MT201

Unit 2

Problem solving through objects

Course team

Developer:	Herbert Shiu, Consultant
Designer:	Dr Rex G Sharman, OUHK
Coordinator:	Kelvin Lee, OUHK
Member:	Dr Reggie Kwan, OUHK

External Course Assessor

Professor Jimmy Lee, Chinese University of Hong Kong

Production

ETPU Publishing Team

The Open University of Hong Kong
30 Good Shepherd Street
Ho Man Tin, Kowloon
Hong Kong

Contents

Introduction

In *Unit 1* you learned that hardware and software must cooperate to enable a computer system to perform the desired operations. This unit concentrates on discussing software development in the object-oriented paradigm.

Software development is creative and interesting; you will have a sense of success when you see a computer performing operations according to your programming instructions. From *Unit 3* onwards, we discuss the ways to write small and subsequently more complicated programs in the Java programming language. Beforehand, we need to give you an overview of the ways to build software.

In the early days of software development, because of the limited capabilities of the computer hardware, programs could only be small and simple (compared to the current ones). In line with improvements in computer hardware that has faster processors and larger memory space, software developers can now build more and more sophisticated software applications. Software development requires team effort. Starting to write programs immediately just sitting down at a computer is not recommended if you want to build non-trivial software applications. There needs to be a systematic way to analyse and design the software to be built and to manage every step in the software's development.

This unit discusses a common practice in building software. This practice covers the lifespan of the software's development, known as the *software development cycle*. Furthermore, we discuss how to analyse problems and design the software using the object-oriented approach along with the benefits of such an approach. The discussion is language independent and applicable to any object-oriented programming language, including Java of course.

In the past, you might have thought that knowledge of writing programs and software development was something like topics in philosophy, which are extremely abstract and that only Einstein-like people can learn. In this unit, we introduce a systematic way to analyse a problem and design the software accordingly. Even though the programs developed in this unit are not complete — they are just program segments — they give you skeletons of the final products. At appropriate points in later units, we will make them complete and you'll see the whole picture.

Objectives

At the end of the *Unit 2*, you should be able to:

1 *Describe* the software development cycle.

2 *Describe* classes and objects and their relationships.

3 *Describe* and *apply* object-oriented analysis and design.

4 *Describe* the inheritance between a general class and specific classes.

The software development cycle

Since the introduction of programming languages, programmers have been developing software applications to solve various problems. These software applications are getting more complicated and involve more programmers. As a result, a systematic approach has evolved from these programming experiences. This approach is known as the software development cycle.

Why we need the software development cycle

When you switch on your computer, it starts loading the operating system, probably a version of Microsoft Windows. Then, you might start a word processor or a spreadsheet application. How many software developers do you imagine were involved in building these kinds of software application?

According to the officials that headed the software development project of Microsoft Windows 2000, 2,500 core developers were involved. You could not expect that 2,500 developers simply switched on their computers and started developing the software. To coordinate so many people on a single project is a difficult task on its own, not to mention the millions of lines of 'bug-free' codes that had to be produced at the end of the project.

Another problem in developing software is that even if skilful developers are employed to write the software, the finished product might not be what the users expected. Miscommunication can easily happen between the developers and the users.

To avoid the problems related to software development, including those mentioned above, well-defined steps are needed in handling and managing development. Such a sequence of steps is known as the software development cycle. Your understanding of the cycle can help you understand the activities in building complex software applications and the problems that need to be addressed throughout the process.

Phases

Since the first computer was rolled out over 50 years ago, people have accumulated considerable experience in developing software. They found that software development usually involves some common activities carried out in a particular sequence. Such a pattern was studied and evaluated by various software developers and became a recommended approach for software development.

According to the nature of those common activities undertaken in the lifespan of the software development, the software development cycle can be divided into several phases. They are the:

- requirements phase

- analysis phase
- design phase
- implementation phase
- testing phase
- maintenance phase.

As its name suggests, the software development cycle is cyclical, not linear. The cycle is also called the 'waterfall model' (seen in Figure 2.3 later).

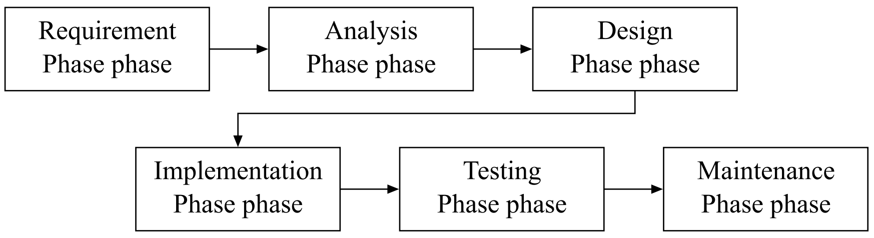


Figure 2.1 The software development cycle (simplified)

Sophisticated software might be too complicated to build with a single pass through the above six phases. A single pass might only complete a portion of the features (or a software module) of the system. For example, a single pass might complete the software module for manipulating an email system. Another pass would be required to complete the graphical user interface module for that system.

Some software systems are released to the users in stages with additional features in each stage. For example, a banking system is required to support *internal* bank account transferral in the first stage. The system to be released in the second stage is required to support *inter-bank* account transferral, and the system of the final stage supports a *Web-based* banking system.

The above scenarios encourage an iterative approach of software development. The entire software development is divided into iterations. Each iteration is targeted at a portion of the system and *each iteration consists of requirement, analysis, design, implementation and testing phases* (see Figure 2.2). After an iteration, another iteration is started until the whole system is completed, leading to the maintenance phase.

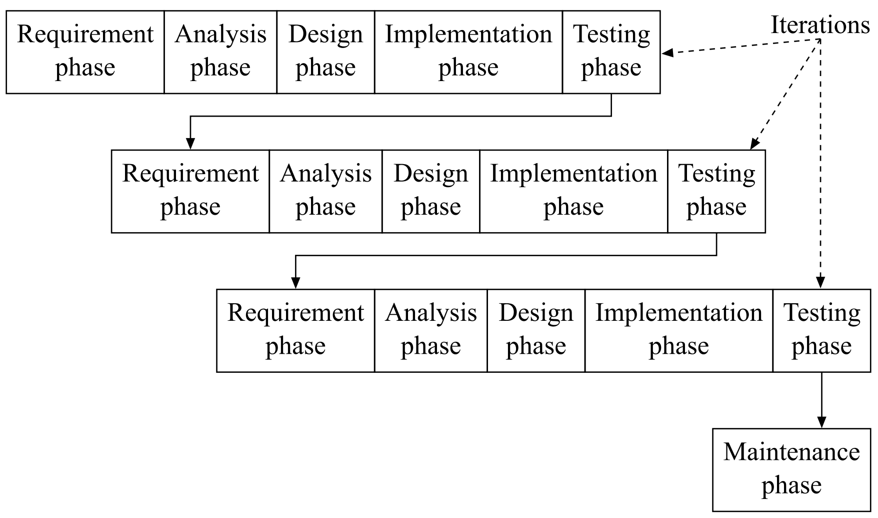


Figure 2.2 The iterative approach to software development

For a simple software system, a single iteration may suffice. A sophisticated software system may require several or tens of iterations. The number of iterations required varies. It depends on the complexity of the system and the decisions of the developers.

Different phases require different developer capabilities, and some developers may be more skilful in the tasks of some phases. The above iterative approach enables the development to be carried out in a pipeline fashion. For example, while some developers are working on the testing phase of an iteration, another group of developers has already started working on the requirement phase for the next iteration. As a result, the overall pace of the software development can be improved.

After all iterations, the software system can be released to the users. The software project then enters the maintenance phase.

The following subsections discuss the tasks to be undertaken in each phase.

Requirements phase

Before you start building a software system, and especially when it is for other people, you have to thoroughly understand the problems to be solved. Error-free but wrong software (not satisfying users' needs) is useless.

The first phase in software development is the requirement phase, which aims at collecting the requirements (needs) from the users or clients who asked you to develop the software. The requirements may include limitations of the software system.

The needs refer to the functionalities of the software to be built. For example, with respect to a payroll software system, the requirements can be what types of staff are to be supported, what types of payment method are required and what types of report must be provided.

The limitations are the constraints of or on the software, such as the budget, hardware, and time limits. For example, the users may require any single operation to take less than one second to complete, and the software has to be completed in three months with a budget of less than HK$200,000.

The action to be taken in the phase includes interviewing the clients who requested the software and the users who will use the software. Furthermore, if the users are currently using a software system that needs enhancement or replacement, it is necessary to find the deficiencies of the existing software.

A common problem that occurs in this phase is miscommunication between the clients/users and the software developers. Users typically use their own business terminology or even jargon and usually cannot clearly express their requirements to the developers. Developers who know how to build software may have little understanding of the business involved. From time to time, the developers have to seek clarification from the users.

In the requirement phase, developers usually prepare a document known as a *problem statement* that summarizes the requirements of the clients, including the limitations of the software. It can help both the users and the developers clarify the needs and limitations early in the process.

Analysis phase

After the requirement phase, you have the problem statement that describes the problem or scenario in detail. Based on this document, it is possible to identify the functionalities and features of the software to be built.

In the analysis phase, software developers further investigate the problem statement and determine the following:

1. A list of people (roles) who will use the software. For example, the users who will use a banking system are categorized into customers, tellers, supervisors and managers.

2. A list of functionalities or operations that the software supports and which operations are performed by which roles. For example, in a banking system, deposit, withdraw and transfer functions are account operations. Tellers are allowed to perform all these operations, whereas supervisors can modify transactions made by tellers.

3. A list of operational scenarios of an operation under different conditions. For example, if the customer's bank account has sufficient money, the withdrawal operation is allowed. Otherwise, the withdrawal operation fails and the software should alert the user.

4. A list of business entities involved and their attributes and behaviours. For example, a transferral operation involves two bank

accounts and a transaction record. Each bank account has an attribute, ‘balance’, and two operations, ‘deposit’ and ‘withdraw’.

The above lists can help you explore the details of the software required. If you cannot complete these lists, you have not collected enough information from the users. You should repeat the requirement phase and re-interview the users (or use other means) to obtain the missing information.

At the end of the phase, the developers should be thoroughly clear about what the users expect the software to be and do and clear about the details of all functionalities.

In the case studies provided in this unit, you’ll see how you can derive the roles, functionalities, operational scenarios and the business entities involved.

Design phase

Up to the analysis phase, the software developers examine the software from the business point of view. In the design phase, the developers transform the requirements into technical designs.

The first two phases (requirement and analysis) are *programming-language independent* because the aim of these two phases is to collect the requirements of the system and to understand the operational details. Once you thoroughly understand the requirements of the software system, you can develop the system using any programming language you choose.

Since the object-oriented approach has many advantages, many software development projects use it. Using this approach, software developers try to model real-world scenarios and system operations by objects.

Objects of the same type, and entities with the same set of behaviours and attributes, are defined with a single *class* definition (the relationship between classes and objects is discussed very soon). In the design phase, the class definitions are designed so that they can be implemented in the implementation phase.

Based on the result of the analysis phase, programmers can determine the attributes of the objects. Furthermore, they know that there might be many scenarios for a single operation. They need to consolidate these scenarios and determine a well-defined sequence of steps of an operation or behaviour to be executed under different scenarios.

At the end of the design phase is a list of classes. Each class has a list of attributes and behaviours. Furthermore, there are well-defined steps for each object’s behaviour.

Implementation phase

The implementation phase aims at converting the class designs derived in the design phase to actual class definitions in the chosen programming language. The Java class definitions of the examples and case studies in this unit are provided to illustrate the way to represent conceptual class designs in the language. They are neither complete nor can they be compiled. From *Unit 3* onwards, we discuss the ways to implement the classes, their attributes and behaviours using Java.

Testing phase

The requirements obtained in the requirement phase have been analysed in the analysis phase and converted into programs (or sets of class definitions) after the design and implementation phases. Then, you have to make sure that the programs do what you expected. This is the main purpose of the testing phase.

In the testing phase, software developers usually verify the programs by feeding test data to them and checking whether the outputs are exactly the same as the expected results. If they are not the same, the programs must be scrutinized and checked for errors.

After the errors are found and corrected, the programs are re-tested. Such a process is repeated until all likely errors are corrected. After the testing phase, the software system is ready to be released to the users for real operations.

Unit 4 discusses the approaches to uncover errors in programs. You'll gain experience in pinpointing errors as you write more programs. Note that not all errors can be found and corrected if the system is large and sophisticated. That is one reason that you need the maintenance phase to fix the errors found after the software has been released.

Maintenance phase

Even if the software system is ready and the users can use it for real operations, it is not the end of the story. The software system might take six months to complete, but the users are going to use it for the next few years. During this period, the users might uncover some problems that could not detected before the software system was released. Furthermore, the world changes rapidly and the software system may need to be updated or enhanced according to the changes in business operations. Therefore, software developers are required to maintain the software system so that users can keep on using it.

Maintaining a software system usually involves modifications of the source code. It can be a nightmare and take a lot of effort if it was not properly designed and implemented. Therefore, the maintenance cost throughout the whole period can be even greater than the sum of the first five phases. This is the reason you need a systematic way to build software systems. As you gain experience in developing software, you will know how and what to modify to fit the user's requirements.

When the expected output of a phase does not meet the predetermined specifications, it might be necessary to perform some of the tasks in the early phases again — remember that the software development cycle is *cyclical*. For example, while you design the software system in the design phase, you may find that some information is missing and you cannot complete a class definition. When this happens, it is necessary to perform the tasks in the requirement phase again to collect the missing information. After the necessary information is collected, you can continue the analysis phase and the design phase.

A more complete diagram

Figure 2.3 shows the software development cycle in a more complete form. After the tasks of a phase are completed, the tasks of the next phase will be started. Otherwise, the remaining tasks of the same phase should be done. These flows are shown through solid lines. For example, after meeting the needs of the analysis phase, the design phase can begin. The model also allows going back to a previous phase if something needs to be done at that phase. These are shown mainly through dotted lines. For example, if some ambiguities of user requirements are found in the implementation phase, you need to go back to the requirement phase to ask the users.

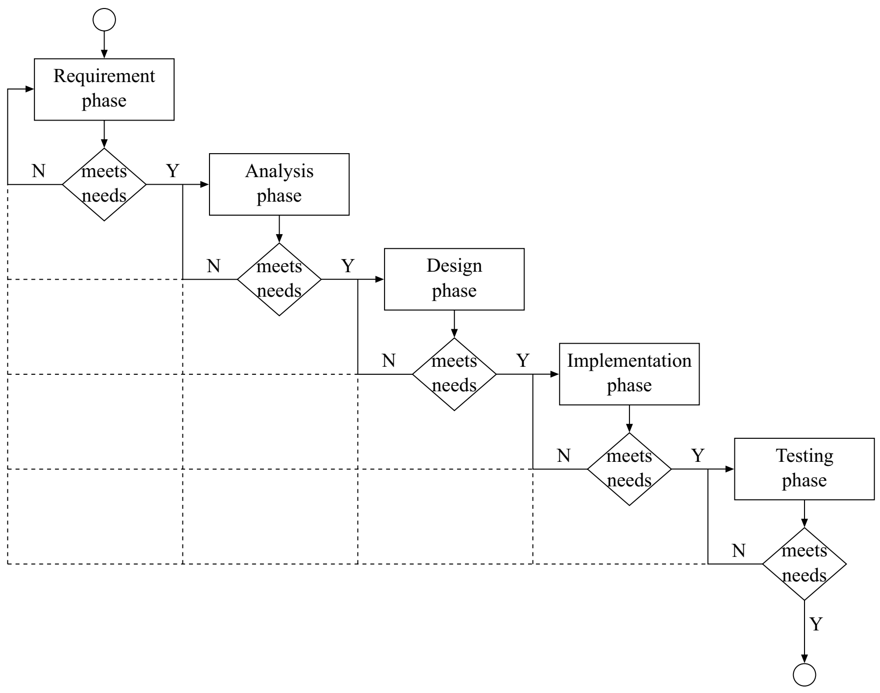


Figure 2.3 A more complete representation of the software development cycle

What are objects?

Software developers get user requirements in the first phase of the software development cycle. The tasks involved in this phase are usually implementation independent. For example, if you have the requirements of a flight reservation system, you can then build a flight reservation software system using any programming language and approach you choose. Of course, the nature of the software system can affect your choice of programming language and the development approach.

Software developers use the object-oriented approach to analyse the problem using object models and design the software system with object-oriented methodologies. Consequently, they implement the design using an object-oriented programming language.

Very soon in the unit, we use the object-oriented approach to analyse and model the scenarios in the case studies and design the software system. *Unit 3* discusses how to implement the software design using our chosen object-oriented programming language, Java.

The object-oriented approach is based on classes and objects. Therefore, you must understand classes and objects — and the relationships among them.

The first question is: ‘What are objects’? Our world is made up of objects — a chair, a television set, a computer — everything around us is an object. A desktop computer and a notebook computer are similar in the sense that they both have a keyboard, a display unit, a hard disk and so on; they are characterized by attributes such as memory size, processor type and speed, and hard disk storage. Furthermore, both can accept keystrokes and mouse inputs, and both can generate the results based on the input data.

Of course, you probably know that a desktop computer and a notebook computer are in the same category, ‘computer’. In the object-oriented paradigm, we call such a category a *class*. From this perspective, both computers are instances (or objects) of the `Computer` class.

Real-world objects vs computer objects

Object-oriented software tries to model real-world objects using software objects. If it can achieve this, it can easily manipulate the software objects (as if they were real-world objects) and get the result you want. Since real-world objects have their own characteristics (e.g. the colour of a ball is red), software objects have attributes to mimic the characteristics. As real-world objects can perform different actions or actions that might be done to them (e.g. *hit* the red ball), software objects can have behaviours that correspond to the actions.

Apart from real-world *physical* objects, some *abstract* concepts or ideas, such as a bank account, can be modelled as objects. You can have bank statements or a passbook for the account, but you cannot physically touch a bank account. It is not a physical object, but it may have characteristics such as its ‘balance’, and actions can be done on it such as ‘credit’ and ‘debit’.

You develop software to mimic the objects so that you can manipulate them to solve your problems. When the developed software is executed, computer objects are created. They have attributes and perform appropriate operations when they receive messages (from other objects).

A real-world object might have many characteristics and actions. However, computer objects model only a subset of those that are related to the specific problem. Therefore, the same real-world object might be modelled differently in different software.

Another difference between real-world objects and computer objects is that computer objects only exist while the software is executing. The object states need to be stored properly so that they can be recreated the next time you execute the software.

Object concepts: attribute, state, message and behaviour

Suppose you want to model a washing machine using software. First, you need to collect the information about a washing machine in the requirement phase. Afterwards, in the analysis phase, you find that a washing machine is usually characterized by its brand name, model number, capacity, the time required to wash a load of clothing, power consumption rate and its dimensions.

For example, you are going to model a washing machine with the following information:

Brand name: ABC
Model number: WM-201
Capacity: 5 litres
Time required: 45 minutes
Power consumption rate: 50 watts
Dimensions: 880 mm (height), 530 mm (width), 530 mm (depth)

The above list is a list of washing machine properties (and their values). In the object-oriented paradigm, the properties and their values are the *attributes* and the *states* of an object. The states reflect the current statuses or situations of the object. It is not necessary to update the values for the above attribute list of a washing machine object. However, if you consider the properties ‘clothing weight in drum’ and ‘working status’ of a washing machine, these two properties can change over time. For example, if the washing machine is not used, the clothing weight is ‘zero’ and its working status is ‘idle’. If you put some clothes in it, the clothing weight changes. When you press the ‘Start Washing’ button, it starts working; its ‘working status’ is changed to ‘working’.

You can see that objects always have attributes, and their values (or states) may change over time, especially if they are manipulated in any way.

Furthermore, a washing machine is designed to wash the clothing in its drum. Therefore, a washing machine object has a *behaviour* — ‘wash’. However, your washing machine will never start by itself — you have to press the ‘Start Washing’ button on the panel. Similarly, an object in a software system will never perform its behaviour unless you send a *message* to it. In software systems, a message is sent to an object to request it to perform its behaviour, which is like you pressing the ‘Start Washing’ button.

Another important observation is that if you have two washing machines in your home, and if you press the ‘Start Washing’ button of washing machine A, washing machine B will never start washing — only washing machine A will. This is an obvious observation in the real world. Similarly, with respect to a software system in which two washing machine objects exist, sending a ‘start’ message to the first washing machine object should only start the first one, not the second one.

An object usually has more than one behaviour. For example, a washing machine can stop washing at any time if you press the ‘Stop Washing’ button on the panel. Similarly, you use a message type or name to request an object to perform different behaviours. If a washing machine object is requested to start washing, a message of type ‘start’ is sent to it; a message of ‘stop’ is sent to it to request it to stop.

There is a one-to-one mapping between the message types and object behaviours. If an object has a behaviour such as ‘doSomething’, it is expected that it can accept a message type ‘doSomething’ and perform the ‘doSomething’ behaviour.

Some behaviours need supplementary data so that they can perform accordingly. For example, assume your washing machine allows you to set a timer so that it can start automatically. When you set the timer, you need to specify the amount of time. Then, the message type is ‘setTimer’ and a value of 10 (for 10 hours) is sent to the washing machine so that it sets the timer accordingly.

Sometimes, an object’s behaviour may give you some feedback. For example, there is a display or indicator on the washing machine panel showing the time elapsed. With the software washing machine object, it is necessary to send a message, saying ‘getTimeElapsed’, to it and it returns the time elapsed on its display panel.

The attributes and behaviours of a washing machine object are identified in the analysis phase. In the design phase, you can design a class definition that describes (or models) washing machine objects, because a class definition is used as a blueprint for all objects of the class. Such a class definition is shown in Figure 2.4.

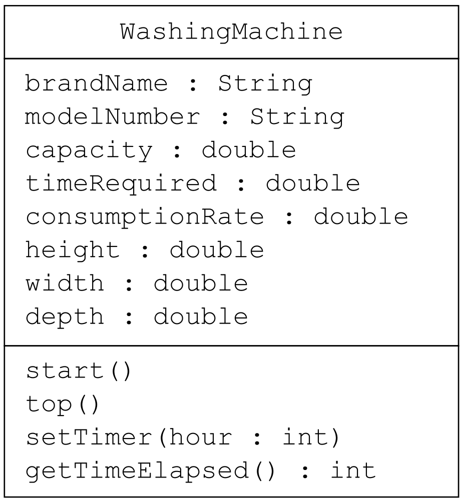


Figure 2.4 A class definition for a washing machine

The class diagram in Figure 2.4 is divided into three regions. The top region shows the name of the class (or category) of the objects. The middle and bottom regions show the attributes and behaviours that each object of the class has. As there are four behaviours — `start`, `stop`, `setTimer` and `getTimeElapsed` — a `WashingMachine` object can accept message types `start`, `stop`, `setTimer` and `getTimeElapsed`. The parentheses that follow the message name (or behaviour name) shows whether supplementary data is required. The behaviours `start`, `stop` and `getTimeElapsed` do not require supplementary data, whereas the `setTimer` behaviour needs supplementary data in the form of 'hour(s)'.

The words `String`, `double` and `int` are used in the diagram (Figure 2.4) to specify the data types of the attributes, supplementary data of messages and the feedback from a behaviour. The '`: int`' after the `getTimeElapsed` behaviour indicates that such a behaviour returns feedback of type `int`. Data types are discussed in detail in *Unit 3*.

You can now visualize the washing machine mentioned earlier, as shown in Figure 2.5:

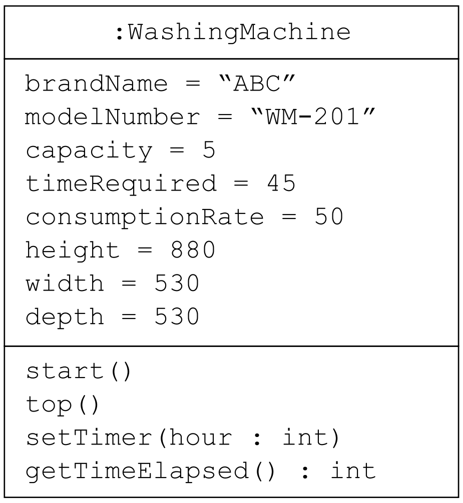


Figure 2.5 Definition of a washing machine object

The first region denotes the name and the class name of the object. For example, it is a `WashingMachine` object named `shared` and you can use the following format to specify it:

```
shared:WashingMachine
```

The specific name for an object is usually optional, and the following format is therefore used to represent an anonymous `WashingMachine` object:

```
:WashingMachine
```

The second region shows the states (current property values) of the object, and the last region identifies the behaviours that the object has.

Based on the class design, it is then possible to write the class definition with the Java programming language in the implementation phase.

Accomplishing tasks by objects

In object-oriented programming, tasks always involve objects. To complete a task, you expect some results will be obtained from the objects or that the states of some objects will be changed. For example, when you press the calculator buttons to enter a sequence of numbers, you expect it to give you the sum of all of the numbers. Similarly, in a bank account transferral operation, you expect that the balances (one of the states of bank accounts) of the source account and destination account will be decreased and increased respectively. You are interested in how those objects behave during the whole process.

The following section discusses two scenarios. A task is completed for each of them. The software development cycle is used as a guideline and gives you an overview of how conceptual scenarios can be modelled with objects and what you can get after each phase. We further elaborate on the skills in each phase in later sections.

Example: transferring money between bank accounts

Transferring money from one account to another is a basic operation of a banking system. Suppose you have obtained the following description in the requirement phase:

> The bank system transfers an amount of money from a source account to a destination account. It first withdraws the amount from the source account and then deposits the same amount to the destination account.

Then you analyse the above description in the analysis phase and determine that the scenario involves three objects. One is the `Bank` object; the other two are `BankAccount` objects. The objects cooperate with each other to accomplish the transfer task as follows.

- The user of the system sends a message `transfer` to the `Bank` object with information `sourceAccount`, `destinationAccount` and the `amount`.
- The `Bank` object subsequently sends a `withdraw` message with the amount of money to the `sourceAccount` object.
- Then, the `Bank` object sends a `deposit` message with the same amount of money to the `destinationAccount` object.
- After some messages have been sent and some action taken by some objects, the task of money transfer has been completed by the objects cooperating with each other.

Example: using a washing machine

Suppose that you are a washing machine manufacturer. Your factory produces a series of computerized washing machines with different capacities. To design the program for washing the clothes when the 'Start Washing' button is pressed, you collect the requirements of such an operations as:

1. The 'Start Washing' button of the washing machine is pressed.
2. The incoming water valve is opened for 5 minutes.
3. The drum motor starts at normal speed for 15 minutes.
4. The outgoing water valve is opened.
5. The drum motor starts at high speed for 5 minutes.
6. The outgoing water valve is closed.
7. The incoming water valve is opened for 5 minutes.
8. The drum motor starts at normal speed for 15 minutes.
9. The outgoing water valve is opened.
10. The drum motor starts at high speed for 5 minutes.
11. The outgoing water valve is closed.

Steps 2 to 6 are dedicated to the washing process, and steps 7 to 11 are for rinsing the clothes. The above sequence has been simplified for illustration. In reality, a washing machine usually determines the required water level first according to the weight of the clothes. Furthermore, the incoming water valve is opened until the water reaches a particular level. For simplicity, the above description controls the amount of water based on time.

Suppose the objects `WashingMachine`, `DrumMotor`, `IncomingValve` and `OutgoingValve` have been found to be appropriate. The washing operation can be accomplished by messages sent among the objects. Figure 2.6 summarizes the steps. Let's see how the task can be done.

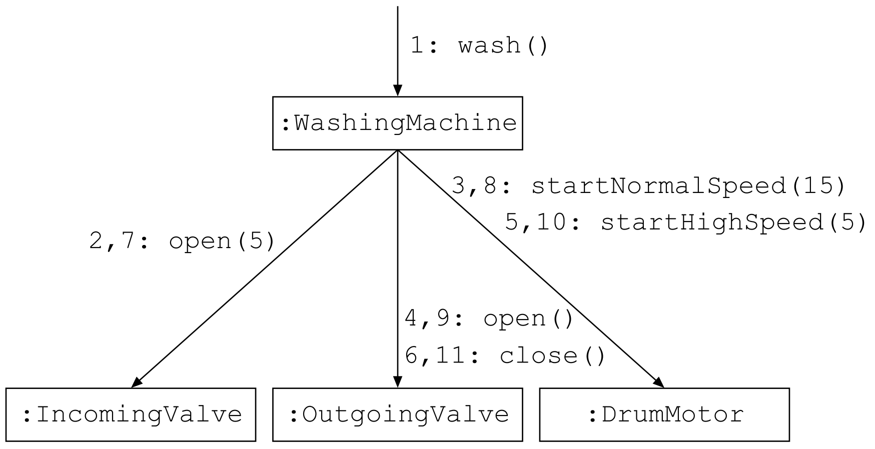


Figure 2.6 Modelling the washing machine objects

In Figure 2.6, the `WashingMachine` object receives a message `wash` (denoted by '`1: wash()`') if the user presses the 'Start Washing' button of the washing machine. The number '`1`' before `wash()` is the line number shown in the requirement. After receiving the `wash` message, the washing machine sends messages to other parts so that the washing operation can be performed. For example, to perform the action in line 2 of the requirement, the washing machine sends an `open` message to the `IncomingValve`. Since the `IncomingValve` must be open for a particular length of time, a supplementary value `5` is given with the message. You should expect to see `2: open(5)` in the figure. However, a slightly different version `2,7: open(5)` is given, which is a concise version of two messages `2: open(5)` and `7: open(5)`. After 5 minutes, the washing machine sends a `startNormalSpeed` message to the `DrumMotor` asking it to start operating at normal speed for 15 minutes. Afterwards, an `open` message is sent to the `OutgoingValve` from the `WashingMachine` (`4: open()`in the figure). Other operations are similar.

Now you can see how objects can do a task by exchanging messages with each other.

Classes and objects

From the previous section, you can see that an operation can be modelled based on sending messages among the objects that are involved in the operation. In the bank account transfer example, one bank object and two bank accounts are involved. Basically, these two bank accounts have the same sets of attributes and behaviours. Therefore, they are considered to be of the same type or class in the analysis.

The following section elaborates on the relationship between classes and objects.

The relationship between classes and objects

During the analysis of a problem, you’ll usually discover that it involves similar entities or objects that have the same sets of attributes and behaviours. Real-world human-made objects such as washing machines are manufactured in a factory according to their specifications. Similarly, software objects are created in computers according to their specifications, which include their attributes and behaviours. The specifications are called *class definitions*.

When an application is executed, the relevant class definitions are used to create objects so that the objects have the appropriate attributes and behaviours. For example, a bank account class can be used to create the two bank account objects that are involved in the bank account transferral example. Therefore, a class is the blueprint for a group of objects of the same category. It is pretty much like a cookie cutter that makes cookies that look alike. There is a single cookie cutter but it can produce a lot of similar cookies.

Interactions among objects

Each object has its set of behaviours, but these behaviours are not performed unless it receives a message. For example, the bank account object will never change its balance unless it receives a message to perform such a behaviour.

The messages that an object receives are usually sent from another object or objects. Whenever an object receives a message, it performs the behaviour according to the type of the message and the supplementary data. The behaviour can be updating its attributes or subsequently sending messages to other objects to trigger them to perform other operations.

The key observation is that the only interaction among objects is the messages sent among them. When the message sender sends a message to a message recipient, the sender knows that the recipient can interpret the message and perform a behaviour accordingly, based on its states. It is not the message sender's concern how the recipient object will perform. It is up to the recipient object to determine its own way of performing the operations when it receives messages.

A clear example can be seen in the pizza-ordering scenario discussed in *Unit 1*. When the customer service staff member of the restaurant receives a message from a customer about the details of the pizza, he or she sends a message to the chef in the kitchen, giving those details. It is unnecessary for the customer service staff to understand how the chef prepares the pizza. Afterwards, customer service sends a message to the delivery unit concerning the customer's address. He or she is not required to know what form of transportation the delivery staff will use. The customer service staff member is only sure that the delivery unit will deliver the pizza to the customer's home. All involved staff are required to perform their own operations to the best of their ability.

The benefit of such interaction among objects is that every object (the agent) is only required to perform its own operations based on the attributes and the supplementary data in the messages when it receives particular messages. Each object performs its own part. All objects cooperate, and they constitute a complex operation. It is much easier for analysts to determine how an individual object performs a single small operation than understanding every operational detail or the changes of all objects involved in a complex operation.

Deriving objects from a given problem

We have discussed how a task can be accomplished with objects sending messages to other objects. However, we have not yet discussed how those objects were identified and, in the washing machine example, how the diagram showing the interactions among the objects was derived.

In the requirement phase, you obtain clear statements about what the problem is — and the details are probably in plain English. Using the object-oriented approach, you then determine the objects (and hence the classes) that are involved in the problem in the analysis phase.

The following section discusses the steps in determining the objects and their attributes and behaviours from the previous examples. After such information is obtained, you can then design the classes and give an outline for the class definition implementation in Java. Also in this section are two case studies that are a little bit more complicated so that you can see how the techniques can be used generically.

More than one result might be derived from the problem, as the result depends on your understanding on the problem and tradeoffs you considered. Sometimes, it is hard to say which of the options is better. No matter what result you derive, you have to make sure that the derived objects can fully model all scenarios.

The steps in deriving objects

The problem statement obtained from the requirement phase is analysed in the analysis phase. The core activity in the analysis phase is to identify the objects involved, along with their attributes and behaviours.

In the following section, we discuss a systematic way to derive objects and their attributes and behaviours. We use the previous examples to demonstrate. Two more case studies are provided for further elaboration.

When you are given the problem statement (in plain English sentences about the problem to be solved) the subsequent steps are:

1 Determine the objects and hence the classes involved.

2 Determine the object attributes.

3 Determine the object behaviours.

Determining object attributes

The first step is to determine objects from the problem statement. The potential candidates can be obtained by highlighting (e.g. underlining) the *nouns* and *noun phrases* in the sentences. The list of nouns and noun phrases allows you to identify the entities or objects that must be modelled in your software application.

Some of the nouns are objects and some are object attributes. Others might refer to an object already identified, or they might not be useful in the model. If a noun represents an entity that exists in the problem, it is a candidate object to be modelled. Then, you should consider whether it is necessary to model such an object by a class definition. The decision is based on your understanding of the problem, so there is no concrete answer. Furthermore, you should group the nouns that refer to the same type of entity and give a name to it, which is a class name for the group of entities.

If a noun does not refer to an entity, it might be related to an object, and it is then an attribute of the related object (and hence the class). Otherwise, it might be supplementary information that a behaviour requires.

If you find a noun does not play any role in the problem or scenario, you can simply ignore it. After grouping and filtering the list of nouns and noun phrases, you should have a list of class names and their attributes.

Finally, you should determine whether an object comprises other identified objects or an object has other objects as its components. If so, the object has an attribute for each component object, and you can give a name to each one. Furthermore, if an object always sends messages to a few fixed objects, the corresponding class definition has an attribute for each of them.

Determining object behaviours

By identifying the *verbs* in the problem statement, you can obtain a list of potential object behaviours. You should verify whether two different verbs are actually referring to the same behaviour. If so, you should choose one of them to represent the two. Determine the object (and hence the class) that has the behaviours. Furthermore, you should notice that some verbs only specify that the objects have an attribute; they are not considered to be an object behaviour.

At the end of this process, you have a list of object behaviours that you can add to the related classes. You can model the scenarios presented in the problem statement by message-sending among objects. You can verify whether the object behaviours suffice to model the scenarios and whether the behaviours need supplementary information to perform the behaviour.

Finally, the list of class and their attributes and behaviours is obtained.

Transferring money between accounts

Now, let’s revisit the first bank account transferral problem. In the requirement phase, the problem statement was found to be:

> The bank system transfers an amount of money from a source account to a destination account. It first withdraws the amount from the source account and then deposits the same amount in the destination account.

Then, in the analysis phase, you should highlight the nouns or noun phrases in the problem statement first, as follows:

> The <u>bank system</u> transfers an <u>amount of money</u> from a <u>source account</u> to a <u>destination account</u>. It first withdraws the <u>amount</u> from the <u>source account</u> and then deposits the same <u>amount</u> in the <u>destination account</u>.

From the problem statement, the nouns are identified and sorted as follows:

amount
amount of money
bank system
destination account
source account

Then, you can prepare a table:

Objects or classes	**Class name**	**Remarks**
amount		Supplementary information of a behaviour
amount of money		Same as amount (duplicated)
bank system	`Bank`	
destination account	`BankAccount`	
source account	`BankAccount`	

For each noun listed in the first column, ‘Objects or classes’, you should determine whether it is a candidate object to be modelled by class definitions. If so, give a class name to each one (in the second column). You should choose class names to reflect their purpose. If the class names you choose contain more than one word, you can concatenate (join) them as a formal class name.

If the noun does not need a class definition, state the reasons why in the remarks column.

Both destination and source accounts are accounts. They belong to the same class and are modelled by the class `BankAccount`.

Then, you can highlight all verbs in the problem statement:

> The bank system ***transfers*** an amount of money from a source account to a destination account. It first ***withdraws*** the amount from the source account and then ***deposits*** the same amount in the destination account.

The verbs in the problem statement indicate that the corresponding behaviours are involved in the scenario. Therefore, you can further analyse which objects are responsible for the behaviours.

The problem statement clearly shows that the bank system (`Bank`) performs the `transfer` behaviour. For the `withdraw` and `deposit` behaviours, the sentence in the problem statement is in the format like 'the bank system withdraws the amount from the source account', which seems that the `withdraw` behaviour is performed by the bank system (`Bank`). However, if you consider which object is affected, it is the source account: After the `withdraw` behaviour is performed, the balance of the source account is decreased. We therefore consider that `withdraw` is a behaviour of the source account; the bank system (`Bank`) requests it to perform a `withdraw` behaviour to decrease its balance. By the same argument, the destination account has the `deposit` behaviour, and the bank system (`Bank`) requests it to perform the behaviour to increase its balance.

Based on the problem statement, the list of identified objects and their behaviours, you could draw a diagram (like Figure 2.7) showing the interactions among the objects:

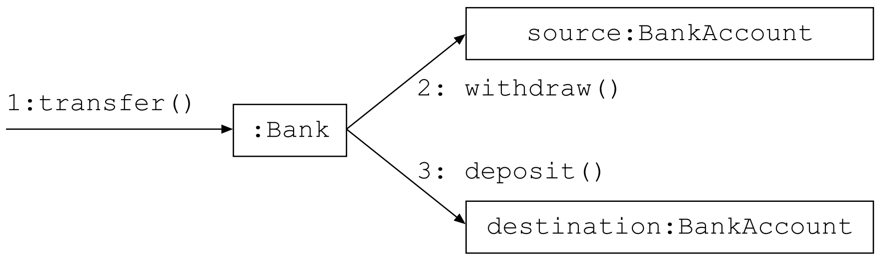


Figure 2.7 Interactions among objects in the bank account transferral problem

Furthermore, it is found that both the `withdraw` and `deposit` behaviours require the amount of money as supplementary data so that the account can perform the behaviour accordingly. Subsequently, the `Bank` object must provide such data to the source and destination accounts. Therefore, the `transfer` behaviour of the `Bank` object must get such data from the `transfer` message.

Furthermore, the `Bank` object sends `withdraw` and `deposit` messages to different bank account objects for different bank transferral operations. Therefore, the user of the system must specify the source account and destination account as well. Then, the diagram becomes as shown in Figure 2.8:

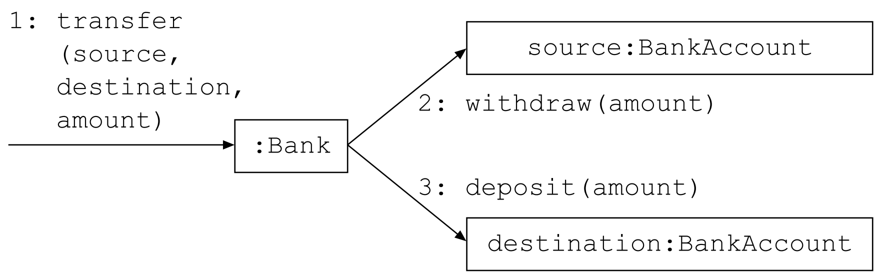


Figure 2.8 Supplementary data in the bank account transferral problem

The list of behaviours and the above figure can help you verify whether the list is complete or not. If all identified behaviours cannot cooperate to complete the task, the list might be incomplete. It is therefore necessary to re-analyse the problem and then review the lists of objects and behaviours. In the example, the three behaviours suffice to complete the bank transferral operation, and you should now be comfortable with the findings in the analysis phase.

As it is found that both source account and destination account are of the same class `BankAccount`, it is possible to model them using a single class definition only. For the `Bank` object, you define a `Bank` class to model it.

In the design phase, you decide that a bank account should maintain its own balance. As a result, an attribute `balance` is added to it. Therefore, you can derive the following class diagrams as shown in Figure 2.9:

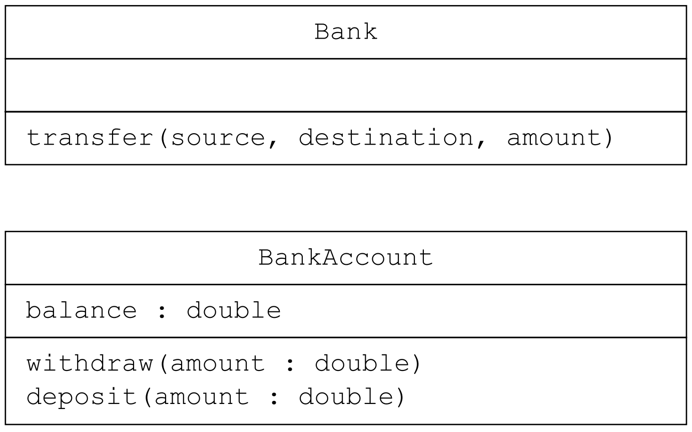

Bank
transfer(source, destination, amount)
BankAccount
balance : double
withdraw(amount : double)
deposit(amount : double)

Figure 2.9 Class definitions in the bank account transferral problem

As the outcomes of the design phase are to be implemented in the Java programming language, you have to consider some language-specific issues. For example, the Java programming language needs the type for each data item so that it can manipulate the data properly.

In the example, the source account and destination account are instances of the `BankAccount` class; the amounts for all behaviours are of type `double` (which allows real numbers to be stored). Data types are discussed in detail in *Unit 3*. Then, the class designs become as shown in Figure 2.10:

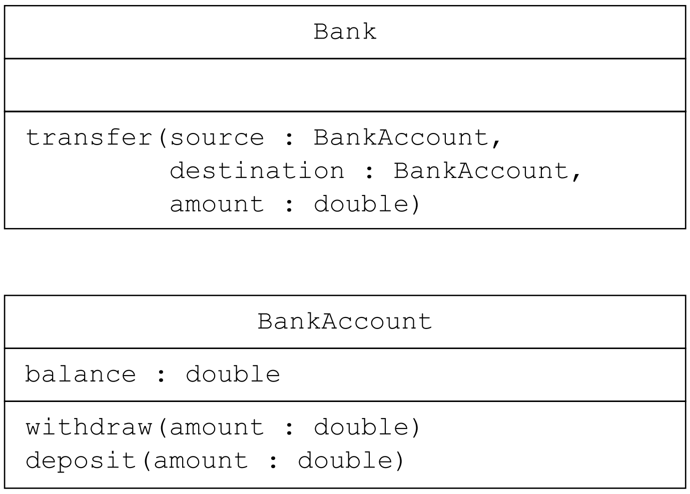

Bank
transfer(source : BankAccount,
destination : BankAccount,
amount : double)
BankAccount
balance : double
withdraw(amount : double)
deposit(amount : double)

Figure 2.10 Class definitions in the bank account transferral problem

Up to this point, the `Bank` object does not have any attributes. However, when you investigate its other operations, you might find that you have to add some attributes to it.

Finally, in the implementation phase, you ‘translate’ the class diagram to Java class definitions. The class definitions shown below might not be complete, since you haven’t learned enough Java yet. They are shown so that you can get an idea about the final products:

```
// The definition of the Bank class
public class Bank {
   // No attributes

   // Behaviours
   public void transfer(BankAccount source,
            BankAccount destination,
            double amount) {
      // send message to source account to withdraw
      money
      source.withdraw(amount);
      // send message to destination account to
      deposit money
      destination.deposit(amount);
   }
}

// The defintion of the BankAccount class
public class BankAccount {
   // Attributes
   private double balance;

   // Behaviours
   public void withdraw(double amount) {......}
   public void deposit(double amount) {......}
}
```

The problem statement and the diagram clearly state that the `transfer` behaviour of the `Bank` class sends messages `withdraw` and `deposit` to the source account and destination account respectively. Therefore, the behaviour can be concretely defined as in the above definition of class `Bank`.

For the second example, you developed the washing machine problem statement in the requirement phase as:

1 The Start Washing button of the washing machine is pressed.

2 The incoming water valve is opened for 5 minutes.

3 The drum motor starts at normal speed for 15 minutes.

4 The outgoing water valve is opened.

5 The drum motor starts at high speed for 5 minutes.

6 The outgoing water valve is closed.

7 The incoming water valve is opened for 5 minutes.

8 The drum motor starts at normal speed for 15 minutes.

9 The outgoing water valve is opened.

10 The drum motor starts at high speed for 5 minutes.

11 The outgoing water valve is closed.

In the analysis phase, you highlight all nouns or noun phrases in the problem statement first:

1 The Start Washing button of the washing machine is pressed.

2 The incoming water valve is opened for 5 minutes.

3 The drum motor starts at normal speed for 15 minutes.

4 The outgoing water valve is opened.

5 The drum motor starts at high speed for 5 minutes.

6 The outgoing water valve is closed.

7 The incoming water valve is opened for 5 minutes.

8 The drum motor starts at normal speed for 15 minutes.

9 The outgoing water valve is opened.

10 The drum motor starts at high speed for 5 minutes.

11 The outgoing water valve is closed.

You can then obtain a list of nouns or noun phrases:

drum motor
high speed
incoming water valve
minute
normal speed
outgoing water valve
Start Washing button
washing machine

Like the previous example, the following table can help you analyse the nouns:

Objects or classes	**Class name**	**Remark**
drum motor	`DrumMotor`	
high speed		Notice the phrase ‘starts at high speed’. ‘High speed’ is used to describe the action ‘starts’. It is related to an action and it is not necessary to model it as a class.
incoming water valve	`IncomingValve`	
minute		Not modelled, since it is a unit.
normal speed		See the remark of high speed.
outgoing water valve	`OutgoingValve`	
Start Washing button		Not modelled, because it is used to request the washing machine to perform its operations (a message).
washing machine	`WashingMachine`	

The class name of the incoming water valve and the outgoing water valve are chosen to be `IncomingValve` and `OutgoingValve` as they are shorter yet self-explanatory. We can then highlight the verbs in the problem statement:

The Start Washing button of the washing machine ***is pressed***.

The incoming water valve ***is opened*** for 5 minutes.

The drum motor ***starts*** at normal speed for 15 minutes.

The outgoing water valve ***is opened***.

The drum motor ***starts*** at high speed for 5 minutes.

The outgoing water valve ***is closed***.

The incoming water valve ***is opened*** for 5 minutes.

The drum motor ***starts*** at normal speed for 15 minutes.

The outgoing water valve ***is opened***.

The drum motor ***starts*** at high speed for 5 minutes.

The outgoing water valve ***is closed***.

Although ‘user’ is not explicitly mentioned, the problem statement implicitly assumes that the user presses the ‘Start Washing’ button, which actually sends a signal to the washing machine to ask it to start washing. Therefore, it is the washing machine that accepts a message `wash` and it performs the `wash` behaviour. So, you can determine the following:

1 The washing machine has the `wash` behaviour.

2 The incoming water valve has the `open` behaviour.

3 The drum motor has the `start` behaviour.

4 The outgoing water valve has the `open` and `close` behaviours.

Then, based on the problem statement, you could draw the following diagram (Figure 2.11) to show the sequence of message-sending and hence the behaviours.

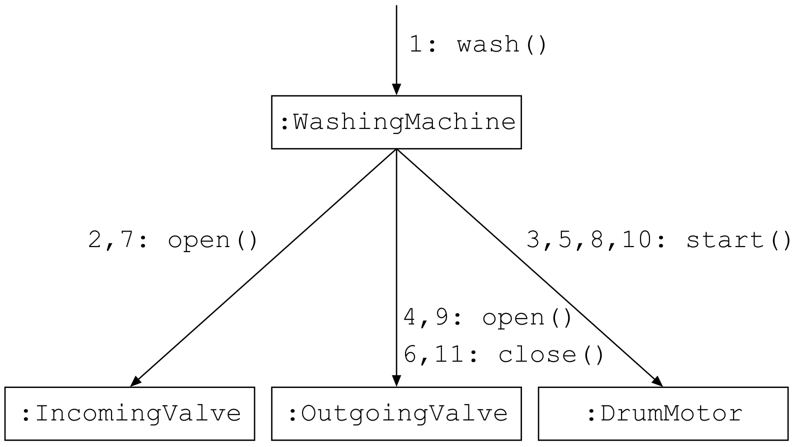


Figure 2.11 Interaction among objects in the washing machine example

You can determine message (or behaviour) names yourself, so that you can easily determine the behaviour effects or results.

Is this the end of the analysis phase? The indicator is that if you can model the original scenario based on the sequence of object behaviours, the phase is completed.

With respect to Figure 2.11, can it model the original washing scenario? Not yet, because some behaviours need supplementary data such as the time the behaviours take. For example, the `open` behaviour of the incoming water valve and the `start` behaviour of the drum motor need such supplementary data. Then, you can enhance Figure 2.11 as shown in Figure 2.12:

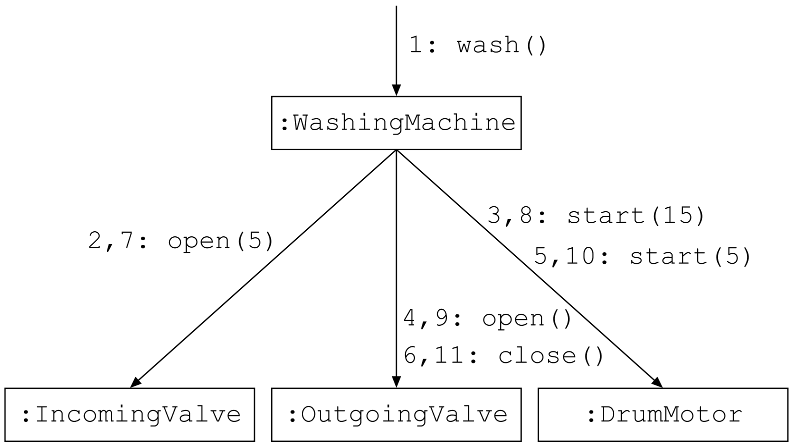
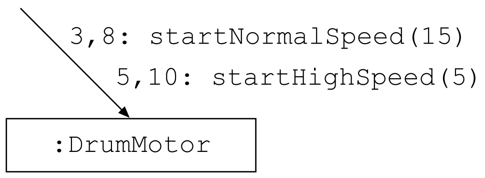


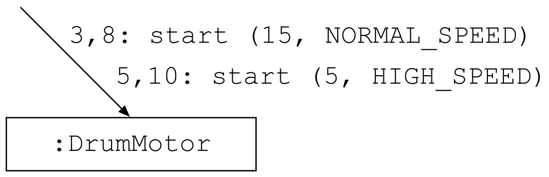
Figure 2.12 Supplementary data in the washing machine problem

Does Figure 2.12 model the scenario stated in the problem statement perfectly? Not yet! Step 3 and step 8 are the washing process, and the drum motor rotates at normal speed for 15 minutes; steps 5 and 10 are the rinsing process in which the drum motor rotates at high speed for 5 minutes. Therefore, it is necessary to identify the difference. You can do so by changing the message type or providing extra supplementary information with the message.

For the former method, the figure is modified to be:



The second approach provides supplementary data:



The first approach uses two message types, `startNormalSpeed` and `startHighSpeed`. The second approach uses a single message type `start` but extra information is required so that the drum motor knows whether it should rotate at normal speed or high speed.

It is up to you to choose the better approach. In this example, the first approach was chosen. The diagram becomes as shown in Figure 2.13:

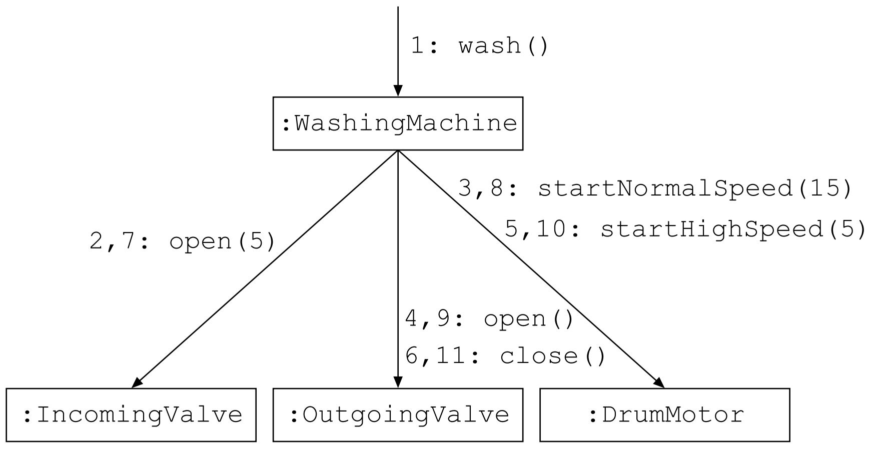


Figure 2.13 The complete analysis of the washing machine problem

This is the end of the analysis phase. You have obtained a list of objects and a well-defined sequence of message sending. Then, you can start the design phase to design the necessary classes.

First of all, you know that there are four classes:

- `WashingMachine`
- `IncomingValve`
- `OutgoingValve`
- `DrumMotor`

For each message type that these objects accept, you should add a behaviour with the same name to it. If a behaviour needs extra information, provide each piece of information with a name and the type of the information. For example, the `open` behaviour of the incoming water valve needs supplementary data for the amount of time the behaviour lasts; that is, a time limit. As the time limit is assumed to be the number of minutes, its value is any whole number and its type is chosen to be `int`, which means integer in the Java programming language. The class behaviours are shown in Figure 2.14.

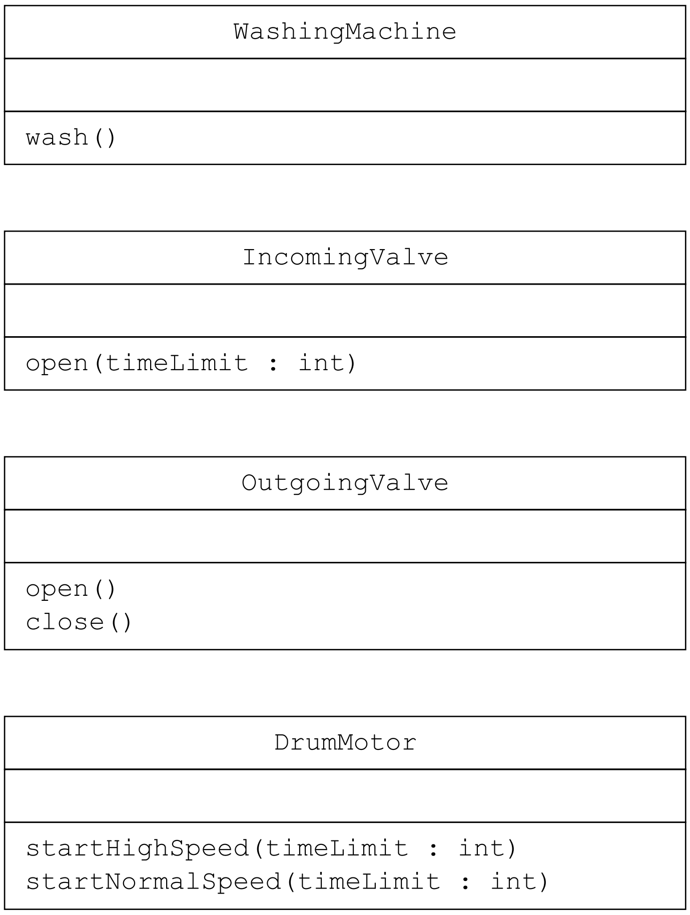


Figure 2.14 Class behaviours for the washing machine problem

Since the incoming water valve, outgoing water valve and drum motor are parts of the washing machine, `WashingMachine` has one instance of `IncomingValve`, `OutgoingValve` and `DrumMotor` as its attributes. The objects in the diagrams can be anonymous, but they *must be named* in class designs and in the class definition written in the Java programming language. Therefore, provide a name for each of them. For example, in Figure 2.15, the name `incomingValve` is used to denote the `IncomingValve` object.

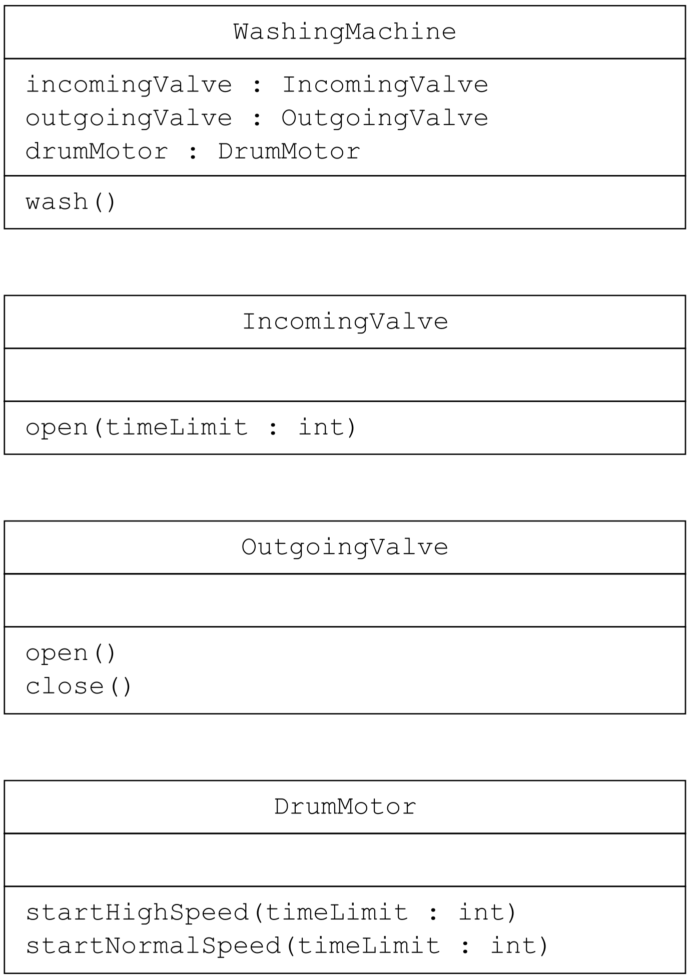

WashingMachine
incomingValve : IncomingValve
outgoingValve : OutgoingValve
drumMotor : DrumMotor
wash()
IncomingValve
open(timeLimit : int)
OutgoingValve
open()
close()
DrumMotor
startHighSpeed(timeLimit : int)
startNormalSpeed(timeLimit : int)

Figure 2.15 Attributes and behaviours for the washing machine problem

Based on such class designs, you can implement them using the Java programming language as:

```
public class WashingMachine {
    // Attributes
    IncomingValve incomingValve;
    OutgoingValve outgoingValve;
    DrumMotor drumMotor;

    // Behaviour
    wash() {
        // Washing
        incomingValve.open(5); // Requirement Phase Line 2
        drumMotor.startNormalSpeed(15); // Requirement Phase Line 3
        outgoingValve.open();// Requirement Phase Line 4
        drumMotor.startHighSpeed(5); // Requirement Phase Line 5
        outgoingValve.close();// Requirement Phase Line 6
        // Rinsing
        incomingValve.open(5); // Requirement Phase Line 7
        drumMotor.startNormalSpeed(15); // Requirement Phase Line 8
        outgoingValve.open();// Requirement Phase Line 9
        drumMotor.startHighSpeed(5); // Requirement Phase Line 10
        outgoingValve.close();// Requirement Phase Line 11
   }
}

public class IncomingValve {
    // No Attribute

    // Behaviour
    public void open(int timeLimit) {......}
}

public class DrumMotor {
    // No Attribute

    // Behaviours
    public void startNormalSpeed(int timeLimit) {......}
    public void startHighSpeed(int timeLimit) {......}
}

public class OutgoingValve {
    // No Attribute

    // Behaviour
    public void open(){......}
    public void close(){......}
}
```

You don't need to understand the details of the above Java codes.

The previous examples illustrate how to model the money transferral of a banking system and a washing machine. Two further case studies are provided below on common problems for commercial software systems. After you have studied these two cases, you should have a general idea of how these systems can be built using the object-oriented approach. The first one is a stock-trading scenario in which a stock brokerage house maintains the holdings of its customers. The other case study is a payroll problem that a human resources system might support.

Case study: a stock-trading problem

In a stock-trading software system, it is necessary to housekeep the customers' portfolios; that is, the amounts and types of stock the customers are holding. If a customer successfully buys or sells stocks, the system must update his or her portfolio accordingly. In this case study, you are only concerned with the scenario when the customer successfully purchases or sells a particular amount of stock in the stock market.

In the requirement phase, if you are not an experienced dealer in a brokerage house but a software developer from a third-party software company, you must interview the staff in the brokerage house for the operational details. After a few interviews and clarifications, you determine the following problem statement:

A stock brokerage house maintains a customer list with names and identity card numbers. For each customer, it is necessary to maintain a holding list. Each holding contains the holding amount, and the stock number identifies each holding.

If a new customer joins the stock brokerage house, the name and the identity card number are added to the customer list.

If a customer successfully purchases a stock for a particular amount, the stock brokerage house gets the customer from the customer list based on the identity number and gets the customer holding from his or her holding list by the stock number. The holding amount is increased for that amount.

If a customer successfully sells a stock for a particular amount, the stock brokerage house gets the customer from the customer list based on the identity number and gets the customer holding from his or her holding list by the stock number; its holding amount is decreased by that amount.

Then, in the analysis phase, the problem statement is analysed by identifying the nouns and noun phrases as follows:

A stock brokerage house maintains a customer list with names and identity card numbers. For each customer, it is necessary to maintain a holding list. Each holding contains the holding amount, and the stock number identifies each holding.

If a new customer joins the stock brokerage house, the name and the identity card number are added to the customer list.

If a customer successfully purchases a stock for a particular amount, the stock brokerage house gets the customer from the customer list based on the identity number and gets the customer holding from his or her holding list by the stock number. The holding amount is increased for that amount.

If a customer successfully sells a stock for a particular amount, the stock brokerage house gets the customer from the customer list based on the identity number and gets the customer holding from his or her holding list by the stock number; its holding amount is decreased by that amount.

By sorting and eliminating the duplicated nouns or noun phrases, you can obtain the following table:

Objects or classes	**Class name**	**Remark**
amount		supplementary information of a behaviour
customer	`Customer`	
customer list	`CustomerList`	
holding	`Holding`	
holding amount		attribute of class `Holding`
holding list	`HoldingList`	
identity card number		attribute of class `Customer`
name		attribute of class `Customer`
stock	`Stock`	
stock brokerage house	`BrokerageHouse`	
stock number		attribute of `Stock` and `Holding` classes

Now, you can investigate the behaviours by highlighting the verbs:

A stock brokerage house ***maintains*** a customer list with names and identity card numbers. For each customer, it is necessary to ***maintain*** a holding list. Each holding ***contains*** the holding amount, and the stock number ***identifies*** each holding.

If a new customer ***joins*** the stock brokerage house, the name and the identity card number are ***added*** in the customer list.

If a customer successfully ***purchases*** a stock for a particular amount, the stock brokerage house ***gets*** the customer from the customer list based on the identity number and ***gets*** the customer holding from his or her holding list by the stock number. The holding amount is ***increased*** for that amount.

If a customer successfully ***sells*** a stock for a particular amount, the stock brokerage house ***gets*** the customer from the customer list based on the identity number and ***gets*** the customer holding from his or her holding list by the stock number; its holding amount is ***decreased*** by that amount.

The first paragraph provides some basic information on the required data to be maintained by a brokerage house. The remaining three paragraphs describe three operations of the brokerage house.

The first operation takes place when a new customer joins the stock brokerage house. You could say that a customer joins the stock brokerage house or a stock brokerage house adds a new customer, but which one is preferable? To answer such a question, you should remind yourself that if an object performs a behaviour, its states are changed to reflect the effect of the behaviour. 'Customer joins a stock brokerage house' implies some states of a customer are updated; 'a stock brokerage house adds a customer' means some states of the stock brokerage house are affected. From this perspective, 'a stock brokerage house adds a customer' seems to be better.

For such an 'add customer' operation, the problem statement states that extra information — name and identity card number — is required. Therefore, the first operation is visualized as in Figure 2.16:

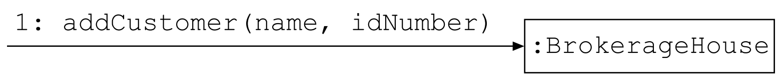


Figure 2.16 Initial relationship in the addCustomer operation

As the `BrokerageHouse` object maintains (or has) a `CustomerList`, the `BrokerageHouse` should add the customer information to the list. From the point of view of object-oriented analysis, it is the `CustomerList` that adds the customer information to itself. Therefore, the responsibility of the `BrokerageHouse` object is to create a `Customer` object and request the `CustomerList` object to add the `Customer` object to itself. The behaviour name `put` is chosen. As the identity card number identifies each customer, two pieces of supplementary information are provided with the `put` message. Therefore, the diagram is enhanced, as shown in Figure 2.17:

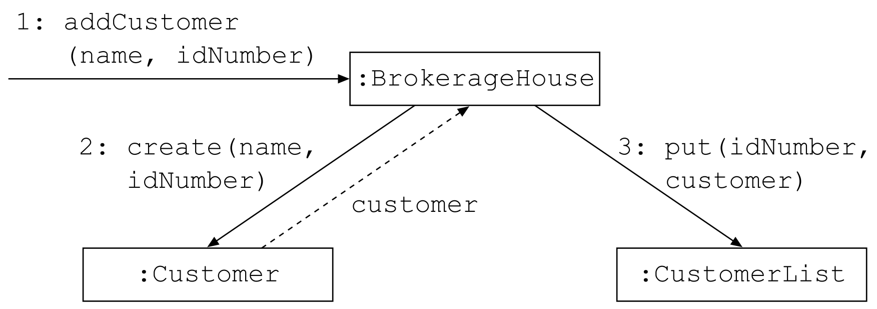

1: addCustomer (name, idNumber)
:BrokerageHouse
2: create(name, idNumber)
customer
3: put(idNumber, customer)
:Customer
:CustomerList

Figure 2.17 Enhancing the addCustomer operation

In Figure 2.17, there is a `CustomerList` object. It is a list of `Customer` objects as mentioned in the problem statement. There are several ways to implement a `CustomerList` in the Java programming language. *Unit 5* discusses one way to implement a Customer List in the Java programming language.

For the second operation, based on the same argument as the 'add customer' operation, it is considered to be the behaviour of the brokerage house as shown in Figure 2.18.

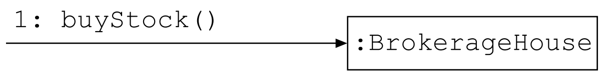


Figure 2.18 The buyStock operation in the stock-trading problem

According to the problem statement, the stock brokerage house locates the customer's information in the customer list by supplying supplementary information such as his or her identity card number. Therefore, the diagram is modified as shown in Figure 2.19:

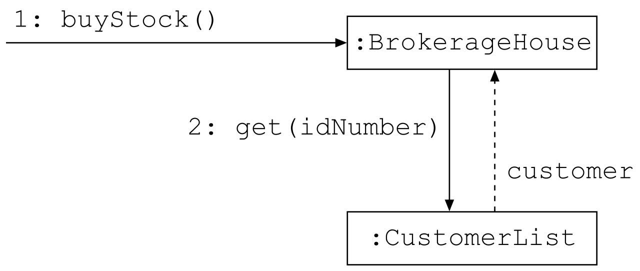


Figure 2.19 Enhancing the buyStock operation

Figure 2.19 is enhanced so that on receiving a request to `buyStock`, the `BrokerageHouse` object sends a request to the `CustomerList` object to retrieve the information of the customer. The supplementary data of the customer's 'identity card number' is required, so that the `CustomerList` object can tell who the customer is. Therefore, there is a return arrow (a dashed line) showing that the `Customer` information is returned to the `BrokerageHouse` object.

However, there is a problem. How can the `BrokerageHouse` object provide the customer identity card number to the `CustomerList`, as the `BrokerageHouse` object receives a request (or message) to buy stock only? Therefore, the `buyStock` behaviour must provide the `BrokerageHouse` object with the customer identity card number so that it can subsequently send the number to the `CustomerList` object. As a result, the diagram is modified as shown in Figure 2.20:

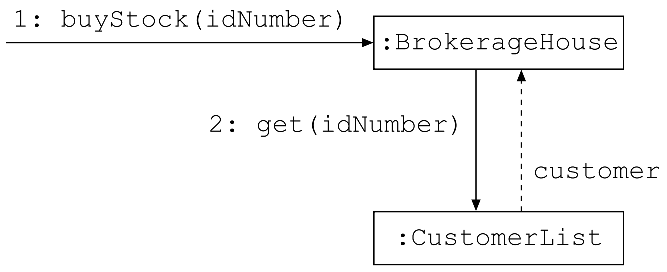


Figure 2.20 Further enhancing the buyStock operation

Now, the `BrokerageHouse` object has obtained the information about the `Customer` object. The subsequent operations include getting a `Holding` object from the customer `HoldingList` object and increasing the holding `amount`. Such behaviours are preferably coordinated by the `Customer` object instead of by the `BrokerageHouse` object, because each `Customer` object should manipulate his or her own `HoldingList`. Therefore, the subsequent behaviours are visualized as shown in Figure 2.21:

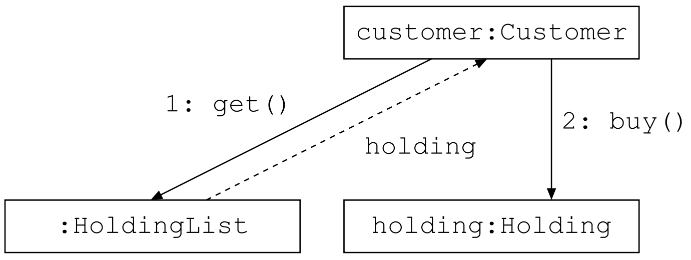


Figure 2.21 Further enhancing the buyStock operation

To locate the `Holding` object, the `HoldingList` object needs the `stockNumber` for identification. Therefore, the `get` behaviour of the `HoldingList` object needs supplementary data in the form of `stockNumber`. For the `Holding` object, the `buy` behaviour needs other supplementary data in the form of the amount of stock purchased. Therefore, the above diagram is further enhanced as shown in Figure 2.22:

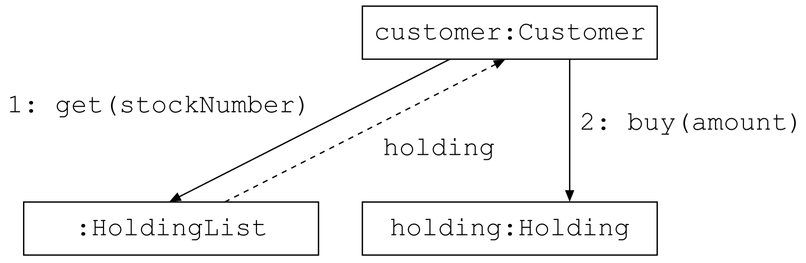


Figure 2.22 Further enhancing the buyStock operation

The `Customer` object will never perform the above sequence of operations unless it receives a message from another object. For this scenario, it is reasonable to state that the `Customer` object performs the sequence of operations if it receives a `buyStock` message from the `BrokerageHouse` object. Therefore, by combining the above two disjointed diagrams and reordering the sequence numbers, the complete operation is shown as in Figure 2.23:

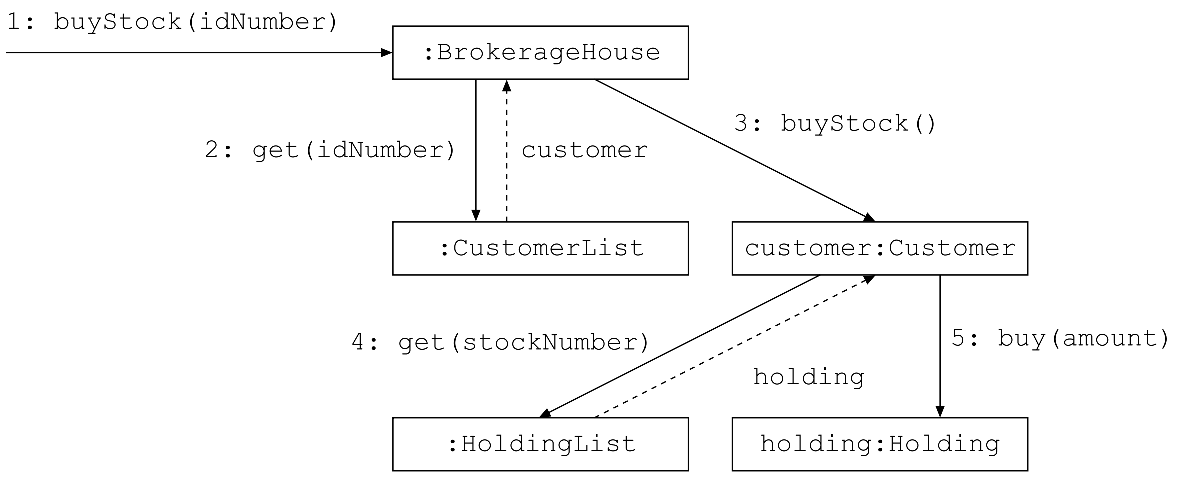


Figure 2.23 The complete buyStock operation

As steps 4 and 5 need supplementary data such as `stockNumber` and `amount`, the message `buyStock` that the `Customer` object receives must contain these two pieces of supplementary data as well. Furthermore, if step 3 needs these two pieces of supplementary data, the `BrokerageHouse` object must obtain such information from the first message. Therefore, the diagram is further enhanced as in Figure 2.24:

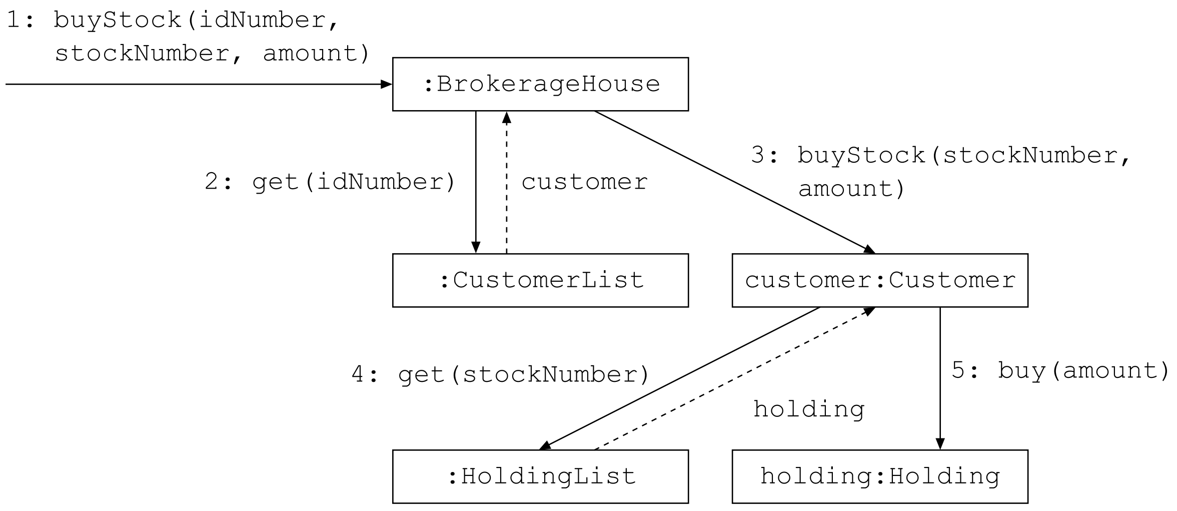


Figure 2.24 The enhanced final buyStock operation

After the above-mentioned enhancements, the above diagram is derived as the final version. Please verify the sequence of behaviours in the figure to see how it models the scenario. You can see that in the process to determine the above diagram, you have to ask yourself repeatedly whether the diagram can model the entire operation. If it is not complete, try to find a way to enhance it.

Up to now, although you might understand how the diagram is enhanced incrementally until it is complete for the above scenario, you might think that you cannot derive such a diagram by yourself. Don't panic. After some practice, you will be able to do it.

If you derive a version and you think it is complete, you may start the design and implementation phases to write the class definitions. If you find that something is missing or imperfect, you can repeat the analysis phase to analyse the scenario and improve the diagram. As you gain more experience, you can prevent yourself from repeating the phases and you'll take less time to complete the whole process. This is a common learning process for software developers.

For the `sellStock` operation, the operational sequence is the same except the message types are different. All '`buy`' messages are replaced with '`sell`' messages. The diagram is therefore as shown in Figure 2.25:

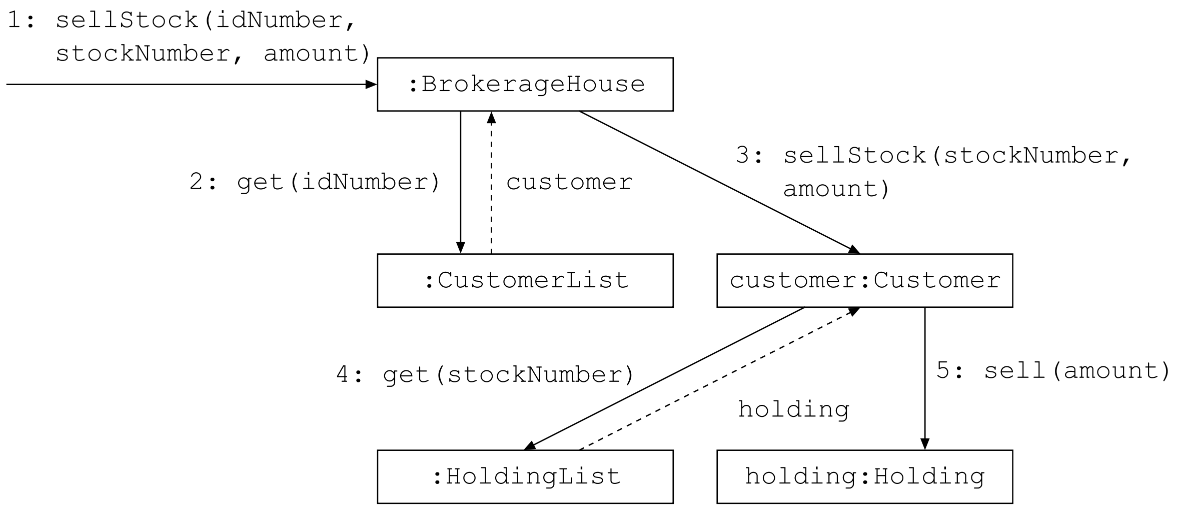


Figure 2.25 The complete sellStock operation

Great! We have derived the three diagrams for the three operations of the stock-trading problem. This is the end of the analysis phase and the start of the design phase. The tasks in the design phase are to derive the class definitions so that they can be implemented by writing Java programs in the implementation phase.

At the beginning of the analysis phase, we identified six objects of different categories (and hence six classes) by locating the nouns in the problem statement; only five of them were used in the diagrams. Therefore, it is necessary to model only five categories (or classes) of objects. They are `BrokerageHouse`, `CustomerList`, `Customer`, `HoldingList` and `Holding`.

The software library that comes with the Java Software Development Kit (Java SDK) provides a `Map` class. Instances of the `Map` class can store (key, value) pairs in which the keys are unique. Since `CustomerList` is used to store (`idNumber`, `customer`) pairs, the `Map` class is suitable for this purpose. Similarly (`stockNumber`, `holding`) pairs are stored in `HoldingList` and the use of the `Map` class is appropriate. You should re-use the existing class definitions as much as possible, because they have been used and tested by a lot of software developers.

Therefore, you have to model the remaining `BrokerageHouse`, `Customer` and `Holding` classes only.

First of all, based on the table derived by the analysis of the nouns, you can determine the class definitions with attributes as shown in Figure 2.26:

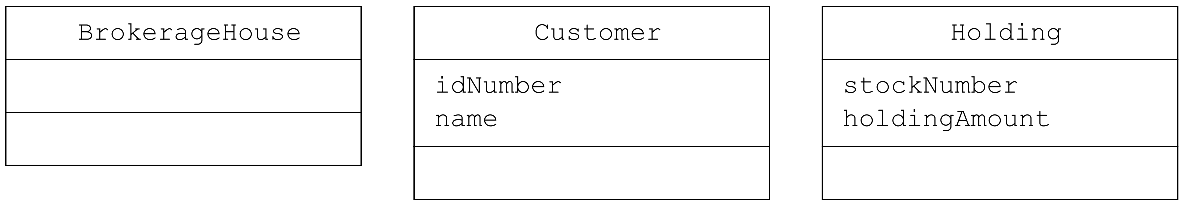


Figure 2.26 Class definitions for the stock-trading problem

Then, you have to determine the type of each attribute so that the Java programming language can handle the data properly. As all attributes except the holding amount should be textual, you can choose the type `String` for them. The holding amount must be a whole number; the type `int` is therefore chosen. Then, the class diagrams become as shown in Figure 2.27:

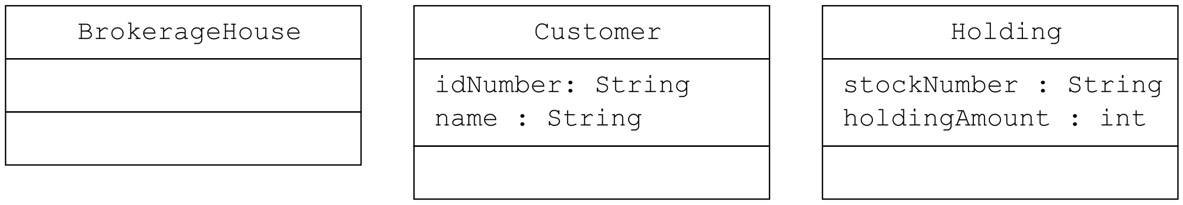


Figure 2.27 Enhancing the class definitions for the stock-trading problem

Now, you can further enhance the class diagram based on the three operational diagrams. We said that if an object always sends messages to fixed objects, you can add attributes for those recipient objects to the class. In our case, the `BrokerageHouse` object keeps on sending messages to the `CustomerList` object, and each `Customer` object sends messages to its `HoldingList` object. Therefore, a `CustomerList` attribute is added to the `BrokerageHouse` class, and a `HoldingList` attribute is added to the `Customer` class. As mentioned, a `Map` comes with the Java library that can be substituted for `CustomerList` and `HoldlingList`, so the class definitions are enhanced as shown in Figure 2.28:

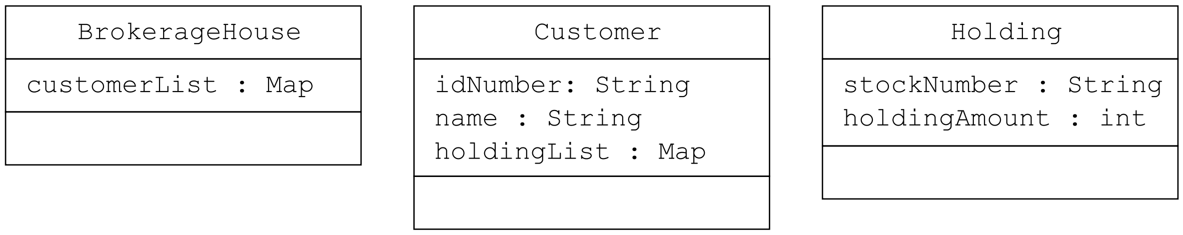
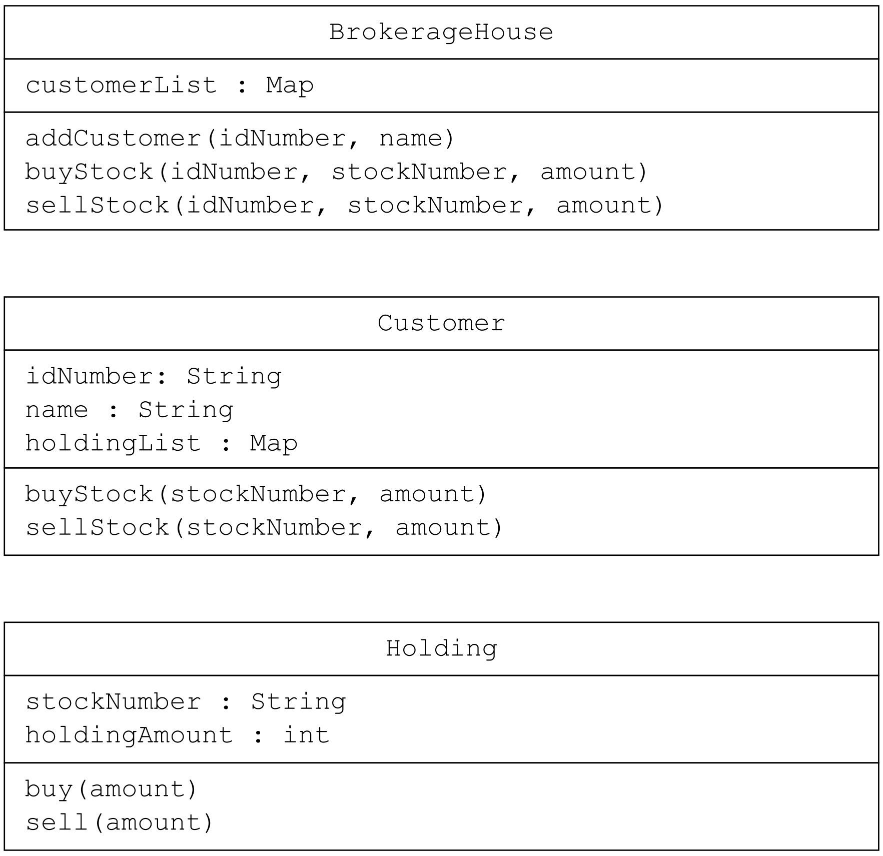


Figure 2.28 Adding attributes to the class definitions for the stock-trading problem

You can now add the behaviours to the class definitions. If an object receives a particular type of message, the corresponding class must define a behaviour for it. Based on the three operation diagrams, you can enrich the class diagrams as shown in Figure 2.29:



```
BrokerageHouse
customerList : Map
addCustomer(idNumber, name)
buyStock(idNumber, stockNumber, amount)
sellStock(idNumber, stockNumber, amount)

Customer
idNumber: String
name : String
holdingList : Map
buyStock(stockNumber, amount)
sellStock(stockNumber, amount)

Holding
stockNumber : String
holdingAmount : int
buy(amount)
sell(amount)
```

Figure 2.29 Adding behaviours to the class definitions for the stock-trading problem

It is also necessary to specify the types of supplementary information of the messages. Therefore, the class diagrams are enhanced as shown in Figure 2.30:

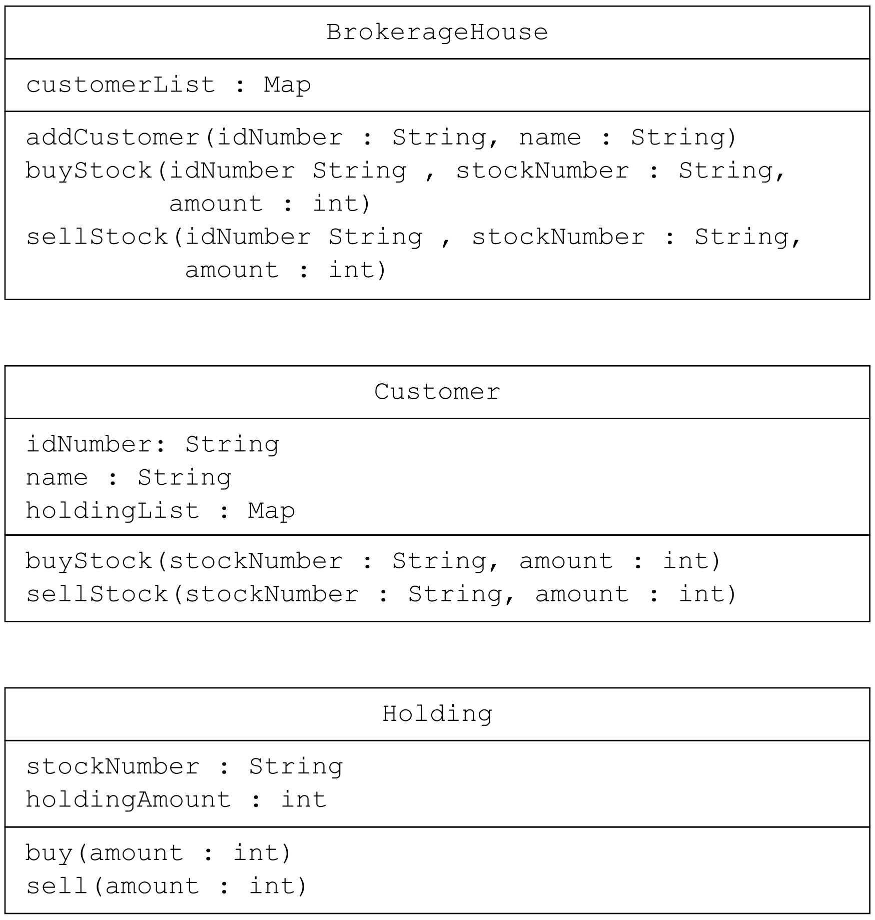

BrokerageHouse
customerList : Map
addCustomer(idNumber : String, name : String)
buyStock(idNumber String , stockNumber : String, amount : int)
sellStock(idNumber String , stockNumber : String, amount : int)
Customer
idNumber: String
name : String
holdingList : Map
buyStock(stockNumber : String, amount : int)
sellStock(stockNumber : String, amount : int)
Holding
stockNumber : String
holdingAmount : int
buy(amount : int)
sell(amount : int)

Figure 2.30 Specifying data types in the class definitions for the stock-trading problem

You have completed the design phase. Now you can now start to implement the classes with the Java programming language. The following three class definitions is one way to implement the class definitions in the Java programming language, but remember we'll take you through classes and objects in *Unit 3* in much greater detail:

```
// Definition of the BrokerageHouse class
public class BrokerageHouse {
  // Attribute
  private Map customerList;

  ......

  // Behaviours
  public void addCustomer(String idNumber, String name) {
    Customer customer = new Customer(idNumber, name);
    customerList.put(idNumber, customer);
  }

  public void buyStock(String idNumber,String stockNumber,int amount) {
    Customer customer = (Customer) customerList.get(idNumber);
    customer.buyStock(stockNumber, amount);
  }
  public void sellStock(String idNumber,String stockNumber,int amount){
    Customer customer = (Customer) customerList.get(idNumber);
    customer.sellStock(stockNumber, amount);
  }
}

// Definition for the Customer class
public class Customer {
  // Attributes
  private String idNumber;
  private String name;
  private Map holdingList;

  ......

  // Behaviours
  public void buyStock(String stockNumber, int amount) {
    Holding holding = (Holding) holdingList.get(stockNumber);
    holding.buy(amount);
  }

  public void sellStock(String stockNumber, int amount) {
    Holding holding = (Holding) holdingList.get(stockNumber);
    holding.sell(amount);
  }
}

// Definition of the Holding class
public class Holding {
  // Attributes
  private String stockNumber;
  private int holdingAmount;

  // Behaviours
  public void buy(int amount) {
    holdingAmount = holdingAmount + amount;
  }

  public void sell(int amount) {
    holdingAmount = holdingAmount - amount;
  }
}
```

The above class definitions in the Java programming language are still incomplete. However, you can see that the attributes and behaviours determined in the earlier phases can be implemented in the Java programming language. *Unit 3* through *Unit 10* provide ways to complete the missing parts.

Case study: a payroll calculation problem

Here is another case study — a payroll calculation problem. It is a common operation in a human resources system to determine the total salary of all staff.

In the requirement phase, you interview the staff in the human resources department and obtain the following problem statement about the calculation of staff members' salaries in the company:

> The company maintains a staff list. For each staff member on the staff list, his or her salary is calculated to be the number of working days in the month times the daily wage.
>
> To calculate the total salary of all staff, all staff salaries in the staff list are added to get the total.

The problem statement is comparatively simpler than the previous examples and case studies.

In the analysis phase, the nouns and noun phrases in the problem statement are identified as:

> The <u>company</u> maintains a <u>staff list</u>. For each <u>staff member</u> on the <u>staff list</u>, his or her <u>salary</u> is calculated to be the <u>number of working days in the month</u> times the <u>daily wage</u>.
>
> To calculate the <u>total salary of all staff</u>, all <u>staff salaries</u> in the <u>staff list</u> are added to get the <u>total</u>.

The identified nouns in the problem statements are listed below:

Objects or classes	**Class name**	**Remark**
company	`Company`	
staff list	`StaffList`	
salary		to be calculated by a Staff object
number of working days in the month		supplementary information of a behaviour
daily wage		attribute of the Staff class
total salary of all staff		to be calculated by a Company object
staff	`Staff`	

According to the scenario stated in the problem statement, the first behaviour needs to obtain a `Staff` object from the `StaffList` object, as shown in Figure 2.31.

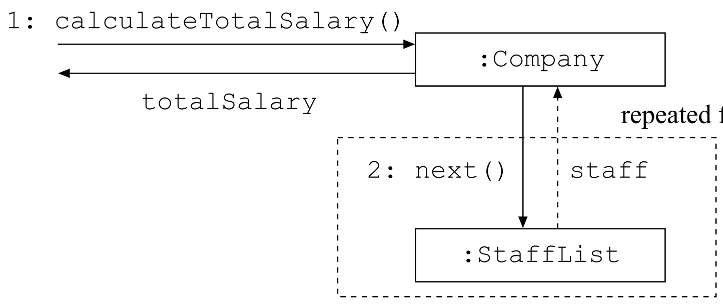


Figure 2.31 Obtaining a Staff object from the StaffList object

The `StaffList` object is similar to the `CustomerList` and `HoldingList` objects in the previous case studies. The difference is that it is required to return the next `Staff` object from the `StaffList` object. In the previous stock trading case study, the `CustomerList` object returns a `Customer` object according to the customer identity card number. In this case, a message `next` is sent repeatedly to the `StaffList` object to get the next `Staff` object until all `Staff` objects are obtained.

In the real world, a staff member in the human resources department calculates the salary of a staff member. However, for software development using the object-oriented approach, all information about an object should be determined by itself. Therefore, it is preferable to request a `Staff` object to determine its own salary.

As a result, once the `Staff` object is returned from the `StaffList` object, a message `calculateSalary` is sent to it so that each `Staff` object can calculate his or her own salary. Therefore, the diagram is further enhanced as shown in Figure 2.32:

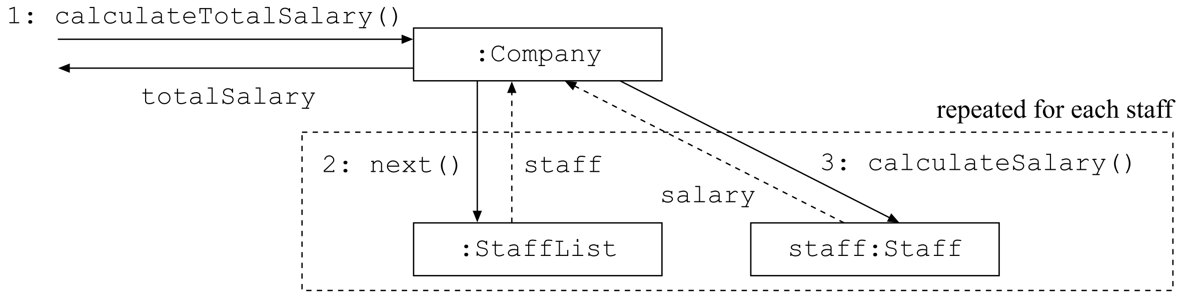


Figure 2.32 Operation to calculate and collect salary information

The `Staff` object in Figure 2.32 is named to denote that it is the one to be returned by the `next` behaviour of the `StaffList` object.

The calculation of the salary by a `Staff` object needs two pieces of information — number of working days in the month and daily wage. The `Staff` object maintains its own attribute ‘daily wage’, but the number of working days in the month should be obtained from the message that is sent to it. Therefore, supplementary data concerning the 'number of working days' is added to the message of `calculateSalary` that is sent to a `Staff` object. As the `Company` object has to provide such data to each `Staff` object, it also needs such data from the message that is sent to it. As a result, the message `calculateTotalSalary` needs such data as well. The diagram is enhanced as in Figure 2.33:

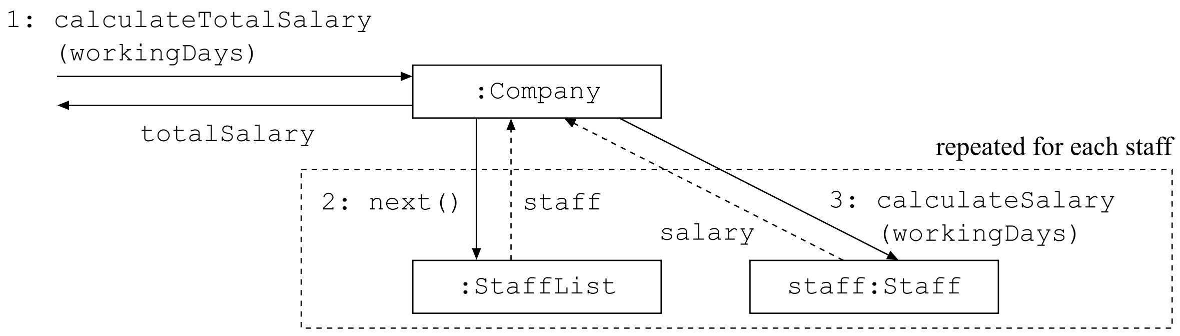


Figure 2.33 Enhancing the payroll calculation problem

By tracing the sequence in Figure 2.33, you can verify that the sequence of behaviours can fulfil the task of the total salary calculation. It is the end of the analysis phase; you can now start the design phase to derive the class designs.

First of all, there are three categories of objects, — `Company`, `StaffList` and `Staff`. The problem involves a lot of `Staff` objects, as each one models a staff member in the company. However, we just need to derive a class `Staff` to model them all. When the program is executed, the class definition of `Staff` can be used to create the necessary `Staff` objects to model all staff members.

For the previous case study, you used `Map` to generically model the `StaffList`. However, if you study the library document of the Java programming language, the `Map` does not support getting the next `Staff` without providing any supplementary data. Therefore, you still need to write the `StaffList` class as started in Figure 2.34.

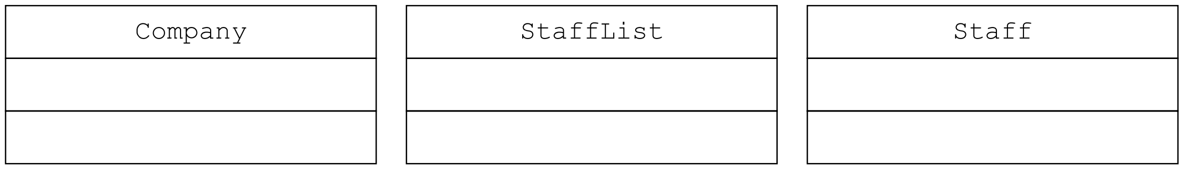


Figure 2.34 Class definitions for the payroll calculation problem

Then, you can add those identified object attributes to the class designs. Among the three classes, only the `Staff` class needs an attribute for 'daily wage' (`dailyWage`); it can be any number. Therefore, its type should be `double` as shown in Figure 2.35.

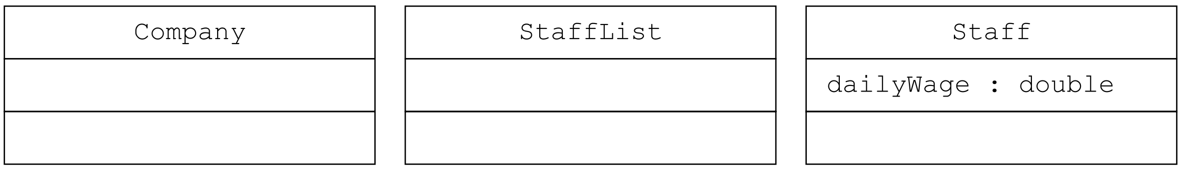


Figure 2.35 Class definitions for the payroll calculation problem

Now, you have to study whether some objects always send messages to other fixed objects. In this case study, the `Company` object always sends a `next` message to the `StaffList` object. Therefore, an attribute for the `StaffList` object is added as shown in Figure 2.36.

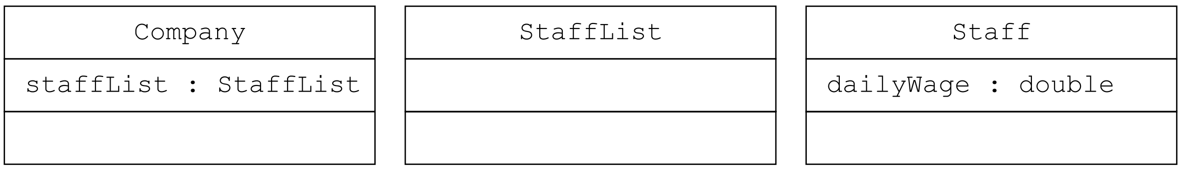


Figure 2.36 Adding attributes to the class definitions for the payroll calculation problem

Now, you can add the behaviours to the classes by studying the operational diagram. The `Company` object receives a message type `calculateTotalSalary` with supplementary data 'number of working days' and will return the total salary. The `StaffList` object can receive a message `next` and return the next `Staff`. For a `Staff` object, it receives a message `calculuateSalary` with supplementary data 'number of working days' and returns the salary. Based on these findings, you can enrich the class diagrams as shown in Figure 2.37.

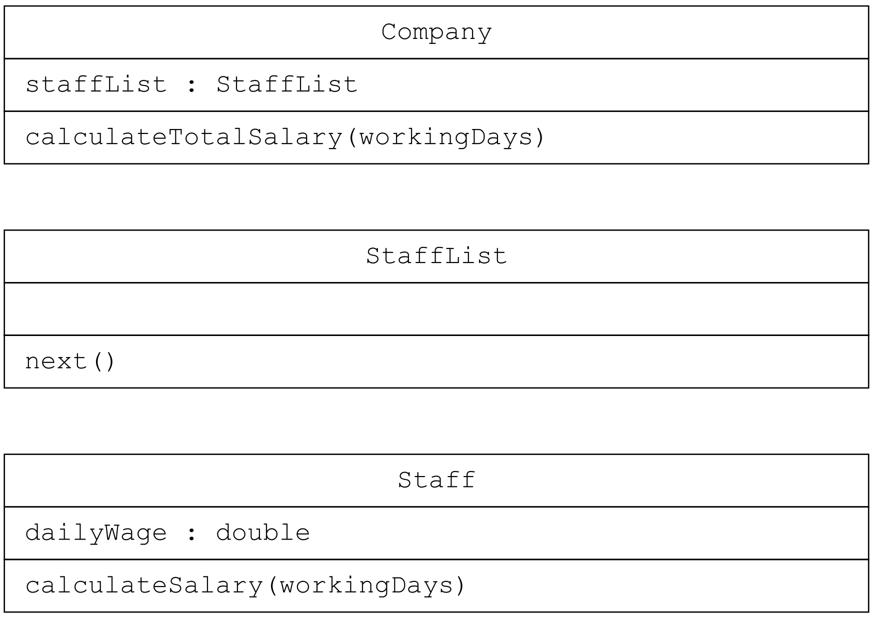


Figure 2.37 Adding behaviours to the class definitions for the payroll calculation problem

Then, you have to determine the type of supplementary data and the type of return values of the behaviours, if any. The class designs are enhanced as shown in Figure 2.38.

Company
`staffList : StaffList`
`calculateTotalSalary(workingDays : int) : double`

StaffList
`next() : Staff`

Staff
`dailyWage : double`
`calculateSalary(workingDays : int) : double`

Figure 2.38 Adding supplementary data types to the class definitions for the payroll calculation problem

This is the end of the design phase. You can now implement the class designs with the class definitions written in the Java programming language:

```
// Definition of class Company
public class Company {
   // Attribute
   private StaffList stafflist;

   ......

   // Behaviour
   public double calculateTotalSalary(int workingDays) {
      double total = 0.0;

      ......

      return total;
   }
}

// Definition of class StaffList
public class StaffList {
   // Attribute

   ......

   // Behaviours
   public Staff next() {
      Staff staff = null;

      ......

      return staff;
   }
}

// Definition of class Staff
public class Staff {
   // Attribute
   private double dailyWage;

   ......

   // Behaviour
   public double calculateSalary(int workingDays) {
      return workingDays * dailyWage;
   }
}
```

The examples and case studies illustrate the software development process using the object-oriented approach. The activities performed in each phase of the software development cycle are summarized below:

1 In the requirement phase, you collect the necessary information about the problem or scenarios. You probably have to interview the involved users. You should continue to collect the requirements until you can present the scenarios of the problem clearly.

2 You investigate the scenario to determine the involved entities (objects) and their interactions in the analysis phase. Based on the analysis, you should be able to draw diagrams showing the sequences of messages being sent, so that you can clearly visualize their interaction and sequence.

   If you find that the diagram is disjointed — that is, some objects have no direct or indirect communications with other objects or the required result cannot be obtained — it probably means you have not collected enough information. Then, you should repeat the activities in the requirement phase.

   Furthermore, the object attributes are identified in this phase from the problem statement.

3 Based on the interaction among the objects, you can derive the class designs of the involved objects in the design phase.

   Based on the message-sending diagram obtained in the analysis phase, you can determine the entities involved and hence the classes. If there is an arrow that points at an object in the diagram, it means that the object can accept a message and the class definition for the object must have such a behaviour. When the object attributes are identified in the analysis phase, you can derive the class designs.

4 Then, you can use the Java programming language to implement the class definition in the implementation phase.

We have not discussed the tasks to be undertaken in the testing and maintenance phases, as our current concern is to develop the class definitions that can model the scenarios in the problem. When you have acquired enough knowledge of programming in the Java programming language, we will progressively discuss the tasks to be performed in these phases.

You should now have found that the object-oriented approach is not that theoretical and you can really manage it. Please use the following self-test to verify your understanding of the object-oriented approach in software development.

Self-test 2.1

Study the following problem statement about what a teacher does when he or she is preparing a student test result summary.

> Each class has a student list.
>
> To set the mark of the subject Chinese for a student, the student is located in the student list by the student number and the mark of the subject Chinese is set for the student.
>
> The operations for setting the marks of subjects English and Mathematics are similar.

Please analyse the problem statement and design the classes that can model the above scenario. You don't have to convert the class designs into actual class definitions in the Java programming language.

Common attributes and behaviours among objects of different classes

From the case study of the payroll calculation system, there is only one type of staff and a single salary calculation method applies to all staff members. In real cases, there may be several types of staff such as clerks, managers and directors. They have some common attributes, such as name, staff number and so on. However, their salary calculation methods might be different. For example, a clerk gets paid for overtime work, a manager is entitled to some allowances and the salary of a director is commission based.

The problem for us is: Is it necessary to write different class definitions for `Clerk`, `Manager` and `Director` from scratch after one of them has been developed? After you have read the following section, you will know that the object-oriented paradigm allows class definitions to share common attributes and behaviours.

In the software development cycle using the object-oriented approach, you derive the classes, their attributes and behaviours from the problem statements in the analysis phase. Furthermore, you modelled the scenario of the sequence of behaviours by sending messages among the objects. While you are analysing the object attributes and behaviours, you might find (decide) that some of them have common attributes and behaviours.

For example, in the payroll calculation problem, if the problem is changed to support different types of staff member, such as clerk, manager and director, what you can find is that it is necessary to model `Clerk`, `Manager`, and `Director`. By further analysis, you find that these three classes share common attributes and behaviours as shown in Figure 2.39.

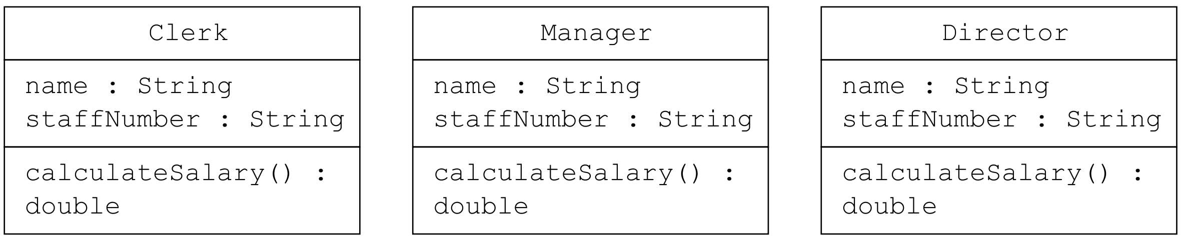


Figure 2.39 Common attributes among employees

(For clarity, the supplementary data with the `calculateSalary` message is omitted and only the common attributes and behaviours are shown.)

The object-oriented paradigm enables you to extract the common attributes and behaviours among classes. A new class is defined that has them; all others are defined as extensions. For example, a general class `Staff` is defined that has the two attributes, name and staff number, and the behaviour `calculateSalary`. Then, the classes `Clerk`, `Manager` and `Director` are defined to be the extensions of the `Staff` class definition. Such a scenario is shown in Figure 2.40:

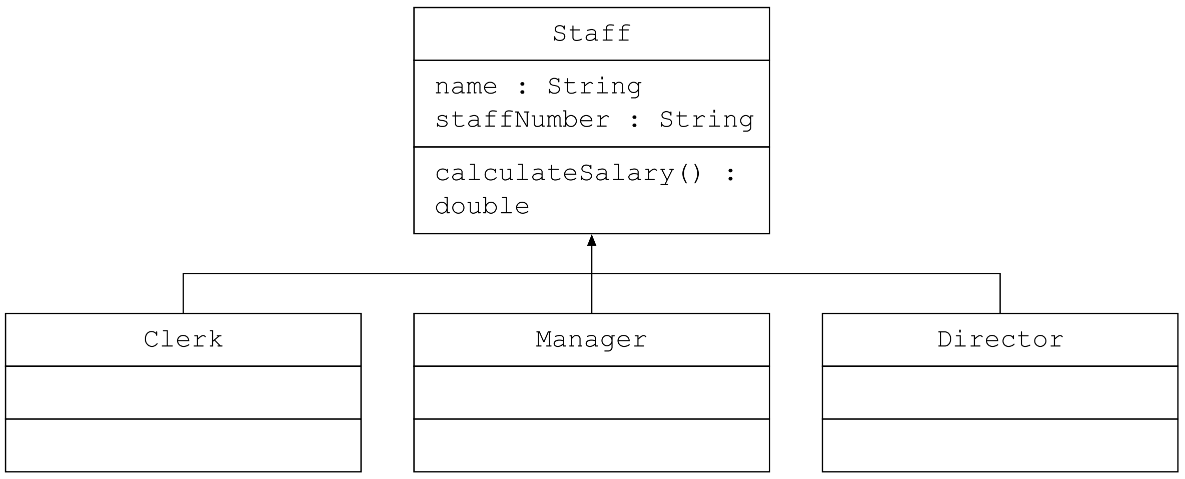


Figure 2.40 Extensions of a staff class

We mentioned that a clerk has an overtime pay rate, a manager has a cash allowance and a director has a commission rate. Furthermore, the salary calculation methods for different `staff` types are different. Therefore, we will define the attributes and the individual 'calculate salary' behaviours for the classes `Clerk`, `Manager` and `Director`. Figure 2.41 presents this idea.

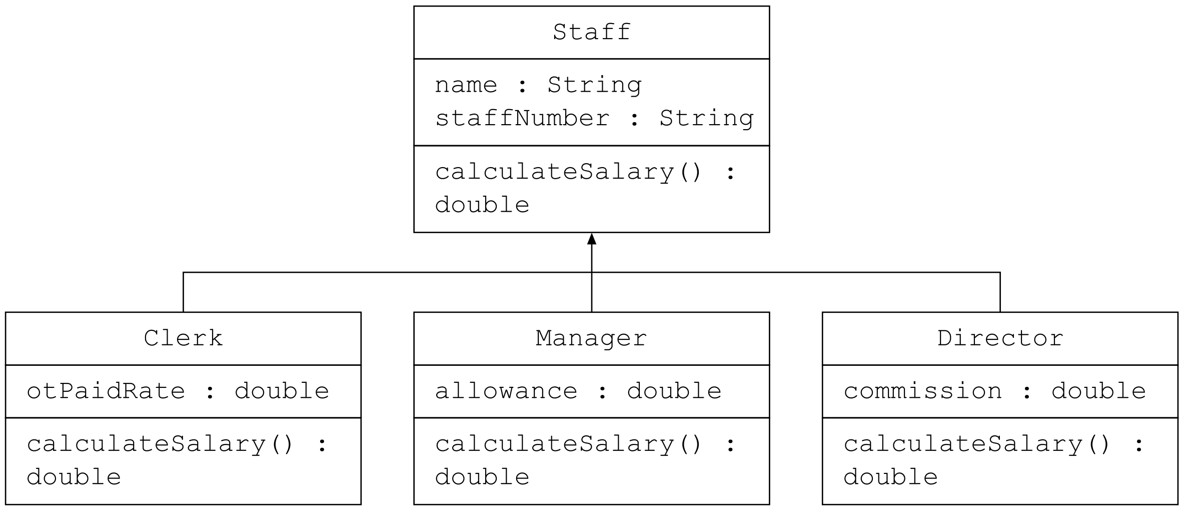


Figure 2.41 Defining the attributes in the extensions of the staff class

The classes `Clerk`, `Manager` and `Director` use their own `calculateSalary` methods instead of that of the class `Staff`. This is called *method overriding*. Although it is necessary to write one more class definition, a great advantage of the above class design is that it minimizes the duplicated codes when you write the class definitions during the implementation phase. The way to write the classes `Clerk`, `Manager` and `Director` are the extensions only. If the common attributes and behaviours in the `Staff` class are changed, the changes are implicitly applied to the `Clerk`, `Manager` and `Director` classes as well. Such a feature greatly minimizes the efforts to maintain a common set of attributes and behaviours in the classes.

Furthermore, as the classes `Clerk`, `Manager` and `Director` are the extensions of the `Staff` class, they have all the attributes and behaviours that the `Staff` class has. The implication is that as an object of the `Staff` class can accept a message `calculateSalary`, so the objects of the class `Clerk`, `Manager` and `Director` can accept the same message type as well. In the object-oriented paradigm, we say that the `Clerk`, `Manager` and `Director` classes *inherit* the attributes and behaviours from the `Staff` class. Such a phenomenon is known as *inheritance*.

A `Clerk` object is also a `Staff` object, but a `Staff` object is not necessarily a `Clerk` object. As the `Clerk`, `Manager` and `Director` classes define their own `calculateSalary` behaviours, the salary calculation of a `Clerk` object can involve the overtime pay rate, whereas those of `Manager` object and `Director` object can involve the allowance and commission respectively.

A benefit of using such an approach is that it enables programs to be written to manipulate general `Staff` objects. As objects of classes `Clerk`, `Manager` and `Director` can be treated as objects of `Staff`, they must accept the same messages and perform the behaviours that a `Staff` object accepts but perhaps perform them in their own ways.

If you can determine the similarities among the classes and use an extra class to obtain them in the analysis phase, it is then possible to design the classes based on that finding. This technique can prevent you from duplicating the codes in the implementation phase and greatly reduce your effort in the testing and maintenance phases. In this unit, the activities in the analysis phase did not include the steps in determining the similarities among objects (and hence classes). *Unit 7* discusses the way to use such techniques in various phases in the software development cycle.

Case studies

The payroll calculation problem is a typical example that involves common attributes and behaviours among classes. Here are two more case studies that involve classes that have common attributes and behaviours.

Different bank account types

A customer can open different types of bank account in a bank. Let's consider savings accounts and time deposit accounts. The accounts should have attributes such as account owner, account number, balance, account open date, currency, and interest rate. Furthermore, they should have a behaviour such as `createAccount()`.

Figure 2.42 shows the relationships among them.

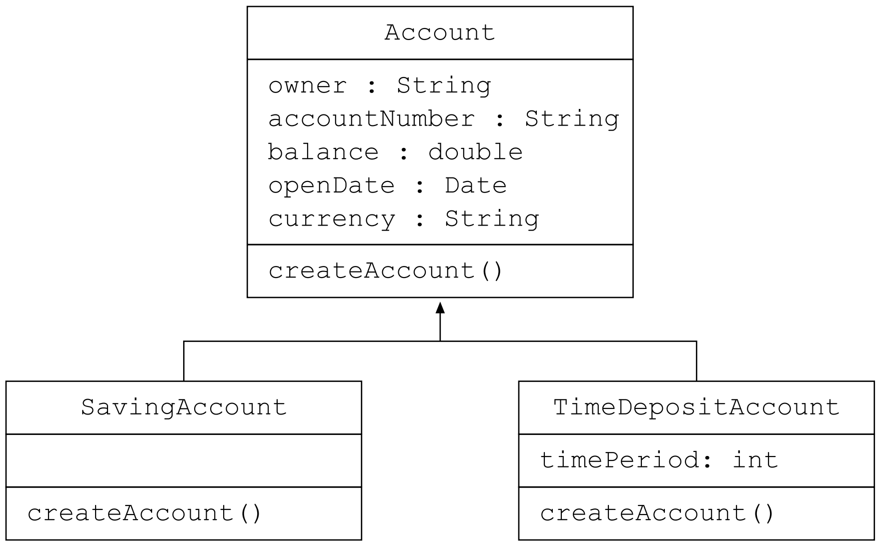


Figure 2.42 Relationship among bank account classes

You can see that the class definitions of `SavingAccount` and `TimeDepositAccount` can be derived based on the `Account` class. The implementation is much simpler, and it is a preferable way to implement a bank system with different bank account types.

Different clothing types

Suppose that you want to write a software system to be used in a dry cleaner's that uses a computerized washing machine to wash customers' clothes. It is necessary to use a `Clothing` object to model the clothes that a customer brings to the shop.

What the system needs to know about the clothes is the weight, price, request date and the way to wash them. The attributes weight, price and request date are common to all `Clothing` objects. However, the behaviour required to wash them is different. Therefore, we can derive two more classes, `WaterWashableClothing` and `DryWashableClothing`. The relationships among the three classes are shown in Figure 2.43.

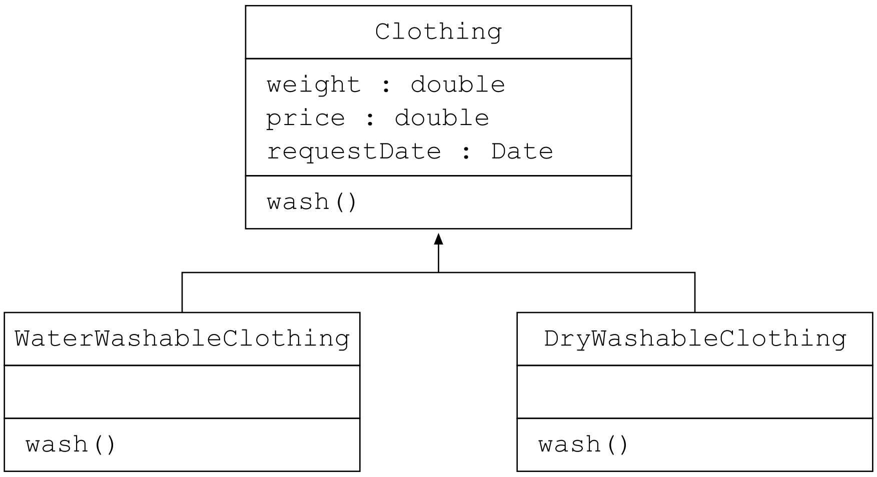


Figure 2.43 Relationship among the clothes washing classes

A `WaterWashableClothing` object and a `DryWashableClothing` object are generically treated as a `Clothing` object. Then, when a bag of customer's clothing gets washed, a message `wash` is sent to the object. If it is `WaterWashableClothing`, it uses water to wash the clothing. If it is `DryWashableClothing`, it uses a cleaner to wash the clothing.

You can see that the object itself determines the way to manipulate itself; you will never see dry-clean-only clothing washed in water!

A general class and its specific classes

Researchers can develop software systems that simulate and predict the outcomes of real-world situations. Such real-world situations may involve many objects and each object performs differently based on its current environment and its own states. From this perspective, software development with the object-oriented paradigm fits such requirements perfectly, because each object can model a real-world entity; the system repeatedly sends messages to the objects so that they can perform in their own ways. At the end, the simulation software can find out the object states to determine the required outcome.

For example, you are going to simulate an isolated environment for animals to determine the numbers of each type at the end of the simulation. Concerned animals include lions, elephants, hawks, owls, lizards and snakes.

There are a lot of objects in such an environment, but you only need to derive these six classes. When the simulation software is executed, an object is created for each animal.

In the analysis phase, you find that there are objects (and classes) that share common attributes and behaviours. For example, they all have weight, age and life expectancy, and common behaviours such as eat, sleep, move and reproduce. Hawks and owls can fly as well.

If you design the classes for the animals individually, you have to duplicate the common attributes in each class definition, which can be a nightmare if you add one attribute to each later on. Therefore, you should use the technique presented earlier to use a generic class to capture the similarities among the classes as shown in Figure 2.44. For example:

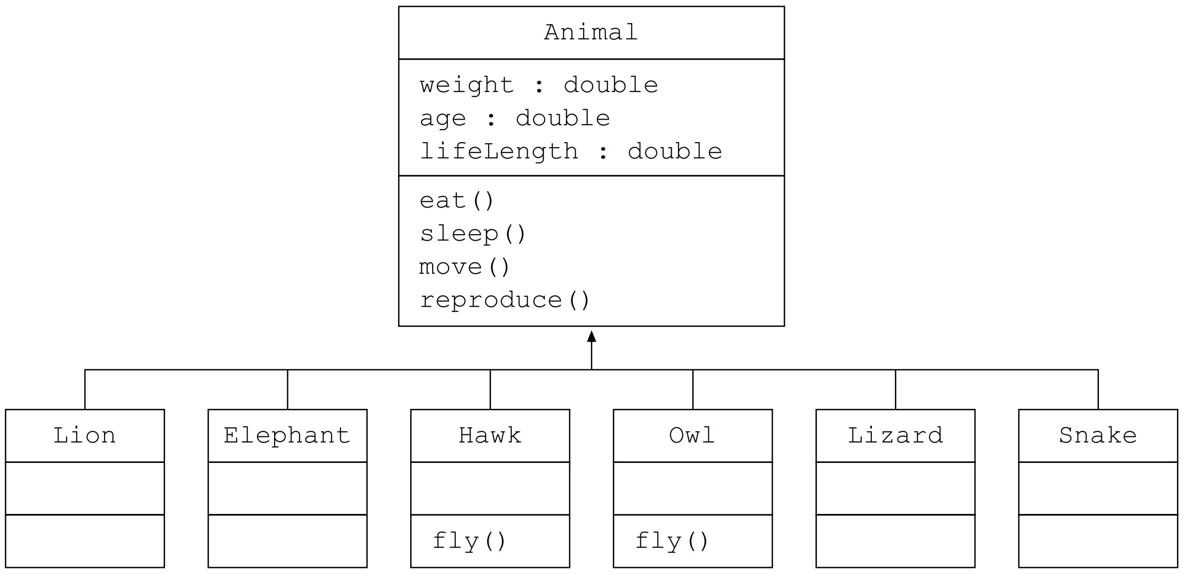


Figure 2.44 Creation of a generic Animal class with extension classes

Furthermore, the `hawk` and `owl` class share the common behaviour `fly()`. Therefore, you could use a `Bird` class to show the similarity of `Hawk` and `Owl` as shown in Figure 2.45.

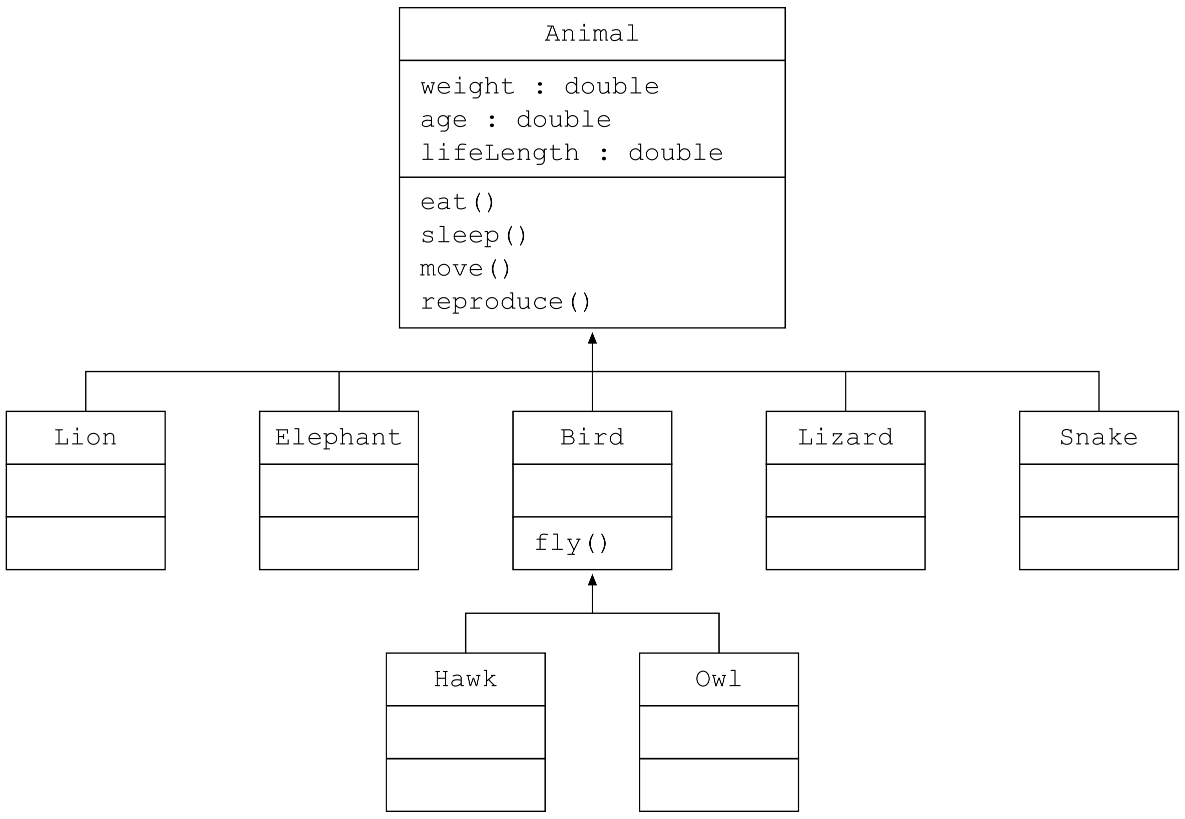


Figure 2.45 Extending the Animal class

With the above class diagrams, you can see that the `Animal` class defines all attributes and the four basic behaviours of all. The `Bird` class defines the common behaviour `fly` for `Hawk` and `Owl`.

The above diagram shows the relationships among classes, usually known as the class hierarchy. The `Animal` class that defines the common attributes and behaviours is considered to be a *general* class and the others are considered to be *specific* classes. The relationship among classes `Bird`, `Hawk and Owl`, `Bird` is considered to be the general class of the `Hawk` and `Owl`; `Hawk` and `Owl` are regarded as the specific classes of `Bird`. Furthermore, `Bird` is a specific class of the general class `Animal`.

In a class hierarchy, an arrow is drawn from the specific class pointing to the general class. It clearly shows the general class and all its specific class. Therefore, you can easily determine that classes `Hawk` and `Owl` have the attributes and behaviours of the `Bird` class and indirectly have the attributes and behaviours of the `Animal` class.

A specific class has all the attributes and behaviours of its general class. If the specific class needs a behaviour to be performed differently, it can define it in its own way. For example, the way an elephant eats is significantly different from the way a lion does. Therefore, the classes `Elephant` and `Lion` can define their own behaviour `eat()`. The other behaviours can perform in the way a general `Animal` does.

If you are given several classes, you can determine whether some are general classes of the others by studying their behaviours and attributes. If some of the classes share common behaviours and attributes, the set of common behaviours and attributes can be extracted and obtained by a general class. If such a class does not exist in the given classes, you can create a new class for such purposes.

The benefits of determining the relationship among general and specific classes

Well, you should now understand that relationships exist between different objects (and classes). The significance is that if an object of the general class can accept a particular type of message, the object of any specific class (with respect to the general class) can accept the message as well and perform properly. But, what are the benefits of such relationships to software development?

The payroll calculation example illustrates such a benefit. We said that the classes `Clerk`, `Manager` and `Director` are the specific classes of the general class `Staff`. As the `Staff` class accepts the message `calculateSalary`, all specific classes accept it and perform according to their own behaviours. Therefore, for a software application that calculates payroll, we can use an object of the `Clerk` class to model real clerical staff. Similarly, we can model real managers and directors with objects of the `Manager` and `Director` classes respectively. Then, we can simply send the same message `calculateSalary` to every object, and they will calculate their own salaries. Isn’t that convenient?

If the company has a new category of staff, say, Engineer, and its salary calculation method is different from the others, does it affect the payroll calculation? With the object-oriented programming paradigm, the effect is minimal. It can be achieved by modelling real-world engineers using the class `Engineer` as a specific class of the general class `Staff`.

Now, there are four categories of staff in total — clerk, manager, director and engineer. When you want to calculate the payroll, you send the same message `calculateSalary` to each of these objects. You are sure that every object will calculate its own salary, and it is not necessary to change the operations in other parts of the software system.

The final benefit can be illustrated by the bank accounts case study. If there is a new bank account type that supports overdrawing the account (with a limit), it means that the account balance can be negative. Such an account type is basically a savings account, but its `withdraw` operation is acceptable provided that the overdrawn amount is less than the preset limit.

You can see that most of the operations except the `withdraw` behaviour of a `SavingAccount` class are the same. Therefore, you can design a new class `OverdrawSavingAccount` as a specific class of

`SavingAccount` by providing a modified `withdraw` operation. That’s all! Most of the behaviours of `SavingAccount` have been tested and you can be sure that the behaviours — except `withdraw` of the `OverdrawSavingAccount` — are free of error. Such an approach can greatly improve the reliability of the software.

Self-test 2.2

For each of the following lists, identify the general classes and the specific classes. Draw diagrams to show their relationship.

1 car, racing car, jeep

2 facsimile machine, photocopier, telephone, electrical appliance

Summary

In this unit, we have discussed why you need a systematic approach to develop software applications because of the increasing complexity of software. Software developers may take months or even years to develop a software application, and the whole process involves many tasks and sub-tasks. The tasks can be grouped into six phases: requirement, analysis, design, implementation, testing and maintenance. We discussed the tasks involved and the goals to be achieved in each phase.

Object-oriented software development not only involves the use of object-oriented programming language, it also promotes an object-oriented approach to analyse the problem and design the solution. After the requirement phase, you have a problem statement about the problem to be solved. Based on this problem statement, you can derive the involved objects (and hence classes) and their attributes and behaviours using the object-oriented approach. In a few words, you identify the nouns and noun phrases and determine which of them are classes and which are attributes. Then, you inspect the verbs and determine the related behaviours. Afterwards, you can model the scenario with the identified objects. The entire operation is carried out by message-sending among objects and performing the behaviours accordingly. Then, you can derive the class designs in the design phase and finally implement them by writing class definitions in the Java programming language in the implementation phase.

Later in the unit, we discussed the *benefits* of developing software applications using the object-oriented approach. You can use a systematic way to assign the role(s) of each object and their behaviours. Inheritance enables programmers to extend the software applications by extracting common behaviours and behaviours to minimize duplicated code, and to develop new programs based on existing codes.

References

Microsoft Officials Brian Valentine and Iain McDonald Look Back on the Development Effort Behind Windows 2000,
http://www.microsoft.com/presspass/features/1999/12-15w2krtm.asp

Glossary

Analysis phase	The phase during which the problem statement is analysed to determine the list of roles of those who will use the software application, the list of required functionalities and the list of business entities involved.
Design phase	Based on the findings obtained from the analysis phase, software developers further investigate how to implement them with an object-oriented programming language. The classes are further enhanced to suit implementation considerations, such as performance.
Implementation phase	In the implementation phase, the derived conceptual classes are implemented as program codes using the chosen object-oriented programming language.
Maintenance phase	When a software system is developed and tests thoroughly free of errors, it is released to the users for actual operation. The maintenance phase is the period when the users are using the software system, and the software developer may be required to maintain it if the users require feature enhancements or if errors are determined.
Problem statement	A document that clearly specifies the requirements and the constraints of the software application to be built.
Requirement phase	This phase denotes the period when the software developers collect the requirements and constraints from the software users or clients.
Software development cycle	It denotes the whole period from the beginning when the decision to build the software application is made, to the end when the software application is released to the users. It includes a sequence of phases: requirement, analysis, design, implementation and testing.
Testing phase	The testing phase denotes the period when the software developers uncover the errors of the software application before releasing it to the users or client.

Suggested answers to self-test questions

Self-test 2.1

Based on the problem statement, we can identify the nouns in it:

> Each class has a student list.
>
> To set the mark of the subject Chinese for a student, the student is located in the student list by the student number, and the mark of the subject Chinese is set for the student.
>
> The operations for setting the marks of subjects English and Mathematics are similar.

According to the identified nouns, the following table is prepared:

Objects or classes	**Class name**	**Remark**
Class	`Class`	
Student List	`StudentList`	
Mark of the subject Chinese		Attribute of the Student class
Student	`Student`	
Student number		Attribute of the Student class
Operation		Not modelled as it denotes a behaviour
Marks of subjects English and Mathematics		Attributes of the Student class

Then, we can identify the verbs in the problem statement:

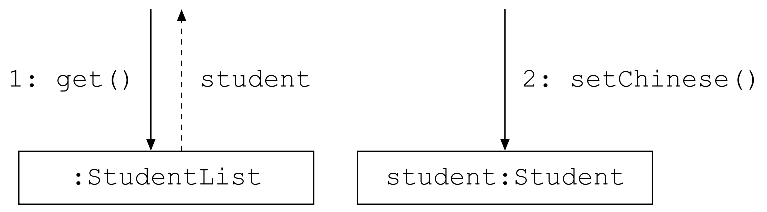
> Each class ***has*** a student list.
>
> To ***set*** the mark of the subject Chinese for a student, the student is ***located*** in the student list by the student number, and the mark of the subject Chinese is ***set*** for the student.
>
> The operations for setting the marks of subjects English and Mathematics ***are*** similar.

To model the scenario of setting the mark of the subject Chinese, the following diagram message sending (or behaviour) sequence is determined:

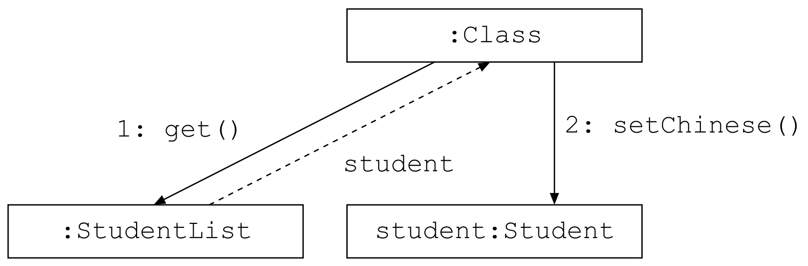
A message is sent to the StudentList object to get a Student object.

A message is sent to the Student object to set the mark of the subject Chinese.

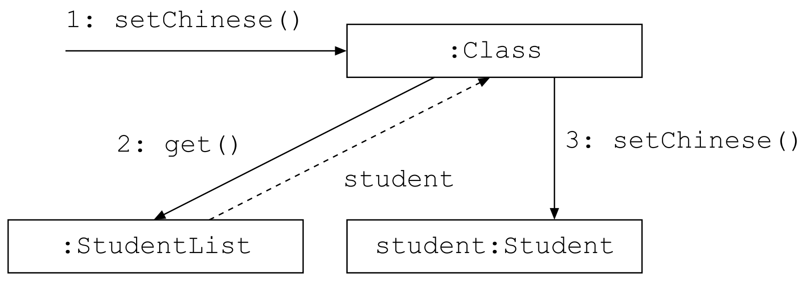
Based on this finding, we can derive the following diagram:



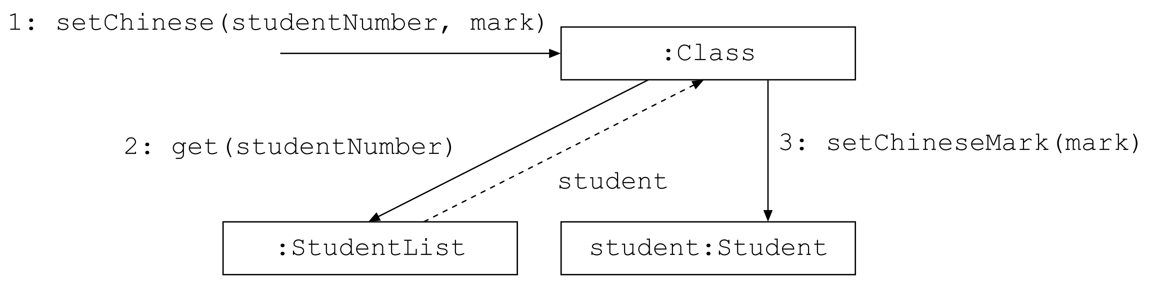
The message-sending diagram is disjointed, which means it needs an object to coordinate such message sending. Through further studies, you might find that it is preferable to choose the `Class` object to play the role of coordinator. Therefore, the diagram is enhanced to be:



As the `Class` object will never initiate itself to start the behaviour, it must receive a message so that it subsequently sends messages to the `StudentList` and `Student` objects. Therefore, the diagram is updated to be:



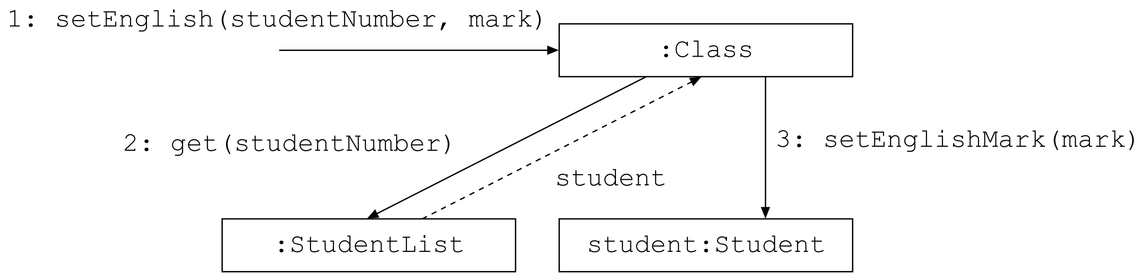
The `Student` object is located through a student number; it requires a mark to be set for the Chinese subject. Therefore, the `get` behaviour of the `StudentList` object needs supplementary data for ‘student number’, and the `setChinese` behaviour of the `Student` object needs supplementary data in the form of `mark`. As the `Class` object has to provide these two data items to the `StudentList` and `Student` objects respectively, it needs to get these two data items from the message it receives. Therefore, you can enrich the diagram with the supplementary data required:

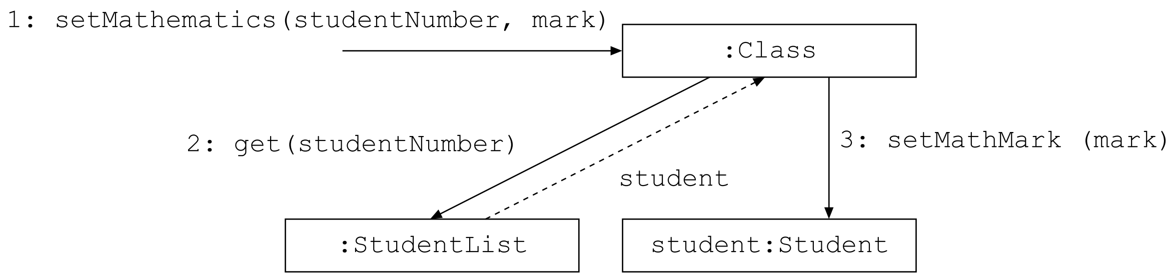


After verification, the sequence of behaviours presented in the above diagram can complete the required operation.

The behaviour of setting the marks for the subjects English and Mathematics is pretty much the same, except that it is necessary to request the `Student` object to set the English or Mathematics subjects instead of Chinese. Therefore, a different message type `setEnglish` and `setMathematics` is used. Similarly, the `Class` object needs two more message types so that it can tell which subject is concerned.

Therefore, the diagrams for setting the marks of subject English and Mathematics are:





That is the end of the analysis phase. You can now derive the class designs. Similar to the `StaffList` object mentioned in this unit, `StudentList` can be substituted by `Map` provided by the Java programming language software library. Therefore, it is only necessary to derive the class definition of `Class` and `Student`.

Based on the table of identified nouns, you can identify the attributes of the class `Class` and `Student`.

`Class`

`Student`

`chineseMark`
`englishMark`
`mathMark`

Then, we have to provide types for the attributes of the `Student` class. The marks are determined to be of type `double`. The `Student` class becomes:

`Student`

`chineseMark : double`
`englishMark : double`
`mathMark : double`

As the `Class` object always sends messages to the same `StudentList` object, an attribute for the `StudentList` is added to the `Class` class. The type of the attribute is `Map` as it can perform as a `StudentList`.

`Class`
`studentList : Map`

Now, you can add a behaviour to a class if its object can receive that type of message. Based on the three operation diagrams, you can enhance the class designs as:

Class
studentList : Map

setChinese(studentNumber, mark)
setEnglish(studentNumber, mark)
setMathematics(studentNumber, mark)

Student
chineseMark : double englishMark : double mathMark : double

setChineseMark(mark)
setEnglishMark(mark)
setMathMark(mark)

Finally, you have to specify the type of supplementary data and the feedback of the behaviours, if any. Students are assumed to be numbered using whole numbers, such as 1, 2, 3 and so on. Subject marks can be any number with or without a fraction. Therefore, the student number and mark are determined to be of type `int` and `double` respectively. Then, the class designs become:

Class
studentList : Map

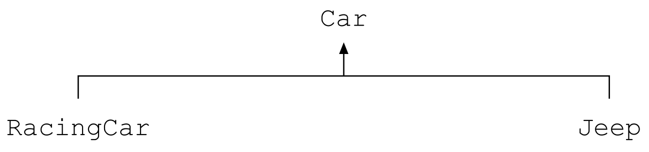
setChinese(studentNumber : int, mark : double)
setEnglish(studentNumber : int, mark : double)
setMathematics(studentNumber : int, mark : double)

Student
chineseMark : double englishMark : double mathMark : double

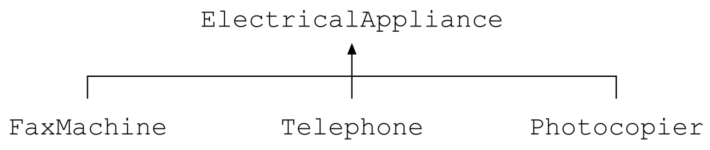
setChineseMark(mark : double)
setEnglishMark(mark : double)
setMathMark(mark : double)

Self-test 2.2

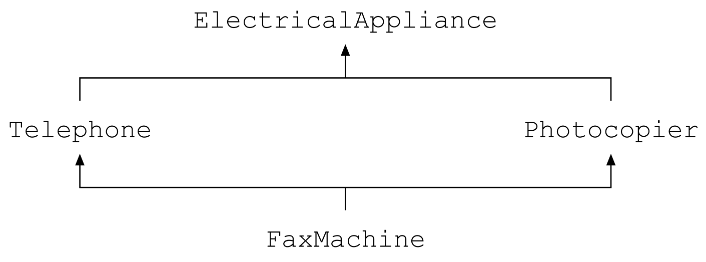
1 For the classes `Car`, `RacingCar` and `Jeep`, `RacingCar` and `Jeep` are the specific classes of the general class `Car` because a 'racing car is a car' and 'a jeep is a car'. Therefore, the class hierarchy is:



2 Facsimile machine, telephone and photocopier are the specific class of the general class electrical appliance, because they are all specific types of electrical appliance.



However, it is interesting that a facsimile machine usually has the functionality of a telephone, and one can consider it as a telephone with additional features. Similarly, a facsimile machine usually has a copying function like a photocopier. Therefore, facsimile machine can be considered as a specific type of telephone and a specific type of photocopier. You can use the following class hierarchy to represent such a relationship:



Such a relationship is usual in the object-oriented paradigm. *Unit 7* discusses this issue.