

## ICT340

# Application Analysis and Design

School of Science and Technology

Study Guide



# **Course Development Team**

<b>Head of Programme</b>	: Assoc Prof Zhu Yongqing
<b>Course Developer(s)</b>	: Dr Pamela Siow
<b>Technical Writer</b>	: Maybel Heng, EMR
<b>Video Production</b>	: Claudia Lim, EMR

© 2023 Singapore University of Social Sciences. All rights reserved.

No part of this material may be reproduced in any form or by any means without permission in writing from the Educational Media & Resources, Singapore University of Social Sciences.

ISBN 978-981-5057-03-4

## **Educational Media & Resources**

*Singapore University of Social Sciences  
463 Clementi Road  
Singapore 599494*

## **How to cite this Study Guide (APA):**

Siow, P. (2023). *ICT340 Application analysis and design*. Singapore University of Social Sciences.

Release V1.1.0

Build S1.0.15, T1.7.2



# Table of Contents

## Course Guide

1. Welcome.....	CG-2
2. Course Description and Aims.....	CG-3
3. Learning Outcomes.....	CG-5
4. Learning Material.....	CG-6
5. Assessment Overview.....	CG-7
6. Course Schedule.....	CG-9
7. Learning Mode.....	CG-10

## Study Unit 1: Introduction to Application Analysis and Design

Learning Outcomes.....	SU1-2
Overview.....	SU1-3
Chapter 1: Software Processes.....	SU1-4
Chapter 2: Requirements Engineering.....	SU1-28
Chapter 3: System Modelling.....	SU1-47
Chapter 4: Requirements Specification Document.....	SU1-71
Summary.....	SU1-73
Formative Assessment.....	SU1-75
References.....	SU1-87

## **Study Unit 2: Structural Object-Oriented Modelling**

Learning Outcomes.....	SU2-2
Overview.....	SU2-3
Chapter 1: Analysing Requirements for Classes.....	SU2-4
Chapter 2: Case Studies.....	SU2-45
Summary.....	SU2-59
Formative Assessment.....	SU2-60
References.....	SU2-70

## **Study Unit 3: Dynamic Object-Oriented Modelling I: Analysing and Constructing Models with Walkthroughs and Sequence Diagrams**

Learning Outcomes.....	SU3-2
Overview.....	SU3-3
Chapter 1: Model Driven Architecture.....	SU3-4
Chapter 2: Sequence Diagrams.....	SU3-7
Chapter 3: Walkthroughs.....	SU3-20
Chapter 4: Visualising Walkthroughs.....	SU3-48
Summary.....	SU3-62
Formative Assessment.....	SU3-63
References.....	SU3-89

## **Study Unit 4: Dynamic Object-Oriented Modelling II**

Learning Outcomes.....	SU4-2
Overview.....	SU4-3
Chapter 1: More Analysis with Sequence Diagrams.....	SU4-4
Chapter 2: State Diagram.....	SU4-16
Chapter 3: Implementation of State Machine Diagram.....	SU4-35
Summary.....	SU4-43
Formative Assessment.....	SU4-44
References.....	SU4-65

## **Study Unit 5: Implementations, Integration and Testing**

Learning Outcomes.....	SU5-2
Overview.....	SU5-3
Chapter 1: Implementation.....	SU5-5
Chapter 2: Integration.....	SU5-31
Chapter 3: Testing.....	SU5-42
Summary.....	SU5-52
Formative Assessment.....	SU5-54
References.....	SU5-74

## **Study Unit 6: Design Patterns**

Learning Outcomes.....	SU6-2
Overview.....	SU6-3

Chapter 1: Introduction to Design Patterns.....	SU6-4
Chapter 2: Behavioural Design Patterns.....	SU6-7
Chapter 3: Structural Design Patterns.....	SU6-45
Chapter 4: Creational Design Patterns.....	SU6-58
Summary.....	SU6-75
Formative Assessment.....	SU6-76
References.....	SU6-91

## List of Tables

<b>Table 1.1</b> Standard Template/Form.....	SU1-72
<b>Table 2.1</b> Requirements and Classes Inferred.....	SU2-9
<b>Table 2.2</b> Analysing Requirements for Attributes.....	SU2-14
<b>Table 2.3</b> Analysing Requirements for Multiplicities.....	SU2-36
<b>Table 2.4</b> Multiplicities among the Classes.....	SU2-38
<b>Table 2.5</b> Multiplicities for the Classes in the LUS.....	SU2-49
<b>Table 4.1</b> Elements of a State Machine.....	SU4-18
<b>Table 4.2</b> State Changes for Events in Book class.....	SU4-35
<b>Table 4.3</b> State Changes for Events in Bank Account class.....	SU4-38



## **List of Figures**

<b>Figure 1.1</b> System Development Life Cycle.....	SU1-11
<b>Figure 1.2</b> A Banking System integrated with other systems.....	SU1-13
<b>Figure 1.3</b> Waterfall Approach.....	SU1-17
<b>Figure 1.4</b> Incremental Development.....	SU1-19
<b>Figure 1.5</b> Reuse-Oriented Development Approach.....	SU1-20
<b>Figure 1.6</b> Plan-driven and Agile Development.....	SU1-21
<b>Figure 1.7</b> The Process Improvement Cycle.....	SU1-25
<b>Figure 1.8</b> User and System Requirements.....	SU1-29
<b>Figure 1.9</b> Examples of Non-Functional Requirements for the Mentcare System.....	SU1-32
<b>Figure 1.10</b> Types of Non-Functional Requirements.....	SU1-32
<b>Figure 1.11</b> The Requirements Elicitation and Analysis Process.....	SU1-35
<b>Figure 1.12</b> Blood Pressure Monitor.....	SU1-44
<b>Figure 1.13</b> Requirements Evolution.....	SU1-46
<b>Figure 1.14</b> Requirements Change Management.....	SU1-46
<b>Figure 1.15</b> System Context for the Weather Station.....	SU1-49
<b>Figure 1.16</b> High-level architecture of Weather Station.....	SU1-50
<b>Figure 1.17</b> Use Cases for the Library Application.....	SU1-51
<b>Figure 1.18</b> Library System with Use Cases for the Library Application.....	SU1-53

<b>Figure 1.19</b> Examples of Actors.....	SU1-55
<b>Figure 1.20</b> Library System with Use Cases and Actors.....	SU1-55
<b>Figure 1.21</b> Library System with Generalisation Relationships.....	SU1-57
<b>Figure 1.22</b> Dependency include Relationships.....	SU1-58
<b>Figure 1.23</b> Dependency extend Relationships.....	SU1-58
<b>Figure 1.24</b> Activity Diagram for Withdraw Money.....	SU1-69
<b>Figure 1.25</b> Activity Diagram showing Correspondence to Use Case Description.....	SU1-70
<b>Figure 2.1</b> The primary objectives of a class in an application.....	SU2-6
<b>Figure 2.2</b> A class and a property in different applications.....	SU2-18
<b>Figure 2.3</b> The generalisation relationship.....	SU2-19
<b>Figure 2.4</b> The generalisation relationship with an abstract superclass.....	SU2-20
<b>Figure 2.5</b> An association between two classes.....	SU2-22
<b>Figure 2.6</b> Two ways to show an association.....	SU2-23
<b>Figure 2.7</b> Showing the correct associations.....	SU2-24
<b>Figure 2.8</b> An association between two classes.....	SU2-24
<b>Figure 2.9</b> The object diagram shows the actual association.....	SU2-25
<b>Figure 2.10</b> The class association diagram (so far).....	SU2-27
<b>Figure 2.11</b> Customer is associated with Car through a Contract.....	SU2-27
<b>Figure 2.12</b> Customer is associated with Car.....	SU2-28
<b>Figure 2.13</b> A derived association.....	SU2-28

<b>Figure 2.14</b> An example of a derived association.....	SU2-29
<b>Figure 2.15</b> Another example of a derived association.....	SU2-29
<b>Figure 2.16</b> A misleading derived association.....	SU2-30
<b>Figure 2.17</b> Showing multiplicities in a class diagram.....	SU2-30
<b>Figure 2.18</b> An object diagram showing the one to one association.....	SU2-31
<b>Figure 2.19</b> An association with its multiplicities.....	SU2-32
<b>Figure 2.20</b> One to one multiplicity.....	SU2-32
<b>Figure 2.21</b> Many to one multiplicity.....	SU2-33
<b>Figure 2.22</b> One to many multiplicity.....	SU2-33
<b>Figure 2.23</b> An example of one to many multiplicity.....	SU2-33
<b>Figure 2.24</b> Another example of one to many multiplicity.....	SU2-34
<b>Figure 2.25</b> Many to many multiplicity.....	SU2-34
<b>Figure 2.26</b> An object diagram with many to one association.....	SU2-35
<b>Figure 2.27</b> A recursive association.....	SU2-35
<b>Figure 2.28</b> An association with role names.....	SU2-36
<b>Figure 2.29</b> The class association diagram with multiplicities.....	SU2-38
<b>Figure 2.30</b> Considering constraints in the class association diagram.....	SU2-39
<b>Figure 2.31</b> Object diagram for the class association diagram in Figure 2.30.....	SU2-39
<b>Figure 2.32</b> A class association diagram with a redundant association.....	SU2-41
<b>Figure 2.33</b> An aggregation relationship.....	SU2-43

<b>Figure 2.34</b> A composition relationship.....	SU2-43
<b>Figure 2.35</b> The class association diagram.....	SU2-46
<b>Figure 2.36</b> The class association diagram.....	SU2-50
<b>Figure 2.37</b> The class association diagram for the Mentcare System.....	SU2-53
<b>Figure 2.38</b> The updated class association diagram for the Mentcare System.....	SU2-54
<b>Figure 3.1</b> MDA transformations.....	SU3-5
<b>Figure 3.2</b> Multiple platform-specific models.....	SU3-6
<b>Figure 3.3</b> Use Case Diagram for a Library System.....	SU3-7
<b>Figure 3.4</b> Class Diagram for Purchasing System.....	SU3-8
<b>Figure 3.5</b> Types of messages.....	SU3-12
<b>Figure 3.6</b> The basic structure of a sequence diagram.....	SU3-13
<b>Figure 3.7</b> Checking password at ATM.....	SU3-16
<b>Figure 3.8</b> Ordering Multiple Items.....	SU3-17
<b>Figure 3.9</b> Doing a Transaction at an ATM.....	SU3-18
<b>Figure 3.10</b> Withdraw cash at an ATM.....	SU3-19
<b>Figure 3.11</b> The class association diagram for car rental application.....	SU3-21
<b>Figure 3.12</b> The role of the orchestrating class.....	SU3-23
<b>Figure 3.13</b> Adding the orchestrating class to prepare for the walkthroughs.....	SU3-24
<b>Figure 3.14</b> Adding the input to prepare for the walkthroughs.....	SU3-26

<b>Figure 3.15</b> Adding the orchestrating class to the class diagram.....	SU3-27
<b>Figure 3.16</b> Using a collection for the value of the instance variable.....	SU3-28
<b>Figure 3.17</b> An arrowhead is added to the association.....	SU3-29
<b>Figure 3.18</b> Direction of navigation between the classes.....	SU3-29
<b>Figure 3.19</b> A user wishes to find the manager in charge of a particular branch.....	SU3-31
<b>Figure 3.20</b> Where we are in our walkthrough.....	SU3-31
<b>Figure 3.21</b> The multiplicity shows there is only one manager for each branch.....	SU3-32
<b>Figure 3.22</b> The class Branch from the structural (static) analysis.....	SU3-33
<b>Figure 3.23</b> The class Branch modified as a result of our walkthrough.....	SU3-33
<b>Figure 3.24</b> Another navigation arrowhead is added.....	SU3-34
<b>Figure 3.25</b> A fragment of the class diagram showing the association being implemented.....	SU3-35
<b>Figure 3.26</b> How the application gets the particulars of a manager.....	SU3-36
<b>Figure 3.27</b> The method to be added to the orchestrating class.....	SU3-39
<b>Figure 3.28</b> The walkthrough to find the contract for a given car.....	SU3-39
<b>Figure 3.29</b> The method to be added to the orchestrating class.....	SU3-42
<b>Figure 3.30</b> The walkthrough to assign a new manager to a branch.....	SU3-42
<b>Figure 3.31</b> Assigning a new Manager object to a Branch object.....	SU3-43
<b>Figure 3.32</b> Illustrating a bi-directional association.....	SU3-45
<b>Figure 3.33</b> A bi-directional association has to be consistent.....	SU3-46

<b>Figure 3.34</b> An inconsistent bi-directional association.....	SU3-46
<b>Figure 3.35</b> The design for the orchestrating class so far.....	SU3-49
<b>Figure 3.36</b> The method in the orchestrating class will look for the Branch object.....	SU3-49
<b>Figure 3.37</b> An object sending a message to itself.....	SU3-50
<b>Figure 3.38</b> The design for the orchestrating class so far.....	SU3-50
<b>Figure 3.39</b> Sending a message to the Branch object that has been located.....	SU3-51
<b>Figure 3.40</b> The getter method for the instance variable manager.....	SU3-51
<b>Figure 3.41</b> A message for the Manager object in Step 3.....	SU3-52
<b>Figure 3.42</b> Messages in the sequence diagram.....	SU3-52
<b>Figure 3.43</b> Association between two classes.....	SU3-54
<b>Figure 3.44</b> Message passing between two objects.....	SU3-55
<b>Figure 3.45</b> Association between two classes.....	SU3-55
<b>Figure 3.46</b> Message passing between two objects.....	SU3-56
<b>Figure 3.47</b> The two variations of the class Admin.....	SU3-58
<b>Figure 3.48</b> The sequence diagram for Variation 1.....	SU3-59
<b>Figure 3.49</b> The sequence diagram for Variation 2.....	SU3-60
<b>Figure 3.50</b> The getter method for the instance variable name.....	SU3-60
<b>Figure 4.1</b> Assigning a new manager to a branch.....	SU4-5
<b>Figure 4.2</b> Assigning a new manager to a branch in sequence diagram.....	SU4-6
<b>Figure 4.3</b> A one-directional association between branch and manager.....	SU4-6

<b>Figure 4.4</b> Creating a one-directional association between branch and its manager.....	SU4-7
<b>Figure 4.5</b> Class association diagram to re-assigning a manager to another branch.....	SU4-9
<b>Figure 4.6</b> Part of class association diagram relevant to remove a car.....	SU4-10
<b>Figure 4.7</b> Class association diagram to remove a car from a contract.....	SU4-12
<b>Figure 4.8</b> Class association diagram to remove a car from a contract.....	SU4-13
<b>Figure 4.9</b> Sequence diagram to remove a car from a contract.....	SU4-14
<b>Figure 4.10</b> An object created from its class with its own state.....	SU4-16
<b>Figure 4.11</b> Object state change.....	SU4-17
<b>Figure 4.12</b> UML symbol showing the state of an object.....	SU4-19
<b>Figure 4.13</b> UML symbol showing a transition between two states.....	SU4-19
<b>Figure 4.14</b> UML symbol showing an event causing the transition.....	SU4-20
<b>Figure 4.15</b> UML symbol showing an event with condition and action.....	SU4-20
<b>Figure 4.16</b> UML initial and final pseudo-states.....	SU4-21
<b>Figure 4.17</b> State Machine Diagram of an ATM.....	SU4-22
<b>Figure 4.18</b> State Machine Diagram of a microwave oven.....	SU4-23
<b>Figure 4.19</b> Substates of the Operation state.....	SU4-24
<b>Figure 4.20</b> The same object with the same message can give different responses.....	SU4-26
<b>Figure 4.21</b> The actual state of an object has an impact on its response.....	SU4-26
<b>Figure 4.22</b> A state machine diagram with guards.....	SU4-27

<b>Figure 4.23</b> Internal activities of a State.....	SU4-27
<b>Figure 4.24</b> StateX can come about from several other states, but eventA has to take place for this to happen.....	SU4-29
<b>Figure 4.25</b> A state with an Entry event.....	SU4-30
<b>Figure 4.26</b> Internal activities of the "Menu Visible" State.....	SU4-30
<b>Figure 4.27</b> A state machine diagram shows the Messages that an object should understand.....	SU4-32
<b>Figure 4.28</b> A state machine diagram that models a stopwatch.....	SU4-33
<b>Figure 4.29</b> A Book class with methods obtained from the state diagram.....	SU4-36
<b>Figure 4.30</b> A BankAccount class with methods obtained from the state diagram.....	SU4-40
<b>Figure 5.1</b> The class association diagram for car rental application.....	SU5-6
<b>Figure 5.2</b> The class association diagram for car rental application.....	SU5-10
<b>Figure 5.3</b> A fragment of a class association diagram.....	SU5-10
<b>Figure 5.4</b> Implementing the association in the class Car.....	SU5-11
<b>Figure 5.5</b> Implementing the association in the class Contract.....	SU5-12
<b>Figure 5.6</b> Writing the code for the assignManager() method.....	SU5-17
<b>Figure 5.7</b> The association between Admin and Branch.....	SU5-18
<b>Figure 5.8</b> The association between Admin and Branch.....	SU5-21
<b>Figure 5.9</b> An object receiving a message.....	SU5-21
<b>Figure 5.10</b> The object that receives the message.....	SU5-22

<b>Figure 5.11</b> The Admin object sends messages when it executes the assignNewManager() method.....	SU5-23
<b>Figure 5.12</b> The receivers of the messages.....	SU5-23
<b>Figure 5.13</b> Setting up a bi-directional association.....	SU5-25
<b>Figure 5.14</b> The system and the external world.....	SU5-31
<b>Figure 5.15</b> Application with many types of interfaces.....	SU5-32
<b>Figure 5.16</b> An application with more than one interface.....	SU5-34
<b>Figure 5.17</b> Keeping the interfaces and the application separate by hiding implementation details from each other.....	SU5-35
<b>Figure 5.18</b> A Simple User Interface.....	SU5-38
<b>Figure 5.19</b> Output of Python program.....	SU5-40
<b>Figure 5.20</b> A model of the software testing process.....	SU5-43
<b>Figure 6.1</b> Design using inheritance.....	SU6-9
<b>Figure 6.2</b> Design with new behaviour.....	SU6-9
<b>Figure 6.3</b> Extracting the behaviours.....	SU6-10
<b>Figure 6.4</b> Design using interfaces.....	SU6-11
<b>Figure 6.5</b> The Strategy Pattern.....	SU6-12
<b>Figure 6.6</b> Two behaviours.....	SU6-13
<b>Figure 6.7</b> The Duck class.....	SU6-16
<b>Figure 6.8</b> State machine of the toy capsule dispenser.....	SU6-23
<b>Figure 6.9</b> State machine of the toy capsule dispenser with new requirement.....	SU6-24

<b>Figure 6.10</b> State machine of the toy capsule dispenser with new requirement.....	SU6-25
<b>Figure 6.11</b> State Pattern class association diagram.....	SU6-26
<b>Figure 6.12</b> The Observer Pattern.....	SU6-37
<b>Figure 6.13</b> Observer Pattern class association diagram.....	SU6-37
<b>Figure 6.14</b> Weather Monitoring application.....	SU6-39
<b>Figure 6.15</b> Applying Observer Pattern to Weather Monitoring application.....	SU6-40
<b>Figure 6.16</b> Adapter required.....	SU6-46
<b>Figure 6.17</b> Adapter Pattern class association diagram.....	SU6-47
<b>Figure 6.18</b> Decorator class association diagram.....	SU6-52
<b>Figure 6.19</b> Decorators wrapping a component.....	SU6-53
<b>Figure 6.20</b> Starbuzz beverage application using the decorator pattern.....	SU6-54
<b>Figure 6.21</b> Factory Pattern class association diagram.....	SU6-59
<b>Figure 6.22</b> Pizza Store application using factory pattern.....	SU6-61
<b>Figure 6.23</b> Chocolate boiler.....	SU6-69
<b>Figure 6.24</b> Chocolate boiler class.....	SU6-69
<b>Figure 6.25</b> Singleton class diagram.....	SU6-71

## **List of Lesson Recordings**

Use Case Modelling.....	SU1-4
Structural Analysis.....	SU2-4
Dynamic Analysis.....	SU3-4
Implementations.....	SU4-4
State Diagram.....	SU5-5
Design Patterns – Strategy Pattern.....	SU6-4



# Course Guide

**Application Analysis and Design**

## 1. Welcome



*Presenter: Dr Pamela Siow*



This streaming video requires Internet connection. Access it via Wi-Fi to avoid incurring data charges on your personal mobile plan.

Click [here](#) to watch the video.<sup>i</sup>

Click [here](#) for the transcript.

Welcome to the course ***ICT340 Application Analysis and Design***, a 5 credit unit (CU) course.

This StudyGuide will be your personal learning resource to take you through the course learning journey. The guide is divided into two main sections – the **Course Guide** and **Study Units**.

The **Course Guide** describes the structure for the entire course and provides you with an overview of the **Study Units**. It serves as a roadmap of the different learning components within the course. This **Course Guide** contains important information regarding the course learning outcomes, learning materials and resources, assessment breakdown and additional course information.

---

<sup>i</sup> <https://suss.ap.panopto.com/Panopto/Pages/Viewer.aspx?id=72845232-7ca5-4820-b5a0-ae070095a27f>

## 2. Course Description and Aims

ICT340 Application Analysis and Design provides students with the foundations for the analysis and design of application systems using the object-oriented paradigm. It introduces the key concepts necessary for subsequent practical object-oriented analysis and design in the real world that begins with a structural model and then proceeds with dynamic modeling. At the end of the course, students would be able to develop object oriented analysis design and implement the design in modeling and programming languages. Students use Python as the implementation language and UML as the modeling language. A case study will allow students to demonstrate what they learned in mainstream software application systems.

### Course Structure

This course is a 5-credit unit course.

There are six Study Units in this course. The following provides an overview of each Study Unit.

#### **Study Unit 1 – Introduction to Application Analysis and Design**

This unit introduces application analysis and design as an engineering activity as performed by an analyst. The use of a systematic and structured methodology to perform requirements analysis is emphasized. It also introduces the application development life cycle models along with reasons and situations for when to use them. Case studies are presented to explain various concepts.

#### **Study Unit 2 – Structural Object-Oriented Modelling**

This unit discusses how to develop a structural model based on a given set of requirements. The structural model defines the system components and their arrangements that make it possible for the system to meet the requirements and use cases.

Using an object-oriented approach, detailed coverage of use case models and class models with **UML (Unified Modeling Language)** is included.

## **Study Unit 3 – Dynamic Object-Oriented Modelling I**

The aim of this study unit is to develop dynamic models to determine the interactions among objects and achieve the required system behaviour. Using walkthroughs and UML sequence diagrams, the system is modeled as it is executing.

## **Study Unit 4 – Dynamic Object-Oriented Modelling II**

This unit continues from the previous unit to model the behaviour of a single object as it interacts with its environment. Using UML State diagrams, an object is designed for valid interactions.

## **Study Unit 5 – Implementation, Integration and Testing**

This unit utilizes the design outputs from the structural and dynamic models to realize the system by translating the design to code in Python. Integration of the user interface with the codes is also covered. Finally, testing is discussed with a review of the various testing required for a system.

## **Study Unit 6 – Design Patterns**

This unit introduces you to design patterns and takes you through several of the most used object-oriented patterns. An overview of patterns in various categories is given and examples of the featured pattern in action are discussed. A real-world context to these patterns is given with codes given in Python.

### 3. Learning Outcomes

#### Knowledge & Understanding (Theory Component)

By the end of this course, you will be able to:

- Analyse user requirements for non-ambiguity, correctness, completeness and consistency for system design
- Develop the structural and dynamic models based on system design
- Appraise the associations among a set of classes as part of a structural processing
- Demonstrate the application of design patterns in system design

#### Key Skills (Practical Component)

By the end of this course, you will be able to:

- Formulate report to document design analysis well
- Construct components of a system with a modelling language
- Implement structural and dynamic model in Python

## 4. Learning Material

To complete the course, you will need the following learning material(s):

### Required Textbook(s)

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.

Freeman, E., & Robson, E. (2020). *Head first design patterns* (2nd ed.). Sebastopol: O'Reilly Media, Incorporated.

If you are enrolled into this course, you will be able to access the eTextbooks here:



To launch eTextbook, you need a VitalSource account which can be created via Canvas (iBookStore), using your SUSS email address. Access to adopted eTextbook is restricted by enrolment to this course.

## 5. Assessment Overview

The overall assessment weighting for this course is as follows:

Assessment	Description	Weight Allocation
Assignment 1	Online Quiz	12%
Assignment 2	Tutor Marked Assignment	18%
Assignment 3	End of Course Assignment	70%
<b>TOTAL</b>		<b>100%</b>

The following section provides important information regarding Assessments.

### Assessment:

There will be continuous assessment in the form of online quizzes, one tutor-marked assignment (TMA) and one end-of-course assignment (ECA). In total, this continuous assessment will constitute 30 percent of overall student assessment for this course, and ECA 70 percent. The three assignments are compulsory and are non-substitutable. These assignments will test conceptual understanding of both the fundamental and more advanced concepts and applications that underlie analysis and design of applications. It is imperative that you read through your Assignment questions and submission instructions before embarking on your Assignment.

### Passing Mark:

To successfully pass the course, you must obtain a minimum passing mark of 40 percent for each of the TMA and ECA components. That is, students must obtain at least a mark of 40 percent for the combined assessments. For detailed information on the Course grading policy, please refer to The Student Handbook ('Award of Grades' section under

Assessment and Examination Regulations). The Student Handbook is available from the Student Portal.

### **Non-graded Learning Activities:**

Activities for the purpose of self-learning are present in each study unit. These learning activities are meant to enable you to assess your understanding and achievement of the learning outcomes. The type of activities can be in the form of Formative Assessment, Quiz, Review Questions, Application-Based Questions or similar. You are expected to complete the suggested activities either independently and/or in groups.

## 6. Course Schedule

To pace yourself and monitor your study progress, pay special attention to your Course Schedule. It contains study-unit-related activities including Assignments, Self-Assessments, and Examinations. Please refer to the Course Timetable on the Student Portal for the most current Course Schedule.

**Note:** Always make it a point to check the Student Portal for announcements and updates.

## 7. Learning Mode

The learning approach for this course is structured along the following lines:

- a. Self-study guided by the study guide units. Independent study will require *at least 3 hours per week.*
- b. Working on assignments, either individually or in groups.
- c. Classroom Seminars.

### StudyGuide

You may be viewing the interactive StudyGuide, which is the mobile-friendly version of the StudyGuide. The StudyGuide is developed to enhance your learning experience with interactive learning activities and engaging multimedia. You will be able to personalise your learning with digital bookmarking, note-taking, and highlighting of texts if your reader supports these features.

### Interaction with Instructor and Fellow Students

Flexible learning—learning at your own pace, space, and time—is a hallmark at SUSS, and we strongly encourage you to engage your instructor and fellow students in online discussion forums. Sharing of ideas through meaningful debates will help broaden your perspective and crystallise your thinking.

### Academic Integrity

As a student of SUSS, you are expected to adhere to the academic standards stipulated in the Student Handbook, which contains important information regarding academic policies, academic integrity, and course administration. It is your responsibility to read and understand the information outlined in the Student Handbook prior to embarking on the course.

# **Study Unit**

# **1**

## **Introduction to Application Analysis and Design**

## Learning Outcomes

By the end of this unit, you should be able to:

1. Identify the various approaches to design and analysis
2. Perform basic analysis to derive the requirements for a system
3. Analyse and validate requirements
4. Create requirements specification document using templates and UML diagrams  
(use cases and activity diagrams)

## Overview

This unit introduces application analysis and design as an engineering activity as performed by an analyst. The use of a systematic and structured methodology to perform requirements analysis is emphasised. It also introduces the application development life cycle models along with reasons and situations for when to use them. Case studies are presented to explain various concepts.

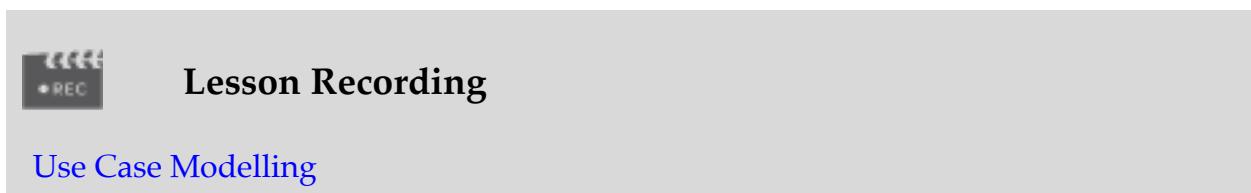


### Read

The study guide is developed based on Part 1 (Chapters 1 – 9) and Part 4 (Chapters 22 – 25) of the book, Sommerville, I. (2016). *Software Engineering, 10th Ed.*

# Chapter 1: Software Processes

## 1.1 Introduction



In the current digital revolution, the computer software has become an indispensable technology for business, science and engineering. With the internet, the lives of billions have changed as software evolved from being a product to being offered as a service with 'just-in-time' functionality for online shopping, news and other personal needs. Software embedded in transportation, medical, telecommunication, entertainment, home, office and other systems continues to change the way people interact with these systems. Emerging technologies in artificial intelligence and machines resulted in critical systems that were once mechanically controlled or controlled by people, are now dependent on code. As time passes and new software is created, more and more people are required to maintain these computer programs for corrections, adaptations and enhancements. Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs. Hence, expenditure on software represents a significant fraction of GNP in all developed countries.

The software community continues to develop technologies to make it easier, faster and less expensive to build and maintain high quality computer programs. Due to the vast diversity of software, it is highly unlikely that a unified approach would emerge that would help software developers.

This course is a study in software engineering which is the "application of a systematic, disciplined, quantifiable approach to solve business problems through the design,

development, operation and maintenance of software; that is, the application of engineering to software"<sup>1</sup>. As with any engineering activity, a software engineer starts with problem definition and applies tools, methods and processes of the trade to obtain a problem solution. Software engineering is a software process focused on the phased approach of developing a software or system, commonly known as Software or Systems Development Life Cycle (SDLC).

SDLC is not a rigid prescription of how to build computer software, but rather it is an adaptable approach to allow the software team to pick and choose the appropriate activities, work actions and tasks. An activity is carried out to meet a broad objective, e.g. communicate with stakeholders. Stakeholders are anybody that has a vested interest in the system development. This includes the system owners, the system administrators, users and software engineers. An action is a set of tasks that produce a major work product, e.g. a design model. A task is a small, well-defined objective that produces a tangible outcome, e.g. coding a module. The intention is to produce software that meets requirements, is timely and within budget.

The software engineer or systems analyst must understand and analyse the requirements of the business and design software to meet the requirements. They should be able to understand the business processes in multiple disciplines. They should be able to communicate with experts of different domains and gather information, seek clarifications, resolve inconsistencies, ensure completeness of the requirements and extract an abstract model of the requirements. They produce the software system design to meet the requirements, test the system and gracefully evolve with the evolving business needs for many years in the future.

Computer programmers receive specifications from software engineers and turn program design into written instruction codes for computers to follow.

---

<sup>1</sup> [www.computer.org/sevocab](http://www.computer.org/sevocab)



## Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.18-24.



## Activity 1.1

What is software engineering?

How does it differ from systems engineering?



## Watch

System Engineering: <https://tinyurl.com/systemsengg>

## 1.2 Essential Attributes of a Good Software

The goal of software engineering is to produce a high quality system or application within a given time frame and satisfy customers' needs. Besides ensuring that the software does what it is supposed to, the organisation of the programs and associated documentation, the behaviour of the software during execution are also a concern. For example, the software's response time to a user query and how easy it is to make changes to the program code. Certain attributes are more important and are dependent on the application. For example, an interactive game must be responsive, a banking system must be reliable and a cloud application must be available. In general, useful indicators of a good software are correctness, acceptability, dependability and security, efficiency and maintainability.

### 1.2.1 Correctness

Correctness is the degree to which the software performs its required function. Applications must operate correctly or it provides little value to its users.

### 1.2.2 Acceptability

Software must be acceptable to the type of users for which it is designed. If it is not easy to use, it is often doomed to failure, even if the functions that it performs are valued. Software must also be understandable and relates to the way the business process is executed. Lastly, the software must be compatible with other systems that the users use. Aesthetics or the look and feel of a software should also be considered.

### 1.2.3 Dependability and Security

Software dependability includes reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Software has to be secure so that malicious users cannot access or damage the system. Sensitive content should be protected and data transmission should be secure. An estimate of the business risk should be assessed on the system in the event of potential application failures when the application is in operation, and contingency measures to handle the event.

### 1.2.4 Efficiency

Software should not make wasteful use of system resources such as memory or processor cycles. Efficiency therefore includes responsiveness, processing time, resource utilisation, etc. The causes of inefficiency are usually violations of good architectural and coding practices.

### 1.2.5 Maintainability

Maintainability is the ease with which a program can be corrected if an error occurs, adapted if its environment changes, or enhanced if the customer requirements change. This is a critical attribute as the customer has put in an investment in the software.

and requirement changes are inevitable due to a changing business environment. Poor maintainability is typically the result of thousands of minor violations of best practices in documentation and basic programming practices.



## Watch

Attributes of good software:

<https://www.youtube.com/watch?v=BfXlt1mQ6Lo>

## 1.3 Software Engineering Ethics

Software engineering has evolved into a respected, worldwide profession with commitment to the health, safety and welfare of the public. This is the result of software engineers behaving in an ethical and morally responsible way with wider responsibilities than mere technical skills. The IEEE/ACM Software Engineering Code of Ethics specifies the eight principles. Each of these eight principles is equally important. Refer to Figure 1.3 of Software Engineering by Sommerville, p.29.



## Read

Software Engineering Code of Ethics by IEEE/ACM

<https://ethics.acm.org/code-of-ethics/software-engineering-code/>

In 2012, Singapore established the Personal Data Protection Act (PDPA) to regulate the flow of personal data among organisations. This came at a time where vast amounts of personal data were collected, used and transferred with increasingly sophisticated technology. Ethics is not more than simply upholding the law and plays a significant role

in software quality, the culture, and the attitudes of software engineers. Software engineers have a professional responsibility in the area of confidentiality, competence, intellectual property rights and computer misuse.



### Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.29-32.

## 1.4 System Development Life Cycle

Software or system development is complex. Developing a quiz app for an iPhone is different from building an international banking system or building a safety monitoring system for a nuclear reactor. Guidance is required to avoid getting lost in the maze of what needs to be done. The System Development Life Cycle (SDLC) defines the processes needed to design, build, test and maintain a software. It is a predictable phased approach to developing software. Each phase has a set of activities (things that need to be done), methods (how to do them), best practices (how best to do them) and deliverables (what to show) to assist the software engineer. It is accepted as an organised, controlled and stable approach which ensures that activities, actions and tasks performed are appropriate for the software to be produced.

As a guide for software engineers, the SDLC is important because it is a consistent process. Every system development has the same phases. It also makes sharing of resources (e.g. system analysts, system designers, software developers) between projects possible. This creates efficiencies when a software developer's skills are identified and allocated appropriately. Documentation produced is consistent and this reduces the lifetime costs in maintaining the systems. Lastly, dividing the system development into several phases allows easier project management and promotes quality and control. Before the next phase can proceed, the previous phase is signed off by the software owners and project managers.

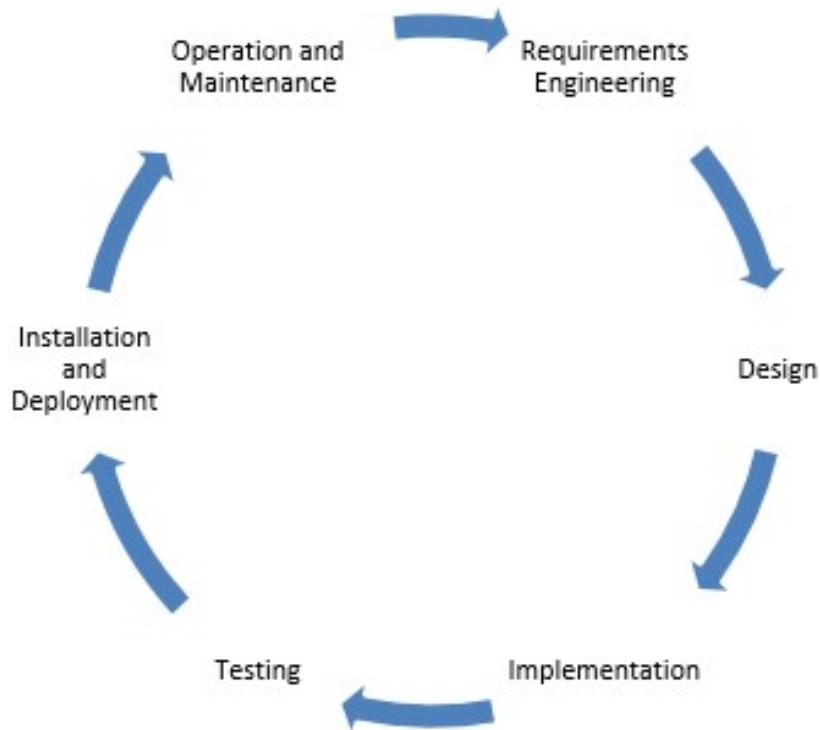
The entire process is best explained with a linear flow of the phases as follows:

- a. Requirements Engineering
- b. System Design
- c. System Implementation
- d. System Testing
- e. System Installation and Deployment
- f. System Operation and Maintenance

As time passes, there might be significant changes that are required in the system. This could arise from any of the following:

- a. Competitive pressures from other firms
- b. Advances in technology that make possible new features and functions
- c. Changes in the business environment
- d. Feedback from users and customers
- e. Changes in priorities from the management

As a result, it may be necessary to begin work on a new development. Thus the series of activities in the phases would be repeated as shown in Figure 1.1. This is why the term System Development Life Cycle (SDLC) is often used to describe the entire process.



**Figure 1.1** System Development Life Cycle

Each phase in the SDLC is described briefly in the following sub-sections.

#### 1.4.1 Requirements Engineering

This is the "What" phase. What the system should do is identified here. It is where system objectives are understood and requirements specifications are stated. In this course, we will refer to this set of requirements specifications as the negotiated statement of requirements.

The software engineer or system analyst gathers requirements from the stakeholders and analyse the requirements of the system to draw up the system specifications. The software engineer will review the input data, output data, the processing, the storage, the scope, the constraints and any other information relevant to the system. The result is a set of specifications that would be used for the design of the system.

## 1.4.2 System Design

This is the "How" phase. In this phase, answers to questions on how to do it are obtained. Questions could be: how best to develop the software, what are the various parts and how these parts fit together, etc.

The software engineer would design the system based on the analysis done in the previous phase. Depending on the methodology adopted, various tools and techniques may be used to assist in the design process. At this point of time, the system architecture, data structures, basic network configurations, hardware configurations may also be identified.

## 1.4.3 System Implementation

The "Do" phase. In this phase, code is written according to the system design.

The programmers will be involved in this phase. A programming language would also be used. Various software tools could be adopted to generate the code. The programs are written or generated based on the design specifications drawn up by the system engineer or system designer. The various components of the system are integrated into a complete system. As a general rule, programmers are not permitted to deviate from the given specifications without prior approval from the system engineer or the user.

## 1.4.4 System Testing

In the testing phase, the constructed system is put into a special test environment to check for errors and bugs and checked to ensure that the design fulfils the system requirements.

Various tests are conducted to ensure that the system will work as required. These are conducted at several points. Individually, the programmers will ensure that their code works. As a group, the developers will ensure that the system works, both on its own and when integrated with the rest of the systems in the organisation. The final tests are the acceptance tests where the users will check that the system executes as specified in the negotiated statement of requirements.

### 1.4.5 System Installation and Deployment

Most applications today are not standalone systems. They are required to work with other systems or need to exchange data with them. Thus, **in this phase, after the system is installed in the production environment, before an application goes into operation, it has to be integrated with the rest of the systems in the organisation.**



**Figure 1.2** A Banking System integrated with other systems

**As part of the deployment process, the system is "cutover" to become operational. All user data are also transferred from the old system, if any, to the new system. The transfer of data is known as data conversion.**

**At the same time, the developers would hand over the documentation for the system.** These are the user manuals such as the application configuration guide, the training guides, the operating procedures, the **clerical procedures** and so on.

In a major project, there could also be many other activities taking place simultaneously to deploy the application. For example, the project manager may have to ensure the following:

- Evaluate the suitability of cloud deployment for the application.
- Evaluate hardware availability in the market and shortlist suitable vendors.
- Prepare the tender document for purchasing the hardware and evaluate tender submissions from the vendors.
- Write papers of recommendations for approval of purchases, depending on the value involved.
- Coordinate the delivery, installation and testing of hardware.
- Coordinate the installation and testing of the networks.
- Plan for the physical facilities, including the physical space, furniture and equipment, power cabling, air-conditioning and so on.
- Prepare the facilities and the sites for installing the hardware. This may include the central computer room for the servers and the individual locations for the client terminals.
- Arrange for the non-technical user training and the technical operator training.

#### 1.4.6 System Operation and Maintenance

Finally, the users would take over and run the system. The technical staff would shift into a supporting role, taking charge of the technical aspects of running the application. This is the maintenance phase of the SDLC. The system is continually assessed as to whether it continues to meet its requirements as well as being evaluated for performance. Changes are then made to the system if these are not met. If the system meets the requirements of the user, the operation of the system could last several years.

Two broad categories of activities are generally recognised in maintaining the system:

- a. Resolving problems

Systems in real life are highly complex. It is very difficult to test them so thoroughly that all problems in the system are removed before it is put into operation. Software problems do arise due to undiscovered flaws or unexpected circumstances. Maintenance to resolve problems is an ongoing activity that may require a team of computer staff assigned to the job.

b. Enhancing the system

Business environment changes constantly. Requests for new features to be added to an operational system are very common. Some of these may be accommodated within the current design of the system. Others may require major design changes.

One task for the system developers is to differentiate between a system problem and an enhancement not stated in the original specifications. Users commonly do not distinguish between these. Typically, many regard any needs not in the system as a "system problem" that has to be resolved the day before.

As previously noted, there might be enhancements that require significant changes to the system as time passes. The developers would return to draw up a set of new requirements. Thus, the cycle of development activities would be repeated.



### Reflect 1.1

Is software engineering difficult? Why?

Is SDLC still relevant? Are there alternatives?



## Watch

SDLC in 9 mins!:

<https://www.youtube.com/watch?v=i-QyW8D3ei0>



## Watch

Why is software engineering important?

<https://www.youtube.com/watch?v=R3NzTt0BTWE>



## Activity 1.2

Briefly discuss why it is usually cheaper in the long run to use software engineering methods and techniques for software systems.

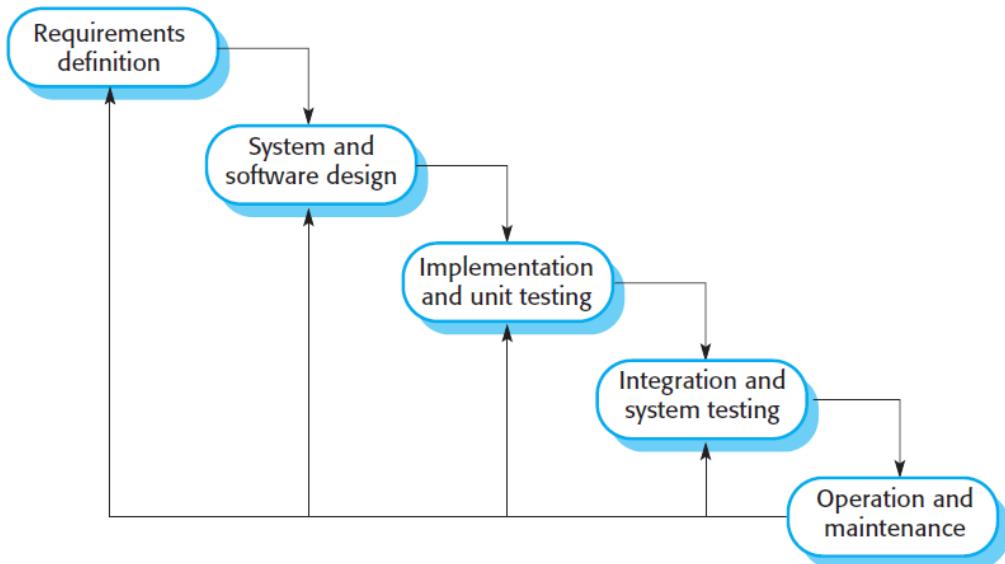
(Question reproduced from Q1.3 on page 41 of Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

## 1.5 System Development Approaches

There are various approaches to the SDLC process. Each has its own advantages and disadvantages. All accommodate the generic SDLC activities as described in the previous section, but each applies a different emphasis to these activities and defines a process flow that invokes each activity in a different manner. In this section, you will learn about common development approaches.

### 1.5.1 Waterfall Approach/Model

The early inspiration for software lifecycle came from other engineering disciplines, where the SDLC activities usually proceed in a sequential manner. This method is known as the Waterfall process or Waterfall approach because developers build monolithic systems in one fell swoop. It requires completing the artefacts of the current phase before proceeding to the subsequent one.



**Figure 1.3** Waterfall Approach

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

Figure 1.3 shows the phases in the Waterfall approach. The drawback of this approach is the difficulty of accommodating change after the process is underway. Just like a physical waterfall where water cannot flow upwards, the Waterfall approach does not permit going back to previous phases. This approach is therefore only appropriate when requirements are well defined and changes only limited during the design phase. In reality, few business systems have stable requirements. The Waterfall model is mostly used for large systems engineering projects where a system is developed at several sites. In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

### 1.5.2 Incremental Development Approach/Model

Incremental development approach or Prototyping model uses the exploratory approach to develop a system, as shown in Figure 1.4. Its key features are:

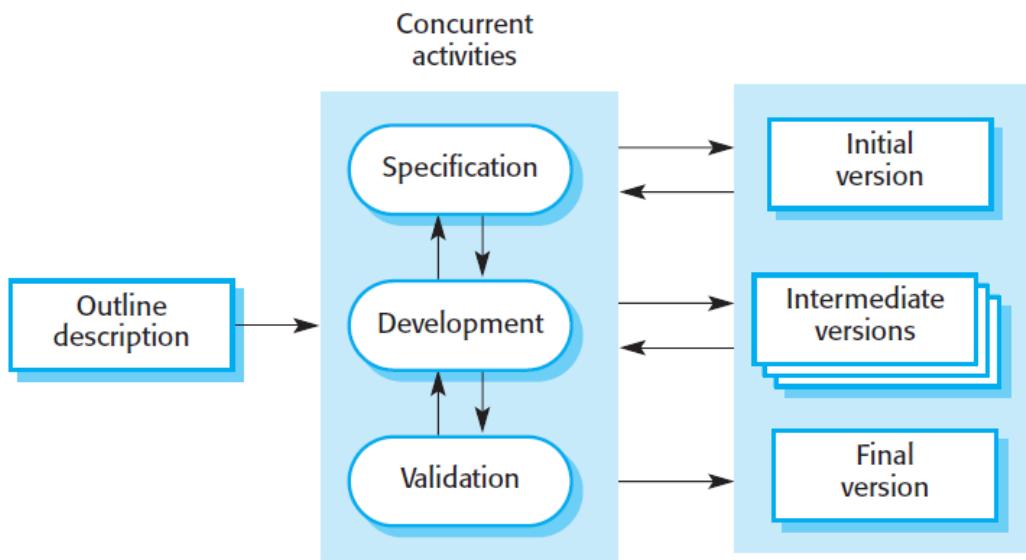
- Developing an initial version (basic prototype) rapidly and inexpensively.
- Letting the users experiment with it and give their feedback.
- Iterating over the process of incorporating user feedback and refining the prototype several times.

The incremental development model combines elements of linear and concurrent process flows such that as time progresses, deliverable "increments" of the software are produced. Usually, the initial increment has the basic requirements addressed but many other features remain undelivered. This basic prototype is used by the customer and undergoes detailed evaluation with a plan to develop the next increment. The next increment would address the modification of the first version to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced. Incremental development can be a plan-driven approach where the increments are identified in advance, or an agile approach where the early increments are identified, but the development of later increments depends on progress and customer priorities. See Section 1.5.4 on the Agile Development Approach.



Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.50-51.

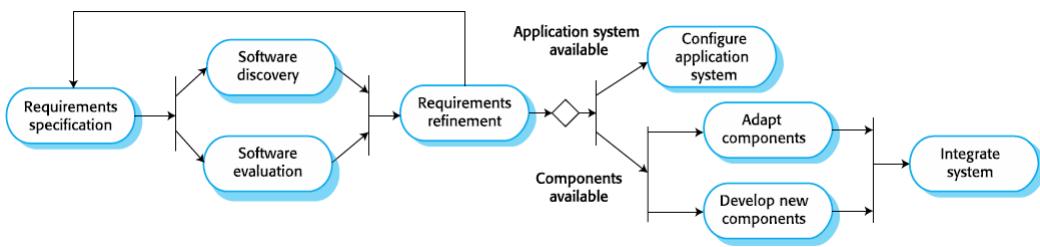


**Figure 1.4** Incremental Development

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

### 1.5.3 Integration and Configuration

Reuse-oriented model, as shown in Figure 1.5, is based on software re-use where systems are integrated from existing components or application systems. We refer to these software as Commercial-off-the-shelf (COTS) systems which are ready-made and available for sale to the general public. Examples of such systems are Microsoft Office Google G-Suite, Learning Management Systems from Canvas, Blackboard, etc., ERP/CRM systems from SAP, SalesForce, etc. Elements in the software may be configured to adapt their behaviour and functionality to a user's requirements. Re-use is now the standard approach for building many types of business systems.



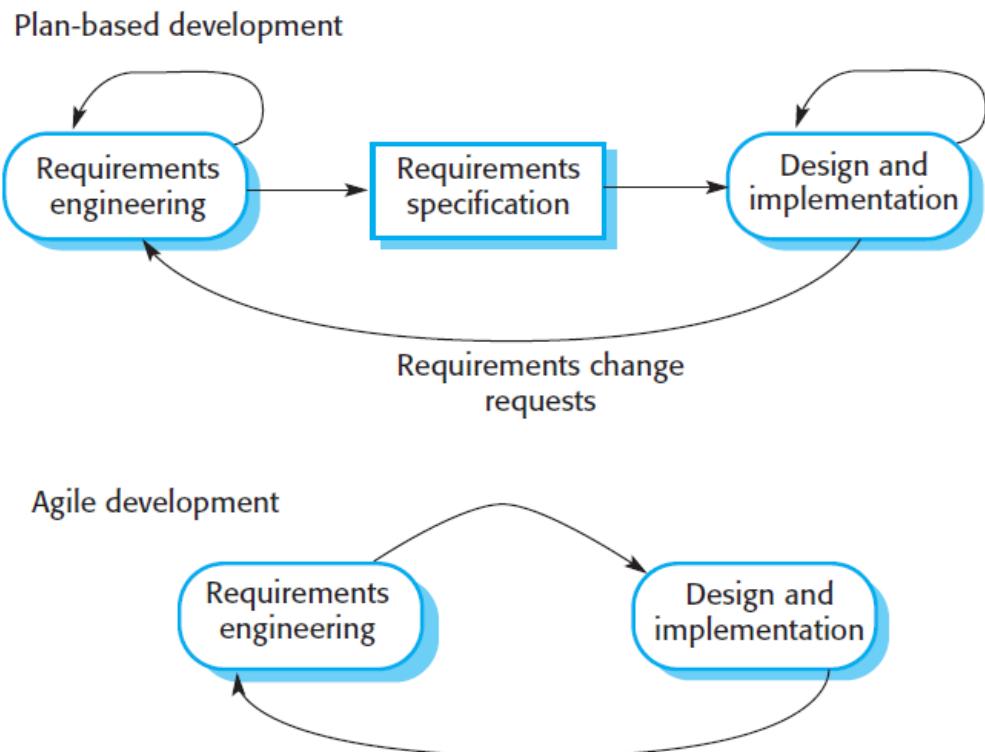
**Figure 1.5** Reuse-Oriented Development Approach

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

The reuse-oriented model has reduced costs and risks since the software is not developed from scratch. The delivery and deployment of the system will be faster than other approaches. However, it is hard to find a perfect software component or service that meets the requirements. Hence requirements are sometimes compromised and developers will have loss of control over the evolution of re-used system elements.

#### 1.5.4 Agile Development Approach

Plan-based development discussed above does not meet requirements of some of the business. More specifically, when the business needs software that is quickly produced and ever-changing, the agile development approach is more suitable.



**Figure 1.6** Plan-driven and Agile Development

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

Rapid development and delivery is now often the most important requirement for software systems. The modern business environment often dictates fast-changing requirements and it has become practically impossible to produce a set of stable software requirements. Software has to evolve quickly to reflect changing business needs. Agile development methods emerged in the late 1990s and aims to radically reduce the delivery time for working software systems.

In agile development, program specification, design and implementation are inter-leaved as shown in Figure 1.6. The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation. New versions are frequently delivered for evaluation. Such a rapid release of new versions is made possible with the support of extensive tools (e.g. automated testing tools) to develop, test

and maintain the system. Agile process encourages more focus on working code than documentation.



### Watch

An excellent short video that discusses 3 most impactful differences between agile and waterfall approach:

<https://www.youtube.com/watch?v=jL1VOF5JgPQ>

## 1.5.5 Case Study Example

A particular university has tasked a software development team to design a mobile app to collect students' feedback after each seminar. We discuss two development approaches the development team could use.

### 1. Waterfall Approach

The waterfall approach would be used if the stakeholder in the university does not want to spend time with the development team. He/she only wants to specify the requirements and leave it to the development team to produce the app in the project time specified.

If this is the case, the development team would analyse the requirements and use elicitation techniques to clarify and confirm the requirements. These include minimally interviewing the stakeholder to understand their requirements and gathering documents such as questions and types of answers expected, student data, etc. The development team would then design the app, write the codes, test the app and ensure all goes well with the app before inviting the stakeholder to test the app. If the stakeholder requires changes, the development team would go back to the phase that requires changes to be done, e.g. design phase, and

work goes forward again from there. When the stakeholder is satisfied with the app, the app is released for use. The app is now in operation and maintenance phase.

## 2. Agile Development

The agile development approach would be used if the stakeholder in the university is willing to spend time with the software development team throughout the process of developing the app in the specified project time.

The development team may decide on a number of sprints (or iterations) for the project. As this is a simple project, the development team may decide to complete the app in 3 sprints (producing 3 versions). During the first sprint, the development team would gather and clarify requirements from the stakeholder, including any data (e.g. questions with types of answers) required. The development team would then proceed to design the app and write the code paying close attention to ensure that the app would work in a basic mode. At the beginning of the second sprint, the stakeholders would be invited to test the app and give feedback on the way the app worked including its 'look and feel' and gather further required data (e.g. student data, etc.). The development would then proceed to improve the app with such feedback. In the third and final sprint, the stakeholders would again be invited to test the app and give feedback. The feedback would be used to improve the app and may also include testing the app on a select number of students (part of the audience who would eventually use the app) before the app is finally presented to the stakeholder for acceptance and putting the app into operation.

In conclusion, the development approach to use depends on the stakeholder involvement and project complexity. In this case, the app is a simple one and the approach to use therefore depends on stakeholder availability. If the project is complex, and the stakeholder is not available, using the agile approach may also be appropriate.



### Activity 1.3

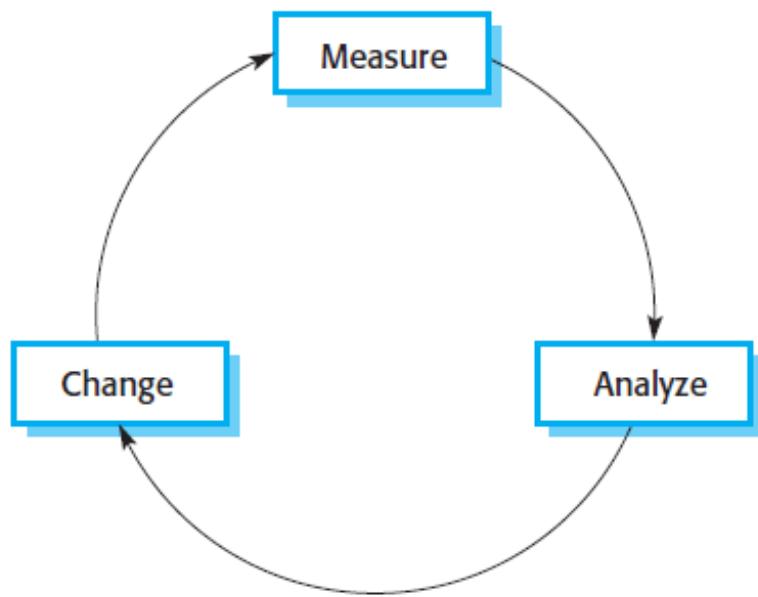
Suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems. Explain your answer according to the type of system being developed:

1. A system to control antilock braking in a car.
2. A virtual reality system to support software maintenance.
3. A university accounting system that replaces an existing system.
4. An interactive travel planning system that helps users plan journeys with the lowest environmental impact.

(Question reproduced from Q2.1 on page 70 of Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

## 1.6 Process Improvement

Defining and improving the various processes (e.g. the process of students registering for courses) are essential for efficient and successful operation of any company / organisation. Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time. It includes improving the process, project management and introducing good software engineering practices. Many software companies have turned to software process improvement as a way of enhancing the quality of their software, reducing costs or accelerating their development processes. The process improvement is a continuous process of measurement, analysis and change, as shown in Figure 1.7.



**Figure 1.7** The Process Improvement Cycle

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

A widely known process improvement model is the Capability Maturity Model (CMM), developed at the Software Engineering Institute (SEI), Carnegie Mellon University (CMU). It provides organisations with the essential elements of effective processes. The term "maturity" relates to the degree of formality and optimisation of processes, from ad hoc practices, to formally defined steps, to managed result metrics, to active optimisation of the processes. In 2002, a new Capability Maturity Model Integration (CMMI) was released.



### Read

- i. Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.65-68.
- ii. the original CMM tech report

[https://resources.sei.cmu.edu/asset\\_files/  
TechnicalReport/1993\\_005\\_001\\_16211.pdf](https://resources.sei.cmu.edu/asset_files/TechnicalReport/1993_005_001_16211.pdf)



## Activity 1.4

What are the goals and metrics for software measurement?

What are the five levels of CMM?

## 1.7 Case Studies

The following case study examples are referenced in this study guide.

### 1.7.1 Car Rental Company

The following is the statement of requirements for a car rental company, ABC Car Rentals.

For ease of reference, we have numbered each paragraph of the requirements:

#### Background information

1. ABC Car Rentals has several branches. Each branch has its own collection of rental cars and is uniquely identified by its location.
2. The company employs managers and branch workers that work in the branches. Each branch has one manager who is assisted by a few branch workers. A manager is responsible for only the branch assigned to him/her.
3. Not all employees of the company are branch workers or managers. For example, there are quality controllers who monitor the efficiency of all the branches.
4. Not every branch has cars with it. This is because some branches may be undergoing renovation or a new branch may not have cars delivered to it yet.
5. Customers sign a contract to rent a car. No customer may sign more than one contract. However, some customers who are business entities may wish to sign a contract for more than one car. For efficiency reasons, these cars must come from the same branch.

6. At the end of a contract, a customer may return a car to any branch of the company. However, the station workers will send the car back to the branch that is responsible for that car.

#### Purpose

7. The ABC Car Rental System (ABCCRS) will support the company's operations by providing the following reports through the Internet:
  - a. Information about each branch, including its location, status (operational or under renovation) and the manager in charge.
  - b. Given a branch location, information about the company's employees, including their names and employee numbers and, for branch workers, their overtime rate.
  - c. Given a branch location, information about the company's cars, including the engine capacity and the vehicle registration number.
  - d. Information about customers, including their names and addresses.
  - e. Information about each current contract, including the start date, duration of the contract and the customer name.

### 1.7.2 Mentcare System

Please refer to the above system in Section 1.3.2 of the book, Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.34-36.

### 1.7.3 Weather Station

Please refer to the above system in Section 1.3.3 of the book, Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.36-38.

## Chapter 2: Requirements Engineering

A requirement describes a capability or condition to which a system must conform. Requirements are supposed to represent "what" the system should do as opposed to "how" the system should be built. This is to prevent "solution bias", i.e. it should not suggest certain solutions or preclude others.

Requirements engineering encompasses the tasks that lead to an understanding of the requirements. It is a mechanism for understanding what the stakeholder wants. This is a major software engineering activity that builds a bridge to design and implementation. To avoid problems from arising during the requirements engineering process, the requirements may be distinguished as user requirements and system requirements.

### a. User requirements

These are high-level abstract requirements. They may contain broad statements of the system features required, written in natural language, supplemented with diagrams of what the system is expected to provide to system users and the constraints under which it must operate.

### b. System requirements

These are detailed descriptions of what the system should do which include the system's functions, services and operational constraints. The document that contains the system requirements may be part of the contract between the system buyer and the software developers.

The Mentcare system user and system requirements are shown in Figure 1.8. As you can see, the user requirement is quite general and the system requirements provide more specific information about the services and functions of the system that is to be implemented.

Requirements engineering involves various stakeholders in defining the problem and specifying the solution. Any person or organisation who is affected by the system in

some way and so who has a legitimate interest is a stakeholder. There are different stakeholder types with varying interests - managers, clients (owners), end-users, business analysts, system architects and developers, testing and quality assurance engineers, project managers, the future maintenance organisation, owners of other systems that will interact with the system-to-be, etc. For example, clients will be interested in costs and timelines; end users will be interested in how to effectively implement the functionalities and features. The stakeholders for the Mentcare system are patients, doctors, nurses, medical receptionists, IT staff, medical ethics manager, health care managers and medical records staff. Before being accepted as the official document for development, the requirements are reviewed, clarified, negotiated and agreed among all the parties. This document, called the Requirements Specification Document, will be discussed in detail in Chapter 3.

#### User requirements definition

- 1.** The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

#### System requirements specification

- 1.1** On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
- 1.2** The system shall generate the report for printing after 17.30 on the last working day of the month.
- 1.3** A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
- 1.4** If drugs are available in different dose units (e.g. 10mg, 20mg, etc.) separate reports shall be created for each dose unit.
- 1.5** Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

**Figure 1.8** User and System Requirements

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)



## Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.102-105.

Requirements can be classified into functional and non-functional requirements. The next two sections discuss these two types of requirements.

## 2.1 Functional Requirements

Functional requirements describe what a system does or is expected to do. We can view this as the services that the system provides for a human user or another system, how the system should react or behave in certain situations. These may be calculations, data manipulations, and processing. The typical information for functional requirements is as follows:

a. Inputs

This refers to the data that the system requires and the commands that the system should accept. The inputs could come from a human user or another system.

b. Outputs

This refers to the data that the system should produce and the commands that the system might generate. This could be printed reports, screen displays or information to be transmitted to another system.

c. Processing

Processing refers to the transformation of the input data into the output data. This includes all behaviours of that process. For example, if the system allows a user to login, it should handle valid and invalid logins. The description should be given in terms that the user understands. For example, instead of saying the

system will run a utility to perform a task, say that a small application will be executed.

d. **Scope**

This refers to the data being processed. Typical information includes how many of each type of data, what the range of values are and what are their attributes and characteristics.

e. **Storage**

Storage refers to the data that the system should retain. These are the data that would be required by the user or other systems at a later point in time. No implementation details such as storage format should be included since this is for the user requirements.

## 2.2 Non-Functional Requirements

Non-Functional requirements support functional requirements in that they describe how well these requirements are provided. These include performance requirements, security, reliability or usability. They describe the constraints imposed on the system as a whole and arise due to budget constraints, organisational policies, and the need for interoperability with other software or hardware systems or external factors such as safety regulations or privacy legislations. Hence these must be stated up-front and not only after the system is completed.

Non-functional requirements can be classified into product, organisational and external requirements. Figure 1.9 shows examples of these requirements as applied to the Mentcare system. Figure 1.10 shows the various types of non-functional requirements.

**PRODUCT REQUIREMENT**

The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 08:30–17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

**ORGANIZATIONAL REQUIREMENT**

Users of the Mentcare system shall identify themselves using their health authority identity card.

**EXTERNAL REQUIREMENT**

The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

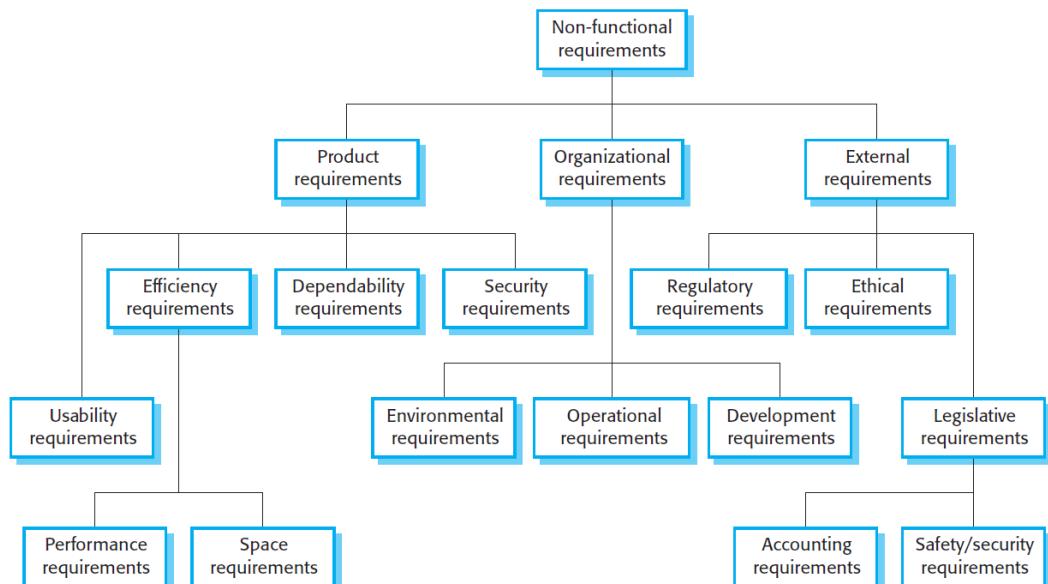
**Figure 1.9 Examples of Non-Functional Requirements for the Mentcare System**

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)



## Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.105-111.



**Figure 1.10 Types of Non-Functional Requirements**

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)



### Activity 1.5

The following is a brief description of an application that monitors the central air-conditioning system for a manufacturing plant that produces sensitive high technology equipment. The temperature must be maintained within a narrow acceptable range.

As part of the air-conditioning, the air is also filtered to remove tiny particles. The pressure drop across the filters is monitored to ensure effective performance of the filtration. When the filters are clotted with dust particles, the pressure difference across the filters will increase. The application should monitor the temperature of the air at ten different locations of the plant. It should record the temperature readings at fifteen-minute intervals to a database. It should also raise an alarm if any of the temperature readings cross the permissible limits. The application should also measure the pressure drop across the filters at three locations. If the pressure drop exceeds the pre-set limits, a signal should be sent to the maintenance personnel for the filters to be replaced.

The application should run on a PC-based system and should not require more than 512MB of memory. The PC should upload the temperature data to the server at the end of every working day.

Identify the functional and non-functional requirements from the above description.

## 2.3 Requirements Engineering Process

The processes used for requirements engineering vary widely depending on the application domain, the people involved and the organisation developing the requirements. However, there are a number of generic activities common to all processes.

These are:

- a. Requirements elicitation
- b. Requirements analysis
- c. Requirements validation
- d. Requirements management

Each of these processes will be discussed in the next four sections.

### 2.3.1 Requirements Elicitation and Analysis

The objective of requirements elicitation or requirements gathering process is to understand the current work of the customers, how the new system will help them to do the work better, the services of the new system and the operational constraints. The process involves all stakeholders. Requirements elicitation is very hard to do due to the following reasons:

- a. Stakeholders find it difficult to articulate what they want the system to do.
- b. Requirements engineers, without experience in the customer's domain may not understand their requirements.
- c. Different stakeholders express common requirements in different ways that can lead to conflict.
- d. Political factors may influence requirements to be mandatory or desirable.
- e. Economic and business environment makes changes inevitable.

Requirements elicitation is done together with requirements analysis. There are four stages in the requirements elicitation and analysis process:

- a. Requirements discovery

Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

- b. Requirements classification and organisation

Organising and grouping related requirements into coherent clusters.

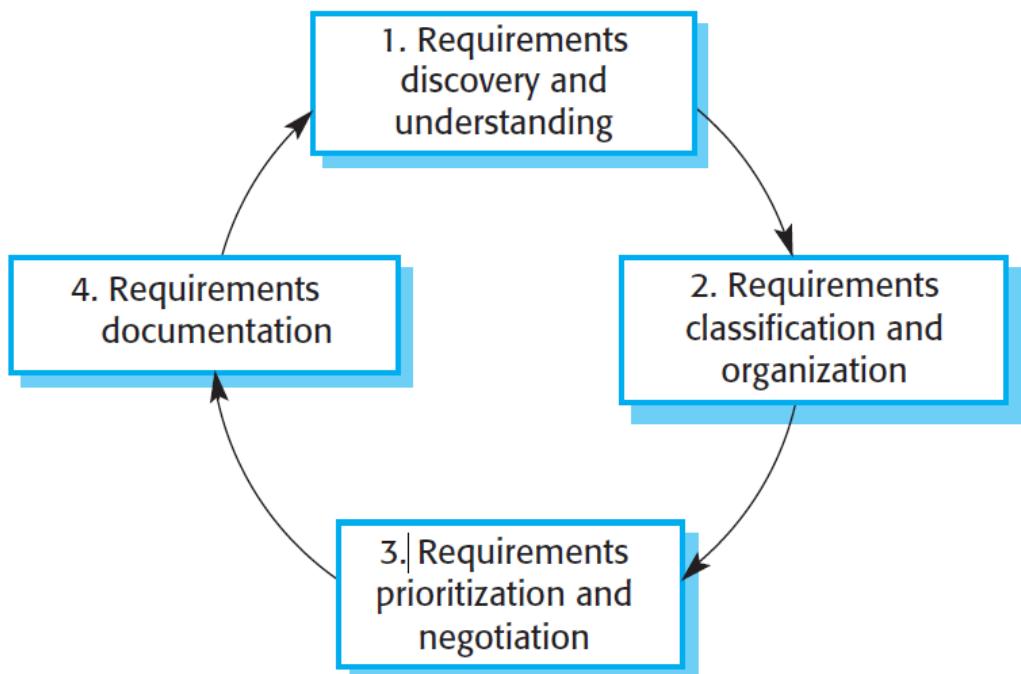
- c. Prioritisation and negotiation

Prioritising requirements and resolving requirements conflicts.

#### d. Requirements specification

Requirements are documented and input into the next round of the spiral.

Figure 1.11 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The analyst's understanding of the requirements improves with each round of the cycle. The process cycle starts with requirements discovery and ends when requirements document has been produced. Requirements documentation will be discussed in Chapter 3.



**Figure 1.11** The Requirements Elicitation and Analysis Process

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

### 2.3.2 Requirements Elicitation Techniques

Requirements elicitation is to collect information about the existing and required new system. Stakeholders and existing documents are the key sources of information. The techniques used to requirements discovery are listed below:

- **Interviews** – Formal and Informal interviews. Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system. Interviewers need to be open-minded without pre-conceived ideas of what the system should do. The interviewer need to prompt the user to talk about the system by suggesting requirements rather than simply asking them what they want.
- **Questionnaires (and surveys)** – Questions can be open (many different answers are possible) and/or closed (fixed set of possible answers, e.g. MCQs). This is another major technique for information gathering.
- **Ethnography (Observations)** – The developer spends a considerable time observing and analysing how people actually work. Requirements that are derived from cooperation and awareness of other people's activities. **Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.**
- **Stories and Scenarios** – Scenarios and user stories are real-life examples of how a system can be used. They describe how a system may be used for a particular task. Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect to the story.



### Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp. 115-119.

### 2.3.3 Requirements Validation

**Requirements validation is concerned with ensuring that the requirements define the system that the customer really wants. The requirements specification is examined to ensure that all software requirements have been stated unambiguously, that inconsistencies, omissions and errors have been detected and corrected and that the work**

products conform to the standards established for the process, the project and the product.

The requirements specification document is discussed in Chapter 3.

Requirements error costs are high, so validation is very important. Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error. This review process is held among the system analysts and the users. Requirements are checked for the following quality characteristics:

a. **Correct.**

A set of requirements is correct when it states accurately the features and functions to best support the customer's needs.

b. **Unambiguous.**

A set of requirements is clear and does not have any ambiguity. Developers and users may interpret any ambiguous requirement in different ways. For example, for the Mentcare System, a functional requirement for "search" could be:

A user shall be able to search the appointments list for all clinics.

The user could interpret this as, given a patient name, search for the patient name in all clinics for all appointments. This is because, a medical staff often has to attend to patients who arrive at a different clinic for their appointment and so needs to be informed to go to the correct clinic. The developer may interpret this as, given any clinic name, list all the appointments at that clinic – which requires the medical staff to do more "search" to attend to patients who have appointments at another clinic.

The requirement should be reviewed to clarify the exact type of search required.

c. **Complete.**

The requirements should not have any missing information. All features and functions required by the customer are included. Depending on the requirement, different component parts may be missing and therefore incomplete. Refer

to Sections 2.1 and 2.2 for the components required for functional and non-functional requirements. Consider the following non-functional requirement:

On loss of power, the battery backup must support normal operations.

Non-functional requirements should specify a minimum amount of some quality factor. The above requirement lets you guess as to how long the battery backup will be used. Will it be used as long as the battery lasts? A more complete requirement would be:

On loss of power, the battery backup must support normal operations for 20 minutes.

**d. Consistent.**

A set of requirements is inconsistent when one part contradicts another part. For example:

The electronic batch records shall be Part 11 compliant. An on-going training program for 21 CFR Part 11 needs to be established at the sites.

Are there 2 regulations here or are they the same regulations? A more consistent requirement could be stated as:

The electronic batch records shall be 21 CFR Part 11 compliant. An on-going training program for 21 CFR Part 11 needs to be established at the sites.

Keeping related features and functions grouped together allows contradictions to be easily found. Hence requirements should be systematically organised.

**e. Realism.**

This is a check on whether the requirements can be implemented given available budget and technology.

**f. Verifiability.**

The requirements should be verifiable for their correct implementation. This is done by writing a set of statements that can demonstrate that the delivered system meets each specified requirement. Examples:

The product shall work well.

The output of the program shall usually be given within 5 seconds.

These requirements cannot be verified because it is impossible to define the terms "well" or "usually". Requirement statements should use concrete terms and measurable quantities to be verifiable.

Though it is easy to keep requirements complete and consistent for small systems, it is hard for large systems. Large systems have many different stakeholders with differing levels of understanding about the system and hence inconsistencies are often present. Deep analysis is required to identify the inconsistencies and omissions.

In addition to these qualities, the system analyst should also ensure that the requirements exclude:

a. **Design Directives.**

A design directive is a requirement that constraints the choices of the designer. The system analyst should be able to analyse the requirements and from there, produce a most suitable design for the system. He/she should be free to decide on the best design for the system based on the agreed requirements. Some examples of design directives are:

- One indexed data file should be used for each tutorial group.
- The system should use a client-server architecture.
- The database should be properly normalised.

b. **Implementation Directives.**

An implementation directive is a requirement that limits the choices of the developer. System developers should implement the system according to the design of the system. They should be free to decide on the best implementation for the system based on the design. Some examples of implementation directives are:

- The system should be tested with a group of students before implementation is considered complete.
- The system should be developed using an object-oriented language.

c. **Platitudes.**

A **platitude** is a statement that is said so often that it is obvious and meaningless. A set of requirements should be concise as it is a working document. **Unnecessary statements that have insufficient details and do not add to the contents should be removed.** Some examples of platitudes are:

- The system should be user-friendly.
- The system should be cheap and good.

Instead of user-friendly, the requirement should specify, for example, how the help facilities should be provided to the user, so that the user finds it easy to use the system.

The primary requirements validation mechanism is a technical review. The review team that validates requirements includes software engineers, customers, users and other stakeholders who conduct a systematic manual examination of the requirements specification looking for the above qualities. Prototyping is also used, where a basic executable model of the system is developed and checked with users whether their requirements are met. Lastly, since requirements are testable, test cases are generated to verify the requirements. If it is hard to develop a test case for a requirement, it would be hard to implement as well.

**Example 1: Vehicle Defects Monitoring System**

A major vehicle manufacturer with markets worldwide has manufacturing plants in Asia, Europe, North America and South America. It wants to implement a system for monitoring defects encountered in its vehicles. **Here is a part of the requirements for the system:**

1. The system should provide information for its quality control personnel about the defects encountered in all the current and past models of vehicles that its various plants manufactured. The information is gathered from its network of distributors with their maintenance workshops. These maintenance workshops are a source of useful information because they provide the maintenance for the warranty period. The information is transmitted in real time over the Internet to the database in the central computer at its headquarters in Asia.
2. The system should retain information about occurrence of defects in the past. Listed below are the options that should be available to the quality control staff for retrieving these records. The options should be easy to use. The system should provide a response within one second of the selection of an option.
3. Details by type: This option provides information about the defects by type. Sometimes there could be an inherent design flaw that may require investigation.
4. Details by count: This option gives the number of defects encountered. This could be useful to monitor the trend in the quality standard.
5. Details by plant: This option gives information about the average number of defects encountered for vehicles produced by a specified manufacturing plant. This would be useful to monitor the performance of each plant.
6. Details by season: This option provides information about defects reported during each of the four seasons of a year. This could be useful for investigating the impact of the environment on the performance of the vehicles.

Review the requirements given and identify the problems that have to be corrected before you can begin any development work.

The following are some of the major problems with the requirements:

#### **Paragraph 1:**

There is no mention of how far back the system should include for the "past" models.

Clarification is needed as to whether historic data about such models are required. If so, how these data are to be obtained is a significant issue. There should also be a reference

to a document that spells out the defects information being transmitted to the central database. Presumably, information such as defects should be categorised in some way. The information available in the paragraphs that follow may not be adequate to help the analyst in organising them for future retrieval. There could also be the possibility that the distributor workshops may report problems arising from accidents. It may be necessary to sieve out data arising from accidents rather than manufacturing and design defects. This is not as straightforward as it seems because some accidents may be due to manufacturing or design flaws.

To complicate the picture further, there could also be defects arising from faulty repairs by unauthorised workshops. There could also be vehicles, manufactured by one of the plants but imported elsewhere by a parallel importer and thus, maintained by workshops that do not belong to the distributor during the warranty period.

**Paragraph 2:**

There is no mention of how far back into the past should information be maintained. For example, defects for models that are discontinued for a few years may not be of interest to the quality controllers since the technology for them could already be obsolete. It is also not clear whether only defects reported during the warranty period should be included.

The statement "The options should be easy to use." is a platitude and is too vague. If there are any special needs that the quality controllers have for "easy to use", the details should be spelt out in the requirements.

**Paragraph 3:**

This requirement is too vague. The word "type" should be explained. Quality controllers for a large manufacturing company would want more than simply "defects by type". For example, does "type" refer to the type of vehicle, the type of the vehicle model design, the type of engine, the type of control system and so on?

**Paragraph 4:**

This requirement is again too vague. It does not indicate how the "count" should be made. For example, does it refer to all types of defects, all types of vehicle and all the distributors and plants worldwide? Over what period the count should be made? Should it be for defects during the warranty period only? Is the same defect encountered in two distinct vehicles counted as one or two? Is the same defect encountered twice for the same vehicle counted as one or two?

**Paragraph 5:**

This requirement does not specify clearly what is meant by "average". The term "average" could be interpreted as average number of defects per month, per year or other time intervals. It could also be referring to average number of defects per vehicle, per hundred vehicles and so on, produced by the plant. It could also be referring to the average number of defects per vehicle type or per model produced by the plant. Furthermore, the paragraph stated that the information is used to monitor the performance of a plant. There are issues such as a vehicle manufactured in year one with a defect that is reported in year two. Should the defect be counted as that in year one or year two? For one case, the system has to check the year of manufacture. For the other, the system simply takes the date of the report.

**Paragraph 6:**

Besides the usual questions about the definition of the term defects, this requirement requires several clarifications. These include how the seasons are precisely determined. There is also the definition of the term seasons for temperate countries where the weather is not like the tropics, equivalent to summer, but neither does it resembles the entire four seasons.

**Lessons to learn from this example:**

This example shows several important practical points to note:

- Drawing up a set of requirements is painstaking and detail work. Close examination of a set of requirements often reveals many details that have to be resolved.
- Considerable amount of clarifications is often required. Some omissions can have a big impact on the design and implementation of the system.
- The analyst cannot rely completely on the users' initiative to tell him or her everything that is required.
- The analyst must never assume especially if his or her knowledge of the application area is inadequate.
- The work of the analyst is not simpler than that of a programmer. Hopefully, this example helps to dispel any such notion.

### **Example 2: Automatic Patient Monitoring System**

Imagine that you have to develop an automatic patient monitoring system to monitor vital signs of a home-bound patient, especially blood pressure and heart rate. The system is required to read out the patient's heart rate and blood pressure from the devices and compare them against specified safe ranges. Assume the devices have appropriate interfaces (APIs – Application Program Interface) to read the data to the automatic patient monitoring system. The system has activity sensors to detect when the patient is exercising and adjust the safe ranges. The system must alert a remote hospital when the heart rate or blood pressure is out of range. (Note that the measurements cannot be taken continuously, since heart rate is measured over a period of time, say 1 minute, and it takes time to inflate the blood-pressure cuff.) The system must also (i) check that the analogue devices for measuring the patient's vital signs are working correctly and report failures to the hospital; and, (ii) alert the owner when the battery power is running low.



**Figure 1.12 Blood Pressure Monitor**

(Source: <https://www.omronhealthcare-ap.com/ap/category/8-blood-pressure-monitor>)

The requirements for the automated patient monitoring system are listed below:

**REQ1:** The system shall periodically read the patient's vital signs, specifically heart rate and blood pressure.

**REQ2:** The system shall detect abnormalities in the patient's vital signs.

**REQ3:** The system shall alert the remote system at the hospital when an abnormality is detected.

**REQ4:** The system shall detect when the patient is exercising and adjust the safe ranges of vitals.

**REQ5:** The system shall verify that its sensors are working correctly. The system shall report sensor failures to the remote hospital.

**REQ6:** The system shall monitor its battery power. The system shall alert the owner when its battery power is low.

### 2.3.4 Requirements Change Management

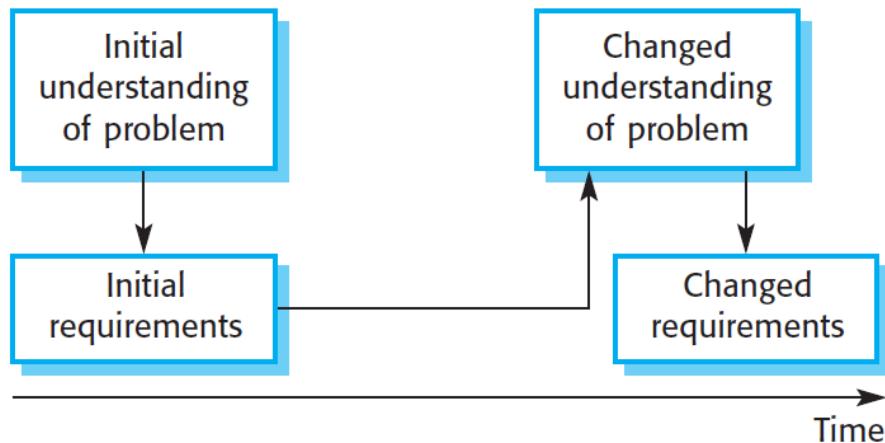
The desire to change requirements persists throughout the life of the system, esp. for large systems. Some reasons for these changes could be:

- a. The business and technical environment of the system changed after installation.
- b. The people who pay for the system and the users of that system are rarely the same people.
- c. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

Requirements management is a set of activities that help the project team identify, control and track requirements and changes to requirements during the requirements engineering process and system development. New requirements due to changed understanding of the problem as a system are being developed and after it has gone into use is shown in Figure 1.13. You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You

---

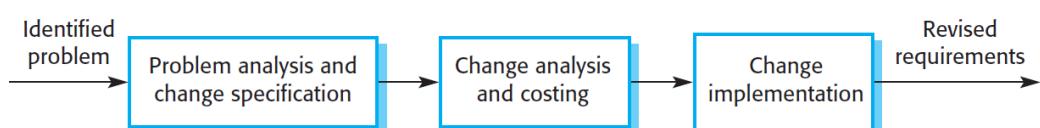
need to establish a formal process for making change proposals and linking these to system requirements.



**Figure 1.13 Requirements Evolution**

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

For each identified problem, the requirement management process goes through the stages of problem analysis to check that it is valid and the effect of the proposed change is assessed before a decision is made as to whether to proceed with the requirements change. A decision made to implement the change would mean that all documentation (e.g. requirements document, system design and implementation) will be updated with the change. The process is shown in Figure 1.14.



**Figure 1.14 Requirements Change Management**

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

## Chapter 3: System Modelling

System modelling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. Unified Modelling Language (UML) graphical notations are commonly used in system modelling.

Models are abstracts (details are omitted) of the system to easily understand the system. Models of the system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. In a model-driven engineering process (Brambilla, Cabot, & Wimmer, 2012), engineers can generate a complete or partial system implementation from system models.

Different models are used to describe different perspectives of the system. In this course we will be using the following models:

- **Context Models:** Context models are used to illustrate the operational context of a system – they show what lies outside the system boundaries. Social and organisational concerns may affect the decision on where to position system boundaries.
- **Architectural Models:** Architectural models are used to show the system and its relationship with other systems.
- **Interaction Models:** Used to describe interaction perspective of the system, where you model the interactions between a system and its environment, or between the components of a system. UML Use Cases and Activity Diagrams are used to design interaction models.
- **Structural Models:** Used to describe structural perspective of the system, where you model the organisation of a system or the structure of the data that is processed by the system. UML class and object diagrams are used to design structural models.
- **Dynamic Models:** Used to describe behavioural perspective of the system, where you model the dynamic behaviour of the system and how it responds to events.

UML state machine diagrams and sequence diagrams are used to design dynamic models.

We can look at the development of an object-oriented application with UML through two major modelling efforts.

- a. Modelling to understand the system boundaries and major components. Context models and Architecture models are used.
- b. Modelling to understand the functional requirements. Interaction models are used.
- c. Modelling to perform the design. Structural and Dynamic models are used.

### 3.1 Context Models & Architecture Models

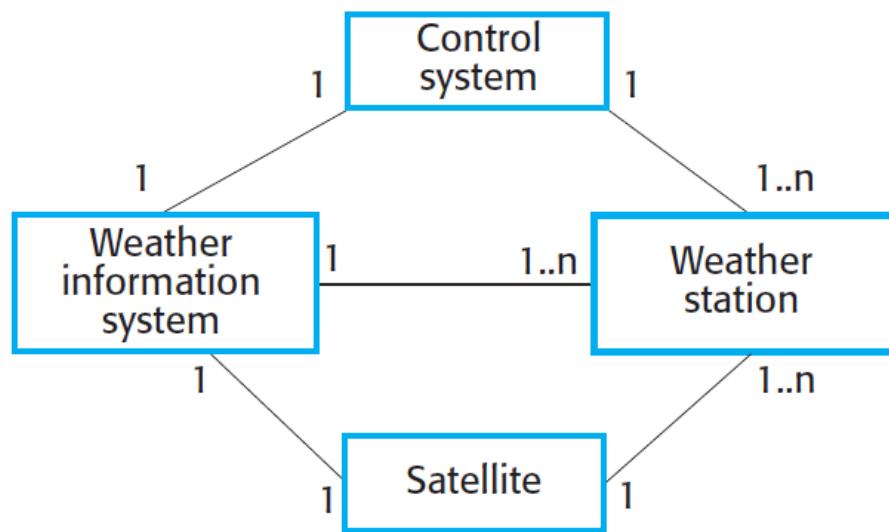
Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

#### **Example 3:** Weather Station – Context Model

Wilderness weather stations are deployed in remote areas. Each weather station records local weather information and periodically transfers this to a weather information system, using a satellite link. Refer to this case study in Section 1.3.3 of the book, Sommerville, I. (2016). *Software Engineering, 10th Edition*, pp.36-38.

The context model for the weather station is shown in Figure 1.5. It shows the boundaries of the weather station system. The numbers '1' and '1...n' indicate the type of relationship between the weather station system and other systems in its boundary. For example, the notation 1 and 1...n between the Control station and weather station indicates that One control station controls One or More weather stations.

Once interactions between the system and its environment have been understood, you use this information for designing the system architecture. System architectures can be designed with different views or perspectives. In this course, we are interested in component level (or package level) architecture. You identify the major components that make up the system and their interactions, and then may organise the components using an architectural pattern such as a layered or client-server model.

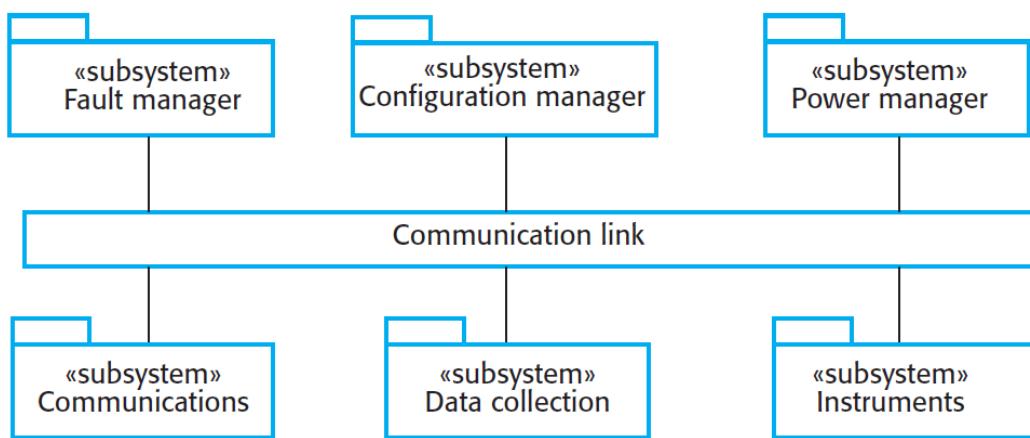


**Figure 1.15** System Context for the Weather Station

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

#### **Example 4:** Weather Station – Architecture Model

The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.



**Figure 1.16** High-level architecture of Weather Station

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition.*)

## 3.2 Use Cases and Interaction Modelling

### 3.2.1 Use Cases

The term **use case** is used to refer to a function that an application has to provide to the users. Use cases are powerful graphical tools for modelling the system requirements. Use cases are part of the UML specification. Essentially, a **use case** is a set of action sequences that an application performs to provide some benefits or results for the user. Use cases show the interactions between a system and its environment. Use cases are supplemented by more detailed tabular description. A set of use cases should describe all possible interactions with the system.

The use case represents the goal that the user wants to achieve with the application. It has the following characteristics:

- It is a function or task that has to be provided by the application.
- It is an activity that is seen from the user point of view.
- It has value that a user gets from interacting with the application.
- It can consist of more than one related scenario, such as a successful and an unsuccessful outcome.

- It can be used as a communication tool between the system analysts and the users.

All the use cases for an application taken together would represent all the possible ways of using the application.

To illustrate the idea of a use case, consider a library application. Such an application would be expected to provide the following functions to its users:

- Borrow a book
- Return a book
- Reserve a book
- Renew a book
- Search for a book

These would be typical use cases for the library application.

When a use case instance takes place, we call it a scenario. For example, 'Jane borrowed a book from the library this morning' is a scenario for the use case 'borrow book'.

A use case diagram contains all use cases for a system. A use case in a use case diagram is a short descriptive name that is indicative of its function written inside an oval. The name starts with a verb representing the action followed by a noun because it represents an action on an object. Use present tense for the verb, do not use continuous tense. Some use cases for the library application are shown in Figure 1.17.



Figure 1.17 Use Cases for the Library Application

### 3.2.2 The System

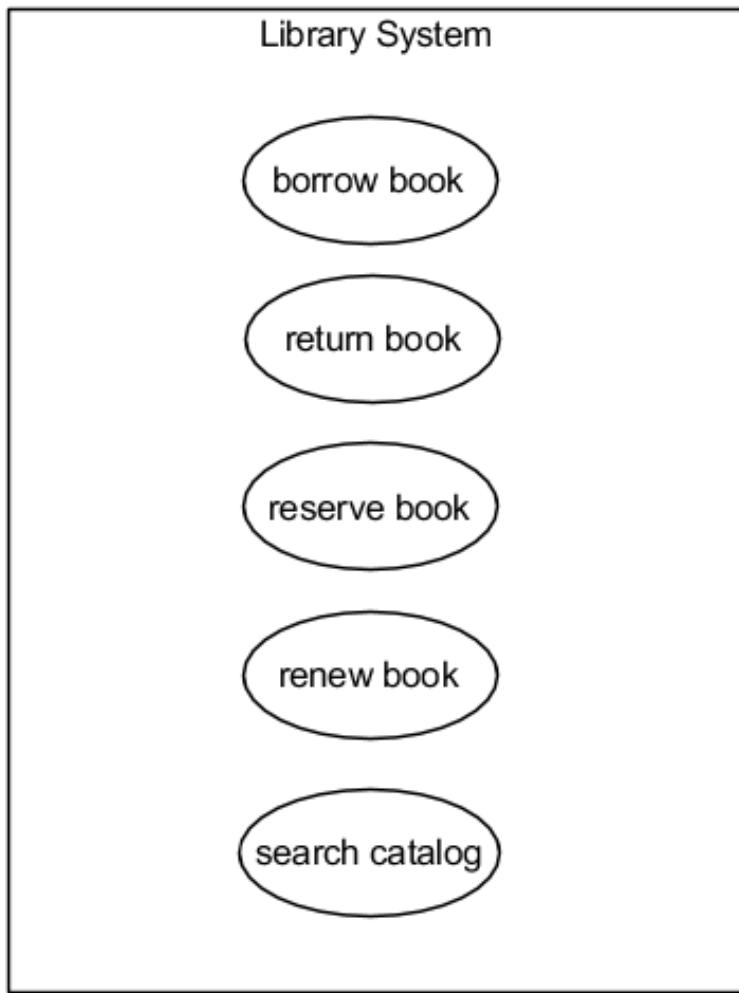
In a use case diagram, the system that you are designing is represented by a rectangle with the name of the system on top. Figure 1.18 shows the Library application in a use case

diagram with the use cases inside the rectangle. This defines the scope of the application to include only these use cases. Anything that is outside this rectangle does not happen in the Library System.

### 3.2.3 The Actor

A use case is always initiated by a user. In UML, this user is known as the actor. The actor directs, instructs, activates or requests the application to perform a function corresponding to a use case.

The use case is performed for the benefit of the actor. It represents an observable service or function that the application provides to the user.



**Figure 1.18** Library System with Use Cases for the Library Application

An actor refers to a user of the application that plays a specific role. The user:

- could be a human user or another computer system
- interacts directly with the application that is being analysed
- is external to the application
- needs something to be done by the application

We have emphasised the phrase "specific role" of a user in the above description. This is because an actor is not necessarily a single person or system. It could refer to different people performing a particular role.

The following helps to clarify this:

- An actor could represent different people that interact with the application in the same way.
  - For example, both library staff and members of the public may conduct a search for a book in the library catalogue. The actor would represent both the library staff and the public members.
- Different actors interact with the application in different ways.
  - In our library example, only library staff can update the library catalogue. A distinct actor would be needed to represent them in this role. Figure 1.20 shows this representation.

Note also the following about an actor:

- An actor can be involved with more than one use case. This is because a user can interact with an application by playing the same role with different objectives.
- A use case must involve at least one actor.
- An actor is not part of the application.
- An actor may either exchange information actively with the application or receive information passively from the application.

There are two types of actors:

- Primary Actor
  - A primary actor initiates a use case. For example, in the Library System, the User is going to initiate most of the use cases. We could also have a Librarian Actor who can update the catalogue.
- Secondary (or Supporting) Actors
  - A secondary actor reacts to a use case. If a User wishes to pay fines on his loans, he can initiate the 'pay fines' use case which will engage the Bank Actor to do the necessary transfers.

In a use case diagram, the Actor is represented by a stick man with a short descriptive name that is indicative of the role from the business perspective. It is a singular "noun" phrase starting with an uppercase letter. Figure 1.19 shows some actors. We would put Primary Actors on the left of the System rectangle and Secondary Actors on the right. In general, if the left-hand side is congested, Primary Actors may also be added on the right-hand side and vice versa. Figure 1.20 shows the Library System with Actors and use cases.

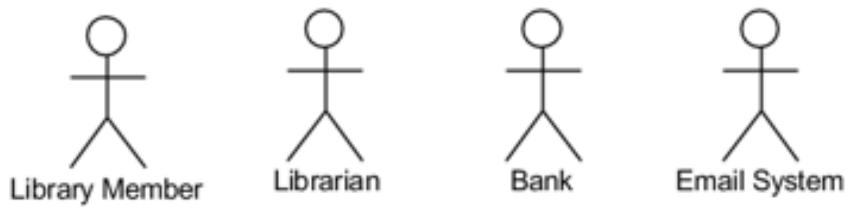


Figure 1.19 Examples of Actors

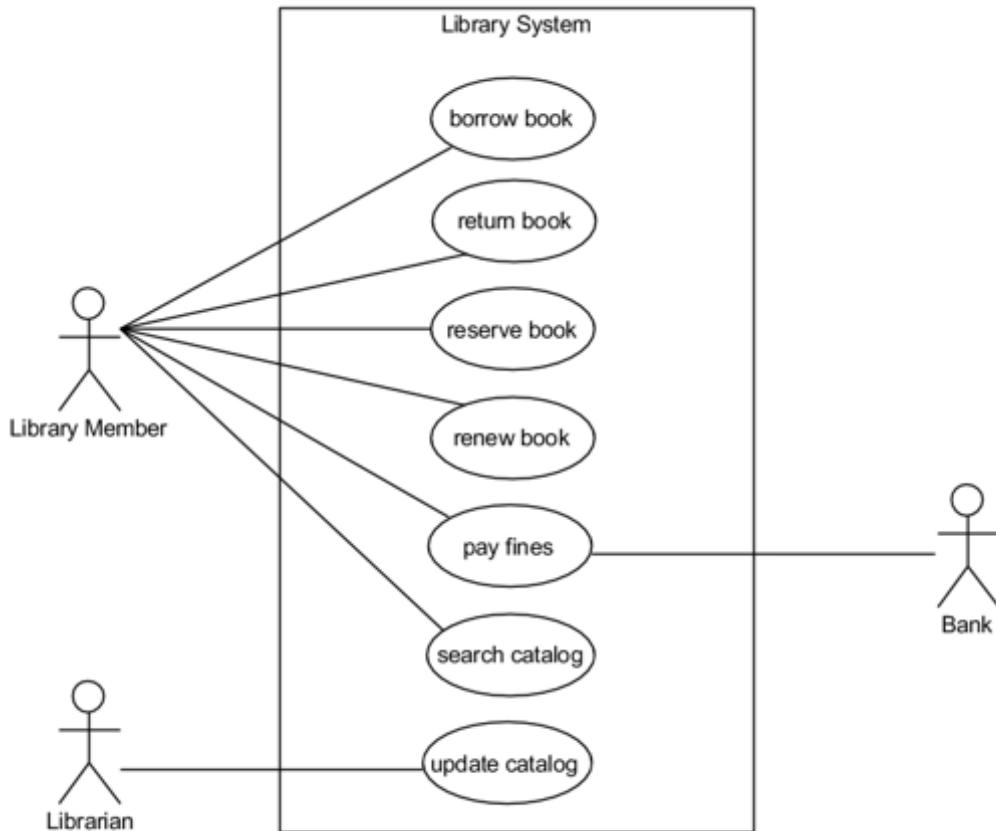


Figure 1.20 Library System with Use Cases and Actors

Figure 1.20 shows three types of actors. The Library Member can borrow book, return book, reserve book, renew book, pay fines and search catalogue. The Librarian can update catalogue. When the Library Member pay fines, the system would interact with the Bank.



## Watch

Simple and excellent tutorial for UML modelling.

<https://www.youtube.com/watch?v=zid-MVo7M-E&list=PLlxzWzkweYXy-CgIWL8Rp59832l-Imy>

### 3.2.4 Relationships in Use Case Diagrams

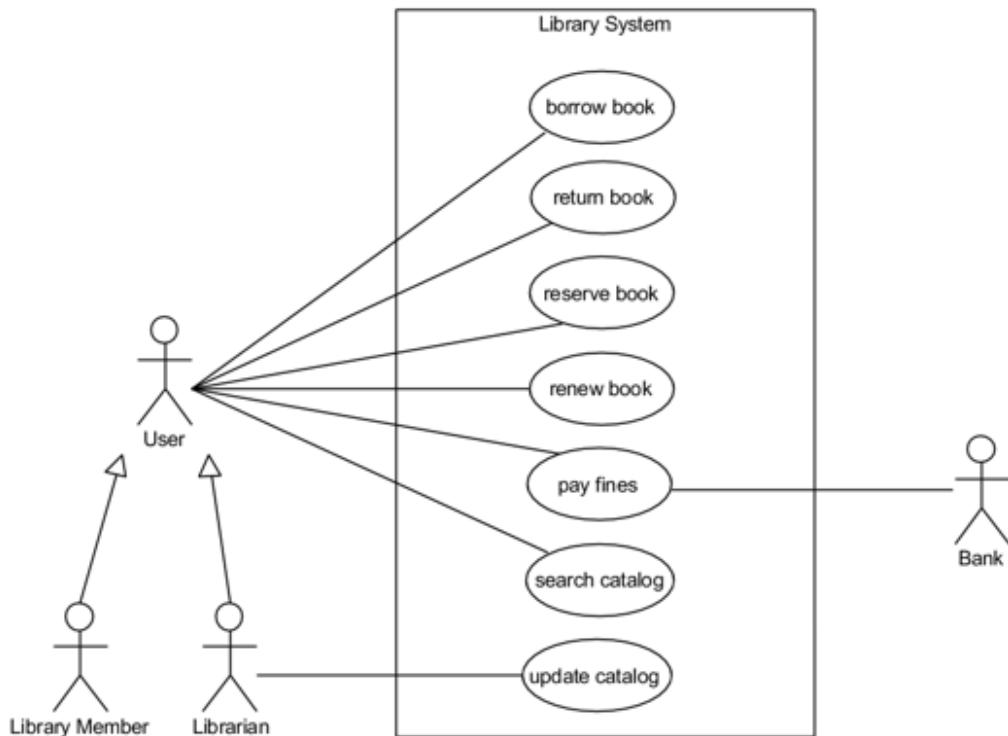
A use case diagram describes relationships between actors and use cases and among different use cases.

#### a. Association

An association is a relation between an actor and a use case. It is the only possible relationship between an actor and a use case. An association indicates that an actor interacts with the system for the specified use cases. Associations between actors and use cases are not directed and have no name. They are represented by solid lines between actors and use cases as shown in Figure 1.20.

#### b. Generalisation

Actors may be related by generalisation relations. The source of a generalisation is a special case of the target. This means that the target can always be replaced by the source but not vice versa. A generalisation relation is represented by a directed arrow with a solid line and an empty triangle as arrow head. Figure 1.21 shows an example. A User is a generalisation of a Librarian. This means that in any use case, a User can be replaced by a Librarian but not the other way around.



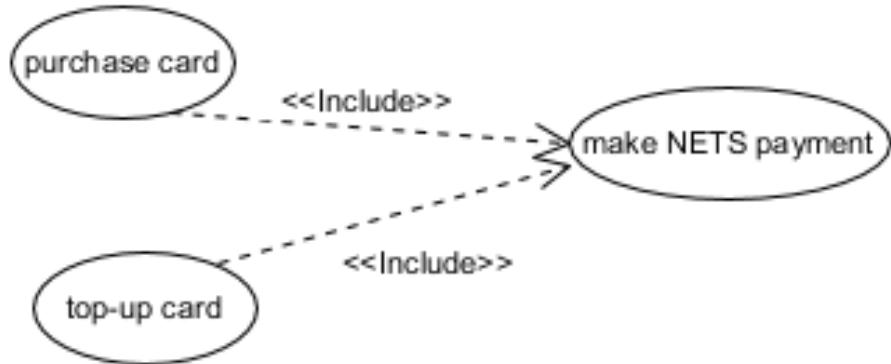
**Figure 1.21** Library System with Generalisation Relationships

### c. Dependencies: Include

A use case may depend on another use case to achieve its goal. An **include** dependency occurs when a use case always uses another one. **Include** dependency shows existing software components being re-used by the application one or more times.

For example, in the EZ-Link system used in Singapore's public transport, we can have use cases 'purchase card' and 'top-up card' that require the 'make NETS payment' to be added to complete their behaviour. The primary use cases are the 'purchase card' and 'top-up card' and the included use case is the 'make NETS payment'. You would not want to execute 'make NETS payment' by itself. This is only done when you do something like purchase card or top-up card. **Figure 1.22** shows the include relationship with a dashed arrow (with an open arrowhead) from the primary use case to the included use case. The arrow has the keyword

**<<Include>>** as its label. Take note that the arrow head is pointing towards the inclusion use case.

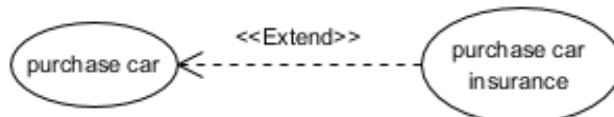


**Figure 1.22** Dependency include Relationships

#### d. **Dependencies: Extend**

A use case may also call another use case but only under special circumstances, for instance, as an alternative. There is a kind of optional behaviour between the primary use case and the extension use case. You can choose whether to execute the extension use case. Like the inclusion use case, extension use cases are not complete use cases and therefore cannot be triggered by an actor.

For example, the use case "purchase car" is meaningful on its own. An actor can purchase a car. It could be extended with optional "purchase car insurance" use case, i.e. when purchasing a car, there is an option to purchase car insurance. Figure 1.23 shows the extend relationship with a dashed arrow (with an open arrowhead) from the extension use case to the primary use case. The arrow has the keyword **<<Extend>>** as its label.



**Figure 1.23** Dependency extend Relationships

### 3.2.5 Analysing Requirements for Use Cases

Given the requirements, finding the use cases involves finding the actors and the tasks.

#### a. Finding Actors

Actors can be found by reviewing the requirements using the following questions:

- Where will the application be used?
- Who will provide the application with information?
- Who will use the information?
- Who will amend, update or remove the information?
- Who will be interested in a particular function or requirement?
- Who will support and maintain the function?
- Whose work will be improved or simplified by the functions in the application?

As an example to illustrate how the list of questions may reveal various actors, we will look at an application that disseminates financial data for a stock exchange. This application disseminates market data such as last traded price, selling price, offer price, trading volume and so on for the various shares traded in the exchange.

The application takes the market data from the share trading system of the stock exchange. It then distributes the data on its website. The data would also be sent to various price vendors such as Reuters, Bloomberg, Channel News Asia and so on. The data will be displayed in their monitors and the tickers running at the lower edge of television programmes. The data are also sent to the various stockbrokers who trade shares for their clients.

The following are examples of the answers that could be obtained for the questions. The answers are not intended to be comprehensive. There would certainly be more answers than those shown here. Our intention is to illustrate

the process of finding actors. It is not our objective to be thorough in our analysis here.

### Question

### Potential Actors

*Where will the application be used?*

Over the internet, this means any member of the general public can enter the website and look at the data.

*Who will provide the application with information?*

The trading system of the stock exchange will provide the market data.

*Who will use the information?*

The stockbroker who trades shares for his clients.

*Who will amend, update or remove the information?*

The exchange administrator will update any necessary information.

*Who will be interested in a particular function or requirement?*

Consider the requirement for system integration. The computer systems of external price vendors would want to receive market data from the application.

*Who will support and maintain the function?*

The technical personnel who maintains the networks and the systems.

*Whose work will be improved or simplified by the functions in the application?*

The market regulators in the exchange who monitor trading activities for any unusual events.

As a result of this exercise, the following could be the actors:

- Members of the general public
- The trading system of the stock exchange

- The stockbroker
- The exchange administrator
- The computer systems of external price vendors
- The technical personnel
- The market regulators in the exchange

b. **Finding Tasks**

Tasks or use cases can be found by reviewing the requirements using questions such as:

- What functions are required from the application?
- What are the tasks of the people involved with the application?
- What creation, storage, amendments, updating and removal of information will be required?
- What are the external changes that must be updated in the application?
- What are the internal changes in the application that its users must be kept informed?

After identifying potential use cases, check through the following questions to ensure that it is a good primary use case:

- Does it provide a single discrete benefit to an actor? Will the actor either gain some information or change the system in some way?
- Can the benefit gained from the use case be described in 30 words or less without the use of words such as "and" or "or"?
- Does the use case provide a significant, complete function with a value in itself?
- Is the actor usually able to achieve the task with a single session of interaction with the application?
- When an actor finishes with a use case, is it possible that he does not need the application any further?

Next, eliminate those that have the following characteristics:

- Is it a task that is not executed by the application itself? For example, is it a physical activity done by the user?
- Is it a task that an actor would never do by itself? In other words, is it a task that must be followed by other tasks in order for it to be completed? For example, logging onto a system is never done by itself but is always followed by another activity.

#### **Example 5:** Use cases of Mentcare system

The following are the use cases of the Mentcare system:

- Register patient
- View personal info
- View record
- Edit record
- Set up consultation
- Export statistics
- Generate report

The following are NOT use cases of the Mentcare system:

- Access performance against local and government targets – manual task
- Check blood glucose level, blood pressure – manual task
- Store data efficiently in centralised database – performance related, no interactions involved
- Patients usually go to nearest clinic – general comment about the system

#### **Example 6:** Use cases of the Weather Station system

- Report weather – send weather data to the weather information system
- Report status – send status information to the weather information system
- Restart – if the weather station is shut down, restart the system

- Shutdown – shut down the weather station
- Reconfigure – reconfigure the weather station software
- Powersave – put the weather station into power-saving mode
- Remote control – send control commands to any weather station subsystem

### 3.2.6 Documenting Use Cases – Use Case Description

The description of a use case provides an overall view of an application function. Each of the use cases should be described in structured natural language. It should generally include the items listed below. You might find variations in various books. The basic intention here is that it should document all the essential information concerning the use case.

- **Use case:** This is the name of a use case. It should be a short phrase beginning with a verb.
- **Initiator:** This is the actor who initiates the interaction.
- **Objective:** A brief explanation of the goal of the case.
- **Pre-conditions:** A brief description identifying any use cases or conditions that must be completed successfully before this use case can proceed.
- **Post-conditions:** The results when the use case is completed successfully. The focus here should be the value that the actor obtained from the interaction and the changes within the system.
- **Assumptions, notes or constraints:** This includes any non-functional requirements, restrictions, clarifications required and so on.
- **Normal scenario:** This covers the main purpose of the use case. It lists the steps of the interactions where everything proceeds smoothly according to plan. It is the main success scenario of the use case.
- **Alternative scenario:** This deals with any variations from the normal scenario due to the actor. The scenarios list separately the steps for each possible alternative action.

- **Exception scenario:** This deals with any variations from the normal scenario due to the system. Again, the scenarios list separately the steps for each possible alternative action.

### Example 7: Withdraw money from an ATM machine

Consider the use case Withdraw money for an ATM of a banking application. The following is a sample of the documentation. Note that to keep things manageable, we have used a somewhat simplified success scenario and omitted many possible alternative scenarios.

<b>Use case</b>	<i>Withdraw money</i>
<b>Initiator</b>	<i>Bank customer</i>
<b>Objective</b>	<i>To withdraw money from an ATM machine.</i>
<b>Pre-conditions</b>	<i>The bank card has been inserted into the ATM machine.</i>
<b>Post-conditions</b>	<i>The customer will have withdrawn money from his account. The balance of his account will be reduced by the amount withdrawn.</i>
<b>Assumptions, notes or constraints</b>	<i>The customer has an account with positive balance. The withdrawal amount is in multiples of \$10.</i>
<b>Clarification required</b>	<i>Is there an upper limit to the amount that can be withdrawn at any one time?</i>

#### Main scenario:

1. A Bank customer selects the option to withdraw money.
2. The application prompts for amount to withdraw.
3. The Bank customer enters a multiple of \$10 as the amount to be withdrawn.

4. The application checks that the balance in the bank account is sufficient.
5. The application checks that there is sufficient money in the machine.
6. The application prompts for confirmation of amount keyed in.
7. The Bank customer confirms the amount is correct.
8. The application returns the bank card, eject money and print the receipt.
9. The Bank customer collects the bank card, money and receipt.
10. The use case ends.

**Alternative scenario 1: The bank account balance is insufficient.**

- 4.1 The application displays there is insufficient balance in the bank account.
- 4.2 The application proceeds to step 2 for the Bank customer to re-enter amount.

**Alternative scenario 2: The money in the machine is insufficient.**

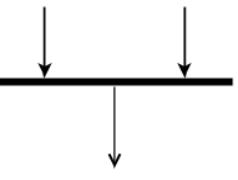
- 4.1 The application displays there is insufficient notes in the machine.
- 4.2 The use case ends.

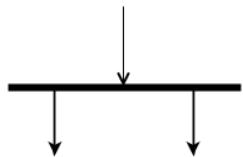
**Notes:**

- a. Each use case starts with the actor initiating an interaction with the application.
- b. When the application requires an input from the actor, it uses the word "prompt" and the next step would be the actor responding to the prompt.
- c. Numbering of the alternative scenario is in the format X.Y were X is the step in the main scenario that could branch to this alternative and Y is the sequence numbering of the alternative steps.
- d. An alternative scenario may result in an end to the use case or a return to the main scenario (creating a loop).

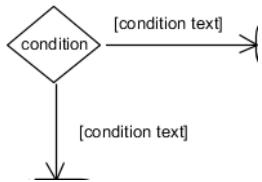
### 3.2.7 Activity Diagrams

Activity diagrams graphically describe the steps performed in a UML use cases. It shows how the activities are coordinated among the various parties involved. The symbols used in activity diagram and their meanings are listed below.

<u>Symbol</u>	<u>Name</u>	<u>Description</u>
	Start symbol	Represents the beginning of a process or workflow in an activity diagram. It can be used by itself or with a note symbol that explains the starting point.
	Activity symbol	Indicates the activities that make up a modelled process. These symbols, which include short descriptions within the shape, are the main building blocks of an activity diagram.
	Connector symbol (Transition)	Shows the directional flow, or control flow, of the activity. An incoming arrow starts a step of an activity; once the step is completed, the flow continues with the outgoing arrow.
	Joint Symbol / Synchronisation Bar	Combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a

**Fork Symbol**

time. Represented with a thick vertical or horizontal line.

**Decision Symbol &  
Condition Text**

Splits a single activity flow into two concurrent activities. Symbolised with multiple arrowed lines from a join.

Represents a decision and always has at least two paths branching out with condition text to allow users to view options. This symbol represents the branching or merging of various flows with the symbol acting as a frame or container.

Condition text is placed next to a decision marker to let you know under what condition an activity flow should split off in that direction.

**End Symbol**

Marks the end state of an activity and represents the completion of all flows of a process. It can be used by

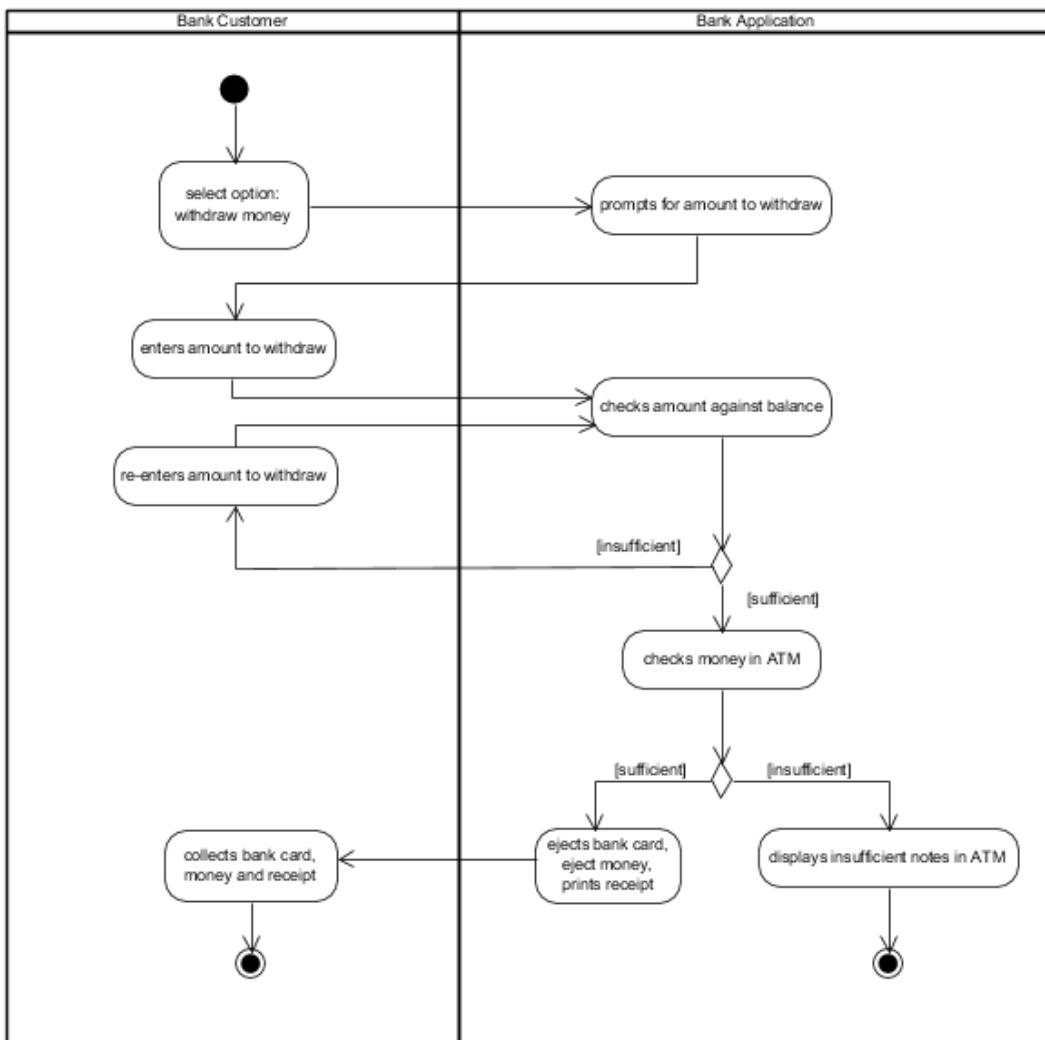
itself or with a note symbol  
that explains the end point.

### Example 8: Withdraw money from an ATM machine

Figure 1.24 shows the activity diagram for the Withdraw money use case documented in the previous section.

Note the following about the diagram:

- The activity is divided into two columns or "swim lanes": the Bank Customer and the Bank Application. These correspond to the two parties involved in the use case. Each of the activity boxes is placed in the swim lane according to the party executing the action.
- The start symbol should be located at the swim lane for the actor who initiates the use case.
- Each activity box corresponds to one step of the action in the scenarios. You should cross-reference the diagram with the main scenario and alternative scenarios for the use case and see this for yourself.
- There may be one or more end symbols in an activity diagram but there can only be one start symbol.



**Figure 1.24** Activity Diagram for Withdraw Money

Figure 1.25 shows a fragment of the activity diagram and a portion of the main scenario of the use case. Note how the activity box corresponds to each step in the main scenario.

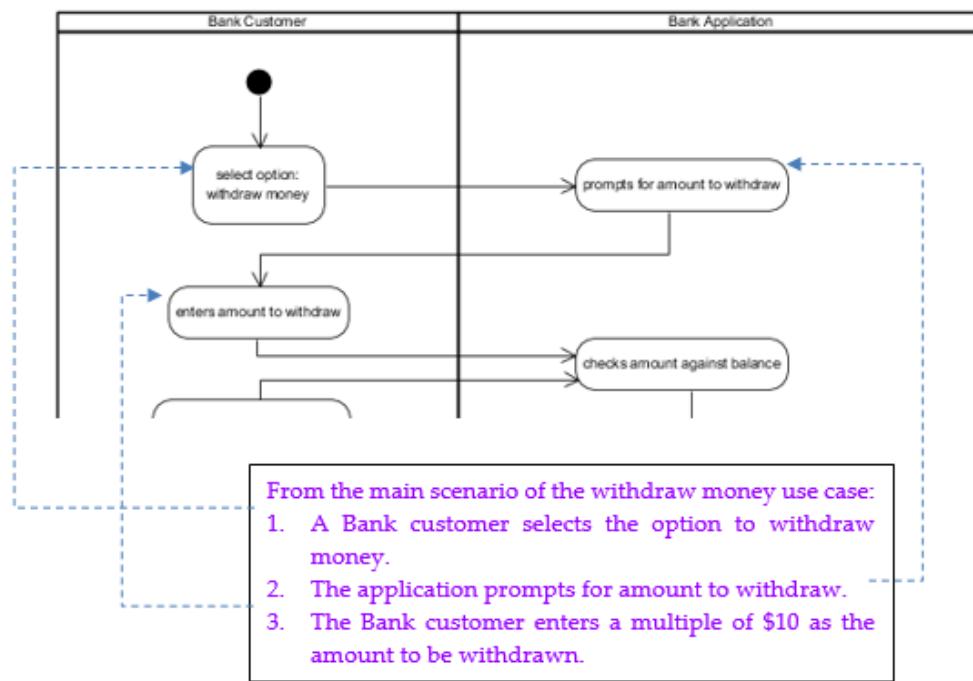


Figure 1.25 Activity Diagram showing Correspondence to Use Case Description

## Chapter 4: Requirements Specification Document

Requirements specification is the process of writing down the user and system requirements in a requirements document. User requirements have to be understandable by end-users and customers who do not have a technical background. System requirements are more detailed requirements and may include more technical information. The requirements may be part of a contract for the system development. It is therefore important that these are as complete as possible.

### 4.1 System Documentation

The software requirements document is the official statement of what is required of the system developers. It should include both a definition of user requirements and a specification of the system requirements. It is NOT a design document. As far as possible, it should be a set of WHAT the system should do rather than HOW it should do it.

Requirements specification are written in the following forms:

- a. As numbered sentences in natural language supplemented by diagrams and tables. It is expressive, intuitive and universal for everyone to understand. However, it is hard to achieve precision and may lead to confusions by mixing up functional and non-functional requirements.
- b. Using a standard form or template in natural language. Each field provides information about an aspect of the requirement. See example in Table 1.1 for the Automatic Patient Monitoring System.
- c. Using graphical notations such as the Unified Modelling Language (UML). UML use cases and scenarios are popular graphical form for requirements specification.

**Table 1.1** Standard Template/Form

<b>Function</b>	Detect abnormalities in the patient's vital signs
<b>Description</b>	Compute whether the current vital signs reading falls out of range and generates alert message
<b>Inputs</b>	Current vital signs reading, safe range values, sensors data on exercising state
<b>Source</b>	Current vital signs reading from the device, safe range values from memory, sensors data on exercising state
<b>From Outputs</b>	Abnormality status of vital signs
<b>Destination</b>	Messaging module



## Activity 1.6

Document the requirement REQ4 in Example 2 Automatic Patient Monitoring System above using the standard template given in Table 1.1.



## Read

Mentcare requirements document:

<https://www.dropbox.com/s/61a7hg7tbw4p0nx/Mentcare%20requirements%20document.pdf?dl=0>

## Summary

The following points summarise the contents of this study unit:

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- **Functional requirements** are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- **Non-functional requirements** often constrain the system being developed and the development process being used.
- They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.
- Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organisation, requirements negotiation and requirements documentation.
- You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.
- Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- The software requirements document is an agreed statement of the system requirements. It should be organised so that both system customers and software developers can use it.
- Requirements validation is the process of checking the requirements for correctness, clarity, consistency, completeness, realism and verifiability.
- Business, organisational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

- Use cases are a powerful graphical tool for modelling and documenting the system requirements. Use cases are part of the UML specification. The use case represents the goal that the user wants to achieve with the application.
- Activity diagrams enhance each use case by graphically showing the steps/workflow involved in the use case.

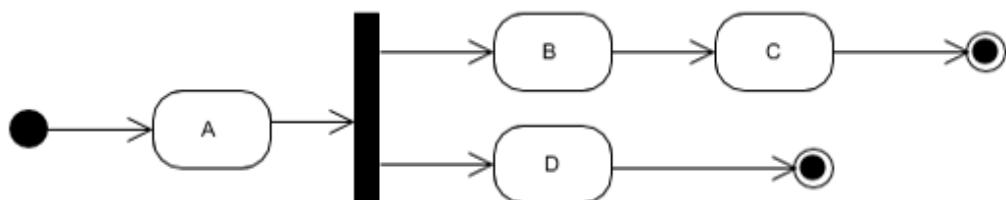
## Formative Assessment

1. Which is the most critical phase of the System Development Life Cycle for a project team?
  - a. Requirements Engineering
  - b. System Design
  - c. System Implementation
  - d. System Operation and Maintenance
2. When analysing requirements, what do you typically do with the information elicited?
  - a. Take detailed notes of what everyone said and convey a compiled summary to the developer.
  - b. Break it down, sum it up, look for gaps, model it, diagram and figure out what more to elicit.
  - c. Create a detailed plan of the tasks needed to get the requested work done.
  - d. Turn over the information to the project manager.
3. What type of requirement is this?

The system shall provide appropriate viewers for the user to read documents in the document store.

  - a. Functional Requirement
  - b. Non-Functional Requirement
  - c. Design Directive
  - d. Implementation Directive
4. Which one of the following is NOT a benefit of using ethnography to elicit information?

- a. Enables you to learn what the customer is thinking about while using the product.
  - b. Allows you to understand the user's emotions and feelings when performing certain functions.
  - c. Helps you understand how the user interacts with the product or system.
  - d. Helps you understand the broader domain requirements of the application.
5. Which question does a user story NOT answer?
- a. How
  - b. Who
  - c. What
  - d. Why
6. What is the purpose of a use case diagram?
- a. To show the features of the system.
  - b. To show the functionality of the system.
  - c. To show how the users interact with the system.
  - d. To show what is approved for the system.
7. Given the following activity diagram, which one of the following sequences is NOT possible during its execution?



- a. A → D
- b. A → B → C
- c. A → B → D

d. A → B → D → C

# Solutions or Suggested Answers

## Activity 1.1

Software engineering is about designing and developing software. System engineering also includes physical components such as hardware.

Software engineering is computer-based engineering while system engineering covers a broader education area and includes Engineering, Mathematics and Computer Science.

## Activity 1.2

The majority of the costs involved in the life cycle of a product since its inception until it is decommissioned are in changing the software after it has been put to use. Using software engineering methods can facilitate and reduce the cost of software maintenance.

## Activity 1.3

1. Anti-lock braking system: This is a safety-critical system and so it requires a lot of up-front analysis before implementation. It certainly needs a plan-driven approach to development with the requirements carefully analysed. A waterfall model is therefore the most appropriate approach to use, perhaps with formal transformations between the different development stages.
2. Virtual reality system: This is a system where the requirements will change and there will be an extensive user interface components. Incremental development with, perhaps, some UI prototyping is the most appropriate model. An agile process may be used.

3. University accounting system: This is a system whose requirements are fairly well-known and which will be used in an environment in conjunction with lots of other systems such as a research grant management system. Therefore, a reuse-based approach is likely to be appropriate for this.
4. Interactive travel planning system: System with a complex user interface but which must be stable and reliable. An incremental development approach is the most appropriate as the system requirements will change as real user experience with the system is gained.

## **Activity 1.4**

The long-term goal of software measurement is to use measurement to make judgements about software quality. A range of metrics can be used to measure a system's attributes.

Examples include (source: <https://stackify.com/track-software-metrics/>)

- Lead time (how long it takes for ideas to be developed and delivered as software)
- Cycle time (how long it takes to change the software system and implement that change in production)
- Code churn (the no. of lines of code that were modified, added or deleted in a specified period of time)
- Mean time between failures (MTBF) (how well software recovers and preserves data)

CMMI (Capability Maturity Model Integration) is a framework for process improvement and can be used to guide improvement across a project team or entire organisation.

The 5 Levels are:

Level 1 – Initial (No process area)

Level 2 – Managed or Repeatable (Disciplined process)

Level 3 – Defined (Standard, consistent process)

Level 4 – Quantitatively Managed (Predictable process)

Level 5 – Optimising (Continuously improving process)

## **Activity 1.5**

The following are the functional requirements:

- The application should monitor the temperature of the air at ten different locations of the plant.
- The application should record the temperature readings at fifteen-minute intervals to a database.
- The application should also raise an alarm if any of the temperature readings cross the permissible limits.
- The application should also measure the pressure drop across the filters at three locations.
- The application should send a signal to the maintenance personnel if the pressure drop exceeds the pre-set limits.
- The application should upload the temperature data to the server at the end of every working day.

The following are the non-functional requirements:

- The application should run on a PC-based system.
- The application should not require more than 512 MB of memory.

## **Activity 1.6**

<b>Function</b>	Detect when the patient is exercising
<b>Description</b>	Adjusts the safe ranges of vitals

<b>Inputs</b>	Current vital signs reading, safe range values, sensors data on exercising state
<b>Source</b>	Current vital signs reading from the device, safe range values from memory, sensors data on exercising state
<b>From Outputs</b>	Exercising status of vital signs
<b>Destination</b>	Messaging module

## Formative Assessment

1. Which is the most critical phase of the System Development Life Cycle for a project team?
  - a. Requirements Engineering
 

**Correct. This is because if done incorrectly, the end product might not be what the customer wants and leads to project failure. Refer to Study Unit 1, Section 1.4.**
  - b. System Design
 

Incorrect. If system design is not done correctly, it is 3 to 6 times more expensive to fix than if discovered in the requirements engineering phase. Refer to Study Unit 1, Section 1.4.
  - c. System Implementation
 

Incorrect. Fixing a problem during implementation may be easy, but it may not fix the problem of incorrect requirements in the requirements engineering phase. Refer to Study Unit 1, Section 1.4.
  - d. System Operation and Maintenance

Incorrect. Finding and fixing a problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase. Refer to Study Unit 1, Section 1.4. (Source: <https://betterembsw.blogspot.com/2016/08/boehms-top-10-software-defect-reduction.html>).

2. When analysing requirements, what do you typically do with the information elicited?

- a. Take detailed notes of what everyone said and convey a compiled summary to the developer.

Incorrect. This is only one of the activities that need to be done. The information collected needs to be classified and organised too. Refer to Study Unit 1, Section 2.3.1.

- b. Break it down, sum it up, look for gaps, model it, diagram and figure out what more to elicit.

**Correct. Requirements elicitation and analysis is an iterative process. Refer to Study Unit 1, Section 2.3.1.**

- c. Create a detailed plan of the tasks needed to get the requested work done.

Incorrect. The work is not done yet. Information elicited needs to be classified and organised, prioritised and negotiated with the stakeholder. Refer to Study Unit 1, Section 2.3.1.

- d. Turn over the information to the project manager.

Incorrect. The work is not done. There is a need to do requirements analysis with the information elicited. Refer to Study Unit 1, Section 2.3.1.

3. What type of requirement is this?

The system shall provide appropriate viewers for the user to read documents in the document store.

a. Functional Requirement

Incorrect. This does not describe what the system would do. Refer to Study Unit 1, Sections 2.2 and 2.3.

b. Non-Functional Requirement

**Correct. This describes the usability of the system. Refer to Study Unit 1, Sections 2.2 and 2.3.**

c. Design Directive

Incorrect. This does not describe how the system is to be designed. Refer to Study Unit 1, Sections 2.2 and 2.3.

d. Implementation Directive

Incorrect. This has nothing to do with how the system is to be implemented. Refer to Study Unit 1, Sections 2.2 and 2.3.

4. Which one of the following is NOT a benefit of using ethnography to elicit information?

a. Enables you to learn what the customer is thinking about while using the product.

Incorrect. This is a benefit as the ethnographer can ask questions about why the customer does those steps. Refer to Study Unit 1, Section 2.3.2.

b. Allows you to understand the user's emotions and feelings when performing certain functions.

Incorrect. This is a benefit as the ethnographer can see the user's reaction and hence understand whether the function is easy or difficult to execute. Refer to Study Unit 1, Section 2.3.2.

c. Helps you understand how the user interacts with the product or system.

Incorrect. This is a benefit as it allows the ethnographer to understand the way the product or system is used and these could be inputs to development work to simplify the steps for the user. Refer to Study Unit 1, Section 2.3.2.

- d. Helps you understand the broader domain requirements of the application.  
**Correct. Because you are only observing the current work done, it does not give insights as to what else is possible. Refer to Study Unit 1, Section 2.3.2.**

5. Which question does a user story NOT answer?

- a. How

**Correct. A user story does not explain HOW the task or process is to be done. Refer to Study Unit 1, Section 2.3.2.**

- b. Who

Incorrect. A user story states WHO is involved in the particular task or process. Refer to Study Unit 1, Section 2.3.2.

- c. What

Incorrect. A user story describes WHAT people do, WHAT information they use and produce, and WHAT systems they may use in this task or process. Refer to Study Unit 1, Section 2.3.2.

- d. Why

Incorrect. A user story gives the reason WHY the task or process is needed. Refer to Study Unit 1, Section 2.3.2.

6. What is the purpose of a use case diagram?

- a. To show the features of the system.

Incorrect. Features of a system is documented in the functional and non-functional requirements. Refer to Study Unit 1, Section 3.2.4.

- b. To show the functionality of the system.

**Correct. A use case diagram shows the functionalities of the system and its interaction with the environment. Refer to Study Unit 1, Section 3.2.4.**

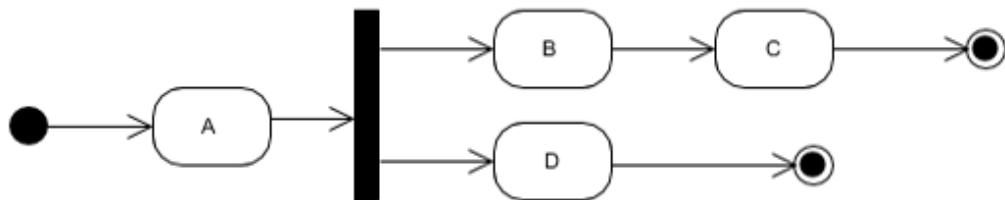
- c. To show how the users interact with the system.

Incorrect. A use case diagram shows the interactions between the system and its environment, not just the users. Refer to Study Unit 1, Section 3.2.4.

- d. To show what is approved for the system.

Incorrect. Approval for a system is obtained before the project starts. Refer to Study Unit 1, Section 3.2.4.

7. Given the following activity diagram, which one of the following sequences is NOT possible during its execution?



- a.  $A \rightarrow D$

Incorrect. This sequence is correct. B and C may not get a chance to execute once the system completes execution of D. Refer to Study Unit 1, Section 3.2.7.

- b.  $A \rightarrow B \rightarrow C$

Incorrect. This sequence is correct. D may not get a chance to execute once the system completes execution of C. Refer to Study Unit 1, Section 3.2.7.

- c.  $A \rightarrow B \rightarrow D$

Incorrect. This sequence is correct. C may not get a chance to execute once the system completes execution of D. Refer to Study Unit 1, Section 3.2.7.

- d.  $A \rightarrow B \rightarrow D \rightarrow C$

**Correct. This sequence is incorrect. After the execution of D, the final node is reached which terminates all activities within the diagram. Refer to Study Unit 1, Section 3.2.7.**

## References

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.

Brambilla, M., Cabot, J., & Wimmer, M. (2012). *Model-driven software engineering in practice*. Morgan Claypool.



# Study Unit

# 2

Structural Object-Oriented  
Modelling

## Learning Outcomes

By the end of this unit, you should be able to:

1. state the outcome or deliverable from performing a structural analysis
2. explain the ultimate objective of having a class in an application
3. analyse the text of an application requirement to identify possible classes, attributes, relationships and multiplicities
4. state the reasons for rejecting something as a class
5. discuss the considerations in deciding whether something should be a class or an attribute
6. prepare the class description from a set of application requirements
7. prepare class association diagrams

## Overview

In this unit, we will discuss how to develop the structural model based on a given set of requirements. Recall that a set of requirements is a brief description of what a new system will do as agreed among all the parties involved with the system. Structural or static model defines the system components (parts) and their arrangements that make it possible for the system to meet the requirements and use cases. We will do this by looking for the following from the requirements:

1. classes
2. attributes
3. generalisation relationships
4. associations
5. multiplicities of associations



### Read

The study guide developed based on Part 1 (Chapters 1 – 9) and Part 4 (Chapters 22 – 25) of the textbook:

Sommerville, I. (2016). *Software Engineering, 10th Edition.*

# Chapter 1: Analysing Requirements for Classes



## Lesson Recording

### Structural Analysis

The first activity in structural analysis is to conduct a textual analysis of the statement of requirements. The requirements are analysed to look for the following:

1. classes
2. attributes
3. generalisation relationships
4. associations
5. multiplicities of associations

The result or deliverable from analysing the requirements is a structural model of the application. This consists of:

1. The class description
2. The class association diagram

## 1.1 Identifying Classes

Identifying classes can be more like an art rather than a science. We may provide some guidelines here. However, these are not precise rules that will give you definite answers on what would be classes in the final design. Having a clear idea of the purpose for having a class and doing some exercises will help you in getting the necessary skills.

To identify classes, we do a textual analysis on the requirements. We go over the text of the requirements to look for nouns or noun phrases. These would give you good potential candidates for classes in the application.

Typically, potential classes fall into one of the following categories:

a. **Tangible Things**

These are the physical things that are relevant to the application. For example, in a library system, there would be tangible things such as books, magazines, video discs and so on. These are things which the system needs to maintain information about for the purpose of loans and stock control.

b. **Intangible Things**

These refer to non-physical things that are relevant to the application. For example, an educational system would have intangible things such as course modules, enrolments, degree programmes and so on. Information about these would be necessary for the system to administer student registrations and enrolments for courses.

c. **Events**

An event could be a happening, an incident, an interaction, a situation and so on. For example, a sports association could organise various events such as swimming, athletics, out-door games and so on. A system that provides administrative support to the association would be required to maintain information about the events.

d. **Roles**

These refer to the roles of the people that are relevant to the application. For example, in an educational system, there would be roles such as students, tutors, lecturers and so on. Information about these would be required in the system for its operations.

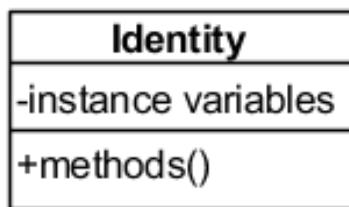
e. **Organisational Units**

An organisational unit can be a team, a tutorial group, a department, a branch, a hospital ward and so on. For example, an educational system would be required to maintain information about tutorial groups to support the operations of a university.

The final decision on whether something should be a class in an application is based on the concept of a class. A class is in an application for two primary objectives:

- To maintain information about itself, through its attributes;
- To provide services for the application, through its methods.

Figure 2.1 below shows the structure of a class with these two objectives:



**Figure 2.1** The primary objectives of a class in an application

Remember that the above guidelines, using tangible things, intangible things, events, roles and organisation units, will only give you potential candidates for classes. It does not imply that every tangible thing, for example, should have a corresponding class in the application.

Therefore, you should not say that you have selected something to be a class just because it is a tangible thing. You would still need to study the requirements to see whether the application has to maintain any information about that tangible thing.

Watch

UML Class Diagram Tutorial:

<https://www.youtube.com/watch?v=UI6lqHOVHic>



## Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*, Chapter 5, Section 5.3, pp.149-153.

We will now go through the process of finding candidate classes through the examples described in Study Unit 1, Section 1.7.1: ABC Car Rentals.

### 1.1.1 Textual Analysis

Textual analysis is to go through the paragraphs of the requirements and highlight all the nouns and noun phrases. Repetitions of the same noun are left out. If both the singular and plural forms are present, the singular form will be selected. See the identified noun and noun phrases underlined in the following paragraphs 1 to 6:

1. The ABC Car Rentals has several branches. Each branch has its own collection of rental cars and is uniquely identified by its location.
2. The company employs managers and branch workers that work in the branches. Each branch has one manager who is assisted by a few branch workers. A manager is responsible for only the branch assigned to him or her.
3. Not all employees of the company are branch workers or managers. For example, there are quality controllers who monitor the efficiency of all the branches.
4. Not every branch has cars with it. This is because some branches may be undergoing renovation or a new branch may not have cars delivered to it yet.
5. Customers sign a contract to rent a car. No customer may sign more than one contract. However, some customers who are business entities may wish to sign a contract for more than one car. For efficiency reasons, these cars must come from the same branch.

6. At the end of a contract, a customer may return a car to any branch of the company. However, the station workers will send the car back to the branch that is responsible for that car.
7. The **ABC Car Rental System (ABCCRS)** will support the company's operations by providing the following reports through the **Internet**:
  - a. **Information** about each branch, including its location, **status** (operational or under renovation) and the manager in charge.
  - b. Given a branch location, information about the company's employees, including their names and **employee numbers** and, for branch workers, their **hourly overtime rate**.
  - c. Given a branch location, information about the company's cars, including the **engine capacity** and the **vehicle registration number**.
  - d. Information about customers, including their names and **addresses**.
  - e. Information about each current contract, including the **start date**, **duration** of the contract and the customer name.

The following are the nouns highlighted. The singular version of the noun is used.

ABC Car Rentals	branch	car	location
Company	branch worker	manager	employee
Quality	efficiency	renovation	customer
Contract	business entity	ABCCRS	report
Internet	information	status	employee
name	hourly overtime rate	address	start date
duration	engine capacity	vehicle registration number	

Again, this seems to be a rather long list for considerations just for one page of requirements. After working on a few exercises, you will find that you can quickly leave out those that are obviously not likely to be classes.

### 1.1.2 Finding Classes

The next stage is to identify all the nouns that could correspond to a class and eliminate all those that are unlikely to be classes.

A good way to know that something should be a class is when the application has to maintain information about it. Paragraph 7 of the requirements has helpfully provided details of these. Table 2.1 shows the main item of interest inferred by each subparagraph.

**Table 2.1 Requirements and Classes Inferred**

Requirements	Classes inferred
a. <i>Information about each branch, including its location, status (operational or under renovation) and the manager in charge.</i>	Branch
b. Given a branch location, <i>information about the company's employees, including their names and employee numbers and, for branch workers, their hourly overtime rate.</i>	Employee Branch Worker Manager, by implication
c. Given a branch location, <i>information about the company's cars, including the engine capacity and the vehicle registration number.</i>	Car
d. <i>Information about customers, including their names and addresses.</i>	Customer

e. <i>Information about each current contract, including the start date, duration of the contract and the customer name.</i>	Contract
--	----------

We would also remove the following from the list of nouns:

<u>Noun</u>	<u>Reason for Removal</u>
ABC Car Rentals	The proposed application is concerned with the rental of cars. This noun is the name of the company.
location	The application need not maintain any information about locations other than its value. This is more like an attribute of a branch.
company	This refers to the company itself.
quality controller	No information has to be maintained about quality controllers. We can verify this with the users.
efficiency	This is a general descriptive term elaborating on the job of some staff.
renovation	The application need not maintain any information about renovations.
business entity	This is an alternative term for customers.
ABCCRS	This refers to the application being designed and developed.
information	This is a general descriptive term. The application need not maintain information about information.

<b>status</b>	The application need not maintain any information about the status of a branch other than its value. This is more like an attribute of a branch.
<b>name, employee number</b>	The application need not maintain any information about names and employee numbers other than their values. These are more like attributes of employees.
<b>hourly overtime rate</b>	For the same reason, this is an attribute of a branch worker.
<b>engine capacity, vehicle registration number</b>	These are attributes of a vehicle.
<b>address</b>	This is an attribute of a customer.
<b>start date, duration</b>	These are attributes of a contract.
<b>Internet</b>	The application need not maintain information about the Internet. The Internet is mentioned as a means to access the information.
<b>report</b>	This pertains more to a function that the application should provide.

**As a summary, these nouns are eliminated for the following categories of reasons:**

1. **They are outside the scope of the system.** This eliminates nouns such as ABC Car Rentals, company and renovation.
2. **They are general descriptive terms.** This eliminates nouns such as efficiency and information.
3. **They are alternative words for another term.** This removes business entity since the term customer includes business entity. There is also no apparent need to

distinguish between business entity and other type of customers in the given requirements.

4. They are attributes of something else. This takes away nouns such as name, address, engine capacity, duration and so on.
  5. They refer to actions, functions or behaviour that the application should provide. This means that terms such as report should not be a class.
  6. They have to do with the user interface. The user interface is considered separately to keep the domain model as independent of the interface as possible. This removes the term Internet.
  7. It is a term that refers to the system being developed. This eliminates ABCCRS.

Again, while applying these guidelines, the answer is not always clear and unambiguous. It is possible for one guideline to suggest that a noun need not be a class while another might indicate that it should be retained as a class. The final decision might be clearer when the dynamic analysis is done.

The following classes are identified:

<b>Employee</b>	a role
<b>Manager</b>	a role
<b>BranchWorker</b>	a role
<b>Branch</b>	an organisational unit
<b>Car</b>	a tangible thing
<b>Customer</b>	a role
<b>Contract</b>	an intangible thing (or

The class Contract may also be regarded as an event since it has a start time and duration.



## Activity 2.1

### A chain of department stores

The GO Company has several department stores, each with four to eight departments. The following is a description of the requirements of a system for a proposed application that will support its operations:

The company's staff consists of store managers, department supervisors and salespersons. The responsibilities of these staff members are:

- A store manager is the staff in charge of an entire store. Each store has only one store manager.
- A department supervisor looks after the daily operations of one or more departments of a store. Each department has only one department supervisor.
- A salesperson works in a department. Every department has at least two salespersons.

Every quarter, a sales performance review exercise is carried out for each department. Each such exercise is chaired by the store manager of that store and is carried out for all the departments at the same time. All department supervisors of that store would attend the review.

The system supports the company's operations by performing the following:

- Recording information about the staff in each store, including their names, telephone numbers and educational qualifications. For a salesperson, the commission earned should be included. For a store supervisor, the supervisor grade should also be included.
- Recording information about each store, including its location and its size.
- Recording information about each department of a store, including details such as its size, sales turnover and quarterly profits.

- Providing information such as the date and the persons present for each quarterly review.

From these requirements, determine the classes that would be required for the application. Note again that the requirements are not comprehensive to keep the question manageable.

## 1.2 Identifying Attributes

**After identifying the potential classes, we go through the text again to find anything that the application needs to know about them. For a well-written set of requirements, most of these attributes are easy to find.** However, this is not necessarily always straightforward, as we will see later.

We will be using the two application examples that have been given in the previous section to identify attributes.

**Table 2.2** Analysing Requirements for Attributes

<u>Requirements (selected paragraphs only)</u>	<u>Attribute</u>
1. The ABC Car Rentals has several branches. Each branch has its own collection of rental cars and is uniquely identified by its location.	<i>location</i> is an attribute of Branch with a unique value.
7. The ABC Car Rental System (ABCCRS) will support the company's operations by providing the following reports through the Internet:	
a. Information about each branch, including its <u>location</u> , <u>status</u> (operational or under renovation) and the manager in charge.	<i>location</i> and <i>status</i> are attributes of Branch.

b. Given a branch location, information about the company's employees, including their <u>names</u> and <u>employee numbers</u> and, for branch workers, their <u>hourly overtime rate</u> .	<i>name</i> and <i>employeenumber</i> are attributes of Employee.
c. Given a branch location, information about the company's cars, including the <u>engine capacity</u> and the <u>vehicle registration number</u> .	<i>engine capacity</i> and <i>vehicle registration number</i> are attributes of Car.
d. Information about customers, including their <u>names</u> and <u>addresses</u> .	<i>name</i> and <i>address</i> are attributes of Customer.
e. Information about each current contract, including the <u>start date</u> , <u>duration of the contract</u> and the customer name.	<i>start date</i> and <i>duration</i> are attributes of Contract.

Note the following in Table 2.2:

- a. In paragraph 7(a), the attribute status is given additional comments in brackets (operational or under renovation). These are the possible values of the attribute status. The values are mutually exclusive. Thus, it is not necessary to have two separate attributes such as isOperational and isUnderRenovation.
- b. Paragraph 7(a) says:

Information about each branch, including its location, status (operational or under renovation) and the manager in charge.

However, our list of attributes has only location and status. The manager is recognised previously as a distinct class in the application. Some course texts advocate that manager should be recognised as an attribute of branch only when we perform the dynamic analysis. At this stage, our focus is on the structural or static analysis. Therefore, we should leave out any attribute for which we have identified to be a class in the application.

Another thing to notice is that paragraphs 7(a) to (e) are information required for reports that the application has to provide. For paragraph 7(a), the report would probably require the name of the manager in charge although the requirements merely mention "the manager in charge".

- c. Paragraph 7(e) says:

Information about each current contract, including the start date, duration of the contract and the customer name.

The list of attributes for Contract should have only the start date and duration. The report may require the name of the customer pertaining to a contract. But customer name is not an attribute of Contract. Customer name is an attribute of the class Customer.

Then the question is how could the report obtain the customer names for the contracts? Again, the actual answer to this question will be given when we perform dynamic analysis. In the meantime, when we draw the class association diagram, we may note that the class Contract should be associated with the class Customer.

### 1.2.1 Thing versus Property

A noun can be selected to be a class or an attribute. Consider the nouns, student and name, for a student registration application in an educational institution:

- Student is a noun that is likely to be a class we need to implement. We would expect that the application would need to maintain information about students.
- Name is a noun that is unlikely to be a class we need to implement. It would simply be a string of characters and it is likely to be an attribute in a class such as Student.

Therefore, the decision that you often have to make is when a noun is a class and when it is an attribute. This is the question of whether something should be a thing or a property:

- If it is a thing, then it would be an object and you would need to implement a class for it.

- If it is a property, then you would need to identify an appropriate class and define an instance variable for it.

Again, the guideline is whether the application has to maintain some information about it. If it does, then it is likely that a class is needed.

### Example 1: Cars

Consider a car in two different applications: an application for a car rental company and an application for an educational institution.

We would normally expect the application for the car rental company to have a class for car. This is because, as part of its operations, it would need the system to maintain information about a car. These could include the vehicle registration number, the model, the year of manufacture, the purchase price (for calculating depreciation against taxes on revenue) and so on.

For the application in the educational institution, we might also be interested in cars. For example, the institution might issue carpark labels for lecturers so that they can park in the staff carpark. However, our interest is restricted only to the registration number of the car and nothing else. Thus, an attribute alone in the class Lecturer would contain enough information about a car as far as this application is concerned.

Thus, a car is a "thing" in the car rental application but a "property" in the educational application. Figure 2.2 helps to show the difference.

In summary, Car is a class in the car rental application because the application has to maintain information about cars.

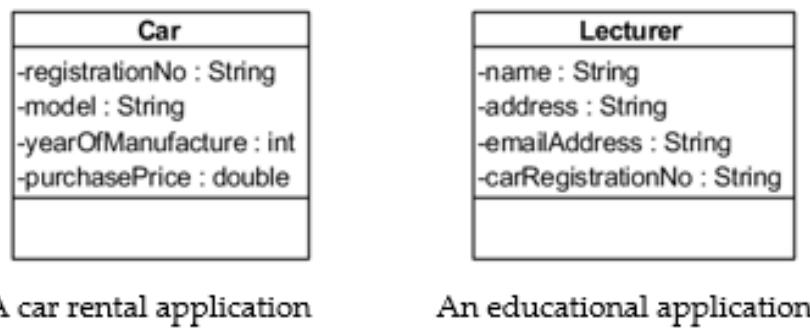


Figure 2.2 A class and a property in different applications

### 1.3 Generalisation Relationships

In a well-structured class hierarchy, a **superclass** is a generalisation of its subclasses. Arranging related classes into a **class hierarchy** is a good way to design an application to take advantage of re-use.

In our application example 1, paragraph 7(b) says:

Given a branch location, information about the company's employees, including their names and employee numbers and, for the branch workers, their hourly overtime rate.

We have noted that this implies three classes for the application:

**Employee**      The application needs to maintain information about the employees – their names and employee numbers.

**BranchWorker**      The application needs to maintain information about the branch workers – their hourly overtime rate.

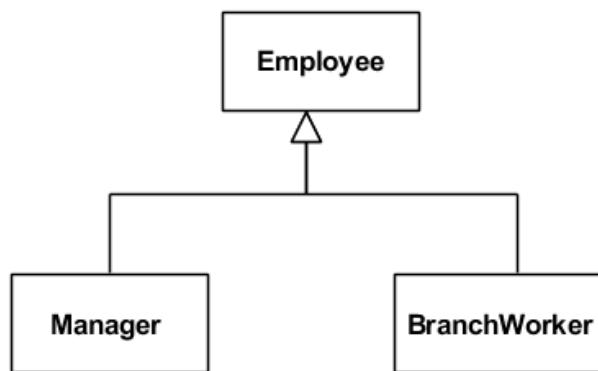
**Manager**      This has to be a distinct class since managers are employees but they do not have an hourly overtime rate.

We can arrange these classes into the class hierarchy shown in Figure 2.3. Their attributes would be:

<b>Employee</b>	name and employee number
<b>BranchWorker</b>	hourly overtime rate
<b>Manager</b>	(no additional attributes specified)

Observe carefully how the requirements state that BranchWorker has an additional attribute hourly overtime rate. This differentiates it from Manager. As a consequence, we can have class hierarchy where BranchWorker is a specialisation of Employee, distinct from Manager.

Note also that you would need to demonstrate your understanding of the concepts involved here. This means that in your answers for your exercises and assignments, you should not repeat the superclass attributes in its subclasses.



**Figure 2.3** The generalisation relationship

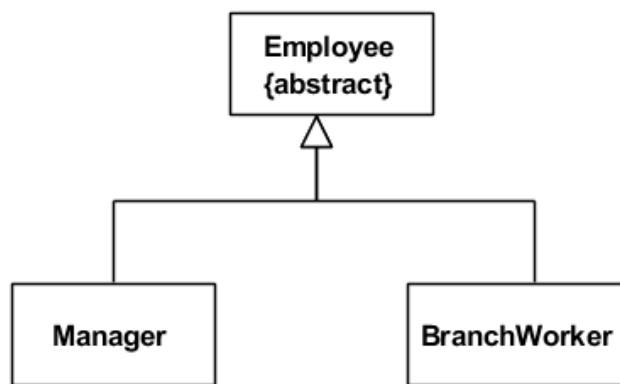
### 1.3.1 Abstract Classes

In object-oriented programming, an abstract class is one that would not have any instances. Any instances that you need to create would have to be created from one of its concrete subclasses.

Sometimes, we can tell from the requirements that a class is abstract. For example, our textual analysis might show that only managers and branch workers but no other types

of staff are involved with the branch operations. If that is the case, then the diagram for the generalisation relationship should be shown in Figure 2.4.

However, for our example, the requirements indicated a possibility that, besides managers and branch workers, quality controllers are another category of employees who might be involved later. At this stage, there is not enough information about how to deal with the quality controllers. Thus, we would leave out the **modifier abstract**.



**Figure 2.4** The generalisation relationship with an abstract superclass

### 1.3.2 Summary of Analysis so far

Our analysis so far has given us the classes, their attributes and their generalisation relationships. With these results, we are able to produce the class description. Recall that the class description is one of the two parts that constitute the structural model for the application:

- The class description
- The class association diagram

The following shows the class description for our discussion example:

<b>Class:</b>	Branch
<b>Attributes:</b>	location, the location of the branch status, the status (operational or under renovation) of the branch
<b>Class:</b>	Car
<b>Attributes:</b>	engineCapacity, the engine capacity of the car registrationNumber, the vehicle registration number of the car
<b>Class:</b>	Employee, superclass of BranchWorker and Manager
<b>Attributes:</b>	name, the name of the employee employeeNumber, the employee number of the employer
<b>Class:</b>	BranchWorker, subclass of Employee
<b>Attributes:</b>	hourlyRate, the hourly overtime rate of the branch worker
<b>Class:</b>	Manager, subclass of Employee
<b>Attributes:</b>	refer to superclass
<b>Class:</b>	Customer
<b>Attributes:</b>	name, the name of the customer

	address, the address of the customer
<b>Class:</b>	<b>Contract</b>
<b>Attributes:</b>	startDate, the starting date of the contract durationOfContract, the length of the contract

## 1.4 Associations

When an object-oriented application is running, various objects will be created from the classes defined or used in the system. These objects do not exist in isolation from each other. They interact among themselves to provide the functions of the application. In object-oriented terminology, objects collaborate among themselves to achieve an objective.

Before an object can collaborate with others, it has to know about their existence. One of the ways that this is done is through associations. A class is said to be associated with another class if their objects are directly related in some way.

### 1.4.1 UML Notation

Figure 2.5 shows an example of the UML notation for an association. In this example, we have two classes, Author and Book, connected by a straight line.

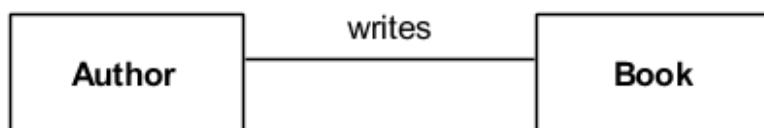


Figure 2.5 An association between two classes

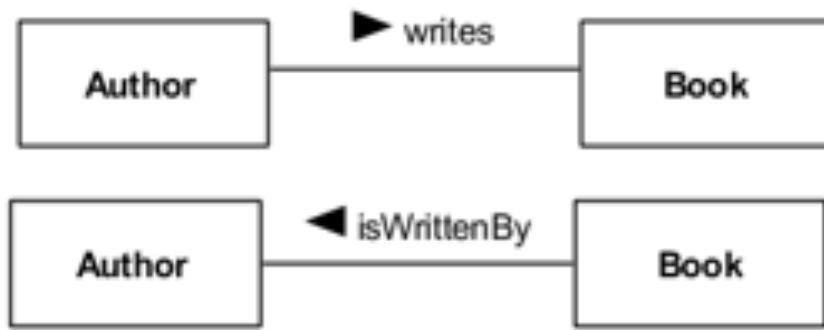
An association is given a name. This is normally a verb or a preposition that reflects the relationship represented by the association. In our example, we call the association writes since an author writes books.

Naturally, an application would involve many more classes. Thus, Figure 2.5 would form part of a **large class diagram** or **class association diagram**, as it is sometimes called.

Note that the above figure also shows that the class Book retains the label Book in singular, regardless of the number of books written by an author. Later, we shall see how the UML notation can handle such a situation when we discuss the **topic of multiplicities**. For now, we will focus only on one part of the association.

The name of an association could also show the relationship in reverse, such as `isWrittenBy` where a book is written by an author.

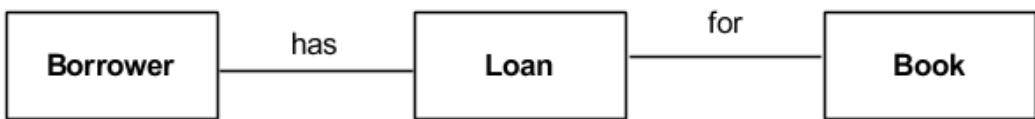
We could show the direction of the association using the notation in Figure 2.6 if we wish. If no direction is shown, the default understanding of the relationship is from the class on the left to the class on the right, or from the class on top to the class at the bottom.



**Figure 2.6** Two ways to show an association

### 1.4.2 Direct Associations

At this stage, we should emphasise that an association is drawn between two classes only if there is a direct relationship between them. Sometimes, the situation is not obvious. For example, assume that we are given three classes: Borrower, Loan and Book for a library application. Our objective is to show the association between a borrower and the books he or she borrowed. This diagram is shown in Figure 2.7.



**Figure 2.7** Showing the correct associations

We might think that a borrower should be associated with a book. But an association should not be drawn between Borrower and Book. With our given objective, the borrower is only associated with those books that he or she has borrowed through a loan. The borrower is not associated with just any book.

### 1.4.3 Finding Associations

To illustrate how to find associations from the requirements, we will use our application example 1, the car rental company from the previous unit. Paragraph 2 in the requirements of this example says:

The company employs managers and branch workers that work in the branches. Each branch has one manager who is assisted by a few branch workers. A manager is responsible for only the branch assigned to him or her.

The phrase "each branch has one manager" is associating a Branch object with a Manager object. We can see that each branch has a particular manager in charge of it and each manager is in charge of a specific branch.

Thus, we can conclude that a manager is directly associated with a branch. To reflect this, we draw a class association diagram as shown in Figure 2.8.



**Figure 2.8** An association between two classes

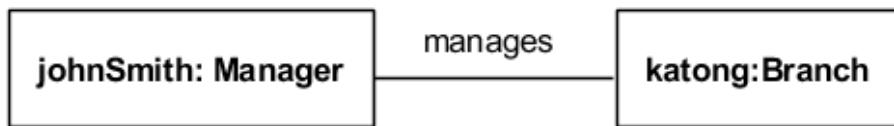
Again, an application would involve many more classes. Thus, Figure 2.8 is only part of a bigger diagram since we are only talking about two classes here.

#### 1.4.4 Association between Objects

At this stage, it is useful to point out that the association is actually between objects, not classes. Although we show the association in a class diagram, it is the actual objects that are associated with each other.

For example, we could have a manager called John Smith and a branch identified as Katong Branch.

The object diagram in Figure 2.9 shows the actual association between the manager, johnSmith (a Manager object) and the branch, katong (a Branch object).



**Figure 2.9** The object diagram shows the actual association

Remembering this is important when we discuss invariants or constraints among the classes in the application.

#### 1.4.5 Study Example: A Car Rental Company

We will now look for the rest of the associations for the application example of the car rental company. We will do this in two steps:

- highlight the classes that we have identified
- determine the link between these classes

By now, you might have noticed from the previous section that verbs, prepositions or their corresponding phrases are used as names of associations. Thus, in our search, we can look for verbs or prepositions linking the words that give us the classes.

## 1. Textual Analysis

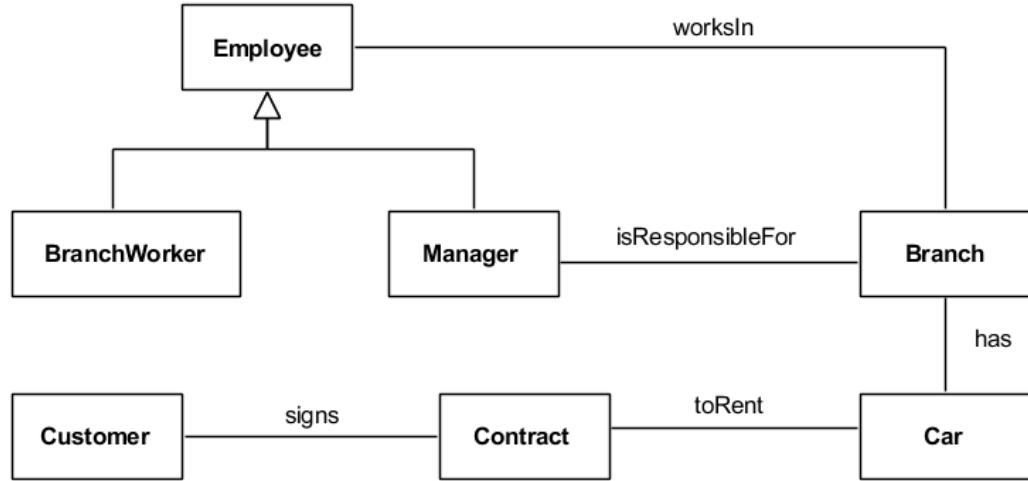
Table 2.3 shows our search for associations. Only the paragraphs of the requirements that have something of interest are included. In the table, the words corresponding to the classes are highlighted in italics. The verbs or prepositions connecting these words are shown underlined.

Table 2.3: *Analysing Requirements for Associations*

Requirements (selected paragraphs only)	Associations inferred
1. The ABC Car Rentals has several branches. Each <i>branch</i> <u>has</u> its own collection of rental <i>cars</i> and is uniquely identified by its location.	Branch <u>has</u> Car
2. The company employs <i>managers</i> and <i>branch workers</i> that <u>work in</u> the <i>branches</i> . Each branch has one manager who is assisted by a few branch workers. A manager <u>is responsible for</u> only the branch assigned to him or her.	Employee <u>works in</u> Branch Manager <u>isResponsibleFor</u> Branch
5. <i>Customers</i> <u>sign</u> a contract to <u>rent</u> a <i>car</i> . No customer may sign more than one contract. However, some <i>customers</i> who are business entities may wish to sign a contract <u>for</u> more than one <i>car</i> . For efficiency reasons, these cars must come from the same branch.	Customer <u>signs</u> Contract Contract <u>for</u> Car (We shall use toRent to be more specific)

An association exists between two classes if there is a direct link between the objects of those classes. The emphasis here is on the word **direct**. Later, we will see that it is possible to link any classes arbitrarily but we would not want to do so.

In the meantime, we can draw the **class association diagram** as shown in Figure 2.10 based on the associations that we have uncovered.



**Figure 2.10** The class association diagram (so far)

## 2. **Analysing Associations**

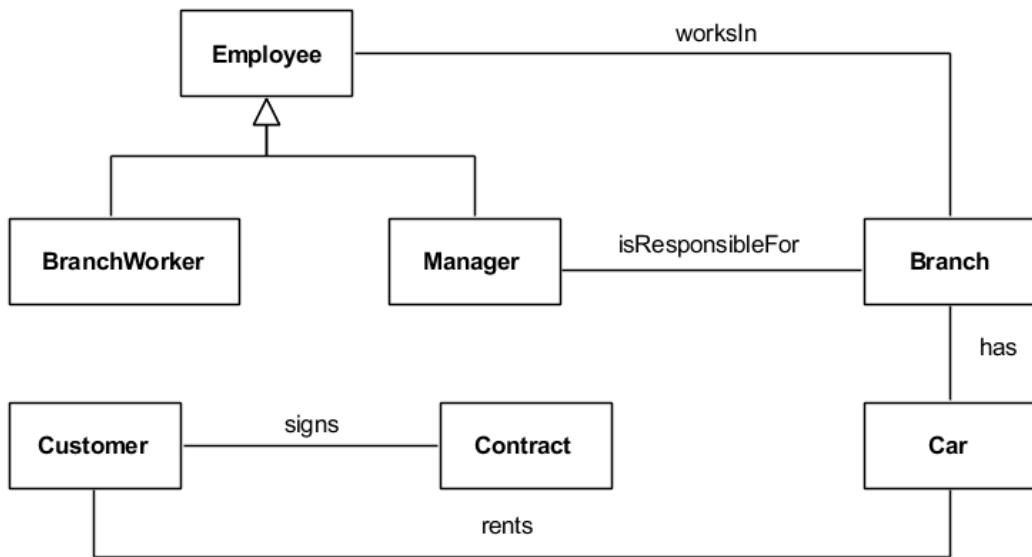
As noted previously, an association exists between two classes if there is a direct link between the objects of those classes. There are associations that seem obvious but they may not necessarily be correct.

Look at the following associations in Figure 2.11 which shows a customer is linked to the car via a contract that he or she has signed for.



**Figure 2.11** Customer is associated with Car through a Contract

We could have drawn the class association diagram as shown in Figure 2.12 which shows that a customer is linked to a car. This seems obviously required since the customer is there to rent a car. However, it also implies that a customer can be linked to any car, not just the one that he signed a contract for.

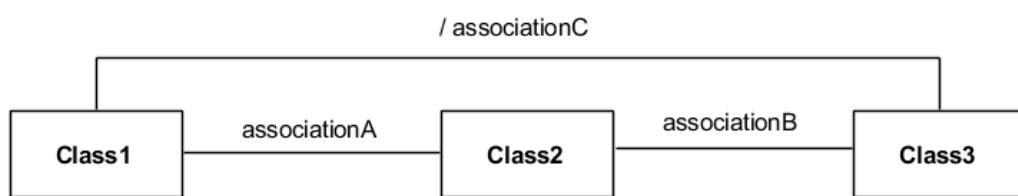


**Figure 2.12** Customer is associated with Car

Therefore, the association between Contract and Car is the one to put in the class association diagram. This association shows that a customer is linked only to the car that he has signed a contract for.

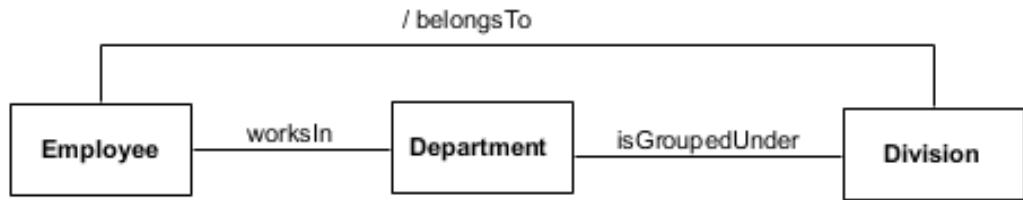
#### 1.4.6 Derived Associations

When classes are linked together with associations, it is often possible to add another association based on those already shown in the diagram. Such an association is known as a derived association. If it is added to a class association diagram, the association name of the derived association is preceded by a slash, as shown in Figure 2.13.



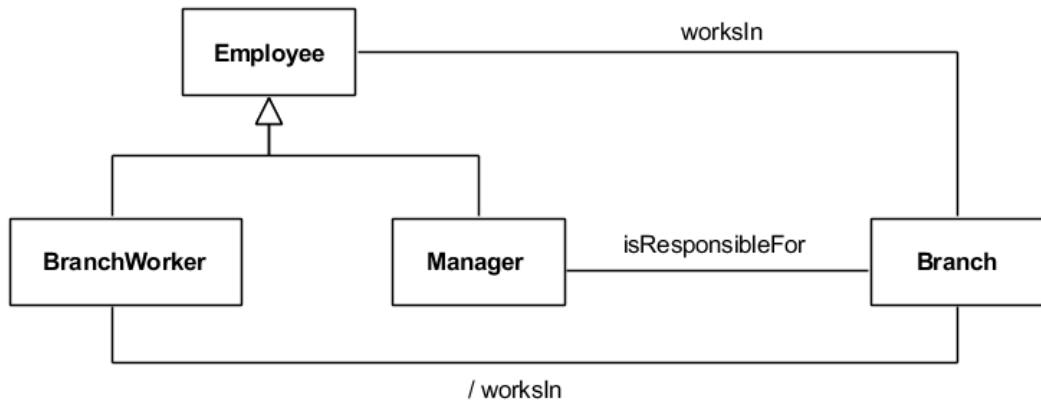
**Figure 2.13** A derived association

Figure 2.14 shows an example of a derived association. It has three classes, Employee, Department and Division for a human resource application in a large company.



**Figure 2.14** An example of a derived association

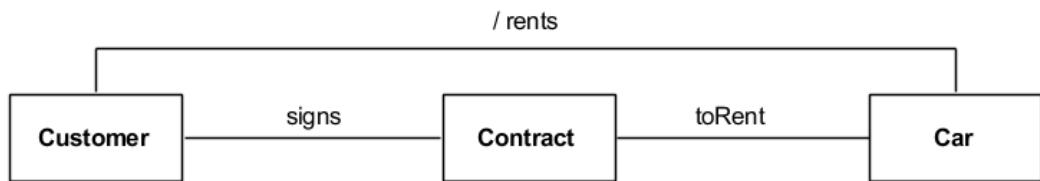
The association `belongsTo` is a derived association because we can see that an employee works in a department and a department is grouped under a division. Thus, the derived association actually adds no new information to the design. In fact, we shall see in later units that it complicates the system implementation unnecessarily.



**Figure 2.15** Another example of a derived association

Figure 2.15 is another example that shows a derived association for the classes `BranchWorker` and `Branch`. Again, the derived association adds no new information to the design. The fact that branch workers work in a branch is already represented by the association between `Employee` and `Branch`. This is because a branch worker is an employee. Thus, the derived association adds unnecessary clutter to the diagram. It should be avoided when it is redundant.

An unnecessary association can also be misleading. Figure 2.16 shows a derived association between `Customer` and `Car`. As we have noted previously, a customer should be associated only with the car that is in his contract, rather than just any car.



**Figure 2.16** A misleading derived association



### Read

Class Diagram for the Mentcare System.

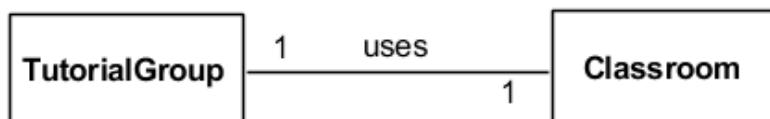
Sommerville, I., *Software Engineering*, 10th Edition, pp.149-151.

## 1.5 Multiplicities

Multiplicity is concerned with the number of objects that can be involved on each side of an association at any one point of time.

For example, consider a classroom that is used by a tutorial group of students. A classroom would normally be occupied by one tutorial group. And one tutorial group would occupy a single classroom. We say that the multiplicity between the classes TutorialGroup and Classroom is one to one.

The class association diagram would be drawn with the multiplicities as shown in Figure 2.17.



**Figure 2.17** Showing multiplicities in a class diagram

A classroom could of course be used by many tutorial groups over the days. Also, a tutorial group could use a few different classrooms over a year. However, at any point of time, one tutorial group would use only one classroom and one classroom would be used by only one tutorial group. With such an association, it is possible to find the classroom if we are given the tutorial group or vice versa.

The object diagram in Figure 2.18 shows a tutorial group TG5 using a classroom #05-01. We say that the TutorialGroup object tg05 is linked to the Classroom object room0501 via the association uses.

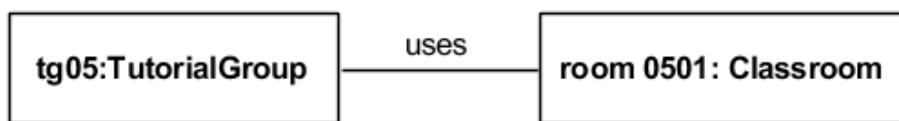


Figure 2.18 An object diagram showing the one to one association

We have previously discussed the association between a manager and a branch from the car rental company example.

Here is another look at paragraph 2 of the requirements for this company:

The company employs managers and branch workers that work in the branches. Each branch has one manager who is assisted by a few branch workers. A manager is responsible for only the branch assigned to him or her.

From this description, we can conclude the following:

- Each branch is managed by only one manager.
- Each manager is responsible for only one branch.

There is a one to one relationship between a manager and a branch. As before, we show it in a class diagram as in Figure 2.19.



**Figure 2.19** An association with its multiplicities

### 1.5.1 Types of Multiplicities

To ensure that we understand the idea of multiplicity, we interpret the association in Figure 2.19 in both directions, from the perspective of each object, as follows:

- For each branch, there is one manager.
- For each manager, there is one branch.

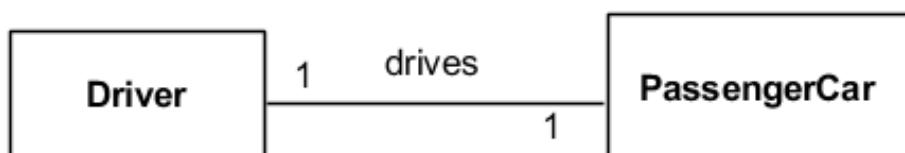
Note carefully the two words "for each".

It is not necessary that the association is always one to one. There could be one to many, many to one and many to many.

To illustrate these possibilities, take a passenger car and the various people that can be associated with it. The following shows the various possibilities for the multiplicities and how we indicate them in a class diagram.

#### 1. One to one multiplicity

In a traffic police computer system, *for each driver*, there is one car. *For each car*, there is one driver at any point of time. Note how we are talking about being at one point of time.



**Figure 2.20** One to one multiplicity

## 2. Many to one multiplicity

*For each passenger*, there is one car. *For each car*, there can be none or one to four passengers at any one point of time.

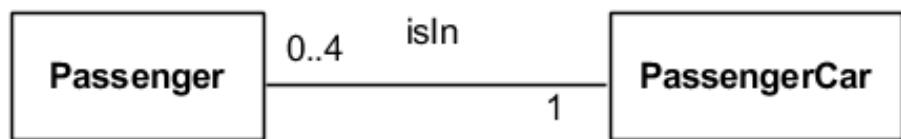


Figure 2.21 Many to one multiplicity

## 3. One to many multiplicity

In the Registry of Vehicles system, *for each registered owner*, there can be one or more cars. *For each car*, there can be only one registered owner at any point of time. *Note that a person is not called a registered owner if he is not associated with a car. This means that to be a registered owner, there must be at least one car associated with the person.*

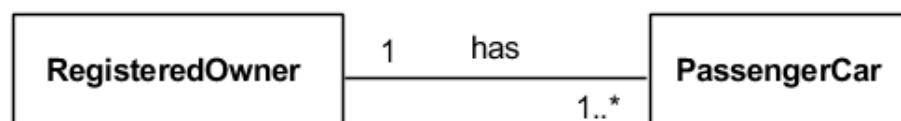


Figure 2.22 One to many multiplicity

Figure 2.23 shows how *an object of Month is associated with 28 to 31 objects of the class Day*.

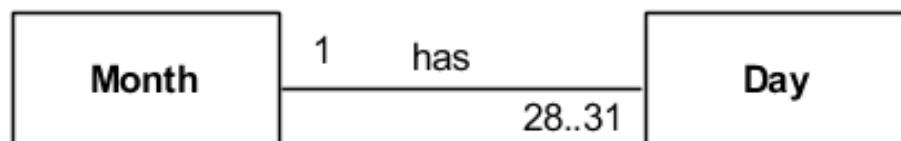
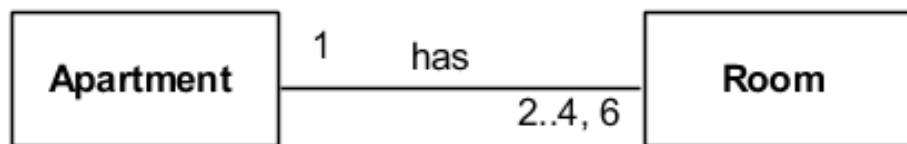


Figure 2.23 An example of one to many multiplicity

Figure 2.24 shows that in a particular condominium, there are apartments with two, three, four or six rooms, where the six-roomed apartment is the roof top penthouse. The purpose of these notations is to show the quantitative (or many) relationships between the objects of two classes. As long as the notations follow the UML convention and are clear to the reader, the diagram would be acceptable.



**Figure 2.24** Another example of one to many multiplicity

#### 4. Many to many multiplicity

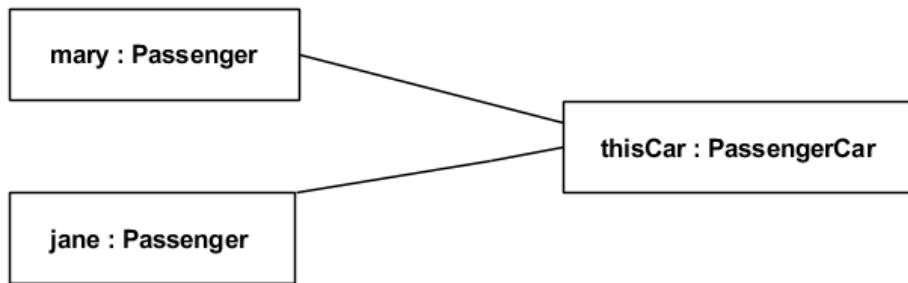
In a repair workshop system, for each technician, there can be none, one or more cars. For each car, there can be one or more technicians repairing it.



**Figure 2.25** Many to many multiplicity

### 1.5.2 Associations in an Object Diagram

As noted previously, the association is between actual objects although they are shown in a class diagram. When an object diagram is used, we could have the following as an example:



**Figure 2.26** An object diagram with many to one association

This object diagram in Figure 2.26 shows the particular situation where there are two passengers associated with a passenger car.

### 1.5.3 Recursive Associations

An association need not be between two distinct classes. Figure 2.27 shows a recursive association where only one class is involved.

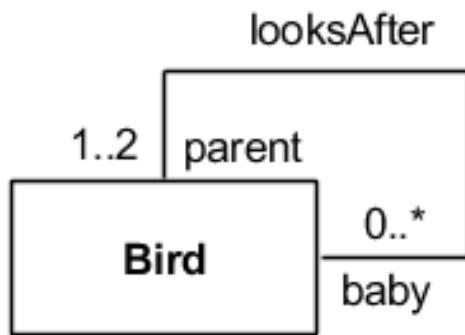


**Figure 2.27** A recursive association

The association shows that one or two birds may look after some unspecified number of (baby) birds.

### 1.5.4 Role Names

If necessary, role names can be added to an association to make the relationship clearer.



**Figure 2.28** An association with role names

Figure 2.28 shows that one or two parent birds may look after several baby birds.

### 1.5.5 Study Example – A Car Rental Company

We will now determine the multiplicities for the associations of the car rental company.

We will go through the requirements again and analyse the text.

**Table 2.3** Analysing Requirements for Multiplicities

Requirements (selected paragraphs only)	Multiplicities inferred
1. The ABC Car Rentals has several branches. Each <i>branch</i> has its own collection of rental <i>cars</i> and is uniquely identified by its location.	For each branch, there are many cars.
2. The company employs <i>managers</i> and <i>branch workers</i> that work in the <i>branches</i> . Each branch has one manager who is assisted by a few branch workers. A manager is responsible for only the branch assigned to him or her.	For each branch, there is one manager. For each branch, there are many employees (manager and branch workers). For each manager, there is only one branch.

<p>4. Not every <i>branch</i> has <i>cars</i> with it. This is because some branches may be undergoing renovation or a new branch may not have cars delivered to it yet.</p>	<p>For each branch, there can be no cars.</p>
<p>5. <i>Customers</i> sign a <i>contract</i> to rent a car. No <i>customer</i> may sign more than one <i>contract</i>. However, some customers who are business entities may wish to sign a contract for more than one car. For efficiency reasons, these cars must come from the same branch.</p>	<p>For each customer, there is only one contract.</p> <p>For each contract, there can be more than one car.</p>
<p>6. At the end of a contract, a customer may return a <i>car</i> to any <i>branch</i> of the company. However, the station workers will send the car back to the branch that is responsible for that car.</p>	<p>For each car, there is only one branch.</p>

For some multiplicities, you may need to draw a conclusion from understanding the application. For example, we would expect a contract to be associated with at least one car. Otherwise, the contract is not meaningful. We would also expect that for each car, there is zero or one contract associated with it at any point of time.

In case of doubt, it is always wise to verify the requirements with the user. For example, we might confirm that for each contract, there is only one customer for easy contract administration.

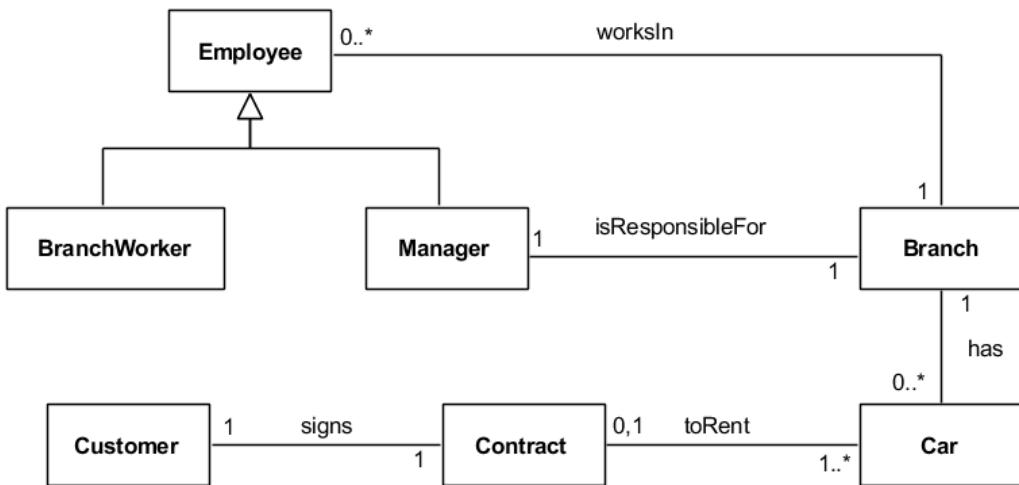


Figure 2.29 The class association diagram with multiplicities

Table 2.4 summarises the multiplicities among the classes.

Table 2.4 Multiplicities among the Classes

<u>Class</u>	<u>Multiplicity</u>	<u>Association</u>	<u>Multiplicity</u>	<u>Class</u>
Employee	0..*	worksIn	1	Branch
Branch	1	has	0..*	Car
Contract	0..1	toRent	1..*	Car
Customer	1	signs	1	Contract
Manager	1	isResponsibleFor	1	Branch

## 1.6 Constraints and Invariants

As noted previously, the structural model consists of the two parts:

1. The class association diagram, showing the associations and multiplicities among the classes

2. The class description, giving the attributes of the classes

There is one category of information that may be required in addition to the details shown in the class diagram and the class description. These are the constraints that the classes may have in their associations. Some course materials refer to these as invariants.

In our example, consider the fragment of the class association diagram in Figure 2.30.

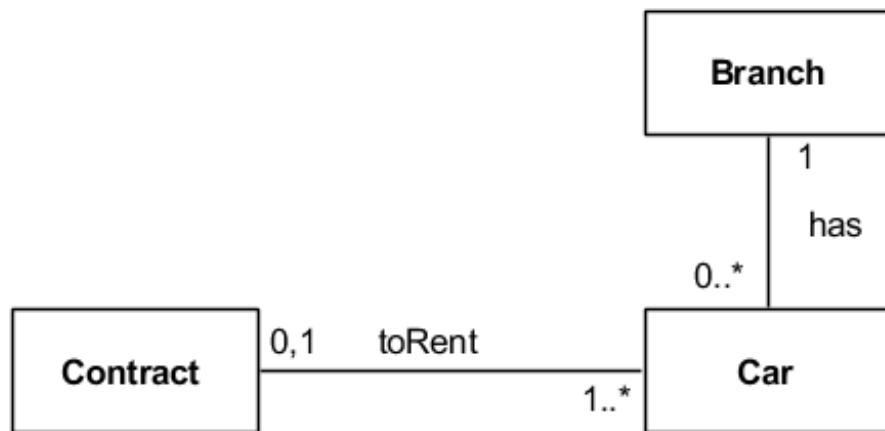


Figure 2.30 Considering constraints in the class association diagram

Now, suppose we have a particular customer, XYZ Corp who is a corporate customer. XYZ Corp wishes to sign a contract to rent two cars. We shall identify these cars as car1 and car2 and the contract as aContract.

Based only on the specifications in the class association diagram and the class description that we have derived so far, we could draw the following object diagram for the rental by XYZ Corp.

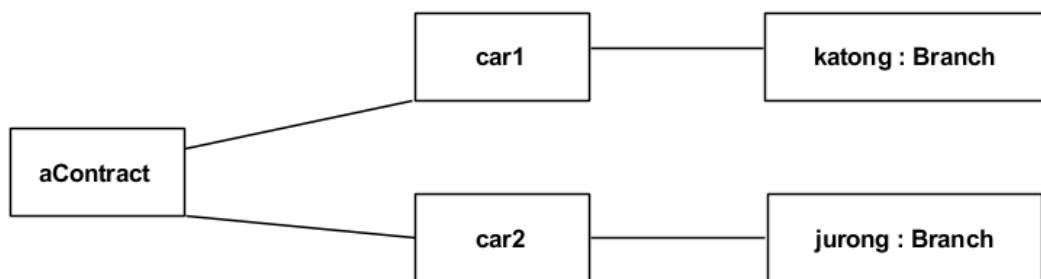


Figure 2.31 Object diagram for the class association diagram in Figure 2.30

To stay focused on our discussion, the diagram in Figure 2.31 shows only the fragment with objects belonging to the classes, Contract, Car and Branch. Notice that it shows a contract for two cars, car1 and car2 where each of the cars comes from a different branch.

However, paragraph 5 of the requirements stated the following:

Customers sign a contract to rent a car. No customer may sign more than one contract.

However, some customers who are business entities may wish to sign a contract for more than one car. For efficiency reasons, these cars must come from the same branch.

Therefore, the object diagram shows a situation where the requirement is violated. But as far as the class association diagram is concerned, the object diagram is correct.

To deal with this situation, we need to have some way to indicate or record the constraint. The best way to do it is to add an invariant to the class description.

Some course text requires invariants to be expressed in a very formal manner. However, experience shows that students often find difficulties in understanding the formal language. We shall use less formal language ("semi-formal"). But it is still important to phrase the constraints correctly in terms of the objects in the structural model.

Here is one way to state the constraint that we have just discussed:

For each instance of the class Contract, there can be more than one instance of Car. Each such instance of Car must be associated with the same instance of Branch. The associations involved in this constraint are:

- *toRent* between the contract and the cars
- *has* between the branch and the cars

Note how we have referred to the instances of the classes involved. Observe also how we have separately identified the associations involved so that it is easy to understand.

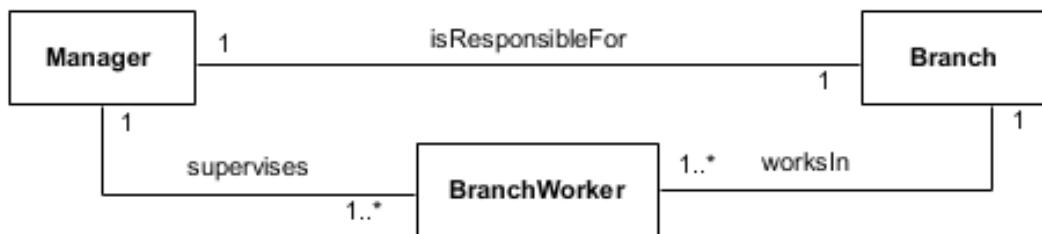
The formal way to state the constraint would have been:

If an instance of Contract is linked via *toRent* with several instances of Car then each such instance of Car must be linked via *has* with the same instance of Branch.

### 1.6.1 Constraints arising from Redundant Relationships

Note that if we put a derived or redundant association into a class diagram, constraints are often required. This is because the associations form a loop connecting the classes involved.

Figure 2.32 shows a fragment of the class diagram with a redundant association.



**Figure 2.32** A class association diagram with a redundant association

Notice that the classes BranchWorker, Manager and Branch form a loop with their associations, isResponsibleFor, supervises and worksIn.

BranchWorker                                  worksIn                                  Branch

Manager    supervises                                  BranchWorker

Manager    isResponsibleFor                          Branch

Stated informally, the constraint that arises from this is:

The branch workers that the manager supervises must be from the branch that the manager is responsible for.

Stated semi-formally,

For each instance of the class Manager, there can be more than one instance of BranchWorker and one instance of Branch. Each such instance of BranchWorker must be associated with that same instance of Branch. The associations involved are:

- supervises between the manager and the branch workers

- worksIn between the branch and the branch workers
- isResponsibleFor between the manager and the branch

Stated formally,

An instance of Manager is linked, via supervises with several instances of BranchWorker and via isResponsibleFor with an instance of Branch. Each such instance of BranchWorker must be linked, via worksIn with that same instance of Branch.

However, we could avoid having to state this constraint when we remove the redundant association, supervises.

Later, when we have to implement the associations by writing the program code, you will find that a redundant association will add considerable amount of work and care in both the design and the coding. At that stage, you would be very pleased to remove all the redundant associations from the diagram.

## 1.7 Aggregation and Composition

When drawing a class association diagram, you might be required to set apart two additional kinds of relationship: aggregation and composition. However, they are not as important as the generalisation relationship that we have discussed in the previous section. It is seldom crucial to the success of the system if they are not identified as special associations. But it is good to be aware of these types of relationships.

Both aggregation and composition occur when something is a part of another. Thus, you can find them when you see phrases such as "consists of", "is made up of" or "is part of".

Figures 2.33 and 2.34 show examples using the UML diagrams. Figure 2.33 shows that a tutorial group consists of students and Figure 2.34 shows that a chessboard consists of squares.

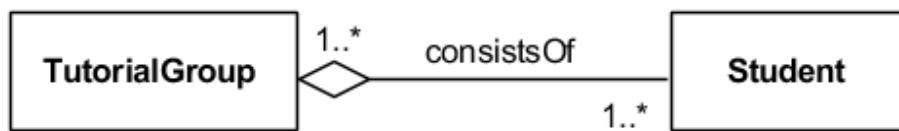


Figure 2.33 An aggregation relationship

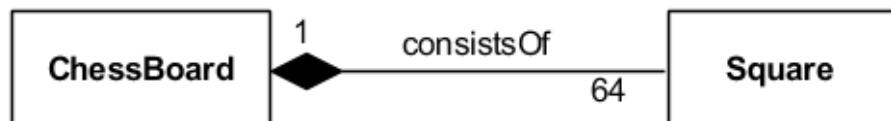


Figure 2.34 A composition relationship

The composition relationship is a "tighter" relationship. The following highlights some of the differences using the chessboard and the tutorial group as illustrations.

- In composition, if you delete the whole, the part will not exist. Thus, removing the chessboard will mean removing the squares as well. This is not necessary the case for a tutorial group. The students can exist even if a particular tutorial group is removed. This is because he or she could be part of the tutorial group of some other modules or he or she might not be assigned with a tutorial group yet.
- In composition, a part can only be part of one whole at most. Thus, a square can only be part of one chessboard. On the other hand, a student can be part of several tutorial groups.



## Read

Sommerville, I., *Software Engineering*, 10th Edition, pp.149-153.



## Activity 2.2

Add aggregations and compositions in the class association diagram given in the textbook, Sommerville, I., *Software Engineering, 10th Edition*, Section 5.3.1, Figure 5.9: Mentcare System.

## Chapter 2: Case Studies

This chapter provides you with more case studies so that you are familiar with developing a structural model. We suggest that you **read the requirements** and **make an attempt to derive the class association diagram** and the **class description**. Then compare your solution **with the text and identify what you can learn from the effort**.

### 2.1 Case Study 1 – Seminar System

The following is a short description of the preliminary requirements of a system for a firm that organises seminars on various professional topics.

The system has to maintain information about each seminar organised by the firm. A typical seminar consists of many sessions. A session usually lasts for either a morning or an afternoon. Each session would have a chairperson who will ensure that the session proceeds smoothly. A chairperson may or may not chair more than one session.

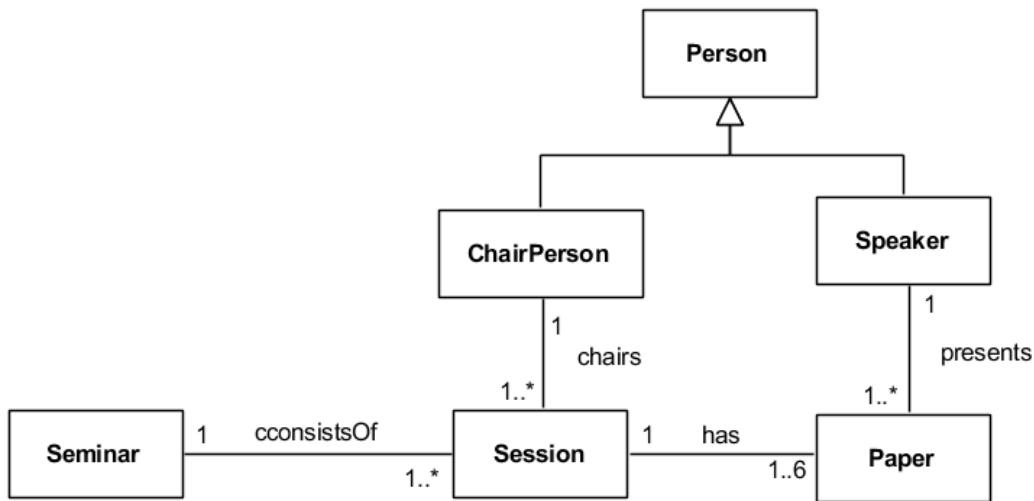
Each session could have up to six papers, each being presented by a speaker. As the seminars are held for professionals, a paper is only presented and discussed in only one session. There is no restriction on the number of papers that a speaker may present.

For the purpose of coordinating all the activities, the system has to maintain information about all the chairpersons and speakers. This would allow these persons to be contactable for any changes in the seminar. The system need not maintain information about any other persons such as participants because that will be done in another system.

The following have been identified as classes: Seminar, Session, ChairPerson, Paper, Speaker and Person.

The tasks for this case study are to identify possible associations and relationships and draw the class association diagram.

Figure 2.35 shows the relationships and associations in a class association diagram.



**Figure 2.35** The class association diagram

The following explains briefly the reasons for the associations and relationships.

#### Relationship

There is a superclass-subclass relationship among Person, ChairPerson and Speaker. The reasons for this are:

- ChairPerson and Speaker have a natural "is a" relationship with Person.
- The description mentioned that the system should help in allowing chairpersons and speakers to be contacted. This implies a common attribute such as contact telephone number.

Person is expected to be an abstract class since the system has to maintain information only about chairpersons and speakers. A person in the system is, therefore, either an instance of ChairPerson or Speaker.

#### Associations

Seminar is directly associated with Session since a seminar consists of many sessions. The multiplicity is one to many since each session is expected to be part of only one seminar.

Session is associated with Paper since each session has up to 6 papers. The multiplicity is one to many since each paper is expected to be part of only one session.

ChairPerson is associated with Session since a chairperson chairs a session. The multiplicity is one to many since each session has only one chairperson but a chairperson can chair many sessions.

Speaker is associated with Paper since a speaker presents a paper.

The multiplicity is one to many since each paper is presented by only one speaker but a speaker can present more than one paper.

The following are also useful points to note (but need not be part of the class diagram or class descriptions):

- ChairPerson is not directly associated with Paper. A chairperson is only associated with those Paper instances that are in the session that he or she chairs.
- Speaker is not directly associated with Session. A speaker is only associated with those Session instances in which he or she presents a paper.
- ChairPerson is not directly associated with Seminar. A chairperson is only associated with the seminar through the Session instance that he or she chairs.
- Speaker is not directly associated with Session. A speaker is only associated with the session through the Paper instance that he or she presents.

## 2.2 Case Study 2 – University System

The following is the statement of requirements for the Local University System (LUS).

### a. Objective of the system

The LUS is used to register students and enrol them in the courses offered by the Local University. The system has to:

- maintain information about the courses offered by the University

- maintain information about the academic staff of the University, including their email addresses and contact telephone numbers
- maintain information about students who have registered with the University, including their allocation to a counsellor
- maintain information about the enrolment of students on courses, including the allocation of a tutor who will assist the students with any academic problems in a course

The system is required to provide reports such as:

- details of tutors and counsellors allocated to a student
- for a given student, details of the courses such as course code and number of credits
- for a given student, details of enrolments such as year of study and courses enrolled

b. **Registration and enrolment**

When a student registers with the University, the following information will be required in the LUS:

- the student identifier and name of the student
- the counsellor who is assigned to assist the student in all general matters

When a student enrolls for a course, the following information will be required in the LUS:

- the status of the enrolment, which is initially set to 'current'
- the course that the student enrolls for
- the tutor who will assist the student with academic queries related to the course

A student who is registered with the University does not necessarily need to be enrolled with any course at all times. The student is allowed a short period to decide which courses to take upon registration and upon completing a course.

If a student withdraws from a course or decides to defer it for a year, the enrolment will be changed to 'withdrawn' or 'deferred' respectively.

### c. Other details

The following provides additional information that might be useful:

- Both counsellors and tutors are academic staff. Counsellors provides general assistance and remains allocated to a particular student as long as the student stays registered with the University. A counsellor can be assigned up to 25 students at any one time.
- Tutors help the students in academic subject matters. In allocating a tutor, the system must ensure that only tutors who specialise in a course would be allocated to students who enrolled for that course. Details of this specialisation is available on another system.
- Other than counsellors and tutors, there are other academic staff whose records have to be in the system. The details of these are not provided at this stage.
- Some courses may not have any enrolments yet if they are just introduced into the curriculum.

Prepare the structural model for LUS. This would consist of the class association diagram and the associated text.

From a review of the requirements, a table for the multiplicities among the classes is prepared and shown in Table 2.6

**Table 2.5** Multiplicities for the Classes in the LUS

<u>Class</u>	<u>Multiplicity</u>	<u>Association</u>	<u>Multiplicity</u>	<u>Class</u>
Counsellor	1	counsels	1..25	Student
Student	1	enrols	0..*	Enrolment

Enrolment	1..*	assignedTo	1	Tutor
Enrolment	0..*	For	1	Course

Figure 2.36 shows the class association diagram.

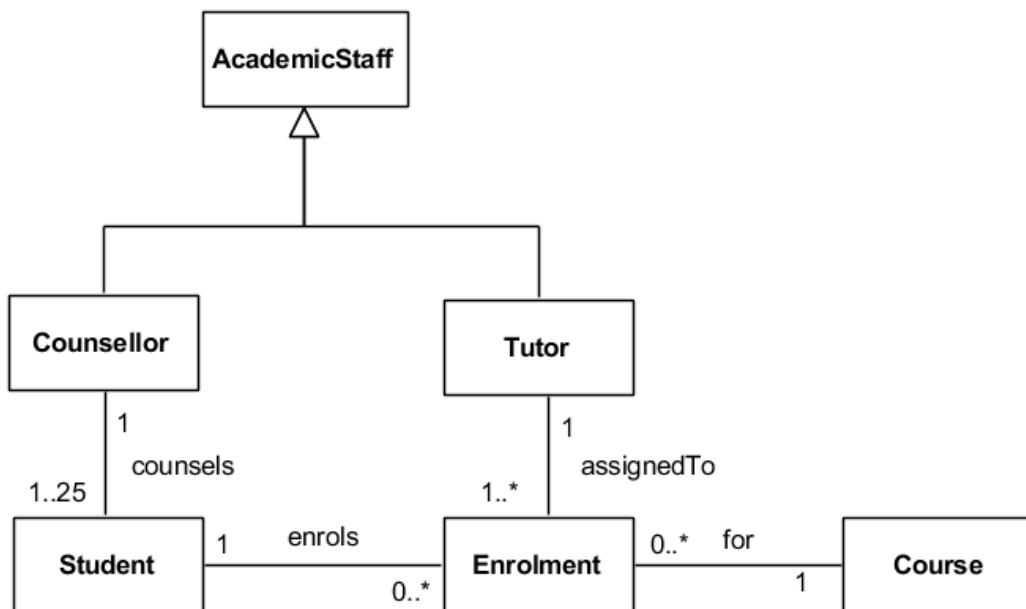


Figure 2.36 The class association diagram

#### d. The class description

**Class:** AcademicStaff, superclass of Tutor and Counsellor

**Attributes:** emailAddress, the email address of the academic staff  
contactTelephone, the contact telephone number of the staff

**Class:** Tutor, subclass of AcademicStaff

**Attributes:** refer to superclass

**Class:** Counsellor, subclass of AcademicStaff

<b>Attributes:</b>	refer to superclass
<b>Class:</b>	Course
<b>Attributes:</b>	courseCode, the course code of the course numberOfCredits, the number of credits earned upon successful completion
<b>Class:</b>	Enrolment
<b>Attributes:</b>	yearOfStudy, the academic year that the enrolment is for status, the status of the enrolment, initially set to 'current'
<b>Class:</b>	Student
<b>Attributes:</b>	studentId, the identifier of the student name, the name of the student

#### e. Constraints

One constraint can be identified from the requirements:

For each instance of the class Enrolment, the instances of Course and Tutor are constrained such that the specialisations of the assigned tutor must include the course in the enrolment. The associations involved in this constraint are:

- *assignedTo* between the tutor and the enrolment
- *for* between the course and the enrolment

It is necessary to state this constraint because an enrolment cannot have any instance of Tutor together with any instance of Course in an unrestricted manner. The tutor and the course that are involved with an Enrolment object must be such that the tutor can tutor students for that course.

The following are also useful points to note (but need not be part of the class diagram or class descriptions):

- The class diagram
  - a. Since the requirements stated that details of tutors, counsellors and courses are required in various reports, classes for these are required in the class association diagram.
  - b. A superclass Person may also be introduced for the classes Student and AcademicStaff. But it is omitted at this stage.
  - c. The class Enrolment is needed to tie up the instances of Tutor, Student and Course for a particular enrolment.
- The class description

Notice that courses is not given as an attribute of Enrolment even though the requirements mentioned "for a given student, details of enrolments such as year of study and courses enrolled". The reasons for this are:

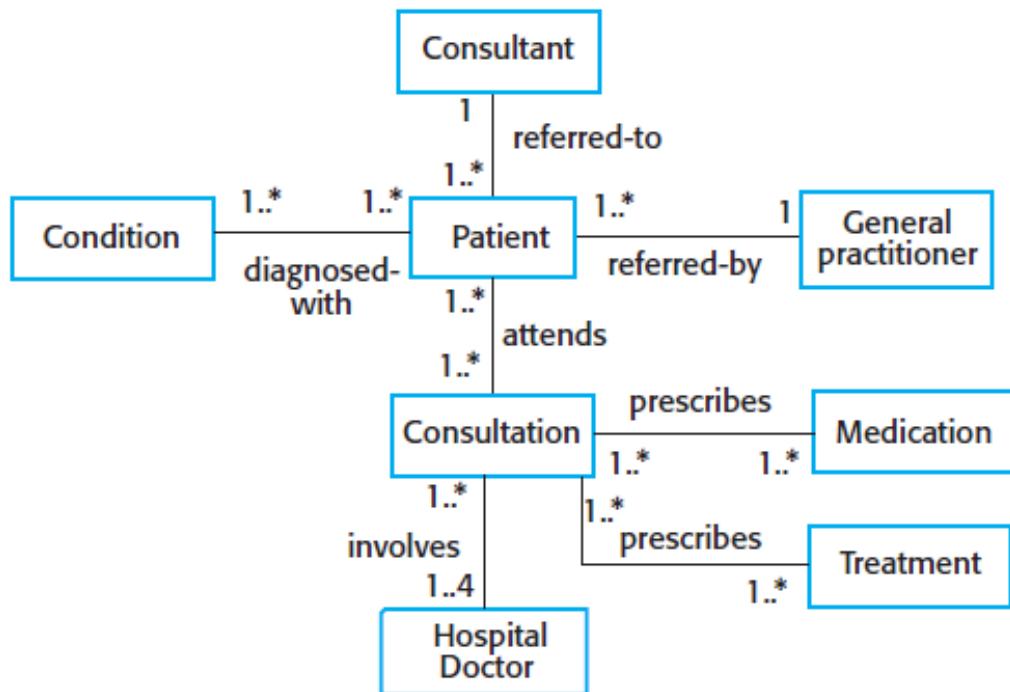
- a. Course is proposed as one of the classes in the system.
- b. The class diagram shows an association between the classes Enrolment and Course.
- c. The dynamic analysis in the next stage will confirm whether courses enrolled would be an attribute of Enrolment.

## 2.3 Case Study 3 – Mentcare System

Please refer to the Mentcare system in Section 1.3.2 of the book, Sommerville, I., *Software Engineering 10th Edition*, , pp.34-36.

- a. The class diagram

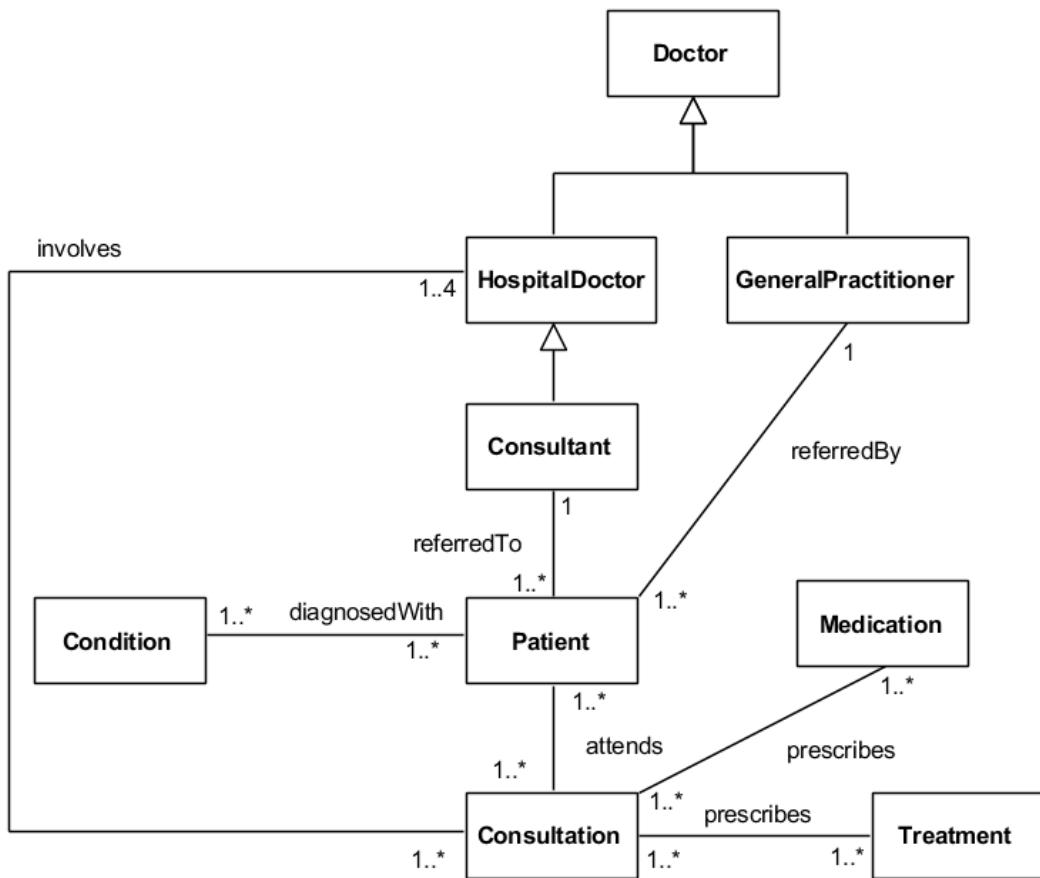
The following have been identified as classes: Consultant, Condition, Patient, GeneralPractitioner, Consultation, Medication, Treatment and HospitalDoctor. Figure 2.37 shows the class association diagram.



**Figure 2.37** The class association diagram for the Mentcare System

(Source: Sommerville, I., *Software Engineering*, 10th Edition)

Upon further analysis, we note that hospital doctors and general practitioners can be generalised to be doctors and consultants are a special type of hospital doctor. Therefore, the class association diagram can be updated as shown in Figure 2.39.



**Figure 2.38** The updated class association diagram for the Mentcare System

The class diagram shown in Figure 2.38 is by no means complete. Further analysis may reveal more classes. This is especially when more information needs to be captured. For example, the **diagnosedWith** relationship between **Patient** and **Condition** shows that a patient may suffer from several conditions and that the same condition may be associated with several patients. If information about the **diagnosedWith** relationship such as the date and time of the diagnosis need to be captured, a new class could be created to store this information as the date and time of the diagnosis are not suitable to be stored in the **Patient** or the **Condition** classes.

**b. The class description**

**Class:** Doctor, superclass of HospitalDoctor and GeneralPractitioner

**Attributes:** doctorId, the unique identifier of the doctor

doctorName, the name of the doctor

contactTelephone, the contact telephone number of the doctor

qualification, the qualifications of the doctor

speciality, the area of speciality of the doctor

**Class:** HospitalDoctor, subclass of Doctor

**Attributes:** refer to superclass

**Class:** GeneralPractitioner, subclass of Doctor

**Attributes:** refer to superclass

**Class:** Consultant, subclass of HospitalDoctor

**Attributes:** speciality, the area of speciality of the doctor

**Class:** Patient

**Attributes:** patientId, a unique patient identifier which is normally their national health number

name, the name of the patient

address, the resident address of the patient

dateOfBirth, the date of birth of the patient  
contact, the phone number of the patient  
familyContact, the name of the family member of the patient

**Class:** Medication

**Attributes:** medicineId, the identifier of the medicine  
medicineName, the name of the medicine  
dosage, the amount of dosage  
unit, the unit of dosage  
sideEffects, the side effects of the medicine

**Class:** Condition

**Attributes:** conditionName, the name of the condition  
conditionDesc, the description of the condition

**Class:** Consultation

**Attributes:** consultationId, the identifier of the consultation  
dateTime, the date and time of the consultation  
notes, the description of the consultation

**Class:** Treatment

**Attributes:** treatmentId, the identifier of the treatment  
treatmentType, the type of treatment

treatmentDesc, the description of the treatment

### c. Constraints

One constraint can be identified from the requirements:

For each instance of the class Consultation, the instances of Consultant and HospitalDoctor are constrained such that the HospitalDoctor that is involved in the Consultation of a Patient is not the same Consultant that is referred to the Patient. The associations involved in this constraint are:

- *referredTo* between the consultant and the patient
- *attends* between the patient and the consultation
- *involves* between the consultation and the hospital doctor

It is necessary to state this constraint because a consultation cannot have any instance of Consultant together with any instance of HospitalDoctor in an unrestricted manner. The patient and hospital doctor that are involved in the Consultation object must be such that the hospital doctor is different from the consultant referred to the patient.



### Activity 2.3

Consider the class association diagram in Figure 2.35 of Case Study 1 – the Seminar System. Suppose a speaker can only present one paper in a session. State an invariant to deal with this situation.



## Activity 2.4

Create a class association diagram by adding associations to the classes given in the textbook, Sommerville, I., *Software Engineering, 10th Edition*, Section 7.1.3, Figure 7.6: Weather station objects. You may also add any additional class(es) deemed necessary. The classes are reproduced here:

WeatherStation
identifier
reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

WeatherData
airTemperatures
groundTemperatures
windSpeeds
windDirections
pressures
rainfall
collect ()
summarize ()

Ground thermometer
gt_Ident
temperature
get ()
test ()

Anemometer
an_Ident
windSpeed
windDirection
get ()
test ()

Barometer
bar_Ident
pressure
height
get ()
test ()

## Summary

The following points summarise the contents of this study unit:

- The outcome or deliverable from performing a structural analysis is a class association diagram which shows the classes and their associations.
- Potential classes can be tangible things, intangible things, events, roles or organisational units.
- The objective of a class is to maintain information about itself, through its attributes and to provide services to other classes through its methods.
- To identify possible classes, a textual analysis of the requirements is performed to identify classes and its attributes.
- Whether a noun is a Thing and therefore a class, or a Property and therefore an attribute, depends on whether the application has to maintain some information about it.
- Generalisation relationships show a hierarchical relationship between classes where the superclass is a generalisation of its subclasses.
- Abstract classes are classes that do not have instances. They are usually the superclass of a generalisation relationship.
- The class description is a documentation of classes, their attributes and their generalisation relationships.
- A class is associated with another class if their objects are directly related.
- The multiplicities for an association between two classes inform the number of objects in one class which are related to another class and vice versa.
- Aggregations and compositions are whole-part relationship between classes. A Class A instance could consist of several Class B instances. An aggregation relationship exists when the Class A instance is destroyed, its Class B instances are not destroyed. A composition relationship is tighter in that when the Class A instance is destroyed, its Class B instances are also destroyed.

## Formative Assessment

1. Identity, attributes and \_\_\_\_\_ are the three things that define an object.
  - a. colour
  - b. motion
  - c. sound
  - d. behaviour
2. What is an instance of a class?
  - a. an object
  - b. an attribute
  - c. an operation
  - d. another class
3. What are attributes?
  - a. The values that we want to keep about the class.
  - b. The data that we want to keep about the class.
  - c. Structure of data that we want to keep about the class.
  - d. Structure of what we want to do with the class.
4. One of these is NOT a possible class in the following scenario:

The customer confirms items in the shopping cart. The customer provides payment and address to process the sale. The system validates payment and responds by confirming the order and provides an order number that the customer can use to check on the order status. The system sends the customer a copy of the order details by email.

- a. Customer
- b. Item
- c. Order

- d. Order number
5. What is "relatedTo" in this diagram?



- a. A link
- b. An association
- c. An association name
- d. A relationship name
6. What is X in this diagram?



- a. Multiplicity
- b. Association
- c. Association name
- d. Association role
7. What is a recursive association?
- a. A class, ClassA has objects that are associated with other objects in ClassA.
- b. A class, ClassA has objects that are associated with one object in ClassB and this object in ClassB is associated with other objects in ClassC.
- c. A class, ClassA has objects that are associated with some objects in ClassB and these objects in ClassB are associated with other objects in ClassC.
- d. A class, ClassA has objects that are associated with objects of another class, ClassB. ClassA also has other objects that are associated with different objects in ClassB.

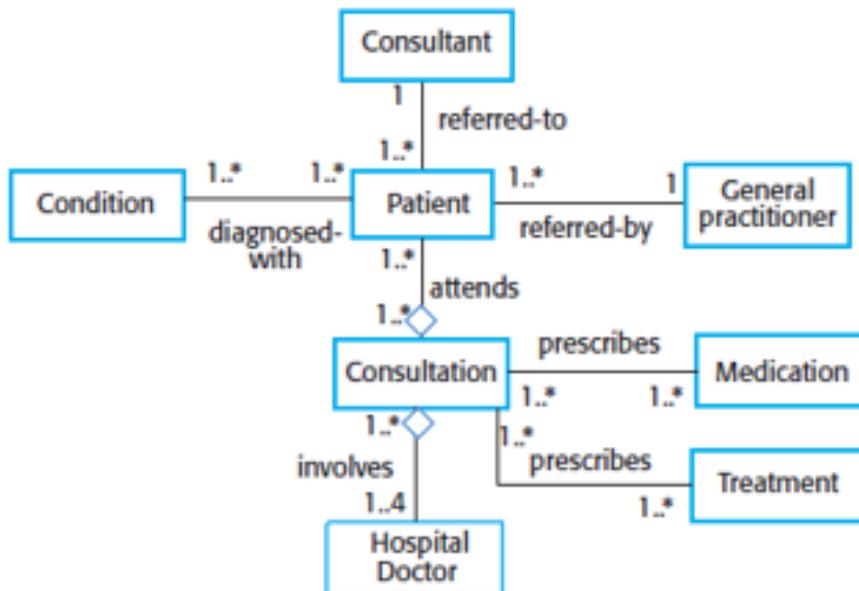
# Solutions or Suggested Answers

## Activity 2.1

Classes for the chain of department stores:

Staff, StoreManager, DeptSupervisor, SalesPerson, Store, Department and Review.

## Activity 2.2



A Consultation is an aggregation of Hospital Doctor and Patient. This means that a consultation consists of many Hospital Doctors and Patients. When the Consultation is deleted, the Hospital Doctors and Patients would still exist in the system. There are no composition relationships. No other associations are whole-part relationships.

## Activity 2.3

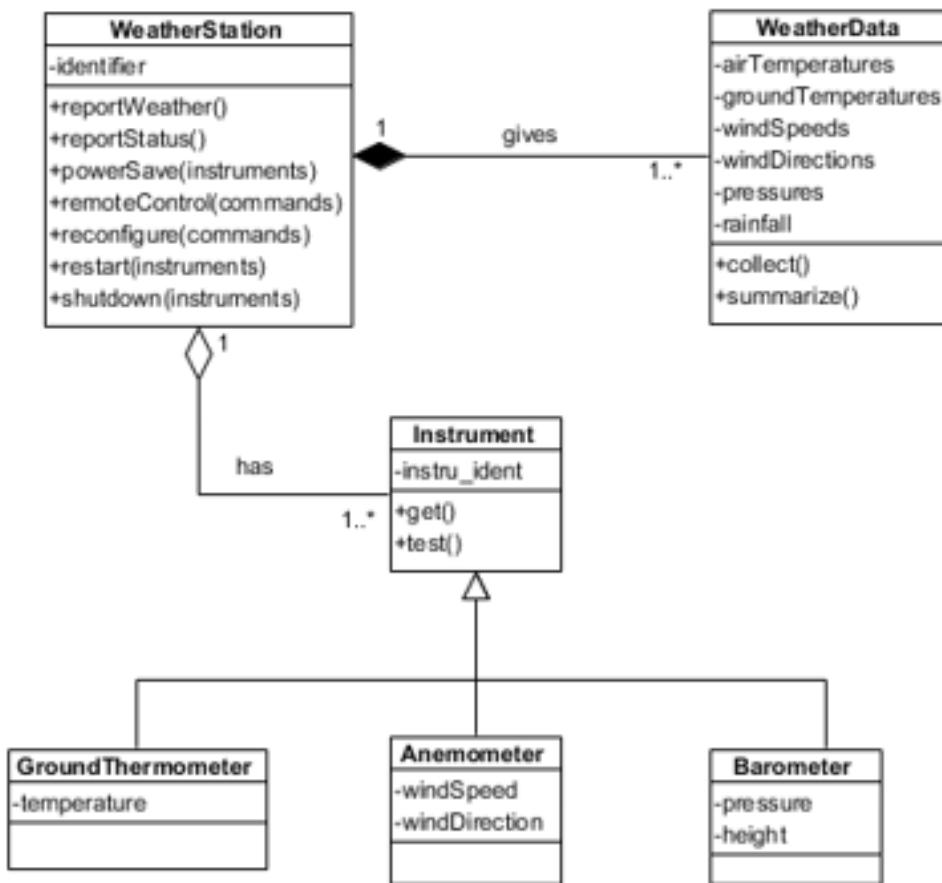
For each instance of the class Paper, the instances of Session and Speaker are constrained such that the speakers in the session are different instances. The associations involved in this constraint are:

- *has* between the session and the paper
- *presents* between the speaker and the paper.

## Activity 2.4

An Instrument superclass may be identified which defines the common features of all instruments such as identifier, get() and test() operations. Aggregation and composition relationships can also be identified.

A possible class association is:



## Formative Assessment

1. Identity, attributes and \_\_\_\_\_ are the three things that define an object.
  - a. colour  
Incorrect. An object may not have any colour. Refer to Study Unit 2, Section 1.1.
  - b. motion  
Incorrect. An object may not move. Refer to Study Unit 2, Section 1.1.
  - c. sound

Incorrect. An object may not produce any sound. Refer to Study Unit 2, Section 1.1.

- d. behaviour

**Correct. The behaviour is what services that the object offers to other objects. This also defines an object. Refer to Study Unit 2, Section 1.1.**

2. What is an instance of a class?

- a. an object

**Correct. An object is an instance of a class. Refer to Study Unit 2, Section 1.2.1.**

- b. an attribute

Incorrect. An attribute is information about the class. Refer to Study Unit 2, Section 1.2.1.

- c. an operation

Incorrect. An operation is the service that a class offers to another class. Refer to Study Unit 2, Section 1.2.1.

- d. another class

Incorrect. Each class has its own instances. Refer to Study Unit 2, Section 1.2.1.

3. What are attributes?

- a. The values that we want to keep about the class.

Incorrect. The class instance (or object) stores the actual value of the attribute. Refer to Study Unit 2, Section 1.1.

- b. The data that we want to keep about the class.

Incorrect. The class instance (or object) stores the data. Refer to Study Unit 2, Section 1.1.

- c. Structure of data that we want to keep about the class.

**Correct. Attributes describe what information we want to keep about the class. Refer to Study Unit 2, Section 1.1.**

- d. Structure of what we want to do with the class.

Incorrect. An operation describes what we want to do with the class. Refer to Study Unit 2, Section 1.1.

4. One of these is NOT a possible class in the following scenario:

The customer confirms items in the shopping cart. The customer provides payment and address to process the sale. The system validates payment and responds by confirming the order and provides an order number that the customer can use to check on the order status. The system sends the customer a copy of the order details by email.

- a. Customer

Incorrect. There is information about the Customer that needs to be stored, e.g. his email. Refer to Study Unit 2, Sections 1.2.1 and 1.3.2.

- b. Item

Incorrect. There is information about the Item that needs to be stored, e.g. what it is. Refer to Study Unit 2, Sections 1.2.1 and 1.3.2.

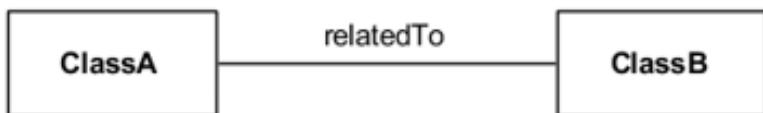
- c. Order

Incorrect. There is information about the Order that needs to be stored, e.g. order number. Refer to Study Unit 2, Sections 1.2.1 and 1.3.2.

- d. Order number

**Correct. Order number is a property to be stored in the Order class. Refer to Study Unit 2, Sections 1.2.1 and 1.3.2.**

5. What is "relatedTo" in this diagram?



- a. A link

Incorrect. A link is used to describe the relationship between 2 objects. Refer to Study Unit 2, Section 1.4.

- b. An association

Incorrect. An association is the relationship between 2 classes. Refer to Study Unit 2, Section 1.4.

- c. An association name

**Correct. This is the name that accompanies the line drawn between 2 classes. Refer to Study Unit 2, Section 1.4.**

- d. A relationship name

Incorrect. The correct name is association name. Refer to Study Unit 2, Section 1.4.

6. What is X in this diagram?



- a. Multiplicity

Incorrect. Multiplicities are shown in pairs. There should be one on the ClassB side too, but there isn't. Refer to Study Unit 2, Section 1.5.4.

- b. Association

Incorrect. Association is the line drawn to connect the two classes. Refer to Study Unit 2, Section 1.5.4.

- c. Association name

Incorrect. Association name is the name assigned to the line drawn to connect the two classes. Refer to Study Unit 2, Section 1.5.4.

- d. Association role

**Correct. The name that is written next to the class on the line drawn to connect the two classes. It further qualifies the relationship of ClassA to ClassB. Refer to Study Unit 2, Section 1.5.4.**

## 7. What is a recursive association?

- a. A class, ClassA has objects that are associated with other objects in ClassA.

**Correct. A recursive association is a self-association, objects in the same class are associated with one another. Refer to Study Unit 2, Section 1.5.3.**

- b. A class, ClassA has objects that are associated with one object in ClassB and this object in ClassB is associated with other objects in ClassC.

Incorrect. A recursive association is a self-association, objects in the same class are associated with one another. There are no second or third classes involved. Refer to Study Unit 2, Section 1.5.3.

- c. A class, ClassA has objects that are associated with some objects in ClassB and these objects in ClassB are associated with other objects in ClassC.

Incorrect. A recursive association is a self-association, objects in the same class are associated with one another. There are no second or third classes involved. Refer to Study Unit 2, Section 1.5.3.

- d. A class, ClassA has objects that are associated with objects of another class, ClassB. ClassA also has other objects that are associated with different objects in ClassB.

---

Incorrect. A recursive association is a self-association, objects in the same class are associated with one another. There is no second class involved. Refer to Study Unit 2, Section 1.5.3.

## References

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.

# Study Unit

# 3

**Dynamic Object-Oriented  
Modelling I: Analysing and  
Constructing Models with  
Walkthroughs and Sequence  
Diagrams**

## Learning Outcomes

By the end of this unit, you should be able to:

1. explain data driven modelling with sequence diagram
2. understand the elements of a sequence diagram
3. explain the objective for performing a walkthrough
4. describe what to look out for when a walkthrough is performed
5. interpret a walkthrough in terms of a real world scenario
6. document a walkthrough with a series of steps
7. visualise a walkthrough with a sequence diagram

## Overview

After generating structural models (Unit 2), the next step is to construct dynamic models. At the end of generating structural model, we get the following documents:

- Class description diagram (classes, attributes and generalisation relationships)
- Class association diagram (associations, multiplicities in associations, aggregations and compositions)

In this unit, we will proceed to generate dynamic models using UML. At the end of this unit, we will have the necessary models of the system to start implementation & testing which is discussed in Unit 5.

Behavioural models are models of the dynamic behaviour of a system as it is executing. While the system is in execution, it needs to interact with its environment. This interaction is either of form data or event. In this unit, we will be discussing these interactions to identify the following for our classes:

1. Instance variables that enable an object to know about other objects in the system, so that they can collaborate among themselves [identified through walkthroughs].
2. Methods that enable an object to provide services to others for their collaborations [identified using sequence diagrams].

# Chapter 1: Model Driven Architecture



## Lesson Recording

Dynamic Analysis

Model-driven architecture (MDA) is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system. Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

- Pros

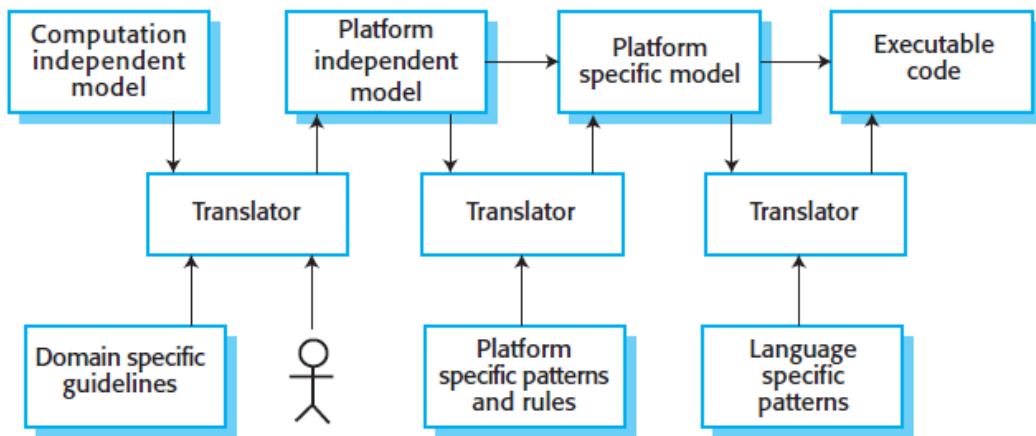
- Allows systems to be considered at higher levels of abstraction.
- Generating code automatically means that it is cheaper to adapt systems to new platforms.

- Cons

- Models for abstraction are not necessarily right for implementation.
- Savings from generating code may be outweighed by the costs of developing translators for new platforms.

## 1.1 Type of Models and Transformations

There are three types of models. Their transformation process to generate code is shown in Figure 3.1.

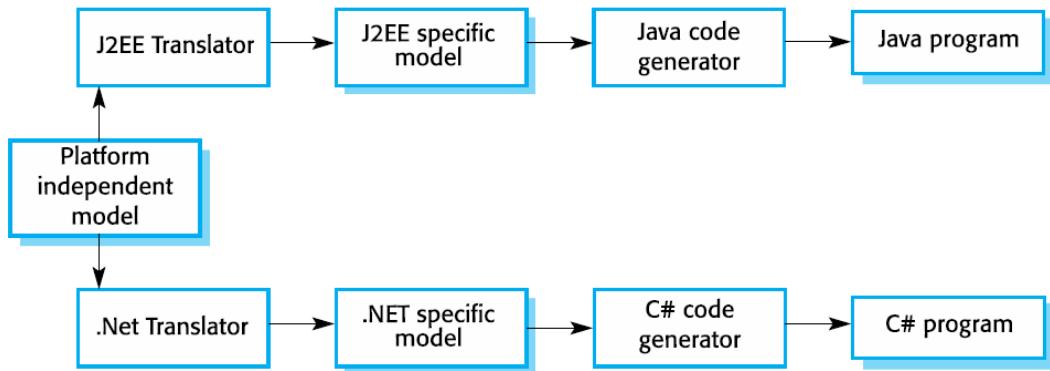


**Figure 3.1** MDA transformations

(Source: Sommerville, I. (2016). *Software Engineering*)

- A computation independent model (CIM)
  - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
- A platform independent model (PIM)
  - These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- Platform specific models (PSM)
  - These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

We can also generate multiple platform specific models as shown in Figure 3.2.



**Figure 3.2** Multiple platform-specific models

(Source: Sommerville, I. (2016). *Software Engineering*)

## Research

Why has MDA not seen good adaptations in the Industry? Which alternate method is widely used?

<https://www.infoq.com/articles/8-reasons-why-MDE-fails/>

Good stackexchange.com forum discussion on the topic:

<https://softwareengineering.stackexchange.com/questions/55679/why-arent-we-all-doing-model-driven-development-yet/55812>

## Chapter 2: Sequence Diagrams

Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing. Data-driven models show the sequence of actions involved in processing input data and generating an associated output. They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system. UML sequence diagrams are used for data driven modelling and analysis.

So far, we have seen a few structural diagrams such as the use case diagram which is great for visualising Actors in a system and what they do. Figure 3.3 below shows a use case diagram for a Library system which shows what a Librarian Actor can do with the system.

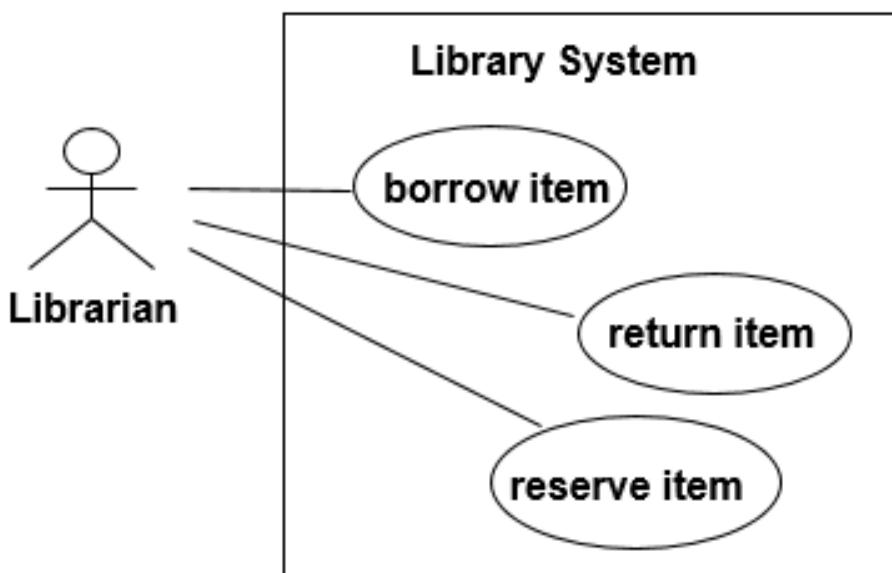


Figure 3.3 Use Case Diagram for a Library System

Another structural diagram is the class diagram which shows the overview of the classes in a system which shows what values need to be stored and how the classes are related including generalisation and composition relationships. Figure 3.4 below shows a class diagram for a Purchasing system.

But these diagrams are not great for representing say, how objects actually interact with one another. We will now look at sequence diagrams which model how the system behaves when it is run, in a visual way. Sequence diagrams are dynamic diagrams which show how different objects change and how they communicate with each other using messages.

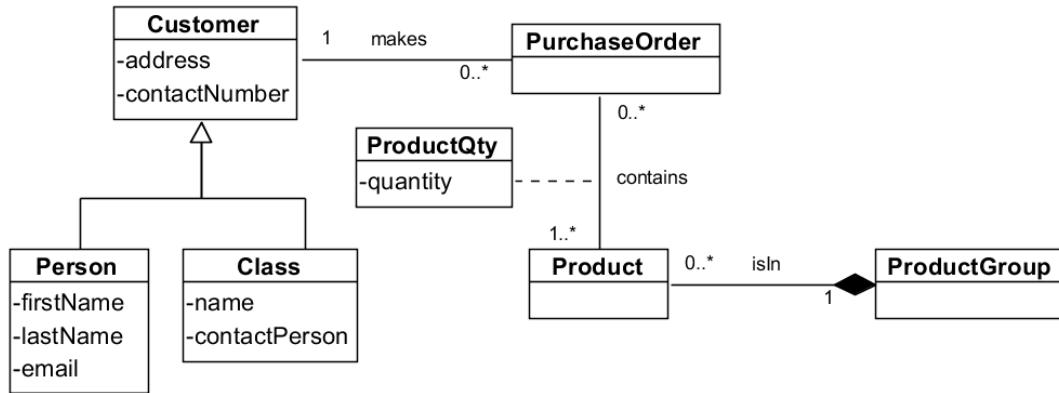


Figure 3.4 Class Diagram for Purchasing System

Consider use case diagrams. The use cases are structural and only show the interaction of the Actor with the system. Now, we want to look at the interactions inside a use case. We want to show how the objects in the use case dynamically interact with each other during system execution. This is known as realisation of the use case. To show this interaction, we need to show the series of messages that a selected set of participants exchanges within a situation in a limited diagram. To develop this model, we need to show which objects participate in the use case and the messages that are passed between them.

However, objects cannot pass a message to another object unless there is an association between the two objects. So, in addition to the use case diagram, a class diagram needs to be developed before a sequence diagram can be created.

A sequence diagram does not describe the entire system, just one particular part of it. So each sequence diagram is for one use case and shows the sequence of messages between a set of objects in that use case. There is a time sequence of when these messages are sent. Alternative messages and loops can also be shown in the sequence diagram.

## 2.1 Elements of a Sequence Diagram

The main elements in a sequence diagram are objects, object lifelines, object activations and messages.

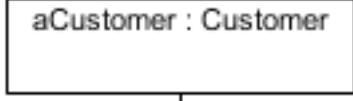
### a. Objects

Objects represent an Actor or Object that interacts in the system. Usually, the Actor initiates the interaction and may be omitted from the diagram. Objects are rectangles that you see horizontally across the top of the diagram. The order in which they are placed is not important. However, to make the diagram neat, it is best to place objects that appear first on the left, so that we do not see a crisscross of messages. Messages would then generally point towards the right of the diagram.

How do we name the objects? There are three ways:

#### 1. Named object of a specific class.

Example: aCustomer : Customer



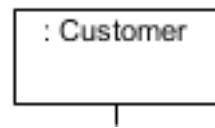
aCustomer : Customer

A colon is used to separate the class name from object name.

This represents an instance, e.g. aCustomer, of that class.

#### 2. Unnamed object of a specific class.

Example: : Customer

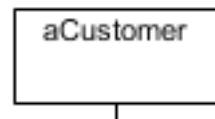


: Customer

This represents all instances of a class.

#### 3. Named object

Example: aCustomer



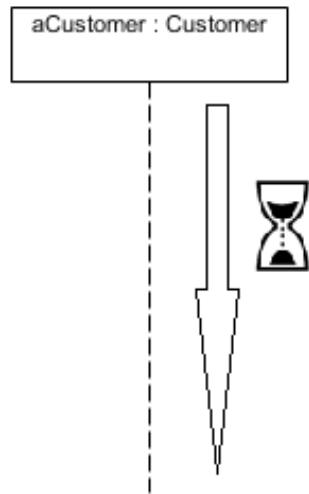
aCustomer

The class name is omitted if it is apparent from the instance name.

## b. Object Lifelines

Beneath the object, we stretch out a dotted vertical line. This represents a lifeline, the timeline of this object where time passes from the top to the bottom. Lifelines represent the objects that participate in an interaction. Messages that go between the objects are attached to this timeline.

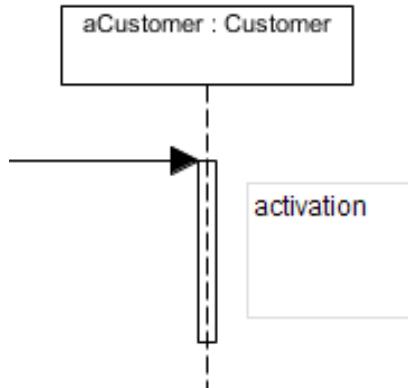
According to UML 2.5 specification, the vertical line may be solid or dashed.



So objects have a lifeline and a life time. They are finite resources and are alive until they are destroyed. Objects can be created and destroyed multiple times on a single lifeline.

## c. Object Activations

Sometimes you will see rectangles on the lifeline called Activation rectangles. These represent the object is active and shows processing being done in response to a message, including waiting time for a return message if it has sent a message to some other object.



A new activation begins when a message arrives for the object, and this arrow points to the top of the rectangle.

Now these boxes are not very important and are optional when drawing on paper, but they do show up when using a UML tool and indicate focus of control, implying where and how much processing occurs.

#### d. Messages

A message is what the sender wants the receiver to do. The message has a name and an argument list which are the information that the receiver needs to do the work. The message is therefore methods of the class in the receiver object.

Figure 3.5 shows the 5 types of messages:

- **Synchronous message**

A synchronous message is represented by a solid arrowhead from the source lifeline to the target lifeline. Because it is synchronous, the sender is blocked from other operations until it receives a response from the receiver. For example, if ClassA object sends a message to ClassB object, it does not continue until ClassB object returns the flow of control back to ClassA object. Likewise, if ClassB object sends a message to another object.

- **Return message**

Return messages are represented by a dashed line with an open arrowhead pointing to the sender. They are a response to the synchronous message to show that the operation is completed and flow of control goes back to the sender object. If there are no values to return to the sender object, we usually do not draw it to avoid clutter. Return messages are also omitted when the message label has an object referencing the result of the message.

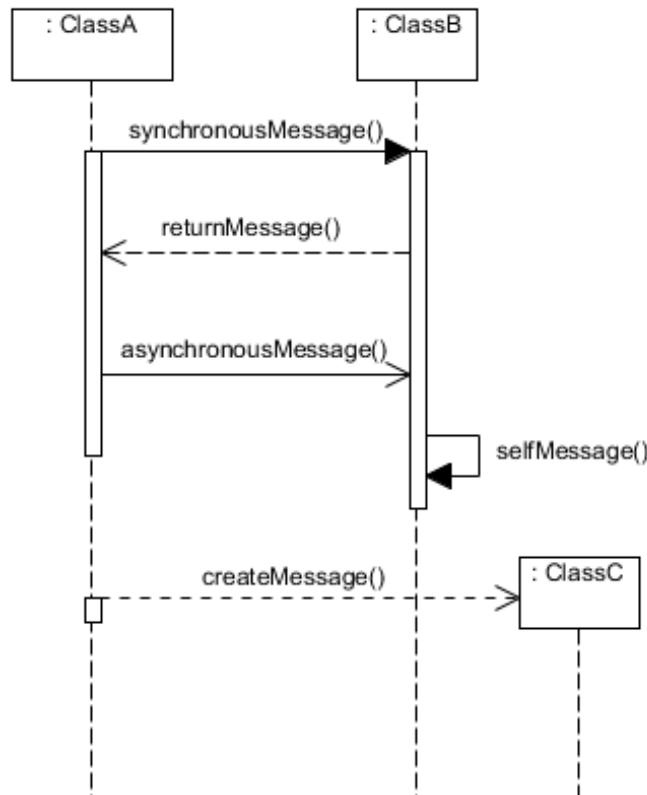


Figure 3.5 Types of messages

- **Asynchronous message**

Asynchronous message is represented by a line with an open arrowhead. Because it is asynchronous, there is no need to wait for a reply and the sender object can carry on with other work. We use asynchronous messages when the message is not time-sensitive or order sensitive. For example, sending a print job to a printer.

- **Self message**

A self-directed message is a message that is sent from the source lifeline to itself. It is represented by a looped line with solid arrowhead. A self message could be to execute another operation belonging to itself.

- **Create message**

A create message represents the creation of an instance of an interaction. It is represented by a dashed line with an open arrowhead. The keyword

**create or new must be used in its message name.** Note that when a **create message** is used, it points to the object created. This newly created object is drawn not on top of the page but drops down to the point of creation. This emphasises that the object is created at this point in time. Objects that are drawn on top of the page denote that they have been previously created before this sequence executes.

### Example:

Figure 3.6 shows a typical sequence diagram drawn using the UML tool Visual Paradigm. You may find some variations in various textbooks due to the versions of UML but the basic ideas are the same.

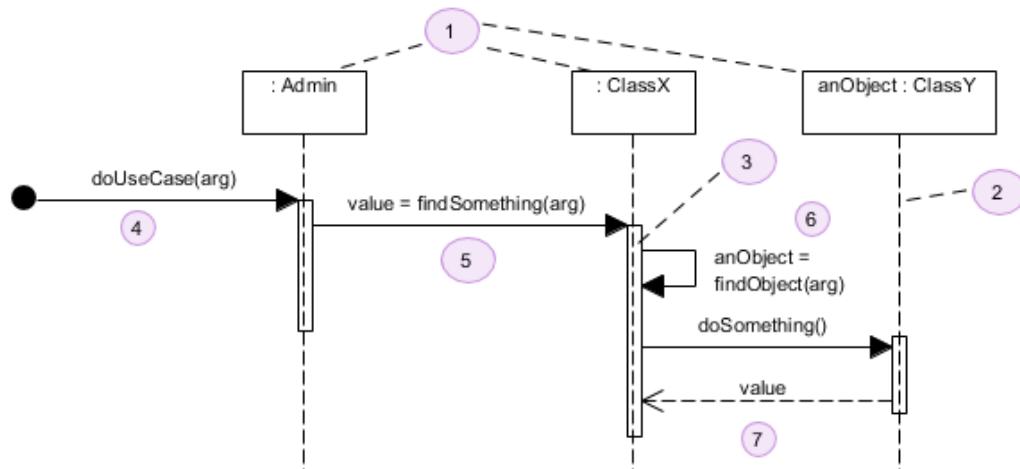


Figure 3.6 The basic structure of a sequence diagram

The following explains each part of the diagram:

1. These are the objects (an unnamed Admin object, an unnamed ClassX object and anObject of ClassY) involved in sending or receiving messages for the use case.
2. This is a lifeline of an object.
3. This shows the activation of the object.
4. This is the message that starts the sequence of messages in the diagram. It may or may not have an argument in the parenthesis. The message is typically received from the external world through the user interface. To avoid clutter, we will not

show the user interface or the actor in our diagrams. However, note that if these are shown, they should be drawn on the left-hand side of the diagram.

5. This shows the message `findSomething()` passed from one object to another. In this case it is sent by the Admin object to the ClassX object. So the Admin object "knows" about the ClassX object. The ClassX object should have the method corresponding to the message `findSomething()`. The message is appropriately named `findSomething()` and not `getSomething()` since a getter is strictly for retrieving an attribute value. It may also have an object (or variable) referencing the result of the message, which in this example is `value`. This object (or variable) must be obtained from subsequent messages (see step 7 below).
6. The `findSomething()` message consists of 2 messages: `findObject()` and `doSomething()`. The `findObject()` message is a self message which uses the argument, `arg`, passed to it to return the result, `anObject`. This means that the ClassX object has the method corresponding to the message `findObject()`. It then passes the message `doSomething()` to `anObject`, which is an instance of ClassY.
7. This represents the reply returned for a message received. The `doSomething()` message to `anObject` of ClassY returns a value. This value is the result of the `findSomething()` message in step 5 above.

The time sequence of the messages and replies are represented by their relative vertical positions in the diagram. Thus, the messages and replies are sent by the various objects in the following sequence:

- `doUseCase(arg)`
- `findSomething()`
- `findObject()`
- `doSomething()`
- reply to `doSomething()`
- reply to `findSomething()` implicit and not shown in the diagram.

## 2.2 Collaborations

The sequence diagram in Figure 3.6 shows that the ClassX object sends the doSomething() message to the ClassY object. In effect, the ClassX object is asking the ClassY object to help in accomplishing its task. We say that the ClassY object is a *collaborator* for the ClassX object.

We can also say that the ClassY object is providing a service to the ClassX object. Thus, we can also call the ClassY object the *server* and the ClassX object the *client*.

## 2.3 Interaction Frames

So far, we have considered only the basic flow of a use case in constructing our sequence diagram. So there is only one possible sequence of messages.

We also need to consider alternative flows which cover a variety of other situations such as

- conditional behaviour, i.e. messages that have a condition and will only be sent if this condition is true, and
- iteration for objects to send messages a number of times.

We use interaction frames to incorporate these alternative flows into the sequence diagram. So you see, we are developing a design that can be easily implemented later.

Interaction frames are rectangular frames that surround a part of or the entire sequence diagram. An interaction frame has a specific operator and a guard (or condition) to determine if the frame is to be executed. The common interaction frame operators are loop (for iteration, similar to while-, for-, do-while loops in programming), opt (for condition, similar to the if-construct in programming) and alt (a condition for branching similar to if-else in programming).

### 2.3.1 Loop

Figure 3.7 shows a sequence diagram that determines the number of times a user is allowed to enter his PIN at an ATM.

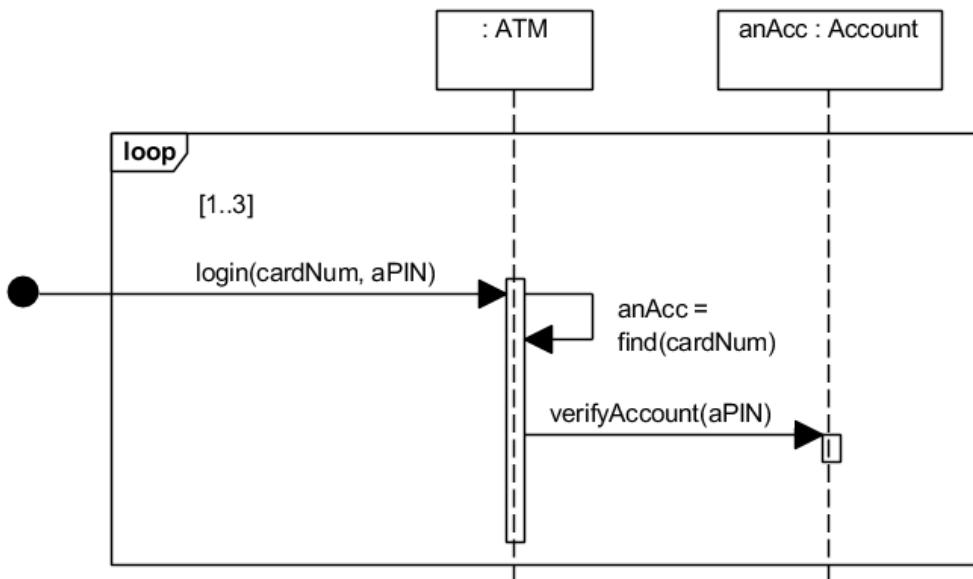
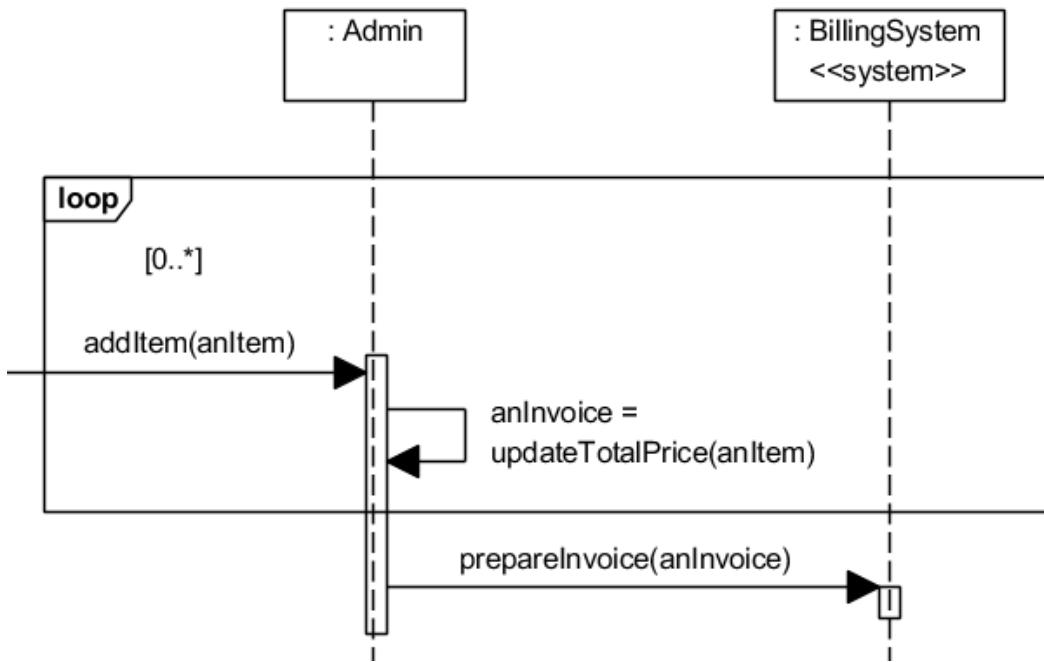


Figure 3.7 Checking password at ATM

The rectangular frame shows that this is a loop interaction frame because the name of the operator appears at the top left corner. The guard (or condition) shows how many times this interaction frame is to be executed. Here we see that it may execute 1, 2, or 3 times. The loop is set up to allow a number of tries (from 1 to 3 attempts). The ATM object finds the Account object, anAcc, and sends it a message, verifyAccount(), to verify whether the aPIN number is correct.

Let's look at another example. Figure 3.8 shows a situation where a user can order multiple items. A loop interaction frame is set up to allow items to be added. Each time an item is added, the total price is updated and stored in the anInvoice object. Note that the loop can execute 0 or more times. This means that if the loop executes zero times, no items are added. This issue needs to be addressed and there are many ways to handle this, which we will look at later. Finally, after the loop has completed, a message, prepareInvoice(), is sent to another system.



**Figure 3.8 Ordering Multiple Items**

There are many ways to write the guard condition in the loop interaction frame. Two ways to do this are:

- As a range.

For example [1..5] or [0..\*]. Please use EXACTLY two dots for range.

- In plain English.

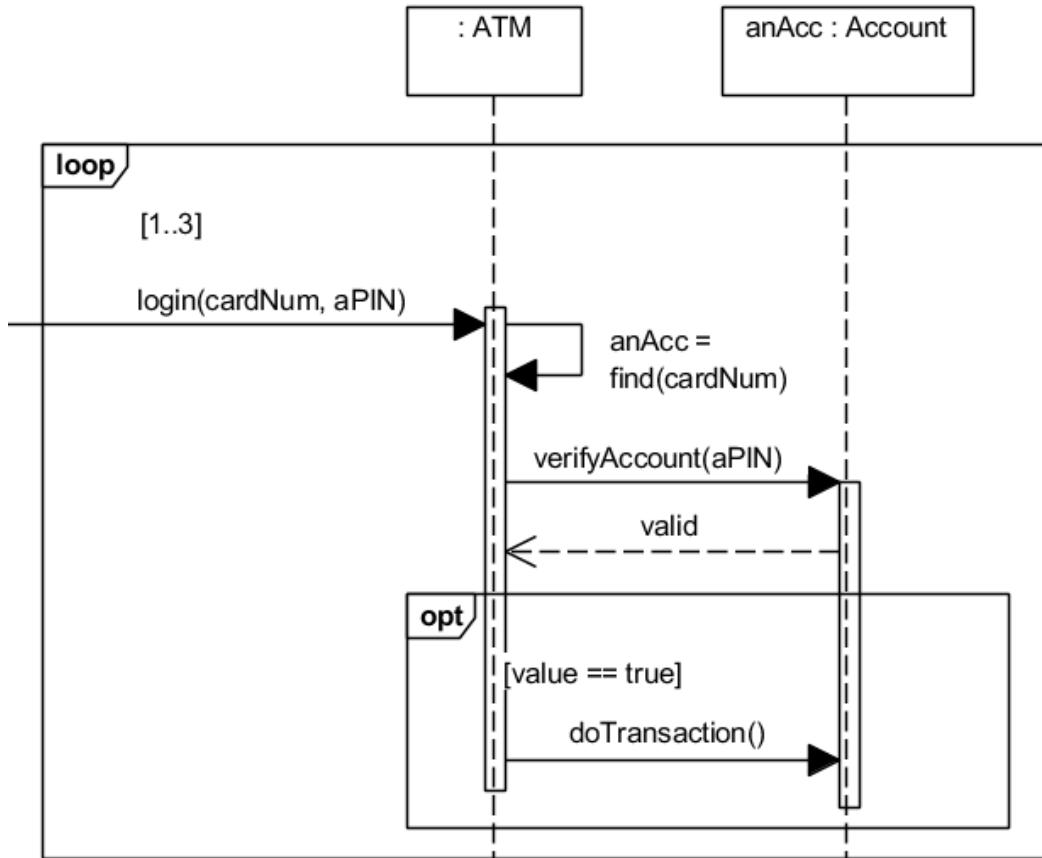
For example, [for all customer accounts] or [for all items in cart].

Do note that square brackets are required.

### 2.3.2 Opt

Let's look at the next frame operator, the opt. The opt allows the framed part of the sequence diagram to be executed only when the guard condition is true. Figure 3.9 shows a sequence diagram for checking a password at the ATM. The user can enter the PIN 1, 2 or 3 times. Each time, the PIN is verified with the account. The result of this verification,

valid, results in an action only if it is true. If the value of valid is false, the system should not allow the user to proceed.



**Figure 3.9** Doing a Transaction at an ATM

Figure 3.9 shows addition information to Figure 3.7. The opt frame with the guard condition checks that valid must be true before the message, doTransaction(), can be sent to the Account object. If the guard value is false, the message inside the opt frame will not be executed.

### 2.3.3 Alt

The alt frame operator is like an if-else condition. Each guard or condition is a check for mutually exclusive alternatives. Figure 3.10 adds additional functionality to the sequence diagram in Figure 3.9. Suppose we wish to withdraw cash at the ATM. Before we allow a withdrawal, the alt operator checks if the amount to withdraw is less than or equal to the

savings amount. If true, withdrawal is allowed, otherwise, a message display("Failure") is executed. Here we see the alt frame inside an opt frame. So frames can be nested.

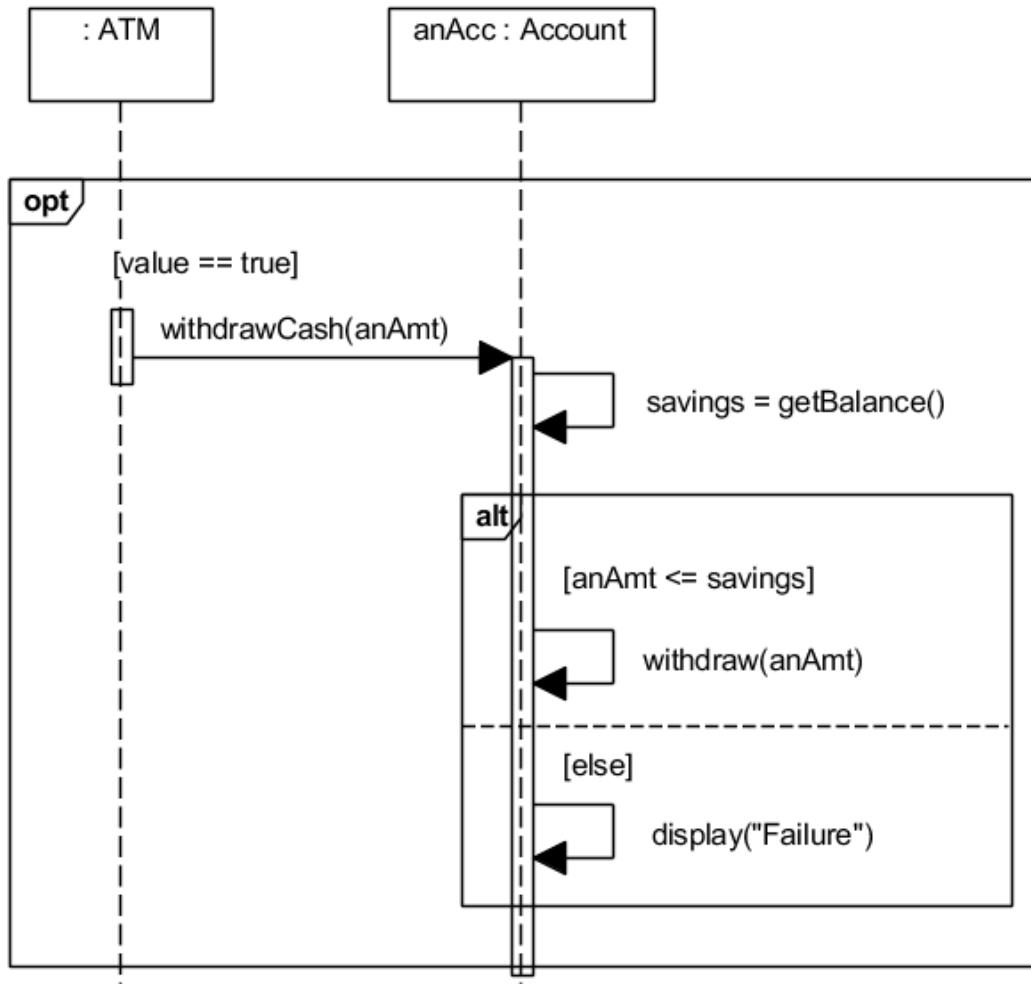


Figure 3.10 Withdraw cash at an ATM

An opt frame only has one section. If an alt frame only has two sections and one of them is empty, etc. there are no messages there, use the opt frame instead.

The alt frame can have more than two sections to show multi-branch if-elif-else condition.

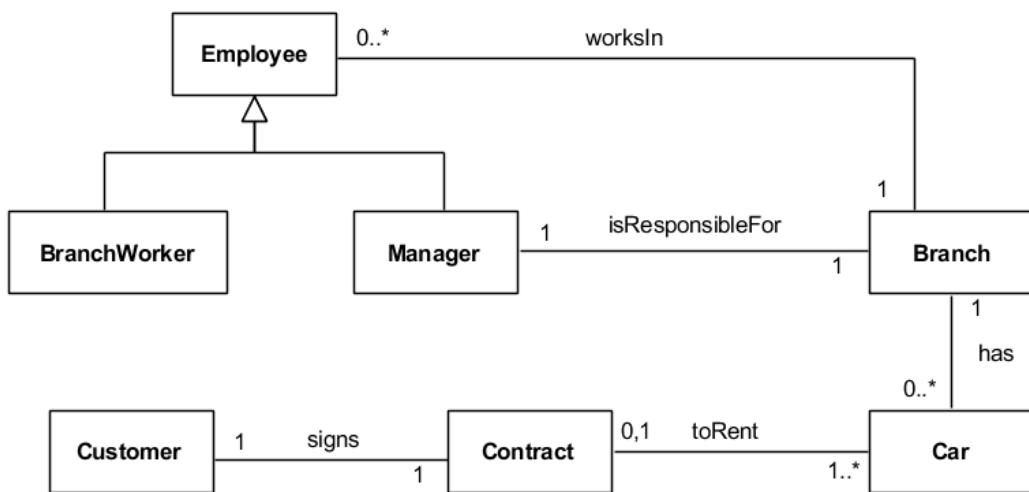
## Chapter 3: Walkthroughs

When we do dynamic modelling to construct the dynamic model, we often use the term **walkthrough** to describe what we are doing. By performing a walkthrough, we will model or work out the behaviour of the objects as they interact among themselves to provide an application function.

The best way to explain the modelling process is to use a simple application and discuss a few examples with it. We will use the car rental application from the previous units for our discussions. We will start by doing a few very straightforward walkthroughs so that you can see the idea easily:

- The first walkthrough we will do is to find the particulars of a manager in charge of a branch that is specified by its location. We will explain this walkthrough in some detail.
- For the subsequent walkthroughs, we will skip some of the details and show only the essential steps.

For easy reference, the structural model for the car rental application discussed in Unit 2 is shown in Figure 3.11. This consists of the class diagram and the class description. We will add more details and update them as a result of our walkthroughs.



**Figure 3.11** The class association diagram for car rental application

<b>Class:</b>	Branch
<b>Attributes:</b>	location, the location of the branch status, the status (operational or under renovation) of the branch
<b>Class:</b>	Car
<b>Attributes:</b>	engineCapacity, the engine capacity of the car registrationNumber, the vehicle registration number of the car
<b>Class:</b>	Employee, superclass of BranchWorker and Manager
<b>Attributes:</b>	name, the name of the employee employeeNumber, the employee number of the employer
<b>Class:</b>	BranchWorker, subclass of Employee
<b>Attributes:</b>	hourlyRate, the hourly overtime rate of the branch worker

<b>Class:</b>	Manager, subclass of Employee
<b>Attributes:</b>	refer to superclass
<b>Class:</b>	Customer
<b>Attributes:</b>	name, the name of the customer address, the address of the customer
<b>Class:</b>	Contract
<b>Attributes:</b>	startDate, the starting date of the contract durationOfContract, the length of the contract

### 3.1 The Orchestrating Class

In dynamic modelling, we need to introduce a class that we shall call the orchestrating class. This class will have only one object in the application. We shall call this object, the orchestrating instance.

Figure 3.12 below shows how the orchestrating class fits into the whole picture. The external world, as represented by the actors in the use cases, interacts with the application through the user interface. The orchestrating instance will then channel requests for actions to the classes in the application.

There are many benefits for having such an arrangement. The most important of these is that the user interface is separated from the classes in the application. As a result, it is possible for an application to take on different user interfaces with minimal changes to the application itself.

Thus, as an example, a retail banking system can be given the appropriate interfaces for Internet banking, telephone banking, banking via the ATM and so on with little changes required in the main application itself.

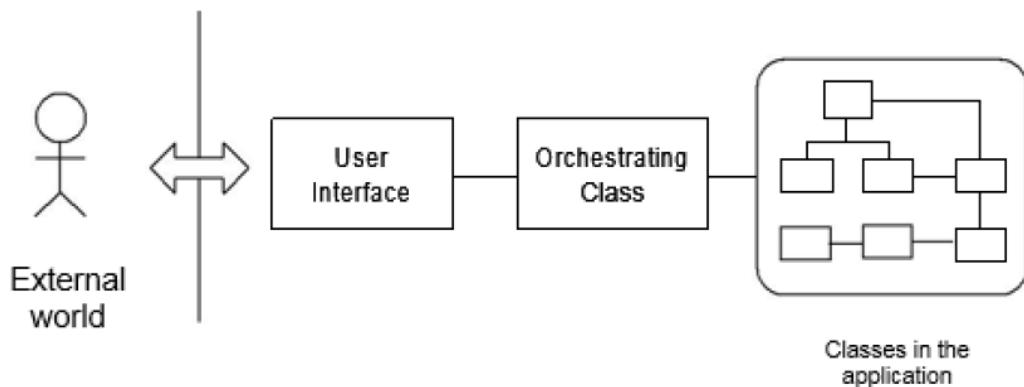


Figure 3.12 The role of the orchestrating class

## 3.2 Preparing for a Walkthrough

Before we begin a walkthrough for our car rental application, we will add the orchestrating class to the class diagram. For convenience, we shall call this class Admin.

In the beginning, we would not know what associations are needed between Admin and the rest of the classes in the diagram. So, we will simply put the rectangle for the orchestrating class at the top of the diagram by itself.

At this point, you might like to note that the use of an orchestrating class in this manner is just one of a few different approaches in designing a system.

However, it is beyond the scope of this module to discuss the other possible approaches.

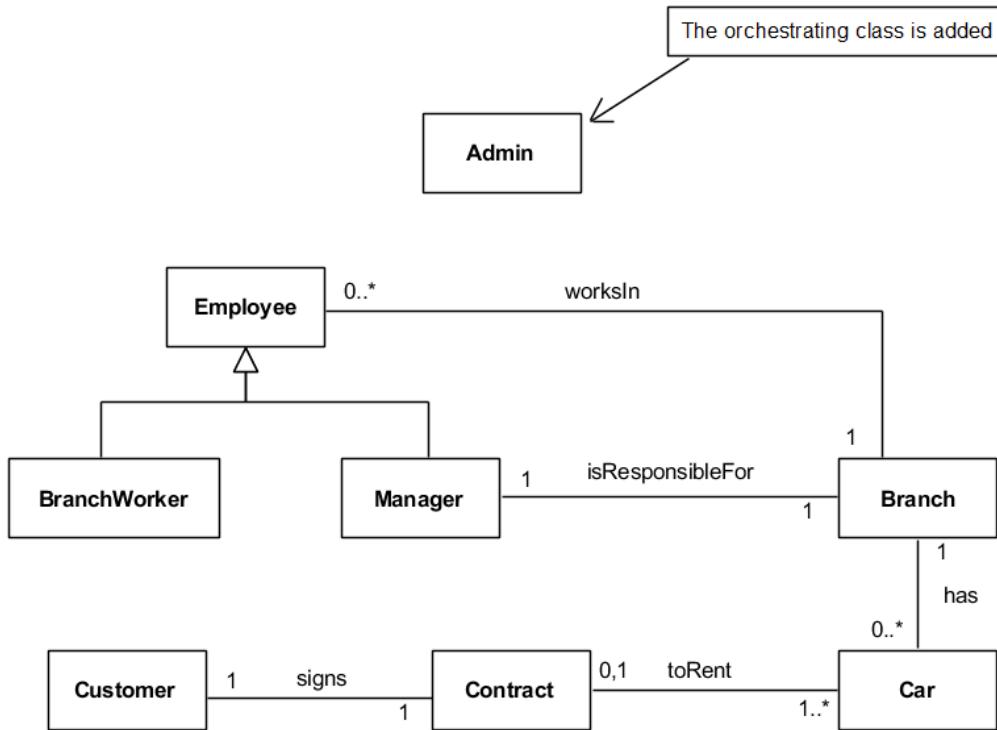


Figure 3.13 Adding the orchestrating class to prepare for the walkthroughs

### 3.3 Performing a Walkthrough (Adding an Instance Variable)

We will need the following to perform a walkthrough:

- The use case to tell us what objective (application function) we are trying to achieve and what we are given in order to achieve the objective.
- The class diagram of the structural model for the initial structural design to work with.
- The class description to be updated according to the outcome from our analysis.

During the process, the class diagram itself would also be updated to show additional information. The result of a walkthrough is that we will be adding new instance variables and methods to the classes that get involved in the walkthrough.

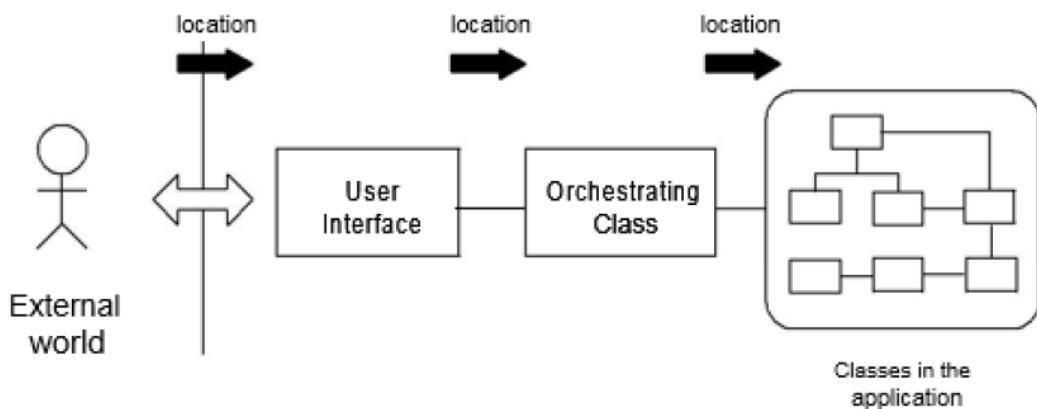
### 3.3.1 Start with the Use Case

Our first walkthrough is to display the particulars of the manager in charge of a branch, given the location of the branch.

Assume one of the functions that the application has to provide is to find the manager in charge of a branch when given the branch location. This means that we could have the following as one of the use cases:

Use case	Show particulars of the manager of a branch.
Initiator	An administrative staff.
Objective	To show the particulars of the manager in charge of a branch.
Pre-conditions	The location of the branch required must be given.
Post-conditions	The application will display the particulars of the manager of the branch specified.
Assumptions, Notes or constraints	None

Examine the use case carefully. The objective is to display the particulars of the manager in charge of a branch and the user supplies the location of the branch to do this. Recall that the requirements have stated that a branch is uniquely identified by its location. Accordingly, in the structural model, the class description has included location as an attribute for the class Branch.



**Figure 3.14** Adding the input to prepare for the walkthroughs

Figure 3.14 shows **diagrammatically**, how branch location is a piece of data that has to be passed to the application. In fact, we would expect location to be a string of characters supplied by the external world.

There are two implications from this:

- **Implication 1: Adding an instance variable**

The orchestrating instance has to know about the branches in the application in order to find the branch manager.

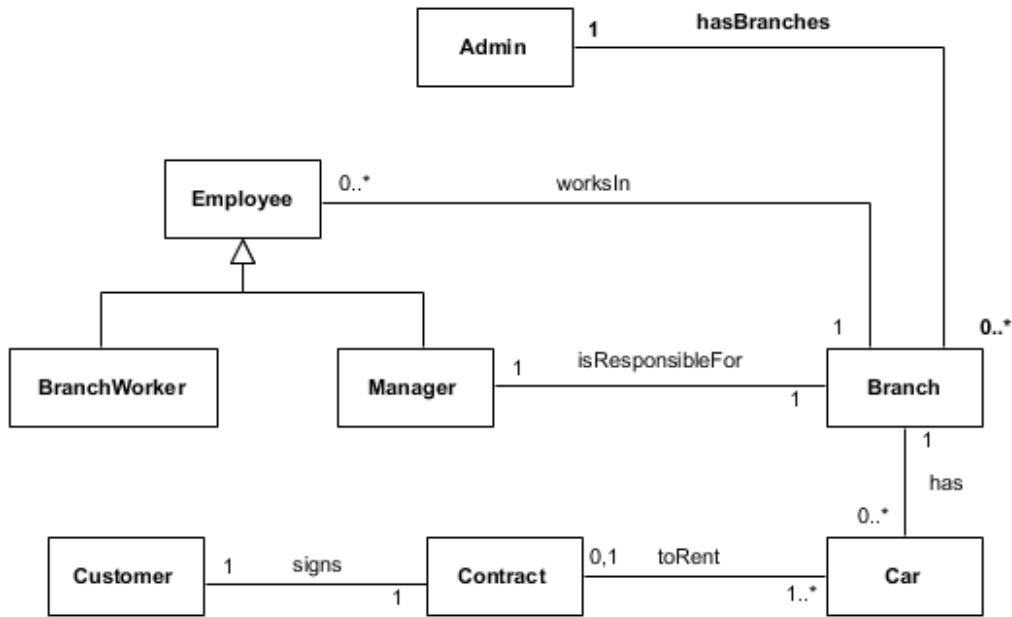
- **Implication 2: Adding a method**

The orchestrating instance has to be able to accept a message to look for a branch manager given the branch location. The class description to be updated according to the outcome from our analysis.

### 3.3.2 Adding an Instance Variable to the Orchestrating Class

This comes from **implication 1**: the orchestrating instance has to know about the branches in the application in order to find the branch manager.

For the class diagram, this means that we need to have an association from the class Admin to the class Branch as shown in Figure 3.15 below:



**Figure 3.15** Adding the orchestrating class to the class diagram

The multiplicity of the association is one to many since there is only one orchestrating instance and there are many branches in the company.

We have a few points to work through here carefully. Take your time to study these points. This is where you lay the foundations for the rest of the analysis and design.

1. We noted that the orchestrating instance has to know about the branches in the company. How could this be achieved?

**Answer:** For the orchestrating instance to maintain information about branches, it has to have an instance variable to do this. Thus, your design for the orchestrating class has to include an instance variable for this purpose.

2. We noted that the multiplicity of the association is one to many since there is only one orchestrating instance and there are many branches. How could that single orchestrating instance keep track of many branches?

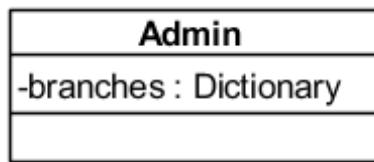
**Answer:** Use array or collection of objects.

3. The next question is what kind of collection we should use for the instance variable?

**Answer:** Depends on what we need to do with the collection. Here, in our use case, we need to look for the branch given a location.

Note that when we say "look for the branch", we are looking for the Branch object using the location given. This means that we need to organise the collection such that we can use the location to get the branch object, for example, a Dictionary collection in Python.

Figure 3.16 shows our design of the orchestrating class Admin, as a result of our walkthrough so far.



**Figure 3.16** Using a collection for the value of the instance variable

We have not reached our objective of getting the particulars of the manager. So, we need to "move" over to the Branch object that we have found and continue our walkthrough from there. Another way of saying this is that we have to *navigate* from Admin to Branch.

In our class diagram, we will now add an arrowhead in the association from Admin, the orchestrating class, to Branch, as shown in Figure 3.17. This arrowhead shows that the Admin class needs to keep track of the branches for us to navigate to the branch that we want.

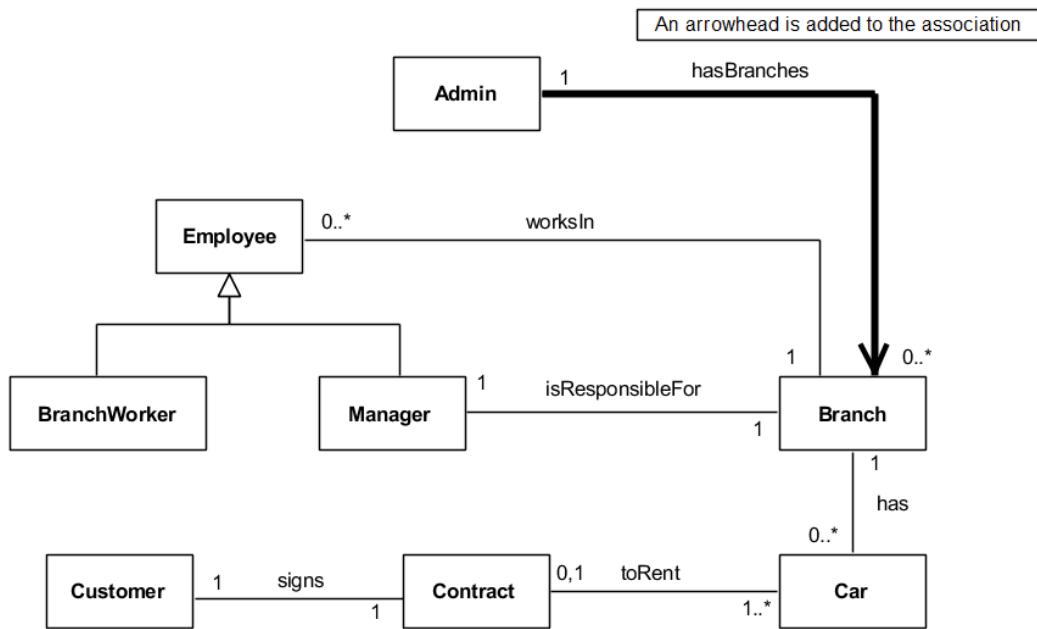


Figure 3.17 An arrowhead is added to the association

### 3.3.3 Implementing an Association

What we have done so far is actually to implement the association

- between the classes Admin and Branch
- in the direction from Admin to Branch.

We have implemented the association by introducing an instance variable in one class (**Admin**) and let this variable reference an object or objects of the other class (**Branch**). We re-draw the relevant portion of the class diagram in Figure 3.18 to see this clearly. Note in particular, the navigation arrow for the association **hasBranches** from Admin to Branch.

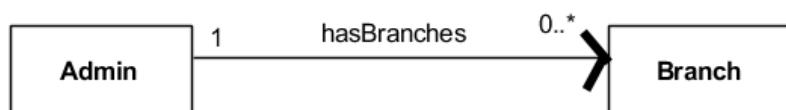


Figure 3.18 Direction of navigation between the classes

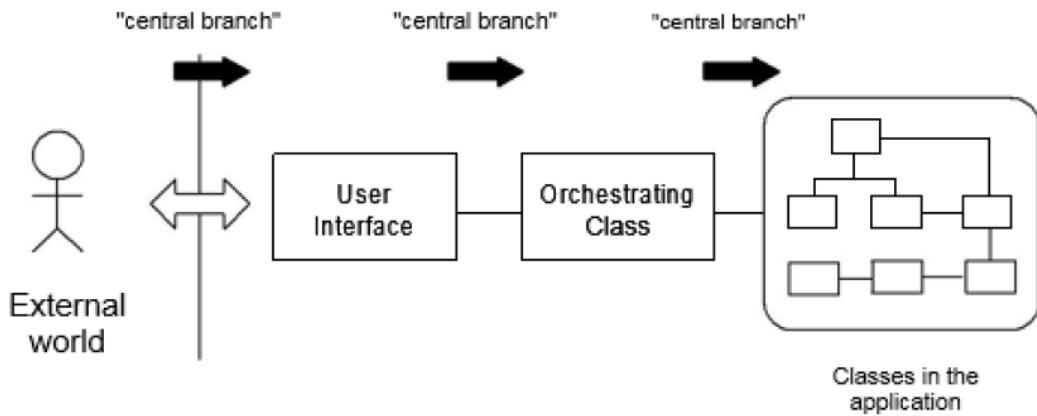
The following table summarises the essential points that you should note carefully:

<b>Class being modified</b>	Admin
<b>Association being implemented</b>	hasBranches
<b>Direction of navigation</b>	From Admin to Branch
<b>Multiplicity</b>	One to many
<b>Instance variable added</b>	Branches
<b>Value of the instance variable</b>	A Dictionary of Branch objects, using the location as the key.
<b>Justifications</b>	<p>An instance variable is added to the class Admin because it has to keep track of branches.</p> <p>A Dictionary is used because:</p> <ol style="list-style-type: none"> <li>1. The multiplicity is one to many. The Admin object has to keep track of many branches.</li> <li>2. The application needs to look for a branch by using the branch location. Thus, the branch location is used as the key in the Dictionary.</li> </ol>

### 3.4 Performing a Walkthrough (Adding Method)

So far, we have completed only one step of our walkthrough, after which the admin maintains information about branches. Now, if the user to query branch manager for a particular branch location (for example, "central branch" as shown in Figure 3.19), the Admin class needs to provide a service or method for the user to pass the location information or message. Hence, we will add a method, `showManager(location: String)`.

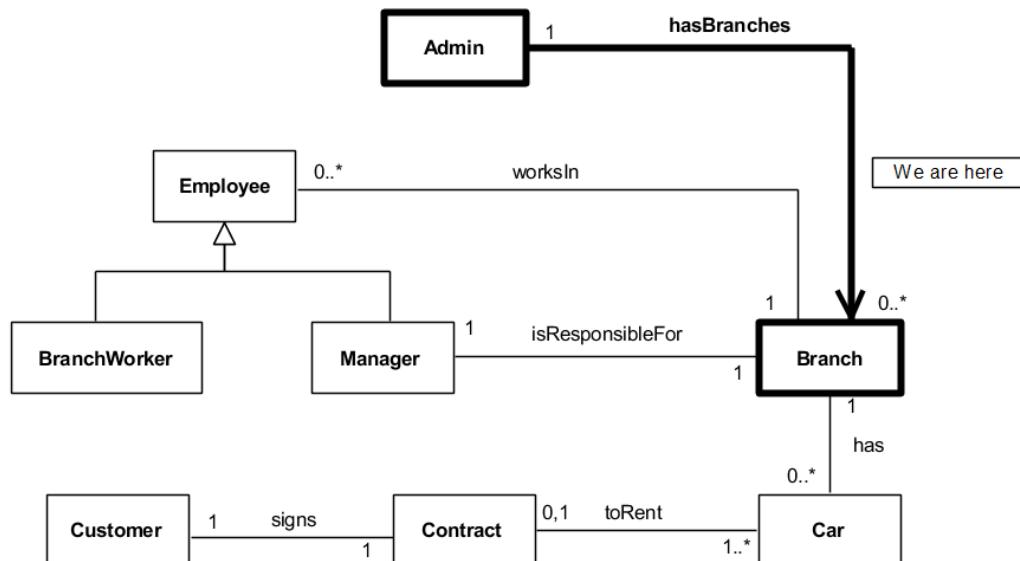
Upon receiving this location message, the orchestrating instance will respond by executing the method `showManager()`.



**Figure 3.19** A user wishes to find the manager in charge of a particular branch

### 3.4.1 Continue with the Walkthrough

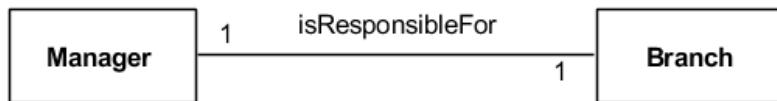
Now, we shall continue with our walkthrough. We have located the branch that we want by using the given location as the key to search the Dictionary collection. Thus, we have found the actual Branch object.



**Figure 3.20** Where we are in our walkthrough

Figure 3.20 shows where we are at this point of the walkthrough. Bear in mind that we are using the class diagram to show this. In fact, the application is executing and we have located the actual Branch object that we want.

With the actual Branch object, it should be easy to find the Manager object. There is only one manager for each branch. Getting the branch should get us its manager.



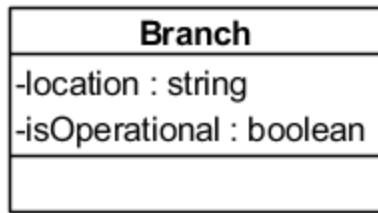
**Figure 3.21** The multiplicity shows there is only one manager for each branch

However, for us to get the manager, again we need the Branch object to keep track of its Manager object. As you would know by now, an object maintains information and keeps track of things through its instance variables. This means that we need to add an instance variable to the class Branch to keep track of its manager.

From structural analysis in Unit 2, we already have the following specifications for the class Branch:

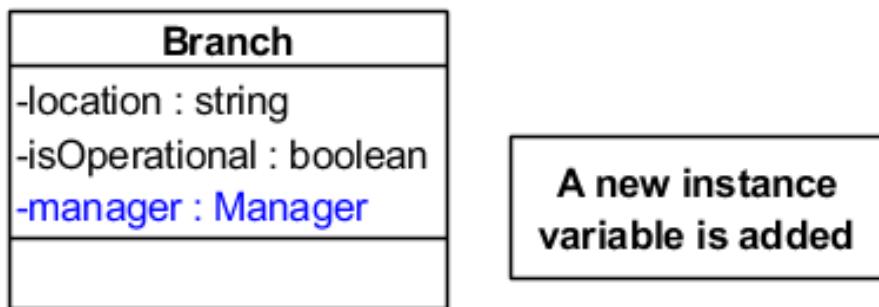
Class:	Branch
Attributes:	location, the location of the branch status, the status (operational or under renovation) of the branch

Arising from this, we could have the class diagram as shown in Figure 3.22 for Branch. For ease of implementation, we have replaced the attribute status with the boolean attribute isOperational.



**Figure 3.22** The class Branch from the structural (static) analysis

With our walkthrough, we would need to modify the Branch class by adding the instance variable to keep track of its manager, as shown in Figure 3.23.



**Figure 3.23** The class Branch modified as a result of our walkthrough

In our class diagram, we will now add an arrowhead from Branch to Manager as shown in Figure 3.24. This arrowhead shows that the Branch class needs to keep track of a Manager object.

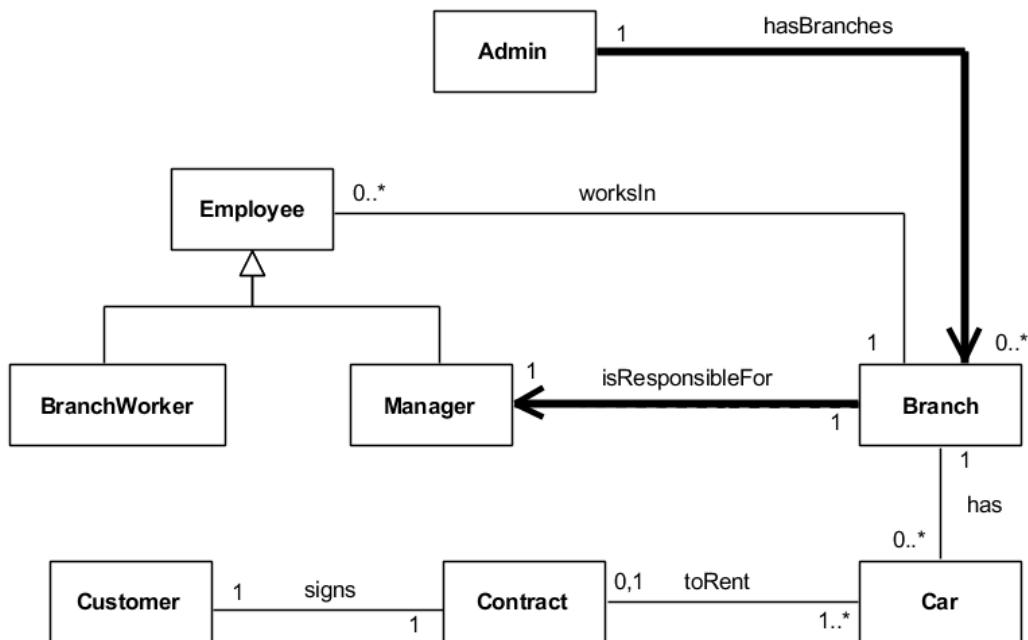


Figure 3.24 Another navigation arrowhead is added

Now, our final objective of the entire walkthrough is to get the particulars of the manager in charge of a branch. Therefore, we need to "move" over to the Manager object and use its methods to get its particulars. If you refer to the class description for the class Manager, these particulars are:

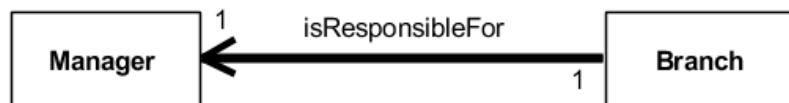
<b>Class:</b>	Manager, subclass of Employee
<b>Attributes:</b>	name, the name of the employee (from the superclass, Employee) employeeNumber, the employee number of the employee (from the superclass, Employee)

For this process, we say that we *navigate* from Branch to Manager.

Upon getting the Manager object, we can display its particulars. However, there are other considerations that we will examine further when we discuss integration with the user interface.

### 3.4.2 Implementing an Association

As a summary of what we have done for this step in the walkthrough, we re-draw the relevant portion of the class diagram shown in Figure 3.25, to emphasise what we are discussing. In particular, we have highlighted the navigation arrow for the association `isResponsibleFor` from `Branch` to `Manager`.



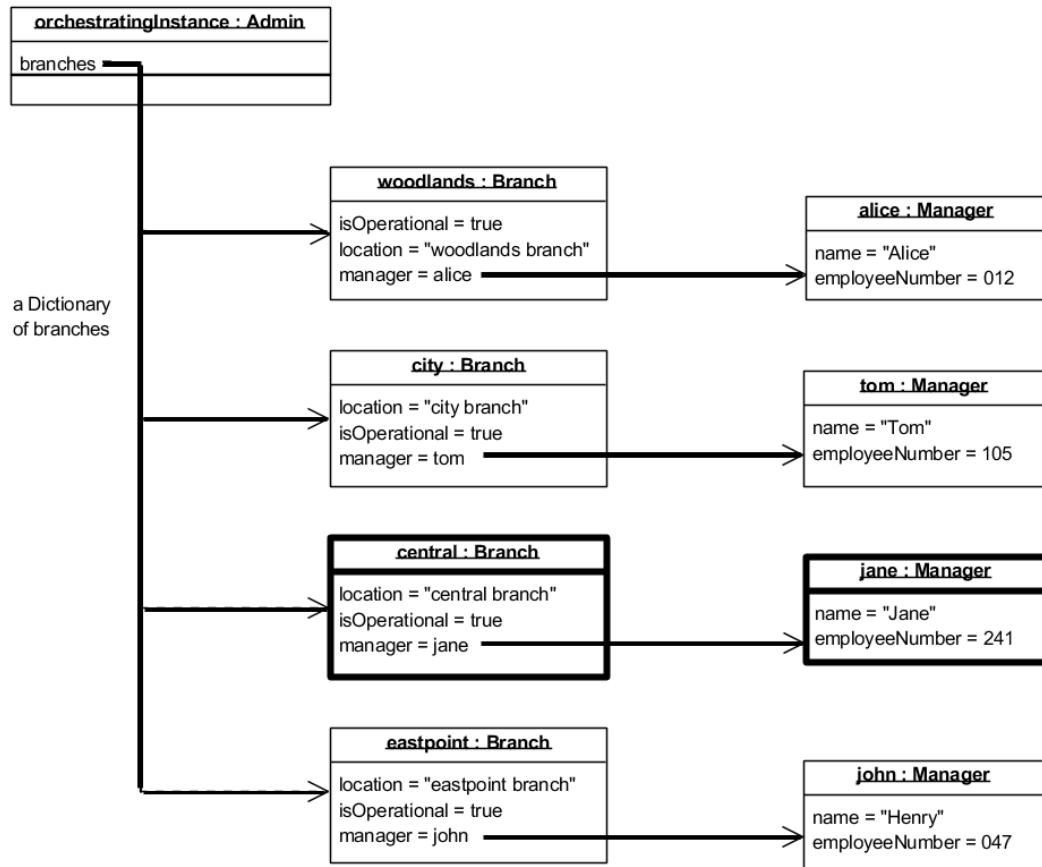
**Figure 3.25** A fragment of the class diagram showing the association being implemented

The following table summarises the essential points in the implementation of the association `isResponsibleFor`:

Class being modified	Branch
Association being implemented	<code>isResponsibleFor</code>
Direction of navigation	from Branch to Manager
Multiplicity	one to one
Instance variable added	<code>manager</code>
Value of the instance variable	a Manager object
Justifications	Since there is only one manager for a branch, an instance variable with a single value is enough. This variable would have a Manager object as its value.

Now that we have completed one walkthrough, observe how we have used the class diagram to help us navigate among the classes.

Next, look carefully at Figure 3.26 on how we are actually dealing with objects that represent the things we want from the application.



**Figure 3.26** How the application gets the particulars of a manager

Since no new association is navigated in this walkthrough, no new implementation of any association is required.



### Activity 3.1

Assume that we have completed the walkthrough to find the manager. Now, suppose we want to get the particulars of a car in a given branch instead of the manager.

- a. What could be the use case for such a situation?
- b. What is the additional information that you might need for this use case? Explain your answer. Draw the diagram showing how the data is passed from the external world to the classes in the application.
- c. How would the class diagram show the navigation that is required?
- d. Why is it that the question begins with "Assume that we completed the walkthrough to find the manager"? What difference does it make whether this is in the question?
- e. What method must be added to the class Admin as a result of this walkthrough?
- f. What is the instance variable that must be added to the class Branch?
- g. What kind of value would this instance variable have?
- h. What are the reasons for your answer to the previous question?
- i. Provide a summary of any association implemented as a result of this walkthrough. Use the format of the table shown under the subheading "Implementing an association".
- j. Provide a summary of the walkthrough using the format as shown under the subheading "Overall View".

## 3.5 Documenting Walkthroughs

After performing a walkthrough, it is a good practice to document the walkthrough so that we are ready for the next step – **visualising the walkthrough**. Let us do this with several walkthroughs.

### Example 1 – Walkthrough: To find the contract associated with a car

Assume that we have done the walkthrough given in Activity 1: "Find the particulars of a car given the branch location and the car registration number". We will perform a walkthrough to find the contract associated with a car.

This walkthrough is based on the following use case:

<b>Use case</b>	Find the contract associated with a car.
<b>Initiator</b>	A marketing staff.
<b>Objective</b>	To find the particulars of the contract for a car.
<b>Pre-conditions</b>	The location of the branch and the registration number of the car required must be given.
<b>Post-conditions</b>	The application will provide the particulars of the contract associated with a specified car.
<b>Assumptions, Notes or constraints</b>	Assume that the car is in the branch and that the cars are identified by their registration numbers.

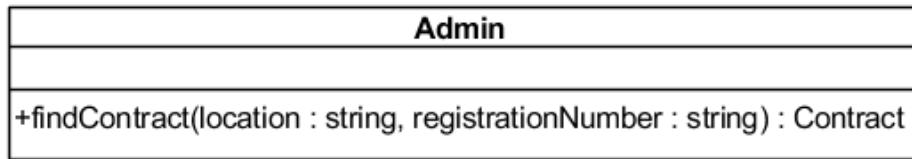
This documentation below summarises how the walkthrough is accomplished:

**Objective:** To find the contract associated with a given car in a given location.

**Given:** A location of a branch, the registration number of a car

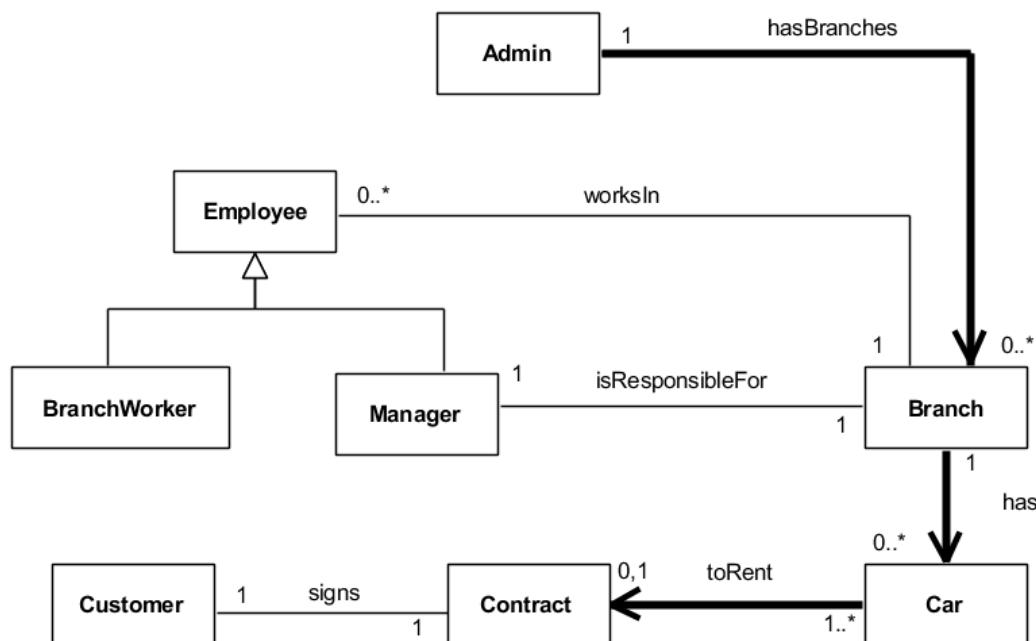
1. Locate the Branch object with the specified location, linked to the orchestrating object via the association hasBranches.
2. Locate the Car object linked via the association has to the Branch object found in step 1.
3. Locate the Contract object linked via the association toRent to the Car object found in step 2.
4. Show the particulars of the Contract object.

Figure 3.27 shows the method that would need to be added to the orchestrating class.



**Figure 3.27** The method to be added to the orchestrating class

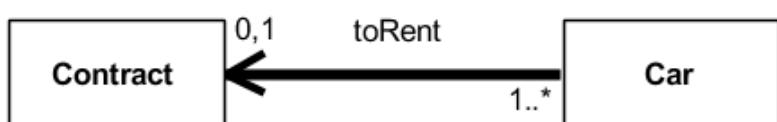
Note that no further information is needed to find the contract once we have the car required. This is because there is at most only one contract for a car, as shown by the multiplicity in Figure 3.28.



**Figure 3.28** The walkthrough to find the contract for a given car

If the car is not rented out with a contract, there would be no contract associated with the car. The application should return a null object in response to the user request.

The following shows the new association that should be implemented:



<b>Class being modified</b>	Car
<b>Association being implemented</b>	toRent
<b>Direction of navigation</b>	from Car to Contract
<b>Multiplicity</b>	many to one (many to zero has no meaning)
<b>Instance variable added</b>	contract
<b>Value of the instance variable</b>	either null or a Contract object
<b>Justifications</b>	Since there is only at most one contract for a car, an instance variable with a single value is enough. This variable would have a Contract object as its value.

### Example 2 – Walkthrough: To assign a new manager to a branch

Assume that we have done the walkthrough given in Activity 1: "Find the particulars of a car given the branch location and the car registration number". We will perform a walkthrough to assign a new manager to a branch.

The class description (from Unit 2) shows that a manager has the following attributes:

<b>Class:</b>	Manager, subclass of Employee
<b>Attributes:</b>	name, the name of the employee (from the superclass, Employee)  employeeNumber, the employee number of the employee (from the superclass, Employee)

We will assume that both attributes, name and employeeNumber, are required when a new manager is created.

This walkthrough is based on the following use case:

Use case	Assign a new manager to a branch.
Initiator	An administrative staff.
Objective	To assign a new manager to a branch.
Pre-conditions	The location of the branch and the relevant particulars (name and employee number) of the new manager must be given.
Post-conditions	The branch would have a new manager.
Assumptions, Notes or constraints	None.

In this walkthrough, we will assign a new manager to a branch identified by its location. The following documents the walkthrough required:

**Objective:** To assign a new manager to a branch.

**Given:** A location of a branch, the name and employee number of the new manager.

1. Locate the Branch object with the specified location, linked to the orchestrating object via the association hasBranches.
2. Create the new Manager object with the given name and employee number.
3. Create the association between the Branch object found in step 1 and the new Manager object created in step 2.

Figure 3.29 shows the method that would be added to the orchestrating class.

```

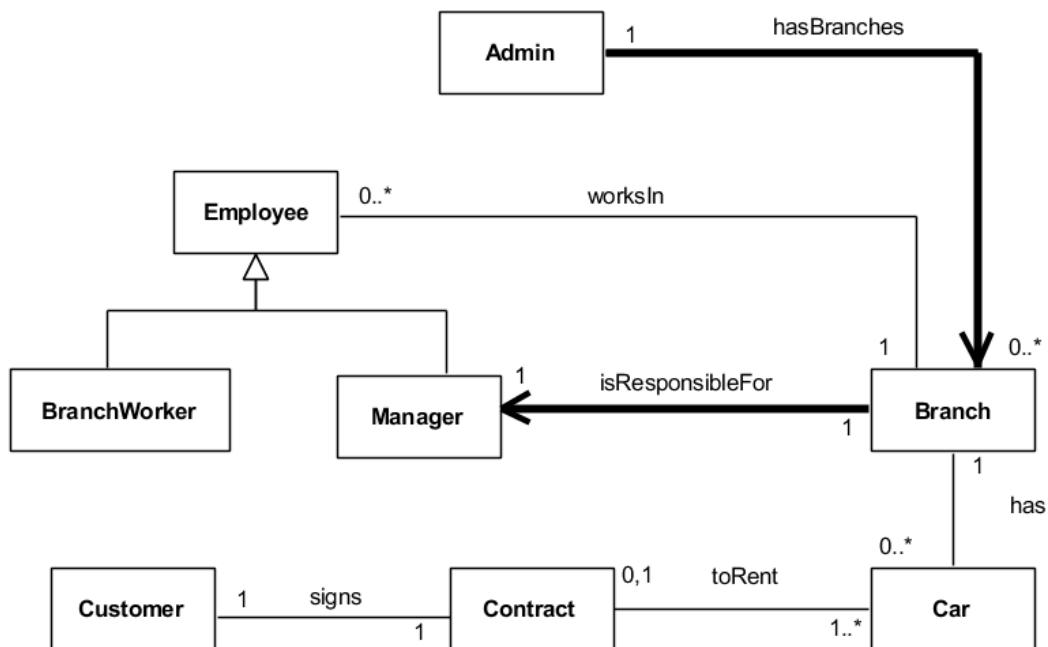
Admin
+findManager(location : string) : Manager
+findCar(location : string, registrationNumber : string) : Car
+findContract(location : string, registrationNumber : string) : Contract
+assignNewManager(location : string, employeeName : string, employeeNumber : integer)

```

**Figure 3.29** The method to be added to the orchestrating class

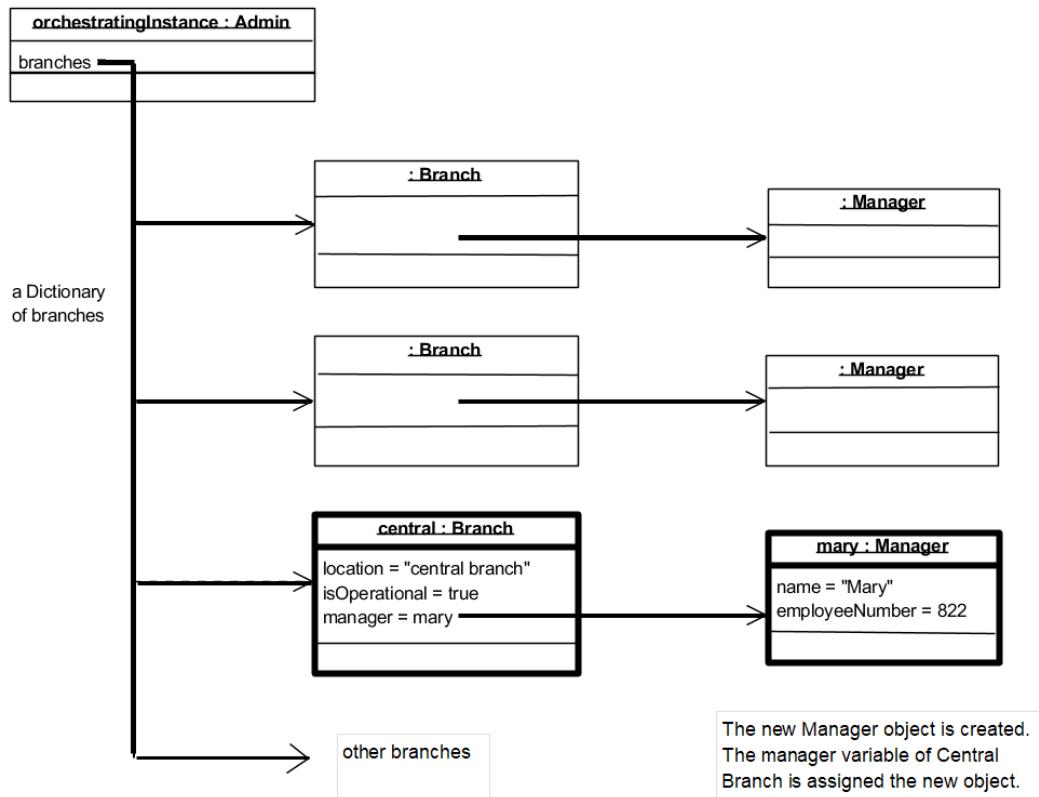
The class association diagram in Figure 3.30 shows the navigation required.

Again, we have shown all the navigation arrows derived so far: Admin to Branch and Branch to Manager. However, only the navigation Admin to Branch is relevant to us in this walkthrough. Our navigation need not proceed on to the Manager class. This is because in this walkthrough, we are creating a new Manager object and assigning it to the instance variable in the class Branch. We are not navigating to the Manager object for any purpose.



**Figure 3.30** The walkthrough to assign a new manager to a branch

Figure 3.31 below shows a scenario where the Central Branch is assigned a new manager, Mary, with employee number 822.



**Figure 3.31** Assigning a new Manager object to a Branch object

Since no new association is navigated in this walkthrough, no new implementation of any association is required.



## Activity 3.2

In our discussion, we need to create a new Manager object because we are having a new manager. This is shown in step 2 of the use case (see below). Now, why is step 3 necessary? After all, the class association diagram has already shown the association between the class Branch and Manager.

1. Locate the Branch object with the specified location linked to the orchestrating object via the association `hasBranches`.
2. Create the new Manager object with the given name and employee number.

3. Create the association between the Branch object and the new Manager object.



### Activity 3.3

Assume that you have done the walkthrough given in Activity 1: "Find the particulars of a car given the branch location and the car registration number". Perform a walkthrough to find the cars rented by a customer.

## 3.6 Bi-directional Associations

As the result of two previous walkthroughs "To find the contract associated with a car", and "Find the cars rented by a customer (Activity 3)", the association toRent becomes bi-directional.

The bi-directional association and the instance variables introduced in both classes are shown below.



Class being modified	Contract	Car
Association being implemented	toRent	toRent
Direction of navigation	From Contract to Car	from Car to Contract

Multiplicity	one to many	many to one
Instance variable added	cars	contract
Value of the instance variable	a Set object	either null or a Contract object

It is common in many texts to omit the two arrowheads if the association is bi-directional. However, this might be confusing if we do not keep in mind whether the class diagram is showing the structural (static) model or a dynamic model. A structural model does not have any navigation arrows in the first place. Therefore, for our purpose, we will retain the two arrowheads to represent a bi-directional association.

### 3.6.1 Consistency in Bi-directional Associations

As some of you may have noticed, a bi-directional navigation imposes additional care in developing the application. The objects of the two classes must ensure that the values of their instance variables are consistent.

Take, for example, the association between a manager and the branch under his charge. Suppose this association is bi-directional as shown in the Figure 3.32.

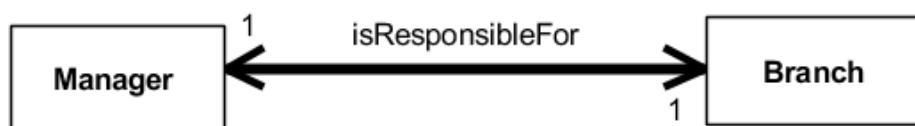
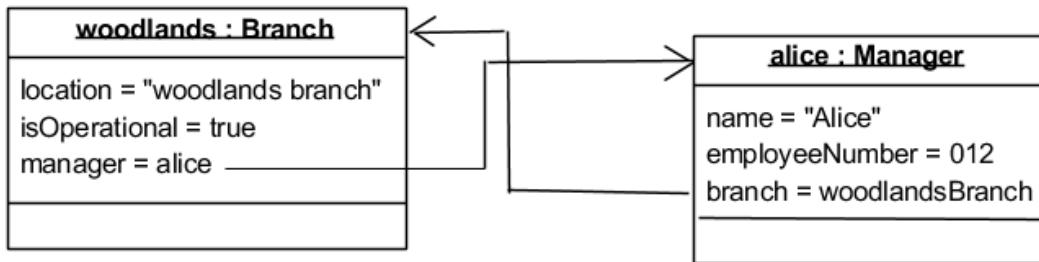


Figure 3.32 Illustrating a bi-directional association

In terms of actual objects, Figure 3.33 shows one particular possibility where Alice is the branch manager for Woodlands Branch:



**Figure 3.33** A bi-directional association has to be consistent

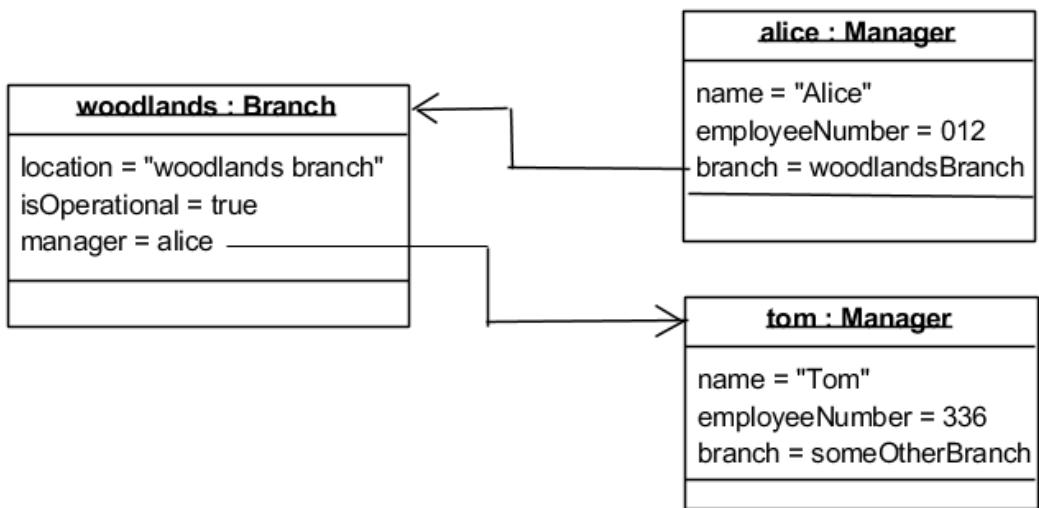
For the association to be consistent, the following must be true:

- The Woodlands Branch has Alice as the manager.
- Alice is the manager of Woodlands Branch.

The association would be inconsistent if it shows the following, for example:

- The Woodlands Branch has Tom as the manager.
- Alice is the manager of Woodlands Branch.

The object diagram in Figure 3.34 below illustrates this inconsistency:



**Figure 3.34** An inconsistent bi-directional association

Therefore, we need to ensure that the association is consistent when we implement the application. We will return to this again when we discuss implementation.

Even at this stage, some of you might notice that if we have redundant associations among the classes, it gets unnecessarily complicated since we need to ensure that the associations are mutually consistent.



## Activity 3.4

Assume that you have done the walkthrough given in Activity 1: "Find the particulars of a car given the branch location and the car registration number". The following is a use case for the car rental application.

The walkthrough will be based on the following use case:

Use case	List the employees working with a manager.
Initiator	An administrative staff.
Objective	To display the names of the employees working with a manager in a branch.
Pre-conditions	The name of the manager must be given.
Post-conditions	A display of the names of employees working with a manager.
Assumptions, Notes or constraints	The name is a valid name of a manager.

Perform a walkthrough for the use case and show the following:

- a. A summary of the steps in the walkthrough
- b. The method that has to be added to the orchestrating class
- c. The class association diagram with the navigation required
- d. The associations that have to be implemented

## Chapter 4: Visualising Walkthroughs

### 4.1 Analysis with Sequence Diagrams

Our first analysis is to display the particulars of a manager in charge of a branch, given the location of the branch. We have already done this in the previous section with a walkthrough. We have used that analysis to add the instance variables for implementing the associations.

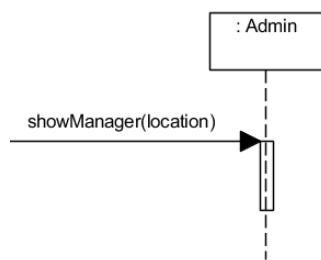
This time round, our intention is to identify all the methods needed as a result of the walkthrough. The result of the previous walkthrough is given below.

**The Walkthrough: Find the manager in charge of a branch when given the branch location**

As a result of this walkthrough, the orchestrating object would need to accept a message such as showManager(location) from the user interface. This means that the orchestrating class would need to have the implementation of the method showManager(location).

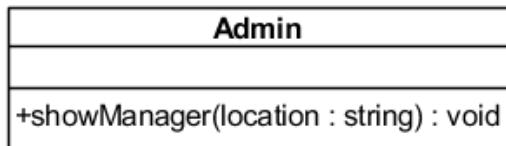
We will use a sequence diagram to show the message being sent to the orchestrating object. We will construct this sequence diagram in stages, as we go through the steps of the walkthrough.

#### 4.1.1 The Methods Required



The first part of the sequence diagram is shown in the diagram on the right.

The class diagram in Figure 3.35 shows the method that we would need for the orchestrating class, Admin.



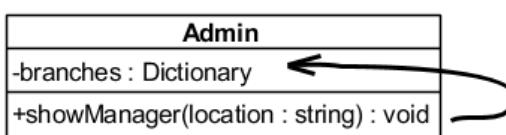
**Figure 3.35** The design for the orchestrating class so far

In order to let us keep things manageable and stay focused, we will omit the message replies at this stage. For this reason, we let the return type of the method be void. It will simply cause the manager particulars to be displayed. We will return to examine message replies later.

We shall next go through each step of the walkthrough again to trace the rest of the messages and methods required to accomplish the use case objective.

**Step 1:** Locate the Branch object with the specified location, linked to the orchestrating object via the association hasBranches.

Recall that the orchestrating class has the instance variable branches to keep track of all the branches. The Branch objects are referenced in a Dictionary collection since there are many branches. Therefore, to achieve step 1, the Admin object has to search the Dictionary collection of Branch objects in its branches instance variable.



**Figure 3.36** The method in the orchestrating class will look for the Branch object

Figure 3.37 shows how we draw the sequence diagram showing the Admin object sending itself a message to get the branch using the given location. The expression

`aBranch = findBranch(location)`

shows that the variable `aBranch` will reference the result returned by the method `findBranch(location)`. In other words, the variable `aBranch` will reference the `Branch` object that we are looking for.

From this, we conclude that the class `Admin` would need another method as shown in Figure 3.38.

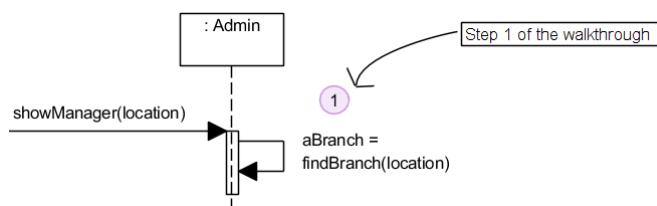


Figure 3.37 An object sending a message to itself

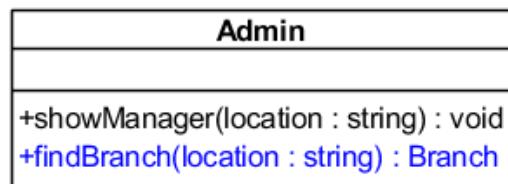
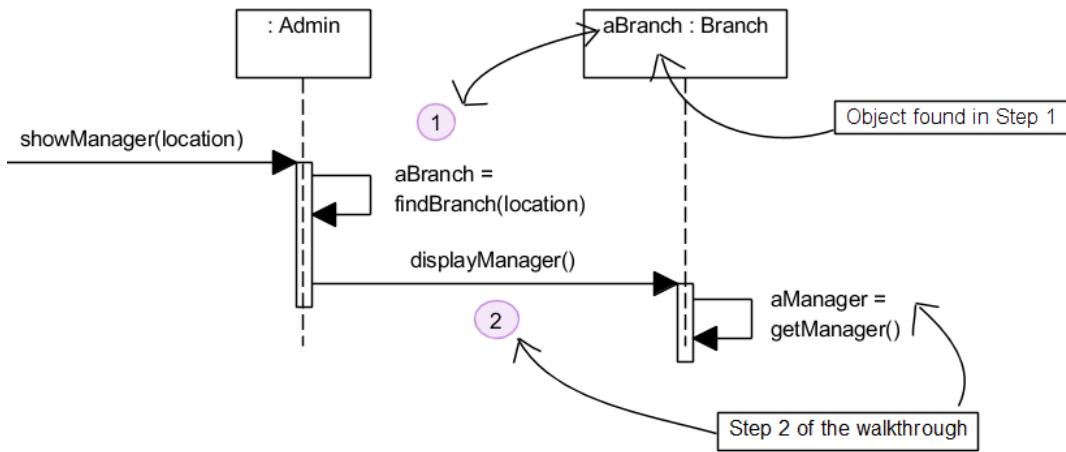


Figure 3.38 The design for the orchestrating class so far

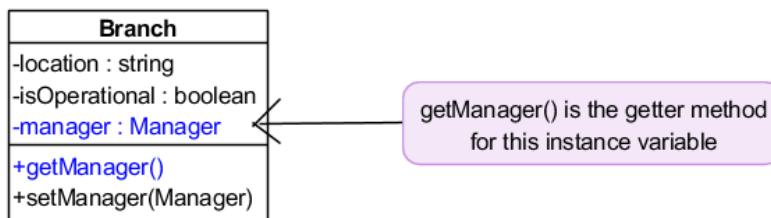
**Step 2:** Locate the Manager linked to the Branch object via the association `isResponsibleFor`.

In step 1, we have located the `Branch` object using the given location. The next step is to send this object a message asking for its manager. This can be conveniently shown in the sequence diagram in Figure 3.39.



**Figure 3.39** Sending a message to the Branch object that has been located

This tells us that the Branch object needs to understand the `displayManager()` message. This message will make use of `getManager()` to look for the manager of the branch. You might notice that this is simply the get message of the class for its instance variable `manager`. Figure 3.40 shows the class diagram reproduced from Unit 2.



**Figure 3.40** The getter method for the instance variable manager

### Step 3: Show the particulars of the Manager object.

The message expression `aManager = getManager()` in step 2 shows that the outcome will be the Manager object of the branch. The variable `aManager` is used to reference this Manager object.

The next step would be to get this Manager object to display its particulars. The sequence diagram in Figure 3.41 shows the message that is sent to the object. For the purpose of illustration, we call this message `display()`. This indicates to us that we need to have the method `display()` in the class Manager.

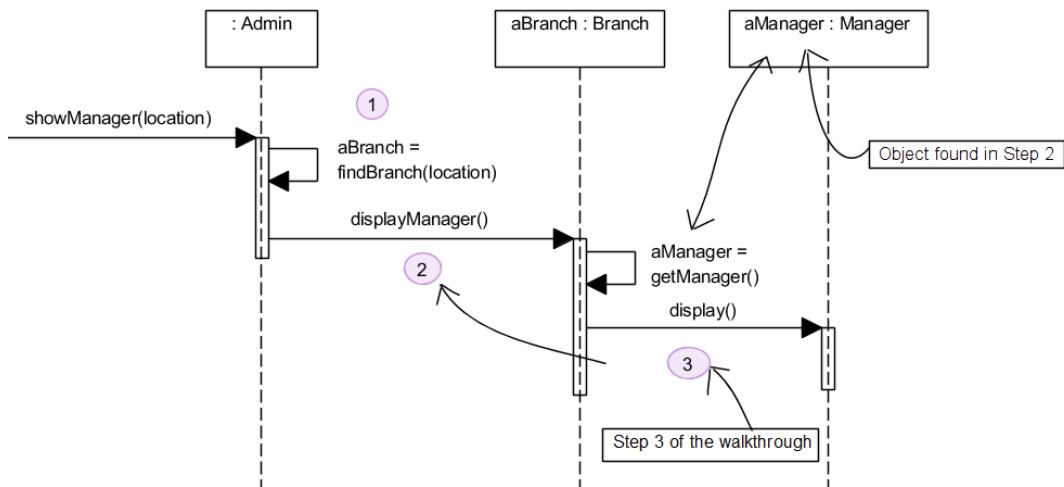


Figure 3.41 A message for the Manager object in Step 3

Examine the sequence diagram again. Observe how the message `displayManager()` invokes the messages `getManager()` and `display()`.

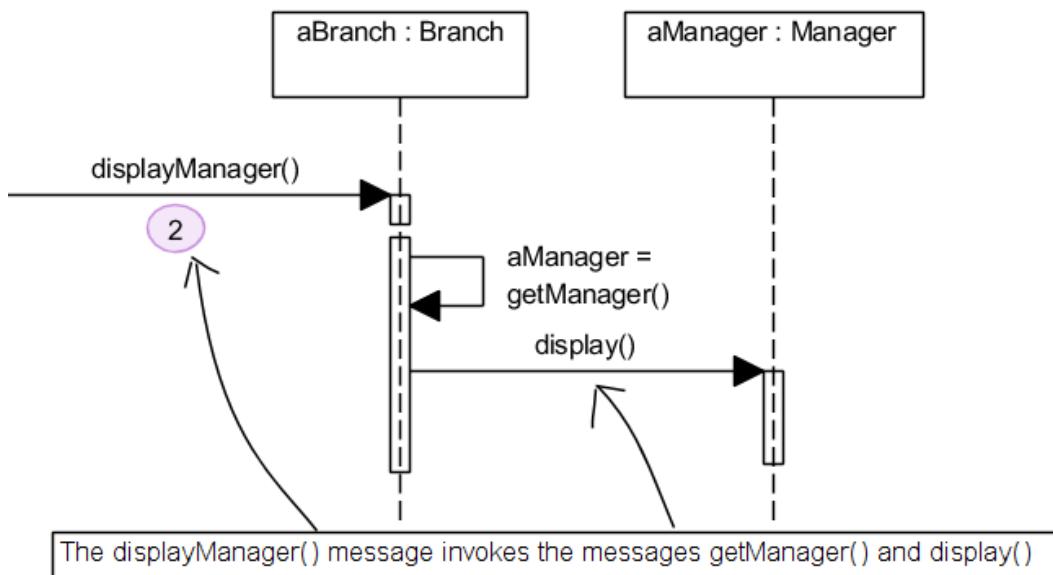


Figure 3.42 Messages in the sequence diagram

#### 4.1.2 The Results of Analysis

As a result of this analysis with the sequence diagram, we conclude that the following methods (in UML style notation) will be needed:

<u>Class</u>	<u>Methods</u>
<b>Admin</b>	showManager(location : string) : void findBranch(location : string) : Branch
<b>Branch</b>	displayManager() : void getManager() : Manager
<b>Manager</b>	display() : void

#### 4.1.3 Collaborations

For step 2, the sequence diagram shows that the Branch object has to carry out the task for the message displayManager(). In order to do so, it sends the message display() to the Manager object.

In effect, the Branch object is asking the Manager object to help in accomplishing its task. We say that the Manager object is a collaborator for the Branch object.

We can also say that the Manager object is providing a service to the Branch object. Thus, we can also call the Manager object the server and the Branch object the client.

Collaborators, clients and servers are roles that the objects play. These roles depend on messages being sent. An object can be a client for one message but a server for another message.



### Activity 3.5

So far in our discussions, we have omitted the message replies in order to focus on the main passing of messages. In Figure 3.42, how would the replies look like if they are added to the diagram?

## 4.2 Message Passing between Objects

In this section, we will take a look at the **messages and replies passed among the objects**.

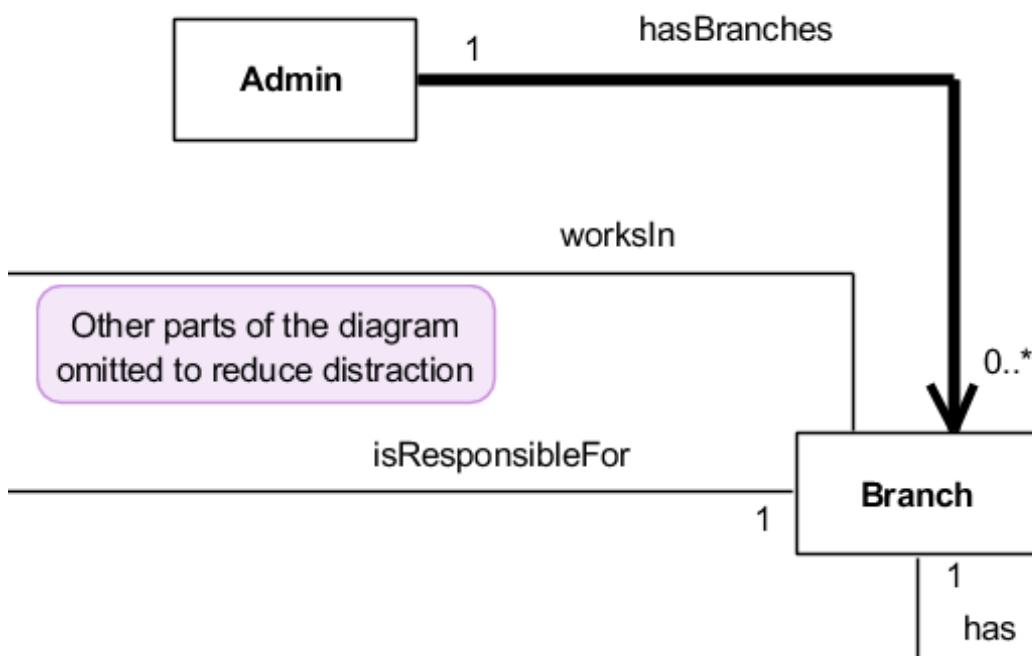
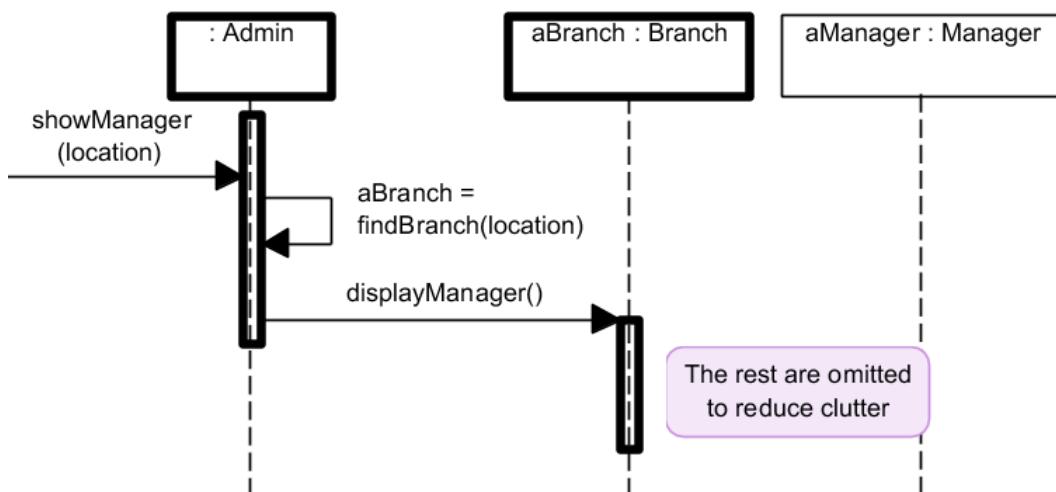


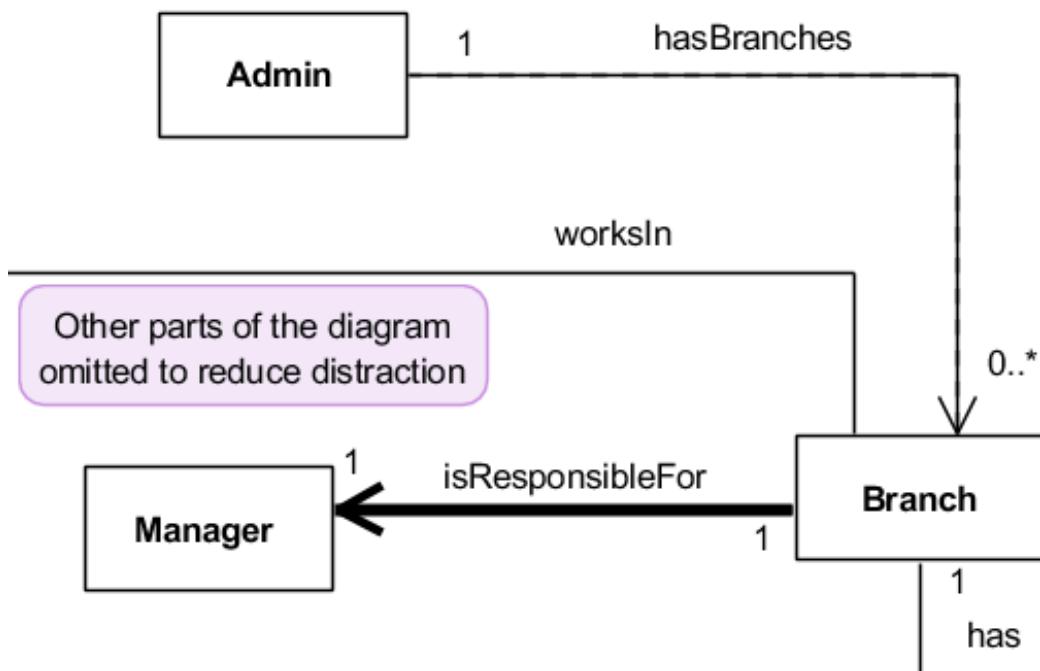
Figure 3.43 Association between two classes

The partial class diagram above highlights the **association between the classes Admin and Branch**. The sequence diagram in Figure 3.43 shows a **message passed from the Admin object to a Branch object**. The **message passing is possible because the Admin object is associated with (knows about) the Branch object**.



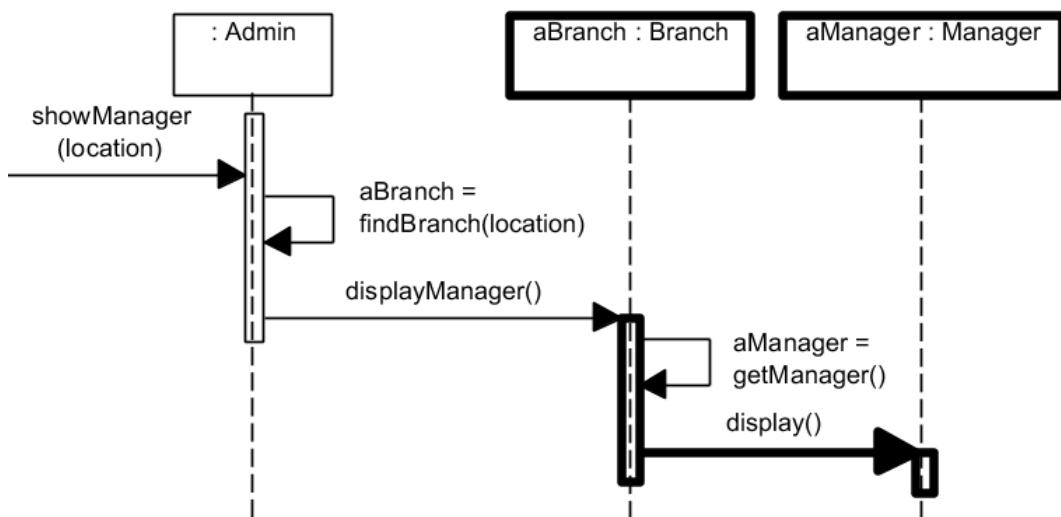
**Figure 3.44** Message passing between two objects

Similarly, the association between classes Manager and Branch shows that messages can be passed between a Manager object and the Branch object that the manager is in charge of.



**Figure 3.45** Association between two classes

The sequence diagram in Figure 3.46 shows a message passed from the Branch object to its Manager object. Again, the message passing is possible because the Branch object is associated with (knows about) its Manager object.

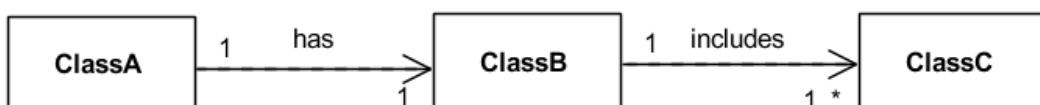


**Figure 3.46** Message passing between two objects

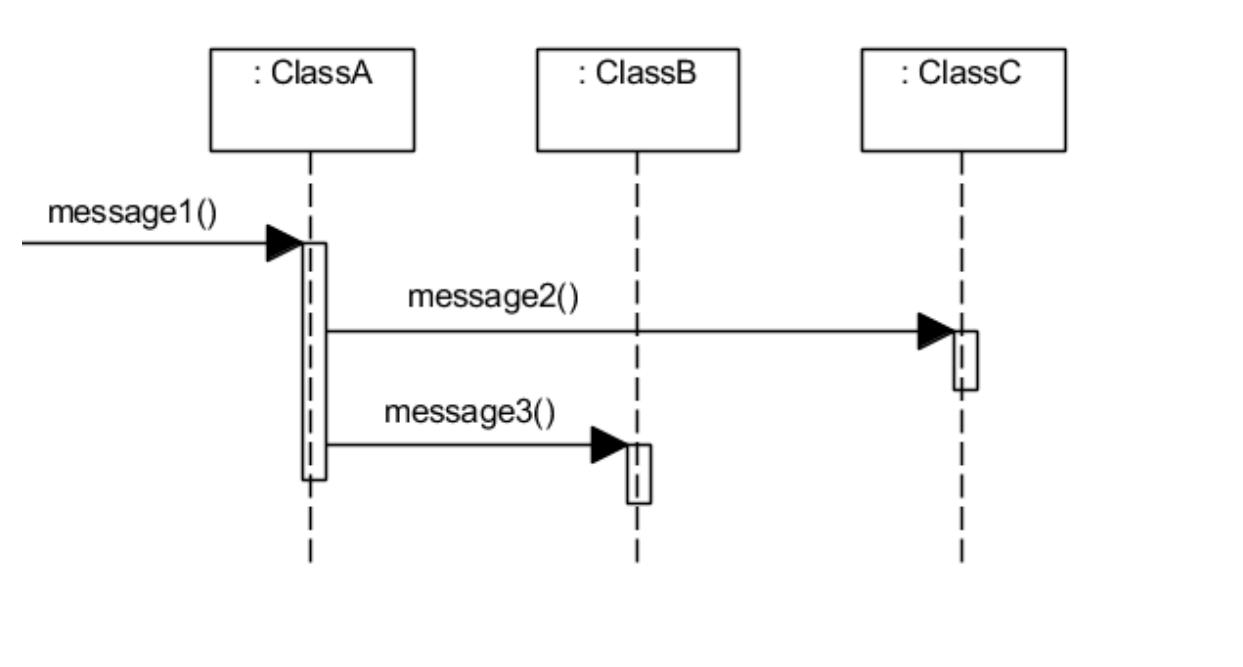


### Activity 3.6

Suppose we are given the following class association diagram:



Assuming that the class diagram given is not to be modified, briefly explain which of the following message passings is not possible.



#### 4.2.1 The Message Replies

In our discussions so far, we have not shown the message replies in the sequence diagram. We have simplified our use case to stay focused and avoid distractions as far as possible. Thus, we have left out the message replies and assume that the method in the orchestrating class `showManager()` has the type `void`.

Now that we have completed one analysis with the sequence diagram, we can explore the situations where there are message replies and we might wish to show them in the sequence diagram.

We will look at two variations of the use case:

**Variation 1:** The use case is required to return the *manager* in charge of the branch with the specified location. The following shows the relevant parts of the use case:

**Use case**

Find the *manager* of a branch.

**Objective**

To get the *particulars of the manager* in charge of a branch.

**Post-conditions**

The application will provide the *particulars of the manager* of the branch specified.

**Variation 2:** The use case is required to return the *name* of the manager in charge of the branch with the specified location. The following shows the relevant parts of the use case:

**Use case**

Find the *name* of the manager of a branch.

**Objective**

To get the *name* of the manager in charge of a branch.

**Post-conditions**

The application will provide the *name* of the manager of the branch specified.

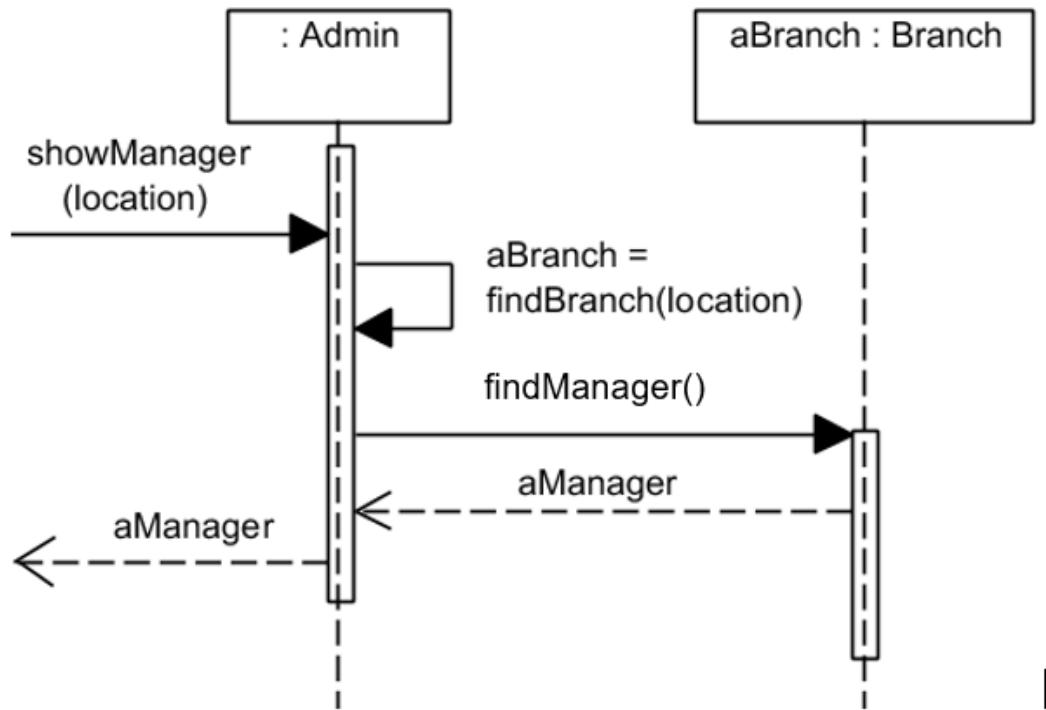
The first variation returns the entire Manager *object* whereas the second variation is required to return only the *name* of the manager.

The return type of the method to be implemented in the Admin class will need to reflect this accordingly. Figure 3.47 shows the two versions of the class diagram for Admin.



**Figure 3.47** The two variations of the class Admin

Figure 3.48 shows the sequence diagram for Variation 1: getting the Manager object, aManager.



**Figure 3.48** The sequence diagram for Variation 1

This diagram shows the `findManager()` message sent from the Admin object to the Branch object. This message would invoke the corresponding `find` method of Branch. The getter method for its instance variable `manager` would return the Manager object that we need.

Compare this sequence diagram with the previous one and note the difference. In our case here, the get message `getManager()` is adequate to do the job for us.

Figure 3.49 shows the sequence diagram for *Variation 2*: getting the name of the manager.

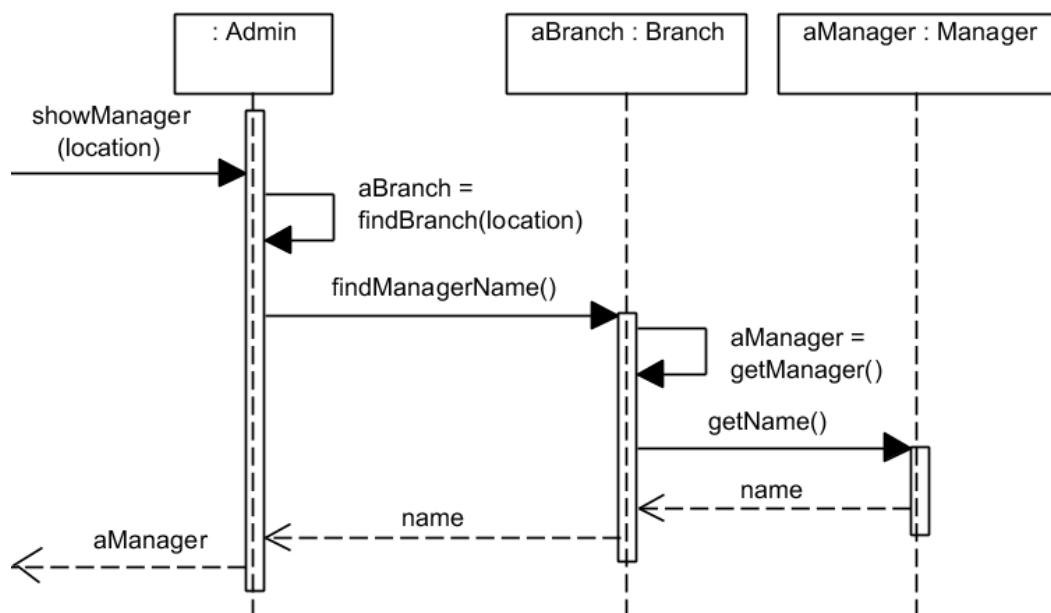


Figure 3.49 The sequence diagram for Variation 2

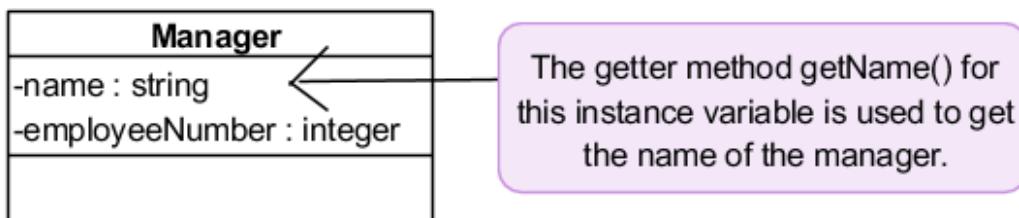


Figure 3.50 The getter method for the instance variable name

At this stage, we have completed our discussions on **dynamic modelling** based on use cases and have focused on explaining the **mechanics of modelling**. In the next chapter, we will model creation and destruction of objects and the behaviour of objects based on events.



### Activity 3.7

Describe the messages in the sequence diagram describing data collection of the weather station system.



### Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*, Chapter 7, Section 7.1.4, pp.204-207.

## Summary

The following points summarise the contents of this study unit:

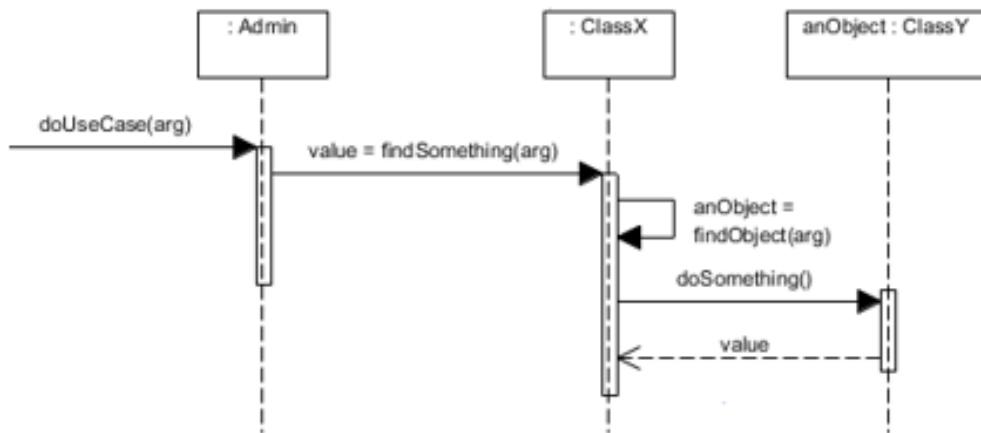
- The objective of performing a walkthrough is to document a process. It is summarised with a series of steps to document what needs to be done when a use case is executed.
- The required documents required for a walkthrough are the use case diagram, the class association diagram and the class description.
- The role of an orchestrating class is to orchestrate the behaviour of a use case and channel requests for actions to the participating classes in the application. The orchestrating instance could send messages to one or more objects as required.
- The side effects of a walkthrough are that the class description will be updated according to the outcome of the analysis done.
- After a walkthrough is completed, there would be addition of instance variables and methods to some participating classes.
- Walkthroughs for a bi-directional association would introduce new instance variables in both classes in the association.
- The purpose of a sequence diagram is to model interactions between active objects in a system over time.
- The elements of a sequence diagram are lifeline, activations, messages and objects.
- A sequence diagram is a visual representation of a walkthrough.
- A sequence diagram can be analysed for object collaborations which results in identifying methods in participating classes through message passing between objects.

## Formative Assessment

1. What are domain models?
  - a. a computation independent model
  - b. a model that uses UML models to show the structural model of a system
  - c. a platform specific model
  - d. a platform independent model
2. Pick the element that is not found in a sequence diagram.
  - a. Classes
  - b. Lifelines
  - c. Messages
  - d. Activation rectangles
3. What is the purpose of a walkthrough?
  - I. To work out the behaviour of objects interacting among themselves.
  - II. To model the behaviour of one application function.
  - III. To add new instance variables to the classes in the class association diagram.
  - IV. To add new methods to the classes in the class association diagram.
    - a. I and II only
    - b. III and IV only
    - c. I, III and IV only
    - d. I, II, III and IV
4. Which of the following is required before a walkthrough can be performed?
  - I. Use case diagram
  - II. Class diagram
  - III. Class description
  - IV. Activity diagram

- a. I only
- b. II and III only
- c. I, II and III only
- d. II, III and IV only

5. Which methods belong to the class ClassX?



- I. doUseCase(arg)
- II. findSomething(arg)
- III. findObject(arg)
- IV. doSomething()

  - a. I only
  - b. II only
  - c. II and III only
  - d. II, III and IV only

6. Which one of the following statements about the orchestrating class is incorrect?

- a. The orchestrating class channels inputs from the user interface to the classes in the application.
- b. There is a different orchestrating object for each walkthrough.
- c. A walkthrough always starts with the orchestrating class.

- d. A walkthrough is an operation in the orchestrating class.
7. For the following fragment of a class diagram, why is special care required for the bi-directional association `consistsOf` during walkthroughs?



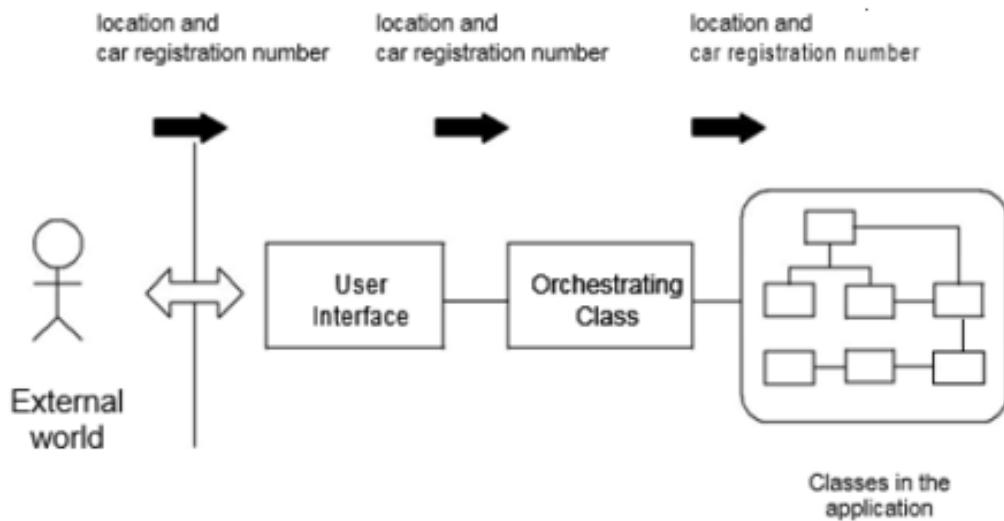
- a. This is because the bi-directional association is not shown in the (structural) class association diagram.
- b. This is so as to determine the value of instance variables to be added to each class.
- c. This is so as to determine the operations that are to be added to each class.
- d. For consistency

# Solutions or Suggested Answers

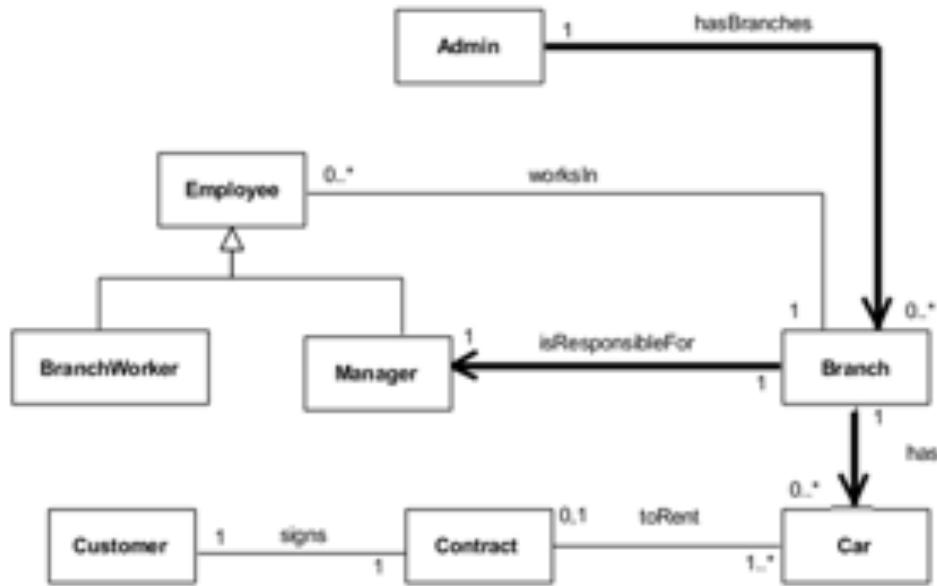
## Activity 3.1

- a. Use case      Find the particulars of a car.
- Initiator      A marketing staff.
- Objective      To get the particulars of a car in a given branch.
- Pre-conditions      The location of the branch and the registration number of the car required must be given.
- Post-conditions      The application will display the particulars of a car in the branch specified.
- Assumptions, Notes or constraints      Assume that the car is in the branch and that the cars are identified by their registration numbers.
- b. The additional information is the registration number of the car. A branch is expected to have many cars. These cars would normally be identified by their registration number.

The following shows the data being passed to the application:



- c. The class diagram showing the navigation required is shown below. Notice that we have retained the navigation arrow from Branch to Manager in the diagram.



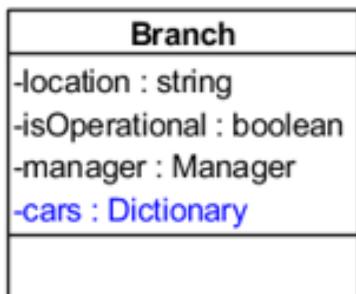
- d. The differences that this phrase makes are:

- The navigation arrow from Branch to Manager would not be present in the class diagram if that "find manager" walkthrough is not done yet.

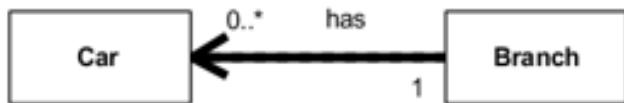
- ii. The instance variable branches in Admin would need to be added in this walkthrough if that "find manager" walkthrough is not done yet.
- e. The method that would be required is shown below. This method is required for the Admin class to understand a request (accept a message) for the particulars of a car.



- f. The following shows the instance variable that would be required. This instance variable enables a Branch object to keep track of the cars in the branch.



- g. The instance variable would have a Dictionary object as its value.
- h. The reasons are:
  - a. The instance variable has to have many values, each of which is a Car object. Therefore, we would need a collection object.
  - b. We need to locate a car by its registration number. Therefore, a collection object such as a Dictionary would be appropriate.
    - i. The registration numbers would be used as keys in the Dictionary since they are used to locate the cars.
    - ii. The values of the Dictionary would be Car objects since we are keeping track of cars.
- i. The association implemented is:



Class being modified	Branch
Association being implemented	has
Direction of navigation	from Branch to Car
Multiplicity	one to many
Instance variable added	cars
Value of the instance variable	a Dictionary of Car objects, using their registrationNumber as the key
Justifications	<p>An instance variable is added to the class Branch because it has to keep track of its cars.</p> <p>A Dictionary is used because</p> <ul style="list-style-type: none"> <li>a. The multiplicity is one to many. The Branch object has to keep track of many cars.</li> <li>b. The application needs to look for a car by using its registration number. Thus, the registration number is used as the key in the Dictionary.</li> </ul>

- j. The summary of the walkthrough:

Objective: To find the particulars of a car in a given branch

Given: A location of a branch, the registration number of a car

- a. Locate the Branch object with the specified location, linked to the orchestrating object via the association hasBranches.
- b. Locate the Car object linked to the Branch object via the association has.
- c. Show the particulars of the Car object.

As a result of the walkthrough, we have added an instance variable cars to the class Branch.

## Activity 3.2

Step 3, creating the association between the Branch object and the new Manager object is necessary because associations are between objects, not classes. We have emphasised this previously when we gave the following advice:

- We will need to use the class diagram for our modelling. The class diagram from the structural model shows us the static design of the system. It will provide us the guideposts to find our way around.

Remember that this is the reason why we are using the class diagram.

- When we do the modelling, remember that we are dealing with an application that is executing to provide the function desired. Thus, we are dealing with actual objects even though we are using the class diagram to show us the way.

Therefore, although the class association diagram has already shown the association between the class Branch and Manager, we need to create the association between the actual objects: the new Manager object and the existing Branch object.

### Activity 3.3

In this walkthrough, we want to list the particulars of all the cars rented by a customer, given the name of the customer.

The class description shows that a car has the following attributes.

Class:	Car
Attributes:	engineCapacity, the engine capacity of the car registrationNumber, the vehicle registration number of the car

The walkthrough will be based on the following use case:

Use case	List the cars rented by a customer.
Initiator	A marketing staff.
Objective	To display some details of all the cars rented by a customer.
Pre-conditions	The name of the customer must be given.
Post-conditions	A display of the details of cars rented by the specified customer.
Assumptions, Notes or constraints	The name given belongs to an existing customer.

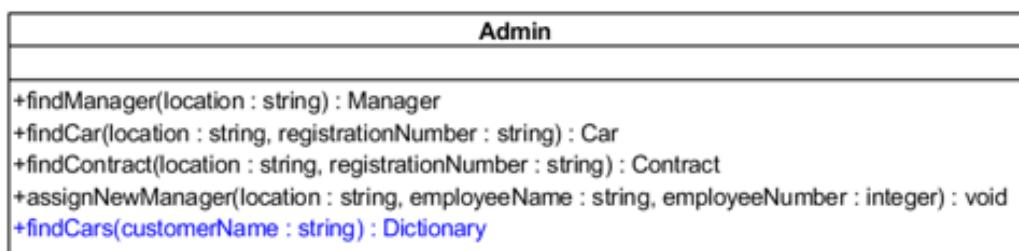
The following is a summary of the walkthrough required:

Objective: To list the details of the cars rented by a customer

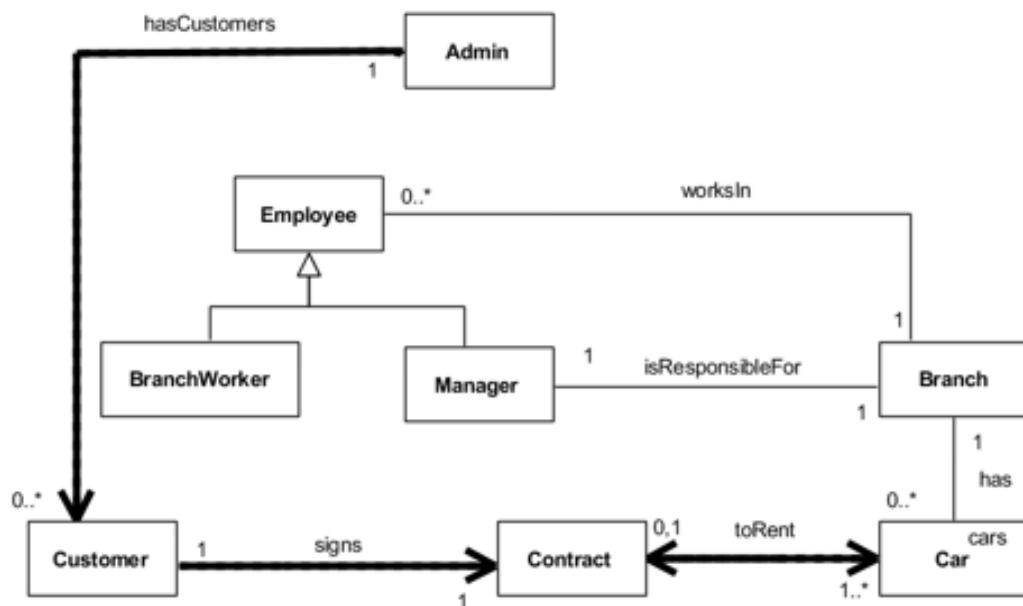
Given: the name of a customer

1. Locate the Customer object with the specified name, linked to the orchestrating object via the association hasCustomers. This is a new association that would have to be added between Admin and Customer.
2. Locate the Contract object linked via the association signs to the Customer object found in step 1.
3. Locate all the Car objects linked via the association toRent to the Contract object found in step 2.
4. Show the particulars of the Car objects found in step 3.

Arising from this walkthrough, the method that would be added to the orchestrating class could be (as shown in blue):



The class association diagram below shows the navigation required.

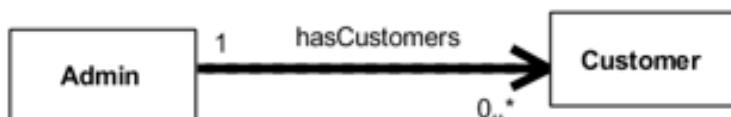


There are two points to note in this class diagram:

- We need to add the new association `hasCustomers` between the classes `Admin` and `Customer`. This is due to the fact that in this use case, we are given the customer name. From the orchestrating class `Admin`, we need to get the customer specified by this name.
- The association `toRent` now has two arrowheads. This shows that the association can be navigated in both directions. The navigation from `Car` to `Contract` arises from one of the previous use cases. The navigation from `Contract` to `Car` arises from the use case that we are currently discussing.

The multiplicity from `Admin` to `Customer` is one to many. This is because there is only one orchestrating instance but there should be many customers.

Next, we have to implement the associations for this use case. The first association to be implemented is `hasCustomers`, in the direction from `Admin` to `Customer`:



Class being modified	<code>Admin</code>
Association being implemented	<code>hasCustomers</code>
Direction of navigation	from <code>Admin</code> to <code>Customer</code>
Multiplicity	one to many
Instance variable added	<code>customers</code>
Value of the instance variable	a Dictionary object
Justifications	An instance variable is added to the class <code>Admin</code> because it has to keep track of customers.

	<p>A Dictionary is used because</p> <ul style="list-style-type: none"> <li>• The multiplicity is one to many. The Admin object has to keep track of many customers.</li> <li>• The application needs to look for a customer by using his or her name. Thus, the name is used as the key in the Dictionary.</li> </ul>
--	---

The next association that has to be implemented is signs, in the direction from Customer to Contract:



Class being modified	Customer
Association being implemented	signs
Direction of navigation	from Customer to Contract
Multiplicity	one to one
Instance variable added	contract
Value of the instance variable	a Contract object
Justifications	Since there is only at most one contract for a customer, an instance variable with a single value is enough. This variable would have a Contract object as its value.

The last association that has to be implemented is toRent, in the direction from Contract to Car:



Class being modified	Contract
Association being implemented	toRent
Direction of navigation	from Contract to Car
Multiplicity	one to many (zero to many has no meaning)
Instance variable added	cars
Value of the instance variable	a Set object
Justifications	<p>An instance variable is added to the class Contract because it has to keep track of cars.</p> <p>A Set is used because:</p> <ol style="list-style-type: none"> <li>1. The multiplicity is one to many. The Contract object has to keep track of many cars.</li> <li>2. The cars are unique objects.</li> <li>3. The application does not need to look for a particular car by using a key.</li> </ol> <p>Thus, a Dictionary is not required.</p>

Observe that we have used a Set as the collection object for this association. This is intentional. It shows that it is not always necessary that a Dictionary must be used. The appropriate collection object should be selected depending on the needs of the application.

Observe that we have used a Set as the collection object for this association. This is intentional. It shows that it is not always necessary that a Dictionary must be used. The appropriate collection object should be selected depending on the needs of the application.

We could be given the registration number of the car to look for it in the contract. With this use case, we would then need to change the collection into, say, a Dictionary:

Class being modified	Contract
Association being implemented	toRent
Direction of navigation	from Contract to Car
Multiplicity	one to many
Instance variable added	cars
Value of the instance variable	a Dictionary object
Justifications	<p>An instance variable is added to the class Contract because it has to keep track of cars.</p> <p>A Dictionary is used because:</p> <ol style="list-style-type: none"> <li>1. The multiplicity is one to many. The Contract object has to keep track of many cars.</li> <li>2. The application <b>has to look for a particular car by using its registration number</b>. Thus, a Dictionary is selected.</li> </ol>

## Activity 3.4

- a. The following is a summary of the walkthrough required:

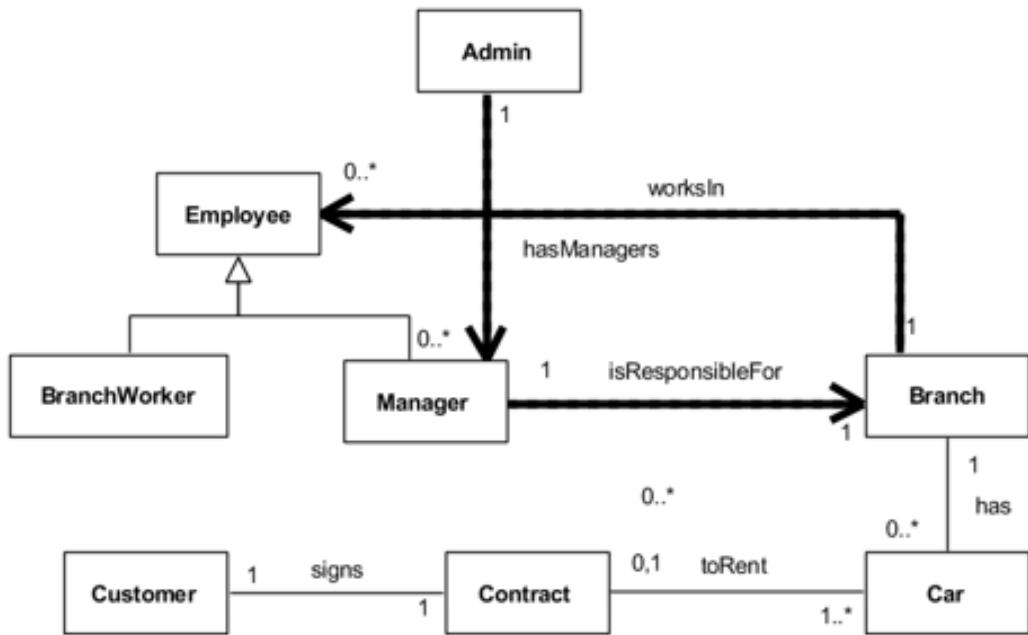
Objective: To list the employees working with a manager

Given: The name of a manager

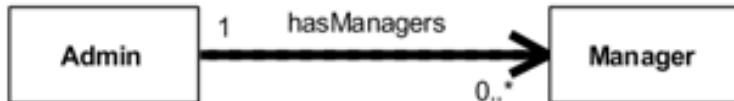
- a. Locate the Manager object with the specified name, linked to the orchestrating object via the association hasManager. This is a new association that would have to be added between Admin and Manager.
  - b. Locate the Branch object linked via the association isResponsibleFor to the Manager object found in step 1.
  - c. Locate all the Employee objects linked via the association worksIn to the Branch object found in step 2.
  - d. Show the names of the Employee objects found in step 3.
- b. Arising from this walkthrough, the method that would be added to the orchestrating class could be:

Admin
+findManager(location : string) : Manager +findCar(location : string, registrationNumber : string) : Car +findContract(location : string, registrationNumber : string) : Contract +assignNewManager(location : string, employeeName : string, employeeNumber : integer) : void +listCars(customerName : string) : Set +listEmployeeNames(managerName : string) : Set

- c. The class association diagram below shows the navigation required.

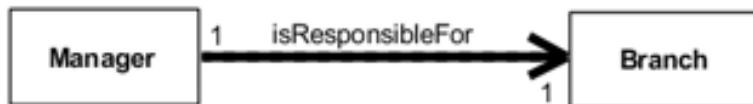


d. The association to be implemented are shown below:



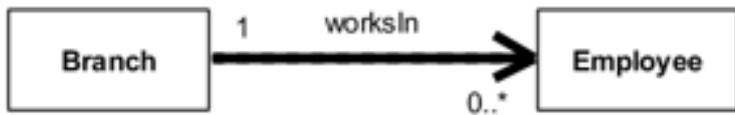
Class being modified	Admin
Association being implemented	hasManagers
Direction of navigation	from Admin to Manager
Multiplicity	one to many
Instance variable added	managers
Value of the instance variable	a Dictionary object

Justifications	<p>An instance variable is added to the class Admin because it has to keep track of managers.</p> <p>A Dictionary is used because:</p> <ul style="list-style-type: none"> <li>a. The multiplicity is one to many. The Admin object has to keep track of many managers.</li> <li>b. The application needs to look for a manager by using his or her name. Thus, the name is used as a key in the Dictionary.</li> </ul>
----------------	--



Class being modified	Manager
Association being implemented	isResponsibleFor
Direction of navigation	from Manager to Branch
Multiplicity	one to one
Instance variable added	branch
Value of the instance variable	a Branch object
Justifications	Since there is only at most one branch for a manager, an instance variable with a single

	value is enough. This variable would have a Branch object as its value.
--	---

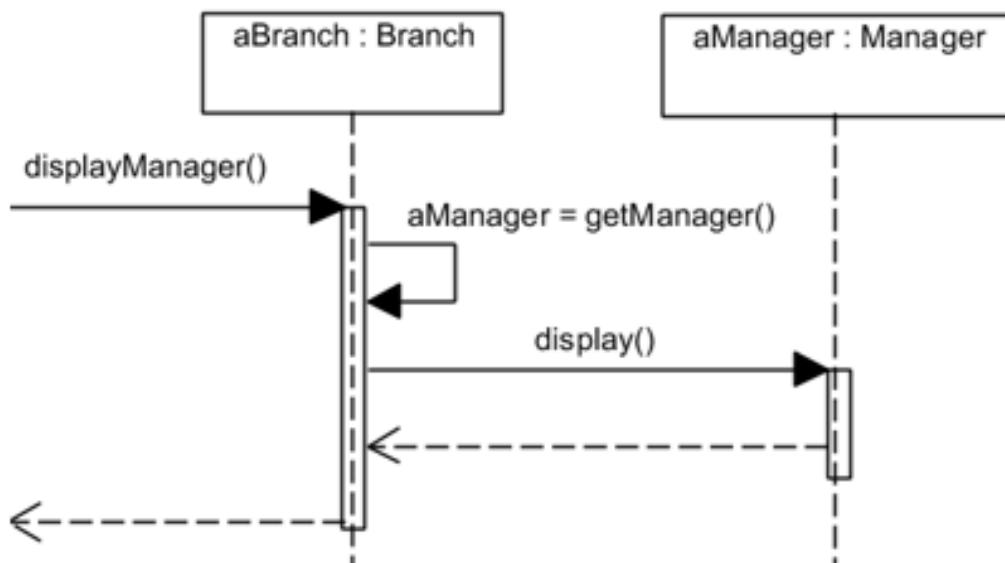


Class being modified	Branch
Association being implemented	worksIn
Direction of navigation	from Branch to Employee
Multiplicity	one to many
Instance variable added	employees
Value of the instance variable	a Set object
Justifications	<p>An instance variable is added to the class Branch because it has to keep track of employees.</p> <p>A Set is used because:</p> <ul style="list-style-type: none"> <li>a. The multiplicity is one to many. The Branch object has to keep track of many employees.</li> <li>b. The employees are unique objects.</li> <li>c. The application does not need to look for a particular employee by</li> </ul>

	using a key. Thus, a Dictionary is not required for this use case.
--	--

### Activity 3.5

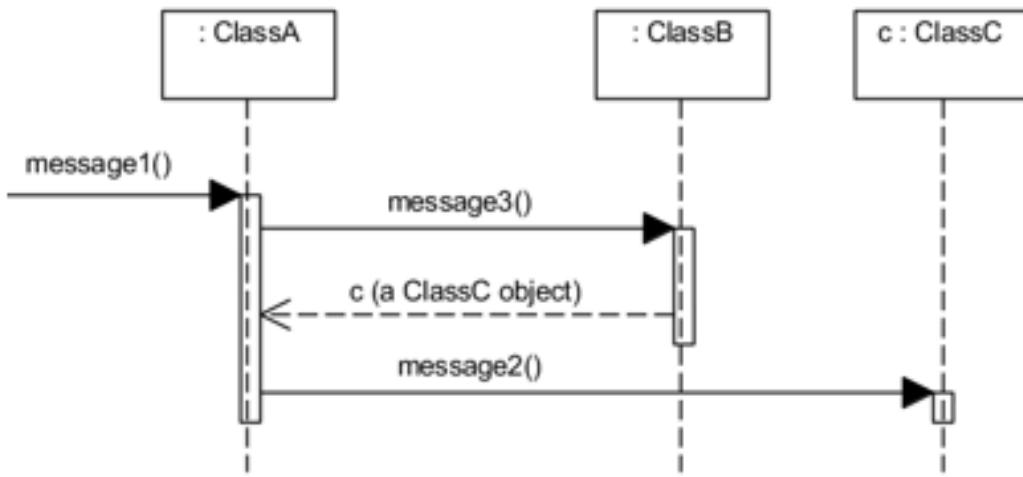
The replies would be as follows:



### Activity 3.6

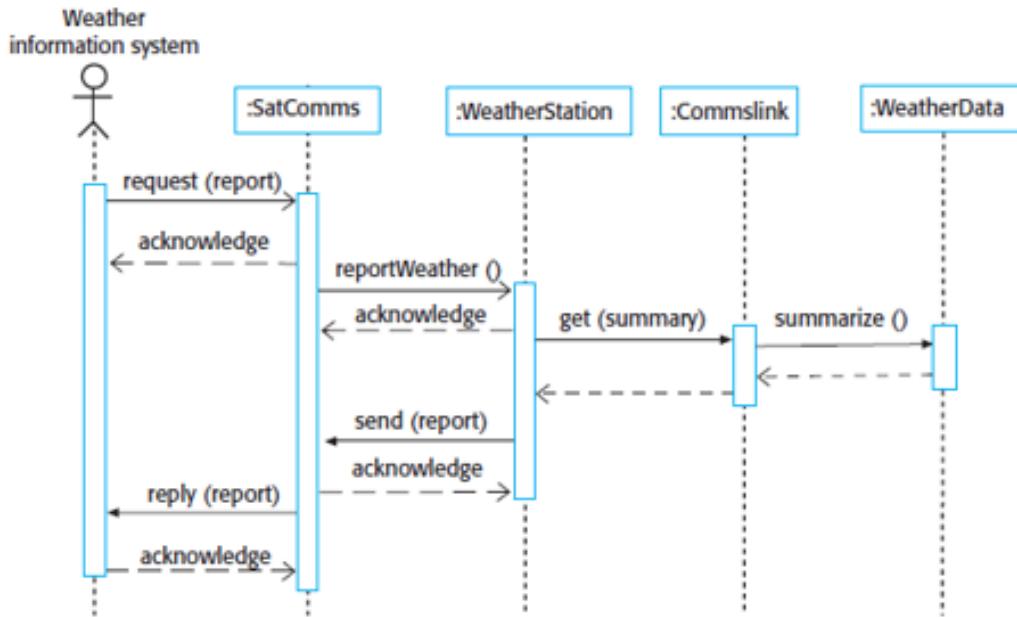
The message `message2()` is not possible. The class diagram shows that there is no direct association between an object of class `ClassA` and an object of class `ClassC`. Recall that we are dealing with actual objects, not classes. The objects have to know of the existence of another object before it can pass a message to it.

However, you might like to note that it might be possible for the following sequence of messages if `message3()` returns an object of `ClassC`. With this sequence, the `ClassA` object would know about the `ClassC` object.



Some course texts require the class diagram to be amended to show an association between ClassA and ClassC. Naturally, this could mean there will be a redundant association if both associations given in the question are retained.

## Activity 3.7



The messages for the above are described below:

1. The SatComms object receives a (asynchronous) request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. An asynchronous message (which has an open arrowhead) means that WeatherInformationSystem does not wait for a reply.
2. SatComms sends a (asynchronous) message to WeatherStation, via a satellite link, to create a summary of the collected weather data.
3. WeatherStation sends a (synchronous) message to a Commslink object to summarise the weather data. In this case, the WeatherStation object class waits for a reply. A synchronous message has a closed arrowhead.
4. Commslink calls the summarise method in the object WeatherData and waits for a reply.
5. The weather data summary is computed and returned to WeatherStation via the Commslink object.
6. WeatherStation then calls the SatComms object to transmit the summarised data to the weather information system, through the satellite communications system.

## Formative Assessment

1. What are domain models?
  - a. a computation independent model  
**Correct. These refer to the important domain abstractions used in a system.**  
**Refer to Study Unit 3, Section 1.1.**
  - b. a model that uses UML models to show the structural model of a system  
Incorrect. This refers to the operation of the system without reference to its implementation. Refer to Study Unit 3, Section 1.1.
  - c. a platform specific model

Incorrect. These are transformations of a platform-independent model to a specific platform. Refer to Study Unit 3, Section 1.1.

- d. a platform independent model

Incorrect. A platform independent model is a model regardless of the underlying implementation. Refer to Study Unit 3, Section 1.1.

2. Pick the element that is not found in a sequence diagram.

- a. Classes

**Correct. A sequence diagram is dynamic and shows how objects interact with each other. Classes are blueprints of the system and are static. Refer to Study Unit 3, Section 2.1.**

- b. Lifelines

Incorrect. Lifelines in a sequence diagram represent the objects that participate in an interaction. Refer to Study Unit 3, Section 2.1.

- c. Messages

Incorrect. To show the interactions in a sequence diagram, a sender sends a message to the receiver. Refer to Study Unit 3, Section 2.1.

- d. Activation rectangles

Incorrect. An activation rectangle represents an active object that is doing some processing. Refer to Study Unit 3, Section 2.1.

3. What is the purpose of a walkthrough?

- I. To work out the behaviour of objects interacting among themselves.
  - II. To model the behaviour of one application function.
  - III. To add new instance variables to the classes in the class association diagram.
  - IV. To add new methods to the classes in the class association diagram.
- a. I and II only

**Correct. The purpose of a walkthrough is to model the behaviour of one application function or one use case, to work out how the objects interact among themselves in the use case. Refer to Study Unit 3, Section 3.3.**

- b. III and IV only

Incorrect. Adding new instance variables and methods is the result of a walkthrough. Refer to Study Unit 3, Section 3.3.

- c. I, III and IV only

Incorrect. The purpose of a walkthrough is to model the behaviour of one application function or one use case, to work out how the objects interact among themselves in the use case. Refer to Study Unit 3, Section 3.3.

- d. I, II, III and IV

Incorrect. The purpose of a walkthrough is to model the behaviour of one application function or one use case, to work out how the objects interact among themselves in the use case. Adding new instance variables and methods is the result of a walkthrough. Refer to Study Unit 3, Section 3.3.

4. Which of the following is required before a walkthrough can be performed?

- I. Use case diagram
- II. Class diagram
- III. Class description
- IV. Activity diagram

- a. I only

Incorrect. A use case, not a use case diagram, is required since a walkthrough will work out the behaviour of the objects for one use case. Refer to Study Unit 3, Section 3.3.

- b. II and III only

**Correct. Yes, a class diagram and the class description are required, as well as the use case that we are going to perform the walkthrough on. Refer to Study Unit 3, Section 3.3.**

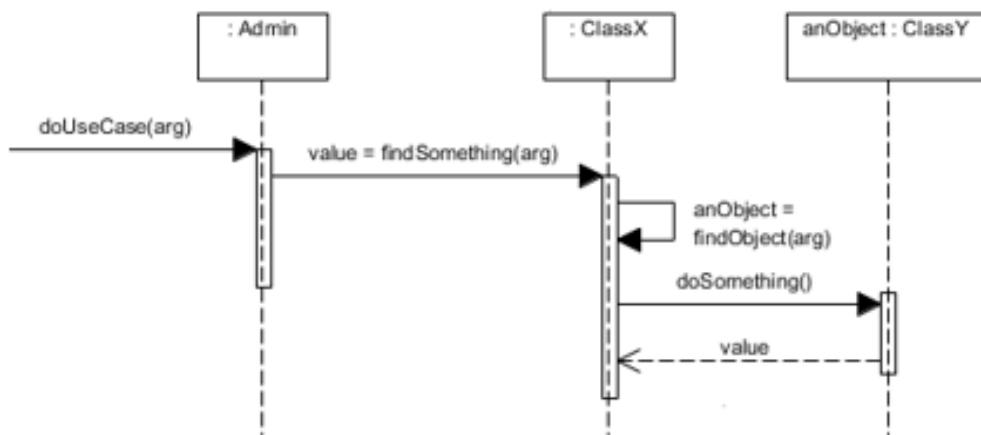
- c. I, II and III only

Incorrect. A use case, not a use case diagram, is required since a walkthrough will work out the behaviour of the objects for one use case. The class diagram and class description are also required. Refer to Study Unit 3, Section 3.3.

- d. II, III and IV only

Incorrect. Yes, a class diagram and the class description are required. An activity diagram is not required since it shows all activities of a process which may or may not be part of the system involving the use case on hand. Refer to Study Unit 3, Section 3.3.

5. Which methods belong to the class ClassX?



- I. doUseCase(arg)
- II. findSomething(arg)
- III. findObject(arg)
- IV. doSomething()
- a. I only

Incorrect. The orchestrating class Admin would have this method since the arrowhead of the message falls on its lifeline. Refer to Study Unit 3, Section 3.5.

- b. II only

Incorrect. Yes, the class ClassX would have this method since the arrowhead of the message falls on its lifeline. It also has the findObject(arg) method. Refer to Study Unit 3, Section 3.5.

- c. II and III only

**Correct. Only the arrowheads of these two messages fall on ClassX's lifeline. Hence, ClassX needs to have the implementation of these methods. Refer to Study Unit 3, Section 3.5.**

- d. II, III and IV only

Incorrect. The message doSomething() is a request to the object of ClassY. Hence it belongs to the class ClassY. Refer to Study Unit 3, Section 3.5.

6. Which one of the following statements about the orchestrating class is incorrect?

- a. The orchestrating class is channels inputs from the user interface to the classes in the application.

Incorrect. This is a correct statement as we wish to keep the user interface separate from the classes in the application. Refer to Study Unit 3, Section 3.1.

- b. There is a different orchestrating object for each walkthrough.

**Correct. There is only one orchestrating object in the entire application.**

**Refer to Study Unit 3, Section 3.1.**

- c. A walkthrough always starts with the orchestrating class.

Incorrect. This is a correct statement since we wish to establish whether a particular use case is behaving correctly. Refer to Study Unit 3, Section 3.1.

- d. A walkthrough is an operation in the orchestrating class.

Incorrect. A walkthrough is the process of performing a use case and an operation may be created for the orchestrating class for this purpose. Refer to Study Unit 3, Section 3.1.

7. For the following fragment of a class diagram, why is special care required for the bi-directional association `consistsOf` during walkthroughs?



- a. This is because the bi-directional association is not shown in the (structural) class association diagram.

Incorrect. Class association diagrams also show the bi-directional association without the navigation arrows. Refer to Study Unit 3, Section 3.6.

- b. This is so as to determine the value of instance variables to be added to each class.

Incorrect. The value of instance variables depends on the multiplicities and not whether the association is bi-directional. Refer to Study Unit 3, Section 3.6.

- c. This is so as to determine the operations that are to be added to each class.

Incorrect. Operations are not identified during walkthroughs. Refer to Study Unit 3, Section 3.6.

- d. For consistency

**Correct. We need to ensure that the Department object that is associated with the Branch object is the same Branch object that is associated with the Department object. Refer to Study Unit 3, Section 3.6.1.**

---

## References

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.



# Study Unit 4

**Dynamic Object-Oriented  
Modelling II**

## Learning Outcomes

By the end of this unit, you should be able to:

1. show where and when a new object is created or removed in a sequence diagram
2. describe the state of an object and explain the need for state changes
3. identify the elements of a State Machine diagram
4. show the life cycle of an object with a State Machine diagram
5. discuss the circumstances in which the State Machine diagram is used
6. write programs for the class with a State Machine diagram

## Overview

In this unit, we continue to discuss how to develop dynamic models or behaviour models, to determine the interactions among the objects and achieve the required system behaviour. Recall that behavioural models are models of the dynamic behaviour of a system as it is executing. While the system is in execution, it needs to interact with its environment. In this unit, we will continue to discuss interactions between objects and in particular, how to create a new association between objects and how to remove an association between objects. We will also discuss the interaction in the form of events. UML state machine diagrams are used to design these interactions.

# Chapter 1: More Analysis with Sequence Diagrams



## Lesson Recording

Implementations

In Study Unit 3, we discussed how sequence diagrams may be analysed for object collaborations. This results in identifying methods in participating classes through message passing between objects. In this chapter, we continue with analysis which results in the creation of a new association or deleting an association.

## 1.1 Create a New Association

We will do a few more analyses to illustrate the use of sequence diagrams. The use case for the next analysis is again taken from the previous study unit:

Use case	Assign a new manager to a branch.
Initiator	An administrative staff.
Objective	To assign a new manager to a branch.
Pre-conditions	The location of the branch and the relevant particulars (name and employee number) of the new manager must be given.
Post-conditions	The branch would have a new manager.
Assumptions, Notes or constraints	None

The class association diagram in Figure 4.1 shows navigation relevant to this analysis.

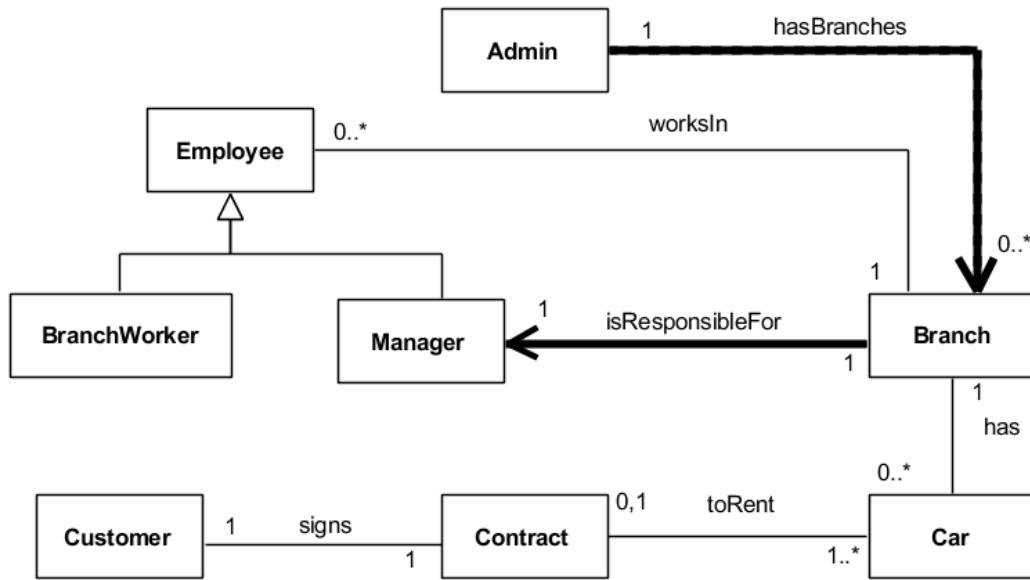


Figure 4.1 Assigning a new manager to a branch

The following method has been added to the orchestrating class **Admin**:

```
assignNewManager(location: string, name: string, employeeNumber: integer) : void
```

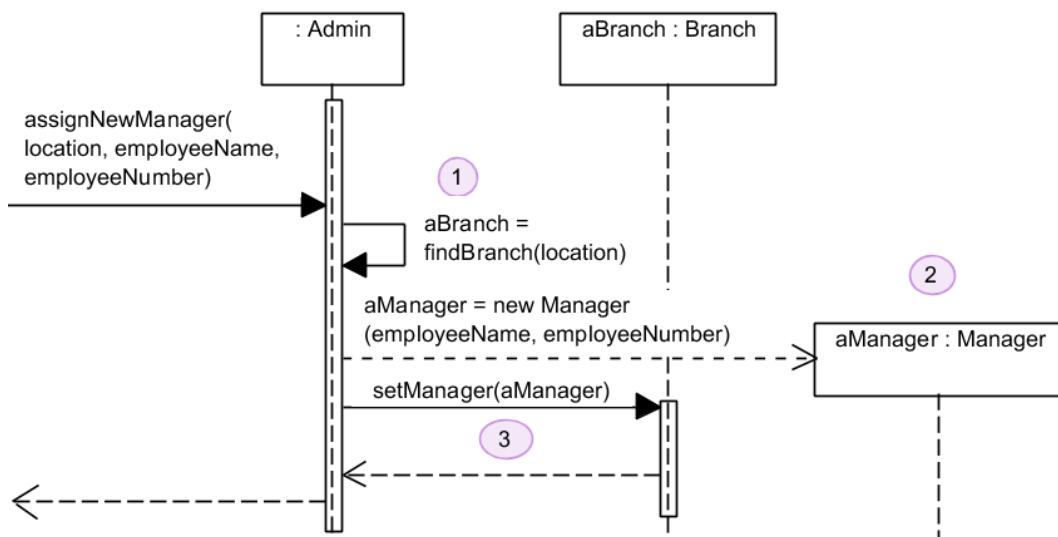
The following is the walkthrough:

**Objective:** To assign a new manager to a branch.

**Given:** A location of a branch, the name and employee number of the new manager.

1. Locate the **Branch** object with the specified location, linked to the orchestrating object via the association `hasBranches`.
2. Create the new **Manager** object with the given name and employee number.
3. Create the association between the **Branch** object and the new **Manager** object.

The sequence diagram for this walkthrough is shown in Figure 4.2.

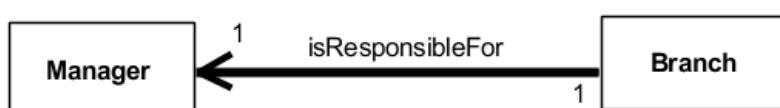


**Figure 4.2** Assigning a new manager to a branch in sequence diagram

In Figure 4.2:

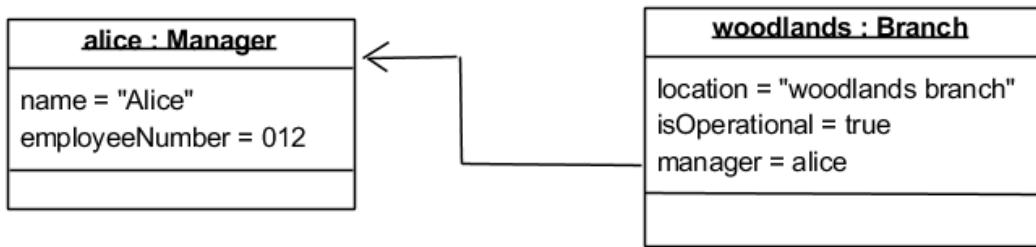
- Step 1 for locating the branch is the same as the previous example.
- Step 2 requires the new Manager object to be created. To do this, the Admin object uses the constructor of the class Manager and the data provided, employeeName and employeeNumber. Note how the sequence diagram is drawn for the object. Its lifeline begins at the point where it is created.
- Step 3 creates the association between the new Manager object and the Branch object. This is done by using the setter method setManager() to assign a value to the instance variable in Branch that implements the association. Recall from the previous section that the association is implemented by an instance variable.

Examine Step 3 carefully again. We have assumed that the navigation between Manager and Branch is one-directional, from Branch to Manager.



**Figure 4.3** A one-directional association between branch and manager

Let us take a scenario and deal with actual objects. In terms of actual objects, we could have the following as one particular possibility where Alice is the branch manager for Woodlands Branch as shown in Figure 4.4.



**Figure 4.4** Creating a one-directional association between branch and its manager

## The Results of Analysis

As a result of this analysis with the sequence diagram, we conclude that the following methods (in UML style notation) will be needed:

<u>Class</u>	<u>Methods</u>
<b>Admin</b>	assignNewManager(location : string, employeeName: string, employeeNumber : string) : void findBranch(location : string) : Branch
<b>Branch</b>	setManager(aManager : Manager) : void

The message for step 2 creates the new Manager object. In Python, this is written as:

```
aManager = Manager(employeeName, employeeNumber)
```

In the use case, the employee name and the employee number are provided for the new manager. As a result, we would specify in the `__init__()` function of the Manager class, these two items of data.



## Activity 4.1

Examine the sequence diagram for the use case "Assign a new manager to a branch". Determine the client, the server and the collaborator for the message setManager().



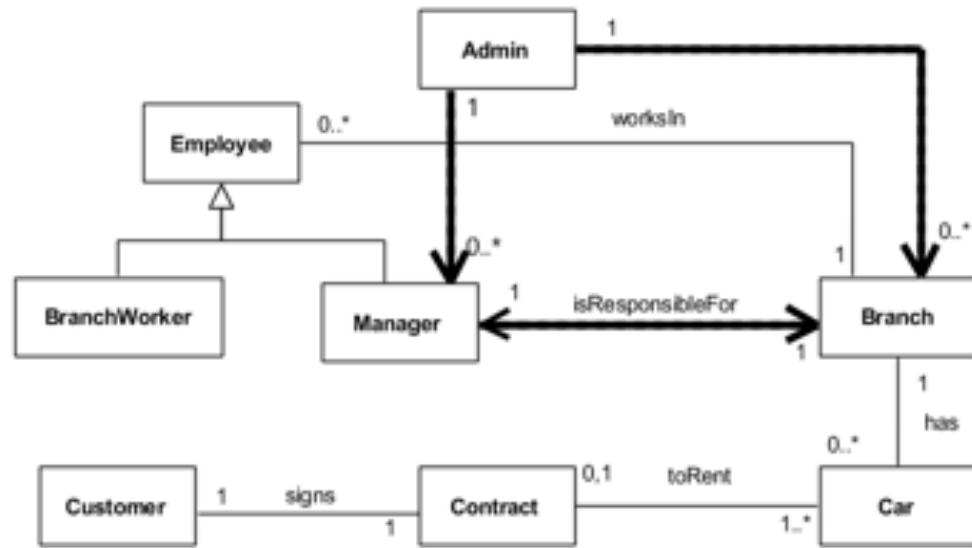
## Activity 4.2

For this question, you are asked to change the branch of a manager. In other words, the manager is assigned to take charge of another (existing) branch.

The following is the use case for this:

Use case	Re-assign a manager to another branch.
Initiator	An administrative staff.
Objective	To assign a branch identified by its location to a given manager, identified by the person's name.
Pre-conditions	The branch is an existing branch.
Post-conditions	The branch would have a new manager.
Assumptions, Notes or constraints	None

Use the class association diagram in Figure 4.5 for your analysis.



**Figure 4.5** Class association diagram to re-assigning a manager to another branch

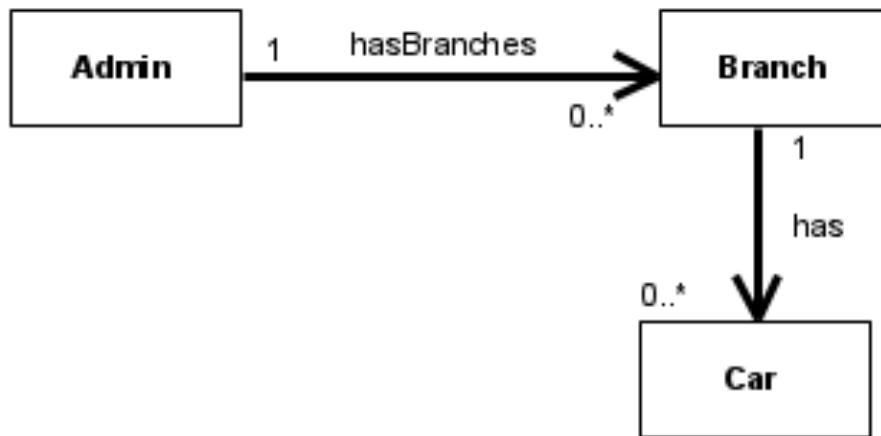
## 1.2 Remove an Object

Suppose we wish to remove an object. For the use case "Remove a car from a branch", an administrative staff wishes to remove a car from a branch for various reasons, for example, the car is aged or sold.

<b>Use case</b>	Remove a car.
<b>Initiator</b>	An administrative staff.
<b>Objective</b>	To remove a car given the car registration number and branch.
<b>Pre-conditions</b>	The car, identified by registration number, does not have a contract with a customer.
<b>Post-conditions</b>	The car is destroyed.

**Assumptions, Notes or constraints**

Figure 4.6 shows the relevant part of the class association diagram involved in this analysis.



**Figure 4.6** Part of class association diagram relevant to remove a car

**The following would be the walkthrough:**

**Objective:** To remove a car

**Given:** The branch name and the registration number of the car

1. Locate the Branch object with the specified name of the branch, linked to the orchestrating object via the association hasBranches.
2. Locate the Car object, linked to the Branch object found in step 1 via the association has.
3. Remove the Car object found in Step 2.



### Activity 4.3

For the use case "Remove a car from a branch", we would like to add a new method such as the following to the orchestrating class Admin:

```
removeCar(location : string, regnNo : string) : void
```

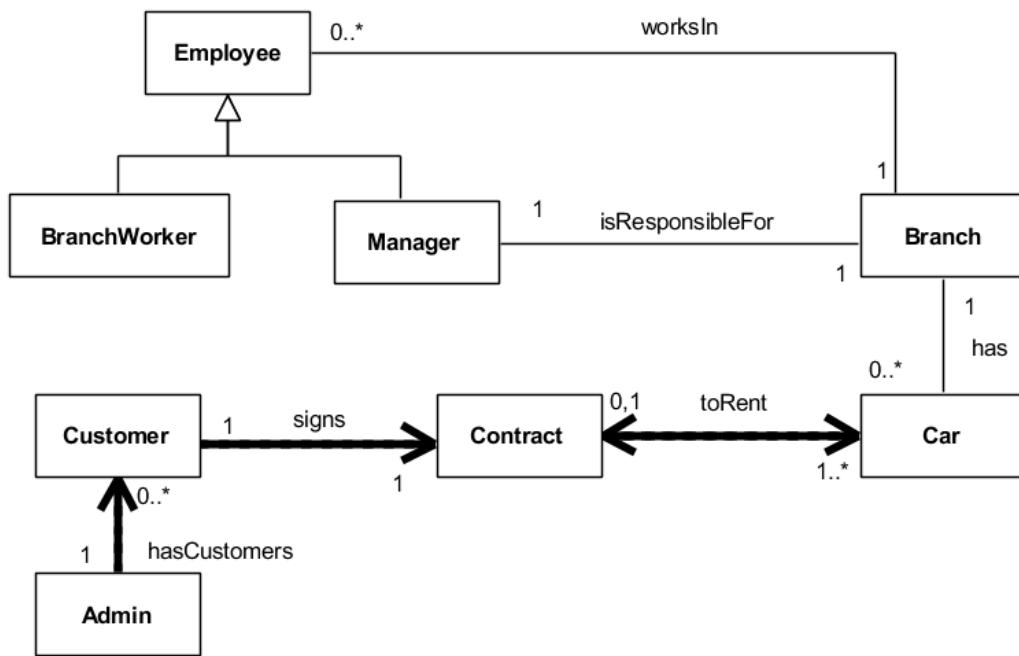
Draw the sequence diagram.

## 1.3 Remove an Association

In the previous section, we have shown how an object is removed. In this section, **we remove an association without destroying any object.** For the use case, a customer wishes to discontinue renting one of the cars in his contract. The car itself remains with the branch and is not destroyed.

<b>Use case</b>	Remove a car from a contract.
<b>Initiator</b>	A marketing staff.
<b>Objective</b>	To remove a car from the contract of a given customer.
<b>Pre-conditions</b>	The customer, identified by name, must have a contract with cars on rental.
<b>Post-conditions</b>	The contract would have one car less under rental.
<b>Assumptions, Notes or constraints</b>	None

We shall use the class association diagram in Figure 4.7 for our analysis.



**Figure 4.7** Class association diagram to remove a car from a contract

The following would be the walkthrough:

**Objective:** To remove a car from a contract.

**Given:** The name of a customer and the registration number of the car.

1. Locate the **Customer** object with the specified **custName**, linked to the orchestrating object via the association **hasCustomers**.
2. Locate the **Contract** object, linked to the **Customer** object found in Step 1 via the association **signs**.
3. Remove the association **toRent** between the **Contract** object in Step 2 and the **Car** object identified by the registration number **regnNo**.

The class association diagram shows that the association between the classes **Contract** and **Car** object is bi-directional. This means that to remove a **Car** object from a contract, we need to update the values of the instance variables in both the **Car** object and the **Contract** object. This involves:

- a. Removing the Car object from the collection of cars in the Contract object since the car is no longer part of the contract.
- b. Removing the Contract object from the Car object since the car is no longer part of the contract.

Figure 4.8 shows what it means to remove an association. When a car is removed from a contract, the association between the Contract object and its Car object is removed.

For this use case, we would need to add a method such as the following to the orchestrating class Admin.

```
removeCarFromContract(custName : string, regnNo : string) : void
```

The sequence diagram that we draw for the walkthrough is shown in Figure 4.9.

Note again that the circled numbers in the sequence diagram correspond to the steps of the walkthrough. They are not part of the UML notation. We have added them for easy reference.

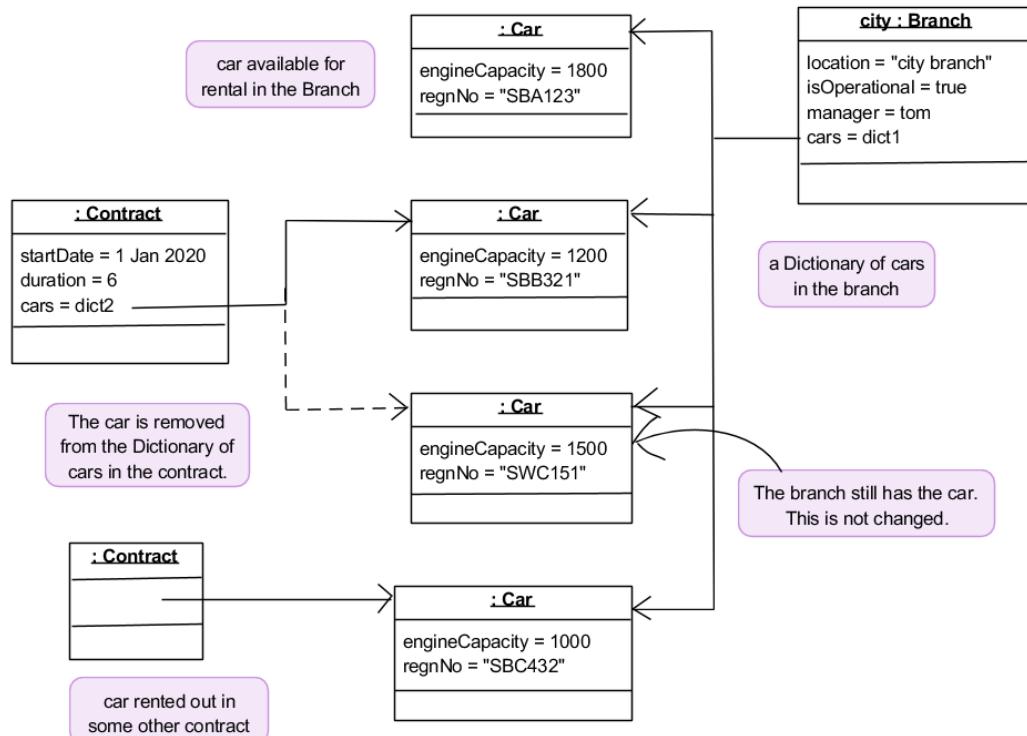
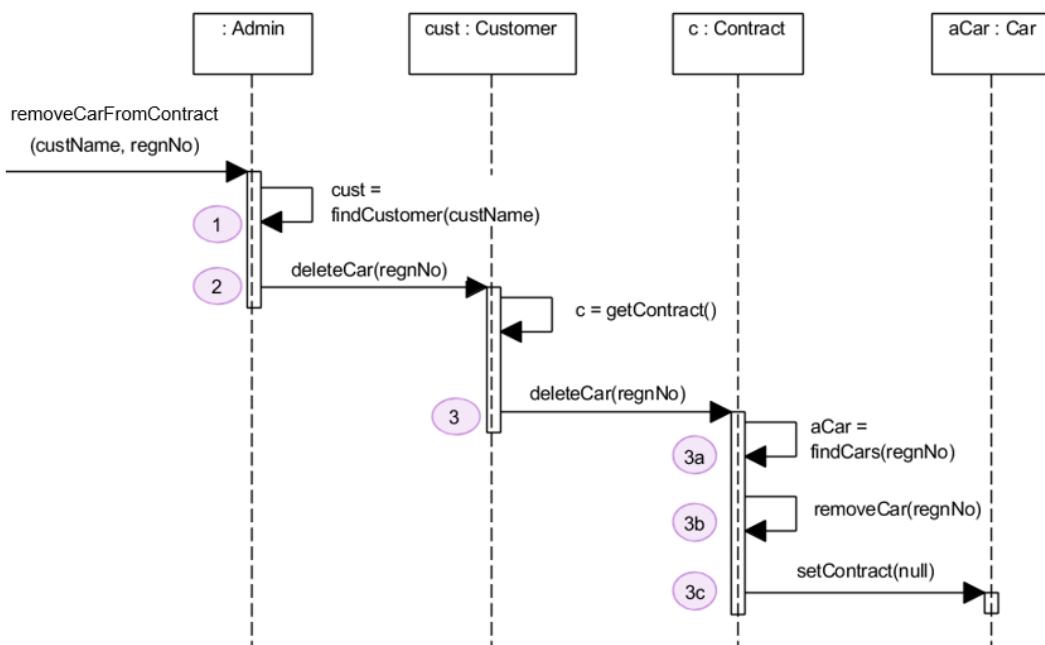


Figure 4.8 Class association diagram to remove a car from a contract



**Figure 4.9** Sequence diagram to remove a car from a contract

At this stage, we have completed our discussions on dynamic modelling based on use cases and have focused on **explaining the mechanics of modelling**. In the next chapter, we will model the behaviour of objects based on events. We have not discussed issues such as whether our approach is the best and how to handle errors during the interactions with the user. Later, we might wish to modify, say, the walkthroughs to improve the design of the application. We will cover such aspects in the units that follow.



#### Activity 4.4

For the use case "Remove a car from a contract":

- Tabulate the methods that the classes involve should have as a result of the analysis using the sequence diagram.
- Show the steps of the walkthrough in the class association diagram.
- Explain why it is necessary for step 3 to be split into steps 3a, 3b and 3c.

- d. Explain why it is appropriate to use a message name such as deleteCar(regnNo) for step 3a whereas the message name setContract() is used for step 3b.



## Activity 4.5

The manager for a branch has resigned. The application has to be updated accordingly. The following is the use case for this:

Use case	Delete a manager of a branch.
Initiator	An administrative staff.
Objective	To delete a manager of a branch due to the reassignment or other reasons. The name of the manager is given for this purpose.
Pre-conditions	The manager is in charge of a branch.
Post-conditions	The branch would have no manager.
Assumptions, Notes or constraints	None

For this question, assume that the multiplicity for the relevant associations allows a branch to have no manager temporarily while a replacement is being assigned.

Provide the walkthrough and show the sequence diagram for the analysis using the class association in Figure 4.5.

## Chapter 2: State Diagram

### 2.1 The State of an Object

From our understanding of the basic concepts in the object-oriented approach, we know that every object has its own set of values for its instance variables. These values taken together constitute the state of the object.

Figure 4.10 below shows the class Room. With this class, we have created an object, rm812. This object has its own set of values for the instance variables. This set of values forms the state of the object rm812.

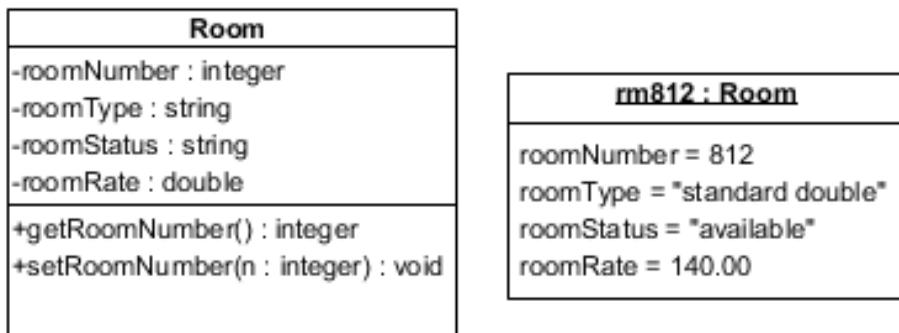


Figure 4.10 An object created from its class with its own state

The current state of an object is a result of the events that have occurred to the object and is determined by the current value of the object's attributes and the links that it has with other objects. Some attributes and links of an object are significant for the determination of its state while others are not. For example, the roomStatus attribute of a Room object determines whether you can assign the room to a guest. Other attributes such as roomType and roomRate have no impact upon its state. When the room is assigned to a guest, the state of the object changes and we could have the object diagram as shown in Figure 4.11.

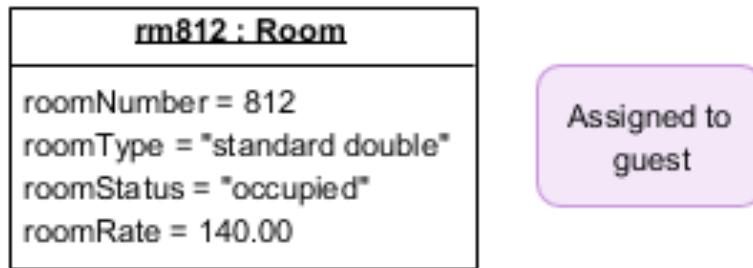


Figure 4.11 Object state change

## 2.2 Introduction to State Machine Diagram

Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone. Event-driven modelling shows how a system responds to external and internal events. UML has the notation called the state machine diagram or state diagram. The state machine diagram is used to model the behaviour of an object based on events. It can be used to show:

- The life cycle of objects. It shows all the possible states that an object can go through from creation to destruction.
- The state-dependent behaviour of objects. It helps us to analyse and understand how an object responds to a message in different ways depending on its state.
- How events can change the state of objects. It helps us to identify the relevant events and, from there, the response that is required from the objects.
- What transitions are possible between states but not others.
- The nature of the transitions among the states of an object. This enables us to analyse how one state can be changed to another.

Drawing a state machine diagram correctly ensures that we analyse and think through what we are dealing with. An incorrect state machine diagram could help to detect gaps or errors in our understanding of object behaviour.

You will also find that using a state machine diagram is helpful when you need to discuss and clarify with the users:

- The changes that can happen to an object
- The different states that an object can take
- How the changes to an object depend on its state

A state machine diagram consists of the following:

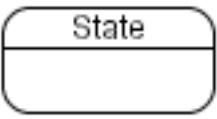
- State
- Transition
- Event
- Initial and final pseudo-states

Often, we might need to differentiate between an event and an action. An event is something that happens to an object whereas an action is the response of the object.

### 2.2.1 Elements of a State Machine Diagram

Table 4.1 shows the basic elements of a State Machine Diagram. We will look at each one of these in detail.

**Table 4.1** Elements of a State Machine

	State
	Transition
	Initial pseudo-state
	Final pseudo-state

### 1. State

This refers to the state of an object. It is shown as a box with rounded corners and a label that indicates the state of the object. Figure 4.12 shows a possible state of a car. Conceptually, an object remains in a state for an interval of time.

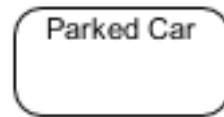


Figure 4.12 UML symbol showing the state of an object

### 2. Transition

This represents the change from one state to another. It is shown as an arrow. A label would indicate the event that will cause the transition between the states. Movement from one state to another is dependent upon events that occur with the passage of time. Figure 4.13 shows a transition from the "Parked Car" source state to the "Moving Car" state.

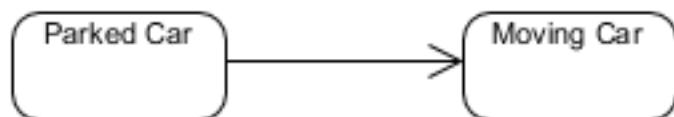
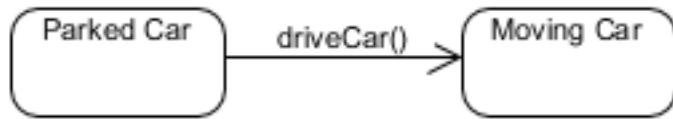


Figure 4.13 UML symbol showing a transition between two states

### 3. Event

This causes the change from one state to another. An event is shown as a message label with the transition arrow. The message identifies the action that will cause the transition between the states. It may include an argument/data if one is needed. Figure 4.14 shows that the transition from "Parked Car" state to "Moving Car" state is triggered when the driveCar() event occurs.

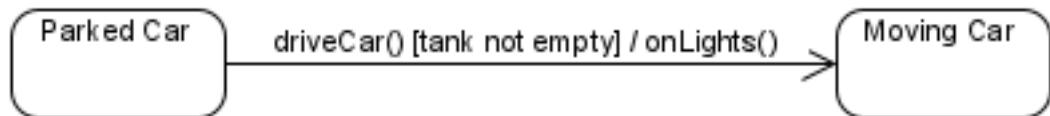


**Figure 4.14** UML symbol showing an event causing the transition

The event label has the following format:

event [ guard condition ] / action-1, ... action-N

The event is the trigger that causes the transition. If there is a guard condition, this is checked to be true before the state change can be effected. The actions after the / are one or more actions that are executed after the state change and are optional.



**Figure 4.15** UML symbol showing an event with condition and action

For example, Figure 4.15 shows the transition from "Parked Car" state to "Moving Car" state is triggered not only when the event `driveCar()` occurs but also the guard condition "tank not empty" is true. After that, the `onLights()` action is executed.

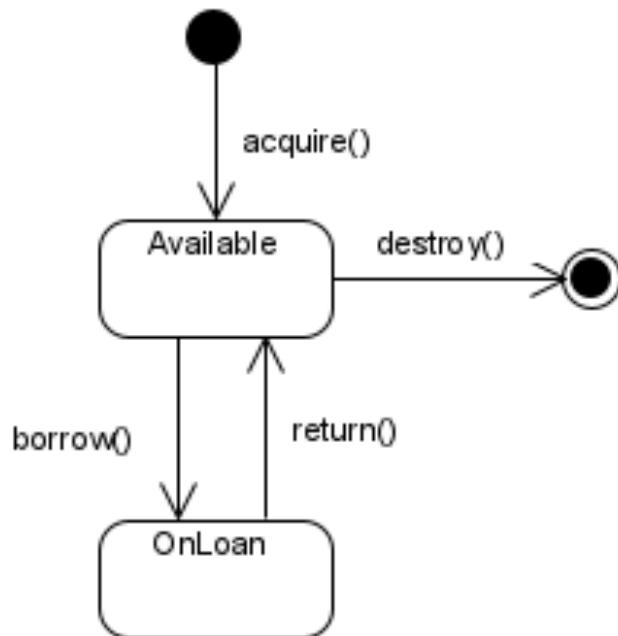
There are 4 types of events:

- A call trigger occurs when an object receives a call of one of its operations either from another object or from itself. Figure 4.14 shows a call trigger.
- A change trigger occurs when a condition in the message label becomes true. This is illustrated in Figure 4.15.

- c. A **signal trigger** occurs when an object receives a **signal**. This type of event is not covered in this course.
- d. A **relative-time trigger** is caused by the passage of a designated period of time after a specified event (and entry to the current state). This is usually written with the keyword after and the time in square brackets. Figure 4.28 shows that after [timeset seconds], the stop watch goes from "CountingDown" state to "Ringing" state.

#### 4. Initial and Final Pseudo-States

These indicate where the object begins and ends its existence. Figure 4.16 shows a simplified state machine showing the state of a book in a library showing these pseudo-states. **A state machine diagram must have only one initial pseudo-state but there can be zero, one or more final pseudo-states.**

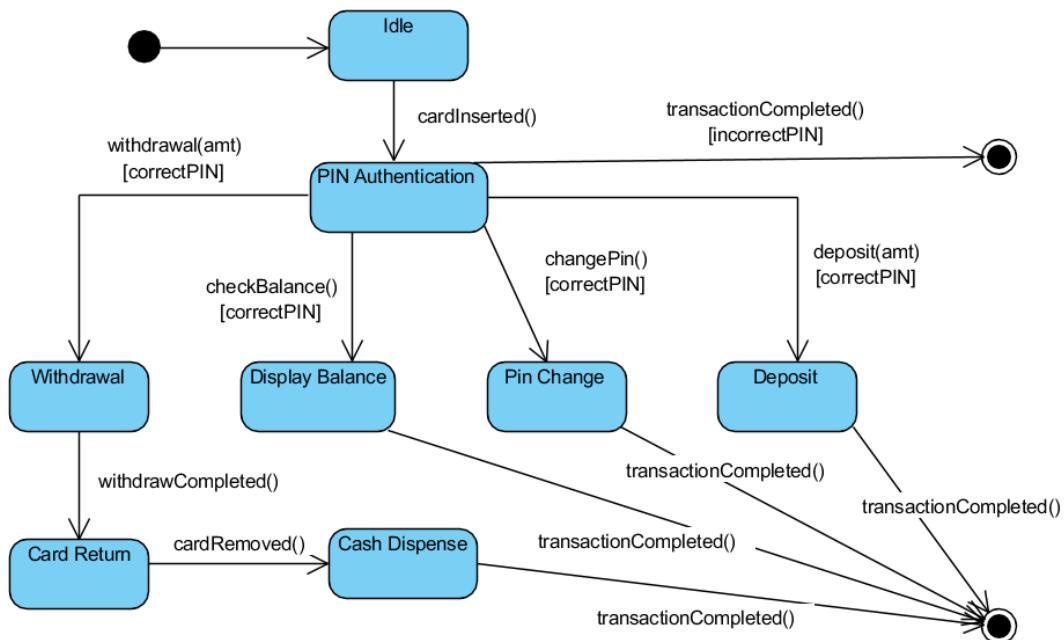


**Figure 4.16** UML initial and final pseudo-states

## 2.2.2 Examples

### Example 1 – ATM

The example in Figure 4.17 shows the state machine diagram of an ATM process that allows four events – withdrawal, check balance, change PIN and deposit – to happen. Note that events are triggered by external systems. In this example, most of the events are triggered by the user using the ATM. Also note that there can be more than one final pseudo-state – for a neater diagram.



**Figure 4.17 State Machine Diagram of an ATM**

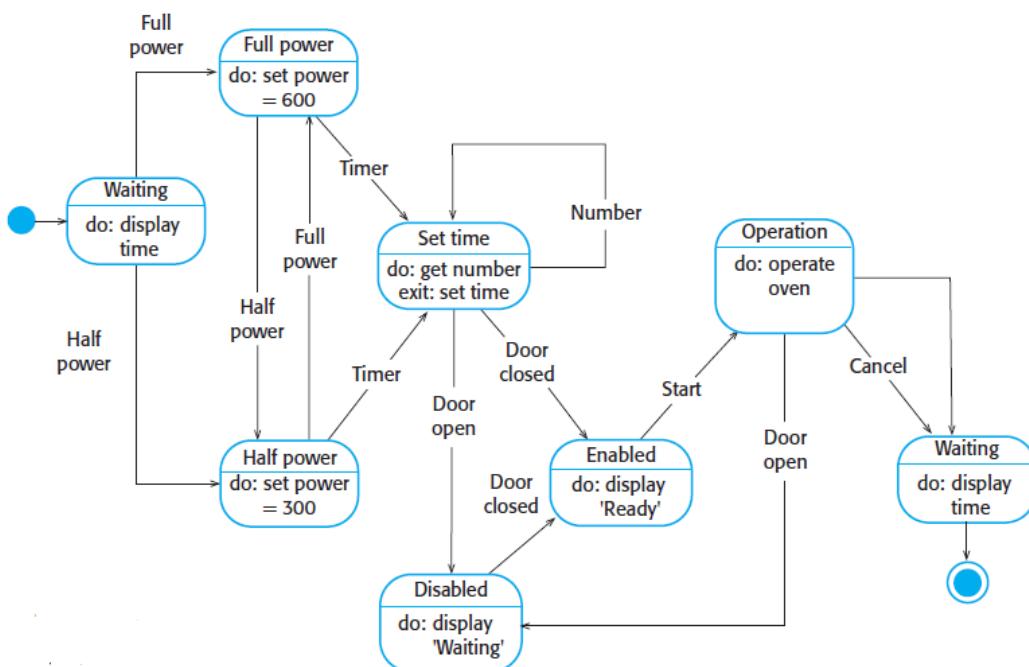
### Example 2 – Microwave Oven

The example in Figure 4.18 shows the state machine diagram of a microwave oven. The sequence of actions in using the microwave is assumed as follows:

- Select the power level (either half power or full power).
- Input the cooking time using a numeric keypad.
- Press Start and the food is cooked for the given time.

Here is a short description of the state machine diagram of the microwave oven:

The microwave oven starts off being in a "Waiting" state and can transit to the "Full power" state or "Half power" state. The microwave oven can change from "Full power" state to "Half power" state and vice versa before it transits to a "Set time" state. The diagram also shows the state of the microwave oven when the door is opened and closed. Most importantly, the Start button can only be pressed when the door is closed and when it is in the "Enabled" state. This causes the microwave to open to go into the "Operation" state (to cook the food) and it returns to the "Waiting" state once the food is cooked or the Cancel button is pressed. Do examine the state diagram for all other possible transitions while operating the microwave oven.



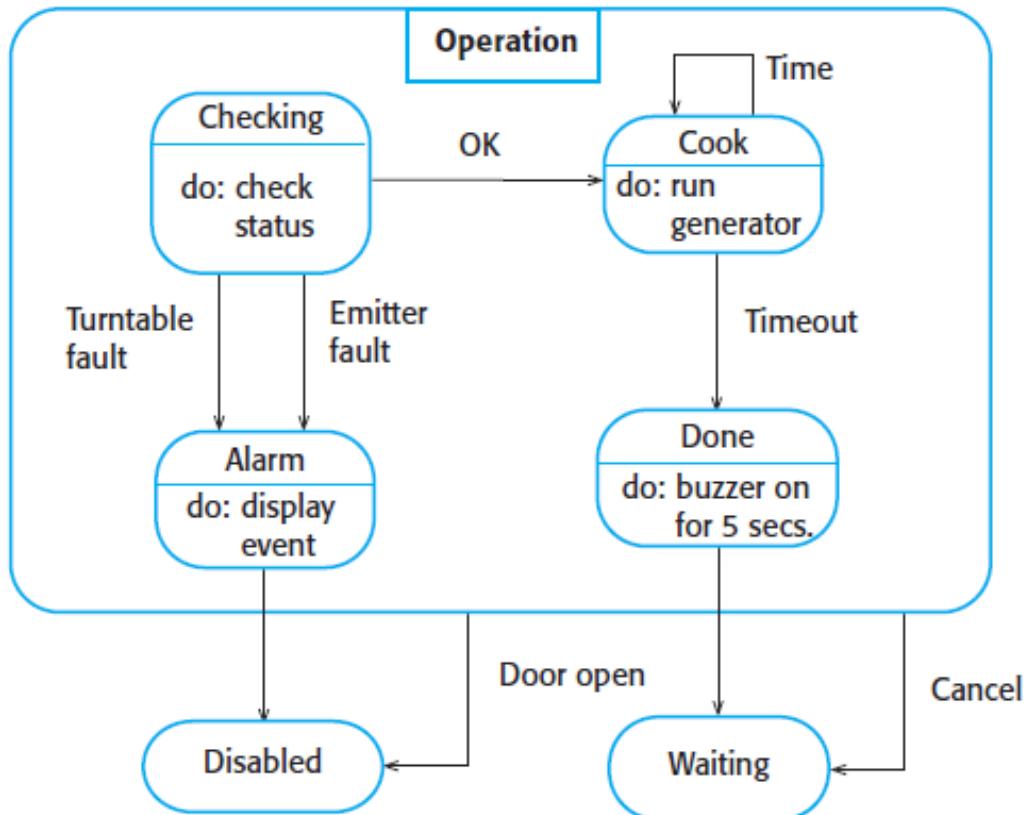
**Figure 4.18** State Machine Diagram of a microwave oven

(Source: Sommerville, I. (2016). *Software Engineering*)

### 2.2.3 Nested Substates

The problem with state machine diagrams is that the number of states grows rapidly, especially for large systems. In such cases, we introduce 'superstate' or 'high-level' state. For example, the microwave oven has "Operation" as a superstate which can be expanded

to a separate state diagram with substates: "Checking", "Cook", "Done" and "Alarm" as shown in Figure 4.19. The do action inside each state is explained in Section 2.3.2.



**Figure 4.19** Substates of the Operation state

(Source: Sommerville, I. (2016). *Software Engineering*)



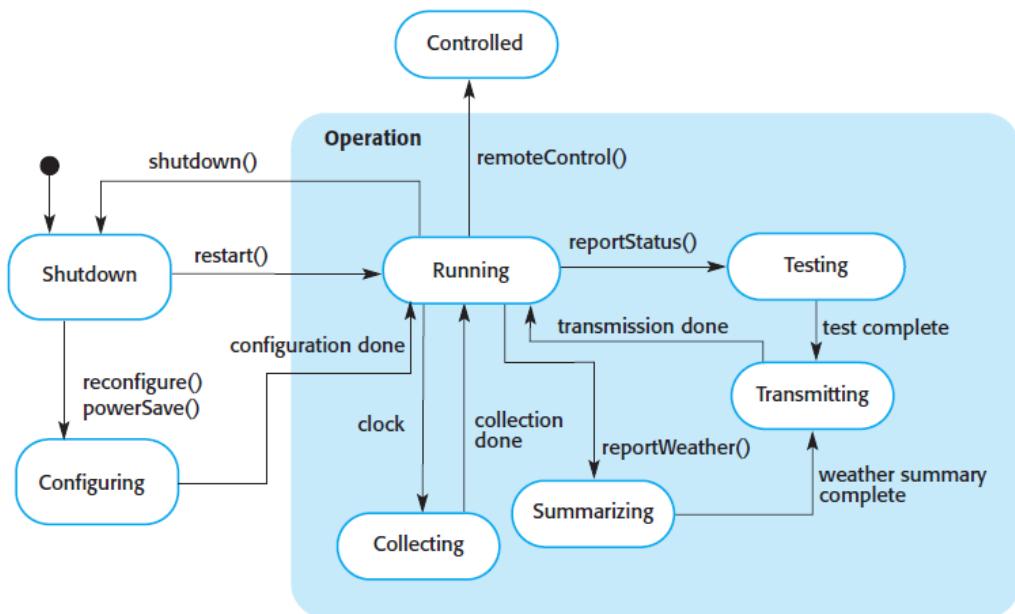
## Read

For more details on events and the states in the diagram read: Sommerville, I. (2016). *Software Engineering*. Section 5.4.



## Activity 4.6

Examine the weather station state machine diagram below and briefly describe what can happen to the weather station during its lifetime.



(Source: Sommerville, I. (2016). *Software Engineering*)

## 2.3 Additional Features of a State Machine Diagram

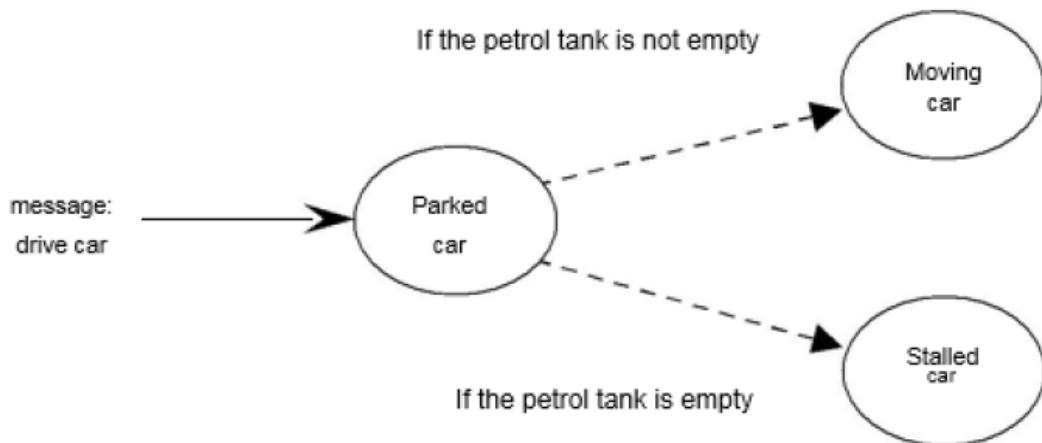
### 2.3.1 Guards

The response of an object to a message often depends on certain conditions. When we say "certain conditions", these conditions could be the values of the attributes within the object.

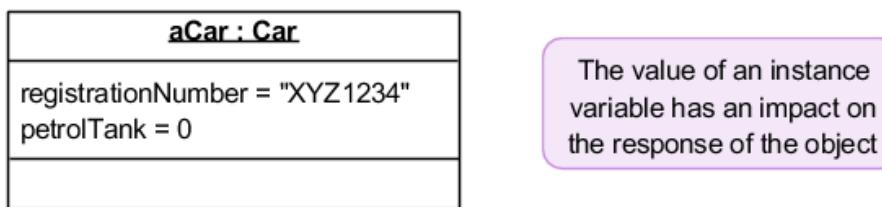
For example, a parked car will change to a moving car if it is driven and its petrol tank is not empty. But if a parked car has an empty petrol tank, driving it will not result in a

moving car. The same event "drive car" acting on the same object gives two responses as shown in Figure 4.20.

This means that we need to examine the actual value of an attribute of the object. In the above example, we would need to know how full the petrol tank is and add this as an attribute in the Car class as shown in Figure 4.21.

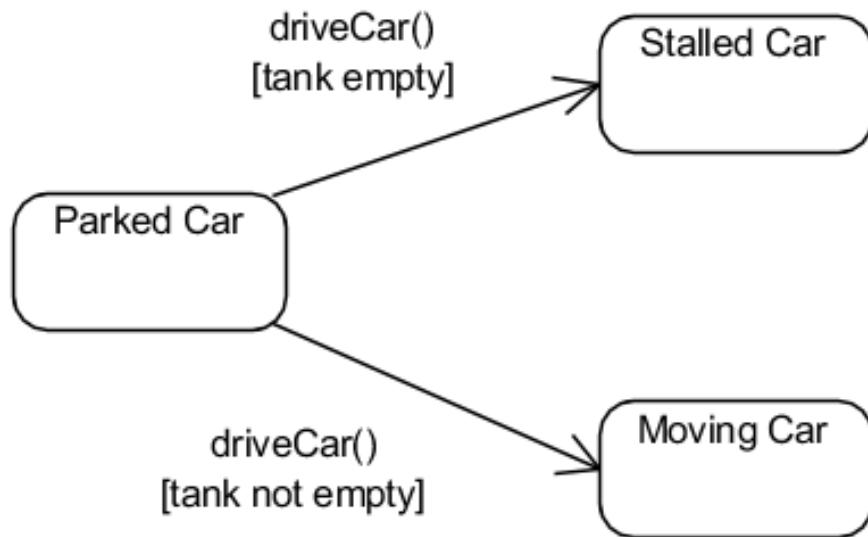


**Figure 4.20** The same object with the same message can give different responses



**Figure 4.21** The actual state of an object has an impact on its response

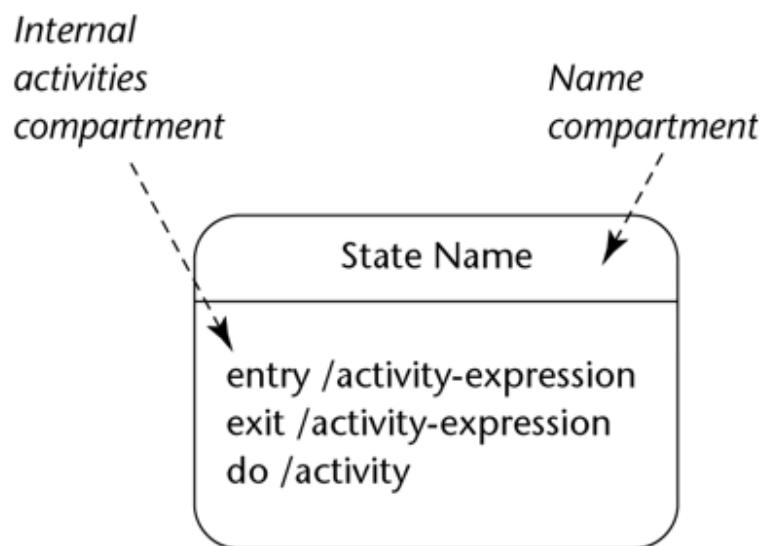
In the state machine diagram, we add a guard to the event to show the difference in response. Figure 4.22 shows that the message `driveCar()` will end up with two possible states depending on whether the petrol tank is empty. Note how the guards are added to the messages. Also note that when a message has two (or more) responses, each message must have a guard condition and all the conditions for this same message must be mutually exclusive.



**Figure 4.22** A state machine diagram with guards

### 2.3.2 Internal Actions or Activities

A state may be drawn with two compartments; a name compartment and an internal compartment that holds the internal actions or activities that are performed when the object is in that state.



**Figure 4.23** Internal activities of a State

Figure 4.23 shows the entry, exit and do notations that specifies the action or activity that is performed when entering, when leaving and while in the state, respectively.

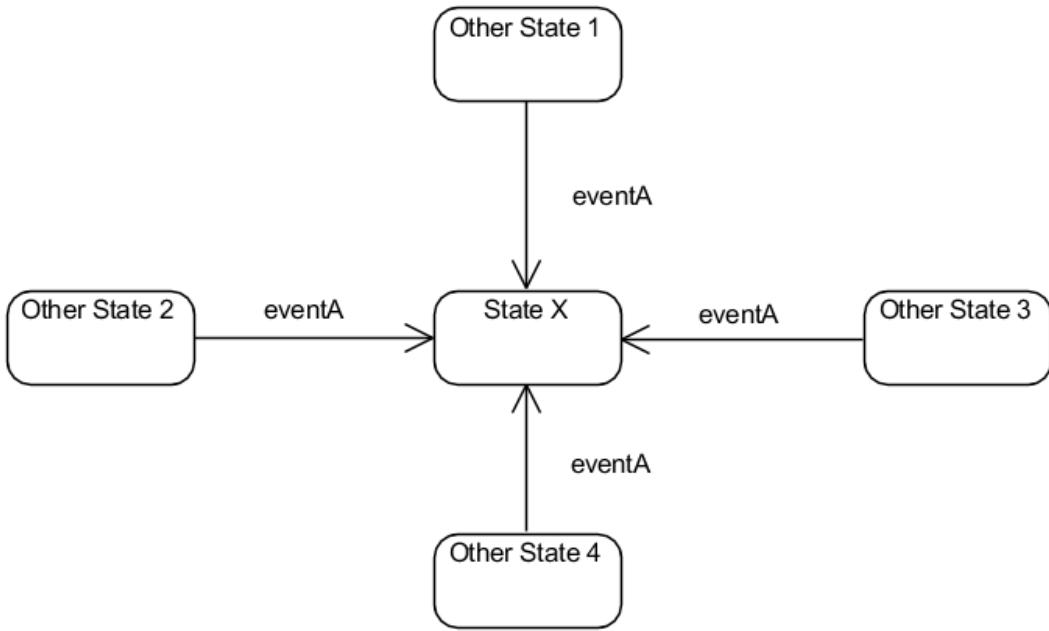
### 1. Entry and Exit Events

The entry event notation is useful for the following two situations:

- a. It is possible for an object to acquire a certain state from several other states. All of these require a certain event to take place.
- b. For an object to acquire a particular state, a certain event should always happen.

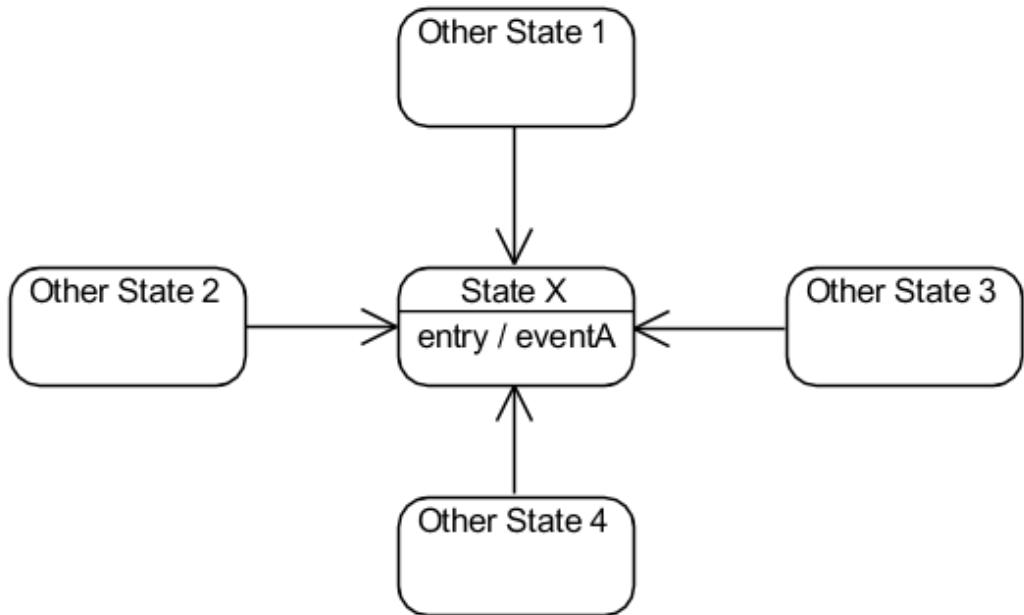
Figure 4.24 shows that an object can enter state X from four other states. For all of these four states, event A should happen before the object acquires state X. You can see that event A is repeated for each of these four states. This is not ideal because if a change is needed to event A we need to do the change four times. Furthermore, writing something four times is tedious and error-prone.

Note that a final pseudo-state may have several (same) events entering it – see the state diagram for an ATM in Figure 4.27.



**Figure 4.24** StateX can come about from several other states, but eventA has to take place for this to happen

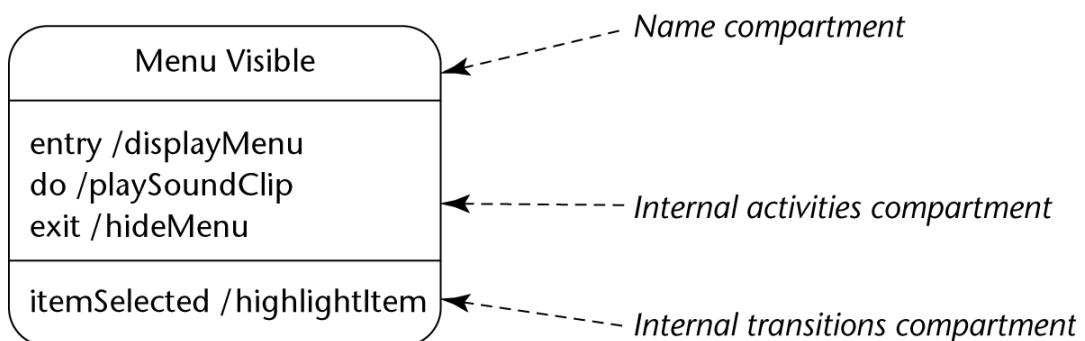
One solution to the problem is to write the common event once inside state X.  
UML has the entry (and exit) event notation for this to avoid repeating the event description. So Figure 4.24 may be re-drawn as shown in Figure 4.25.



**Figure 4.25** A state with an Entry event

## 2. Do Action

The **do** notation in the internal compartment specifies the activity to be performed continuously while in this state. For example, Figure 4.26 shows the actions to be done when the menu becomes visible. The **entry event** causes the menu to be displayed. When the object remains in the "Menu Visible" state, the activity causes a sound clip to be played. Event `itemSelected()` triggers the action `highlightItem()`. Exiting the state triggers the action `hideMenu()`.

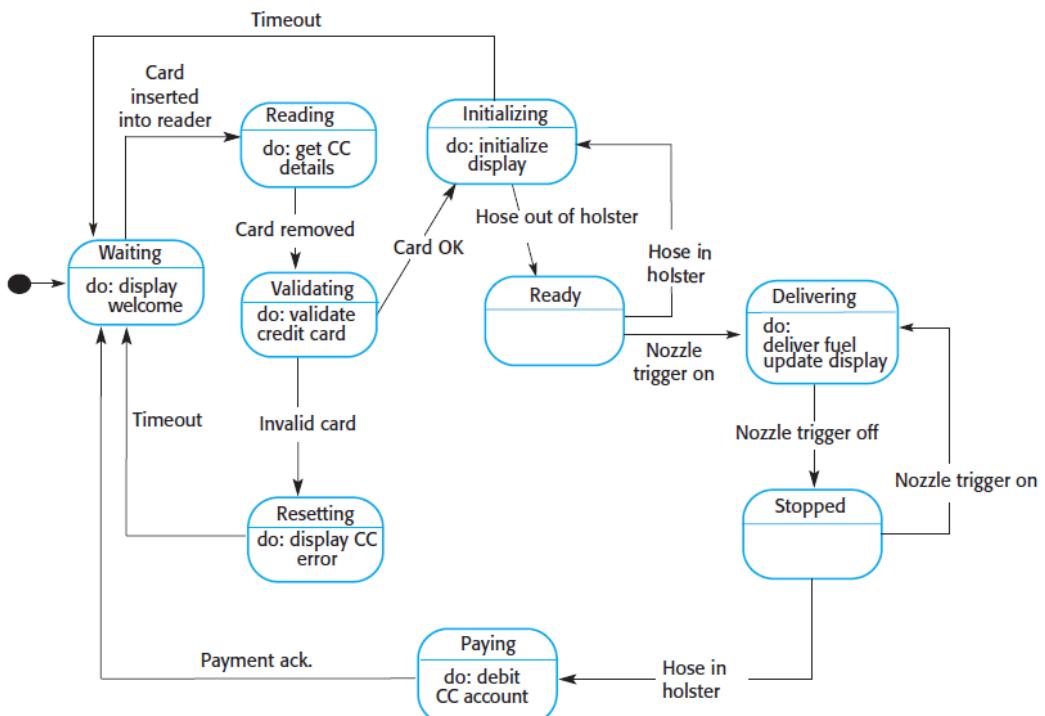


**Figure 4.26** Internal activities of the "Menu Visible" State



## Activity 4.7

Examine the petrol (gas) pump state machine diagram below and explain why there are no entry and exit events in the diagram.



(Source: Sommerville, I. (2016). *Software Engineering*)

### 2.3.3 Using the State Machine Diagram

A state machine diagram is useful for showing what messages an object may respond to. This helps the system analyst to design, implement or test a class. It provides a guide to what methods that the class of the object may need to have.

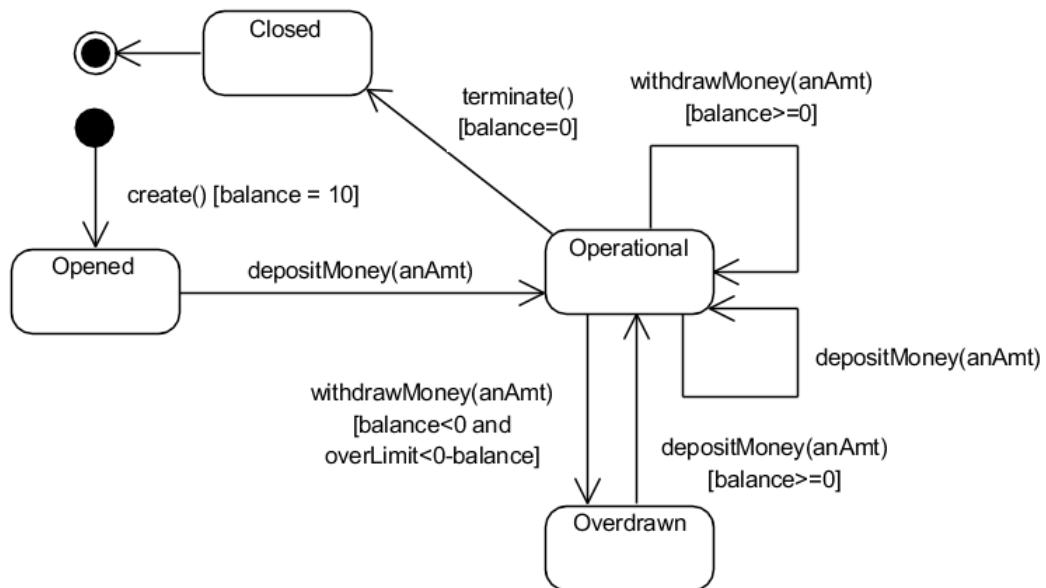
For example, a bank account object would need to have methods for the following actions:

- Create an account
- Deposit money
- Withdraw money
- Terminate account

The class would need to have instance variables for the following:

- Balance
- Over limit

By analysing how the methods would affect the status of a bank account, we would be able to have 4 states ("Opened", "Closed", "Operational" and "Overdrawn") which show the status of the bank account during its lifetime. Figure 4.27 shows the state machine diagram for the bank account.



**Figure 4.27** A state machine diagram shows the Messages that an object should understand



## Activity 4.8

Figure 4.28 shows a state machine diagram that models the behaviour of a stopwatch.

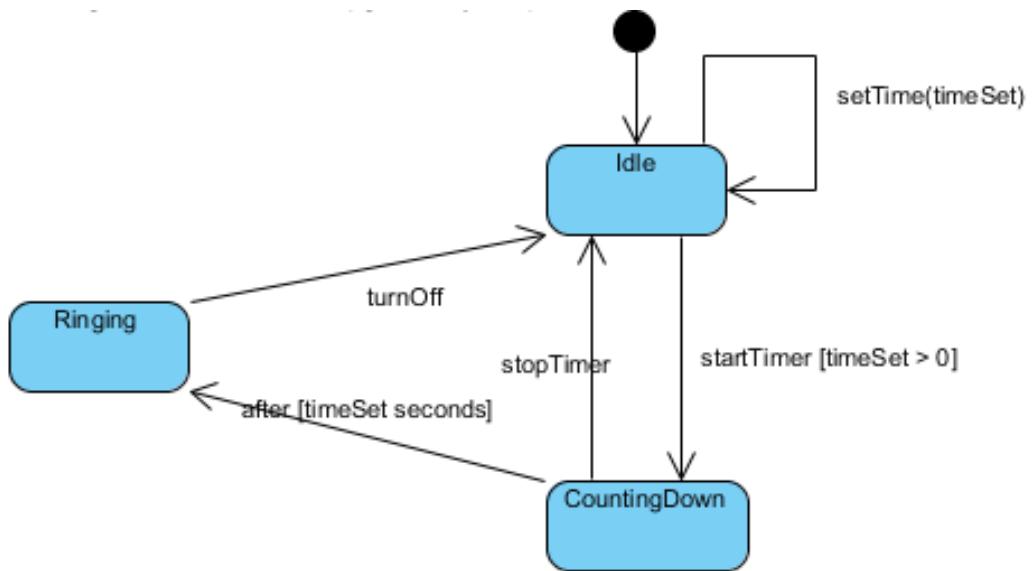


Figure 4.28 A state machine diagram that models a stopwatch

- Briefly describe what each state and transition in the state machine diagram represents.
- The stopwatch automatically stops ringing after 10 seconds. Modify the state machine diagram to incorporate this new requirement.

### 2.3.4 When to use the State Machine Diagram

The state machine diagram is good for showing the behaviour of an object across several use cases. Its focus is on a single object of a particular class. Therefore, it is not suitable for exploring behaviour where several objects interact among themselves.

State machine diagrams should be used only for classes that exhibit interesting behaviour.

They are not necessary if the values of an object's attributes can be readily changed in a simple manner.

Finally, note that we are not drawing a state machine diagram for the sake of drawing a state machine diagram. We are drawing the diagram to help us understand and analyse the application requirements.



## Activity 4.9

Consider the following life cycle of a snack vending machine:

"When the machine is first commissioned, it is idle. When a user selects the type of snack he wants, the machine requests payment. The user inserts coins until the correct amount or more has been entered. The snack is then dispensed, and any change is given. The transaction is now complete and the machine returns to the idle state. The machine can be decommissioned only when it is idle."

- a. Identify the states of the vending machine.
- b. What are the possible transitions between the states?
- c. Draw a state machine diagram for the snack vending machine.

## Chapter 3: Implementation of State Machine Diagram

### 3.1 Translating the State Diagram to Code

You have learnt how to implement classes for an application from your study of the ICT162 Object-Oriented Programming Course. This involves the implementations of instance variables and methods identified for the application.

Now, we will be adding to or modifying those classes to incorporate methods from the state diagram. We do this by first identifying the methods from the analysis performed with the state diagram and implementing these methods.

#### Example 1: Library Book

The best way to explain the implementations is to use an example. We will use the state machine diagram in Figure 4.29 which shows the state of a book in a library. To translate the state diagram to code, we will carry out the following three steps:

**Step 1:** Examine the state machine diagram and list the events that affect a Book object.

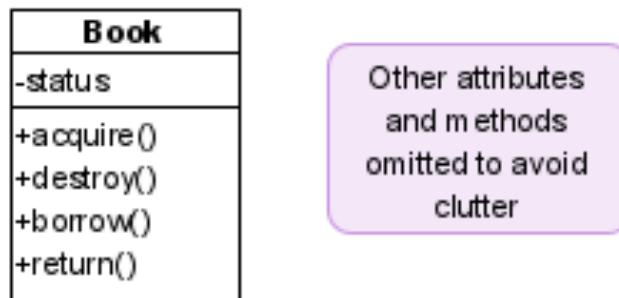
This produces the following table which also shows the state changes when the event happens.

**Table 4.2** State Changes for Events in Book class

Event	Action meaning	Start State	End State
acquire()	A new book is acquired	-	Available
destroy()	A book is removed from	Available	-

	the library collection		
borrow()	Book borrowed	Available	OnLoan
return()	Book returned	OnLoan	Available

**Step 2:** Modify the Book class to incorporate the events.



**Figure 4.29** A Book class with methods obtained from the state diagram

The events will be the additional methods in the class.

**Step 3:** Write the codes for the class Book.

### Source Code: Python

```
class Book:
    def __init__(self, nStatus, ...):
        # other instance variables not shown
        self.__status = nStatus

    # other methods omitted
    def acquire(self):
        self.__status = "Available"

    def destroy(self):
        if self.__status == "Available":
            # statements to implement destroy not shown
            pass
        else:
            print("Not able to destroy book that is on loan.")

    def borrow(self):
        if self.__status == "Available":
            self.__status = "OnLoan"
            # other statements not shown
        else:
            print("Not able to borrow book that is on loan.")

    def returnBook(self):
        if self.__status == "OnLoan":
            self.__status = "Available"
            # other statements not shown
        else:
            print("Not above to return book that is not on loan.")
```

Please take note of the following in the python source code given:

1. Other instance variables belonging to the class Book have been omitted from the python code.
2. Since return is a Python keyword, the method has been renamed returnBook.
3. It is necessary to have the if-else constructs to check that the Book has the correct state for the event before changing the state to avoid errors from arising.
4. Other statements to update the associations with other classes are omitted.

## Example 2: Bank Account

We shall use the bank account with its state machine diagram shown in Figure 4.27.

**Here is a short description of the state machine diagram of a typical bank account:**

A bank account is created with a balance of \$10 and is said to be in "Opened" state. When it is in the "Opened" state, money can be deposited into the bank account and it is then deemed to be in "Operational" state. When it is in "Operational" state, more money can be deposited into it. Money can also be withdrawn from the bank account. However, if the money withdrawn results in a negative balance or less than the over limit, the bank account is deemed to be in "Overdrawn" state. From "Overdrawn" state, the bank account can transit to "Operational" state only when money deposited results in a positive balance. While in "Operational" state, the bank account can be terminated only if the balance is 0. In this case, the bank account is deemed to be in "Closed" state.

**To translate the state machine diagram to code, we again carry out the 3 steps:**

**Step 1:** List the Events that affect the Bank Account.

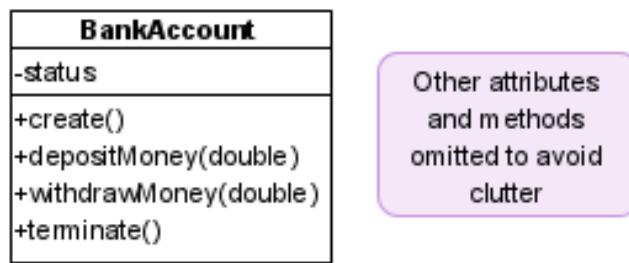
This produces the following table which also shows the state changes when the event happens.

**Table 4.3** State Changes for Events in Bank Account class

Event	Action meaning	Start State	End State
create()	A new bank account is created	-	Opened
depositMoney(anAmt)	anAmt is deposited	Opened or Operational	Operational

depositMoney(anAmt) [balance>=0]	anAmt is deposited which makes the balance positive	Withdrawn	Operational
withdrawMoney(anAmt) [balance>=0]	anAmt is withdrawn and balance is positive	Operational	Operational
withdrawMoney(anAmt) [balance<0 and overLimit<0- balance]	anAmt is withdrawn and balance is negative but within the over limit	Operational	Withdrawn
terminate() [balance=0]	Bank account is closed only when balance is 0	Operational	Closed

**Step 2:** Modify the Bank Account class to incorporate the events.



**Figure 4.30** A BankAccount class with methods obtained from the state diagram

Note that guard-conditions are not shown in the class diagram as these will be implemented in the methods.

**Step 3:** Write the codes for the class BankAccount.

Note that the code must be water-tight and handle all situations, i.e. it must be able to ensure that the status of the bank account is correct at any point in time, after a change in state.

## Source Code: Python

```

class BankAccount:
    def __init__(self, nStatus="Opened", nLimit=1000, ...):
        # other instance variables not shown
        self._balance = 0
        self._status = nStatus
        self._overLimit = nLimit

    # other methods not shown

    # create() method not necessary, can use constructor instead

    def terminate(self):
        if self._balance == 0:
            if self._status == "Operational":
                self._status = "Closed"
            else:
                print("Bank Account not Operational, cannot terminate.")
        else:
            print("Balance not 0, cannot terminate.")

    def depositMoney(self, anAmt):
        if self._status == "Opened" or self._status == "Operational":
            self._balance += anAmt
        elif self._status == "Withdrawn":
            if self._balance + anAmt >= 0:
                self._status == "Operational"
                print("Bank Account now Operational.")
                self._balance += anAmt
            else:
                print("Not able to deposit money.")

    def withdrawMoney(self, anAmt):
        if self._status == "Operational":
            balance = self._balance - anAmt
            if balance < 0 and self._overLimit < 0 - balance:
                self._status == "Withdrawn"
                self._balance = balance
            elif balance >= 0:
                self._balance = balance
            else:
                print("Not able to withdraw money.")
        else:
            print("Not able to withdraw money.")

```



## Activity 4.10

Consider the stopwatch in Activity 8.

- Create a table to show the state changes for events for the StopWatch class.

- b. Implement the StopWatch class, with the methods that change the state of the stopwatch.



## Activity 4.11

Consider the snack vending machine in Activity 9.

- a. Create a table to show the state changes for events for the SnackVMachine class.
- b. Implement the SnackVMachine class, with the methods that change the state of the snack vending machine.

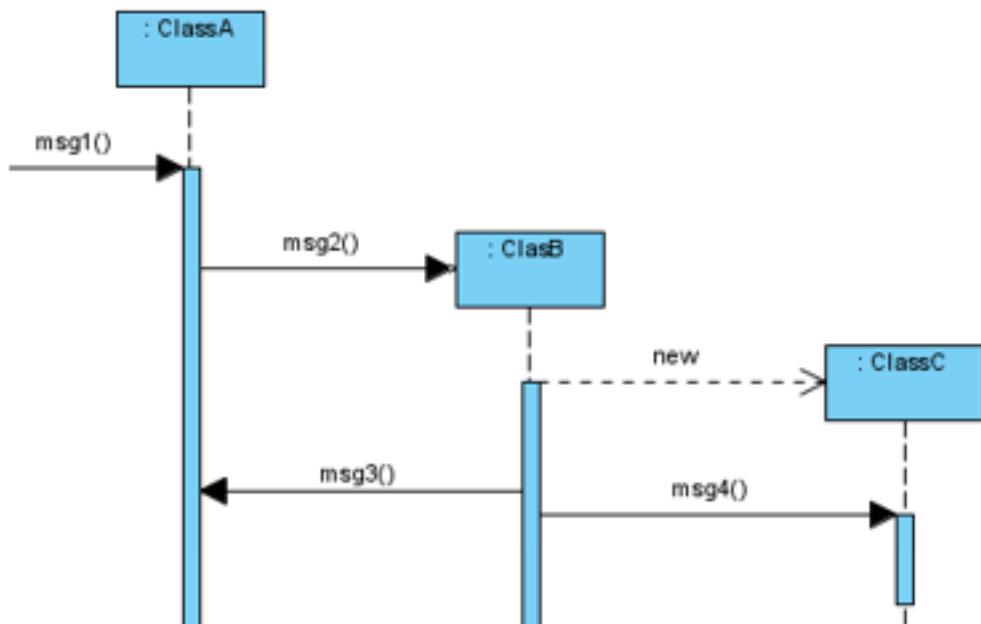
## Summary

The following points summarise the contents of this study unit:

- A sequence diagram can show where and when a new object is created or removed.
- The state of an object is the value of one or more of its significant attributes.
- We study the changes in the state of an object to uncover additional operations that need to be included in the structural model.
- The elements of a State Machine diagram are states, events, transitions and actions.
- A State Machine diagram shows the life cycle of an object, i.e. the different states of the object and the events that can cause the object to change from one state to another.
- The messages that an object can accept are the transitions that can change the state of an object.
- These messages may have guard conditions that allow a change of state only if the condition is true.
- Entry and exit events of a State show the actions that need to be done upon entry and exit to and from a State respectively.
- A State Machine diagram need not be drawn for every class / object. It is drawn only for complex objects to help simplify the design of algorithms.
- A State Machine diagram can be easily implemented by drawing up a table showing the events that change an object's state and implement the methods, taking into consideration the various guard conditions for each method.

## Formative Assessment

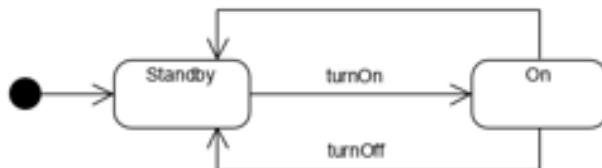
1. The following sequence diagram has errors. You may pick more than one mistake from the choices given.



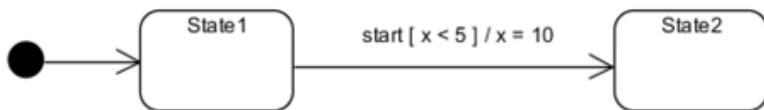
- a. ClassB object should be drawn at the same level as ClassA object.
  - b. ClassC object should be drawn at the same level as ClassA object.
  - c. msg3() should be a return message.
  - d. ClassB object cannot send msg4() to ClassC.
2. Which of the following CANNOT be modelled with a state machine diagram?
- The sequence of messages that fulfils a use case.
  - The possible states of an object.
  - Possible transitions from one state to another.
  - Activities that are executed while the object is in a certain state.
3. Which of the following is NOT allowed in a UML state machine diagram?
- A state machine diagram with one state.

- b. A state machine diagram with two initial states.
  - c. A state machine diagram with two final states.
  - d. A state containing a substate with no final state.
4. When all the possible ways of leaving a state must execute a common behaviour or sequence of behaviours, which of the following should be used?
- a. entry event
  - b. do action
  - c. exit event
  - d. initial state
5. The following state machine diagram models a power-saving device. When the object is created, it is in "Standby" state. When it is turned on, the state becomes "On". When it is turned off, it returns to "Standby" state. In addition, the device in "On" state returns to "Standby" state after 5 minutes.

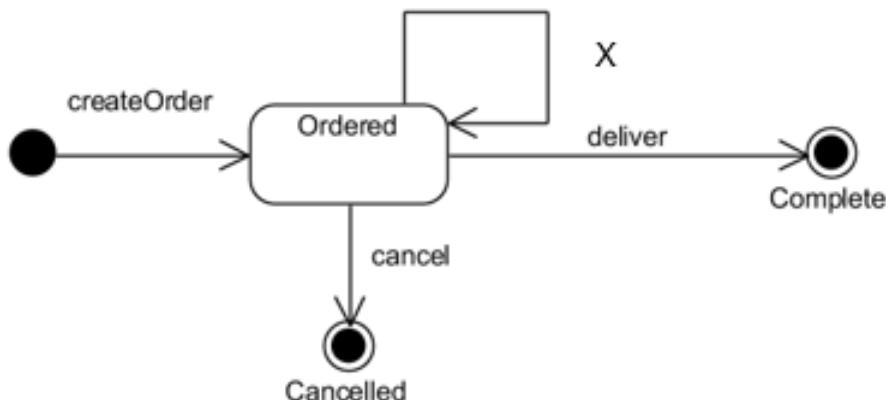
Which of the following is the correct missing event for the state machine diagram?



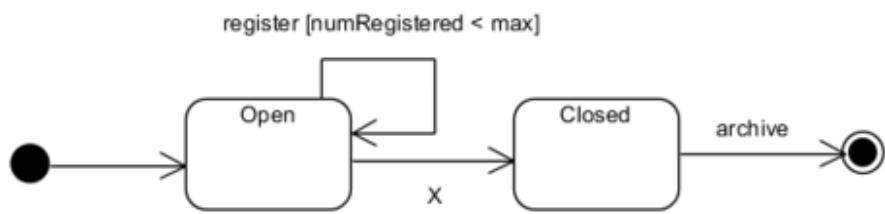
- a. 5 minutes
  - b. after [ 5 minutes ]
  - c. turnoff [ 5 minutes ]
  - d. turnOff
6. Consider the following state machine diagram. The object is in "State1" and the event start is called. Which of the following statements is true if x is 5?



- a. The object stays in "State1" and x remains 5.
- b. The object stays in "State1" and x becomes 10.
- c. The object transits to "State2" and x remains 5.
- d. The object transits to "State2" and x remains 10.
7. A pizza delivery service promises that all orders will be free if they are not delivered within 30 minutes. Which of the following is appropriate for the transition labelled X?



- a. setPrice(0)
- b. after [ 30 mins ] / setPrice(0)
- c. setPrice(0) [ price > 0 ]
- d. setPrice(0) / cancel
8. The state machine diagram below models a Seminar. When it is "Open" for registration, students can register for it. Once the maximum number of students has registered, it becomes "Closed". Which of the following is appropriate for the transition labelled X?



- a. register
- b. register [ numRegistered == max ]
- c. after [ numRegistered == max ]
- d. [ numRegisterd == max ] / register

# Solutions or Suggested Answers

## Activity 4.1

From examining the sequence diagram, the objects with the roles for the getManager() message are:

<u>Object</u>	<u>Role</u>
Admin object	client
Branch object	server, collaborator

## Activity 4.2

We would need to add a method such as the following to the orchestrating class Admin.

reassign(name: string, location: string) : void

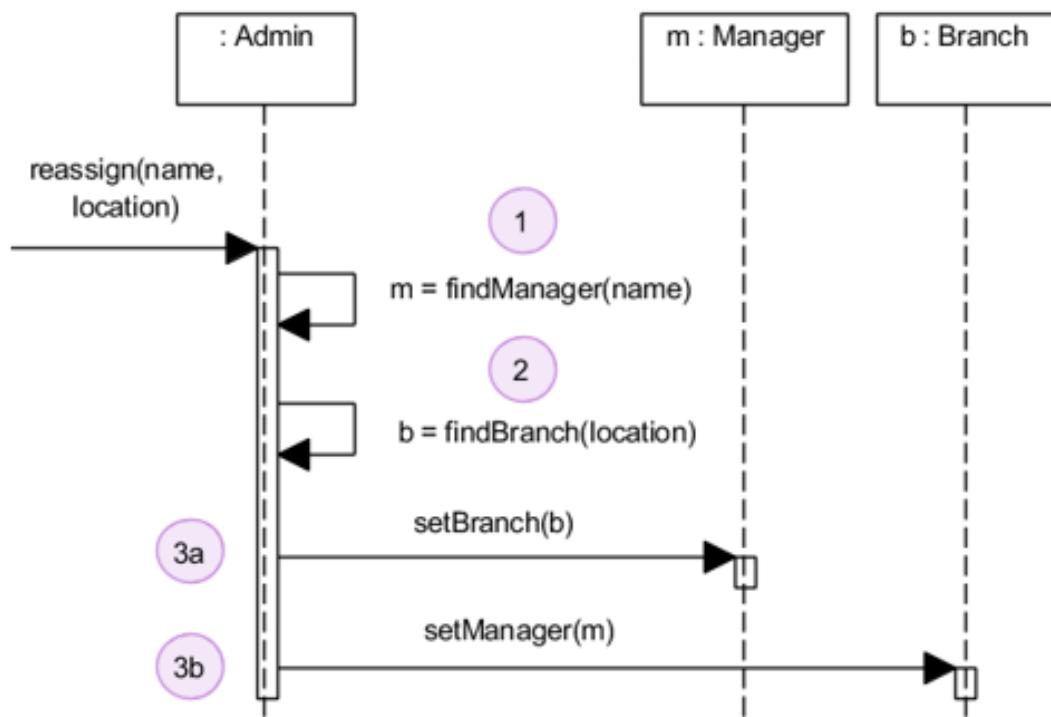
The following would be the walkthrough:

Objective: To reassign a branch to a manager

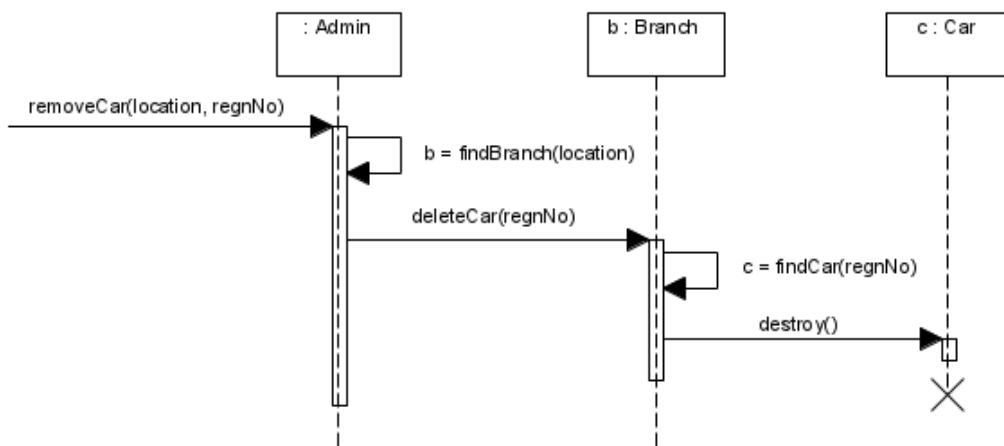
Given: The name of a manager and the location of a branch

1. Locate the Manager object with the specified name, linked to the orchestrating object via the association hasManagers
2. Locate the Branch object with the specified location, linked to the orchestrating object via the association hasBranches
3. Create the association between the Manager object and the Branch object

The Admin object is able to send the messages for both steps 3a and 3b because it knows about the Manager and Branch objects found in steps 1 and 2.



### Activity 4.3



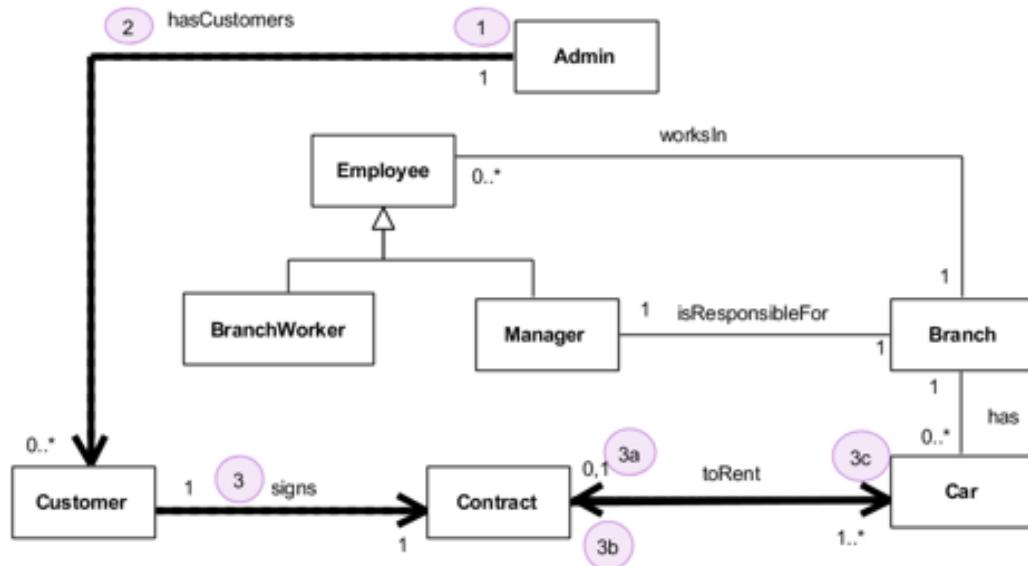
Note that the `destroy()` message does not have a reply as the object is destroyed and can no longer send any (even reply) messages.

## Activity 4.4

a. The methods are:

<u>Class</u>	<u>Methods</u>
Admin	removeCarFromContract(custName: String, regnNo: String): void getCustomer(custName: String): Customer
Customer	deleteCar(regnNo: String): void getContract(): Contract
Contract	deleteCar(regnNo: String): void searchCar(regnNo: String): Car removeCar(regnNo: String): void
Car	setContract(c: Contract): void

b.



- c. The association toRent between Car and Contract is bi-directional. Therefore, to remove the association, the values in both classes have to be modified.
- d. The multiplicity for the association toRent between Contract and Car is one to many. A contract can have many cars. Step 3a is needed to look for the object aCar. Step 3b removes the car from the collection object. Step 3c uses the setter method to remove the contract from the car since a car can be part of only one contract.

## Activity 4.5

We need to add a method such as the following to the orchestrating class Admin.

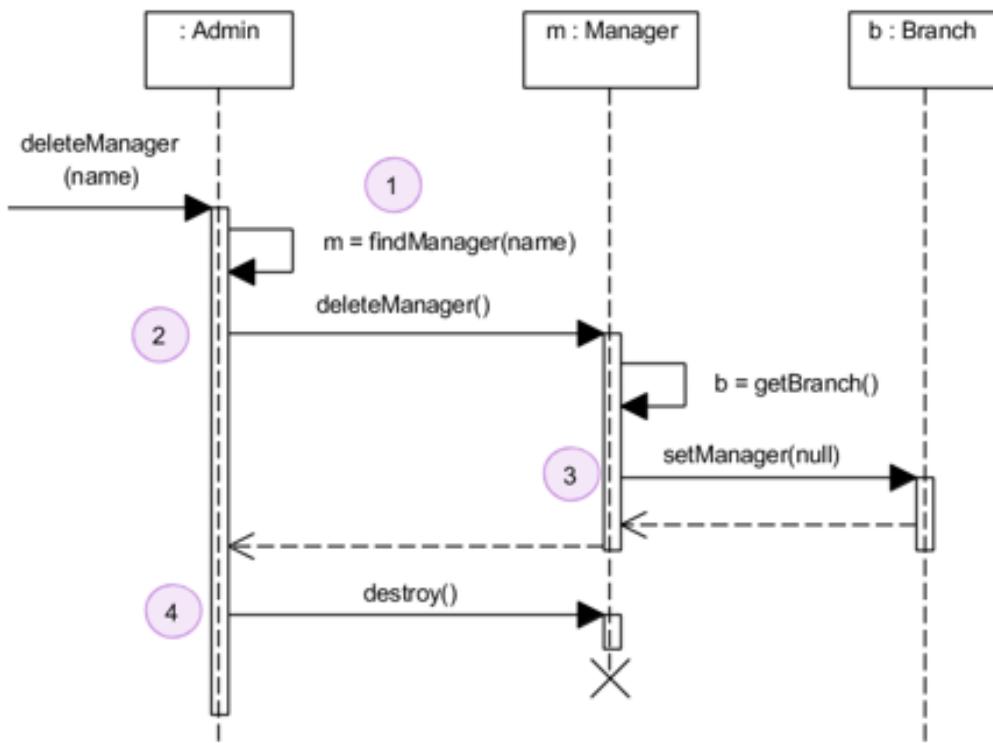
```
deleteManager(name: string) : void
```

The following would be the walkthrough:

Objective: To delete a manager.

Given: The name of a manager

1. Locate the Manager object with the specified name, linked to the orchestrating object via the association hasManagers.
2. Locate the Branch object linked via the association isResponsibleFor to the Manager object found in step 1.
3. Delete the association between the Manager object and the Branch object.
4. Destroy the Manager object.



## Activity 4.6

When the weather station starts, it is in "Shutdown" state. When the events `reconfigure()` and `powerSave()` have been executed, it transits to "Configuring" state. Once the configuration is done, it transits to "Running" state, which is one of the substates of "Operation" state. The "Operation" state has 5 substates – "Running", "Collecting", "Summarizing", "Transmitting" and "Testing".

While the weather station is in "Operation" state, when the clock is executed, it transits from "Running" substate to "Collecting" substate. When collection is done, it transits from "Collecting" substate to "Running" substate. The `reportWeather()` event triggers the transition from "Running" substate to "Summarizing" substate. When the weather summary is complete, from "Summarizing" substate, it transits to "Transmitting" substate. When transmission is done, it transits from "Transmitting" substate to "Running" substate. When in the "Running" substate, three events can happen: (1) the `remoteControl()` can

be executed and it goes to the "Controlled" state; (2) the reportStatus() can be executed and it goes to the "Testing" substate. When the test is complete, it transits to the "Transmitting" substate; (3) when shutdown() is executed, the weather machine goes from the "Running" / "Operation" state to the "Shutdown" state. While in the "Shutdown" state, the restart() event can trigger the weather station to "Operation" state and its "Running" substate.

## Activity 4.7

The entry events are shown as transition arrows going into a state with the event name written as labels.

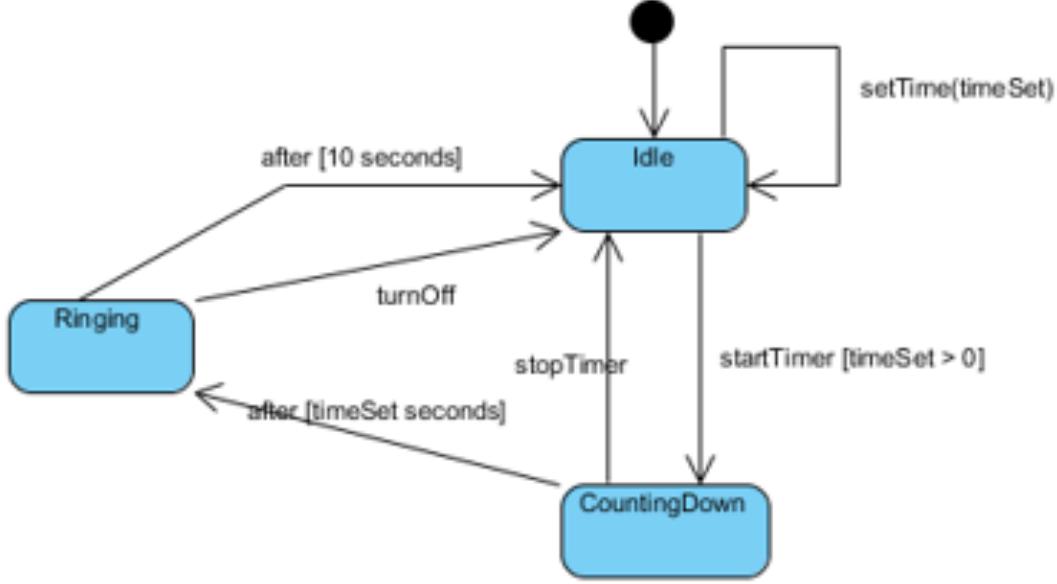
Similarly, the exit events are shown as transition arrows going out of a state with the event name written as labels.

This is because all entry and exit events are different. You do not need to write them as internal events in the state.

## Activity 4.8

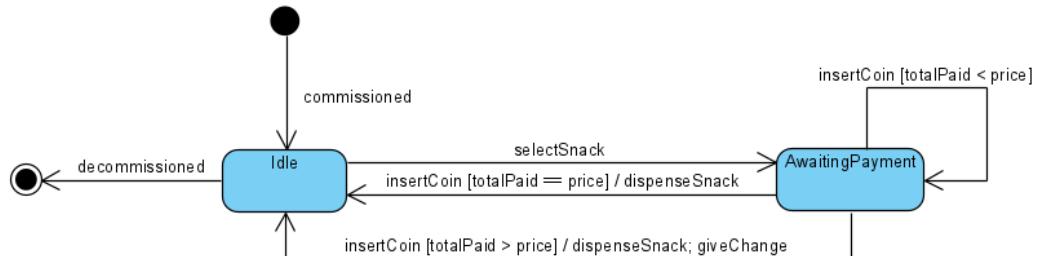
- a.
  - "Idle" state: the Stopwatch is not performing any action. It may or may not be set.
  - "CountingDown" state: the Stopwatch is counting down.
  - "Ringing" state: the Stopwatch is ringing.
  - setTime(timeSet): sets the Stopwatch to ring after timeSet seconds once it is turned on.
  - startTimer: starts the timer to count down. If timeSet is zero or less, nothing happens.

- stopTimer: stops the countdown of the time. Note: for the stopwatch to work properly, the value of timeSet should be updated to be the time remaining.
  - after [timeSet seconds]: the Stopwatch rings after the given number of seconds.
  - turnOff: switches the ringing Stopwatch off.
- b. Add a new trigger "after [10 seconds]" from Ringing to Idle:



## Activity 4.9

- States: "Idle" and "Awaiting Payment"
- See transitions in diagram
- c.



### Activity 4.10

a.

Event	Action meaning	Start State	End State
setTime(timeSet)	Timer is set with time value timeSet	Idle	Idle
stopTimer()	Stops Timer	CountingDown	Idle
startTimer()	Starts Timer	Idle	CountingDown
turnOff()	Turn off Timer	Ringing	Idle
after [timeSet seconds]	Finished counting down	CountingDown	Ringing

- b. The after [timeSet seconds] event is omitted in the StopWatch class since it is an external event – the system clock triggers this event.

```

class Stopwatch:
    def __init__(self, ...):
        # other instance variables not shown
        self._timeSet = 0
        self._status = "Idle"

    # other methods not shown

    def setTime(self, timeSet):
        self._timeSet = timeSet

    def stopTimer(self):
        if self._status == "CountingDown":
            self._timeSet = 0
            self._status = "Idle"
        else:
            print("Cannot stop timer since not counting down.")

    def startTimer(self):
        if self._timeSet > 0:
            if self._status == "Idle":
                self._status = "CountingDown"
            else:
                print("Time not set.")

    def turnOff(self):
        if self._status == "Ringing":
            self._status = "Idle"
        else:
            print("Cannot turn off when not ringing.")

```

## Activity 4.11

a.	Event	Action meaning	Start State	End State
	commissioned()	Snack Vending Machine starts operation	-	Idle
	decommissioned()	Snack Vending Machine stops operation	Idle	-

selectSnack()	A snack is chosen	Idle	AwaitingPayment
insertCoin() [ totalPaid < price ]	Coin inserted	AwaitingPayment	AwaitingPayment
insertCoin() [totalPaid == price]	Coin inserted	AwaitingPayment	Idle
insertCoin() [ totalPaid > price ]	Coin inserted	AwaitingPayment	Idle
dispenseSnack()	Snack dispensed	-	-
giveChange()	Change given	-	-

- b. The following assumptions are given for the program below:

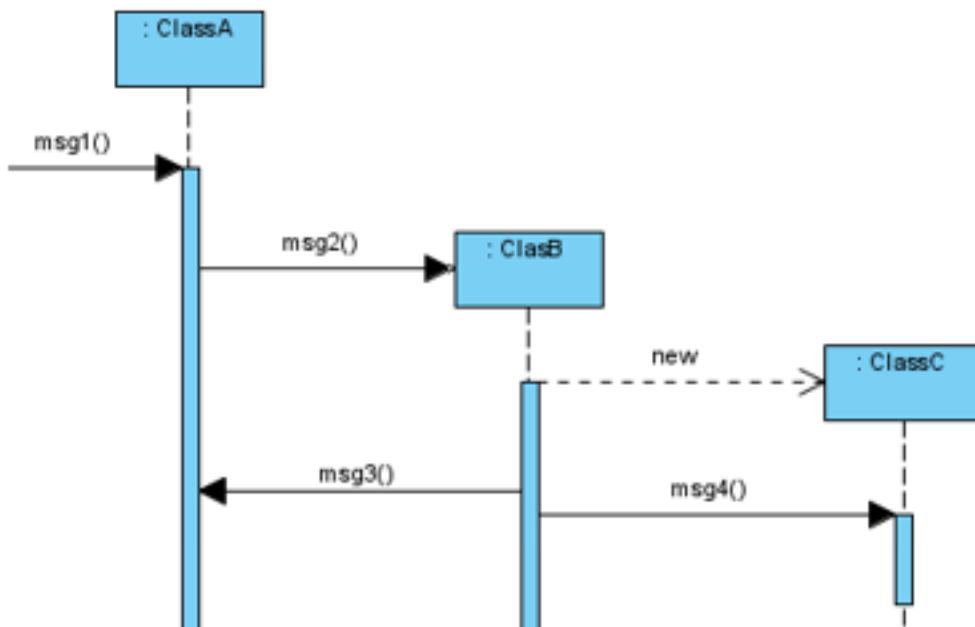
- commissioned() is the constructor (`__init__` function).
- decommissioned() is not implemented as objects can be destroyed using the `del` operator.
- There is only one (type of) snack in the Snack Vending Machine.
- `insertCoin()` has a parameter amount to specify the amount inserted.
- The Snack Vending Machine does not run out of snacks.

Hence the state machine diagram should be updated to reflect the program implemented.

```
class SnackVMachine:  
    def __init__(self, aPrice, aQty, ...):  
        # other instance variables not shown  
        self._price = aPrice  
        self._qty = aQty  
        self._totalPaid = 0  
        self._status = "Idle"  
  
    # other methods not shown  
  
    def selectSnack(self):  
        if self._status == "Idle":  
            self._status == "AwaitingPayment"  
  
    def insertCoin(self, amount):  
        if self._status == "AwaitingPayment":  
            self._totalPaid += amount  
            if self._totalPaid < self._price:  
                print("Awaiting payment")  
            else:  
                if self._totalPaid > self._price:  
                    giveChange()  
                dispenseSnack()  
                self._status = "Idle"  
  
    def dispenseSnack(self):  
        self._qty -= 1  
        print("Snack dispensed.")  
  
    def giveChange(self):  
        print("Change given: {self._totalPaid - self._price}")
```

## Formative Assessment

1. The following sequence diagram has errors. You may pick more than one mistake from the choices given.



- a. ClassB object should be drawn at the same level as ClassA object.

**Correct. ClassA object sends msg2() to ClassB object.**

- b. ClassC object should be drawn at the same level as ClassA object.

Incorrect. ClassB object is creating a new ClassC object (dotted arrow with new keyword). So it should be drawn at the point it was created.

- c. msg3() should be a return message.

**Correct. ClassB object cannot spontaneously send a message to the sender object unless it is sending a return message to the sender.**

- d. ClassB object cannot send msg4() to ClassC.

**Correct. If msg3() is a return message, then ClassB object is no longer "live" and hence cannot send messages to other objects.**

2. Which of the following CANNOT be modelled with a state machine diagram?

- a. The sequence of messages that fulfils a use case.

**Correct. The state machine diagram models behaviour of an object based on events, not on messages in a use case. Refer to Study Unit 4, Section 1.2.**

- b. The possible states of an object.

Incorrect. The state machine diagram shows all possible states of an object in its life cycle. Refer to Study Unit 4, Section 1.2.

- c. Possible transitions from one state to another.

Incorrect. The state machine diagram shows the transitions that are possible between states. Refer to Study Unit 4, Section 1.2.

- d. Activities that are executed while the object is in a certain state.

Incorrect. The state machine diagram shows after an event occurs, the response or activities that are required from the object. Refer to Study Unit 4, Section 1.2.

3. Which of the following is NOT allowed in a UML state machine diagram?

- a. A state machine diagram with one state.

Incorrect. A state machine diagram must have minimally one state. Refer to Study Unit 4, Section 2.2.1.

- b. A state machine diagram with two initial states.

**Correct. A state machine diagram can only have one initial state. Refer to Study Unit 4, Section 2.2.1.**

- c. A state machine diagram with two final states.

Incorrect. A State machine diagram can have many final states for a clearer diagram. Refer to Study Unit 4, Section 2.2.1.

- d. A state containing a substate with no final state.

Incorrect. A substate can have no final state. Refer to Study Unit 4, Section 2.2.1.

4. When all the possible ways of leaving a state must execute a common behaviour or sequence of behaviours, which of the following should be used?

- a. entry event

Incorrect. This is for entry to the state and not exit of the state. Refer to Study Unit 4, Section 2.3.2.

- b. do action

Incorrect. This specifies what must be done when the object is in that state. Refer to Study Unit 4, Section 2.3.2.

- c. exit event

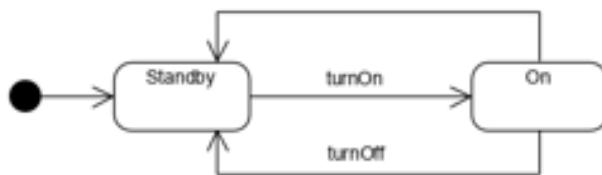
**Correct. The exit event is written once inside the state. Refer to Study Unit 4, Section 2.3.2.**

- d. initial state

Incorrect. This indicates where the object begins its existence. Refer to Study Unit 4, Section 2.1.1.

5. The following state machine diagram models a power-saving device. When the object is created, it is in "Standby" state. When it is turned on, the state becomes "On". When it is turned off, it returns to "Standby" state. In addition, the device in "On" state returns to "Standby" state after 5 minutes.

Which of the following is the correct missing event for the state machine diagram?



- a. 5 minutes

Incorrect. When the device is in On state, it returns to Standby state after 5 minutes. Refer to Study Unit 4, Section 2.1.1.

- b. after [ 5 minutes ]

**Correct. When the device is in On state, it returns to Standby state after 5 minutes. Refer to Study Unit 4, Section 2.1.1.**

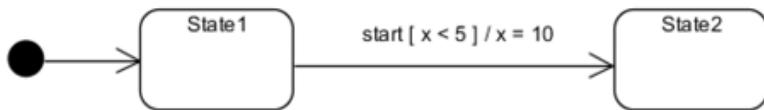
- c. turnoff [ 5 minutes ]

Incorrect. Does not need to turn off the device. Refer to Study Unit 4, Section 2.1.1.

- d. turnOff

Incorrect. Does not need to turn off the device. Refer to Study Unit 4, Section 2.1.1.

6. Consider the following state machine diagram. The object is in "State1" and the event start is called. Which of the following statements is true if x is 5?



- a. The object stays in "State1" and x remains 5.

**Correct. Even though the event start is called, the guard condition is not true. Therefore there is no change in state and x. Refer to Study Unit 4, Section 2.2.1.**

- b. The object stays in "State1" and x becomes 10.

Incorrect. Even though the event start is called, the guard condition is not true. Therefore  $x=10$  is not executed and there is no change in state and x. Refer to Study Unit 4, Section 2.2.1.

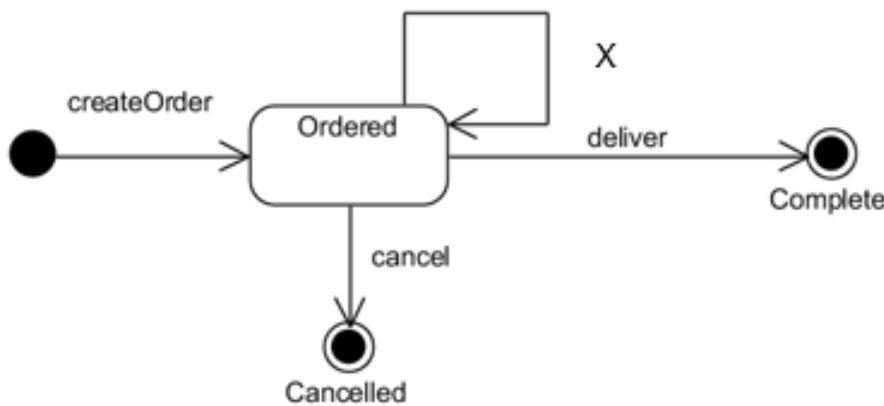
- c. The object transits to "State2" and x remains 5.

Incorrect. Even though the event start is called, the guard condition is not true. Therefore there is no change in state. Refer to Study Unit 4, Section 2.2.1.

- d. The object transits to "State2" and x remains 10.

Incorrect. Even though the event start is called, the guard condition is not true. Therefore  $x=10$  is not executed and there is no change in state and  $x$ . Refer to Study Unit 4, Section 2.2.1.

7. A pizza delivery service promises that all orders will be free if they are not delivered within 30 minutes. Which of the following is appropriate for the transition labelled X?



- a. setPrice(0)

Incorrect. The condition that the pizza is delivered within 30 minutes needs to be considered. Refer to Study Unit 4, Section 2.3.1.

- b. after [ 30 mins ] / setPrice(0)

**Correct. The guard condition must be true before the price is set to 0. Refer to Study Unit 4, Section 2.3.1.**

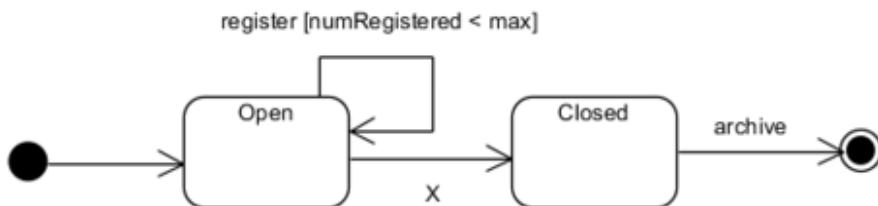
- c. setPrice(0) [ price > 0 ]

Incorrect. The guard condition should check for after 30 minutes and not the price. Refer to Study Unit 4, Section 2.3.1.

- d. setPrice(0) / cancel

Incorrect. There is no mention that the pizza will be cancelled, instead the pizza is to be delivered within 30 minutes. Refer to Study Unit 4, Section 2.3.1.

8. The state machine diagram below models a Seminar. When it is "Open" for registration, students can register for it. Once the maximum number of students has registered, it becomes "Closed". Which of the following is appropriate for the transition labelled X?



- a. register

Incorrect. Transitions that exit a state that have the same event require guard conditions. Refer to Study Unit 4, Section 2.3.1.

- b. register [ numRegistered == max ]

**Correct. Transitions that exit a state that have the same event require guard conditions that are mutually exclusive. Refer to Study Unit 4, Section 2.3.1.**

- c. after [ numRegistered == max ]

Incorrect. after is a time event and requires a guard condition concerning time. Refer to Study Unit 4, Section 2.3.1.

- d. [ numRegisterd == max ] / register

Incorrect. Transitions that exit a state that have the same event require guard conditions. Refer to Study Unit 4, Section 2.3.1.

## References

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.



# Study Unit 5

## Implementations, Integration and Testing

## Learning Outcomes

By the end of this unit, you should be able to:

1. write programs for the classes in the structural model
2. write programs for the classes from walkthrough analysis
3. write programs for the classes in the dynamic model
4. describe the need to keep user interface independent from the application
5. show how a user interface is incorporated in an application
6. describe the goals of testing and various testing required for a system
7. explain three stages of testing
8. discuss how the object-oriented approach makes it easier to do integration testing

## Overview

We have discussed the following stages of software engineering design so far:

### **Unit 1:** Requirements engineering

- Produced requirements specification document (use cases, activity diagrams)

### **Unit 2:** Generating structural models

- Class description diagram (classes, attributes and generalisation relationships)
- Class association diagram (associations, multiplicities in associations, aggregations and compositions)

### **Unit 3:** Generating dynamic models

- Added additional attributes and methods using Walkthroughs and Sequence diagrams to enable dynamic interaction of the system with its environment.

### **What we will be doing in this unit**

Based on the design outputs from structural and dynamic models, we will implement the code in Python to realise the system in this unit. We will also cover testing.

#### This unit consists of three main sections:

- Implementations
  - Writing the code for the classes from structural or static modelling
  - Adding the code to the classes from the walkthroughs in dynamic modelling
  - Adding the code to the classes from the sequence diagrams in dynamic modelling
- Incorporating the user interface for the application
- Reviewing the testing required for a system

**Read**

This study guide is developed based on Part 1 (Chapters 1 – 9) and Part 4 (Chapters 22 – 25) of Ian Sommerville (2016). *Software Engineering, 10th Edition*.

## Chapter 1: Implementation

### 1.1 Writing Codes for a Structural Model



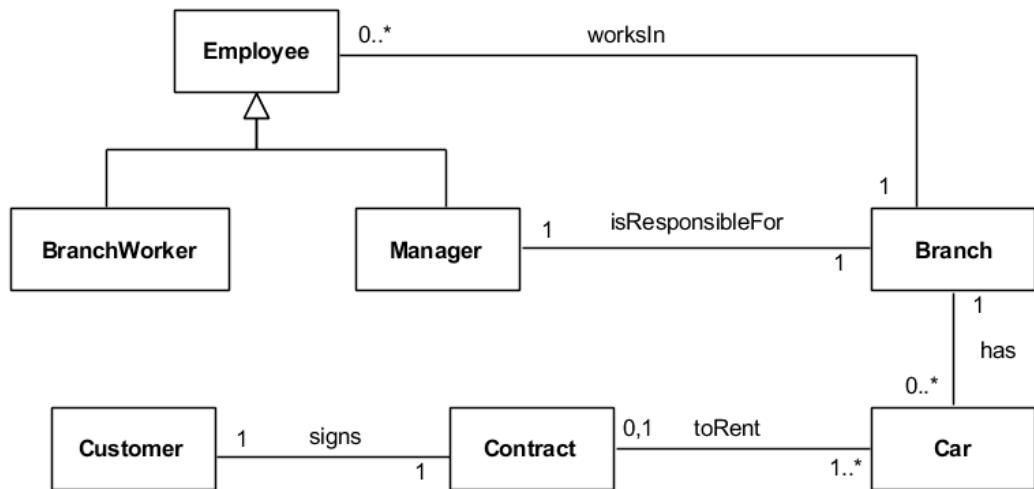
The structural model consists mainly of the following:

- The class association diagram
- The class description

The class description consists of:

1. A list of potential classes,
2. Their superclass-subclass relationships, if any,
3. The attributes that can be identified from the requirements,
4. The constraints or invariants.

The class association diagram shows how the classes should be associated with each other. It reflects our understanding of the requirements at this stage. Our interpretations that give us these associations might need to be refined subsequently when we perform the dynamic analysis. Figure 5.1 shows the class association diagram from the structural model of the car rental application discussed in previous study units.



**Figure 5.1** The class association diagram for car rental application

The following is the class description of the same application:

<b>Class:</b>	Branch
<b>Attributes:</b>	location, the location of the branch status, the status (operational or under renovation) of the branch
<b>Class:</b>	Car
<b>Attributes:</b>	engineCapacity, the engine capacity of the car registrationNumber, the vehicle registration number of the car
<b>Class:</b>	Employee, superclass of BranchWorker and Manager
<b>Attributes:</b>	name, the name of the employee employeeNumber, the employee number of the employer
<b>Class:</b>	BranchWorker, subclass of Employee

<b>Attributes:</b>	hourlyRate, the hourly overtime rate of the branch worker
<b>Class:</b>	Manager, subclass of Employee
<b>Attributes:</b>	refer to superclass
<b>Class:</b>	Customer
<b>Attributes:</b>	name, the name of the customer address, the address of the customer
<b>Class:</b>	Contract
<b>Attributes:</b>	startDate, the starting date of the contract durationOfContract, the length of the contract

We shall implement the classes in Python according to this class description. At this stage, we will assume that the constructors will assign initial values for the given instance variables.

The following are the codes for the classes Branch and Car based on the class description:

## Source Code: Python

```
class Branch:  
    def __init__(self, newLocation, newStatus):  
        self._location = newLocation  
        self._status = newStatus  
  
    @property  
    def location(self):  
        return self._location  
  
    @location.setter  
    def location(self, newLocation):  
        self._location = newLocation  
  
    @property  
    def status(self):  
        return self._status  
  
    @status.setter  
    def status(self, newStatus):  
        self._status = newStatus  
  
  
class Car:  
    def __init__(self, nEngineCapacity, nRegnNo):  
        self._engineCapacity = nEngineCapacity  
        self._regnNo = nRegnNo  
  
    @property  
    def regnNo(self):  
        return self._regnNo  
  
    @regnNo.setter  
    def regnNo(self, nRegnNo):  
        self._regnNo = nRegnNo  
  
    @property  
    def engineCapacity(self):  
        return self._engineCapacity  
  
    @engineCapacity.setter  
    def engineCapacity(self, nEngineCapacity):  
        self._engineCapacity = nEngineCapacity
```



## Activity 5.1

Complete the code for the class description by writing the programs for the following classes:

- Employee
- BranchWorker
- Manager
- Customer
- Contract

## 1.2 Implementing Dynamic Models from Walkthrough Analysis

In the previous section, we have implemented the classes for the car rental application from the structural or static analysis. This involves the implementations of instance variables and methods identified from analysing the text of the application requirements.

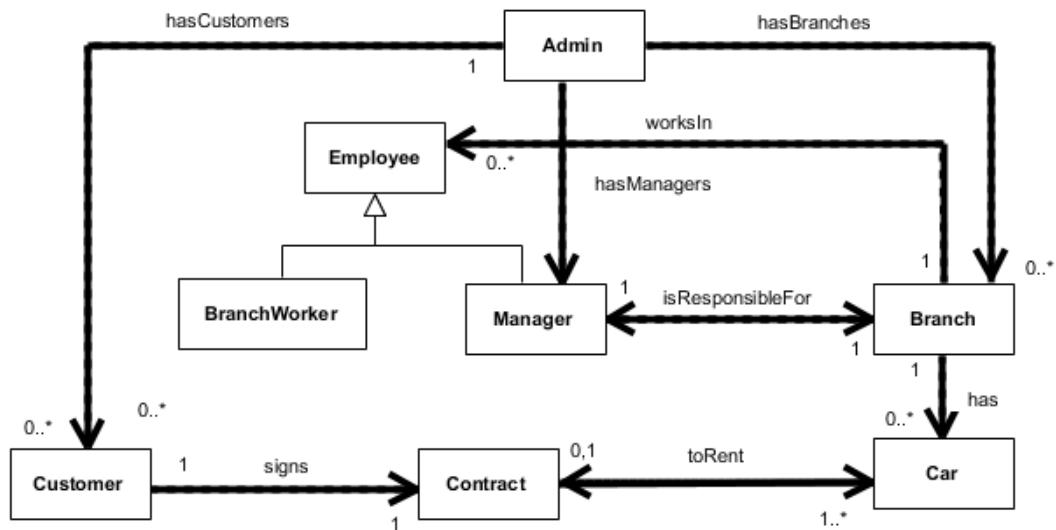
Now, we will be adding to or modifying those classes to incorporate instance variables and methods derived from our dynamic analysis. We will do this in two parts:

1. Implementing the instance variables from the walkthroughs performed for the use cases.
2. Implementing the methods from the analysis performed with the sequence diagrams.

In this section, we will focus on the first part: implementing the instance variables. In the next section, we will implement the methods by writing the codes for them.

### 1.2.1 Implementing Associations

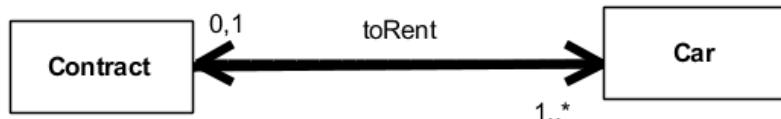
The best way to explain the implementations is to use an example. We will use the car rental application again for our discussions. These discussions will be based on the association diagram in Figure 5.2.



**Figure 5.2** The class association diagram for car rental application

In Figure 5.2, the navigation arrows in the associations show that we have performed the walkthrough analysis for a few use cases. Our task now is to implement these associations according to the results of the walkthrough analysis.

From the class association diagram, we select the following association for implementation.



**Figure 5.3** A fragment of a class association diagram

As a result of the dynamic analysis with the walkthroughs, we found that the classes Contract and Car have to keep track of each other in the application. You can see this in the bi-directional navigation of the association toRent.

The classes Contract and Car are therefore modified as shown in the table below:

Class being modified	Contract	Car
Association being implemented	toRent	toRent
Direction of navigation	from Contract to Car	from Car to Contract
Multiplicity	one to many	one to zero, or one to one
Instance variable added	cars	contract
Value of the instance variable	a Dictionary/HashMap object	either null or a Contract object

This shows that to implement the toRent association, we need to add an instance variable to both classes Contract and Car. The following shows how we would write the code for this implementation. We will separate it into two parts so that it is clear what each part is for:

**Part 1: The code for the class Car**

**Part 2: The code for the class Contract**

**Part 1: The code for the class Car**

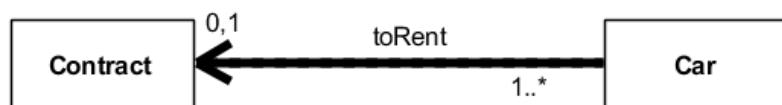


Figure 5.4 Implementing the association in the class Car

This is relatively straightforward since the multiplicity is either zero or one. The modifications required are to add the following code to the class Car:

### Source Code: Python

```

class Car:
    def __init__(self, ..., nContract=None):
        # other instance variables not shown
        self._contract = nContract

    # other methods not shown

    @property
    def contract(self):
        return self._contract

    @contract.setter
    def contract(self, nContract):
        self._contract = nContract

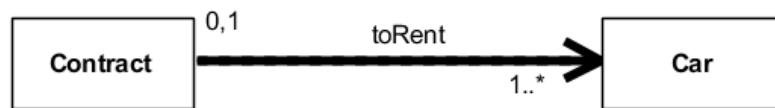
```

Recall that **instance variables with single values commonly come with a pair of accessor and mutator methods. We have included these for Contract accordingly.**

Note that the constructor of Car defines the contract to have a default value, None, if a car is not part of a contract. We could also assign the instance variable contract to be None, if at any point in time there is no contract associated with the car:

```
self._contract = None;
```

### Part 2: The code for the class Contract



**Figure 5.5** Implementing the association in the class Contract

Since the multiplicity is many for the class Car and we need to look for a car using its registration number, we have chosen to use a Dictionary as the collection object. For collections, instead of simple accessor and mutator methods, we need a more comprehensive set of data manipulation facilities, such as the following:

1. Counting the number of cars in a contract
2. Verifying whether a car is present
3. Searching for a particular car
4. Listing and displaying the cars in a contract
5. Adding a car to a contract
6. Removing a car from a contract

The following shows the implementation:

### Source Code: Python

```

class Contract:
    def __init__(self, nStartDate, nDuration):
        # other instance variables not shown
        self._carsDict = {}

    # other methods not shown

    def getNumberOfCars(self):
        return len(self._carsDict)

    def addCar(self, aCar):
        self._carsDict[aCar.regnNo] = aCar

    def hasCar(self, aCar):
        return aCar.regnNo in self._carsDict

    def searchCars(self, aRegnNo):
        return self._carsDict[aRegnNo]

    def listCars(self):
        print(self._carsDict.items())

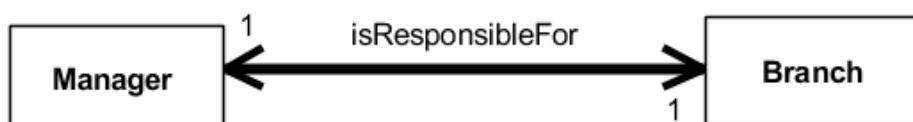
    def removeCar(self, aRegnNo):
        del self._carsDict[aRegnNo]

```



### Activity 5.2

You are given the following fragment of a class association diagram from the car rental application.



Class being modified	Manager	Branch
Association being implemented	isResponsibleFor	isResponsibleFor

Direction of navigation		
Multiplicity		
Instance variable added		
Value of the instance variable		
Justification		

Complete the table below and write the code for the two classes, Manager and Branch based on the given diagram.

Use the convention that we have adopted in this course for your variable names. Use the relevant classes generated to implement the 'toRent' association.

## 1.3 Implementing Dynamic Models with Sequence Diagrams

So far, we have implemented the classes for the car rental application from the following analysis:

- **Structural analysis:** implementing the instance variables and methods from analysing the text of the application requirements.
- **Dynamic analysis:** implementing the instance variables from the walkthroughs performed for the use cases.
- In this section, we will continue with the implementations from our dynamic analysis. Now, we will implement the methods that we have identified from our analysis with sequence diagrams.

As our first example, we will examine the use case "Assign a new manager to a branch". We have performed the walkthrough and drawn the sequence diagram for this use case in the previous study units and arrived at the following messages (methods) to be implemented:

Class	Methods
Admin	assignNewManager(location: String, employeeName: String, employeeNumber: String): void findBranch(location: String): Branch
Branch	assignManager(aManager: Manager): void setManager(aManager: Manager): void
Manager	setBranch(aBranch: Branch): void

Notice that we have left out the message corresponding to step 2 of the walkthrough shown in the sequence diagram. This is the message for constructing the new Manager object:

new Manager (employeeName: String, employeeNumber: String): Manager

We are deliberately deferring our discussion involving constructors to focus on the methods.

## Coding the Methods

### 1. setBranch(aBranch: Branch): void

This is simply the set method for the instance variable branch in the class Manager. We should have previously coded this method to implement the bi-directional navigation for the association isResponsibleFor. There is no new information for modifying it. We shall leave it as a standard set method.

### 2. setManager(aManager: Manager): void

This is the set method for the instance variable manager in the class Branch. Again, we should have coded this method previously to implement the bi-directional navigation for the association isResponsibleFor.

### 3. assignManager(aManager: Manager): void

To write the code for the method, we can examine the sequence diagram. The relevant portion of the diagram shows that the message `assignManager()` should initiate two messages `setManager()` and `setBranch()`:

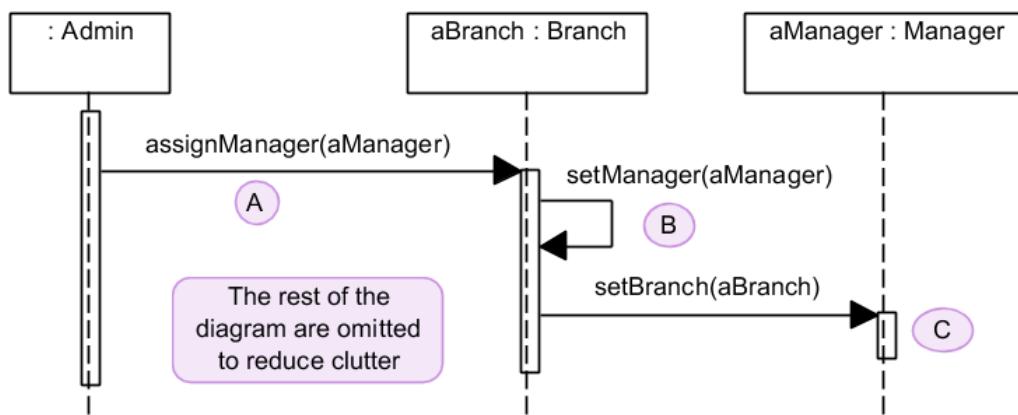


Figure 5.6 Writing the code for the `assignManager()` method

Therefore we could write the method as follows:

#### Source Code: Python

```

def assignManager(self, aManager):
    self._aManagers[aManager.employeeNumber] = aManager # A
    aManager.branch = self # B
# C
    
```

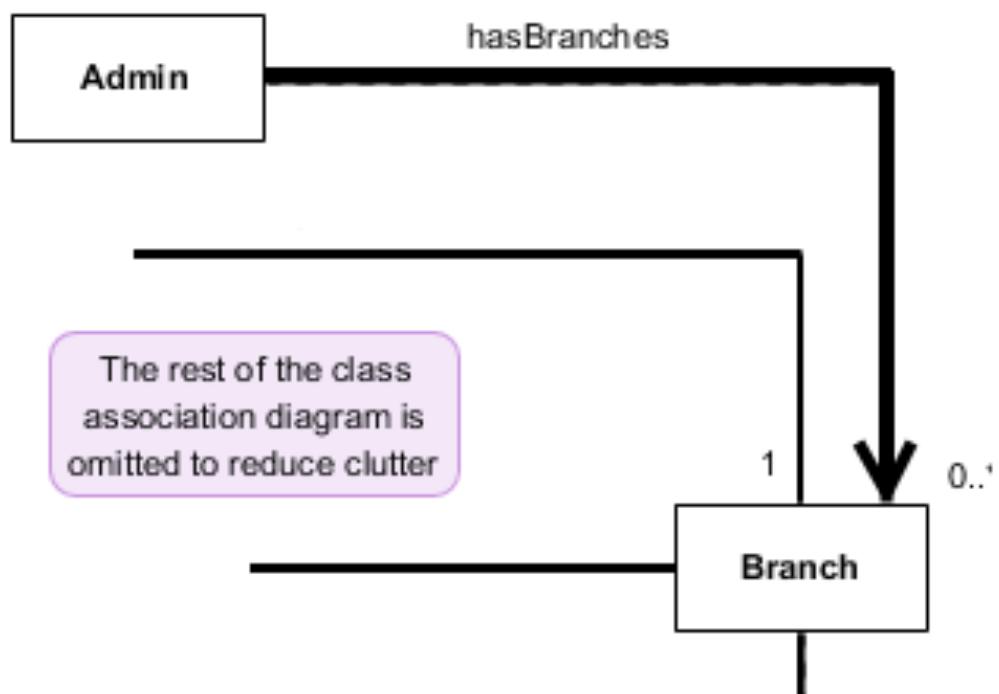
The annotations A, B and C correspond to the messages indicated in the sequence diagram.

This method is written within the `Branch` class since it is the `Branch` object that receives this message. Thus, the pseudo variable `self` in the code refers to the `Branch` object itself.

4. `findBranch(aLocation: String): Branch`

This method is required for the Admin object to locate a Branch object using the location given by the user. Recall that it is a pre-condition in the use case for the branch location to be given in order to assign the new manager to the branch.

Recall also that the class Admin has to keep track of all Branch objects by using a Dictionary. Figure 5.7 and the table below are reproduced from the study unit that discussed the walkthrough.



**Figure 5.7** The association between Admin and Branch

Class being modified	Admin
Association being implemented	hasBranches
Direction of navigation	from Admin to Branch
Multiplicity	one to many

<b>Instance variable added</b>	Branches
<b>Value of the instance variable</b>	A Dictionary of Branch objects, using the location as the key
<b>Justification</b>	<p>An instance variable is added to the class Admin because it has to keep track of branches.</p> <p>A Dictionary is used because</p> <ul style="list-style-type: none"> <li>a. The multiplicity is one to many. The Admin object has to keep track of many branches.</li> <li>b. The application needs to look for a branch by using the branch location. Thus, the branch location is used as the key in the Dictionary.</li> </ul>

This means that we would have the usual methods for managing and manipulating the Branch objects:

<b>int getNumOfBranch()</b>	Returns the number of branches
<b>boolean hasBranch(Branch aBranch)</b>	Verifies whether the branch aBranch is in the list of branches
<b>Branch searchBranches(String aLocation)</b>	Searches the list of branches and returns the branch with the specified location
<b>void listBranches()</b>	Lists all the branches

<code>void addBranch(Branch aBranch)</code>	Adds the branch aBranch
<code>boolean removeBranch(String aLocation)</code>	Removes a branch with the location aLocation from the list of branches

From these methods, we can see that Branch searchBranches(String aLocation) provides precisely the function we need. Therefore, we can adopt this method by amending the sequence diagram and changing the message

- `findBranch(aLocation: String): Branch`
- `to searchBranches(aLocation: String): Branch`
- 5. `assignNewManager(aLocation: String, employeeName: String, employeeNumber: String): void`

Again, the sequence diagram shown in Figure 5.8 helps to show that this method should initiate the messages `searchBranches()`, `assignManager()` and `create the new Manager object`. Therefore, we could write the method as follows:

#### Source Code: Python

```
def assignManager(self, aLocation, employeeName, employeeNumber): # A
    aBranch = self.searchBranches(aLocation) # B
    aManager = Manager(employeeName, employeeNumber) # C
    aBranch.assignManager(aManager) # D
```

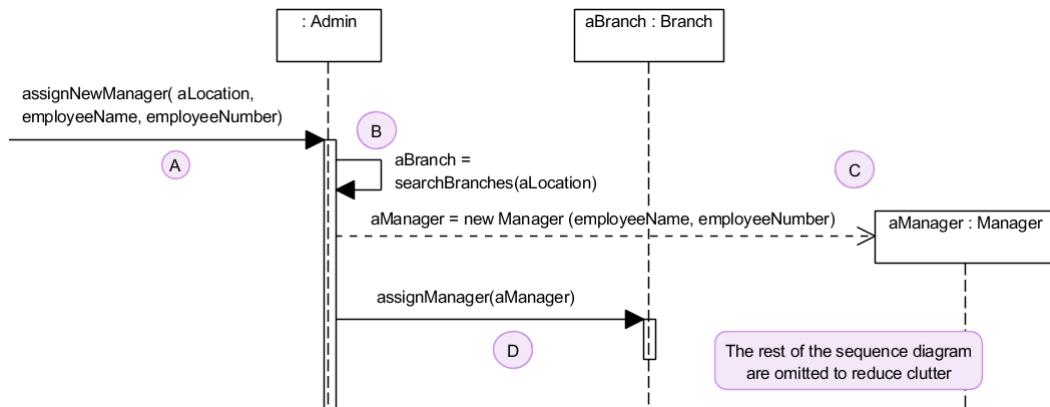


Figure 5.8 The association between Admin and Branch

## 1.4 Message Senders and Receivers

We want to re-look at our previous discussions in terms of basic object oriented programming concepts. We know from these concepts that most statements can be interpreted as follows:

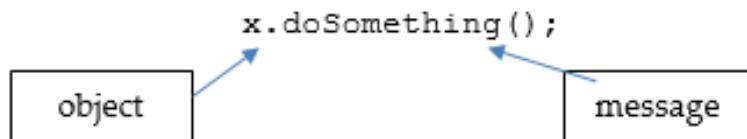


Figure 5.9 An object receiving a message

The sender of this message is the object executing the method `doSomething()`. For example, assume that the class `ClassA` has the method `doSomething()` as follows:

### Source Code: Python

```

class ClassA(object):
    # instance variables and other methods
    def doSomething():
        # statements in method
  
```

### Testing ClassA:

```

objectOfClassA = ClassA()
objectOfClassA.doSomething()
  
```

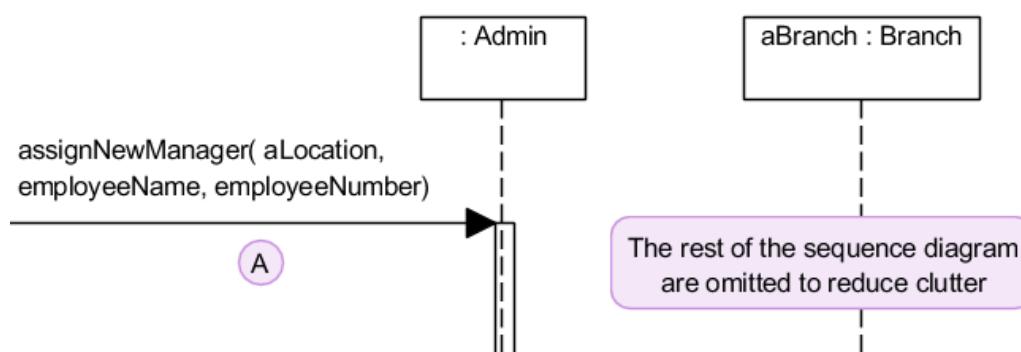
Suppose an object of the class `ClassA` is created as shown above. This object would send the message `doSomething()` to the object to execute `doSomething()`.

We will now re-examine our previous discussion with these concepts and observe how they fit in nicely with the sequence diagrams. In the process, we hope to achieve a better understanding of how the methods are written. We will look at the following three questions:

- Which is the class that you are writing the method for?
  - Which object sends the message?
  - Which object receives the message?

1.     Which is the class that you are writing the method for?

When we are working on the `assignNewManager()` method, the sequence diagram shows that the `Admin` object should receive the message corresponding to this method. Therefore, we are writing the `assignNewManager()` method for the `Admin` class.



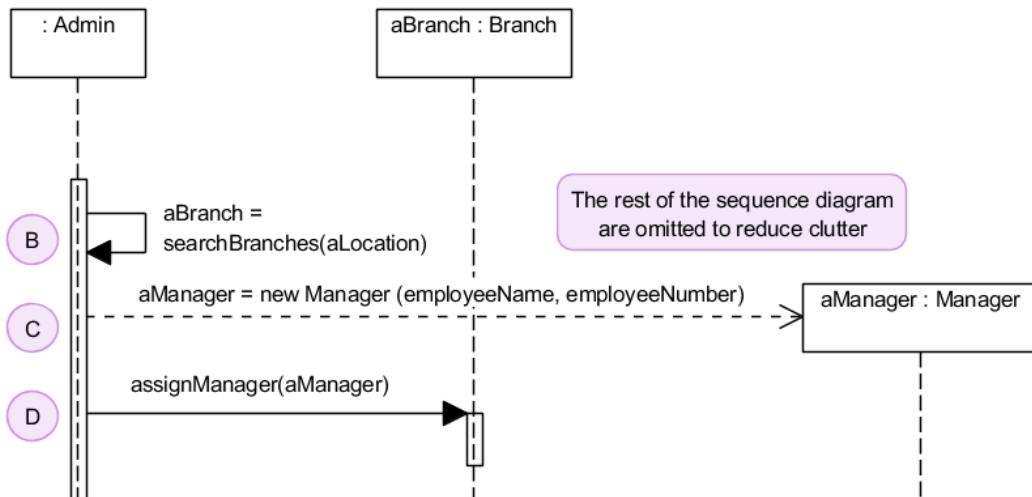
**Figure 5.10** The object that receives the message

**Note:** the message `assignNewManager()` itself could be initiated by an actor through the user interface. Our focus, however, is on the Admin object. So, we shall leave out the question of who sent this message at this point of time.

2. Which object sends the message?

As a result of receiving the message `assignNewManager()`, the `Admin` object sends the messages `searchBranches()` and `assignManager()`. You can see this in the sequence diagram in Figure 5.11. Observe how these messages correspond

to the code inside the method `assignNewManager()`. When the `Admin` object executes this method, it executes these statements and sends the messages.



**Figure 5.11** The `Admin` object sends messages when it executes the `assignNewManager()` method

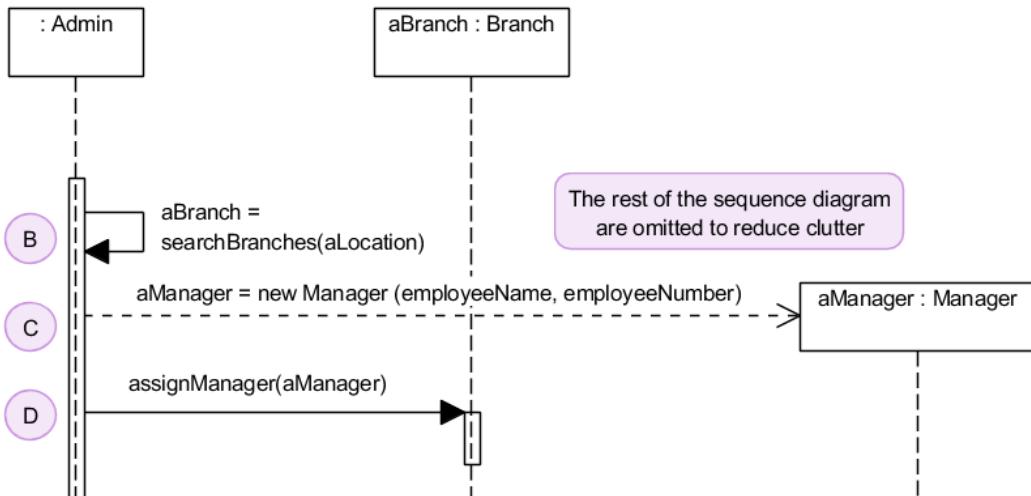
### 3. Which object receives the message?

Now, examine the `assignNewManager()` method again to see which objects are the receivers of the messages:

Line B of the code shows that the message `searchBranches()` is received by the `Admin` object itself. Similarly, line D of the code shows that the message `assignManager()` is received by the object `aBranch`.

Observe how this fits in with what the sequence diagram shows:

**Figure 5.12** The receivers of the messages



## The constructor

From the sequence diagram, you would have noticed that we have used a constructor with only two arguments to create the new Manager object:

```
new Manager(employeeName, employeeNumber)
```

Since we also have an instance variable referencing a Branch object in Manager, we could have used the following instead:

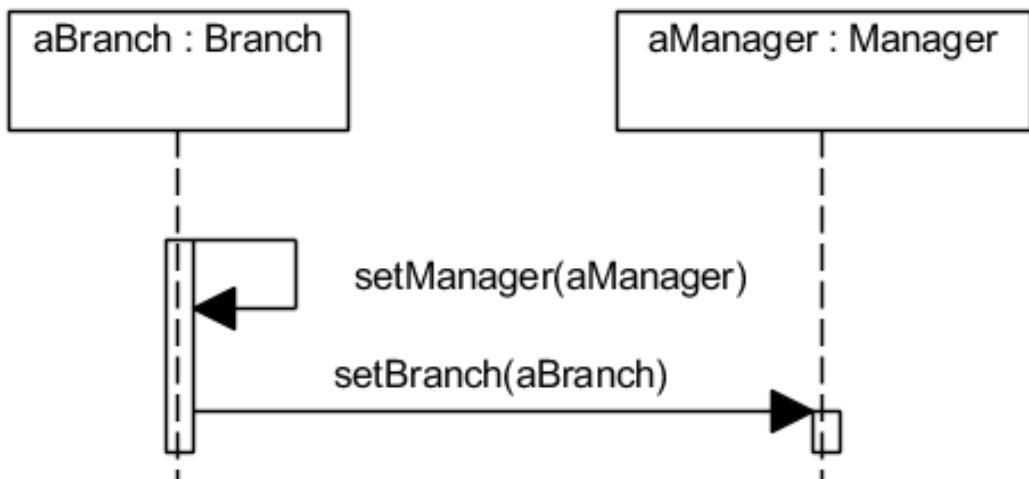
```
new Manager(employeeName, employeeNumber, aBranch)
```

However, we choose not to do so because we wanted to highlight the pattern of messages required for a bi-directional association:

setManager() for the Branch object

setBranch() for the Manager object

The fragment in Figure 5.13 shows the pattern for these messages. [Note that this is not the only way for sending the messages to set up a bi-directional association. However, discussing the alternatives is beyond the scope of this course.]

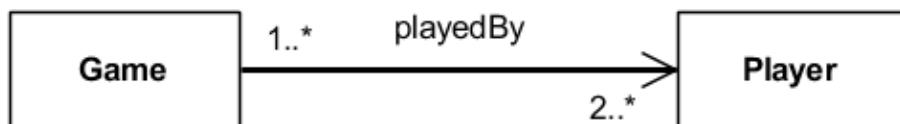


**Figure 5.13** Setting up a bi-directional association

## 1.5 Review Exercises

### Exercise 1

A television game show has many participants called players. The following shows a fragment of the class association diagram with two classes, Game and Player. The **dynamic analysis** has been performed with the class diagram:



#### Source Code: Python

```

class Game:
    def __init__(self):
        self._playersDict = {}
        # other instance variables
  
```

Assume that a player is identified by his name. Write the Python code for the methods corresponding to the collections operations:

<code>int getNumOfPlayers()</code>	Returns the number of players in a game
<code>boolean hasPlayer(Player aPlayer)</code>	Verifies whether the player aPlayer is in the list of players of a game
<code>Player searchPlayers(String name)</code>	Searches the list of players and return the player with the specified name
<code>void listPlayers()</code>	Lists all the players in a game
<code>void addPlayer(Player aPlayer)</code>	Adds a player aPlayer to a game
<code>boolean removePlayer(String aName)</code>	Removes a player with the name aName from the list of players

## Exercise 2

In your code for Exercise 1,

- Identify the statement that actually associates a game with all its players.
- State the software re-use technique used in the implementation of the class.
- Tabulate the methods in the Game class that re-use the Dictionary methods.

## Exercise 3

How would the sequence diagram look for the following code?

### Source Code: Python

```
class ClassA(object):
    # instance variables and other methods
    def doSomething():
        # statements in method
```



### Exercise 4

You are given the following use case, walkthrough and sequence diagram:

**Use case** Show particulars of the manager of a branch.

**Initiator** An administrative staff.

**Objective** To show the particulars of the manager in charge of a branch.

**Pre-conditions** The location of the branch required must be given.

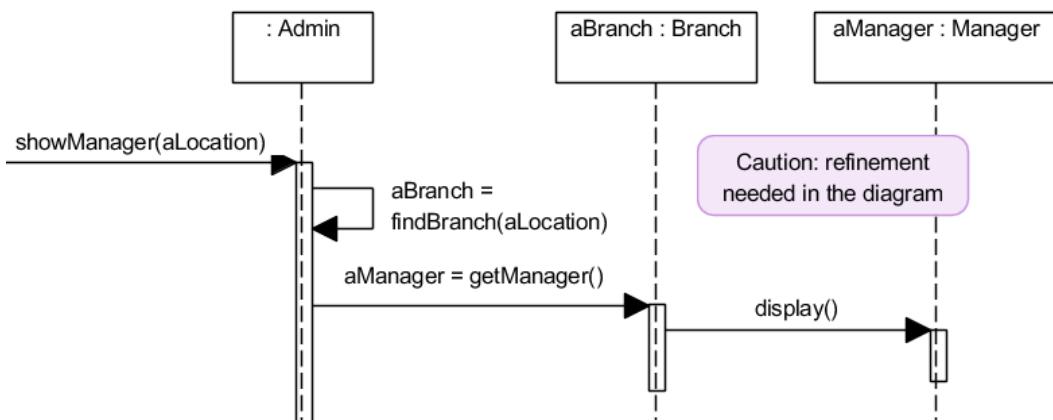
**Post-conditions** The application will display the particulars of the manager of the branch specified.

**Assumptions, Notes or constraints** None

The walkthrough:

1. Locate the Branch object with the specified location, linked to the orchestrating object via the association hasBranches.
2. Locate the Manager object linked to the Branch object via the association isResponsibleFor.
3. Show the particulars of the Manager object.

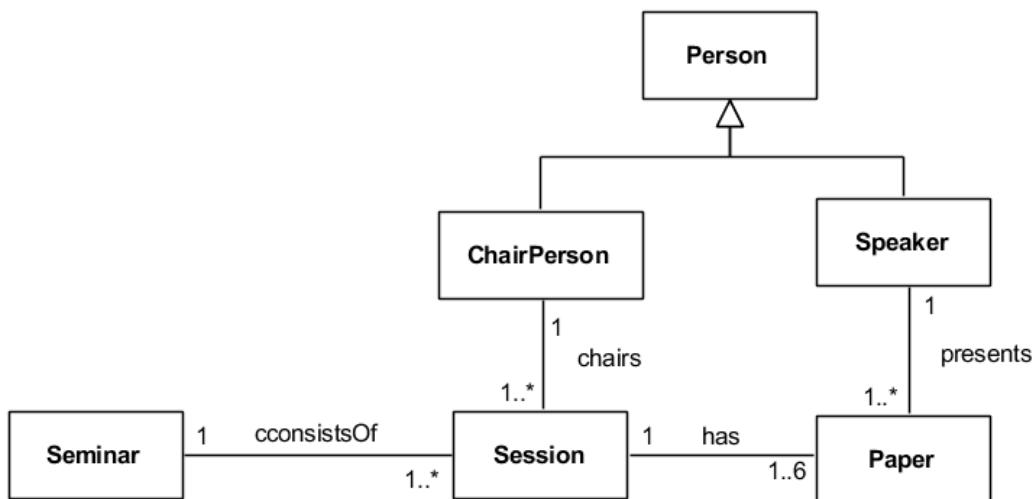
The sequence diagram:



- Revise and improve the sequence diagram shown above.
- Write the programs for the methods required to implement this use case. Use the `print()` statements to display the particulars of the manager.

### Exercise 5

There was a discussion in Study Unit 2, Section 2.1 on an application for managing a seminar. The class association diagram is reproduced below.



For simplicity, we decide to use the application for only one seminar. This means there will be only one Seminar object. We will use this object as the **orchestrating object** for convenience.

For the purpose of this question, we will use the simplified class description below:

<b>Class:</b>	Seminar, this class has only one instance
<b>Class:</b>	Session
<b>Attributes:</b>	sessionNo, the unique session number that identifies a session
<b>Class:</b>	Person, superclass of Speaker and Chairperson
<b>Attributes:</b>	name, the name of the person
<b>Class:</b>	Speaker, subclass of Person
<b>Attributes:</b>	refer to the superclass
<b>Class:</b>	Chairperson, subclass of Person
<b>Attributes:</b>	refer to the superclass
<b>Class:</b>	Paper
<b>Attributes:</b>	title, the title of the paper

The following is one of the use cases that is to be implemented for the application:

<b>Use case</b>	Add a speaker and paper
<b>Initiator</b>	A company staff
<b>Objective</b>	To assign a new speaker and his paper to a session
<b>Pre-conditions</b>	The name of the speaker, the title of the paper and the session number of the session are given

**Post-conditions** The application would have an additional paper for presentation

**Assumptions, Notes or constraints** None

- a. Give the walkthrough required to perform this use case.
- b. Show the class association diagram as a result of this walkthrough.
- c. Draw the sequence diagram for the walkthrough.
- d. Identify and implement the methods for the use case.

Note that you need not add any additional associations to the class association diagram.

Give the walkthrough required to perform this use case.

## 1.6 Model Driven Architecture

Model-driven architecture (MDA) is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system. Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

There are many tools such as enterprise architect, visual-paradigm and pyUML that support MDA. Using these tools, we can model the software system and automatically generate codes based on the model. For example, using Eclipse IDE with PyDev/PyUML, the structural model with all the class specifications can be designed and then Python code can be auto generated from the model. The reverse can be done using pyreverse plug-in to Eclipse. Try it on your own!

## Chapter 2: Integration

### 2.1 Application User Interfaces

In our discussions on the structural and the dynamic modelling, we have focused on the classes within the application itself. We have not talked about the user interfaces.

Any application would have some form of interface through which the external world interacts with it. Figure 5.14 makes use of the figure from the previous units to show a system and its interaction with the external world:

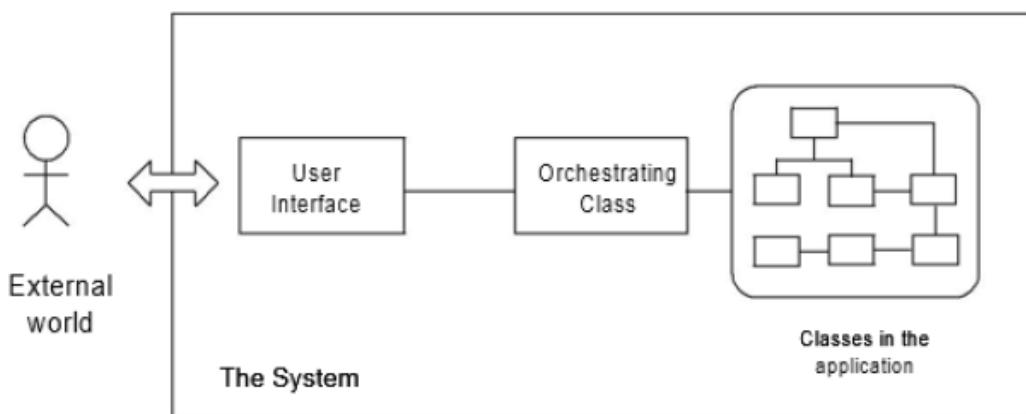


Figure 5.14 The system and the external world

Many large modern systems today have more than one interface through which the external world can access their functions. The following are examples of some systems and their interfaces:

#### System

#### Interfaces

A banking system

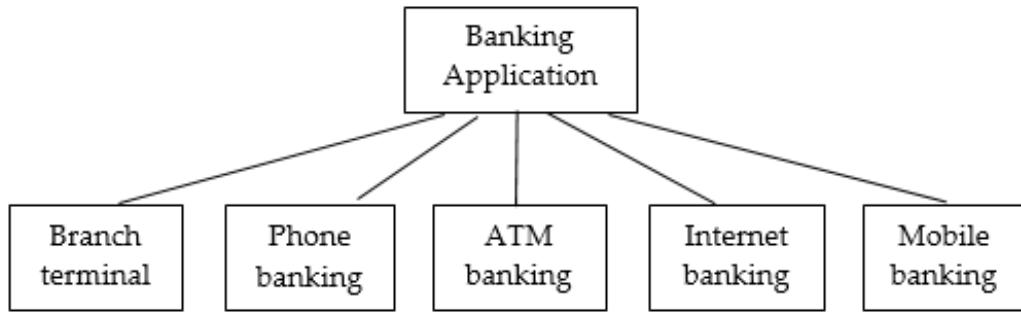
Terminals at the bank branch counter, the ATM, an Internet browser, telephone banking or a mobile phone.

**A bill payment system**

Terminals at the payment counter, payment kiosks, GIRO transfers, telephone payment, a mobile phone or an Internet browser.

**An industrial automation system**

Custom-designed control panel, manually operated remote control or sensor operated controls.



**Figure 5.15** Application with many types of interfaces

These examples show that interfaces come in a wide variety of forms. Some interfaces provide full graphical interactive facilities, incorporating colours, icons and windowing capabilities. Others are limited to abbreviated text or customised physical widgets.

Existing applications often require new interfaces to be incorporated or existing interfaces to be modified. The reasons for this could be:

- **Technology:** technological advances can render existing ways of doing things obsolete
- **Business:** new features or facilities may require existing interfaces to be modified
- **Both technology and business:** the competitive business environment may require new innovative ways to retain or capture market share

**Introducing a new interface may require unnecessary modifications to the application if the system is not carefully planned and designed.**

Therefore, an important concern in designing our systems is to limit as far as possible the changes that might be required if an interface is modified or a new interface is added to a system.

Recall that when we were discussing structural and dynamic modelling, we have deliberately left out the user interface. The intention is to keep the user interface as separate as possible from the application itself.

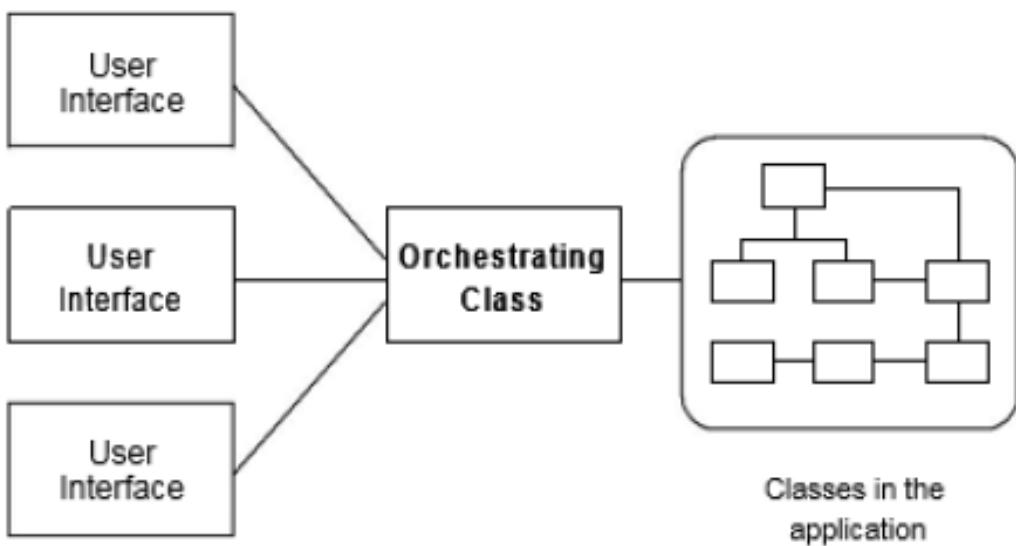
The objective for this separation is to allow:

- Addition of new types of interfaces with little disruptions to the application and its existing interfaces
- Change in an existing interface with little impact on the rest of the system
- Changes in the application itself with minimal impact on all of its interfaces.

To achieve this, the following would be required in the system:

- The various interfaces should be as independent from each other as possible. They should have no knowledge of each other and impose no requirements directly on one another.
- Each interface should only interact with the application and do so in clearly defined ways.
- The interfaces should have no internal knowledge of the application. In addition, the nature of their interactions with the application should be limited to the bare minimum necessary to get the job done.

Recall that we have introduced an orchestrating instance between the user interface and the classes in the application. The following shows an orchestrating class responsible for interacting with more than one interface:

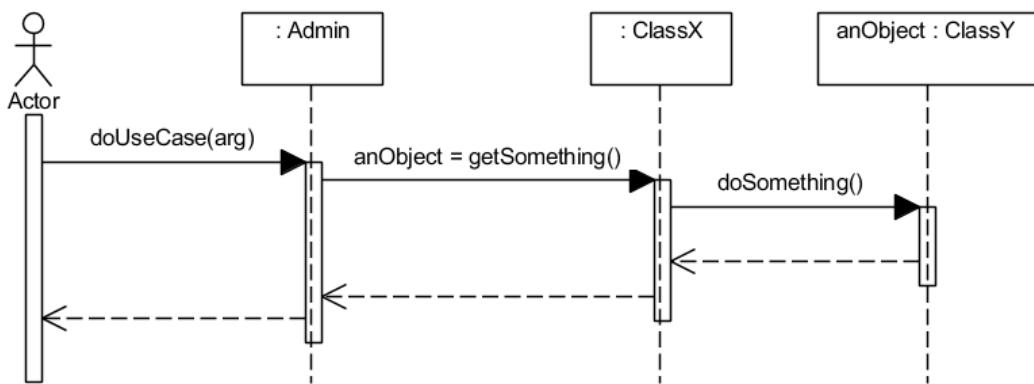


**Figure 5.16** An application with more than one interface

By introducing an orchestrating class with a single instance, we can achieve the following advantages:

- The interfaces need to know only the orchestrating instance. The details of the design and implementation of the classes in the application itself is hidden from the interfaces. The interfaces need not know about these classes and their implementations. When a user interface has to execute any application function, it only needs to send a message to the orchestrating instance.
- The classes in the application need to accept messages or requests for services only from the orchestrating instance. They need not know about the various user interfaces interacting with the system. They need not even be aware which interface is requiring a service from them.

Since the user interfaces do not know about the implementations inside the system, any changes within the system would have limited impact on the interfaces.



**Figure 5.17** Keeping the interfaces and the application separate by hiding implementation details from each other

In the previous discussion, we have introduced one single class as the orchestrating class. This is one way to separate the user interfaces from the classes in the application. Using just one class for this purpose has the disadvantage that the orchestrating class might become overloaded with too many functions when there are many use cases.

Some designers have proposed that it is clearer to have a class corresponding to each actor. Thus, for a library system, there would be a class corresponding to a borrower, another corresponding to a librarian and so on. With this approach, all functions for the use cases relating to a borrower would be grouped in one class BorrowerClass and those relating to the work of a librarian would be grouped in another class LibrarianClass. The user interface would need to know which class has the required function for a particular actor.

Other designers suggest that the system should have a class corresponding to each use case. Thus, a library system would have a BorrowingClass for the use case to borrow items, a ReservingClass for the use case to reserve an item and so on. Each of these classes might have a run() method that would be activated when the function is needed.

For the purpose of this course, we would stick to the design with one central class as the orchestrating class.

In the next section, we will use our analysis for the car rental application to demonstrate integration with a graphical user interface. We will be using the Python tkinter package for demonstration purposes.

We will re-use some of the use cases so that we can now focus on the integration process. Again, we will strive to keep to the basic essentials to avoid being swamped with a multitude of details that have little learning value. This means that we may leave out the complete details typically required for a GUI in a real application. Also, we have covered the design issues pertaining to a good GUI design. We will not be repeating them here unless they are essential for our discussion.

As part of the integration, we will see that incorporating a user interface does impose some requirements on the application even though we try to minimise the impact of doing so.

Therefore, there could be changes required in the use cases, the walkthrough and the sequence diagrams.

## 2.2 Incorporating a User Interface

For our discussion, we will incorporate a user interface with two application functions. The ideas involved can be easily extended to include a full set of functions for the application. We will continue to use our car rental example as our application. **The** functions for which we will incorporate the user interface are:

1. Adding a car to a contract
  2. Removing a car from a contract

### 2.2.1 Example: Adding a Car to a Contract

The following is the use case that we have previously drawn up. An existing corporate customer wishes to rent another car from the company. The customer has specified a car based on certain specifications. To keep things manageable, we have used the engine capacity to represent these specifications.

Use case	Add a car to an existing contract.
Initiator	A marketing staff.

**Objective** To add a car to an existing contract of a given customer.

**Pre-conditions** The customer, identified by name, must have a contract with cars on rental.

The following information are given: the branch from which the car is selected and the minimum engine capacity of the car required.

**Post-conditions** The contract would have one car more under rental.

**Assumptions, Notes or constraints** The additional car comes from the same branch as the other cars in the contract.

As a result of this use case, we have implemented the following method in the class Admin:

#### Source Code: Python

```
def rentAnotherCar(self, name, location, engineCap):
    cust = self.searchCustomer(name)
    contract = cust.getContract()
    br = self.searchBranches(location)
    aCar = br.findCar(engineCap)
    cust.includeCar(contract, aCar)
```

The following are some points to note about this implementation:

1. The method requires as input data the name of the customer, the location of the branch and the engine capacity of the car to be added.
2. The code calls the methods searchCustomer(custName) and searchBranches() of the same class. This method has been implemented as a non-static method. Thus,

you would need to set up an orchestrating object of the class Admin with the customers in the application before you can execute this method.

3. The code also calls the methods getContract() and includeCar() of the Customer object cust. It also calls findCar() of the Branch object br. This method has also been implemented in the previous unit.

As our first attempt in implementing the GUI, the following could be a possibility. Of course, with the actual application, the screen design would not be as plain as this.

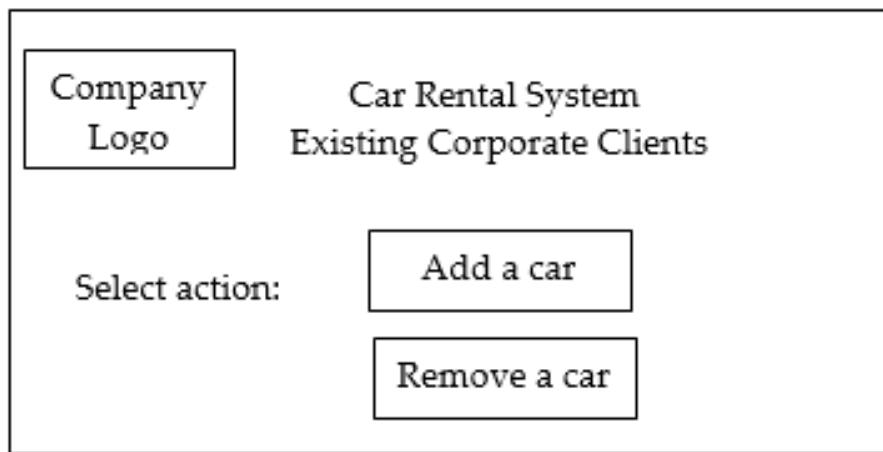


Figure 5.18 A Simple User Interface

In our discussion, we will focus on the first action button "Add a car". One of the possible Python implementation is shown below where the interface code is the main code which is executed first. The program also creates the orchestrating object Admin. This single line is all that is required to link the interface to the application. It does not need to know how the object will execute its method and how the classes within the application will collaborate to provide the function required. The Admin object will set up the application and its data. These include getting all the application data from the databases where necessary. For our purpose, we can simply write a method later for testing of this Admin object.

## Source Code: Python

```
import tkinter as tk
from tkinter import simpledialog as sdg

# Admin class definition to be placed here (omitted to remove clutter)
admin = Admin()

def addCar():
    custName = sdg.askstring("Name prompt", "Enter client name")
    loc = sdg.askstring("Location prompt", "Enter location of the branch")
    engineCap = sdg.askstring("Engine Capacity prompt",
                               "Enter engine capacity required")
    admin.rentAnotherCar(custName, loc, engineCap)

root = tk.Tk()

# width x height + x_offset + y_offset
root.geometry("340x400+30+30")

t1 = tk.Label(root, text="Car Rental System", bg="#00A3DA")
t2 = tk.Label(root, text="Corporate Clients", bg="#00A3DA")

t1.place(x=100, y=30)
t2.place(x=102, y=60)

t3 = tk.Label(root, text="Select Action:", fg="#00A3DA")

t3.place(x = 10, y = 100)
buttonAdd = tk.Button(root, text = "ADD A CAR", fg="#0070C0",
                      command = addCar)
buttonAdd.place(x = 100, y = 100)

buttonRemove = tk.Button(root, text="REMOVE A CAR", fg="#0070C0",
                        command = quit) # empty quit, yet to implement this
buttonRemove.place(x=100, y = 130)
root.mainloop()
```

Figure 5.19 shows the interface when the program is run.

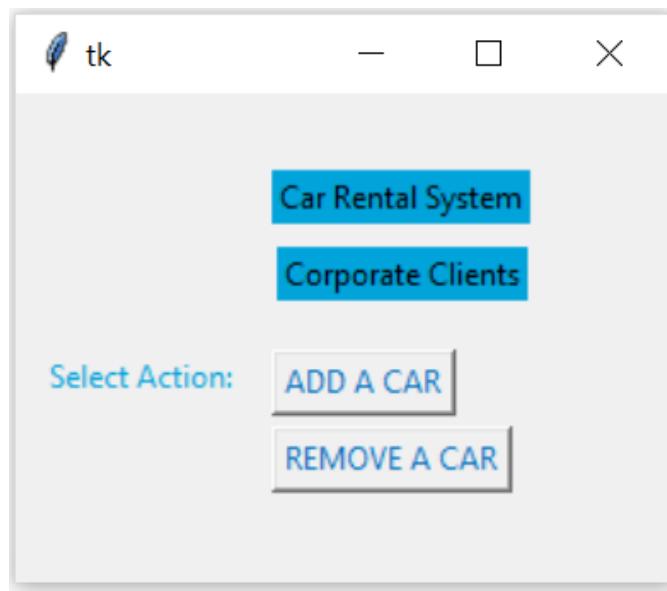


Figure 5.19 Output of Python program



### Activity 5.3

In this question, you are asked to implement the codes to incorporate the user interface for the use case below. We also have discussed this use case in the previous units:

Use case	Remove a car from a contract.
Initiator	A marketing staff.
Objective	To remove a car from the contract of a given customer.
Pre-conditions	The customer, identified by name, must have a contract with cars on rental.
Post-conditions	The contract would have one car less under the customer.
Assumptions, notes or constraints	None

1. What is the method that should be added to the orchestrating class, Admin?
2. What are the input data required by this method?
3. What are the methods that are invoked directly by this method?
4. What are the statements that should be added to the user interface? Base your answer on re-using our simple GUI discussed in the text.
5. What are the methods that need to be added to the classes in the application to execute this function?

## Chapter 3: Testing

### 3.1 Types of Testing

Like all major undertakings, testing is essential to ensure that things work as planned. There are many ways of looking at testing for software development.

One way of looking at testing is to consider whether we examine the code within the program:

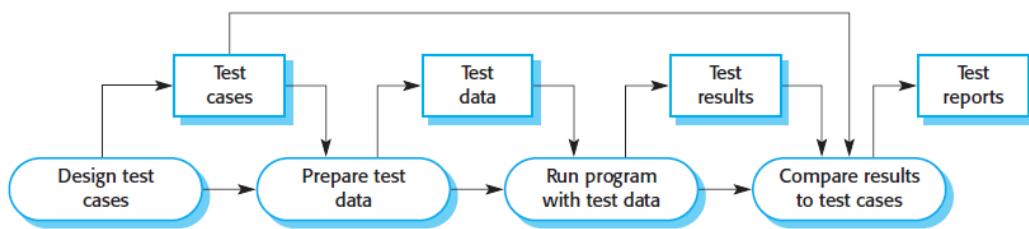
- White box testing
- Black box testing

In white box testing, the tests are conducted based on the code itself. The tester examines the code and creates various test cases to ensure that all the program statements are tested. Thus, the loops, branches, boundaries and complex algorithms are exercised to ensure they work as desired.

In black box testing, the tests are conducted without access to the code inside the program. Thus, the tester would have no knowledge of the internal structure of the program. The tester treats the program as a black box accepting certain input and giving certain output. The test cases are therefore prepared from the functional requirements of the program.

#### 3.1.1 A Model of the Software Testing Process

Figure 5.20 shows a generic model of software testing process.



**Figure 5.20** A model of the software testing process

(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition*))

The testing process has two goals. First is to demonstrate to the developer and the customer that the software meets its requirements. Second is to discover situations in which the behaviour of the software is incorrect, undesirable or does not conform to its specification. These two goals lead to the following type of tests:

- The first goal leads to validation testing
  - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- The second goal leads to defect testing
  - The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

Testing is done in one of the following three stages. We will discuss each of the stages in the following sections:

- **Development testing**, where the system is tested during development to discover bugs and defects.
- **Release testing**, where a separate testing team tests a complete version of the system before it is released to users.
- **User testing**, where users or potential users of a system test the system in their own environment.



## Watch

An excellent introduction to test driven development (TDD):

<https://www.youtube.com/watch?v=dWayn0QsJr8>

## 3.2 Developmental Testing

Development testing includes all testing activities that are carried out by the team developing the system. They can be classified as follows:

- **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods. It is a defect testing process. A unit may be individual functions, methods within an object, or object classes with several attributes and methods.

- **Automated testing:**

Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention. In automated unit testing, you make use of a test automation framework (such as PyUnit for Python) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success or otherwise of the tests.

- **Regression testing:**

Regression testing is testing the system to check that changes have not 'broken' previously working code. In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program. Tests must run 'successfully' before the change is committed.



## Watch

A practical python unit testing using PyUnit unit-test module:

<https://www.youtube.com/watch?v=6tNS--WetLI>

- **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces or integration of units rather than individual units. For example, updating Contract object for renting a car requires accessing methods on the Car object. Messages passed between the objects should be tested for invalid parameters, invalid data types, etc.
- **System testing**, where some or all of the components in a system are integrated and the system is tested as a whole. The components include both off-the-shelf components and in-house developed components. System testing should focus on testing component interactions. System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- **Use case testing:** The use cases developed to identify system interactions can be used as a basis for system testing. Each use case usually involves several system components so testing the use case forces these interactions to occur. The sequence diagrams associated with the use case documents the components and interactions that are being tested.



## Read

Sommerville, I. (2016). *Software Engineering, 10th Edition*. Chapter 8, Sections 8.1.1, 8.1.2, 8.1.3, pp.232-238.



## Activity 5.4

What are the components of automated testing?

Describe the strategies to choose effective test cases.

Complete the code for Contract class (consider all possible alternate scenarios) and set up the components for testing to test using PyUnit framework.



## Read

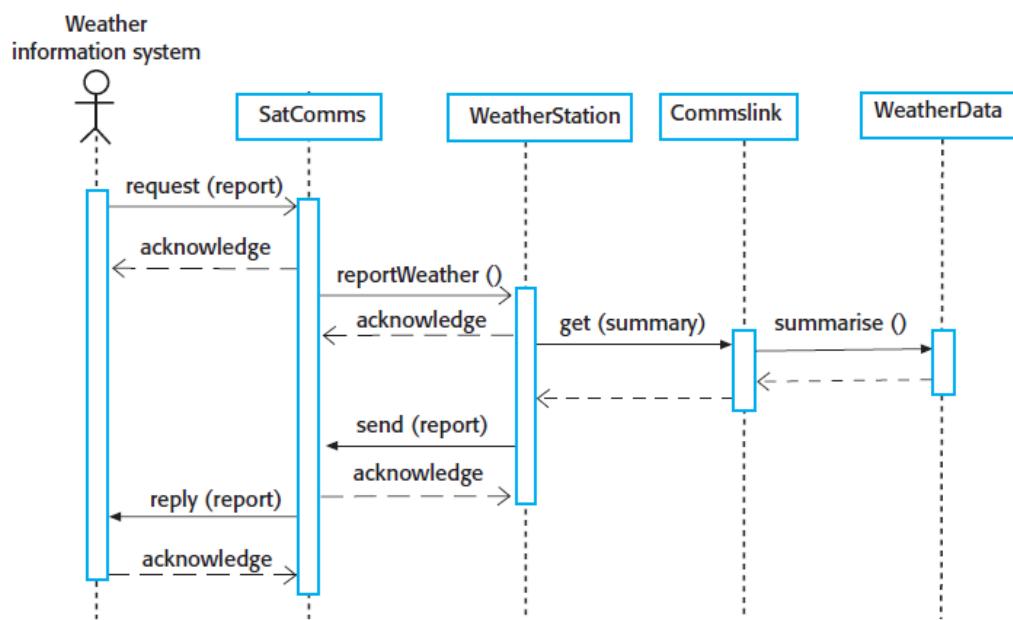
Sommerville, I. (2016). *Software Engineering, 10th Edition*. Chapter 8, Section 8.1.4, pp.240-243.



## Activity 5.5

Write test cases based on any of the use cases or sequence diagrams of the Car Rental System.

For example, the test cases for the weather station system is derived from the following sequence diagram:



(Source: Sommerville, I. (2016). *Software Engineering, 10th Edition*)

Test cases:

- An input of a request for a report should have an associated acknowledgement.  
A report should ultimately be returned from the request.
  - You should create summarised data that can be used to check that the report is correctly organised.
- An input request for a report to the WeatherStation results in a summarised report being generated.
  - Can be tested by creating raw data corresponding to the summary that you have prepared for the test of SatComms and checking that the WeatherStation object correctly produces this summary. This raw data is also used to test the WeatherData object.

### 3.3 Release Testing

Release testing is the process of testing a particular release of a system that is intended for use outside of the development team. The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use. The test has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use. Release testing is usually a black-box testing process where tests are only derived from the system specification. Release testing is a form of system testing. System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).



#### Activity 5.6

Based on the specification of the Menticare system, develop 2 test cases suitable for release testing.

#### 3.3.1 Performance Testing

Part of release testing may involve testing the emergent properties of a system, such as performance and reliability. Performance requirements such as the following are also tested:

- Peak volumes
- Speed of responses
- Security
- System recovery
- Failure of external conditions, such as power or network interruptions

Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable. Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.

### 3.4 User Testing

There are three types of test done by users. These are:

- Alpha testing

Users of the software work with the development team to test the software at the developer's site.

- Beta testing

Beta testing is done if the software is developed for general release to a wide range of users. Well-known examples of these are operating systems, word processing and other widely used software. Beta testing may be done by a collection of voluntary outsiders that represent the typical user. They provide feedback on their experience with using the software in a working environment so that refinements can still be made.

- Acceptance testing

Acceptance tests are done by the owners of the system. They will test the functions of the system to satisfy themselves that their requirements have been met. Acceptance tests are formal tests that the system must pass before an official handover to the owners.

The list below summarises some of the common arrangements for user testing and the people involved. The actual involvements depend on individual projects.

Unit testing

Programmers, system analysts may also be involved.

Integration testing	System analysts and the project manager. Programmers, system administrators, network administrators and database administrators may also be involved.
System testing	The project manager and system analysts. System administrators, network administrators and database administrators may also be involved.
Acceptance testing	The system owners.

## 3.5 Testing with the Object-Oriented Approach

With the object-oriented approach, testing is made easier due to a few important characteristics of the approach:

1. Modular design with low coupling and high cohesion
2. Clearly defined interfaces between objects
3. Re-use of existing software components

### 3.5.1 Testing of Classes

The testing of a class corresponds to the unit testing in traditional approaches. In addition to ensure that each method works, testing a class might include:

1. Testing of its encapsulated state, since the behaviour of an object is dependent on its state.
2. Testing of method invocation, where the result of invoking methods in various sequences has to be checked.
3. Testing of inheritance, where an object may inherit or override the methods of its superclass.
4. Testing of polymorphism, where a variable can reference different types of objects.

### 3.5.2 Integration Testing

This tests to ensure that objects interact correctly to perform the functions required. Use cases and scenarios are good sources for test cases since they reflect the requirements directly.

There can be hundreds of classes involved in the large applications that you find today. The number of possible ways that objects might interact can be huge. However, with the object-oriented approach, the following characteristics might help to lighten the burden:

1. The interfaces between objects are clearly defined. Hidden connections and problematic links among objects are not allowed.
2. Many of the parts to be integrated could be existing components that have been well tested in the past. Their behaviour is well established. What they need as input and what they produce as output are also clearly defined.

Attempts are currently made to develop metrics to measure the complexity of testing object-oriented systems. However, the discussions are beyond the scope of this module as they fall under the subject of software engineering.

## Summary

The following points summarise the contents of this study unit:

- The classes in the structural model can be translated to code directly. In addition, instance variables are added to implement the associations between classes.
- From the sequence diagram, message senders and receivers are identified and added as methods to the corresponding classes.
- For newly created objects, messages to other objects must be sent in accordance to the class association diagram to ensure that the associations are implemented.
- It is important to keep the user interface independent from the application so that a new user interface does not introduce unnecessary modifications to the application.
- After implementation is completed, testing is essential to ensure that the application works as planned. The second goal of testing is to ensure there are no defects.
- The three stages of testing are developmental testing, release testing and user testing.
- Developmental testing is carried out by the development team (programmers, system analysts and project managers) and involves testing of units, components and the system. Use cases can be used in the system testing. Unit tests may be automated in a Test-Driven Development framework.
- Release testing is to ensure that the application is good enough for use and focuses mainly on defect testing or validation testing and may include performance testing. They are performed by programmers, system, network and database administrators
- User Testing includes Acceptance Testing where the owners of the system are satisfied with that the requirements have been met and the system is ready to be handed over to the owners.

- The object-oriented approach to testing ensures that the classes are tested to ensure the methods work (**class tests**) and that the **objects interact with each other well (integration tests)**.

## Formative Assessment

1. Suppose objectA and objectB are objects of the classes ClassA and ClassB respectively. In a sequence diagram, objectB sends a message methodX to objectA. Which of the following statements is correct?
  - a. methodX belongs to objectA.
  - b. methodX belongs to objectB.
  - c. There is a navigation from classes ClassA to ClassB in the class association diagram after a walkthrough.
  - d. ClassA has a generalisation relationship with ClassB.
  
2. Suppose objectA and objectB are objects of the classes ClassA and ClassB respectively. In a sequence diagram, objectB sends a message methodX to objectA. Which of the following is the correct code to depict this?
  - a. objectB.methodX()
  - b. objectA.methodX()
  - c. methodX()
  - d. objectA = objectB.methodX()
  
3. Consider the class association diagram in Figure 5.1 where many employees work in a branch. The following needs to be done to add a new employee to a branch except \_\_\_\_\_.
  - a. Create a new employee
  - b. Create a new branch
  - c. Set the branch to the Employee object
  - d. Set the employee to the Branch object
  
4. Which one of the following is NOT a reason for the interface to be separate from the application?

- a. The interface may change due to technology.
  - b. The interface may change due to the business.
  - c. The interface may change due to the users.
  - d. The interface may change due to the application itself.
5. Which of the following statements describes testing?
- a. To find broken code.
  - b. To evaluate product to find errors.
  - c. To ensure the system development is complete.
  - d. To ensure the user will accept the system.
6. Which of the following best describes white box testing?
- a. User testing
  - b. Structural testing
  - c. Error guessing testing
  - d. Automatic testing
7. The following are testing done by the user, except \_\_\_\_\_.
- a. Alpha testing
  - b. Beta testing
  - c. Acceptance testing
  - d. Release testing

# Solutions or Suggested Answers

## Activity 5.1

The Python codes for the classes are shown below:

```
class Employee:
    def __init__(self, nName, nEmpNo):
        self._name = nName
        self._empNo = nEmpNo

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, nName):
        self._name = nName

    @property
    def empNo(self):
        return self._empNo

    @empNo.setter
    def empNo(self, nEmpNo):
        self._empNo = nEmpNo

class BranchWorder(Employee):
    def __init__(self, nName, nEmpNo, nHourRate):
        super().__init__(nName, nEmpNo)
        self._hourRate = nHourRate

    @property
    def hourRate(self):
        return self._hourRate

    @hourRate.setter
    def hourRate(self, nHourRate):
        self._hourRate = nHourRate

class Manager(Employee):
    def __init__(self, nName, nEmpNo):
        super().__init__(nName, nEmpNo)
```

```
class Customer:  
    def __init__(self, nName, nAddress):  
        self._name = nName  
        self._addr = nAddress  
  
    @property  
    def name(self):  
        return self._name  
  
    @name.setter  
    def name(self, nName):  
        self._name = nName  
  
    @property  
    def address(self):  
        return self._addr  
  
    @address.setter  
    def address(self, nAddress):  
        self._addr = nAddress  
  
class Contract:  
    def __init__(self, nStartDate, nDuration):  
        self._startDate = nStartDate  
        self._duration = nDuration  
  
    @property  
    def startDate(self):  
        return self._startDate  
  
    @startDate.setter  
    def startDate(self, nStartDate):  
        self._startDate = nStartDate  
  
    @property  
    def duration(self):  
        return self._duration  
  
    @duration.setter  
    def duration(self, nDuration):  
        self._duration = nDuration
```

## Activity 5.2

The following is the complete table:

Class being modified	Manager	Branch
Association being implemented	isResponsibleFor	isResponsibleFor
Direction of navigation	from Manager to Branch	from Branch to Manager
Multiplicity	one to one	one to one
Instance variable added	branch	manager
Value of the instance variable	a Branch object	a Manager object
Justification	Since there is only one branch for a manager, an instance variable with a single value is enough. This variable would have a Branch object as its value.	Since there is only one manager for a branch, an instance variable with a single value is enough. This variable would have a Manager object as its value.

The code for the modifications to the classes are:

1. Manager class – add instance variable branch and its associated accessor and mutator methods.
2. Branch class – add instance variable manager and its associated accessor and mutator methods.

```
class Manager(Employee):
    def __init__(self, nName, nEmpNo, aBranch):
        super().__init__(nName, nEmpNo)
        self._branch = aBranch

    @property
    def branch(self):
        return self._branch

    @branch.setter
    def branch(self, aBranch):
        self._branch = aBranch

class Branch:
    def __init__(self, newLocation, newStatus, aManager):
        self._location = newLocation
        self._status = newStatus
        self._manager = aManager

    @property
    def manager(self):
        return self._manager

    @manager.setter
    def manager(self, aManager):
        self._manager = aManager

    @property
    def location(self):
        return self._location

    @location.setter
    def location(self, newLocation):
        self._location = newLocation

    @property
    def status(self):
        return self._status

    @status.setter
    def status(self, newStatus):
        self._status = newStatus
```

## Exercise 1

The Python codes for the methods are shown below:

```

class Game:
    def __init__(self):
        self._playersDict = {}

    def getNumberOfPlayers(self):
        return len(self._playersDict)

    def addPlayer(self, aPlayer):
        self._playersDict[aPlayer.name] = aPlayer

    def hasPlayer(self, aPlayer):
        return self._playerDict.has_key(aPlayer.name)

    def searchPlayers(self, name):
        return self._playersDict.has_key(name)

    def listPlayers(self):
        for aPlayer in self._playersDict.values():
            print(aPlayer.name)

    def removePlayer(self, name):
        if self._playerDict.pop(name):
            return True
        return False

```

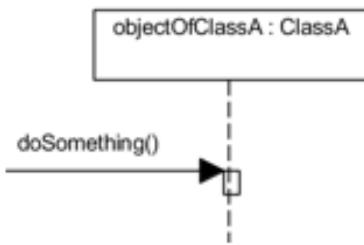
## Exercise 2

- a. self.\_playersDict = {}
- b. re-use by composition
- c. Method in Contract class

Dictionary method used

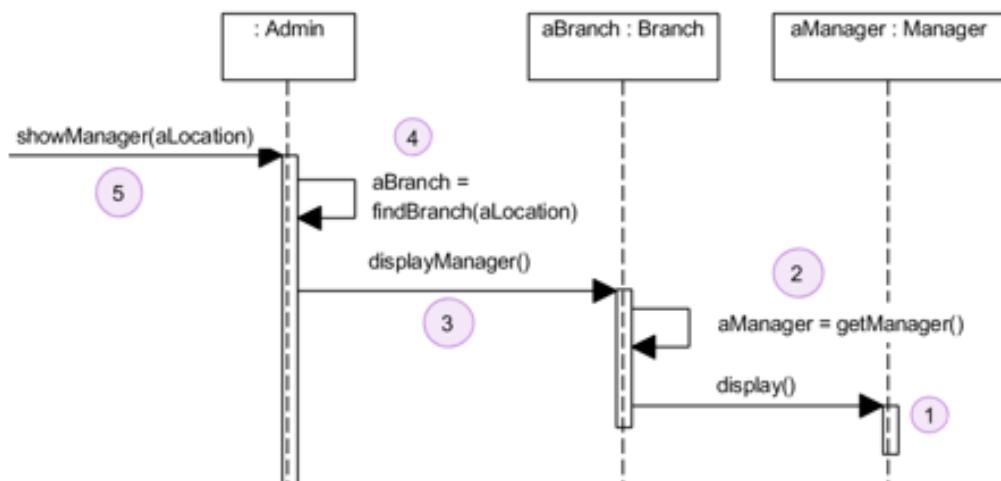
getNumberOfPlayers()	size() or len()
addPlayer()	put()
hasPlayer	containsValue() or has_key()
removePlayer()	remove() or pop()

## Exercise 3



### Exercise 4

- a. The sequence diagram should be amended as follows:



- b. This analysis with the sequence diagram shows that the following methods will be needed:

Manager	display(): void	1
Branch	getManager(): Manager	2
Branch	displayManager(): void	3
Admin	findBranch(aLocation: String): Branch	4
Admin	showManager(aLocation: String): void	5

The implementations for the methods are as follows:

- a. The display() method in the class Manager:

```
def display(self):  
  
    print("Manager name: ", self._name)  
  
    print("Employee number: ", self._employeeNumber)  
  
    print("Branch: ", self._branch.getLocation())
```

- b. The getManager() method in the class Branch: Use the accessor method for the instance variable manager.
- c. The displayManager() method in the class Branch:

```
def displayManager(self):  
  
    aManager = self._manager  
  
    aManager.display()
```

- d. The findBranch() method in the class Admin: Use the collection method previously written for the Dictionary of Branch objects:

```
def Branch searchBranches(self, aLocation)
```

- e. The showManager() method in the class Admin:

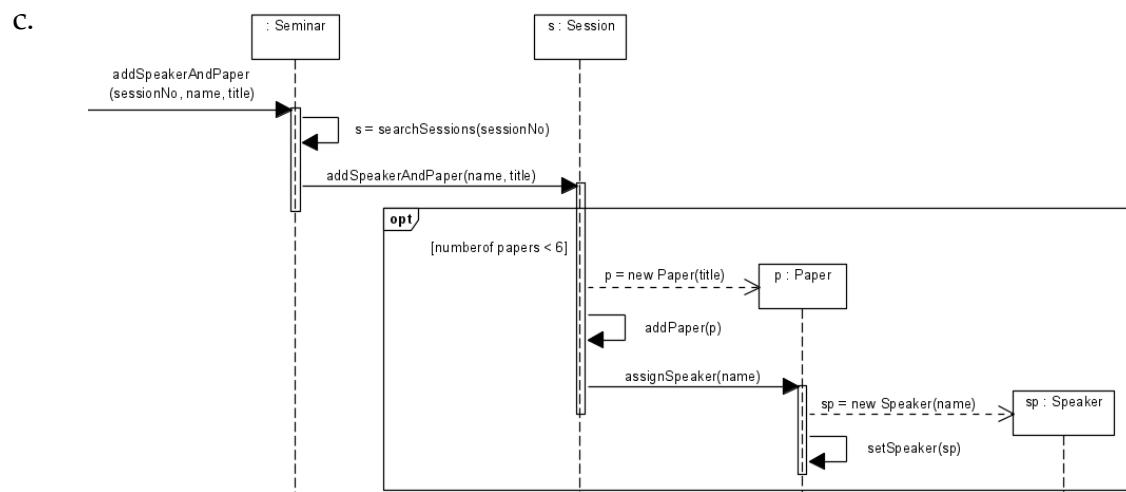
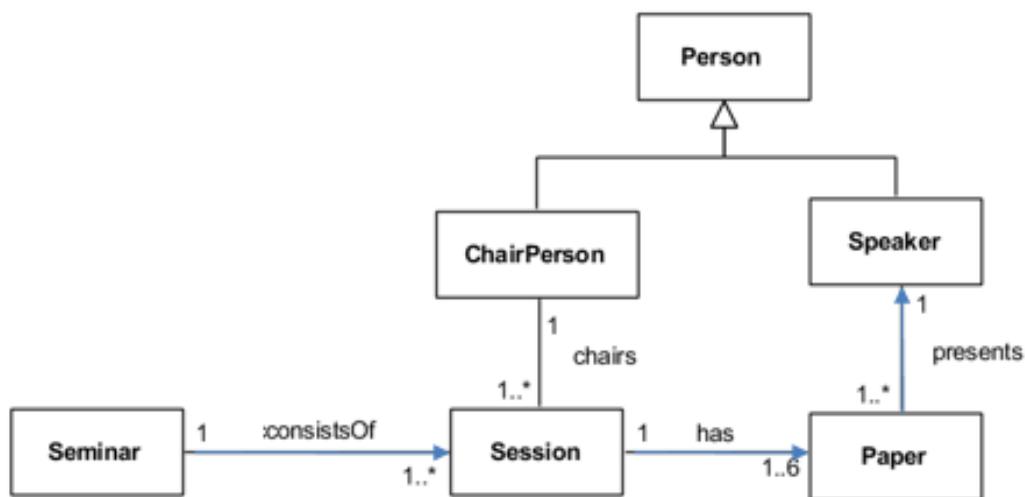
```
def showManager(self, aLocation):  
  
    aBranch = self.searchBranches(aLocation)  
  
    aBranch.displayManager()
```

## Exercise 5

- a. The walkthrough is as follows:

Objective: To add a speaker and her paper to a seminar session

- a. Locate the Session object, using the given session number, linked to the orchestrating instance via the association consistsOf.
- b. Create the new Paper object and its association with the Session object found in step 1.
- c. Create the new Speaker object and its association with the Paper object created in step 2.
- b. The class association diagram is as follows:



- d. Paper      **setSpeaker(sp: Speaker): void**

1

---

Paper	assignSpeaker(name: String): void	2
Session	addPaper(p: Paper): void	3
Session	addSpeakerAndPaper(name: String, title: String): boolean	4
Seminar	searchSessions(sessionNo: integer): Session	5
Seminar	addSpeakerAndPaper(sessionNo: integer, name: String, title: String): boolean	6

- a. The setSpeaker() method in the class Paper

This is the mutator method for the instance variable speaker.

- b. The assignSpeaker() method in the class Paper:

The sequence diagram shows that this method should create the Speaker object and invoke the method setSpeaker().

```
def assignSpeaker(self, name):
    sp = Speaker(name)
    self._speaker = sp
```

- c. The addPaper() method in the class Session:

Use the collection method for the instance variable papers instead of writing a new method:

```
def addPaper(self, p):
```

- d. The addSpeakerAndPaper() method in the class Session:

The sequence diagram shows that this method should create the Paper object, and invoke the methods addPaper() in the class Session and assignSpeaker() in the class Paper.

```
def addSpeakerAndPaper(self, name, title)
    if self._numberOfPapers < 6 :
```

```

    p = Paper(title)

    self.addPaper(p)

    p.assignSpeaker(name)

    return True

else:

```

- e. The searchSessions() method in the class Seminar:

Use the collection method for the instance variable sessions:

```
def searchSessions(self, sessionNo):
```

- f. The addSpeakerAndPaper() method in the class Seminar:

The sequence diagram shows that this method should invoke the method searchSessions() in the Seminar class and the method addSpeakerAndPaper() in the Session class.

```
def addSpeakerAndPaper(self, sessionNo, name, title):
```

```
s = self.searchSessions(sessionNo)
```

```
return s.addSpeakerAndPaper(name, title)
```

### Activity 5.3

Only one class needs to be modified:

Class being modified	Customer
Association being implemented	signs

Direction of navigation	from Customer to Contract
Multiplicity	one to one
Instance variable added	contract
Value of the instance variable	a Contract object
Justification	Since there is only at most one contract for a customer, an instance variable with a single value is enough.

The modified code shows that Customer class has additional instance variable contract and its associated accessor and mutator methods

```
class Customer:  
    def __init__(self, nName, nAddress, aContract):  
        self._name = nName  
        self._addr = nAddress  
        self._contract = aContract  
  
    @property  
    def contract(self):  
        return self._contract  
  
    @contract.setter  
    def contract(self, aContract):  
        self._contract = aContract  
  
    @property  
    def name(self):  
        return self._name  
  
    @name.setter  
    def name(self, nName):  
        self._name = nName  
  
    @property  
    def address(self):  
        return self._addr  
  
    @address.setter  
    def address(self, nAddress):  
        self._addr = nAddress
```

## Activity 5.4

The automated test has 3 parts – setup, call and assertion.

Two strategies that are effective in choosing test cases are: partition testing and guideline-based testing.

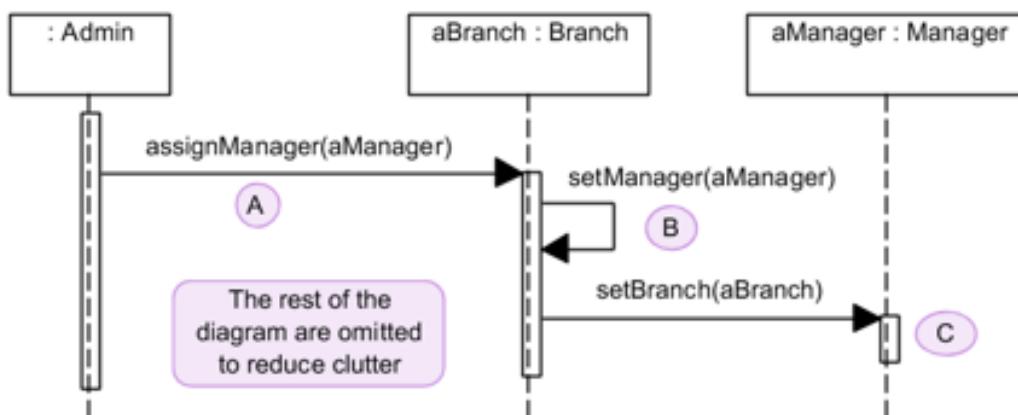
For detailed explanations, read *Sommerville, I., Software Engineering*, pp.233-238.

For the Contract class, testing includes searching for a car that exists or does not exist; add a Car that is already/not already in the Contract; remove a Car that is associated/not associated with the Contract, etc.

PyUnit provides a base class which is a test case to create new test cases. It has methods to set each test routine, methods to control test execution and methods to help determine the outcome of the test. For more information, refer to: <https://docs.python.org/3/library/unittest.html>

## Activity 5.5

Suppose we wish to write test cases for the sequence diagram Figure 5.6, reproduced below:



Test cases:

An input to assign manager should result in an acknowledgement that the manager has been assigned. A message should be displayed to show that the particular manager has been assigned to the said branch.

## Activity 5.6

Consider the following requirements of Menticare system.

- If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.

- If a prescriber chooses to ignore an allergy warning, he shall provide a reason why this has been ignored.

Some of the test cases are:

- Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
- Set up a patient record with a known allergy. Prescribe the medication that the patient is allergic to, and check that the warning is issued by the system.
- Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
- Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
- Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

## Formative Assessment

1. Suppose objectA and objectB are objects of the classes ClassA and ClassB respectively. In a sequence diagram, objectB sends a message methodX to objectA. Which of the following statements is correct?

- a. methodX belongs to objectA.

**Correct. objectB requests the service of methodX from classA. Refer to Study Unit 5, Section 1.4.**

- b. methodX belongs to objectB.

Incorrect. If methodX belongs to objectB, it would send a message to itself. Refer to Study Unit 5, Section 1.4.

- c. There is a navigation from classes ClassA to ClassB in the class association diagram after a walkthrough.
- Incorrect. Since objectB can send a message to objectA, there must be a navigation arrow from objectB to objectA after a walkthrough has been performed. Refer to Study Unit 5, Section 1.2.1.
- d. ClassA has a generalisation relationship with ClassB.
- Incorrect. Generalisation relationships do not dictate message passing. Refer to Study Unit 5, Section 1.4.
2. Suppose objectA and objectB are objects of the classes ClassA and ClassB respectively. In a sequence diagram, objectB sends a message methodX to objectA. Which of the following is the correct code to depict this?
- objectB.methodX()
- Incorrect. If objectB can execute methodX(), it need not send a message to objectA. Refer to Study Unit 5, Section 1.4.
- objectA.methodX()**
- Correct. objectB requests for service from objectA to execute methodX.**
- Refer to Study Unit 5,Section 1.4.**
- methodX()
- Incorrect. methodX() belongs to a class and hence needs an object to invoke it. Refer to Study Unit 5, Section 1.4.
- objectA = objectB.methodX()
- Incorrect. methodX does not belong to objectB. If objectB can execute methodX(), it need not send a message to objectA. Refer to Study Unit 5, Section 1.4.

3. Consider the class association diagram in Figure 5.1 where many employees work in a branch. The following needs to be done to add a new employee to a branch except \_\_\_\_\_.

- a. Create a new employee

Incorrect. A new employee needs to be created as it does not already exist in the system. Refer to Study Unit 5, Section 1.3.

- b. Create a new branch

**Correct. The branch already exists. There is no need to create one. Refer to Study Unit 5, Section 1.3.**

- c. Set the branch to the Employee object

Incorrect. The Employee object needs to know which branch he is working in. Refer to Study Unit 5, Section 1.3.

- d. Set the employee to the Branch object

Incorrect. The Branch object needs to know it has this employee. Refer to Study Unit 5, Section 1.3.

4. Which one of the following is NOT a reason for the interface to be separate from the application?

- a. The interface may change due to technology.

Incorrect. Technology advances may render existing ways of doing things obsolete. Refer to Study Unit 5, Section 2.1.

- b. The interface may change due to the business.

Incorrect. New features or facilities may require existing interfaces to be modified. Refer to Study Unit 5, Section 2.1.

- c. The interface may change due to the users.

**Correct. End-users do not usually influence the interface of an application. Refer to Study Unit 5, Section 2.1.**

- d. The interface may change due to the application itself.

Incorrect. This is the reason why the interface should be separate from the application. Refer to Study Unit 5, Section 2.1.

5. Which of the following statements describes testing?

- a. To find broken code.

Incorrect. This is done during developmental testing but is only one stage of testing. Refer to Study Unit 5, Section 3.1.

- b. To evaluate product to find errors.

**Correct. Software testing is the process of evaluation of a software product to detect differences between given input and expected output. Refer to Study Unit 5, Section 3.1.**

- c. To ensure the system development is complete.

Incorrect. Testing is an important phase in software development but it is not done just to make the system development complete. Refer to Study Unit 5, Section 3.1.

- d. To ensure the user will accept the system.

Incorrect. User testing is one stage of the testing process. Refer to Study Unit 5, Section 3.1.

6. Which of the following best describes white box testing?

- a. User testing

Incorrect. The users do not have knowledge about the internals of the system and hence is not a form of white box testing. Refer to Study Unit 5, Section 3.1.

- b. Structural testing

**Correct. A type of testing based on knowledge of the structure of the program and its components. Refer to Study Unit 5, Section 3.1.**

- c. Error guessing testing

Incorrect. For this technique, the developer uses his/her experience to guess the problems of the system.

- d. Automatic testing

Incorrect. This can also be black box testing as the testing program accepts certain input and checks certain output is obtained. Refer to Study Unit 5, Sections 3.1 and 3.2.

7. The following are testing done by the user, except \_\_\_\_\_.

- a. Alpha testing

Incorrect. Alpha testing is done by a selected group of users to test early releases of a software. Refer to Study Unit 5, Section 3.4.

- b. Beta testing

Incorrect. Beta testing is testing of a release of software done by a larger group of users. Refer to Study Unit 5, Section 3.4.

- c. Acceptance testing

Incorrect. Acceptance testing is done by customers to decide whether to accept the system to be deployed in the customer environment. Refer to Study Unit 5, Section 3.4.

- d. Release testing

**Correct. Release testing is done by the developers to ensure system meets its requirements. Refer to Study Unit 5, Section 3.4.**

## References

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.

# Study Unit 6

Design Patterns

## Learning Outcomes

By the end of this unit, you should be able to:

1. explain the importance of design patterns
2. describe the benefits of design patterns
3. identify problems that can be solved using design patterns
4. describe the principles applied in design patterns
5. describe the key participants in a design pattern
6. draw class association diagrams of design patterns
7. write codes to implement design patterns

## Overview

Developers are expected to write code that not only works, but also is flexible, maintainable and extensible. Applications are likely to undergo multiple changes throughout the development process and even after the application is released, changes for improved features do come in. It becomes a burden if the code is not structured well from the beginning. Object oriented concepts such as encapsulation, inheritance and polymorphism need to be used, but increasingly, design patterns need to be used too.

In this unit, we discuss different types of design patterns, the problems they solve and the benefits of using them. The principles behind several design patterns are also discussed, how the design looks like in a class diagram and how they are implemented.

## Chapter 1: Introduction to Design Patterns

### 1.1 Introduction to Design Patterns



### Lesson Recording

Design Patterns – Strategy Pattern

A design pattern is a way of reusing abstract knowledge about a problem and its solution. Design patterns are a common way of solving well-known problems that crop up in all kinds of applications. A pattern is a description of the problem and the essence of its solution. It should be sufficiently abstract to be reused in different settings. Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism. All design patterns are language neutral.



### Watch

An excellent introduction to Design Patterns:

<https://www.youtube.com/watch?v=0KhDDYwngyQ>

Design patterns were first described in the book 'Design Patterns – Elements of Reusable Object-Oriented Software' written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, also known as the Gang of Four. The book catalogues 23 design patterns as shown below and since then, many more design patterns have been invented.

<u>Creational Patterns</u>	<u>Structural Patterns</u>	<u>Behavioural Patterns</u>
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template
		Visitor

The advantages of using design patterns are:

- Not reinventing the wheel
  - By leveraging the hard work of other developers who had gone through a similar exercise and found good solutions that you can use.
- Building resilient code
  - Design patterns protect the software from changes and additions.

Design patterns are not algorithms and they are not code. It is an approach to thinking about software design that incorporates the experience of developers who had similar problems and fundamental design principles that guide how to structure software designs. The common elements of a pattern are:

- **Name.**

- A meaningful and memorable name that captures the gist of the pattern.

- **Problem description.**

- A design challenge the pattern is trying to address.

- **Solution description.**

- Not a concrete design but a template for a design solution that can be instantiated in different ways. Patterns are specified in terms of their structures, i.e. relationships among elements and behaviour interactions.

- **Consequences.**

- The results and trade-offs of applying the pattern. For example, the design pattern could accomplish better security but have consequence of worse performance.

In the following chapters, we will discuss design patterns conceptually under each category, describe its object oriented design and show the key features of a pattern implemented in code. We will also discuss the design principles behind each design pattern. These principle(s) form the basis for the design pattern and specify the particular quality in the design.

## Chapter 2: Behavioural Design Patterns

Behavioural design patterns are specifically concerned with the communication between objects as a program is running. We will discuss 3 behavioural patterns in this chapter – the Strategy Pattern, State Pattern and the Observer Pattern.

### 2.1 Strategy Pattern

The strategy pattern is used in situations where different algorithms are used interchangeably. The strategy is to let the algorithm vary independently from objects that use it. Common elements of the strategy pattern are shown below.

**Pattern name:** Strategy

**Problem Description:** There are many strategies (or algorithms) that can be used. We do not want to lock in to a strategy but we wish to change strategy dynamically as the need arises.

**Solution Description:** This pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. The strategy is to let the algorithm vary independently from objects that use it. This is achieved by the use of has-a relationships between objects and their behaviours which allows for more flexibility than is-a relationships between objects. Extensibility is also achieved as new behaviour is delegated to a composed object.

**Consequences:** Because each algorithm needs to be written in a new class, the number of classes in the application is increased and leads to maintenance issues.

#### 2.1.1 Strategy Pattern Defined

Sometimes there are different ways (or strategies) to do a task. For example, there are different ways to calculate cost depending on the total amount bought (5% discount given if minimum \$80 bought, 10% discount given if minimum \$150 bought, 12% discount given

if minimum \$200 bought, etc.); different ways that an object in a game can move (fly, run, walk, etc.). Which strategy is used depends on the situation. By keeping the strategies in separate classes and not included in the context class itself, the implementation strategies can be kept secret, i.e. not known to the context class.



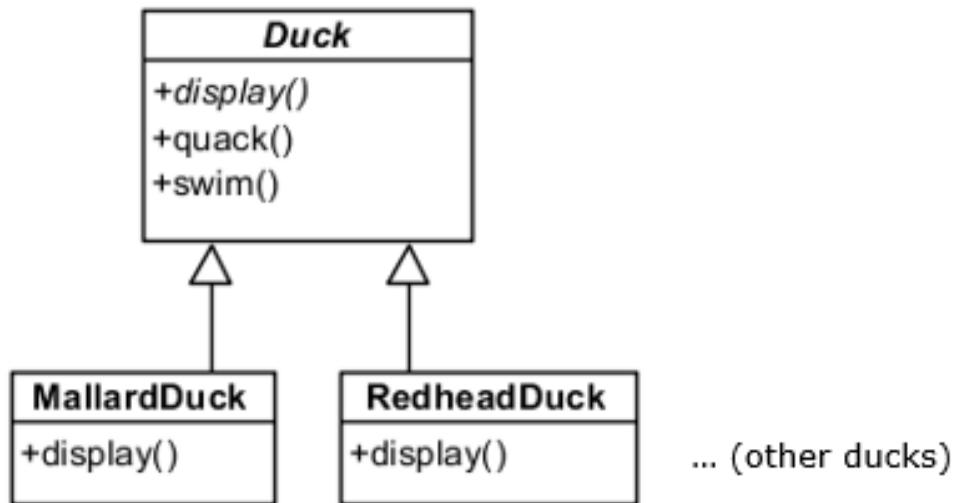
### Read

Freeman, E., & Robson, E. (2020). *Head First Design Patterns*, 2nd Edition, O'Reilly Media, Chapter 1, pp.1-35.

To help you understand the strategy pattern, we will use the duck example from the book, *Head First Design Patterns* by Freeman and Robson. The duck pond simulation game requires a variety of ducks to be in the game. The context is the duck and these ducks have the basic behaviour of swimming and making quacking sounds.

#### 2.1.1.1 Design using Inheritance

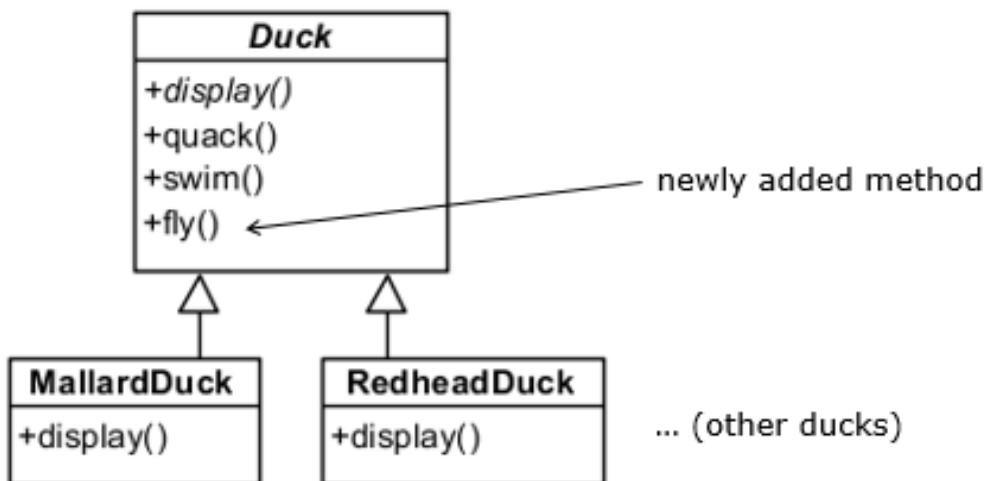
Our initial design is to have a superclass Duck and several duck subclasses to model the mallard duck, red head duck, etc. as shown in Figure 6.1. These duck subclasses would inherit the methods in the superclass: display(), quack() and swim() because all ducks can quack and swim. In addition, the method display() in the subclasses would be overwritten to have statements to implement the display specific to the subclass.



**Figure 6.1** Design using inheritance

Suppose we now have a new requirement that allows ducks to fly. In this case, we could just add the method `fly()` to the superclass `Duck` and all ducks would be able to have this behaviour as shown in Figure 6.2.

However, not all ducks have the same (uniform) behaviour. Suppose we have a rubber duck as a subclass, then we need to overwrite the method `fly()` to do nothing as rubber ducks cannot fly. The method `quack()` needs to be overwritten too since a rubber duck squeaks, not quack.

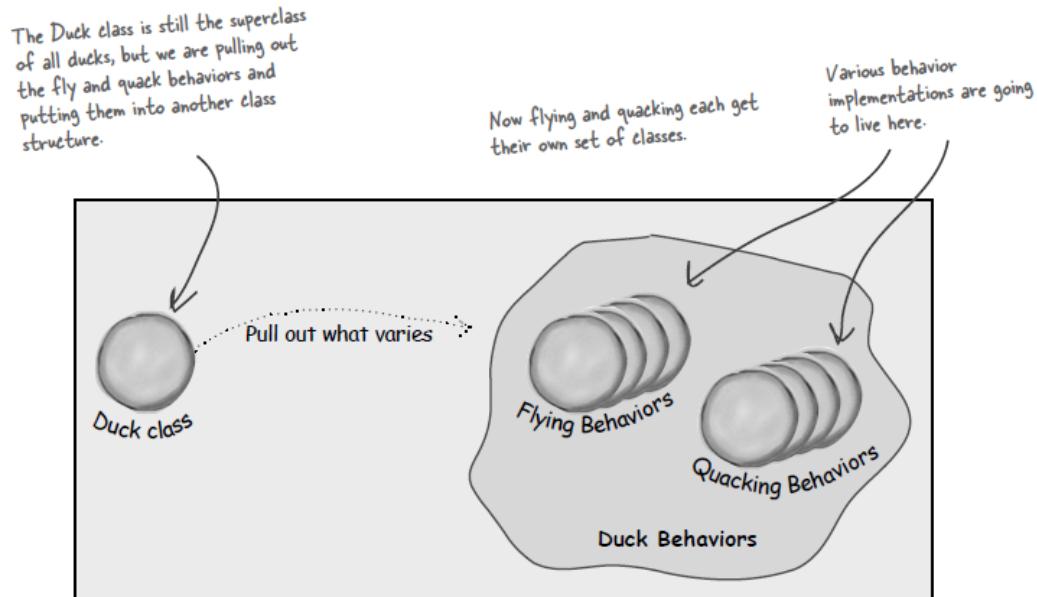


**Figure 6.2** Design with new behaviour

What about new duck types, such as decoy ducks and super ducks? Decoy ducks do not fly and do not quack. Super ducks fly using rockets, and so on. It becomes a nightmare in development as different types of ducks need to be considered individually as each is added to the application.

We need to improve on our design!

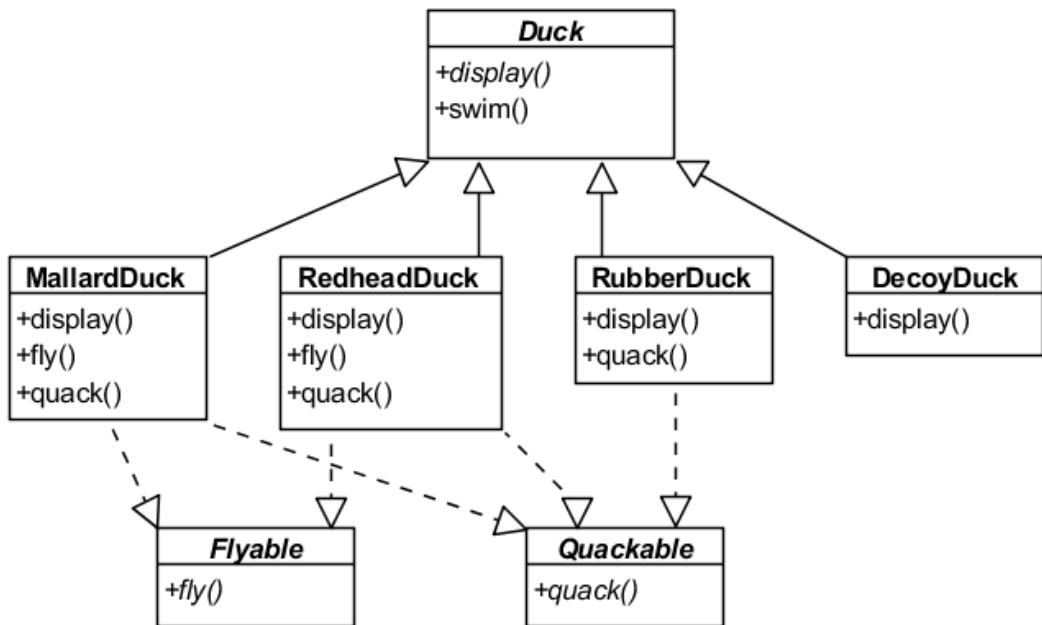
### 2.1.1.2 Design using Interfaces



**Figure 6.3** Extracting the behaviours

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)

Suppose we take out the variations in the duck context class and put these into separate classes. With interfaces, we can have some (but not all) of the duck types behaviours as shown in Figure 6.3. So ducks that can fly like the mallard duck and red head duck will inherit the Quackable interface. Ducks that don't fly and don't quack don't have to inherit any interfaces as shown in Figure 6.4.



**Figure 6.4** Design using interfaces

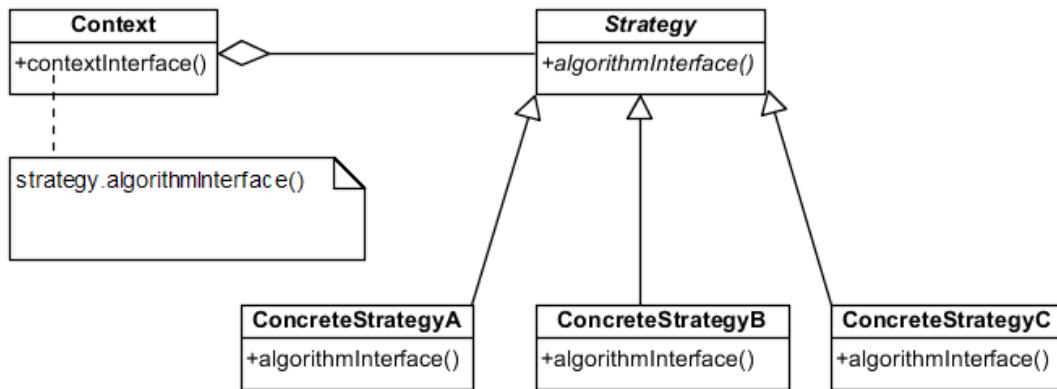


## Watch

Watch this excellent explanation of the Duck example using the Strategy Pattern:

<https://www.youtube.com/watch?v=v9ejT8FO-7I>

Figure 6.5 shows the class **Strategy** which includes the method `algorithmInterface()` that allows the **Context** class to implement a strategy. The concrete **Context** class has an instance variable that references the **Strategy** interface. There may be several **Context** objects that would use the `contextInterface()` method to call the **Strategy**'s `algorithmInterface()` method.



**Figure 6.5** The Strategy Pattern

The key participants of the strategy pattern are:

- **Context**
    - The context is the class that has an association with the Strategy. There may be any number of strategies/algorithms identified. The context is the class that decides which strategy to use.
  - **Strategy**
    - This class is the interface that the Context will use. It has a method algorithmInterface() that the Context object will call.
  - **ConcreteStrategy**
    - These subclasses of the Strategy implements its own version of the algorithmInterface(). You will see that Python implements this differently from other programming languages.

## 2.1.2 Strategy Pattern Implemented

We will use the duck example to illustrate the implementation of the strategy pattern.

Figure 6.6 shows the interfaces used to represent each behaviour. So for the FlyBehaviour, we have different types of flying behaviour: FlyWithWings (for ducks who do normal flying), FlyUnable (for ducks that don't fly like the rubber duck) and FlyWithRocket (for

ducks that fly very fast). The QuackBehaviour has different types of quacking: the normal Quack, the QuackSqueak (for the rubber duck) and the QuackSilence (for the decoy duck).

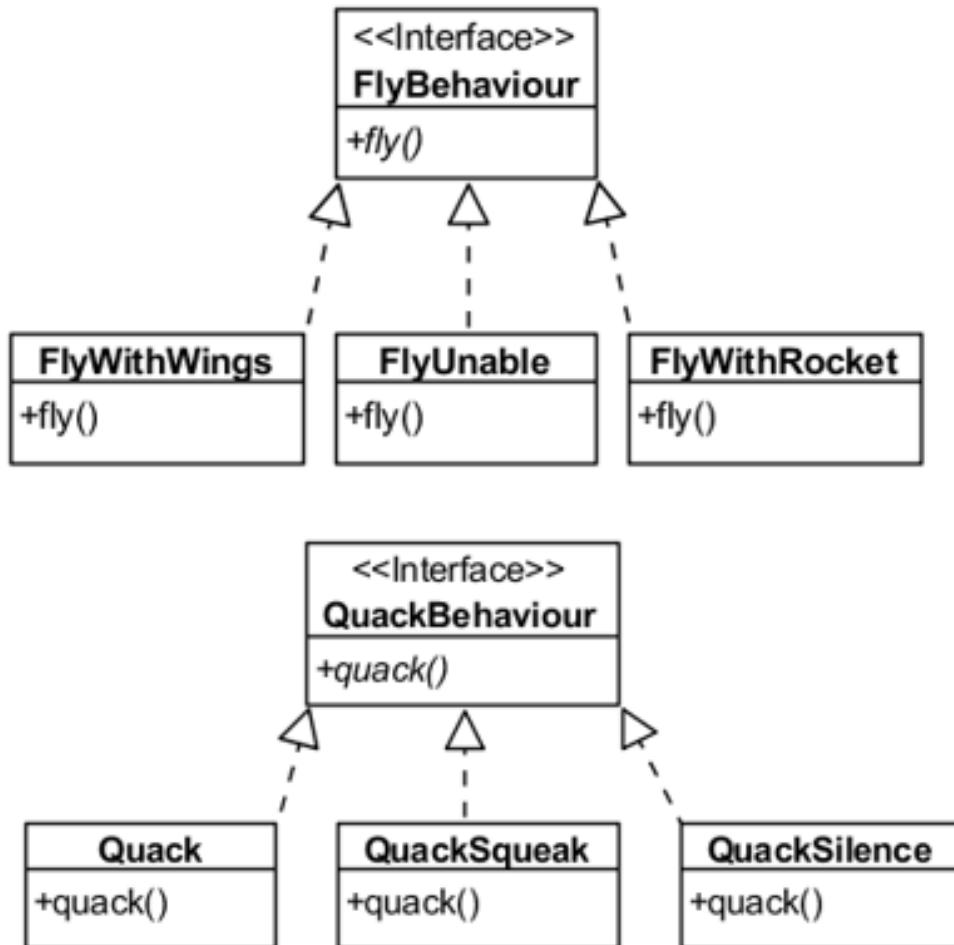


Figure 6.6 Two behaviours

Here is the code for the **FlyBehaviour** and the **QuackBehaviour** Strategy classes. We are importing the `types` module which supports dynamic creation of new types. In this case, we create a new method type. The constructor provides a function as an argument. If a function is passed as an argument, we set the default method to the new function provided by specifying `types.MethodType(function, self)`. This allows a new method to a class to be dynamically found. If the function reference is `None`, the `fly()` method is invoked. The other fly behaviours: **FlyWithWings**, **FlyUnable** and **FlyWithRocket** are implemented as functions.

Similarly, for the QuackBehaviour Strategy class, we create the quack behaviours: Quack (the default method), the QuackSqueak and QuakSilence as functions.

### Source Code: FlyBehaviour

```
import types

class FlyBehaviour:
    def __init__(self, function=None):
        self._name = 'Default strategy'
        # If a reference to a function is provided,
        # replace the fly() method with the given function
        if function:
            self.fly = types.MethodType(function, self)

    def fly(self):
        '''The default flying method'''
        print('I am flying')

    def flyWithWings(self):
        print('Flap Flap')

    def flyWithRocket(self):
        print('Zoooooommmmm')

    def flyUnable(self):
        print('No fly')
```

Here is a test run of the codes:

### Source Code: Test FlyBehaviour

```
s0 = FlyBehaviour()
s0.fly()

s1 = FlyBehaviour(flyWithRocket)
s1.fly()
s1 = FlyBehaviour(flyUnable)
s1.fly()
```

The expected output is:

I am flying

Zooooommمم

No fly

### Source Code: QuackBehaviour

```
import types

class QuackBehaviour:
    def __init__(self, function=None):
        self._name = 'Default quack strategy'
        # If a reference to a function is provided,
        # replace the quack() method with the given function
        if function:
            self.quack = types.MethodType(function, self)

    def quack(self):
        '''The default quack method'''
        print('Quack! Quack!')

    def quackSilence(self):
        print('No sound')

    def quackSqueak(self):
        print('Squeak! Squeak!')
```

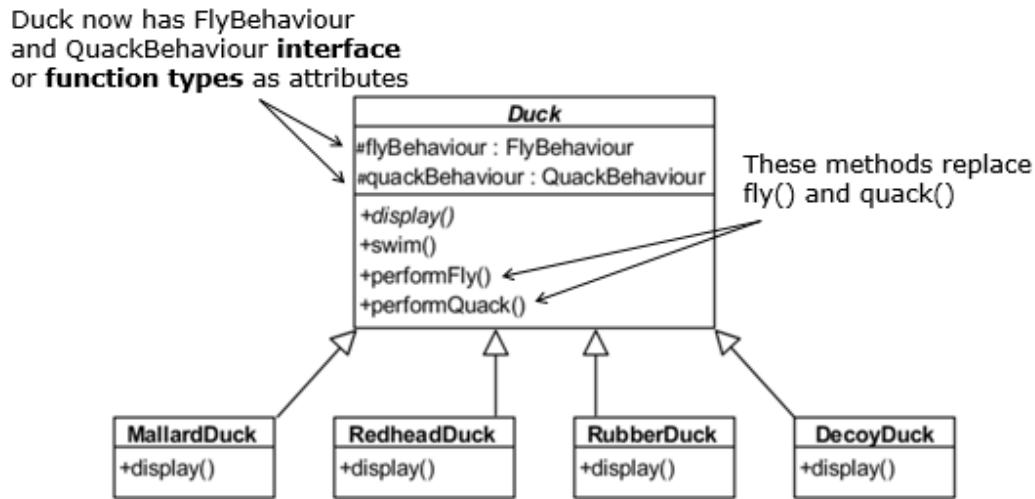


Figure 6.7 The Duck class

We will now be able to design a Duck class that specifies its fly and quack behaviours as shown in Figure 6.7. First we add two instance variables: flyBehaviour and quackBehaviour to the Duck class. Each Duck class will set these variables to reference the specific behaviour it would like at runtime. The Duck class no longer has the methods fly() and quack(). These methods would be placed in the FlyBehaviour and QuackBehaviour classes. Instead, we will have two similar methods called performFly() and performQuack() which will call the fly() and quack() methods respectively. We also have getter and setter methods for the instance variables, a swim() method that just prints 'Splish splash' and the str() method that returns the instance variable name. The code on the next page shows the code for the Duck class.

To test the code, we first create all the behaviours. Then we create a default duck that has the default fly and quack behaviour. We also create a rubber duck that is unable to fly and does not quack and then change its behaviour to fly with rocket and squeaks.

### Source Code: Testing the Duck class

```
s0 = FlyBehaviour()
s1 = FlyBehaviour(flyWithRocket)
s2 = FlyBehaviour(flyUnable)

q0 = QuackBehaviour()
q1 = QuackBehaviour(quackSilence)
q2 = QuackBehaviour(quackSqueak)

d1 = Duck('Default Duck', s0, q0)
print(d1)
d1.performFly()
d1.performQuack()

d2 = Duck('Rubber Duck', s2, q1)
print(d2)
d2.performFly()
d2.performQuack()
d2.flyBehaviour = s1
d2.quackBehaviour = q2
print("Behaviours change")
d2.performFly()
d2.performQuack()
```

The output of the above code is shown below. Note that when the `performFly()` method is called, it executes the `fly()` method of the `FlyBehaviour` class. In a similar manner, when the `performQuack()` method is called, it executes the `quack()` method of the `QuackBehaviour` class.

## Source Code: Duck class

```
class Duck:  
    def __init__(self, name, flyBehaviour, quackBehaviour):  
        self._name = name  
        self._flyBehaviour = flyBehaviour  
        self._quackBehaviour = quackBehaviour  
  
    @property  
    def name(self):  
        return self._name  
  
    @name.setter  
    def name(self, name):  
        self._name = name  
  
    @property  
    def flyBehaviour(self):  
        return self._flyBehaviour  
  
    @flyBehaviour.setter  
    def flyBehaviour(self, flyBehaviour):  
        self._flyBehaviour = flyBehaviour  
  
    @property  
    def quackBehaviour(self):  
        return self._quackBehaviour  
  
    @quackBehaviour.setter  
    def quackBehaviour(self, quackBehaviour):  
        self._quackBehaviour = quackBehaviour  
  
    def performFly(self):  
        self._flyBehaviour.fly()  
  
    def performQuack(self):  
        self._quackBehaviour.quack()  
  
    def swim(self):  
        print('Splish splash')  
  
    def __str__(self):  
        return self._name
```

**Output:**

```
Default Duck
I am flying
Quack! Quack!
Rubber Duck
No fly
No sound
Behaviours change
Zooooommmmm
Squeak! Squeak!
```

### 2.1.3 Strategy Pattern Design Principles

The **strategy pattern** has 3 important principles:

1. **Encapsulate what changes:** Identify the aspects of your application that vary and separate them from what stays the same.

This principle means to take the parts that change and encapsulate them so that in future, when parts are altered or extended, it doesn't affect the parts that don't change. This allows some part of a system to vary independently of all other parts. In the duck simulation, we see that the methods `fly()` and `quack()` work well and don't change. We then pull out these methods and put them into a new set of classes.

2. **Favour composition over inheritance:** Behaviours can change during runtime.

With composition, other classes can re-use our `fly` and `quack` behaviours in the duck simulation. These are no longer hidden in the `Duck` class. New behaviours can also be added without modifying any existing behaviours during runtime. With inheritance, behaviours are locked in during compile time.

3. **Open/Closed Principle.** This is one of the **5 SOLID principles** discussed in ICT162.

Software entities should be open for extension, but closed for modification.

- Bertrand Myer

This principle means to design a common interface (the Closed part of the Principle) but allow for changes in the implementation details (the Open part of the Principle). In the duck simulation, the class FlyBehaviour is written with the types module which assists in dynamic creation of new types. The FlyBehaviour is Closed for modification (no changes allowed after compilation) but Open for extension as a new type can replace the given function during runtime. This is similarly designed for the QuackBehaviour.



### Activity 6.1

For the duck simulation game, suppose some ducks can swim faster than others.

Modify the codes by replacing the method swim() with method performSwim() which calls the method swim() in a new SwimBehaviour class. This class has the normal swim() method that prints 'Splish splash!' and a fastSwim() method that prints 'Splish splash!' 5 times.

Test your codes.

## 2.2 State Pattern

The State pattern allows an object to alter its behaviour when its internal state changes. This is because the pattern encapsulates states into separate classes and assigns the object representing the current state. The object appears to change its class but this is done by simply referencing a different state object. Common elements of the state pattern are shown below.

### Pattern name: State

**Problem Description:** Whenever a new state has to be added or removed from an object, all methods that cause transitions between states are affected. This results in lots of conditional statements that are hard to maintain.

**Solution Description:** This pattern puts each branch in the conditional into a separate class. Each state is therefore an object and can vary independently. The Context class is the class that can have a number of internal states. It has an association to the State class that calls the respective State subclass to handle the events/triggers.

**Consequences:** Because each state needs to be written in a new class, the number of classes in the application is increased and leads to maintenance issues.

#### 2.2.1 State Pattern Defined

From Study Unit 4, we learned about the state of an object which is determined by one or more significant instance variables. When there is a change in the object's internal state, the object changes its behaviour. We accomplished this by using if-else condition blocks of code to perform the different actions based on the stage change. A particular state change is only possible if the required transition is available. So these codes become difficult to maintain as more states are added since a small change in transition logic may lead to change in the condition blocks in each method.



#### Read

Freeman, E., & Robson E. (2020). *Head First Design Patterns*, 2nd Edition, O'Reilly Media, Chapter 10, pp.397-440.

To help you understand the state pattern, we will use the toy capsule from the book, *Head First Design Patterns* by Freeman and Robson but localised to our Singapore context.

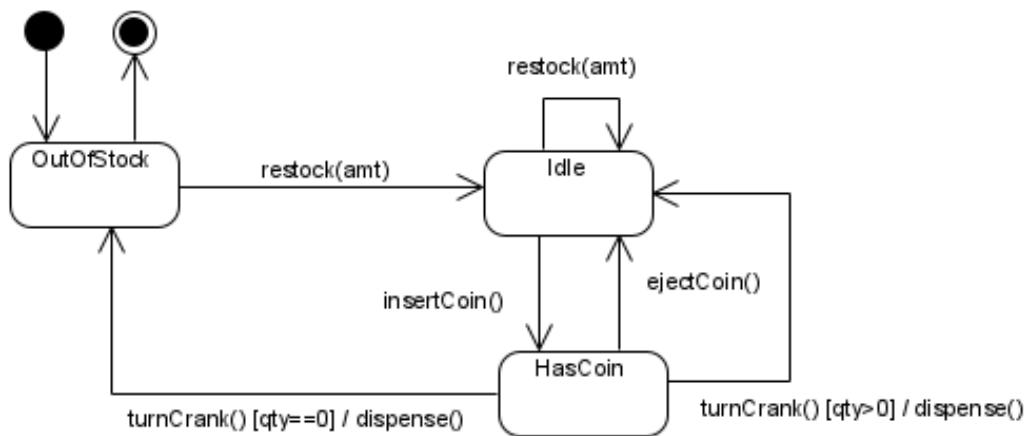


(Source: <http://image.made-in-china.com/6f3j00QvMaeBThIEot/Toy-Capsule-vending-Machine.jpg>)

The toy capsule dispenser can be found in many shops and we will create a high-tech Python-based version of it.

Let's draw a state machine diagram to represent the state of the toy capsule dispenser as it is being used. Our toy capsule dispenser accepts exactly one \$1 coin and does not accept any other types of currency. Examine Figure 6.8 to understand the life cycle of the toy capsule dispenser as it is commissioned until it is decommissioned.

We have identified 3 states of the toy capsule dispenser: OutOfStock, Idle and HasCoin. We have also identified the events or triggers that a user would operate on the toy capsule dispenser: insert a coin, eject the coin, turn the crank and restock (this last event is done by the maintenance staff). The toy capsule dispenser only has one action – which is to dispense a toy. Now write the code!



**Figure 6.8** State machine of the toy capsule dispenser

You know from Study Unit 4, that **coding involves declaring an instance variable, qty (to track of the number of toys)** and **status (to track the state)** for the `ToyMachine` class and **writing a whole bunch of methods**. We need to create a method for each event. Each method uses conditional statements (`if-elif-else`) to determine what behaviour is appropriate in each state and to print error messages if the behaviour is not appropriate. **Because there are 4 events, there will be correspondingly 4 methods.** **Each method would need to check for all the states.** Here we have 3 states. If there are more states, then more `if-elif` would be necessary. That's a lot of code to write. We will leave this exercise to you to try on your own.

### 2.2.1.1 New Requirement

Suppose the Toy company wants to make the dispensers more attractive. It wishes to attract more customers by dispensing two toys (instead of one) 10% of the time.

**Before we modify our program, we need to look at our state machine to see how it can be modified to handle this new requirement.** Figure 6.8 shows one possibility to include this requirement; there are other ways. Note that it is important not to indicate to the customer that he is or is not a winner once he inserts a coin, otherwise he will just insert and eject until he wins!

How does the code change? We would need to add a new Winner state and this would mean having to add a new condition in every single method to handle the Winner state. That's a lot of code to modify. The turnCrank() method is especially messy because the code must be added to check whether there is a winner and then switch to either the Winner state or the Idle state. So this is no good.

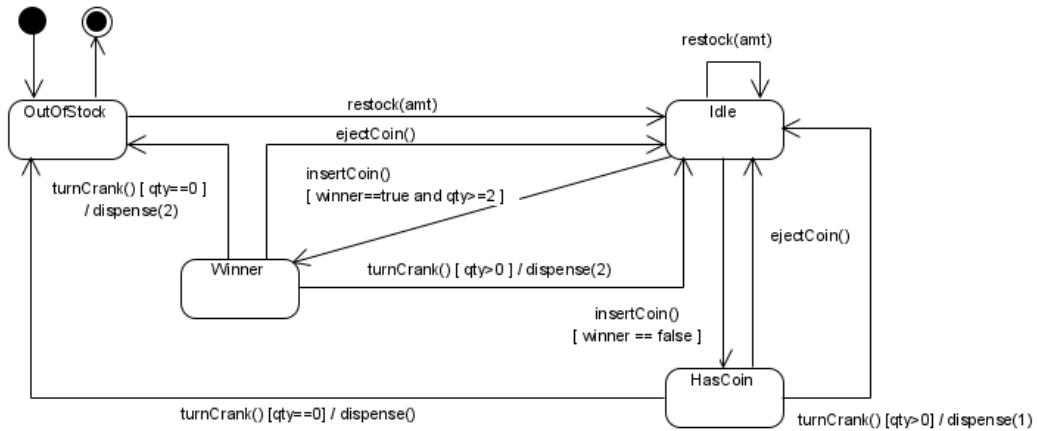
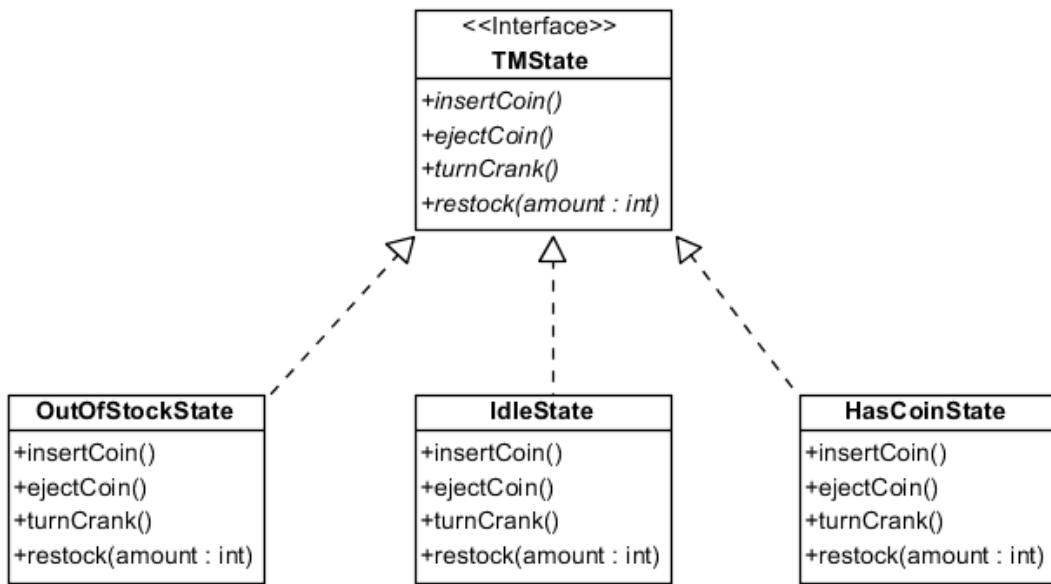


Figure 6.9 State machine of the toy capsule dispenser with new requirement

### 2.2.1.2 Design using Interfaces

Let's look for a better solution that is easier to maintain, and apply what we have learnt from the strategy pattern.



**Figure 6.10** State machine of the toy capsule dispenser with new requirement

First, we define an interface class **TMState** that contains a method for every action in the ToyMachine class. Then we implement a **TMState** class for every state of the machine. These classes will be responsible for the behaviour of the machine when it is in its state. Finally, we get rid of all the conditional code and instead, get the state class to do the work. Figure 6.10 shows the classes for the **TMState**, **OutOfStockState**, **IdleState** and **HasCoinState**. We can add the **WinnerState** later.



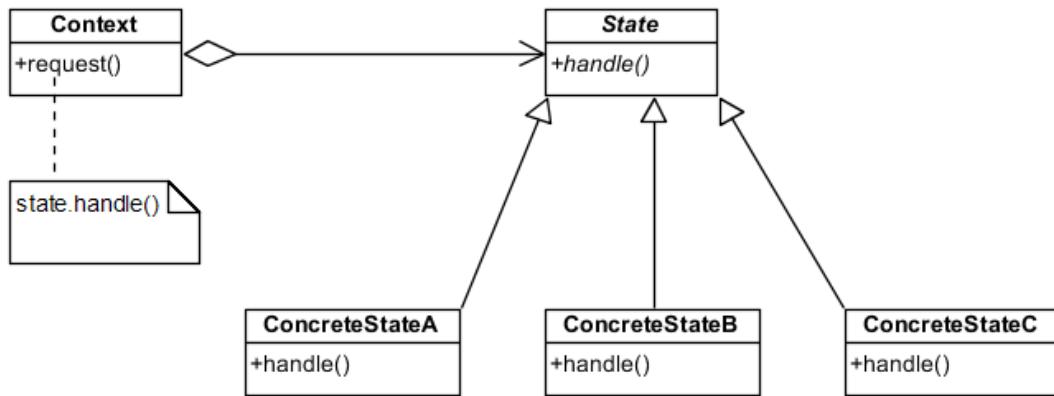
## Watch

Watch this excellent explanation of the State Pattern:

<https://www.youtube.com/watch?v=N12L5D78MAA>

Figure 6.10 shows the class association diagram for the state pattern. The Context class (such as the ToyMachine) has a number of internal states. This class has methods that call the respective State to handle the events / triggers. The class State defines a common interface for all the concrete states. The subclasses ConcreateState all implement the same

interface, so they are interchangeable. They handle the events/triggers from the Context class. Each ConcreteState provides its own implementation for an event/trigger. In this way, when the Context class changes its state, its behaviour will change as well.



**Figure 6.11** State Pattern class association diagram

The key participants of the state pattern are:

- **Context**
  - This class is configured with State objects. It maintains an instance to a ConcreteState object that represents its current state. It has a method request() that invokes the method handle() of the State class.
- **State**
  - This class defines an interface to encapsulate the behaviour of all states in the context. It has a method handle() that the Context object will invoke.
- **ConcreteState**
  - These subclasses implement the behaviour using the State interface. Each subclass implements its own version of the method handle(). We will examine one way of implementing this using Python in the next section.

## 2.2.2 State Pattern Implemented

We will use the toy capsule dispenser to illustrate the implementation of the **state pattern**.

First, we write the **Context** class, **ToyMachine**. This class has **two instance variables**: **qty**, that stores the number of toys in the dispenser and **state**, that stores the state of the toy machine. Its state is set to an **IdleState object** if there are more than one toy in the machine, otherwise, its state is **OutOfStockState**. The **ToyMachine** object is passed to this object so that there is a reference back to the **ToyMachine**. These 2 states are classes that we will write later on. This class also has the methods **addStock()** and **dispense()** and the events (which are now methods in the class) which change the state of the **ToyMachine**: **insertCoin()**, **ejectCoin()**, **turnCrank()** and **restock()**. These 4 methods in turn invoke the respective methods in the **TMState subclass**, using **polymorphism**.

**Source Code: ToyMachine class**

```
class ToyMachine:  
    def __init__(self, amt):  
        self._qty = amt  
        if self._qty > 0:  
            self._state = IdleState(self)  
        else:  
            self._state = OutOfStockState(self)  
  
    @property  
    def qty(self):  
        return self._qty  
  
    @qty.setter  
    def qty(self, qty):  
        self._qty = qty  
  
    @property  
    def state(self):  
        return self._state  
  
    @state.setter  
    def state(self, state):  
        self._state = state  
  
    def addStock(self, amount):  
        self._qty += amount  
  
    def dispense(self, howmany):  
        self._qty -= howmany
```

```
def insertCoin(self):
    self._state = self._state.insertCoin()

def ejectCoin(self):
    self._state = self._state.ejectCoin()

def turnCrank(self):
    self._state = self._state.turnCrank()

def restock(self, amount):
    self._state = self._state.restock(amount)
```

Next, we write the **State class, TMState**. We will use **Python's Abstract Base Class** to define and enforce an interface. For this, we will need to import from the **abc (the abstract base class module)** of the Python library, the **ABC helper class** and the **abstractmethod decorator**. The 4 methods of the class TMState: insertCoin(), ejectCoin(), turnCrank() and restock() would be defined as abstract methods with the keyword, pass. These methods will be implemented in the subclasses: IdleState, OutOfStockState and HasCoinState.

### Source Code: TMState class

```
from abc import ABC, abstractmethod
class TMState(ABC):
    def __init__(self, context):
        self._context = context

    @property
    def context(self):
        return self._context

    @context.setter
    def context(self, context: ToyMachine):
        self._context = context

    @abstractmethod
    def insertCoin(self):
        pass

    @abstractmethod
    def ejectCoin(self):
        pass

    @abstractmethod
    def turnCrank(self):
        pass

    @abstractmethod
    def restock(self, amt):
        pass
```

Lastly, we write the **subclasses**: `IdleState`, `OutOfStockState` and `HasCoinState`. Each of these subclasses will have the **4 methods**: `insertCoin()`, `ejectCoin()`, `turnCrank()` and `restock()`. The respective method will change the state of the context when there is a transition to another state according to the state machine diagram. For example, in the

class IdleState, the method insertCoin() would change the state to HasCoinState. When an action such as dispense() or addStock that needs to be done in a transition, it will also be included in the respective method. If the event (hence method) is not valid when the object is in that state, an appropriate error message will be printed. For example, in the class IdleState, methods ejectCoin() and turnCrank() are not valid events.

### Source Code: IdleState class

```
class IdleState(TMState):
    def __init__(self, context):
        self._context = context

    def insertCoin(self):
        print("You inserted a coin.")
        self._context._state = HasCoinState(self._context)
        return self._context.state

    def ejectCoin(self):
        print("You can't eject your coin, you haven't"\
              " inserted one yet.")
        return self._context.state

    def turnCrank(self):
        print("You can't turn the crank, you haven't"\
              " inserted a coin yet.")
        return self._context.state

    def restock(self, amount):
        self._context.addStock(amount)
        return self._context.state
```

### Source Code: OutOfStockState class

```
class OutOfStockState(TMState):
    def __init__(self, context):
        self._context = context

    def insertCoin(self):
        print("You can't insert coin, machine is out of stock")
        return self._context.state

    def ejectCoin(self):
        print("You can't eject your coin, you haven't"\
              " inserted one yet.")
        return self._context.state

    def turnCrank(self):
        print("You can't turn the crank, the machine"\
              " is empty.")
        return self._context.state

    def restock(self, amount):
        self._context.addStock(amount)
        return self._context.state
```

### Source Code: HasCoinState class

```
class HasCoinState(TMState):
    def __init__(self, context):
        self._context = context

    def insertCoin(self):
        print("You can't insert another coin!")
        return self._context.state

    def ejectCoin(self):
        print("Coin returned")
        self._context._state = IdleState(self._context)
        return self._context.state

    def turnCrank(self):
        self._context.dispense(1)
        print("Toy dispensed. Enjoy!")
        if (self._context.qty > 0):
            self._context._state = IdleState(self._context)
        else:
            self._context._state = OutOfStockState(self._context)
        return self._context.state

    def restock(self, amount):
        print("You can't restock while a customer is"\
              " buying a toy.")
        return self._context.state
```

The codes are tested with the following statements. Examine the results carefully to ensure you understand what it is doing.

### Source Code: Testing the ToyMachine

```
# Testing
toy1 = ToyMachine(1)
toy1.insertCoin()
toy1.turnCrank()
toy1.insertCoin()
toy1.turnCrank()
toy1.restock(10)
print(toy1.qty)
```

Output:

```
You inserted a coin.  
Toy dispensed. Enjoy!  
You can't insert coin, machine is out of stock  
You can't turn the crank, the machine is empty.  
10 .
```

### 2.2.3 State Pattern Design Principles

Did you notice that the state pattern is exactly the same as the strategy pattern? Yes, but they differ in intent.

The state pattern has behaviours encapsulated in state objects and at any time, the context calls one of the states. The current state changes across the set of state objects to reflect the internal state of the context, so the context's behaviour changes as well. The client object therefore knows very little, if any, about these state objects. The state pattern is an alternative to putting lots of conditional statements in the context.

With the strategy pattern, the context is composed with the strategy objects which is specified by the client object. The strategy pattern provides flexibility to change the strategy object at runtime depending on which strategy is most appropriate for the context. The strategy pattern is therefore a flexible alternative to subclassing.

The design principles behind the state pattern are the same principles as those for the strategy pattern. If you recall from Section 2.1.3, these are:

1. Encapsulate what changes: Identify the aspects of your application that vary and separate them from what stays the same.

In the toy capsule dispenser, we see that methods insertCoin(), ejectCoin(), turnCrank() and restock() work well and don't change. We then pull out these methods and put them into a new set of classes.

2. Favour composition over inheritance: Behaviours can change during runtime.

In the ToyMachine class, the state classes are not hidden but can be re-used by any class. New states can also be added without modifying any existing states. The ToyMachine object changes its state during runtime when events are triggered.

3. **Open/Closed Principle:** Classes should be open for extension, but closed for modification.

In the toy capsule dispenser, the various state subclasses are written on a need basis. This assists in dynamic creation of new types. The ToyMachine class is Closed for modification (**no changes allowed after compilation**) but Open for extension as a new state can replace the current state during runtime.



### Activity 6.2

Modify the toy capsule program to include the Winner state.

Test your program.

## 2.3 Observer Pattern

The **observer pattern** can be used in situations where different presentations of an object's state are required. It separates the object that must be displayed from the different forms of presentation. Common elements of the observer pattern are shown below.

**Pattern name:** Observer

**Problem Description:** A subject object needs to be monitored and other observer objects need to be notified when there is a change in the subject.

**Solution Description:** This pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. An abstract class called Subject which has operations such as attaching,

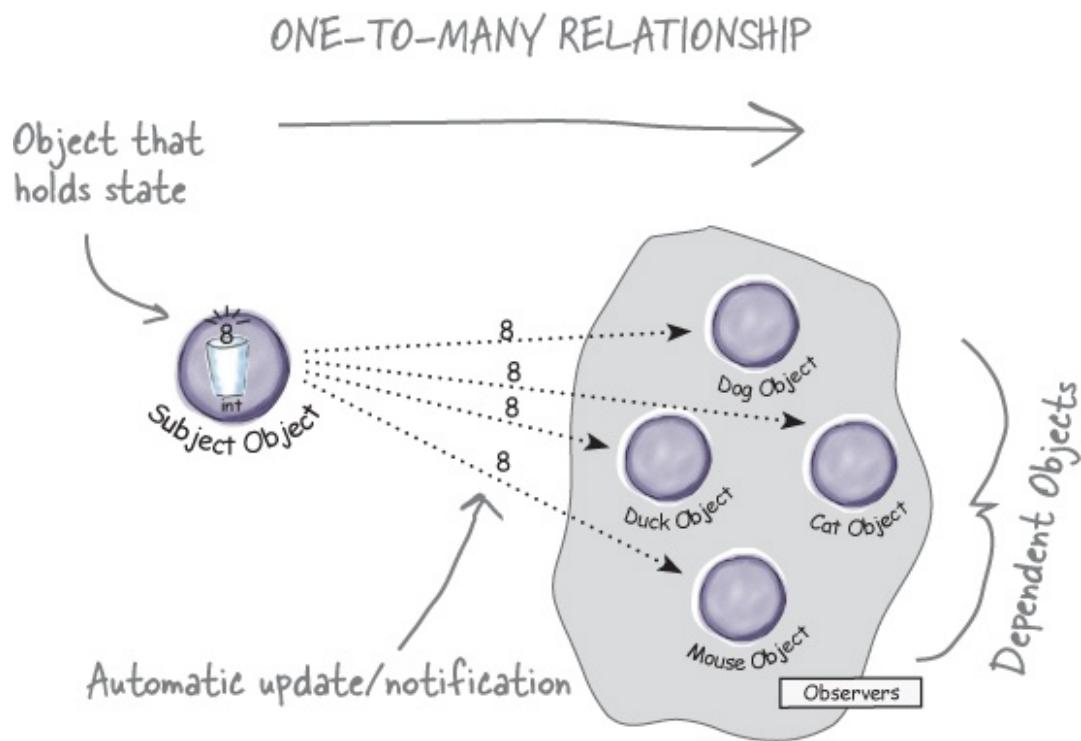
detaching and notifying Observers is needed. Concrete Subject classes inheriting from the abstract Subject class is also needed.

**Consequences:** The Subject only knows the Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimisations may be impractical. Changes to the subject may cause a set of linked updates to the observers to be generated, some of which may not be necessary.

We will explain the observer pattern in the form of a real world analogy. Suppose we have a newspaper subscription. A newspaper agency publishes newspapers. Anyone can subscribe to a newspaper. Whenever there is a new edition, it is delivered to all subscribers, either electronically or physically, depending on the subscription. If a subscriber wishes to unsubscribe from the newspaper, he is allowed to do so and stops receiving new editions. He/she could also re-subscribe later. So the newspaper agency is the publisher. It holds some data of interest (a new edition of the newspaper). When this data changes, all subscribers are notified. In the observer pattern, the publisher is the Subject and the subscribers are the Observers.

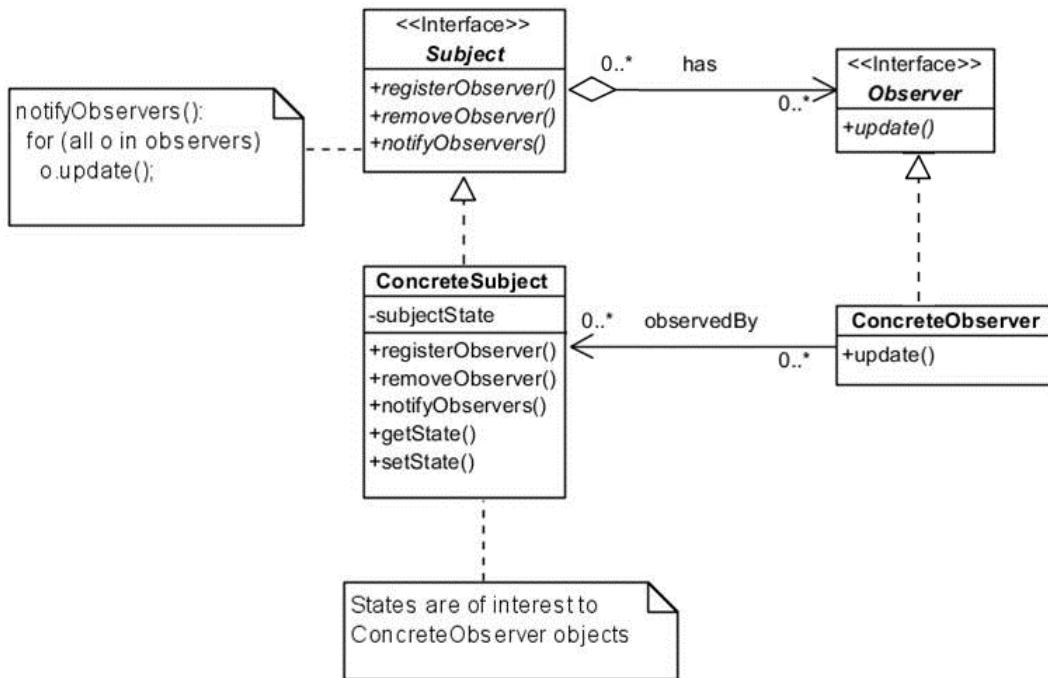
### 2.3.1 Observer Pattern Defined

The observer pattern defines a one-to-many relationship between a set of objects as shown in Figure 6.12. We call these objects the subject and the observer, which is analogous to the publisher and subscriber example in the previous section. We call it a one-to-many because if the state changes in the subject, then the many observers are notified of that state change. All observers are those objects that are dependent on the subject for data. The subject owns the data and there is only one copy of it.



**Figure 6.12** The Observer Pattern

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)



**Figure 6.13** Observer Pattern class association diagram

The class association diagram in Figure 6.13 shows the **Subject interface class** which includes **two methods** that allow observers to register and to remove themselves as observers. It also includes a method for the subject to notify each observer of data changes. **Concrete subjects must implement these methods.** There is also an **observer interface class** that has an **update method** which all **concrete observers** need to implement. A **concrete observer** can be any class that wants to implement the **observer interface**. The **update method** will be called by the subject when the subject's data changes.

The key participants of the observer pattern are:

- **Subject**
  - The subject keeps track of its observers and provides an interface to register / remove observers.
- **Observer**
  - The observer defines an interface for update notification.
- **Concrete Subject**
  - This is the actual object being observed. It sends notification to observers when its state changes.
- **Concrete Observer**
  - This is the observing object. It implements the Observer interface's update method.

### 2.3.2 Observer Pattern Implemented

We will use the weather station example to illustrate the implementation of the **observer pattern**.

The **weather station** is a physical device that acquires **3 types of weather data: humidity, temperature and pressure**. We design a **WeatherData** class that collects these data from the weather station and updates the users' displays of the current weather conditions, weather statistics and a forecast as shown in Figure 6.14.

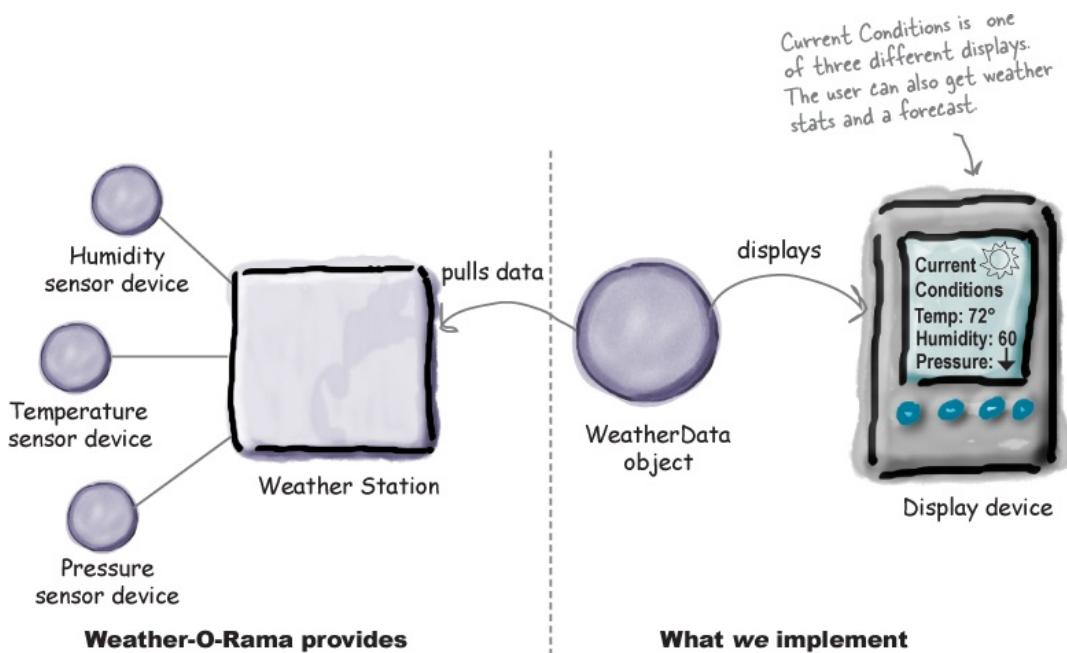


Figure 6.14 Weather Monitoring application

(Source: Freeman, E., & Robson, E. (2004). *Head First Design Patterns*)



## Read

Weather Monitoring application.

Freeman, E., & Robson, E. (2004). *Head First Design Patterns*, Chapter 2, pp.37-42.



## Watch

Watch this excellent explanation of the weather station example using the Observer Pattern:

[https://www.youtube.com/watch?v=\\_BpmfnqjgzQ](https://www.youtube.com/watch?v=_BpmfnqjgzQ) (Observer Pattern)

Figure 6.15 shows the Weather Monitoring application using the **observer pattern**.

First, we write the **Subject** class which has the methods `registerObserver()`, `removeObserver()` and `notifyObservers()`. The method `registerObserver()` checks that the observer is not already in its list of observers before adding it to the list. The method `notifyObservers()` will invoke the method `update()` of all the observers.

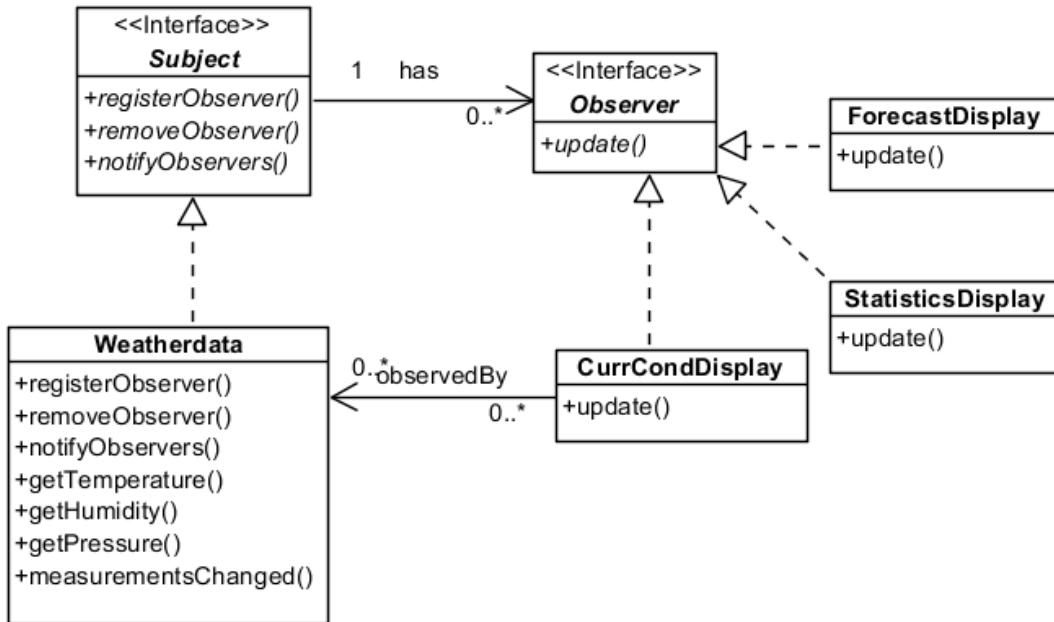


Figure 6.15 Applying Observer Pattern to Weather Monitoring application

### Source Code: Subject class

```
class Subject:

    def __init__(self):
        self._observers = []

    def registerObserver(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def removeObserver(self, observer):
        try:
            self._observers.remove(observer)
        except ValueError:
            pass

    def notifyObservers(self, modifier=None):
        for observer in self._observers:
            if modifier != observer:
                observer.update(self)
```

The **concrete subject**, in this case, the weather station, keeps instance variables **temp**, **humidity** and **pressure**. It has the method **setMeasurements()** which updates its instance variables and invoke the **method notifyObservers()** of its **parent class Subject**.

### Source Code: Concrete Subject class

```
class ConcreteSubject(Subject):

    def __init__(self):
        Subject.__init__(self)
        self._temp = 0
        self._humidity = 0
        self._pressure = 0

    @property
    def temp(self):
        return self._temp

    @property
    def humidity(self):
        return self._humidity

    @property
    def pressure(self):
        return self._pressure

    def setMeasurements(self, temp, humidity, pressure):
        self._temp = temp
        self._humidity = humidity
        self._pressure = pressure
        self.notifyObservers()
```

### Source Code: Observer and Concrete Observer classes

```
class Observer:
    def update(self, subject):
        pass

class CurrCondDisplay(Observer):

    def __init__(self, weatherData):
        self._weatherData = weatherData
        self._weatherData.registerObserver(self)

    def update(self, subject):
        print("Current Condition Display has Temperature {} Humidity {} Pressure {}.\\".
              format(subject._temp, subject._humidity, subject._pressure))
```

The class `Observer` defines the abstract method `update()`. The subclass `CurrCondDisplay`, which is the concrete observer, keeps track of the subject `weatherData` in its instance variable and registers itself with the subject when created. It also implements the method `update()`.

Here is a test of the codes. A concrete subject `s1` is created and a concrete observer `o1` is created and registered with the concrete subject. `s1` then changes the measurements in the next 3 statements.

Examine the output and ensure you understand what the codes are doing.

#### Source Code: Testing the Observer pattern

```
s1 = ConcreteSubject()
o1 = CurrCondDisplay(s1)
s1.setMeasurements(26.6, 65, 30.4)
s1.setMeasurements(27.7, 70, 29.6)
s1.setMeasurements(25.5, 90, 31.2)
```

Output:

```
Current Condition Display has Temperature 26.6 Humidity 65 Pressure 30.4
Current Condition Display has Temperature 27.7 Humidity 70 Pressure 29.6
Current Condition Display has Temperature 25.5 Humidity 90 Pressure 31.2
```

### 2.3.3 Observer Pattern Design Principles

The design principle behind the observer pattern is loose coupling:

Strive for loosely coupled designs between objects that interact.

The subjects and observers are coupled because they interact with each other. However, they are loosely coupled because they really don't know a lot of each other. The subject only knows that the observer implements an interface. It does not need to know the concrete class of the observer. Any class can subscribe to the subject by implementing the observer interface. The subject also does not know any details of the observer. All the subject knows is that it has a list of objects that implement the observable interface. The subject uses this list to notify those observers when something changes. It does not care whether observers have been added, removed or replaced. Any changes made to the subject or observers would not affect the other. This is truly loose coupling.



### Activity 6.3

For the weather station example, create a concrete observer class StatisticsDisplay that displays the average, the maximum and minimum of all the temperatures.

Test your program by creating an Observer object of StatisticsDisplay and statements to add this Observer to the Subject and subsequently removing it.

## Chapter 3: Structural Design Patterns

Structural design patterns describe how classes are actually designed. Extra functionality can be provided by using object-oriented concepts of inheritance, composition and aggregation. We will discuss 2 structural design patterns in this chapter – the Adapter Pattern and the Decorator Pattern.

### 3.1 Adapter Pattern

The adapter converts the interface of a class into another that a client is expecting. It solves the problem that the interfaces are incompatible between classes and the adapter allows them to work together. Common elements of the adapter pattern are shown below.

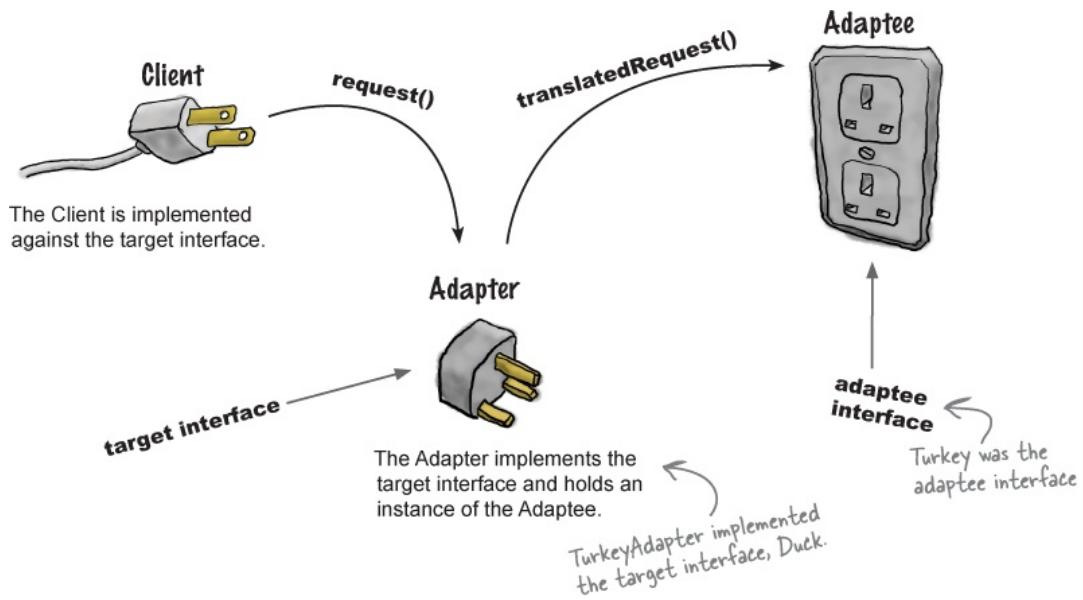
Pattern name: Adapter

**Problem Description:** A client object needs to communicate with another object but their interfaces are incompatible.

**Solution Description:** This pattern defines an **Adaptee class** to translate between the two objects. The Adaptee hides from the client the details of the processing.

**Consequences:** Additional classes are introduced and these may become unmanageable if there are lots of incompatible interfaces in an application.

You must have encountered adapters before. For example, you just bought a home appliance from China and found that its plug does not fit the electrical wall outlet in your home. Figure 6.16 shows that a USA laptop plug does not fit into a European wall outlet. An adapter is required to change the interface of the output into one that fits the laptop.

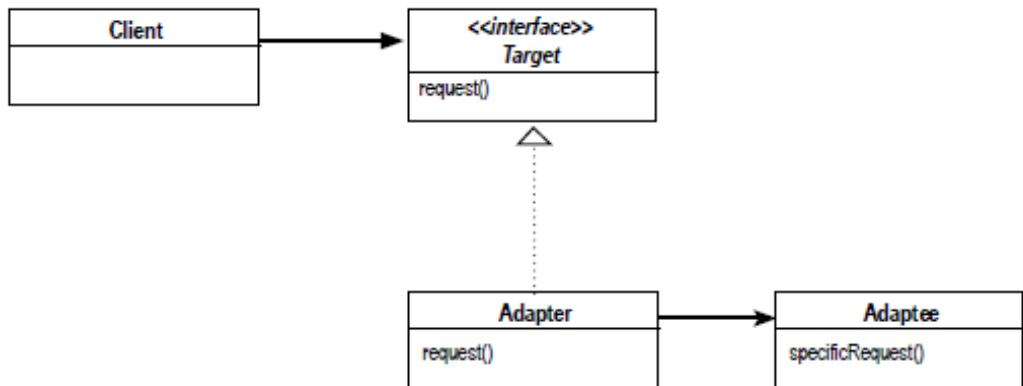


**Figure 6.16** Adapter required

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)

### 3.1.1 Adapter Pattern Defined

There are 3 classes in the adapter pattern: the Client, the Adaptee and the Adapter. The client wants to use the adaptee. But the adaptee has a different interface from the client. The client is not willing to change its existing code to suit the adaptee and the client cannot change the adaptee's code. So we use the adapter that acts as a middleman by receiving requests from the client and converting them into requests that make sense to the adaptee.



**Figure 6.17** Adapter Pattern class association diagram

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)

The key participants of the adapter pattern are:

- **Client**

- The client object makes a request to the adapter object by invoking the method `request()` of the `Target` interface, or directly of the `Adapter` object. It receives the results of the call without knowing there is an adapter doing the translation.

- **Adapter**

- The adapter object translates the method `request()` to the method `specificRequest()` that is understood by the `Adaptee` object. It returns the results of the `specificRequest()` to the client object.

- **Adaptee**

- The adaptee object executes the method `specificRequest()` and returns the result to the adapter object.

### 3.1.2 Adapter Pattern Implemented

We will use the class `Duck` from the strategy pattern of Chapter 2. Suppose we require a duck object in our application but we only have a turkey. Now a duck can quack and fly.

But, turkeys have a different interface from ducks. Turkeys can make a gobble sound and can fly a short distance as shown in the source codes below.

### Source Code: Duck class and Turkey class

```
class Duck:

    def quack(self):
        print('Quack quack')

    def fly(self):
        print("I'm flying!")
```

We write a **TurkeyAdapter class** that takes a Duck object so that this duck can quack by invoking the **turkey's gobble() method** and can fly (like a duck) by invoking the **turkey's fly()** method 5 times, since the turkey can only fly a short distance.

### Source Code: TurkeyAdapter class

```
class TurkeyAdapter(Duck):
    def __init__(self, turkey):
        self._turkey = turkey

    def quack(self):
        self._turkey.gobble()

    def fly(self):
        for i in range(5):
            self._turkey.fly()
```

Let's write some code to test our turkey adapter. Examine the output carefully to ensure you understand what the code is doing.

### Source Code: Testing the TurkeyAdapter

```

print("\nThe duck says:")
duck = Duck()
duck.quack()
duck.fly()

print("\nThe turkey says:")
turkey = Turkey()
turkey.gobble()
turkey.fly()

print("\nThe turkey adapter says:")
duck = TurkeyAdapter(turkey)
duck.quack()
duck.fly()

```

#### Output:

```

The duck says:
Quack quack
I'm flying!

The turkey says:
Gobble gobble
I'm flying a short distance.

The turkey adapter says:
Gobble gobble
I'm flying a short distance.

```

### 3.1.3 Adapter Pattern Design Principles

The **design principle** behind the adapter pattern is the **Dependency Inversion Principle**. This is **one of the 5 SOLID design principles** discussed in ICT162.

**High-level modules should not depend on low-level modules. Both should depend on abstractions.**

**Abstractions should not depend on details. Details should depend on abstractions.**

- Robert C. Martin

The principle means that classes should not depend on each other but depend on an abstraction between them. This can be achieved by introducing an interface between the two classes as shown in Figure 6.17. In our example of the turkey adapter, where a duck is required, but only a turkey is available, we provide a turkey adapter that accepts duck methods but translates these to the methods provided by the turkey.



## Activity 6.4

For the turkey-duck example, we would be exhausted, if there are many other types of birds required. So let's create a generic Adapter class that stores all the objects that fly. To do this, first create an Adapter class that adds an object and its method `fly()` into a dictionary. Then we create a list of fly objects and append these objects to the objects list. Test with the following code:

```
#List to store fly objects
objects = []

#create a Duck object
duck = Duck()

#create a Turkey object
turkey = Turkey()

#Append the objects to the objects list
objects.append(Adapter(duck, fly=duck.fly()))
objects.append(Adapter(turkey, fly=turkey.gobble()))

for obj in objects:
    print("{} says '{}'\n".format(obj.name, obj.fly()))
```

You could create a similar list for noises made by the birds.

## 3.2 Decorator Pattern

The **decorator pattern** allows additional features to be added to an existing object dynamically without using subclassing. We start off with a standard method but wish to make the method fancier by decorating it with additional features. The decorator feature is in-built in Python and you have been using the **@property** decorator which adds an extra function to the method defined to make it act as a getter or setter. In this section, we learn to write our own decorators.

Pattern name: Decorator

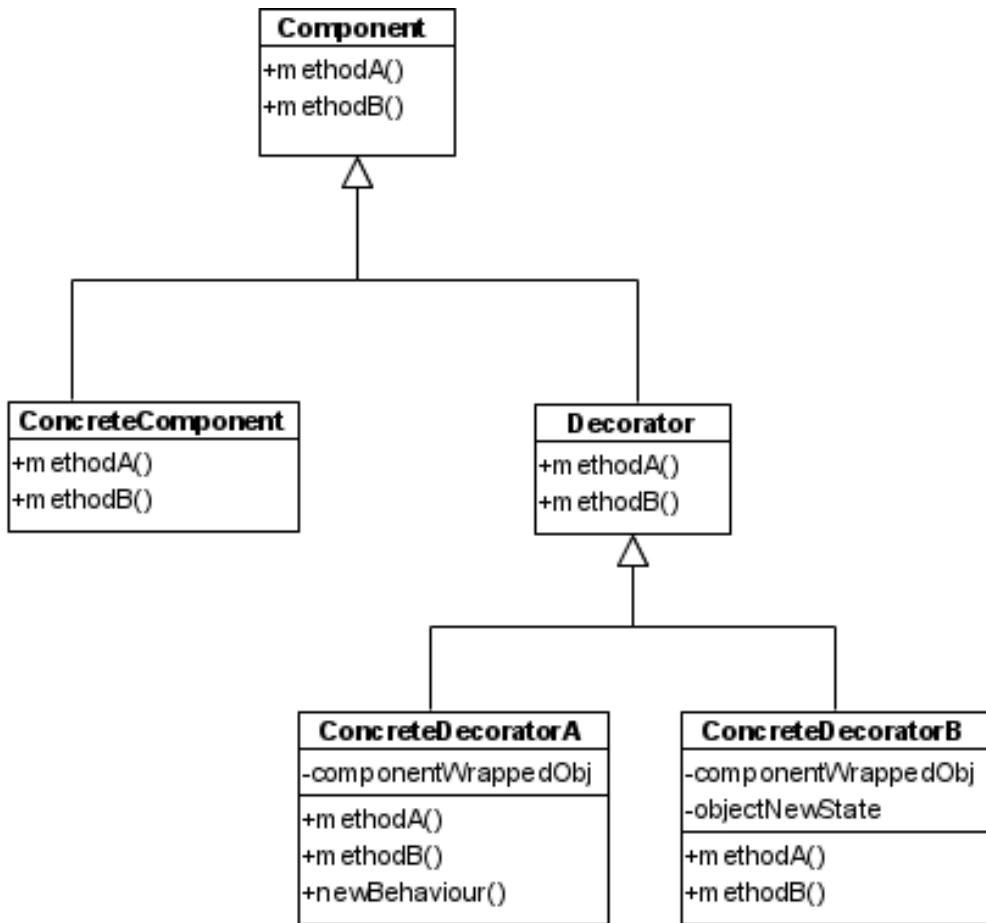
**Problem Description:** Whenever a new requirement is added, inheritance can be used to extend the object behaviour. However, inheritance is done at compile time and this violates one of the SOLID design principles – the Open/Closed Principle.

**Solution Description:** This pattern attaches new features to an object dynamically using composition.

**Consequences:** Similar to the Adapter Pattern, this pattern may create lots of similar decorators, the number of classes in the application is increased and leads to maintenance issues.

### 3.2.1 Decorator Pattern Defined

Suppose we have a Component which needs to be decorated. We create 2 subclasses: the ConcreteComponent class and the abstract Decorator class. New behaviour is added to the class ConcreteComponent dynamically to extend its behaviour. There may be any number of concrete components. Using composition, the class Decorator describes the decoration that can be applied to the component. It has an instance variable that references the Component object. The Decorator can be subclassed with as many ConcreteDecorator classes as necessary, each of which describes a decoration as shown in Figure 6.18.



**Figure 6.18** Decorator class association diagram

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)

The key participants of the decorator pattern are:

- **Component**
  - The component is abstract and describes the object to decorate.
- **ConcreteComponent**
  - Each concrete component describes one type of the component. It can be decorated dynamically at runtime with as many decorators as necessary.
- **Decorator**
  - The decorator is an abstract super type and describes the decoration.

- Concrete Decorator

- Each decorator describes one type of decoration.

 Watch

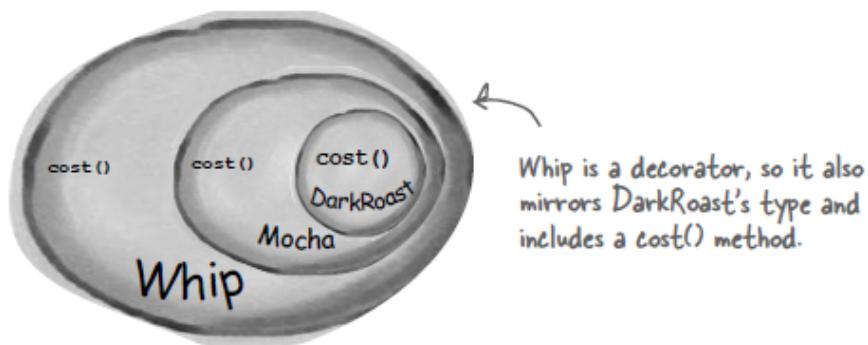
Watch this excellent explanation of the Decorator Pattern:

<https://www.youtube.com/watch?v=GCraGHx6gso>

### 3.2.2 Decorator Pattern Implemented

We will use the **Starbuzz beverage example** from the book, "Head First Design Patterns" by Freeman and Robson to illustrate the **implementation of the decorator pattern**.

In the Starbuzz beverage example, we start with a beverage and decorate it with condiments at runtime. So, when a customer wants a dark roast with mocha and whip, we would take a DarkRoast object, decorate it with the Mocha object and Whip object and call the **method cost()** to add on the condiment costs as shown in Figure 6.19.



So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

**Figure 6.19** Decorators wrapping a component

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)



## Read

Starbuzz beverages application.

Freeman, E., & Robson, E. (2004). *Head First Design Patterns*, Chapter 3, pp.79-92.

Figure 6.20 shows the Starbuzz beverage application using the decorator pattern.

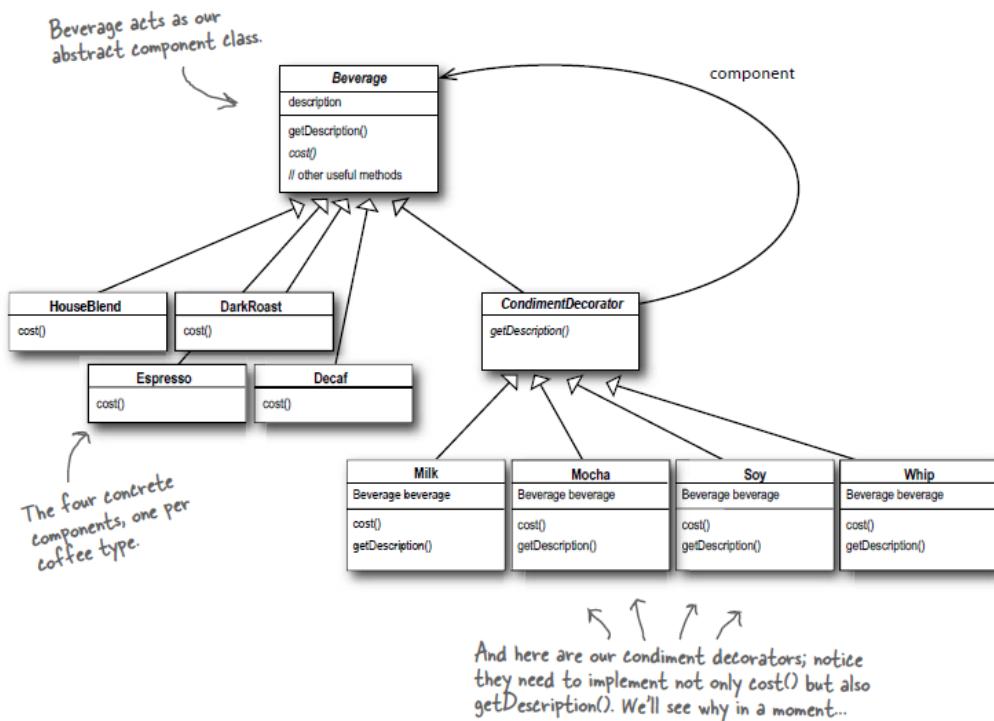


Figure 6.20 Starbuzz beverage application using the decorator pattern

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)

The code below does not use the `@wraps` function which uses `nested (or inner) functions`. We will leave you to explore the use of `@wraps` on your own.

First the class Beverage and its subclasses, Espresso and HouseBlend are created.

### Source Code: Beverage class

```
from abc import ABC
class Beverage(ABC):
    def __init__(self):
        self._description = 'Unknown Beverage'

    @property
    def description(self):
        return self._description

    def cost(self):
        pass

    def __str__(self):
        return self._description
```

### Source Code: Subclasses Espresso and HouseBlend

```
class Espresso(Beverage):
    def __init__(self):
        self._description = 'Expresso'
        self._cost = 1.99

    @property
    def cost(self):
        return self._cost

class HouseBlend(Beverage):
    def __init__(self):
        self._description = 'House Blend Coffee'
        self._cost = 0.89

    @property
    def cost(self):
        return self._cost
```

Next, the **decorators**, **class CondimentDecorator** and its **subclass Mocha** are created. The **decorator mocha** is added as a condiment to the beverage. Extra cost is also added.

### Source Code: CondimentDecorator class and Mocha subclass

```
class CondimentDecorator(Beverage):  
    @property  
    def description(self):  
        pass  
  
    def cost(self):  
        pass  
  
class Mocha(CondimentDecorator):  
    def __init__(self, beverage):  
        self._beverage = beverage  
        self._description = self._beverage.description + " Mocha"  
        self._cost = 0.20 + beverage.cost  
  
    @property  
    def description(self):  
        return self._description  
  
    @property  
    def cost(self):  
        return self._cost  
  
    def __str__(self):  
        return self._description
```

Test the program with the following code fragment which creates an expresso beverage and a house blend beverage that has 2 mocha condiments added.

### Source Code: Testing the Starbuzz Beverage application

```
drink1 = Expresso()  
print('${} : ${:.2f}'.format(drink1, drink1.cost))  
drink2 = HouseBlend()  
print('${} : ${:.2f}'.format(drink2, drink2.cost))  
drink2 = Mocha(drink2)  
print('${} : ${:.2f}'.format(drink2, drink2.cost))  
drink2 = Mocha(drink2)  
print('${} : ${:.2f}'.format(drink2, drink2.cost))
```

Output:

```
Expresso : $1.99
House Blend Coffee : $0.89
House Blend Coffee, Mocha : $1.09
House Blend Coffee, Mocha, Mocha : $1.29
```

### Research – Python Decorators

Python has a module for higher-order functions called `functools`. Higher-order functions act on or return other functions. The `wraps()` function in `functools` is a decorator. It uses the concept of inner (or nested) functions, taking another function and extends its behaviour without explicitly modifying it. It is beyond the scope of this course to cover python decorators. If you are interested, a good introduction is found here: <https://bit.ly/3daEd4o>

### 3.2.3 Decorator Pattern Design Principles

The **design principle** behind the decorator pattern is the **Open/Closed Principle**. In the Starbuzz beverage application, each beverage is closed for modification but open for extension as it may have many condiments added, creating a new beverage using **composition**.



### Activity 6.5

For the Starbuzz beverage example, create subclasses `Soy` and `Whip` of the class `CondimentDecorator`. `Soy` costs 10 cents and `Whip` costs 15 cents.

Test your codes by creating a House Blend coffee with double Mocha and Whip and a HouseBlend coffee with Soy, Mocha and Whip.

How much does each beverage cost?

## Chapter 4: Creational Design Patterns

Creational design patterns focus on the instantiation of objects and provide ways to have more flexibility in how objects are actually created. We will discuss 2 creational patterns in this chapter – the Factory Pattern and the Singleton Pattern.

### 4.1 Factory Pattern

The factory pattern is used in situations where objects need to be created especially in situations where there is uncertainty in the type of objects that are eventually needed in the system. For example, a bank allows customers to have accounts. As time passes, the bank also offers saving accounts and current accounts which are special types of accounts. Subsequently, the bank offers online accounts which is a special type of saving account. Therefore, the system needs to handle all types of accounts. Common elements of the factory pattern are shown below.

**Pattern name:** Factory

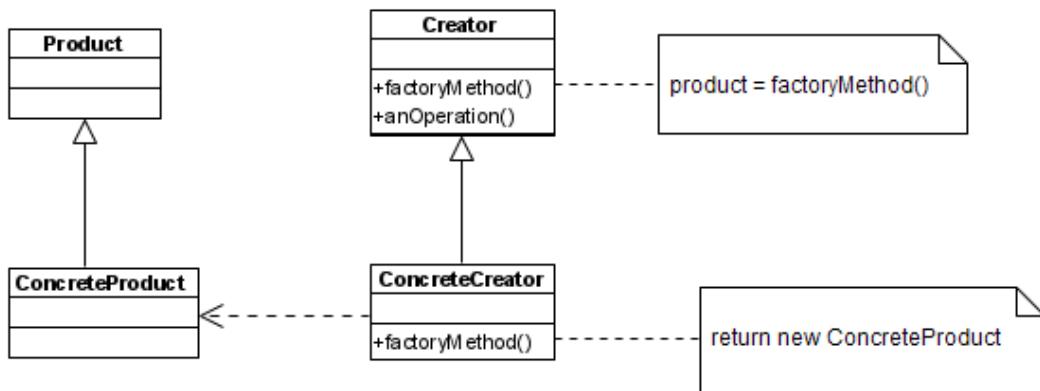
**Problem Description:** We want to be able to create new objects but do not want to know about how to create them, which class to instantiate them in, the logic used, etc.

**Solution Description:** This pattern creates objects for clients using a common interface. It hides the actual implementation of the objects. The clients who use these new objects do not have to worry about creating, managing and destroying them.

**Consequences:** The number of classes in the application may increase since clients may need to sub-class the original creator class. This class hierarchy that is produced results in a high degree of coupling between the classes, leading to maintenance issues of the classes when changes are required.

### 4.1.1 Factory Pattern Defined

In the factory pattern, we have a factory that produces products but leaves the decision to which product to make to the Factory subclasses. To do this, we have a Creator superclass that specifies an abstract method `factoryMethod()` that all Concrete Creator subclasses must implement. In parallel, we have an abstract Product superclass with several Product subclasses which are essentially concrete variations of the Product superclass. Each concrete creator is a factory that can produce a particular concrete product. The client can choose from a variety of concrete creators to determine which kind of concrete product that it wants without it being dependent on a particular factory or a particular product as shown in Figure 6.21.



**Figure 6.21** Factory Pattern class association diagram

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)



### Read

Pizza Store application.

Freeman, E., & Robson, E. (2020). *Head First Design Patterns*, Chapter 4, pp.109-132.

The key participants of the factory pattern are:

- Product
  - The product defines an interface common to all products.
- Concrete Product
  - The concrete product implements the algorithm to create the products. It is the only class that has this knowledge.
- Creator
  - The creator is an abstract class that contains the implementations of all the methods to manipulate products, except for the factory method.
- Concrete Creator
  - The concrete creator implements the factoryMethod(), which is the method that actually produces the products.



## Watch

Watch this excellent explanation of the Factory Pattern:

<https://www.youtube.com/watch?v=EcFVTgRHJLM>

### 4.1.2 Factory Pattern Implemented

We will use the pizza store example from the book, "Head First Design Patterns" by Freeman, E. and Robson, E., to illustrate the implementation of the factory pattern.

In the pizza store example, the local pizza store deals with two different kinds of pizza stores: one for New York style pizza that makes a thin crust pizza and one for Chicago style pizza that makes a thick crust pizza. The orderPizza() method in the pizza store is the same

for both factories, but the `createPizza()` method in the New York store will make New York style pizzas. Similarly, the `createPizza()` method in the Chicago store will make Chicago style pizzas. So the subclass is fully responsible for which Concrete Pizza it instantiates.

Figure 6.22 shows the Pizza Store application using the factory pattern.

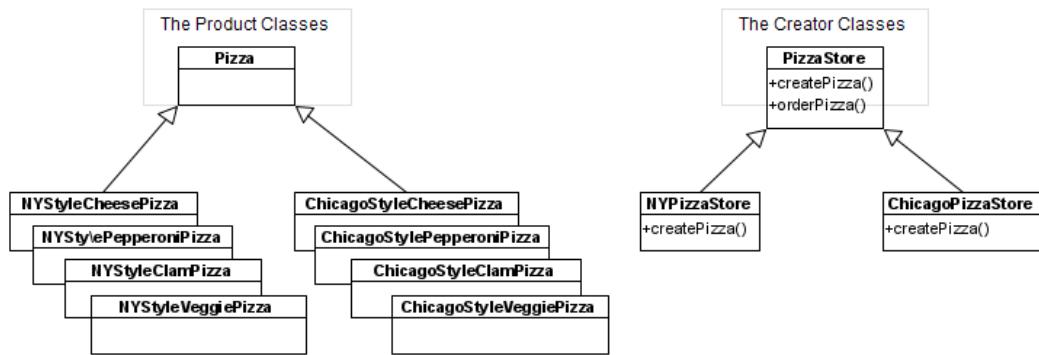


Figure 6.22 Pizza Store application using factory pattern

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)

Here are the codes for the product class and its subclasses.

### Source Code: Pizza class

```
from abc import ABC

class Pizza(ABC):
    def __init__(self, name, dough, sauce):
        self._name = name
        self._dough = dough
        self._sauce = sauce
        self._toppings = []

    @property
    def name(self):
        return self._name

    def prepare(self):
        print("Preparing " + self._name)
        print("Tossing dough...")
        print("Adding sauce...")
        print("Adding toppings: ")
        for aTopping in self._toppings:
            print("\t" + aTopping)

    def bake(self):
        print("Bake for 25 minutes at 350")

    def cut(self):
        print("Cutting the pizza into diagonal slices")

    def box(self):
        print("Place pizza in official Pizza box")
```

The superclass Pizza has methods to prepare, bake, cut and box the pizza.

### Source Code: Pizza subclasses

```
class NYStyleCheesePizza(Pizza):
    def __init__(self):
        super().__init__("NY Style Sauce and Cheese Pizza",\
                         "Thin Crust Dough", "Marinara Sauce")
        self._toppings.append("Grated Reggiano Cheese")

class ChicagoStyleCheesePizza(Pizza):
    def __init__(self):
        super().__init__("Chicago Style Deep Dish Cheese Pizza",\
                         "Extra Thick Crust Dough", "Plum Tomato Sauce")
        self._toppings.append("Shredded Mozzarella Cheese")

    def cut(self):
        print("Cutting the pizza into square slices")
```

Here are the codes for the **creator class and its subclasses**. The **abstract method createPizza()** is the **factory method** for its subclasses to implement since each pizza store creates a different style of pizza.

**Source Code: PizzaFactory class and its subclasses**

```
from abc import ABC
class PizzaStore(ABC):

    def createPizza(self, type):
        pass

    def orderPizza(self, type):
        pizza = self.createPizza(type)
        pizza.prepare()
        pizza.bake()
        pizza.cut()
        pizza.box()
        return pizza

class NYPizzaStore(PizzaStore):
    def createPizza(self, type):
        if type == "cheese":
            return NYStyleCheesePizza()
        else:
            return None
        ...
        if type == "veggie":
            return NYStyleVeggiePizza()
        if type == "clam":
            return NYStyleClamPizza()
        etc.
        ...
```

```
class ChicagoPizzaStore(PizzaStore):
    def createPizza(self, type):
        if type == "cheese":
            return ChicagoStyleCheesePizza()
        else:
            return None
        ...
        if type == "veggie":
            return ChicagoStyleVeggiePizza()
        if type == "clam":
            return ChicagoStyleClamPizza()
    etc.
    ...
```

We test the program with the following code fragment which produces the output as expected. Examine the output carefully to ensure you understand how the factory pattern works.

#### Source Code: Testing the Pizza Factory

```
pizza = ChicagoStyleCheesePizza()
print(pizza.name)

nyStore = NYPizzaStore()
chicagoStore = ChicagoPizzaStore()
pizza = nyStore.createPizza("cheese")
pizza = nyStore.orderPizza("cheese")
print('Ethan ordered a', pizza.name)

pizza = chicagoStore.orderPizza("cheese")
print('Joel ordered a', pizza.name)
```

#### Output:

```
Chicago Style Deep Dish Cheese Pizza
Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
    Grated Reggiano Cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official Pizza box
Ethan ordered a NY Style Sauce and Cheese Pizza
Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
    Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official Pizza box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

#### 4.1.3 Factory Pattern Design Principles

The **factory pattern** is built on 3 important principles:

1. **Encapsulate what changes:** Identify the aspects of your application that vary and separate them from what stays the same.

In the pizza store application, we see that there are different types of pizzas. Instead of writing conditional logic, we move the code to create pizzas into separate classes. This is the **factory method**, `createPizza()` which creates a pizza and returns an object that implements the pizza interface. The creation code is now in one place, nicely separated from the other codes. The **method** `orderPizza()` does not have to worry about creating the pizza; it just invokes this factory method and does the other steps in the pizza making process.

2. **Open/Closed Principle.**

In the pizza store example, an **abstract Pizza class** is created that defines methods to **prepare, bake, cut and box a pizza**. This class is closed for modification.

However, it is open for extension as subclasses of this superclass can be created for different types of pizzas.

### 3. Dependency Inversion Principle.

In the pizza store example, two abstractions, Pizza and PizzaStore are defined. The class Pizza gives you different methods to create all types of pizza based on the dough, sauce and toppings. The class PizzaStore allows you to order a pizza of any type (e.g. cheese, veggie, clam) and in any style (e.g. New York or Chicago) that you wish. By introducing these abstractions, the dependencies between the high-level classes and lower-level classes are removed.



### Activity 6.6

A new California Pizza Store has just opened which offers all types of California Pizza.  
Re-draw the diagram in Figure 6.22 to incorporate this new store.

## 4.2 Singleton Pattern

The singleton pattern solves the problem that only one object is required to be instantiated from a class. This object is needed to manage a shared resource.

Pattern name: Singleton

**Problem Description:** There are objects that need to be created once and available to the entire application for the life of the application. This gives control to the class as to how the object is created, accessed and used in the application. Applications that allow multiple creation of objects of a class may encounter concurrency issues with shared data.

**Solution Description:** This pattern defines a single class that ensures only one instance of this class is created.

**Consequences:** Singleton is tightly coupled with one instance. Hence, if code in the Singleton is changed, it could impact client objects that use the Singleton. The way the

class is to be written will also depend on the programming language used since each programming language has different ways to handle **class variables** and **class methods**, **thread safety**, etc.

#### 4.2.1 Singleton Pattern Defined

For some applications, we may wish to have only one instance of an object. Here are some examples. For a chat room application, we would only want to have one chat room at any point in time. A printer spooler can be called from more than one place in the network but there should only be one printer spooler. There should also be only one database connection in the entire application. A venue ticketing system should have only one object that can issue tickets so that the same seats are not assigned to more than one user.



#### Read

Chocolate Boiler application.

Freeman, E., & Robson, E. (2004). *Head First Design Patterns*, Chapter 5, pp.169-177.



**Figure 6.23** Chocolate boiler

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)

<b>ChocolateBoiler</b>	
-instance	
-empty : boolean	
-boiled : boolean	
+getInstance()	
+fill()	
+boil()	
+drain()	

**Figure 6.24** Chocolate boiler class

To help you understand the Singleton Pattern, we will use the [chocolate boiler example from the book](#), *Head First Design Patterns* by Freeman and Robson. In the chocolate factory, a chocolate boiler as shown in Figure 6.23, is used in the process of making chocolate. It is filled with chocolate and milk, boiled and drained. The factory has to ensure that errors

like draining the unboiled mixture, or filling the boiler when it is already full or boiling an empty boiler does not happen.

The chocolate boiler is designed to be a Singleton class as shown in Figure 6.24. The class variables empty and boiled store boolean values to determine the state of the chocolate boiler. These class variables are used in the class methods fill(), boil() and drain(), to ensure that the rules for filling boiling and draining are followed. The class variable, instance, holds one and only one instance of the Singleton class. This class variable is accessible through the class method getInstance(). Since it is a class method, this is a global access point to the unique class variable, instance.



## Watch

Watch this excellent explanation of the Singleton Pattern:

[https://www.youtube.com/watch?v=hUE\\_j6q0LTQ](https://www.youtube.com/watch?v=hUE_j6q0LTQ)

Figure 6.25 shows the Singleton class, the key participant of the singleton pattern:

- Singleton
  - The singleton class has a class variable that holds one and only instance of the Singleton. It has a getInstance() class method and its own set of data and methods.

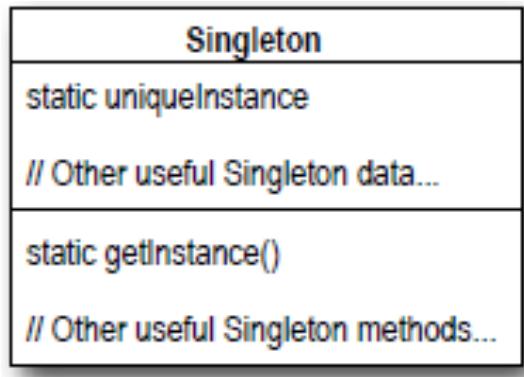


Figure 6.25 Singleton class diagram

(Source: Freeman, E., & Robson, E., *Head First Design Patterns*)

#### 4.2.2 Singleton Pattern Implemented

We will use the chocolate boiler example to illustrate the implementation of the singleton pattern.

The ChocolateBoiler Singleton class has class variables, instance, empty and boiled. It is written with print() statements in the class methods fill(), boil() and drain() to display whether the action was done.

The Singleton class is then tested by creating 2 chocolate boilers, b and b2. Examine the output to satisfy yourself that b2 refers to the same chocolate boiler as b. Note that, in particular, when the b2 chocolate boiler wanted to fill the boiler (see line 10 of the test code), it could not because the chocolate boiler was already full. When the b2 chocolate boiler was assigned in line 8 of the test code, its instance variables empty and boiled were not set to true and false respectively; instead, it was referring to the instance variables in the existing b chocolate boiler.

### Source Code: ChocolateBoiler class

```
class ChocolateBoiler:  
    __instance = None  
    __empty = True  
    __boiled = False  
  
    @classmethod  
    def getInstance(cls):  
        if not cls.__instance:  
            cls.__instance = ChocolateBoiler()  
        return cls.__instance  
  
    def fill(self):  
        if self.__empty:  
            print("\tboiler is filled")  
            self.__empty = False  
            self.__boiled = False  
        else:  
            print("\tcannot fill - boiler is not empty")  
  
    def drain(self):  
        if not(self.__empty) and self.__boiled:  
            print("\tdrain the boiled milk and choc")  
            self.__empty = True  
        else:  
            print("\tcannot drain - boiler is empty or not boiled")  
  
    def boil(self):  
        if not(self.__empty) and not(self.__boiled):  
            print("\tbring the contents to a boil")  
            self.__boiled = True  
        else:  
            print("\tcannot boil - boiler is not empty or not boiled")
```

### Source Code: Testing the ChocolateBoiler class

```
1 print("create boiler")
2 b = ChocolateBoiler.getInstance()
3 print("fill boiler")
4 b.fill()
5 print("boil boiler")
6 b.boil()
7 print("create boiler")
8 b2 = ChocolateBoiler.getInstance()
9 print("fill boiler")
10 b2.fill()
11 print("drain boiler")
12 b2.drain()
```

### Output:

```
create boiler
fill boiler
    boiler is filled
boil boiler
    bring the contents to a boil
create boiler
fill boiler
    cannot fill - boiler is not empty
drain boiler
    drain the boiled milk and choc
```

### 4.2.3 Singleton Pattern Design Principles

The Singleton has 2 important principles:

1. **Encapsulate what changes:** Identify the aspects of your application that vary and separate them from what stays the same.

Because the Singleton class encapsulates the instance of the class, the Singleton class can be designed to allow access to whatever data the Singleton is managing.

In the chocolate boiler application, the class variable instance allows clients to have access to the single, unique instance.

## 2. Single Responsibility Principle

A class should have one, and only one, reason to change.

- Robert Martin

The single responsibility of the Singleton class is to manage the data. This makes the code maintainable because the data is not being managed anywhere else in the application. In the chocolate boiler, different clients, b and b2 are managing the same object.



### Activity 6.7

Another way to write a singleton class is to separate the Context class from the singleton. Here are the steps to do this:

1. Write a normal class, e.g. ChocolateBoiler, that has instance variables empty and boiled and the methods that would ensure that the operations fill, boil and drain are done correctly.
2. Write a singleton class that inherits from the ChocolateBoiler class. Create a class variable, instance with a class method that will return an instance of the ChocolateBoiler object. This makes the singleton class have the ChocolateBoiler class as a global object.

Try it!

## Summary

The following points summarise the contents of this study unit:

- Design patterns are common, repeatable solutions for creating programs. They describe best practices for solving common software design problems that occur again and again.
- Using design patterns not only short-cut the design process by leveraging on the work of other developers who have already gone through a similar issue, it can make your programs resilient to changes.
- Design patterns solve creational (what creation mechanisms to create objects), behavioural (what algorithms should the objects use) and structural (how to assemble objects into larger structures) problems that result in flexible and efficient software.
- Applying object oriented concepts of encapsulation, inheritance and polymorphism is not sufficient to designing good software. Design principles are additional guidelines that are underlying in each design pattern that help you understand the design pattern better.
- Design patterns are represented by class association diagrams and depending on the programming language used, there are different ways to implement the design patterns.

## Formative Assessment

1. An object is an instance of a class. What is a concrete instance when working with design patterns?
  - a. A concrete instance refers to any occurrence of objects that exist during the running of code.
  - b. A concrete instance refers to dynamic object values that are always changing.
  - c. A concrete instance refers to a fixed object value that cannot be changed.
  - d. A concrete instance refers to the behaviour or state of an object.
2. What group of design patterns describes how inheritance can be used to provide additional functionality?
  - a. Structural
  - b. Creational
  - c. Behavioural
  - d. Definitional
3. How are algorithms defined in the strategy pattern?
  - a. The strategy pattern defines mathematics operations as algorithms.
  - b. The strategy pattern defines a family of algorithms, encapsulates each one and makes them interchangeable.
  - c. The strategy pattern defines one instance of a class with algorithms.
  - d. The strategy pattern defines tough programming tasks as algorithms.
4. What is a state pattern?
  - a. It allows object to be notified when state changes.
  - b. It allows subclasses to decide how to implement steps of an algorithm.
  - c. It encapsulates state-based behaviours and uses delegation to switch between behaviours.

- d. It encapsulates interchangeable behaviours and uses delegation to decide which one to use.
5. The Observer pattern has two components: subscriber and publisher. Which type of relationship exists with the subscriber and publisher?
- a. It is a one-to-many relationship. There is one publisher to many subscribers.
  - b. It is a many-to-one relationship. There are many publishers to one subscriber.
  - c. Publishers send messages and subscribers receive the messages.
  - d. Publishers post messages to a message broker, and subscribers register subscriptions to the message broker.
6. Why is the Open/Closed principle important to the Decorator pattern?
- a. The Open/Closed principle explains that Decorator classes should have conditional statements to switch between open and closed.
  - b. The Open/Closed principle explains that Decorator classes should be open for extension, but closed for modification.
  - c. The Open/Closed principle explains that Decorator classes need to be tightly coupled to improve performance.
  - d. The Open/Closed Principle means that Decorator classes can be switched to open or closed classes.
7. How does Factory pattern provide loose coupling and high cohesion?
- a. The Factory pattern is a creational pattern that is flexible and hence uses loose coupling.
  - b. The Factory pattern divides classes into categories and uses abstract methods.
  - c. The Factory pattern uses configuration files instead of coding to create loose coupling.
  - d. The Factory pattern encapsulates object creation logic which makes it easy to change later.

# Solutions or Suggested Answers

## Activity 6.1

```

class SwimBehaviour:
    def __init__(self, function=None):
        self._name = 'Default swim strategy'
        # If a reference to a function is provided,
        # replace the swim() method with the given
function
        if function:
            self.swim = types.MethodType(function, self)

    def swim(self):
        '''The default flying method'''
        print('Splish Splash!')

    def fastSwim(self):
        print('Splish Splash! '*5)

    class Duck:
        def __init__(self, name, flyBehaviour,
quackBehaviour, swimBehaviour):
            self._name = name
            self._flyBehaviour = flyBehaviour
            self._quackBehaviour = quackBehaviour
            self._swimBehaviour = swimBehaviour

        # other methods not shown

        @property
        def swimBehaviour(self):
            return self._swimBehaviour

        @swimBehaviour.setter
        def swimBehaviour(self, swimBehaviour):
            self._swimBehaviour = swimBehaviour

        def performFly(self):
            self._flyBehaviour.fly()

        def performQuack(self):
            self._quackBehaviour.quack()

        def performSwim(self):
            self._swimBehaviour.swim()

```

Test with the following code fragment:

```
d1 = Duck('Default Duck', s0, q0, s0)
print(d1)
d1.performSwim()
d2 = Duck('Rocket Duck', s1, q1, s1)
print(d2)
d2.performSwim()
```

## Activity 6.2

Codes added are in bold. The ToyMachine needs an instance variable, winner. The IdleState's insert() method needs to handle the transitions to the WinnerState. The class WinnerState is defined.

```
from abc import ABC, abstractmethod

class ToyMachine:
    def __init__(self, amt):
        self._winner = False
        self._qty = amt
        if self._qty > 0:
            self._state = IdleState(self)
        else:
            self._state = OutOfStockState(self)

    @property
    def winner(self):
        return self._winner

    @winner.setter
    def winner(self, boolValue):
        self._winner = boolValue

    # other methods no change

    import random

    class IdleState(TMState):
        def __init__(self, context):
            self._context = context

        def insertCoin(self):
            print("You inserted a coin.")
            if random.randrange(1,10)==1:
```

```

        self._context.winner = True
    if self._context.winner == False:
        self._context._state =
HasCoinState(self._context)
        elif (self._context.winner == True) and
(self._context.qty>=2):
            self._context.state =
WinnerState(self._context)
            return self._context.state

# other methods no change

class WinnerState(TMState):
    def __init__(self, context):
        self._context = context

    def insertCoin(self):
        print("You can't insert another coin!")
        return self._context.state

    def ejectCoin(self):
        self._context.winner = False
        self._context._state =
IdleState(self._context)
        return self._context.state

    def turnCrank(self):
        print("in Winner state")
        self._context.winner = False
        if self._context.qty==0:
            self.context._state =
OutOfStockState(self._context)
            self._context.dispense(2)
            print("2 Toys dispensed. Enjoy!")
        elif self._context.qty>0:
            self.context._state =
IdleState(self._context)
            self.context.dispense(2)
            print("2 Toys dispensed. Enjoy!")
        return self._context.state

    def restock(self, amount):
        print("You can't restock while a customer
is"\ \
        " buying a toy.")
        return self._context.state

# No changes to other classes

```

## Activity 6.3

We need to declare a class variable, `_tempList` to store all the temperatures. In the method `update()`, we append the new temperature to the `_tempList`, find the ave, max and min from the `_tempList` and prints them out.

```
class StatisticsDisplay(Observer):

    _tempList = []

    def __init__(self, weatherData):
        self._weatherData = weatherData
        self._weatherData.registerObserver(self)

    def update(self, subject):
        StatisticsDisplay._tempList.append(subject.temp)
        ave = sum(StatisticsDisplay._tempList) /
              len(StatisticsDisplay._tempList)
        print("Statistics Display has Average Temperature
{:.2f} \\
            " Max {:.2f} Min {:.2f}".format(ave,
max(StatisticsDisplay._tempList),
min(StatisticsDisplay._tempList)))
```

Test with the following code fragment:

```
s1 = ConcreteSubject()
o1 = CurrCondDisplay(s1)
o2 = StatisticsDisplay(s1)
s1.setMeasurements(26.6, 65, 30.4)
s1.setMeasurements(27.7, 70, 29.6)
s1.setMeasurements(25.5, 90, 31.2)
s1.removeObserver(o2)
s1.setMeasurements(27.4, 80, 32.25)
```

The output is:

```
Current Condition Display has Temperature 26.6 Humidity 65
Pressure 30.4
Statistics Display has Average Temperature 26.60 Max 26.60 Min
26.60
Current Condition Display has Temperature 27.7 Humidity 70
Pressure 29.6
Statistics Display has Average Temperature 27.15 Max 27.70 Min
26.60
```

```

Current Condition Display has Temperature 25.5 Humidity 90
Pressure 31.2
Statistics Display has Average Temperature 26.60 Max 27.70 Min
25.50
Current Condition Display has Temperature 27.4 Humidity 80
Pressure 32.25

```

Note that each setMeasurements() call results in each observer executing its update() method but when the statistics display was removed, it did not display any output.

## Activity 6.4

```

class Duck:
    def __init__(self):
        self._name = "duck"

    @property
    def name(self):
        return self._name

    def quack(self):
        return 'Quack quack'

    def fly(self):
        return "I'm flying!"

class Turkey:
    def __init__(self):
        self._name = "turkey"

    @property
    def name(self):
        return self._name

    def gobble(self):
        return 'Gobble gobble'

    def fly(self):
        return "I'm flying a short distance."

class Adapter:
    """
        This changes the generic method name to individualized
        method names
    """
    def __init__(self, object, **adapted_method):

```

```

    """Change the name of the method"""
        self._object = object

"""

Add a new dictionary item that establishes
the
mapping between
the generic method name: speak() and the
concrete method
For example, fly() will be translated to
gobble() if the
mapping says so
"""
        self.__dict__.update(adapted_method)

    def __getattr__(self, attr):
        """Simply return the rest of attributes!"""
        return getattr(self._object, attr)

```

The output (using the code fragment given in the question) is:

```

duck says 'I'm flying!'
turkey says 'I'm flying a short distance.'

```

## Activity 6.5

```

class Soy(CondimentDecorator):
    def __init__(self, beverage):
        self._beverage = beverage
        self._description = self._beverage.description+",
    +"Soy"
        self._cost = 0.15 + beverage.cost

    @property
    def description(self):
        return self._description

    @property
    def cost(self):
        return self._cost

    def __str__(self):
        return self._description

class Whip(CondimentDecorator):
    def __init__(self, beverage):

```

```

        self._beverage = beverage
        self._description = self._beverage.description+",",
    +"Whip"
        self._cost = 0.15 + beverage.cost

@property
def description(self):
    return self._description

@property
def cost(self):
    return self._cost

def __str__(self):
    return self._description

# Test:
drink2 = HouseBlend()
drink2 = Mocha(drink2)
drink2 = Mocha(drink2)
drink2 = Whip(drink2)
print('{} : ${:.2f}'.format(drink2, drink2.cost))
drink3 = HouseBlend()
drink3 = Soy(drink3)
drink3 = Mocha(drink3)
drink3 = Whip(drink3)
print('{} : ${:.2f}'.format(drink3, drink3.cost))

```

The output is:

```

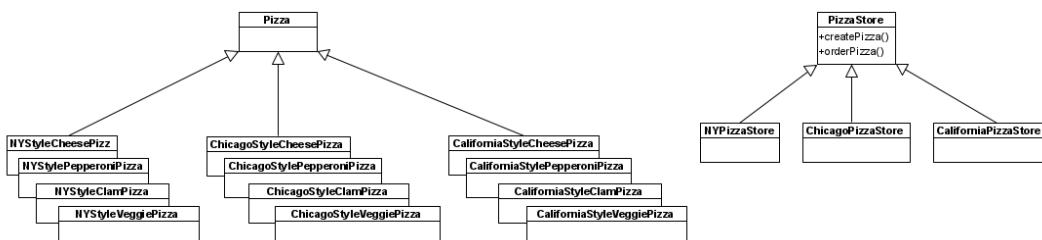
House Blend Coffee, Mocha, Mocha, Whip : $1.44
House Blend Coffee, Soy, Mocha, Whip : $1.39

```

## Activity 6.6

Require a new California Pizza Store subclass of the PizzaStore.

Require to have several California style pizzas subclasses of the Pizza class.



## Activity 6.7

A normal ChocolateBoiler class is defined:

```
class ChocolateBoiler:

    def __init__(self):
        self._empty = True
        self._boiled = False

    def fill(self):
        if self._empty:
            print("\tboiler is filled")
            self._empty = False
            self._boiled = False
        else:
            print("\tcannot fill - boiler is not empty")

    def drain(self):
        if not(self._empty) and self._boiled:
            print("\tdrain the boiled milk and choc")
            self._empty = True
        else:
            print("\tcannot drain - boiler is empty or not
boiled")

    def boil(self):
        if not(self._empty) and not(self._boiled):
            print("\tbring the contents to a boil")
            self._boiled = True
        else:
            print("\tcannot boil - boiler is not empty or
not boiled")
```

A Singleton class is defined that creates a global class ChocolateBoiler.

```
class Singleton(ChocolateBoiler):
    __instance = None

    @classmethod
    def getInstance(cls):
        if not cls.__instance:
            cls.__instance = ChocolateBoiler()
        return cls.__instance
```

## Formative Assessment

1. An object is an instance of a class. What is a concrete instance when working with design patterns?
  - a. A concrete instance refers to any occurrence of objects that exist during the running of code.

**Correct. This concept is used throughout Study Unit 6.**
  - b. A concrete instance refers to dynamic object values that are always changing.

Incorrect. A concrete instance is not a value of an object; it is the actual object. This concept is used throughout Study Unit 6.
  - c. A concrete instance refers to a fixed object value that cannot be changed.

Incorrect. A concrete instance is not a value of an object, it is the actual object. This concept is used throughout Study Unit 6.
  - d. A concrete instance refers to the behaviour or state of an object.

Incorrect. A concrete instance is not just the behaviour or state of an object. It is the actual object itself. This concept is used throughout Study Unit 6.
2. What group of design patterns describes how inheritance can be used to provide additional functionality?
  - a. Structural

**Correct. These design patterns use subclassing to provide additional functionality. Refer to Study Unit 6, Chapter 3.**
  - b. Creational

Incorrect. Creational design patterns provide an interface for objects to be created. Refer to Study Unit 6, Chapter 4.
  - c. Behavioural

Incorrect. Behavioural design patterns deal with algorithms and how objects are related to each other. Refer to Study Unit 6, Chapter 2.

- d. Definitional

Incorrect. Definitional design patterns do not exist.

3. How are algorithms defined in the strategy pattern?

- a. The strategy pattern defines mathematics operations as algorithms.

Incorrect. Algorithms can be any way to do something and may or may not include mathematics operations. Refer to Study Unit 6, Section 2.1.

- b. **The strategy pattern defines a family of algorithms, encapsulates each one and makes them interchangeable.**

**Correct. Refer to Study Unit 6, Section 2.1.**

- c. The strategy pattern defines one instance of a class with algorithms.

Incorrect. This defines the Singleton pattern. Refer to Study Unit 6, Section 3.2.

- d. The strategy pattern defines tough programming tasks as algorithms.

Incorrect. The algorithms in the strategy pattern can be any task/process. Refer to Study Unit 6, Section 3.2.

4. What is a state pattern?

- a. It allows object to be notified when state changes.

Incorrect. This is an observer pattern. Refer to Study Unit 6, Section 2.3.1.

- b. It allows subclasses to decide how to implement steps of an algorithm.

Incorrect. This is a strategy pattern. Refer to Study Unit 6, Section 2.1.1.

- c. **It encapsulates state-based behaviours and uses delegation to switch between behaviours.**

**Correct. Refer to Study Unit 6, Section 2.2.1.**

- d. It encapsulates interchangeable behaviours and uses delegation to decide which one to use.

Incorrect. The behaviours are state-based and the pattern uses delegation to switch between the behaviours. Refer to Study Unit 6, Section 2.2.1.

5. The Observer pattern has two components: subscriber and publisher. Which type of relationship exists with the subscriber and publisher?

- a. It is a one-to-many relationship. There is one publisher to many subscribers.

Incorrect. Yes, it is a one-to-many relationship, but the relationship is loosely coupled. So d. is a better answer. Refer to Study Unit 6, Section 2.3.

- b. It is a many-to-one relationship. There are many publishers to one subscriber.

Incorrect. There is one publisher and many subscribers. Refer to Study Unit 6, Section 2.3.

- c. Publishers send messages and subscribers receive the messages.

Incorrect. Publishers do not send messages directly to the subscribers. Refer to Study Unit 6, Section 2.3.

- d. Publishers post messages to a message broker, and subscribers register subscriptions to the message broker.

**Correct. There is loose coupling between the subscriber and publisher through the message broker. Refer to Study Unit 6, Section 2.3.**

6. Why is the Open/Closed principle important to the Decorator pattern?

- a. The Open/Closed principle explains that Decorator classes should have conditional statements to switch between open and closed.

Incorrect. The Open/Closed principle does not refer to whether a class is open or closed. Refer to Study Unit 6, Section 3.2.3.

- b. **The Open/Closed principle explains that Decorator classes should be open for extension, but closed for modification.**

**Correct. In the Decorator pattern, a class is closed for modification, but the class can be composed into another class which makes it open for extension. Refer to Study Unit 6, Section 3.2.**

- c. The Open/Closed principle explains that Decorator classes need to be tightly coupled to improve performance.

Incorrect. The Open/Closed principle does not refer to whether the classes are tightly or loosely coupled. Refer to Study Unit 6, Section 3.2.3.

- d. The Open/Closed Principle means that Decorator classes can be switched to open or closed classes.

Incorrect. The Open/Closed principle does not refer to whether a class is open or closed. Refer to Study Unit 6, Section 3.2.3.

7. How does Factory pattern provide loose coupling and high cohesion?

- a. The Factory pattern is a creational pattern that is flexible and hence uses loose coupling.

Incorrect. Flexibility does not mean loose coupling is used. Refer to Study Unit 6, Section 3.1.1.

- b. The Factory pattern divides classes into categories and uses abstract methods.

Incorrect. Factory subclasses decide on the type of Product objects to create. Refer to Study Unit 6, Section 3.1.1.

- c. The Factory pattern uses configuration files instead of coding to create loose coupling.

Incorrect. Configuration files are not used. Refer to Study Unit 6, Section 3.1.3.

- d. **The Factory pattern encapsulates object creation logic which makes it easy to change later.**

**Correct. The Factory pattern provides an interface as a superclass to create objects. The subclasses can decide what type of objects to be coupled with at run time hence the loose coupling. Refer to Study Unit 6, Section 3.1.3.**

## References

Freeman, E., & Robson, E. (2020). *Head first design patterns* (2nd ed.). Sebastopol: O'Reilly Media, Incorporated.

