for the customer is shown in Figure 11.14(a).

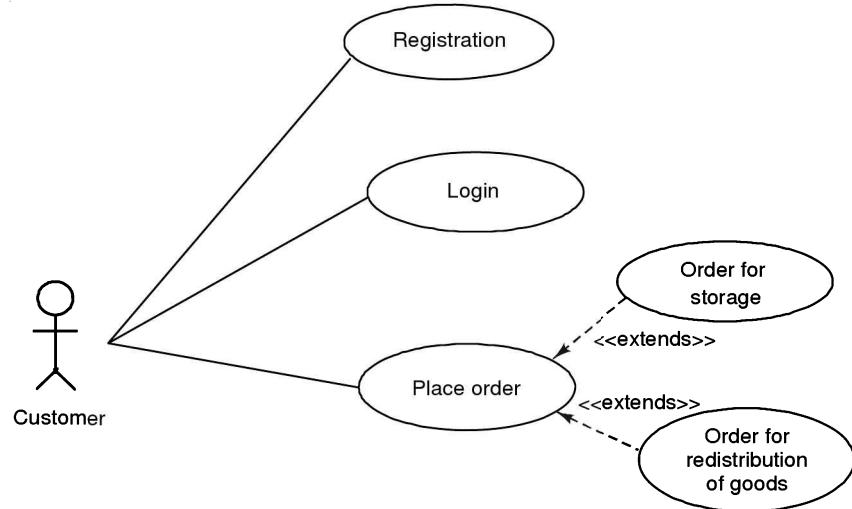


FIGURE 11.14(a) Use-case diagram for customer.

Use-case diagram for office personnel: Figure 11.14(b) shows the use-case diagram for office personnel.

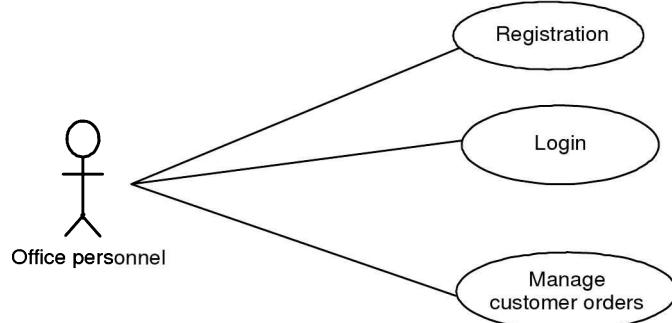


FIGURE 11.14(b) Use-cases for office personnel.

Use-case diagram for foreman: Figure 11.14(c) gives the use-case diagram for the foreman.

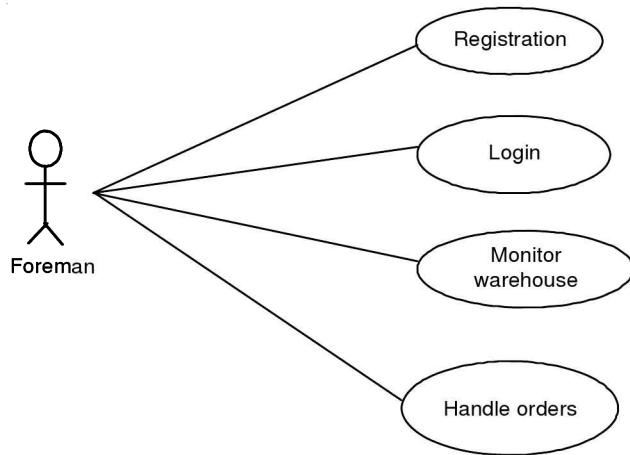


FIGURE 11.14(c) Use-case diagram for foreman.

Use-case diagram for warehouse worker: The use-case diagram for the warehouse worker is shown in Figure 11.14(d).

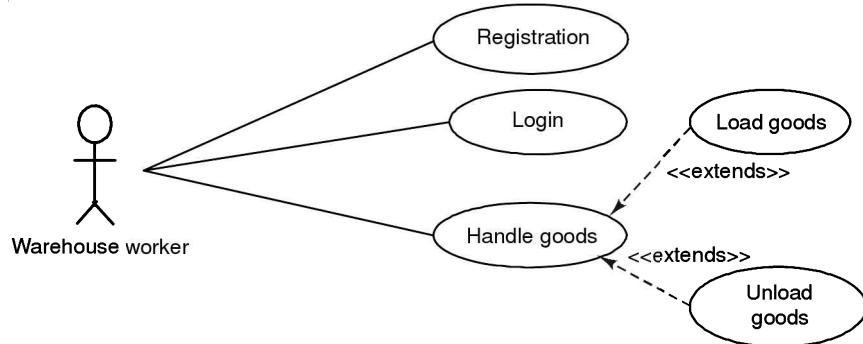


FIGURE 11.14(d) Use-case diagram for warehouse worker.

Complete use-case diagram representing the whole warehouse management system: The complete use-case diagram representing the whole warehouse management system is shown in Figure 11.14(e).

Important points: In the above use-case diagram, apart from use cases obvious from the problem statement as found out in Table 11.2, we have added two additional use cases, i.e. Registration and Login which are not mentioned in the problem statement, but we have to consider it as everybody has a different role in the system.

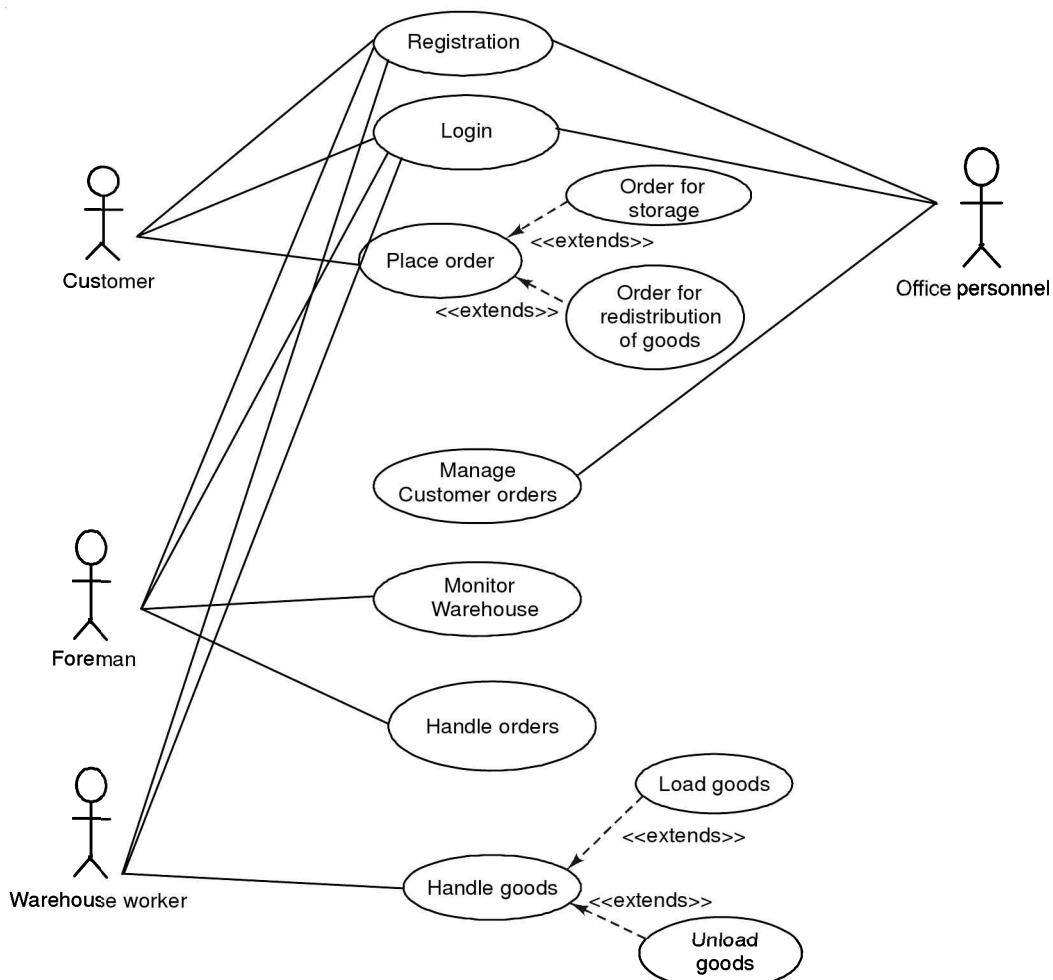


FIGURE 11.14(e) Complete use-case diagram for warehouse management.

11.2.3 Class Diagram

From the analysis of the case we infer the following:

1. Warehouse Management Company has customers.
2. Customer places order.
3. Order is for either storage space or redistribution of goods.
4. Warehouse Management Company provides warehouses.
5. Warehouse stores products.
6. Warehouse Management Company has employees.
7. Employees are office personnel, warehouse worker, truck driver, forklift operator, foreman.

8. Truck driver drives a truck.
9. Forklift operator drives a forklift.
10. Foreman monitors a warehouse.
11. Warehouse worker works in the warehouse.
12. Office personnel receive order.

From the problem statement, identified classes and their relationships are as given in Table 11.3.

TABLE 11.3 Classes and their relationships for warehouse management system

Sr. No.	Class 1	Relationship name	Relationship type	Class 2
1	Warehouse Management Co.	has	Association	Customer
2	Customer	places	Association	Order
3	Order	is	Generalization	Storage order Redistribution order
4	Warehouse Management Co.	provides	Association	Warehouse
5	Warehouse	store	Association	Product
6	Warehouse Management Co.	Has	Association	Employee
7	Employee	is	Generalization	Truck driver Forklift operator Foreman Warehouse worker Office personnel
8	Truck driver	drives	Association	Truck
9	Forklift operator	drives	Association	Forklift
10	Foreman	monitors	Association	Warehouse
11	Warehouse worker	works in	Association	Warehouse
12	Office personnel	receive	Association	Order

The class diagram is shown in Figure 11.15 and the object diagram is depicted in Figure 11.16.

11.3 DESIGN PHASE DIAGRAMS: WAREHOUSE MANAGEMENT SYSTEM

11.3.1 Sequence Diagram

In the above case, actors and use-cases identified can be summarized as given in Table 11.4.

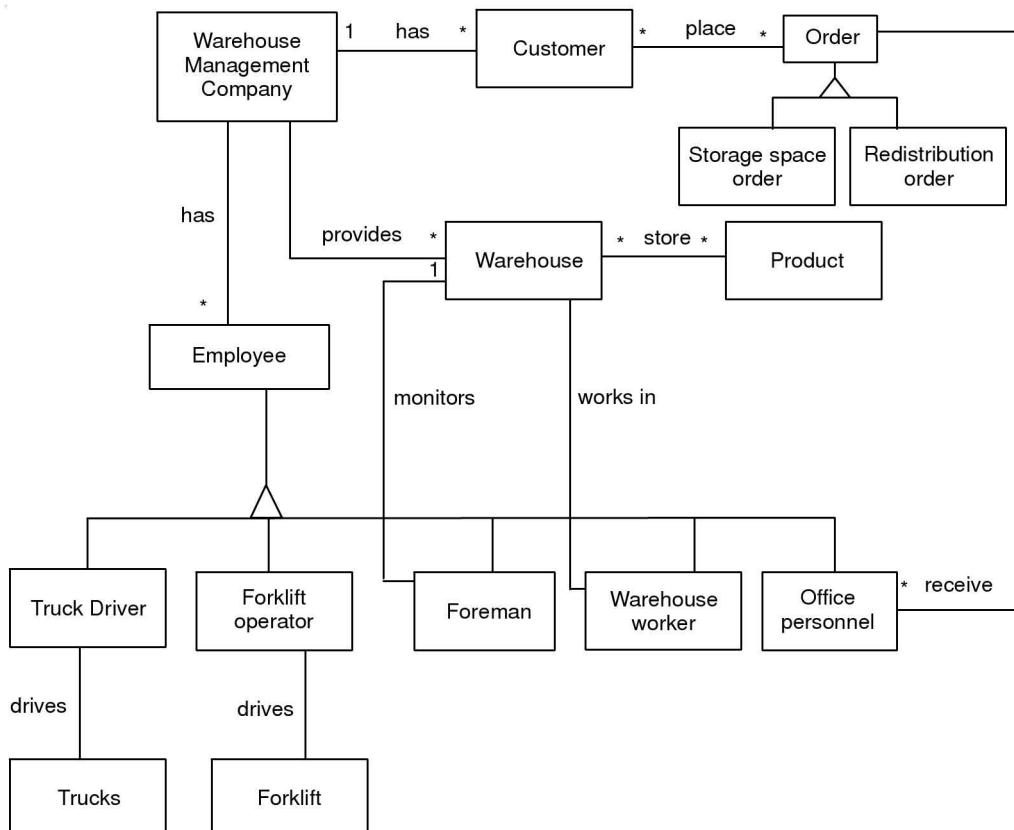


FIGURE 11.15 Class diagram for warehouse management.

During the design process, identify four types of objects interacting with each other to perform the use-case:

1. Actor objects—Person, external system or device that initiates the use-case, e.g. Customer, Office personnel, Foreman, Warehouse worker, etc.
2. Boundary objects—All the interfaces through which than actor interacts with the system, e.g. Login screen, Order form, ItemEntry form, etc.
3. Controller objects—All the objects which coordinate the task, e.g. Security manager, etc.
4. Entity objects—All domain entities identified during the analysis class diagram, e.g. Customer, Order, Employee, Warehouse, Product, etc.

Sequence diagram for registration use-case

The use-case *registration* is performed by all the actors, i.e. Customer and Employees (Office personnel, Foreman, Warehouse worker) for registering as authorized users of the warehouse management system. For the registration use-case, ask four questions and answers to those questions will be the objects interacting with each other for the registration process.

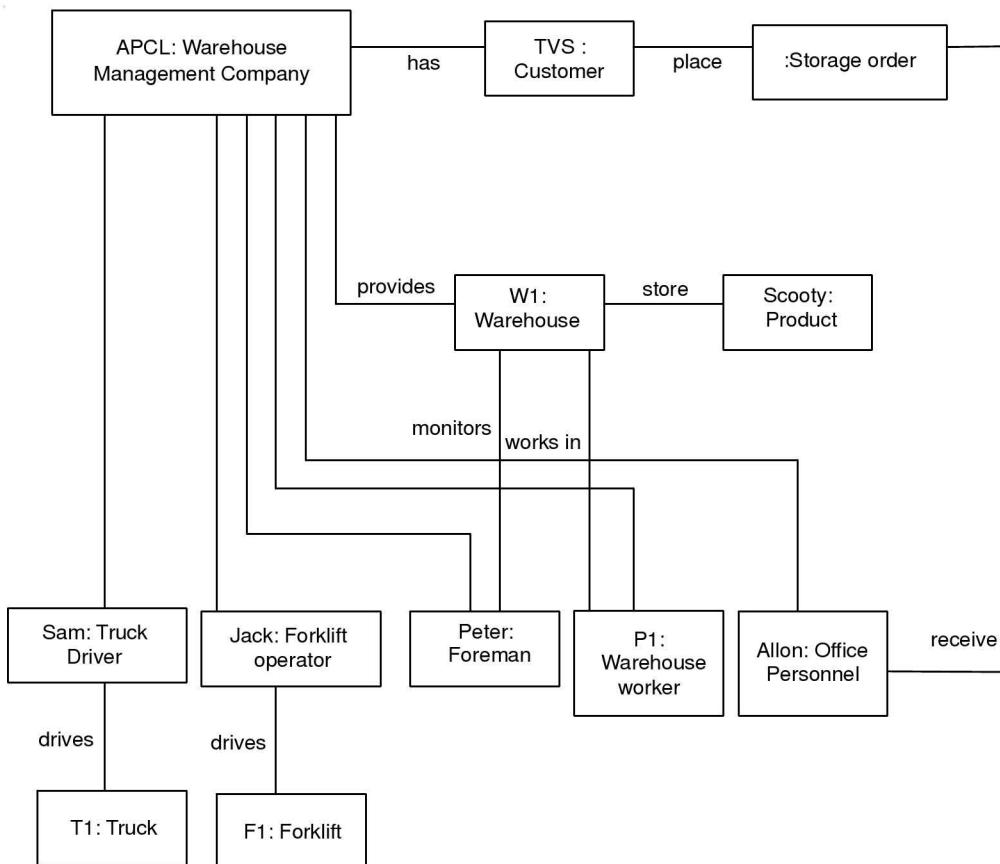


FIGURE 11.16 Object diagram for warehouse management system.

TABLE 11.4 Actors and corresponding use-cases

Actors	Use-Cases
Customer	Registration Login Place order
Office personnel	Registration Login Manage customer orders
Foreman	Registration Login Monitor warehouse Handle redistribution orders
Warehouse worker	Registration Login Handle goods

1. Who will register? (Actor)
Customer/Employee.
2. Through which web page (interface)? (Boundary)
RegistrationForm
3. Who will add the user? (Control)
RegistrationController.
4. Which object holds valid user details? (Entity)
User (Customer, Employee).

After identifying the interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls as shown in Table 11.5.

TABLE 11.5 Classes and corresponding methods for registration use-case

Class Type	Classes	Methods
Actor	Customer/Employee	<i>Not required to specify in this context.</i>
Boundary	RegistrationForm	fillForm()
Control	RegistrationController	receiveDetails() validateDetails()
Entity	User	addUser()

For online registration, the sequence flow will be as follows:

1. Customer/Employee will open the Registration Form.
2. Fill up the registration details on the form (Name, Address, DOB, etc.).
3. Click on the Submit button to submit the details.
4. Before submitting the details to the server, validation for all the fields on the form (Valid e-mail, phone No., etc.) will be carried out.
5. After form validation, the Registration controller object will validate the user, i.e. it will check if the Customer/Employee with the same details already exists or not or any other validation.
6. After user validation, if the user is valid, the user will be added to the database and user and registration confirmation message will be sent to the user.
7. After user validation, if the user is not valid, the user will not be added to the database and registration cancellation message will be sent to the user.

For steps 6 and 7, a combined fragment is used, where there are two alternatives—Valid User and Invalid User. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as illustrated in Figure 11.17.

Sequence diagram for the login use-case

The use-case *Login* is performed by both actors Customer and Employee of APCL Company for an authorized access to the warehouse management system.

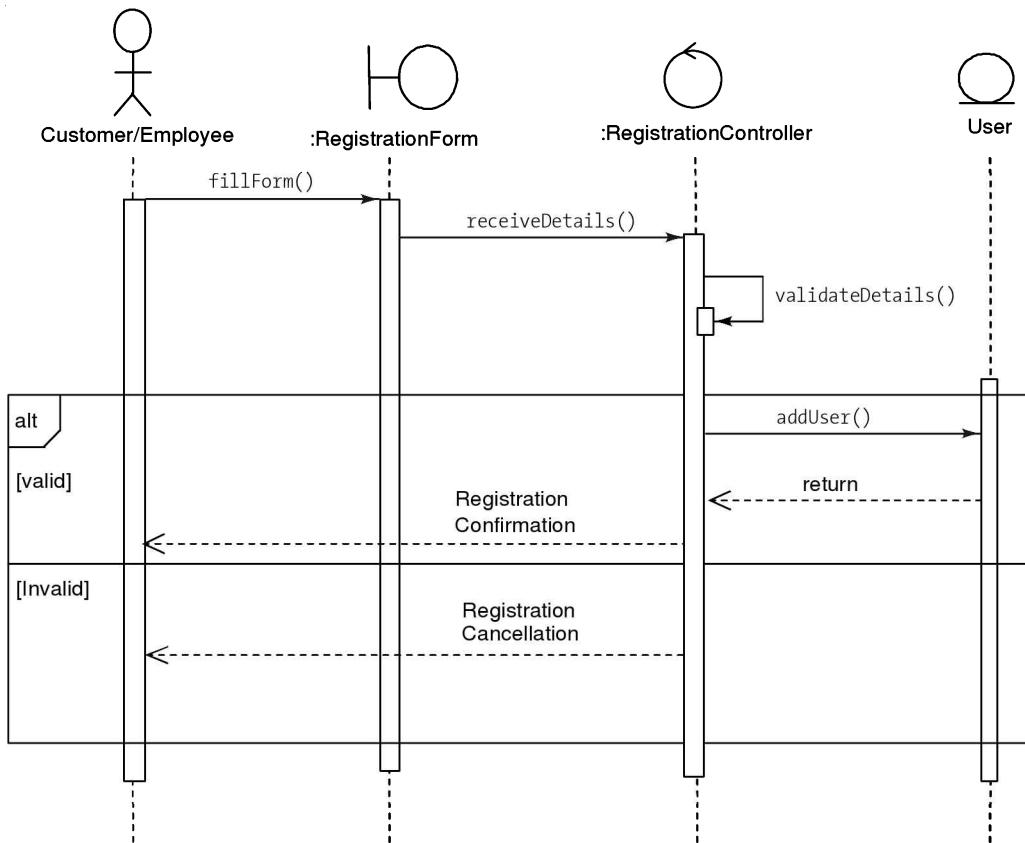


FIGURE 11.17 Sequence diagram for registration use-case.

For the login use-case, ask four questions and answers to those questions will be the objects interacting with each other for the login process.

1. Who will login? (Actor)
Customer/Employee.
2. Through which web page (interface)? (Boundary)
LoginForm.
3. Who will validate the user? (Control)
LoginController.
4. Which object holds valid user details? (Entity)
Users.

After identifying the interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls listed in Table 11.6. The corresponding sequence diagram is shown in Figure 11.18.

TABLE 11.6 Classes and their corresponding methods for login use-case

Class Type	Classes	Methods
Actor	Customer/Employee	<i>Not required to specify in this context.</i>
Boundary	LoginForm	setUserDtls()
Control	LoginController	getUserDtls() validateUser()
Entity	Users	retrieveUserDetails()

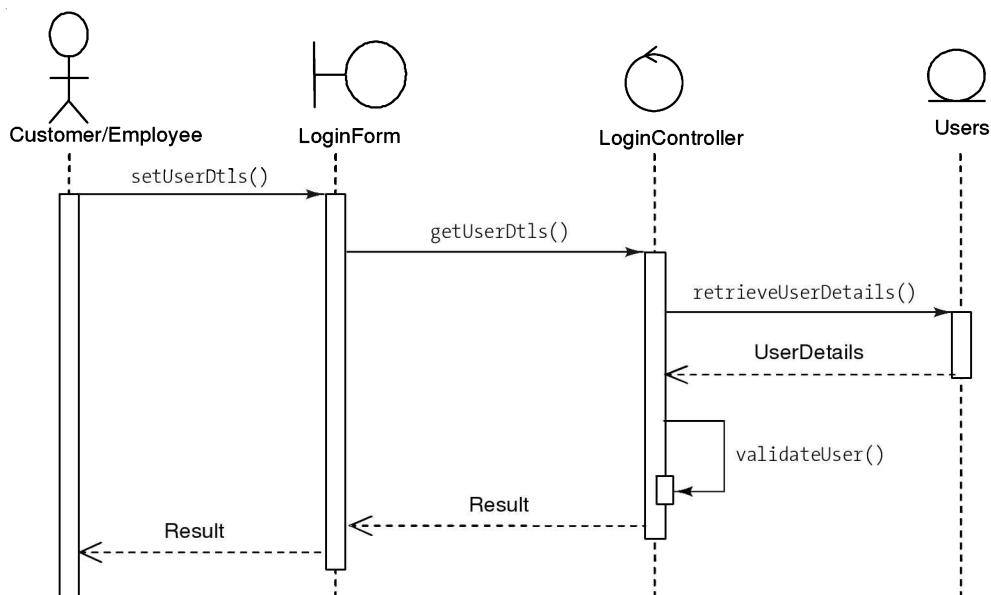


FIGURE 11.18 Sequence diagram for login use-case.

Sequence diagram for place order use-case

The use-case *Place Order* is performed by actor Customer for placing an order either for storage space or redistribution of goods.

For the place order use-case, ask four questions and answers to those questions will be the objects interacting with each other for performing the process.

1. Who will place order? (Actor)
Customer.
2. Through which web page (interface)? (Boundary)
OrderForm.

3. Who will coordinate order? (Control)
OrderController.
4. Which object holds order details? (Entity)
Order.

After identifying the interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls as shown in Table 11.7.

TABLE 11.7 Classes and their corresponding methods for place orders use-case

Class Type	Classes	Methods
Actor	Customer	<i>Not required to specify in this context.</i>
Boundary	OrderForm	fillOrderForm()
Control	OrderController	ReceiveOrder() validateOrder()
Entity	OrderDB	addOrder()

For placing an order online, the sequence flow will be as follows:

Precondition for this use case is that, the *user must login*.

1. Customer will open the Order Entry Form.
2. Fill up the order details on the form (Cust Name, Address, Products, etc.) and submit the form.
3. After submitting, order validation will be carried out.
4. After the order validation, if the order is valid, it will be added to the database and the order table and order confirmation message will be sent to the customer.
5. If the order is not valid, the order will not be added to the database and an order cancellation message will be sent to the customer.

For steps 4 and 5, a combined fragment is used, where there are two alternatives—Valid Order and Invalid Order. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as depicted in Figure 11.19.

Sequence diagram for manage customer orders use-case

The use-case *Manage Customer Orders* is performed by actor Office Personnel for receiving orders from customers and conveying it to the appropriate department in the company.

For the Manage Customer Orders use-case, ask four questions and answers to those questions will be the objects interacting with each other for executing the process.

1. Who will manage customer orders? (Actor)
OfficePersonnel.
2. Through which web page (interface)? (Boundary)
OrderManageForm.

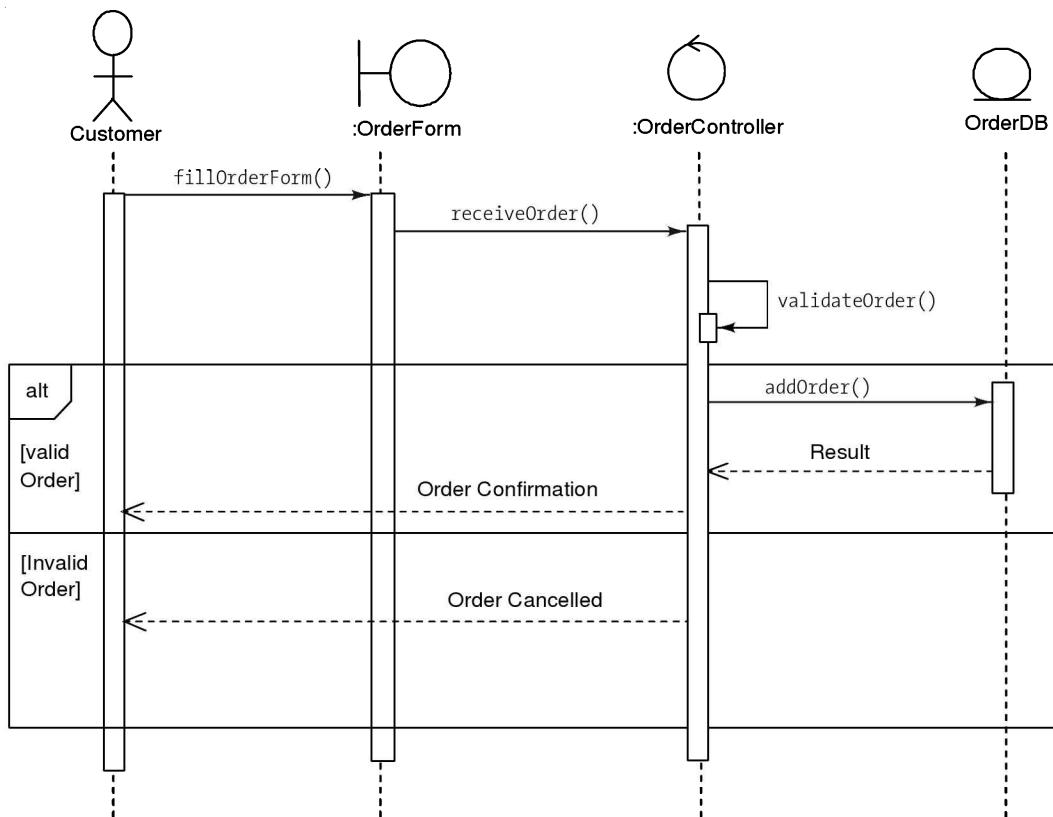


FIGURE 11.19 Sequence diagram for place order use-case.

3. Who will coordinate the process? (Control)
OrderManageController.
4. Which object hold Order details? (Entity)
OrderDB.

After identifying the interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls. All this is listed in Table 11.8.

For the manage customer orders, the sequence flow will be as follows:

Precondition for this use-case is that the *user must login*.

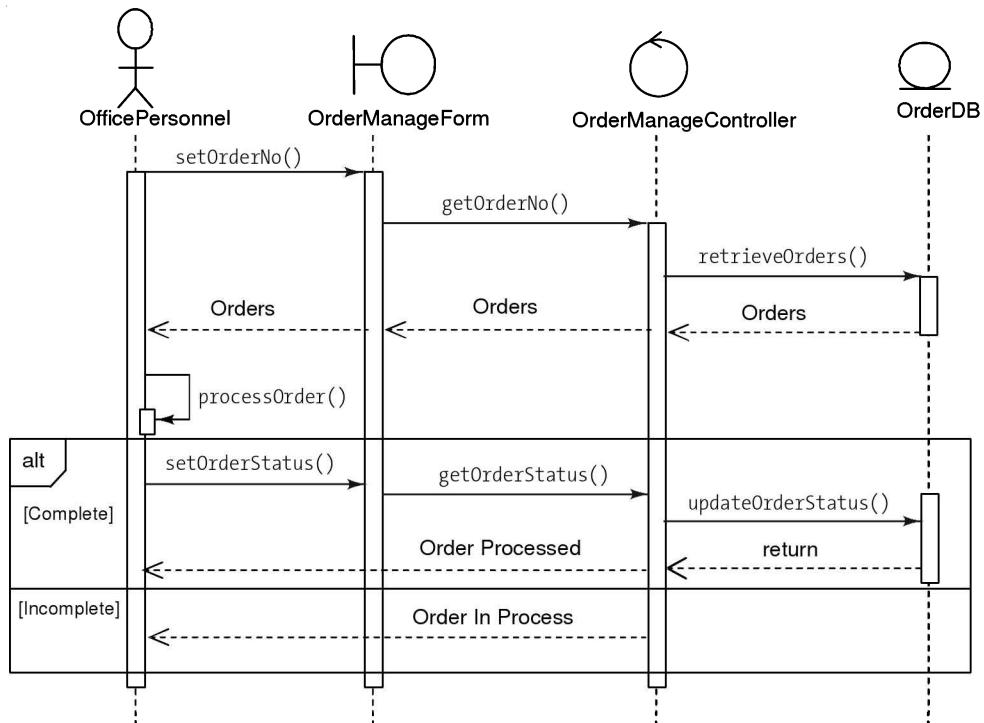
1. Office Personnel will open the OrderManageForm, set order number on the form and submit it.
2. After submitting the form, the OrderManageController will retrieve the orders from the OrderDB table and send it to the Office Personnel.
3. Then the Office Personnel process the Order, i.e. redirect the orders to appropriate departments for processing.

TABLE 11.8 Classes and their corresponding methods for manage customer orders use-case

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	OfficePersonnel	<i>Not required to specify in this context.</i>
Boundary	OrderManageForm	setOrderNo() processOrder() setOrderStatus()
Control	OrderManageController	getOrderNo() getOrderStatus()
Entity	OrderDB	retrieveOrders() updateOrderStatus()

4. If the order is processed completely, then set the corresponding order status and update it in the OrderDB table with the message “Order Processed”.
5. If the order is not processed completely, then the message “Order In Process” will be shown.

For steps 4 and 5, a combined fragment is used, where there are two alternatives—Complete and Incomplete. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used (Figure 11.20).

**FIGURE 11.20** Sequence diagram for managing customer order use-case.

Sequence diagram for monitoring warehouse

The use-case *Monitor warehouse* is performed by actor *Foreman* for receiving and managing the goods in the warehouse.

For the monitor warehouse use-case, ask four questions and answers to those questions will be the objects interacting with each other for the process.

1. Who will monitor warehouse? (Actor)
Foreman.
2. Through which web page (interface)? (Boundary)
MonitorForm.
3. Who will coordinate the process? (Control)
warehouseMonitor.
4. Which object holds item information stored in the warehouse? (Entity)
ItemDB.

After identifying the interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls (Table 11.9).

TABLE 11.9 Classes and their corresponding methods for monitor warehouse use-case

Class Type	Classes	Methods
Actor	Foreman	<i>Not required to specify in this context.</i>
Boundary	MonitorForm	selectItemType()
Control	warehouseMonitor	getItemType() generateItemReport()
Entity	ItemDB	retrieveItems()

For *monitoring warehouse*, the sequence flow will be as follows:

Precondition for this use case is that the *User must login*.

1. Foreman opens the MonitorForm to select the Item type to monitor and submit it.
2. warehouseMonitor will receive the Item type and retrieve items under that type.
3. Then the Item Report is generated and displayed to the Foreman for monitoring it.

The sequence diagram is shown in Figure 11.21.

Sequence diagram for handle orders use-case

The use-case *Handle orders* is performed by actor *Foreman* for receiving orders from the office Personnel and if the order is for storing items in the warehouse, instruct warehouse workers to unload items received from the customer, or if the order is for redistribution of items, then instruct workers to load it in a truck.

For the handle orders use-case, ask four questions and answers to those questions will be the objects interacting with each other for the process.

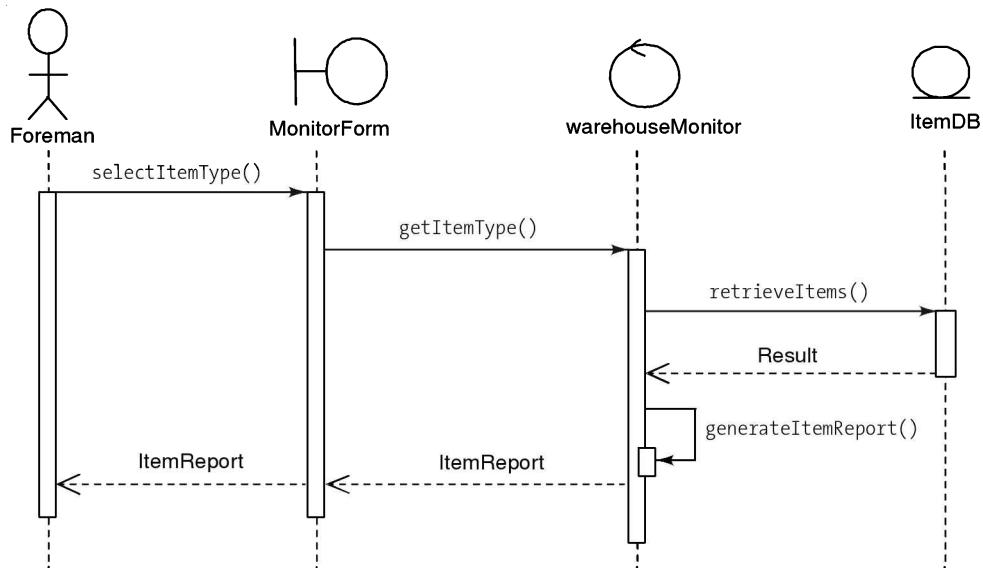


FIGURE 11.21 Sequence diagram for monitor warehouse use-case.

1. Who will handle orders in warehouse? (Actor) Foreman.
2. Through which web page (interface)? (Boundary) OrderHandlingForm.
3. Who will coordinate the process? (Control) OrderHandlingController.
4. Which object hold order information? (Entity) OrderDB.

After identifying the interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls (Table 11.10).

TABLE 11.10 Classes and their corresponding methods for handle orders use-case.

<i>Class Type</i>	<i>Classes</i>	<i>Methods</i>
Actor	Foreman	<i>Not required to specify in this context.</i>
Boundary	OrderHandlingForm	<code>selectOrder()</code>
Control	OrderHandlingController	<code>getOrder()</code>
Entity	OrderDB	<code>retrieveOrders()</code>
Actor	Warehouse worker	<code>instructToLoad()</code> <code>instructToUnLoad()</code>

For the handle order use-case, the sequence flow will be as follows:

Precondition for this use-case is that the *user must login*.

1. Customer will open the OrderHandlingForm, select the order to handle and submit it.
2. OrderHandlingController will receive the order from the Foreman and retrieve the order from the OrderDB table.
3. By looking at what type of order it is either redistribution or storage order, the Foreman will instruct the workers.
4. If the order is redistribution order, the Foreman will instruct the Warehouse workers to load the truck with the items for redistribution.
5. If the order is storage order, the Foreman will instruct the Warehouse workers to unload the items from the truck into the warehouse for storage.

For steps 4 and 5, a combined fragment is used, where there are two alternatives—Redistribution Order and Storage Order. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used in Figure 11.22.

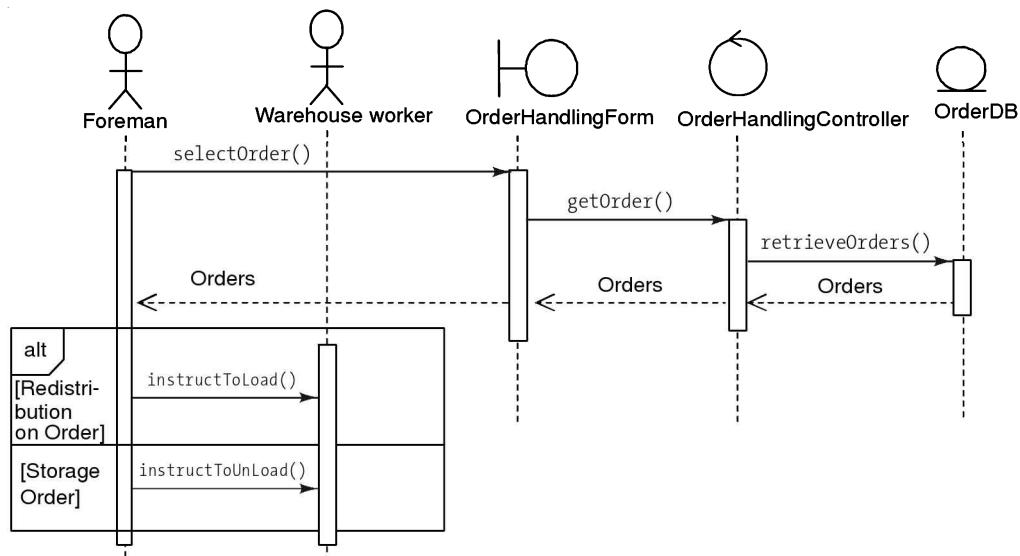


FIGURE 11.22 Sequence diagram for handle order use-case.

Sequence diagram for handle goods use-case

The use-case *Handle Goods* is performed by actor *Warehouse worker* for loading goods in the truck for redistribution purpose and unloading goods from the truck into warehouse for storage.

For handle goods use-case, ask four questions and answers to those questions will be the objects interacting with each other for the handle goods process.

1. Who will handle goods? (Actor)
Warehouse worker.

2. Through which web page (interface)? (Boundary)
GoodsHandlingForm.
3. Who will coordinate the process? (Control)
GoodsHandlingController.
4. Which object holds Item details? (Entity)
ItemDB.

After identifying interacting objects, it is required to find out what sequence of message communication happens among them. Message communication occurs through method calls Table (11.11).

TABLE 11.11 Classes and their corresponding methods for handle goods use-case

Class Type	Classes	Methods
Actor	Warehouse worker	<i>Not required to specify in this context.</i>
Boundary	GoodsHandlingForm	readBarcode()
Control	GoodsHandlingController	getBarcode()
Entity	ItemDB	addItem() removeItem()

For the handle goods, the sequence flow will be as follows:

Precondition for this use case is that the *user must login*.

1. Warehouse worker will open the GoodsHandlingForm, enter the barcode of the item and submit the form.
2. If the items are unloaded from the truck and getting stored into the warehouse, then the GoodsHandlingController will add the item into the ItemDB table.
3. If the items are to be loaded into the truck from the warehouse for the redistribution, then the GoodsHandlingController will remove the item into the ItemDB table.

For steps 2 and 3, a combined fragment is used, where there are two alternatives—Unloading from the truck and loading into the truck. To demonstrate the sequence flow for both alternatives, in the same sequence diagram, the combined fragment is used as shown in Figure 11.23.

11.3.2 Collaboration Diagram

Collaboration diagram for registration use-case

The objects, links and their corresponding methods for the registration use-case are listed in Table 11.12. Figure 11.24 gives the collaboration diagram for, the registration use case.

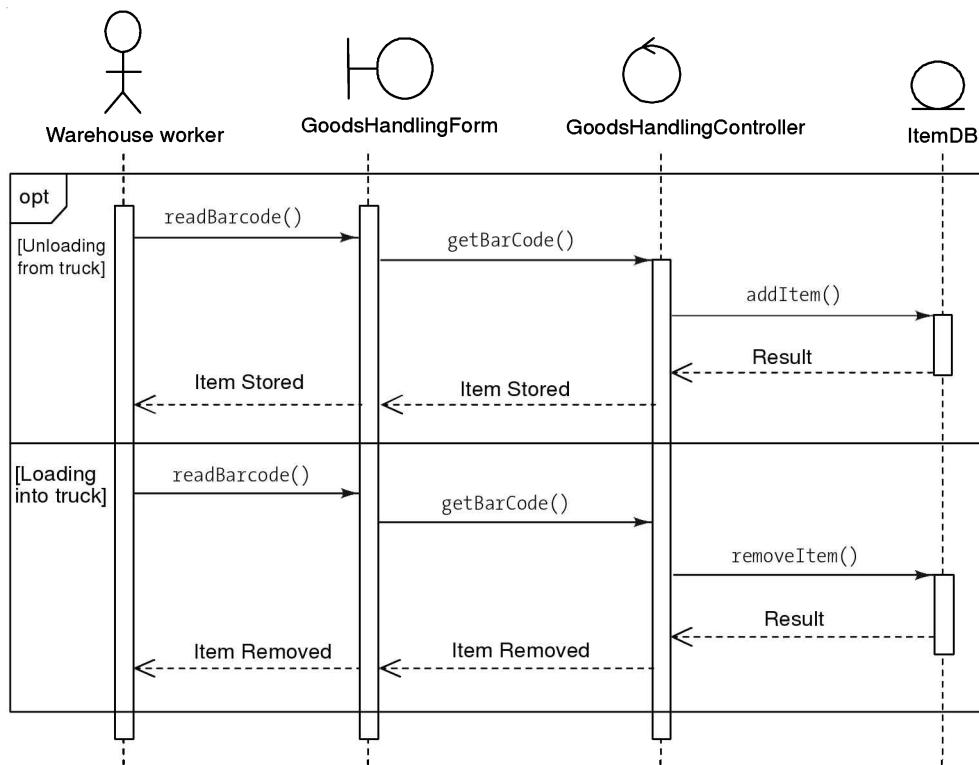


FIGURE 11.23 Sequence diagram for handle goods use-case.

TABLE 11.12 Objects, links and their corresponding methods for registration use-case

Objects	Links	Methods (with sequence number)
Customer/ Employee	Customer/Employee—RegistrationForm	1. <code>fillForm()</code> 2. <code>receiveDetails()</code>
RegistrationForm	RegistrationForm—RegistrationController	3. <code>validateDetails()</code>
RegistrationController User	RegistrationController—User	4. <code>addUser()</code> 5. Registration confirmation/ Cancelled

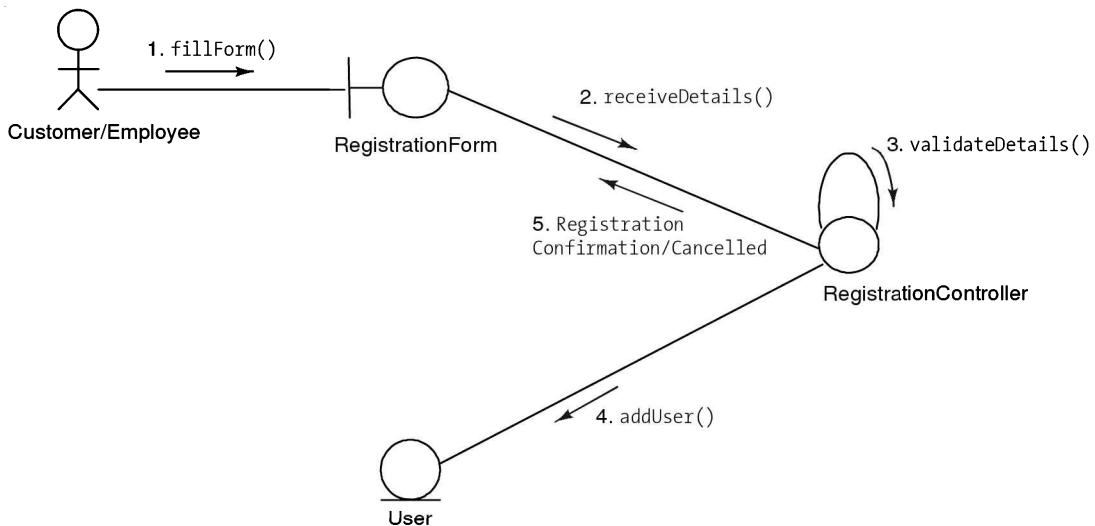


FIGURE 11.24 Collaboration diagram for registration use-case.

Collaboration diagram for login use-case

Table 11.13 lists the objects, links and their corresponding methods. The collaboration diagram for the login use-case is shown in Figure 11.25.

TABLE 11.13 Objects, links and their corresponding methods for login use-case

Objects	Links	Methods (with sequence number)
Customer/Employee	Customer/Employee—LoginForm	1. setUserDts()
LoginForm	LoginForm—LoginController	2. getUserDts()
LoginController	LoginController—User	3. retrieveUserDetails 4. UserDetails 5. validateUser() 6. Result
User		

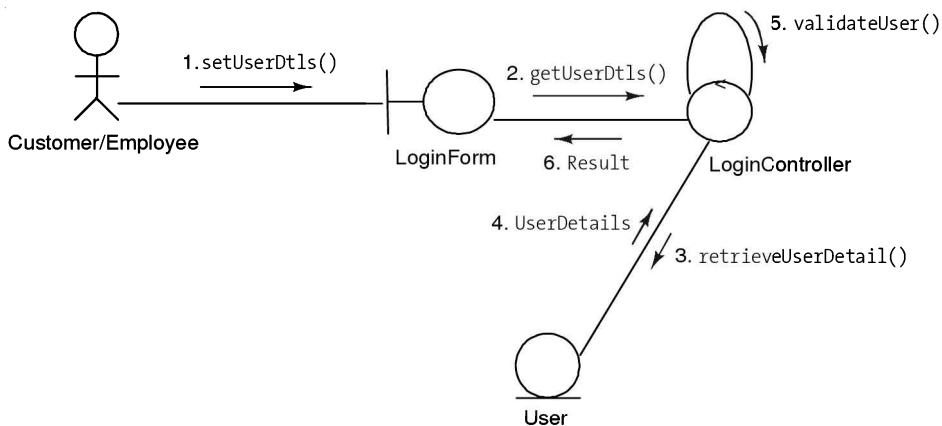


FIGURE 11.25 Collaboration diagram for login use-case.

Collaboration diagram for placing order use-case

Take 11.14 gives the objects, links and methods for the placing order use-case and the collaboration diagram is shown in Figure 11.26.

TABLE 11.14 Objects, links and their corresponding methods for placing order use-case

<i>Objects</i>	<i>Links</i>	<i>Methods (with sequence number)</i>
Customer	Customer—OrderForm	1. fillOrderForm()
OrderForm	OrderForm—OrderController	2. receiveOrder()
OrderController	OrderController—OrderDB	3. validateOrder() 4. addOrder() 5. Result 6. Order Confirmation/ Cancellation
OrderDB		

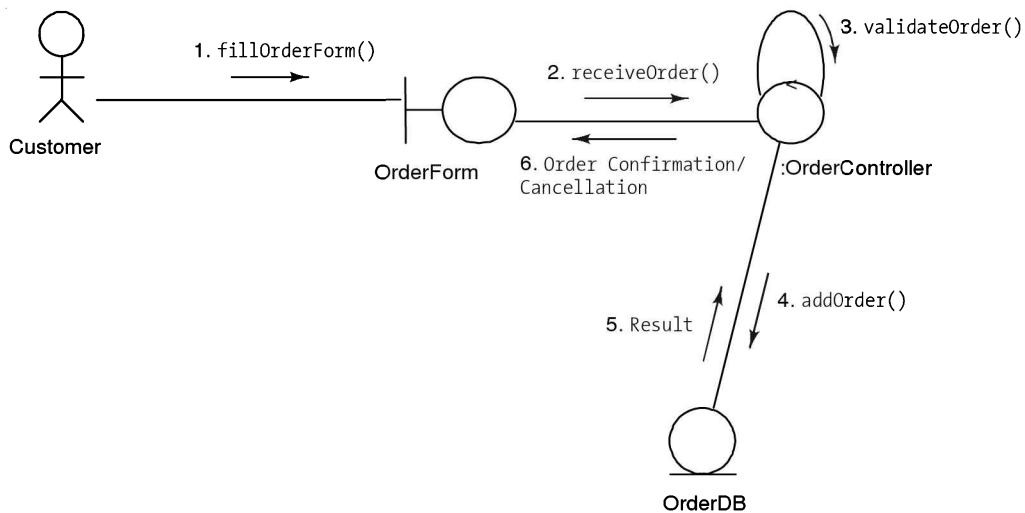


FIGURE 11.26 Collaboration diagram for placing order use-case.

Collaboration diagram for manage customer orders use-case

The objects, links and methods for the manage orders use-case are given in Table 11.15 and the collaboration diagram is shown in Figure 11.27.

TABLE 11.15 Objects, links and their corresponding methods for manage customer order use-case

Objects	Links	Methods (with sequence number)
OfficePersonnel	OfficePersonnel— OrderManageForm	1. setOrderNo()
OrderManageForm	OrderManageForm— OrderManageController	2. getOrderNo()
OrderManageController	OrderManageController— OrderDB	3. retrieveOrders() 4. Orders 5. processOrder() 6. setOrderStatus() 7. getOrderStatus() 8. updateOrderStatus()
OrderDB		

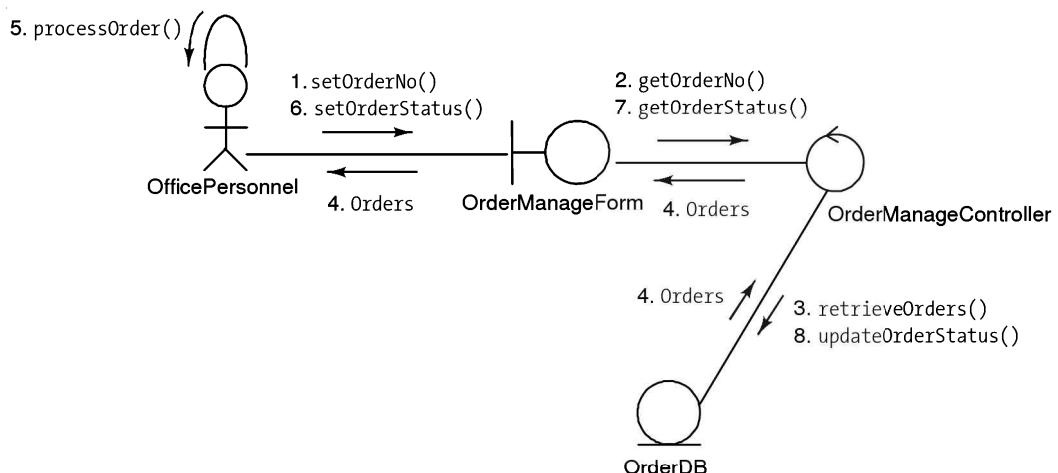


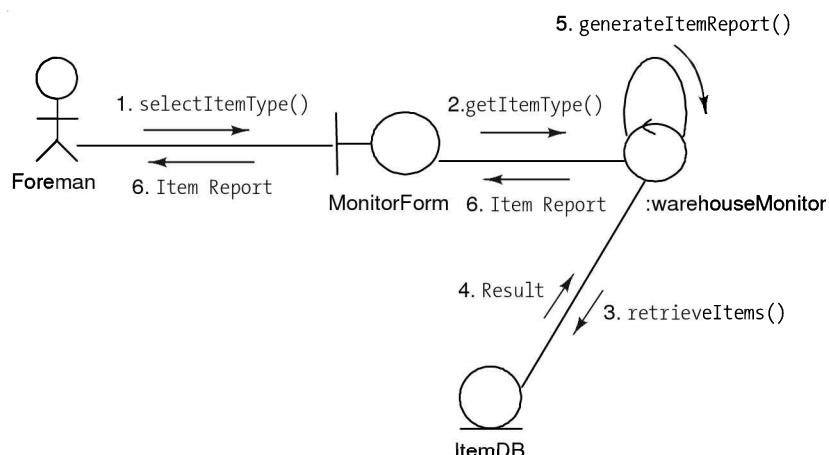
FIGURE 11.27 Collaboration diagram for managing customer order use-case.

Collaboration diagram for monitor warehouse use-case

The objects, links and methods for the monitor warehouse use-case are listed in Table 11.16 and the collaboration diagram is shown in Figure 11.28.

TABLE 11.16 Objects, links and their corresponding methods for monitor warehouse use-case

<i>Objects</i>	<i>Links</i>	<i>Methods (with sequence number)</i>
Foreman	Foreman—MonitorForm	1. selectItemType()
MonitorForm	MonitorForm—warehouseMonitor	2. getItemType()
warehouseMonitor	warehouseMonitor—ItemDB	3. retrieveItems()
ItemDB		4. Result 5. generateItemReport() 6. Item Report

**FIGURE 11.28** Collaboration diagram for monitor warehouse use-case.**Collaboration diagram for handle orders use-case**

The objects, links and corresponding methods for the handle orders use-case are listed in Table 11.17 and the collaboration diagram is shown in Figure 11.29.

Table 11.17 Objects, links and their corresponding methods for handle orders use-case

<i>Objects</i>	<i>Links</i>	<i>Methods (with sequence number)</i>
Foreman	Foreman—OrderHandlingForm	1. selectOrder()
OrderHandlingForm	OrderHandlingForm—OrderHandlingController	2. getOrder()
OrderHandlingController	OrderHandlingController—OrderDB	3. retrieveOrders()
OrderDB	Foreman—Warehouse worker	4. Orders
Warehouse worker		5. instructToLoad()/ instructToUnLoad()

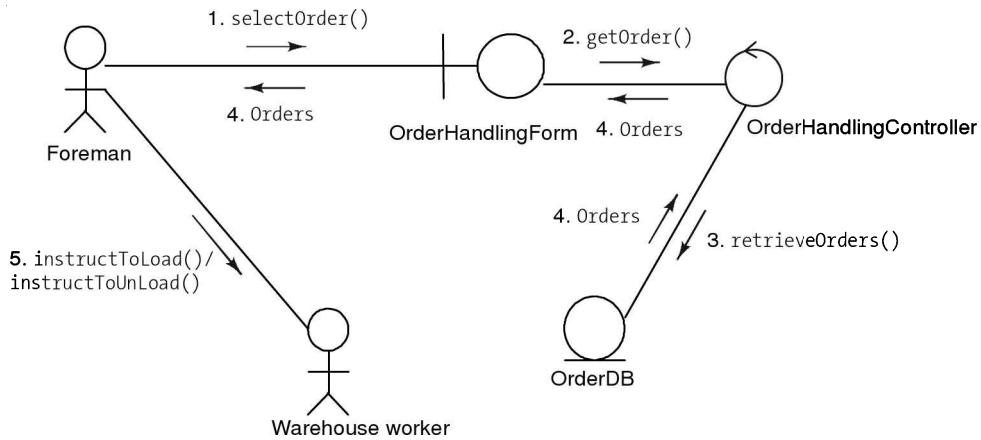


FIGURE 11.29 Collaboration diagram for handle order use-case.

Collaboration diagram for handle goods use-case

Table 11.18 lists the objects, links and methods for the handle goods use-case and Figure 11.30 shows the collaboration diagram.

TABLE 11.18 Objects, links and their corresponding methods for handle goods use-case

Objects	Links	Methods (with sequence number)
Warehouse worker	Warehouse worker— GoodsHandlingForm	1. readBarcode()
GoodsHandlingForm	GoodsHandlingForm— GoodsHandlingController	2. getBarCode()
GoodsHandlingController	GoodsHandlingController—ItemDB	3. addItem() / removeItem() 4. Result 5. Item Stored / Item Removed
ItemDB		

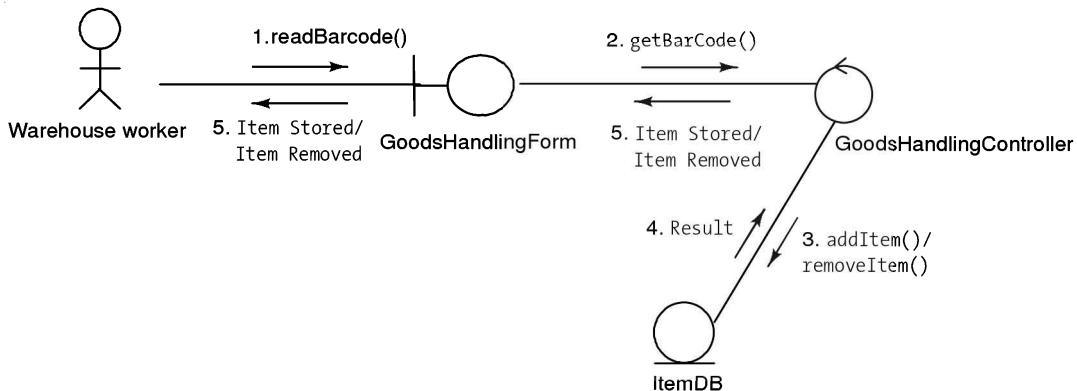


FIGURE 11.30 Collaboration diagram for handle goods use-case.

11.3.3 Statechart diagram

In this problem statement, the object for which the statechart diagram is analyzed is *Order*.

The *Order* object has fixed states throughout its life cycle and there may be some abnormal exits also. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as the complete transaction.

The first state is the waiting state from where the process starts. The next states are arrived for events like *send request*, *confirm request*, *cancel order* and *complete transaction*. These events are responsible for the state changes of the order object.

As the order object has finite states throughout its life cycle, the statechart diagram will be one shot life cycle statechart diagram. Hence there will be one initial state and one or more final states as in Figures 11.31(a) and (b). The intermediate states are shown in Figure 11.31(c). Figure 11.32 shows the transitions.



FIGURE 11.31(a) Initial state: Begin order.



FIGURE 11.31(b) Final state: End order.

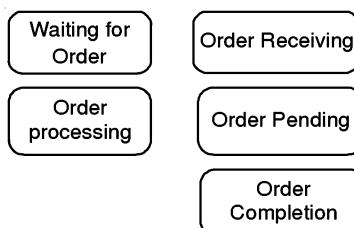


FIGURE 11.31(c) Intermediate states.

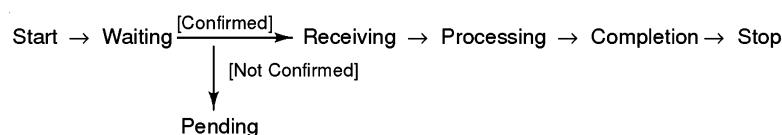


FIGURE 11.32 Transitions.

All the transitions in the above process are trigger-less transitions which occur on completion of the activity during the previous state. The guard condition to enter from *Receiving* state into the *Processing for Completion state* is *confirmed/Not confirmed*. No trigger is required to transit from the state. The complete state transition diagram for the *Order* object is as shown in Figure 11.33.

11.3.4 Activity Diagram

The whole process in the system for which the activity diagram should be drawn is:

- Order (Warehouse space or Redistribution) management

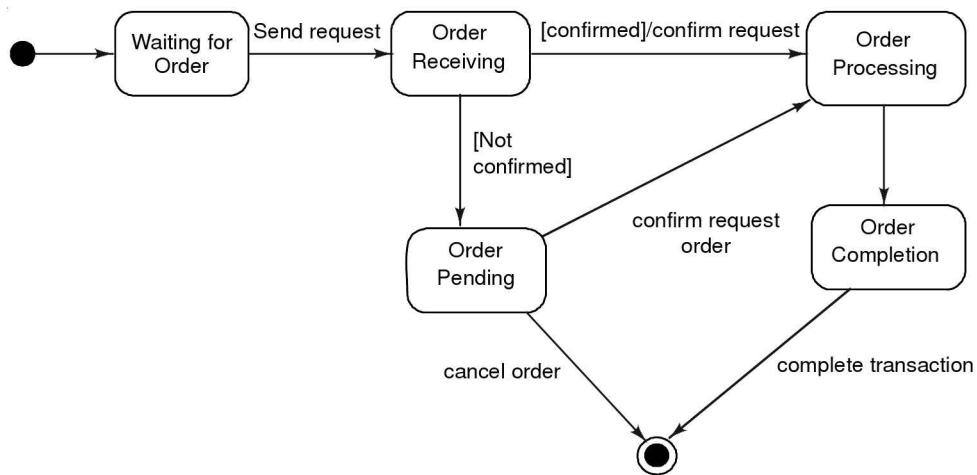


FIGURE 11.33 State transition diagram for order object.

For drawing an activity diagram for any process we,

1. Find out swimlanes if any. To find swimlanes, see if you can span some activities over different organizational units/places.
2. Find out in which swimlane the order management process begins and where it ends. Those will be the initial and final states.
3. Then, identify activities occurring in each swimlane. Arrange the activities in a sequence flow spanning over all the swimlanes.
4. Identify the conditional flow or parallel flow of activities. The parallel flow of activities must converge at a single point using a join bar.
5. During the activities are performed, if any document is generated or used, take it as an object and show the object flow.

Activity diagram for warehouse management system

Beginning with the customer placing an order, till the order gets completed, the following activities are happening in the company.

1. Customer places an online order for warehouse space or redistribution of goods after login.
2. Office personnel receive orders.
3. (a) If the order is for warehouse space, after confirmation the foreman in warehouse receives goods from the customer.
 (b) Foreman instructs warehouse workers to unload goods from the truck.
 (c) Warehouse workers unload goods, barcode items and insert items in the warehouse.
4. (a) If the order is for redistribution of goods, the foreman instructs the warehouse workers to load goods in a truck from one warehouse.
 (b) Truck driver transports goods to the other warehouse.
 (c) In the other warehouse, the foreman instructs the warehouse workers to unload goods from the truck.

- (d) Warehouse workers unload goods, read barcode on each item and insert items in the warehouse.

5. Foreman monitors the warehouse.

From the above sequence of activities, there are five swimlanes identified as Customer, Office personnel, Foreman, Warehouse workers and Truck drivers as in Figure 11.34.

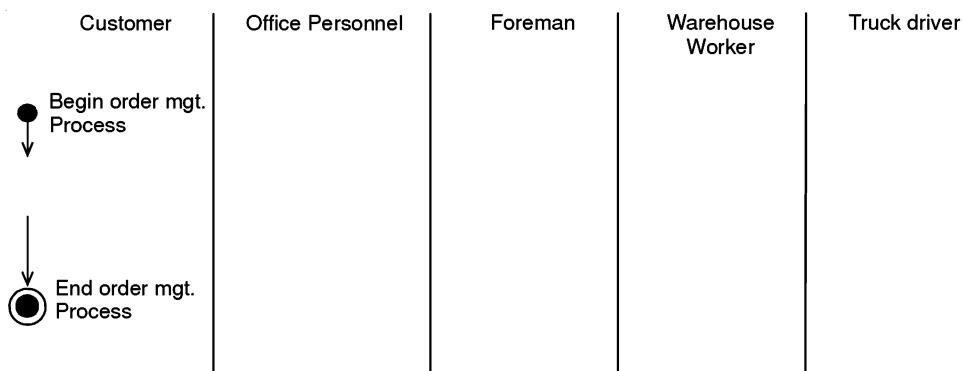


FIGURE 11.34 Swim lanes for order management.

The activities in each of these swimlanes are as shown in Figure 11.35.

As stated in the problem statement, no parallel flow is identified.

One conditional flow occurs after the *Verify user* activity which proceeds to place an order for warehouse space or redistribution of goods if the customer is a valid user or else again ask for the valid details to enter if the customer is not a valid user.

Another conditional flow occurs after the *Check if order is for warehouse space or Redistribution of goods* activity to check if the order is for warehouse space or redistribution of goods.

If the placed order is for warehouse space, receive goods from the customer. If the placed order is for redistribution of goods, load goods in a truck from one warehouse, transport it to another warehouse and unload it in the other warehouse. Each time use the barcode reader while loading and unloading of goods. So the complete activity diagram for the process with the transitions is as shown in Figure 11.36.

11.4 IMPLEMENTATION PHASE DIAGRAMS: WAREHOUSE MANAGEMENT SYSTEM

11.4.1 Component Diagram

The component diagrams illustrate the pieces of software that will make up a system.

So, the major components identified making up this system are as follows:

1. Warehouse management application
2. Employee

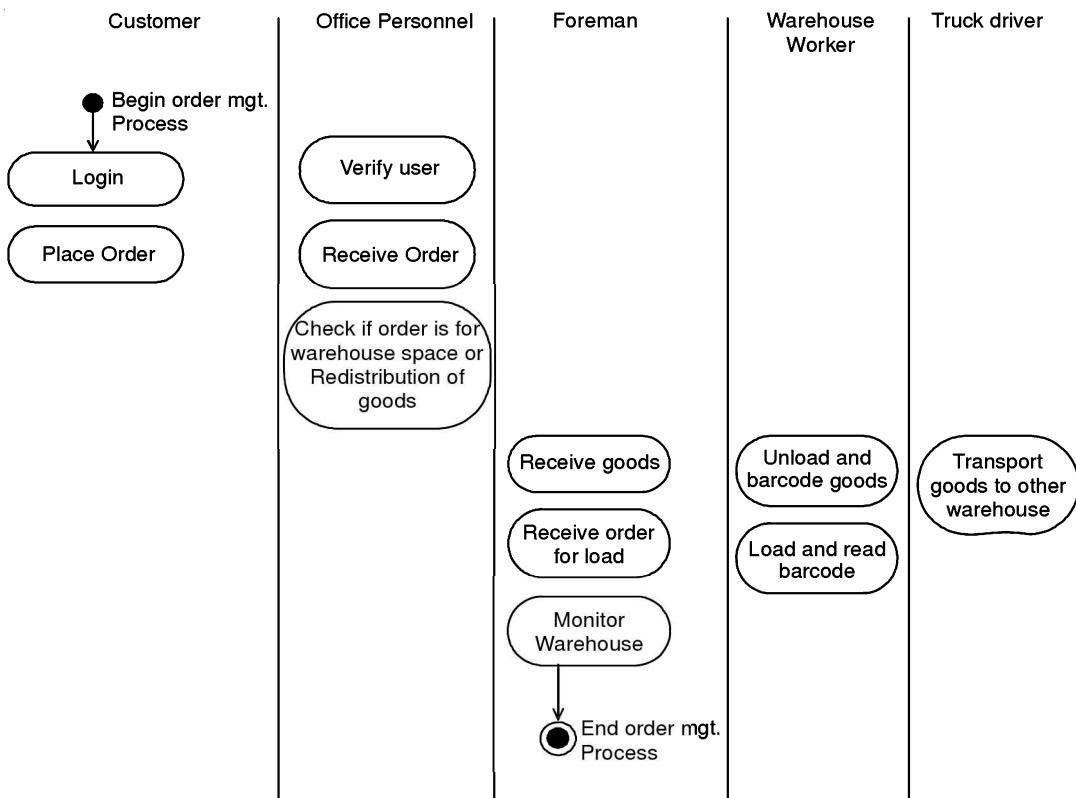


FIGURE 11.35 Activities for order management.

3. Customer
4. Order (Storage or Redistribution)
5. Warehouse and customer database
6. Security and persistence

The system is all about accepting the orders from the customers for warehouse space for storage of goods or redistributing goods from one warehouse to another and processing the order as shown in Figure 11.37.

The main application component in the system is Warehouse Management Application which depends on components identified are Employee, Customer and Order, which are implementing corresponding interfaces IEmployee, ICustomer, and IOrder.

Persistence and Security components represent functional components for storing data into database represented by Warehouse and Customer database component which is the data store component and managing the access control to the system.

11.4.2 Deployment Diagram

- Warehouse management System is an Internet-based application.
- Warehouse management System is based on multi-tier architecture.

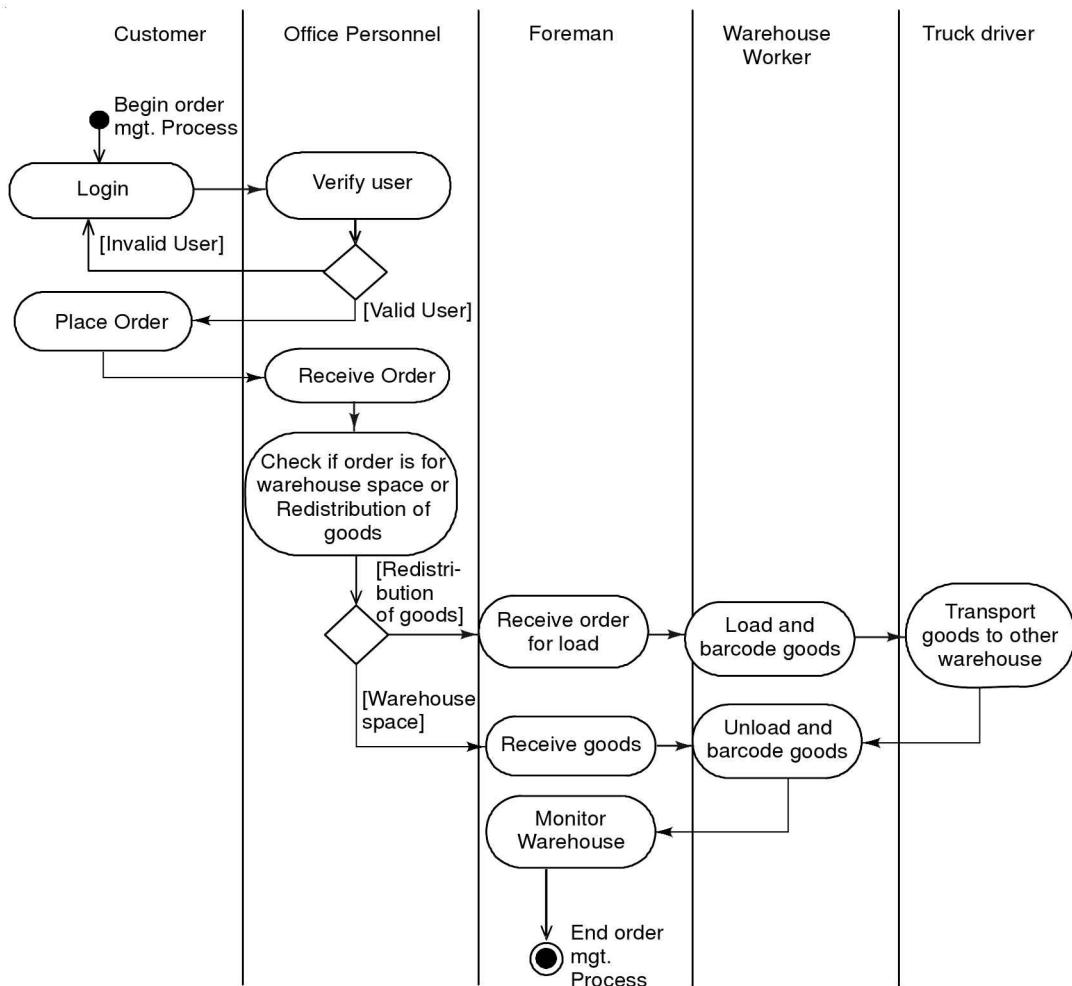


FIGURE 11.36 Activity diagram for order management.

- Hence there will be a database tier holding the warehouse and customer database, an application tier holding the warehouse management application, a client tier holding the client workstation with a browser, which will be used by customers as well as company's employees.
- Network used is WAN.
- Data communication among all the nodes is through the TCP/IP protocol.

In Figure 11.38, the database server is representing the database layer on which the database server, e.g. MS-SQL Server software and Warehouse and Customer database will be installed.

The application server is representing the application tier on which application server software, e.g. the WebSphere application server and warehouse management application (all source code pages, dynamic link libraries, etc.) will be deployed.

The customers and employees of the company access the application through the browser. Hence on the client tier only a browser is required.

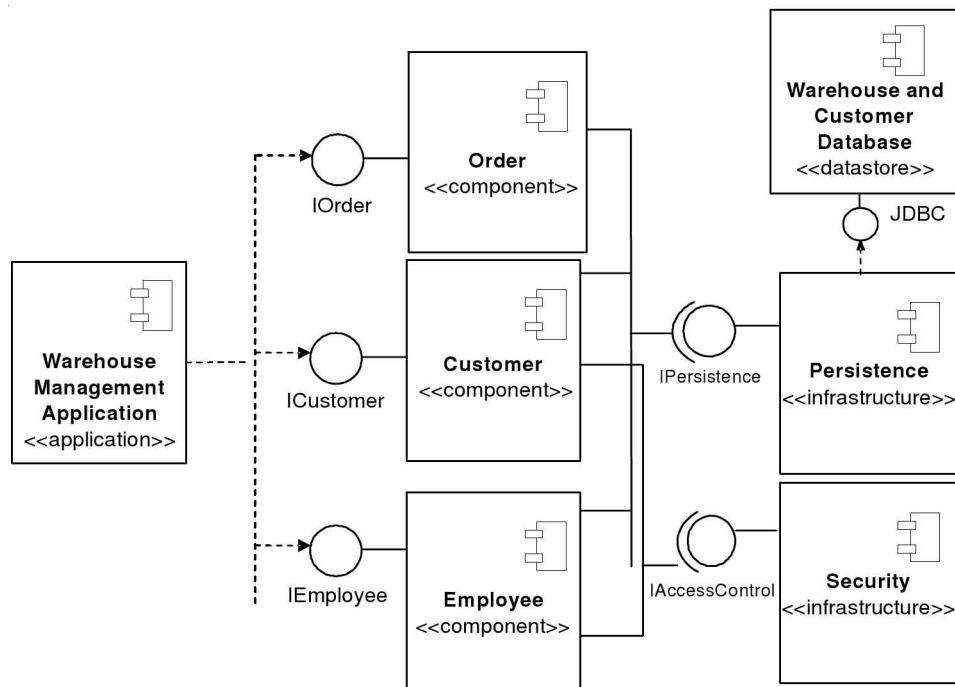


FIGURE 11.37 Component diagram for warehouse management system.

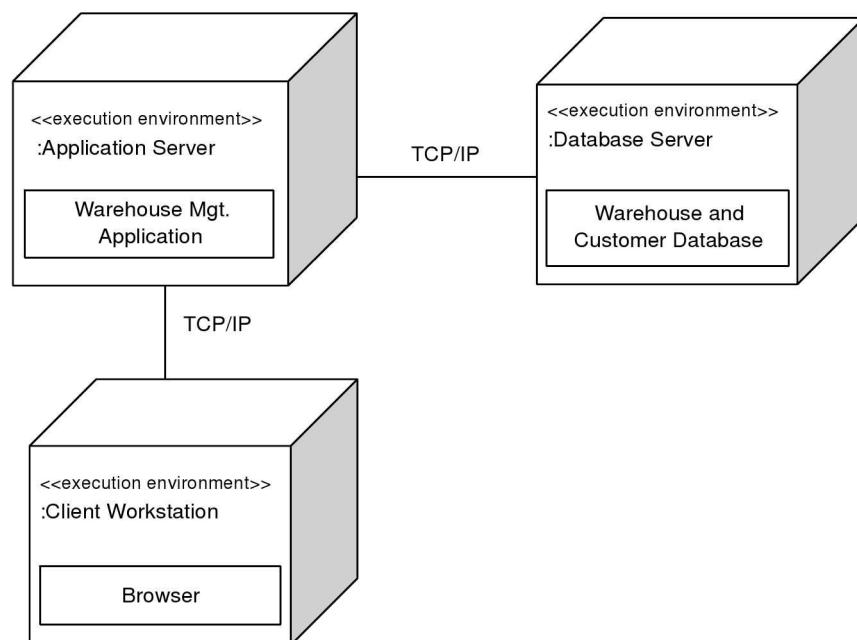


FIGURE 11.38 Deployment diagram for warehouse management system.

CHAPTER

12

Existing Object Oriented Methodologies

12.1 OBJECT ORIENTED METHODOLOGIES AND UML

Since the time object-oriented programming had proven best for building applications and systems, engineers needed a way of presenting large, complex concepts in a simple graphical manner.

Obviously, there is no magic that can lead the software engineer down the path from requirements to software implementation. Design of complex software systems needs an incremental and iterative process. As a cookbook approach is impractical, still, sound design methods bring some much-needed discipline to the development process. The software engineering community has evolved dozens of different object-oriented analysis and design methodologies.

A majority of OOAD methodologies are based on the same foundation of objects, classes, inheritance and relationships. Apart from slight notational and process differences, the emphasis given on different phases by each methodology also matters. A well-defined and expressive notation is important to the process of software development. A standard notation makes it possible for an analyst or a developer to describe a scenario or formulate architecture and then unambiguously communicate those decisions to others.

Among all the emerging object-oriented methodologies, the most popular ones are the Booch Notation designed by Grady Booch, the Object Modelling Technique (OMT) designed by James Rumbaugh, Object-Oriented Analysis designed by Coad and Yourdon, and Object-Oriented Software Engineering by Ivar Jacobson. All these notations were slightly different from each other. For example, Booch's symbol for a class versus OMT's symbol for a class.

As a developer, focus of attention should be given on the OOAD process and not on the notation. Many OOAD processes are based on UML. The point of focus is the OOAD process itself, particularly as a means for capturing requirements and behaviour.

There were some attempts to unify the concepts among object-oriented methodologies. The first successful attempt to combine and unify existing methodologies took place when Rumbaugh, Booch and Jacobson came together at Rational Software Corporation in 1995. The result of their combined efforts was Unified Modelling Language (UML). In 1996, the Object Modelling Group (OMG) issued a request for proposals for a standard approach to object-oriented modelling. UML authors Booch, Rumbaugh and Jacobson worked with the methodologists and developers from different companies to produce a modelling language that would be widely accepted by tool makers, methodologists and developers who would be the users. The final UML proposal was submitted to the OMG in September 1997.

We will have a quick look at the methodologies and their modelling notations developed by James Rumbaugh, Grady Booch, Ivar Jacobson, Coad–Yourdon and Rational Unified Process.

12.2 DIFFERENT MODELS FOR OBJECT ANALYSIS AND DESIGN

Object technology has many different methodologies to help analyze and design computer systems. In most cases these methodologies are very similar, but each has its own way to graphically represent the entities. To understand and use these four methodologies would become difficult, if not impossible, for all projects. The most important point is that the final outcome is what really matters, not the choice of one analysis technique over another technique. Remember, it is more important to do proper analysis and design to meet user requirements than it is to just follow a meaningless procedure.

The object-oriented methodologies require a more iterative process with the same steps analysis, design, coding, testing and maintenance. The iterative process helps reduce confusion around what the system is really supposed to do and what the users really want. The object-oriented software development methods consider the fact that user requirements are going to be changed. However it does not matter which programming language you are using, be it C++ or Java. Furthermore, it does not matter which system development technique you use, you will follow the same five steps in the system development.

12.3 RUMBAUGH'S OBJECT MODELLING TECHNIQUE (OMT) METHODOLOGY

Rumbaugh's methodology is the closest to the traditional approach to system analysis and design. The Rumbaugh methodology has its primary strength in object analysis and design. The object modelling technique (OMT) presented by Rumbaugh has three deliverables the object model, the dynamic model, and the functional model. These three models are similar to traditional system analysis, with the additions for the object model, including definitions of classes along with the classes variables and behaviours.

12.3.1 Object Model

The object model describes the structure of objects in a system: their identity, relationships to other objects, attributes and operations. The object model is represented by an object diagram. The object diagram contains classes connected by association lines. Each class represents a set of individual objects. The association lines represents relationships among classes.

12.3.2 Object Modelling Notations

Figure 12.1 shows the object modelling notations.

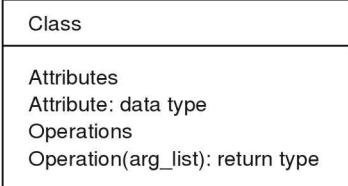
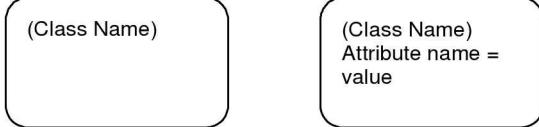
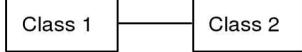
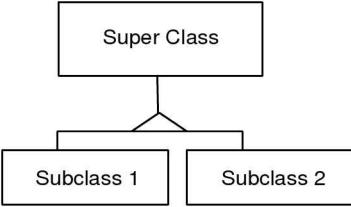
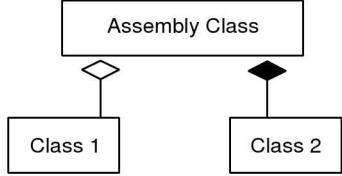
	Class Diagram
	Object Instance
	Association
	Generalization (Inheritance)
	Aggregation Composition

FIGURE 12.1 Object modelling notations.

12.3.3 Dynamic Model

OMT's dynamic model is represented by a state transition diagram that shows how an entity changes from one state to another. Each state receives one or more events, at which time it makes the transition to the next state. The next state depends on the current state as well as the events.

12.3.4 Dynamic Modelling Notation

The dynamic modelling notation is shown in Figure 12.2.

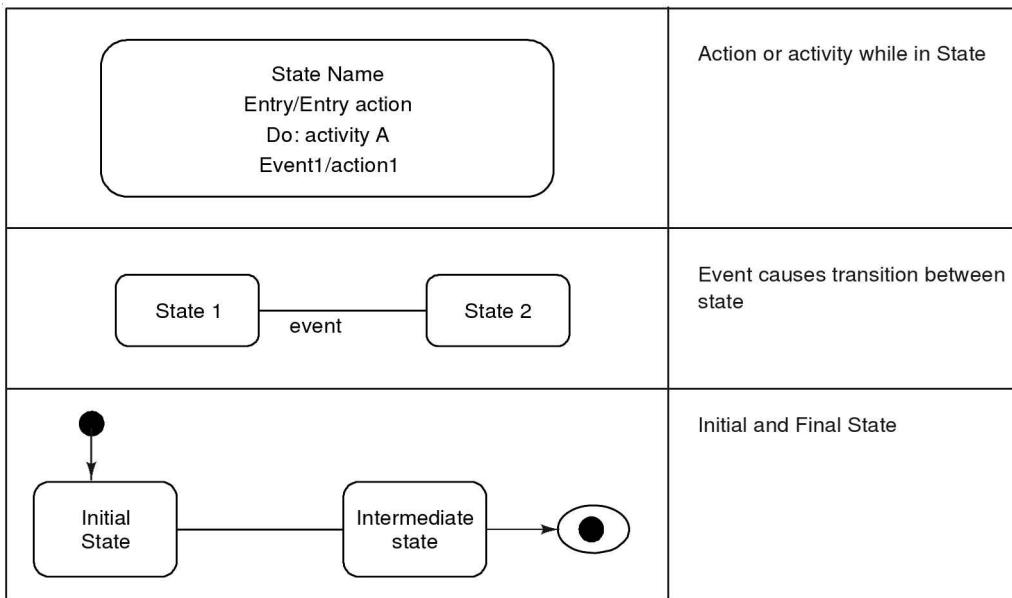


FIGURE 12.2 Dynamic modelling notation.

12.3.5 Functional Model

The functional model is represented by the data flow diagram (DFD) which shows the flow of data between different processes in a business.

12.3.6 Functional Model Notation

The functional model notation is shown in Figure 12.3.

12.4 BOOCH METHODOLOGY

The primary strength of the Booch methodology lies in object-oriented design of the system. The model of object-oriented development in Grady Booch's methodology is shown in Figure 12.4.

Capturing all the fine details of a complex software system in just one view is impossible. It is necessary to understand both the structural and the functional aspect of the objects involved. Booch's object system design method represents four models such as logical model, physical model, static model and dynamic model. Class hierarchies are defined in the logical model, whereas object methods are described in physical models. The logical view of a system describes the existence and meaning of the key abstractions and mechanisms that form the problem space. The physical model of a system describes the concrete software and hardware composition of the system's implementation.

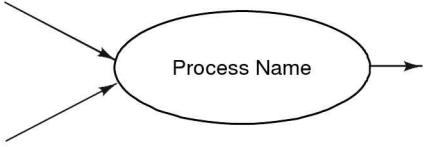
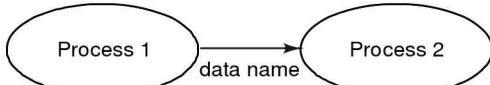
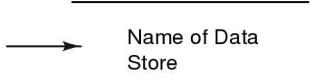
	Process
	Data Flow between Processes
	Data store or File Object
	Actor Object

FIGURE 12.3 Rumbaugh functional model notation.

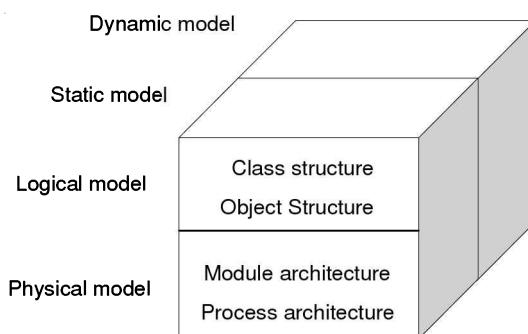


FIGURE 12.4 The models of object-oriented development.

Booch has introduced four static diagrams—class diagrams, object diagrams, module diagrams, process diagrams.

In addition, Booch defines the behavioural aspect of the system by dynamic model where he describes how an object may change state and objects interact with each other through two additional diagrams—state transition diagrams and interaction diagram. Each class may have an associated state transition diagram that indicates the event-ordered behaviour of the class's instance. Similarly, an interaction diagram shows the time or event-ordering of messages as they are evaluated.

12.4.1 Booch's Static Diagrams

Class diagram

A class diagram is used to show the classes and their relationships in the logical view of a system. During analysis, the class diagram indicates the common roles and responsibilities of the entities that provide the system's behaviour.

Classes

Figure 12.5 shows the icon to represent a class which is of cloud shape.

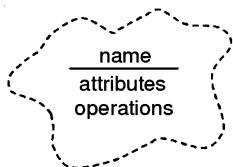


FIGURE 12.5 Class icon.

A name is required for each class. It is useful to expose some of the attributes and operations associated with a class in class diagrams. An attribute may have a name, a class, or both and optionally a default value as shown below:

A	Attributes only
: C	Attribute class only
A : C	Attribute name and class
A : C = E	Attribute name, class and default expression

An operation denotes some service provided by the class. Complete operation signature can be specified as shown below:

N()	Operation name only
R N(Args)	Operation return class, name and formal arguments (if any)

Abstract class

An abstract class is one for which no instances may be created. An abstract class is adorned with the letter "A" placed inside a triangle anywhere inside the class icon as shown in Figure 12.6.

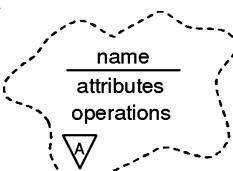


FIGURE 12.6 Abstract class.

Class relationships

Classes collaborate with other classes in a variety of ways. The relationships among classes include association, inheritance, “has”, and “using” relationships, whose icons are summarized as shown in Figure 12.7.

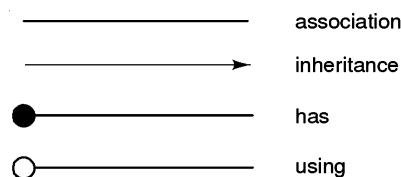


FIGURE 12.7 Class relationship icons.

Associations may be further adorned with their cardinality.

Class adornments

Adornments provide additional information about a class. Adornments notation is created by placing a letter inside the triangle within the class icon as shown in Figures 12.7(a)–(d).

A—Abstract. An abstract class cannot be instantiated.

F—Friend. A friend class allows access to the nonpublic functions of other classes.

S—Static. A static class provides data.

V—Virtual. A virtual class is a shared base class, the most generalized class in a system.

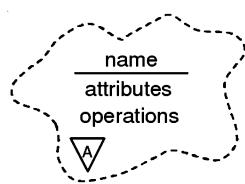


FIGURE 12.7(a) Abstract class.

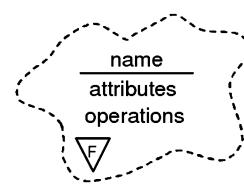


FIGURE 12.7(b) Friend class.

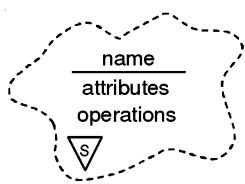


FIGURE 12.7(c) Static class.

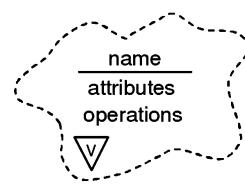


FIGURE 12.7(d) Virtual class.

Metaclasses

A metaclass is the class of a class. It may not itself have any instances, but may inherit from, contain instances of, use, and otherwise associate with other classes. A metaclass is shown in Figure 12.8.

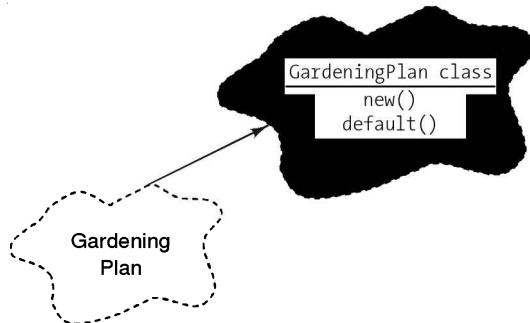


FIGURE 12.8 Metaclasses.

Class categories

A class category represents a group of similar classes. It represents an encapsulated namespace and is shown in Figure 12.9.

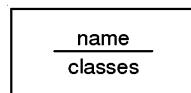


FIGURE 12.9 Class category icon.

Parameterized classes

A parameterized class denotes a family of classes whose structure and behaviour are defined independent of its formal class parameters. The instantiated relationship between a parameterized class and its instantiated class is shown as a dashed line, pointing to the parameterized class as shown in Figure 12.10.

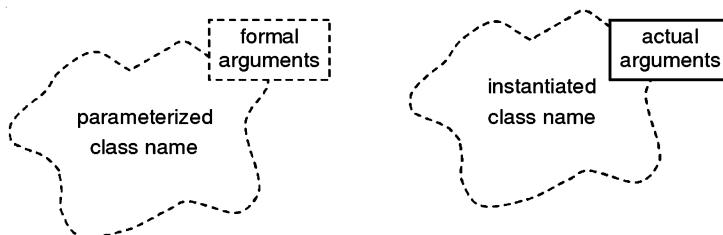


FIGURE 12.10 Parameterized classes.

Class utilities

A class utility denotes a collection of free subprograms or, in C++ non-member functions. The notation of the class utility is shown in Figure 12.11.

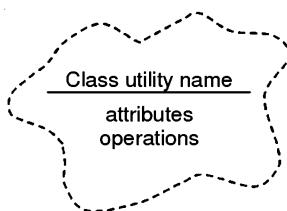


FIGURE 12.11 Class utilities.

Nesting

Classes may be physically nested in other classes, and categories may be nested in other categories as well, to any depth of nesting, typically to achieve some control over the namespace. As shown in Figure 12.12, nesting is indicated by physically nesting icons.

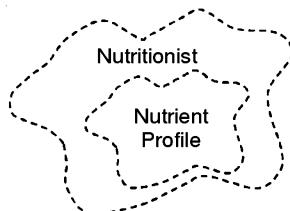


FIGURE 12.12 Nesting.

Export control

Export control signifies who can access the information contained within a class. The public attribute or an operation can be viewed by any other class. The private attribute or the operation is only accessible by the class itself and its friends. The protected attribute makes an attribute or operation visible only to friend classes and classes that inherit it. Access specifying symbols are shown in Figure 12.13.

<no adornment>	public
	protected
	private
	implementation

FIGURE 12.13 Export control.

Object diagram

An object diagram is used to show the existence of objects and their relationships in the logical design of a system. The two essential elements of an object diagram are objects and their relationships.

Objects: Object is an instance of a class represented as shown in Figure 12.14.

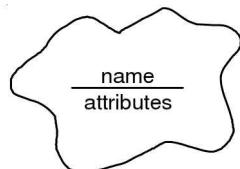


FIGURE 12.14 Object icon.

The name of an object follows the syntax for attributes, and may be written in any of the three following forms:

- A Object name only
- : C Object class only
- A : C Object name and class

Object relationship

Objects interact through their links to other objects represented by the icon shown in Figure 12.15.

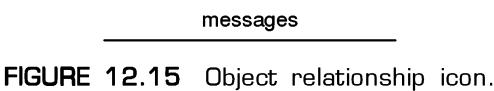


FIGURE 12.15 Object relationship icon.

A link can be adorned with a collection of messages. Each message consists of the following three elements:

- D A synchronization symbol denoting the direction of the invocation.
- M An operation invocation or event dispatch.
- S Optionally, a sequence number.

In the absence of an explicit sequence number, messages may be passed at any time relative to all other messages represented in a particular object diagram.

Module diagram

A module diagram is used to show the allocation of classes and objects to modules in the physical design of a system. It is used to indicate the physical layering and partitioning of the architecture. The two essential elements of a module diagram are modules and their dependencies.

Modules

Each module encompasses the declaration or definition of classes, objects, and other language details. Each module has a name and this name typically denotes the simple name of the corresponding physical file in the development directory.

The main program icon denotes a file that contains the root of a program and is shown as in Figure 12.15(a). The specification and the body icon denote files that contain the declaration and definition of entities respectively and are shown as in Figures 12.15(b) and (c).

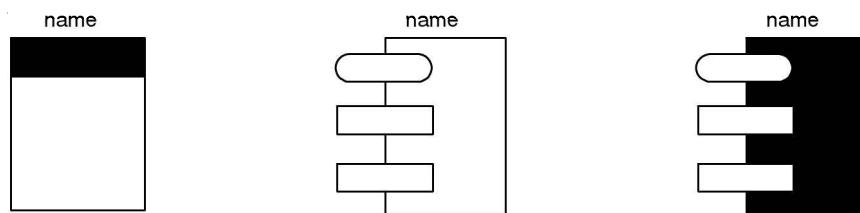


FIGURE 12.15(a) Main program. **FIGURE 12.15(b)** Specification. **FIGURE 12.15(c)** Body.

Dependencies

The only relationship we may have between two modules is a compilation dependency, represented by a directed line pointing to the module upon which the dependency exists as shown in Figure 12.16.



FIGURE 12.16 Dependency.

Subsystems

Subsystems partition the physical model of a system. A subsystem is an aggregate containing other modules and other subsystems. A subsystem can have dependencies upon other subsystems or modules and a module can have dependencies upon a subsystem. It is represented by the icon as shown in Figure 12.17.



FIGURE 12.17 Subsystem icon.

Process diagram

A process diagram is used to show the allocation of processes to processors in the physical design of a system. Process diagrams are used to indicate the physical collection of processors and devices that serve as the platform for execution of the system. The three essential elements of a process diagram are processors, devices and their connections.

Processors

A processor is a piece of hardware capable of executing programs. The processor icon can be adorned with a list of processes. Figure 12.18 shows the icon used to represent a processor.

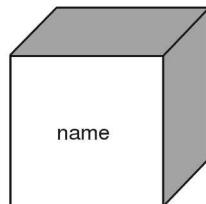


FIGURE 12.18 Processor icon.

Devices

A device is a piece of hardware incapable of executing programs. There are no particular constraints upon device names, and their names may be quite generic, such as modem or terminal. Figure 12.19 shows the icon used to represent a device.

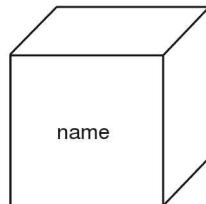


FIGURE 12.19 Device icon.

Connections

Processors and devices must communicate with one another using an undirected or bidirectional line indicating connection.

12.4.2 Booch's Dynamic Diagrams

State transition and interaction diagrams are Booch's dynamic diagrams which illustrate the dynamic nature of an application.

State transition diagram

A state transition diagram is used to show the state space of a given class, the events that cause a transition from one state to another and the actions that result from a state change. The two essential elements of a state transition diagram are the states and state transitions.

States

The state of an object represents the cumulative results of its behaviour. The state name must be unique to its enclosing scope. For a certain state, it is useful to expose the actions associated with a state. The state is represented by the icons shown in Figure 12.20.

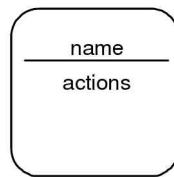


FIGURE 12.20 State icon.

State transitions

An event may cause the state of a system to change which is called state transition drawn using the arrow as shown in Figure 12.21. A state may have a state transition to itself.

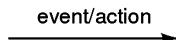


FIGURE 12.21 State transition icon.

Interaction diagram

An interaction diagram is used to trace the execution of a scenario in the same context as an object diagram. Interaction diagrams introduce no new concepts or icons; rather they take most of the essential elements of object diagrams and restructure them. The interaction diagrams focus on events more than on operations and result better at capturing the semantic of a scenario than are object diagrams. Participating objects are written horizontally at the top of the diagram and a dashed vertical line is drawn below each object. Messages are shown horizontally with arrows from the client to the server object dashed line using the same syntax and synchronization symbols used in the corresponding object diagram. The vertical box containing each object line represents the relative time that the flow of control is focused in that object.

12.4.3 Booch Methodology Process

The Booch methodology prescribes a macro development process and a micro-development process.

The micro-development process

The micro-process represents the daily activities of an individual developer or a small team of developers. The micro-process applies equally to a software engineer and a software architect.

The micro-development process consists of the following activities:

1. Identify the classes and objects at a given level of abstraction.
2. Identify the semantics of these classes and objects.
3. Identify the relationships among these classes and objects.
4. Specify the interface and then the implementation of these classes and objects.

The macro-development process

The macro-process serves as the controlling framework for the micro-process and can take weeks or even months. The macro-process is primarily the concern of the development team's technical management, whose focus is subtly different than that of a individual developer.

The macro-process consists of the following activities:

1. Establish the core requirements for the software (conceptualization).
2. Develop a model of the system's desired behaviour (analysis).
3. Create an architecture for the implementation (design)
4. Manage post delivery evolution (maintenance).

12.5 THE COAD-YOURDON METHODOLOGY

The primary strength of the Coad–Yourdon methodology lies in object-oriented analysis. The Coad–Yourdon analysis is based on a technique called “SOSAS” which has five step. The first step, *Subjects*, represents data flow diagrams for objects. The second step, *Objects*, identifies the object classes and the class hierarchies. The third step, *Structures*, decomposes structures into two types, i.e., classification structures and composition structures. The classification structure deals with the inheritance relationships between classes, while composition structure deals with the other relationships among classes. The fourth step, *Attributes*, and the final step *Services* identify all of the behaviours or methods for each class are identified.

Coad and Yourdon’s approach to object-oriented analysis is based on eight principles for managing complexity. Those principles are abstraction, encapsulation, inheritance, association, communication through messages, organizational methods (subdivided into objects and attributes, whole and parts), scale, and categories of behaviour.

The four components of system design by Coad–Yourdon methodology are as follows:

1. Problem domain component—Classes that should be in the problem domain.
2. Human interaction component—Interface classes between objects.
3. Task management component—System-wide management classes.
4. Data management component—Classes needed for database access.

12.5.1 Coad–Yourdon Diagrams

To draw the Coad–Yourdon diagrams, complete the following five steps:

- Step 1:* Find classes and objects
Step 2: Identify the structures

Step 3: Define subjects

Step 4: Define attributes

Step 5: Define services

Class and Object: Objects and classes are abstractions of entities with special services and attributes (Figure 12.22).

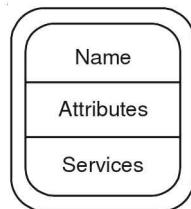


FIGURE 12.22 Class/Object icon.

Whole-part relationships

The whole-part relationship (Figure 12.23) refers to objects that contain one or more other objects. There are several types of whole-part relationships including: assembly-parts (car-wheels), container-contents (directory-files) and collection-members (organization-members).

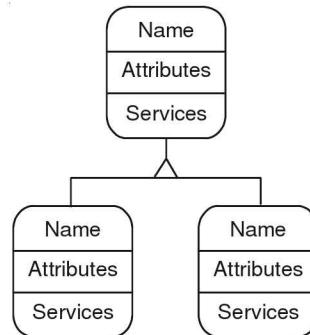


FIGURE 12.23 Whole-part relationship.

Generalization–specialization (gen-spec) relationships

Generalization–specialization relationships refer to classes that inherit attributes and services from other classes. One class can inherit from multiple superclasses (Figure 12.24).

Connections

Connections represents the dependency of one object on the services or processing of another object (Figure 12.25).

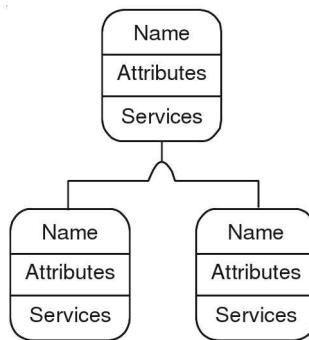


FIGURE 12.24 Generalization–specialization.

Instance connection

Message connection

FIGURE 12.25 Connections.

12.6 JACOBSON'S OBJECT-ORIENTED SOFTWARE ENGINEERING

Ivar Jacobson's methodology covers the entire life cycle of the system. Jacobson's methodology is based on the use-case concept. Object-oriented software engineering (OOSE) includes a requirement, an analysis, a design, an implementation, and a testing model. The relationship between the central use-case diagram and the various models are explained in Figure 12.26.

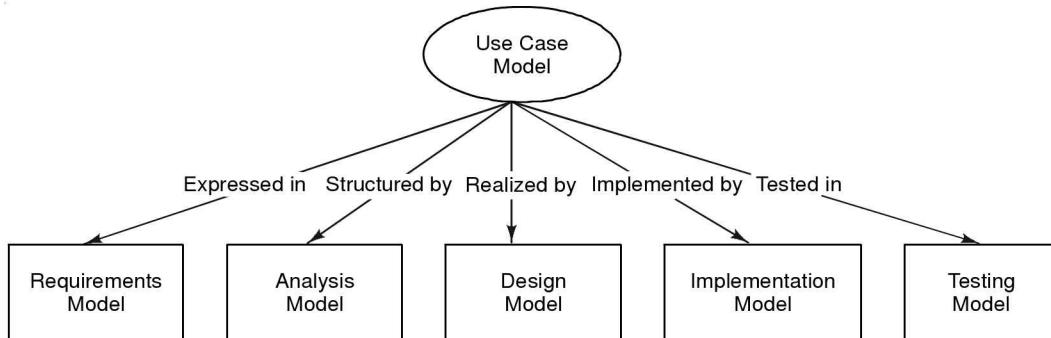


FIGURE 12.26 Jacobson's OOSE models.

Object-Oriented software engineering is also called as “Objectory”. This use-case driven development stresses that use cases are involved in several phases of the development, including analysis, design, validation, and testing. OOSE is based on several different models:

12.6.1 Requirements model

The requirements model includes a problem domain object diagram and use-case diagrams (Figure 12.27). It defines the limits and functionality of a system. The use-case diagrams explain how the outside world interacts with elements of the application system.

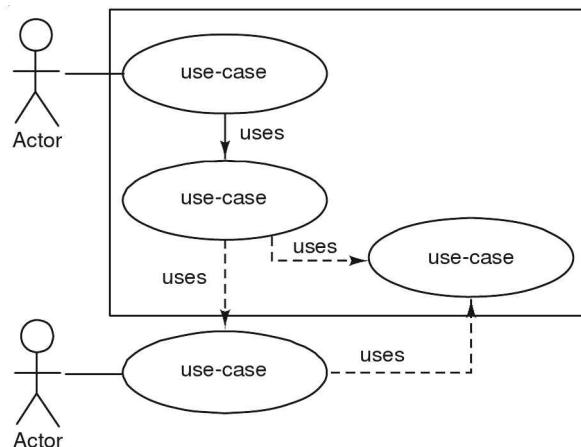


FIGURE 12.27 Use-case diagram.

12.6.2 Analysis Model

Jacobson's analysis model defines three types of objects in a system: boundary objects, entity objects and control objects (Figure 12.28).

Entity object: An entity object [Figure 12.28(a)] is used to model information and associated behaviour that must be stored.

Control object: Control objects [Figure 12.28(b)] represent the use-case logic and coordinates the other classes. It separates the interface classes from business logic classes.

Boundary object: Boundary objects [Figure 12.28(c)] are used to model the interaction between a system and its actors.

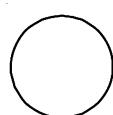


FIGURE 12.28(a) Entity object.

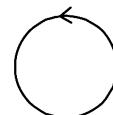


FIGURE 12.28(b) Control object.

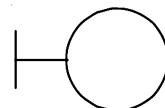


FIGURE 12.28(c) Boundary object.

12.6.3 Design Model

Jacobson's design model shows how the system behaves. There are two types of diagrams under this model: interaction diagrams and state transition diagrams.

12.6.4 Implementation Model

The implementation model represents the implementation of the system.

12.6.5 Testing Model

The test model constitutes the test plans, specifications and reports.

12.7 COMPARISON OF VARIOUS OBJECT-ORIENTED METHODOLOGIES

Many object-oriented methodologies are available to choose from for system development. Each methodology is based on modelling the business problem and implementing the application in an object-oriented fashion. The difference is primarily in the documentation of information and modelling notations. An application can be implemented in many ways to meet the same requirements and provide the same functionality. Two people using the same methodology may produce application designs that look completely different. This does not necessarily mean that one is right and one is wrong, just that they are different. Each method has its strengths and weaknesses. The Rumbaugh method is well suited for describing the object model or the static structure of the system. While Rumbaugh's OMT strongly models the problem domain, OMT models cannot fully express the requirements. Jacobson's methodology has a strong method for producing user-driven requirement and object-oriented analysis models. Jacobson's methodology is more user centered, in that everything in their approach derives from use-cases or usage scenarios. The Booch methodology has a strong method for producing detailed object-oriented design models. The Coad–Yourdon methodology mainly focusses on object-oriented analysis.

EXERCISES

1. Explain Grady Booch's methodology of object-oriented design.
2. Which are the diagrams used in the Booch methodology?
3. Explain Rumbaugh's methodology of object-oriented analysis.
4. Explain Booch's static diagrams with notations.
5. Explain Rumbaugh's three models—Static model, Dynamic model and Functional model.

6. Explain the technique on which Coad—Yourdan's methodology is based.
7. Explain Jacobson's requirement model for a system.
8. Explain Booch's dynamic diagrams with notations.
9. Compare object-oriented methodologies of Rumbaugh, Grady Booch, Coad—Yourdan and Ivar Jacobson.
10. Explain Booch's physical diagrams with notations.

Index

- <<application>>, 99
- <<component>>, 98
- <<database>>, 99
- <<document>>, 99
- <<executable>>, 99
- <<extend>>, 120
- <<extend>> use-cases, 75
- <<file>>, 99
- <<include>>, 120
- <<infrastructure>>, 99
- <<interface>>, 100
- <<library>>, 99
- <<source code>>, 99
- <<table>>, 99
- <<use>>, 102
- <<web service>>, 99
- <<XML DTD>>, 99
- 2-tier architecture, 110
- 3-tier architecture, 108, 110

- Abstract class, 269, 270
- Abstraction, 20
- Activation, 69
- Activation boxes, 69
- Activity, 26
 - diagrams, 23, 82, 89
 - states, 83
- Activity diagram, guidelines for the design of, 86
- Actor object, 72, 73, 76, 78, 120
- Actor/use case pair, 74
- Actors, 39, 42, 116
- Aggregation, 20, 53
- Alternative(s), 70
 - combination fragment, 70

- Analysis model, 15
- Analysis phase diagrams, 112
- Annotation, 31
- Application
 - components, 104
 - server machine, 108
 - tier, 110
- Architecture model, 15
- Artifacts, 26, 30, 106, 114
- Association, 19, 28, 50
 - class, 55, 192
 - end, 51
- AsynchCall, 68
- Asynchronous call, 68
- Attribute, 19, 49

- Ball and socket, 101
- Bidirectional association, 50
- Bidirectional communication, 103
- Booch methodology, 267
- Booch methodology process, 276
- Booch's dynamic diagrams, 275
- Boolean test, 72
- Bottom-up, 103
 - approach, 104
- Boundary class, 73
- Boundary object, 72, 73, 74, 76, 78, 121, 280
- BPMI, 24
- BPMN, 22, 24
- Bull's-eye, 83, 90
- Business
 - design, 25
 - flow, 86
 - modelling, 82

- process, 82, 114
- process diagram, 24, 25, 31, 113
- redesign, 25
- reengineering, 25
- rules, 25

- Class(es), 18, 49, 95
 - adornments, 270
 - category, 271
 - diagram, 23, 48, 75, 118, 119
 - utility, 272
- Classification, 19
- Clean-room technique, 6
- Client application, 111
- Client tier, 110
- Coad–Yourdon methodology, 277
- Code-behind classes, 73
- Cohesion, 22
- Collaborating objects, 77
- Collaboration diagram, 23, 77, 78, 79
- Collaboration diagram, guidelines for the design of, 79
- Combined fragment, 70, 122
- Comments, 114
- Communication, 78
 - diagram, 77
- Complex processes, 31
- Component(s), 22, 89, 98, 100, 102, 105, 106
 - diagram, 23, 98
 - guidelines for the design of, 103
 - models, 15
- Composite state, 91
- Composition, 20, 54
- Computational elements, 105
- Concurrent, 93
 - composite state, 91, 92
 - statechart diagrams, 92
 - threads, 77
- Conditional flow, 70, 87
- Connecting objects, 26, 28
- Continuous loop, 94
 - statechart diagram, 95
- Control object, 280
- Controller object, 72, 73, 74, 76, 78, 121
- Coupling, 21
- Creation state, 90
- Current state, 89

- Data
 - flow diagrams (DFDs), 82
 - flow modelling, 9
 - modelling, 10
 - objects, 30, 114
- Database
 - design, 10
 - layer, 110
 - server, 111
 - server machine, 108
- Decision box, 84
- Dependency(ies), 54, 102, 105, 107
- Deployment
 - components, 100
 - diagram, 23, 105, 108
 - guidelines for the design of, 107
- Design, 18
 - phase, 16
 - phase diagrams, 120
- Device interface class, 73
- Distributed systems, 105
- Do, 91
- Dynamic, 12
 - behaviour, 77
 - diagram, 22, 82
 - model, 266
 - modelling diagram, 67

- Elementary process, 31
- Encapsulation, 20
- Entity behaviour modelling, 9
- Entity object, 72, 74, 76, 78, 121, 280
- Event, 26, 82, 90, 94, 114
 - sequences, 67
- Exclusive gateway, 114
- Execution components, 100
- Export control, 272
- Extends, 41

- Feasibility study, 9
- Final state, 83, 87, 89, 90, 95, 96
- Flexibility, 25
- Flow
 - charts, 82
 - objects, 26
- Focus of control, 69

-
- Forks, 84, 85, 86
 - Frame, 70
 - Friend class, 270
 - Function/data methodology, 10
 - Functional
 - components, 104
 - decomposition, 10
 - model, 267
 - requirements, 36
 - view, 12
 - Gateway(s), 27, 114
 - Generalization, 19, 20, 42, 52
 - Group, 31, 114
 - Guard, 70, 83, 94
 - condition, 71, 83, 84, 86, 94
 - Horizontal dimension, 67
 - HTTP, 108
 - Implementation phase, 16
 - diagrams, 144
 - Incoming transition, 84
 - Incremental model, 3
 - Infrastructure components, 104
 - Inheritance, 21, 52
 - Initial state, 82, 86, 89, 90, 95, 96
 - Instance diagrams, 62
 - Integrity, 25
 - Interaction
 - diagram, 67, 77, 276
 - operands, 70
 - Interface(s), 22, 56, 74, 98, 99, 100, 102
 - classes, 104
 - Intermediate states, 95, 96
 - Internal states, 91
 - Jacobson's object-oriented software engineering, 279
 - Join(s), 84, 85, 86, 87
 - Lane, 30
 - Lifelines, 68
 - Links, 19, 63, 78, 107
 - Logical
 - data modelling, 9
 - system specification, 9
 - Lollipop notation, 100
 - Loop(s), 70, 72
 - combination fragment, 72
 - Macro-development process, 277
 - Maximum iterations, 72
 - Message, 67, 68, 78
 - communication, 76
 - flow, 28
 - Metaclass, 271
 - Method call, 68, 76, 121
 - Method implementation, 100
 - Micro-development process, 276
 - Minimum iterations, 72
 - Modelling data, 103
 - Modularity, 25
 - Module(s), 98, 274
 - diagram, 268, 273
 - Multiple associations, 52
 - Multiplicity, 55, 78
 - Nested states, 91, 92
 - Nesting, 272
 - Nodes, 105, 106
 - Object(s), 18, 62, 78
 - diagram, 23, 62, 119
 - flow, 86, 87, 88
 - message trace, 79
 - model, 265
 - modelling group, 264
 - oriented, 1
 - analysis, 16, 17
 - analysis and design, 15
 - analysis and design methodologies, 264
 - approach, 13
 - design, 16, 18
 - methodology, 12
 - modelling, 15
 - technique, 6

- Objectory, 279
- One shot life cycle, 94
 - statechart diagram, 95
- OOAD methodologies, 264
- OOSE, 279
- Operations, 19, 49, 100
- Option combination fragment, 70
- Options, 70
- Organizational unit, 85
- Outgoing transitions, 84

- Package, 58
- Parallel activities, 84
- Parallel flow, 87
- Parameterized classes, 271
- Physical design, 9
- Polymorphism, 21
- Pool(s), 29
 - and lanes, 113
- Ports, 102, 103
- Primary actors, 74
- Proactive system, 75
- Process
 - diagrams, 268, 274
 - modelling, 10, 25
 - models, 25
- Program specifications, 10
- Prototype, 1
- Prototyping, 5
- Provided interface, 101, 102
- Pseudo state, 90

- Reactive system actors, 74
- Receiver, 78
- Recursive transitions, 93, 95
- Reflexive association, 51
- Required interface, 101, 102
- Requirement analysis, 9
- Requirement specification, 9
- Rumbaugh's object modelling technique method, 265

- Security components, 104
- Self-transitions, 94
- Sender, 78

- Sequence
 - diagram, 23, 67, 75, 77, 78, 79, 120
 - guidelines for the design of, 74
 - flow, 28, 82
- Sequenced messages, 77
- Sequential composite state, 91, 92
- Simple state, 91
- Software
 - components, 98
 - development methodologies, 1
 - development models, 1, 2
 - development techniques, 5
 - prototype, 103
- SOSAS, 277
- Specialization, 19, 20
- Spiral model, 4
- SRS, 10
- State(s), 89, 90, 94, 95, 276
 - chart diagram, 89
 - guidelines for the design of, 94
 - diagram, 89
 - machine diagram, 89
 - transition, 93, 276
 - transition diagram, 89, 266
- Statechart diagrams, 23
- Static
 - class, 270
 - diagrams, 22
 - structure, 77
 - view, 12
- Stereotype, 56, 99
- Structural relationships, 98
- Structure chart, 10
- Structured
 - development, 82
 - programming, 10
 - systems analysis and design, 8
- Sub-partition, 30
- Sub-process, 26
- Substates, 91, 93
- Subsystem, 103, 274
- Super state, 91
- Swimlanes, 26, 28, 85, 86, 87
- SynchCall, 68
- Synchronization, 85
 - element, 84
- Synchronous call, 68

- System
 - analysis, 1
 - analysis and design, 1
 - boundary box, 43
 - design, 1
 - interface class, 73
- TCP/IP protocol, 108, 111
- Technical classes, 104
- Thick client, 111
- Thin client, 111
- Top-down, 67, 103
- Top-down approach, 103
- Transition, 83, 89, 90, 91, 94, 95
- Trigger event, 94
- Triggerless transitions, 94, 96
- Unified modelling language, 22
- Unnamed instance, 68
- Unspecified class, 68
- Usage
 - dependency, 102
 - scenario, 67, 82
- Use-case(s), 36, 39, 42, 73, 82, 90, 116
 - diagrams, 23, 38, 73, 75, 116
 - realization, 67, 73
- User interface, 103
 - class, 73
- Vertical dimension, 67
- Virtual class, 270
- Waterfall model, 2
- Web server, 108
- Web-based application, 108
- Whole-part relationships, 278
- Work product components, 100
- Workflow, 82