Session 7. Unified Modeling Language

In this session we examine the Unified Modeling Language (UML), an industry standard which specifies the graphical notation used for modeling an object-oriented application. Importantly, the UML standard does not define how the notation is to be applied – there is no description or discussion of analysis and design techniques that the notation should be used for. The UML is commonly described in the context of The Unified Process, the object-oriented methodology that the UML was originally developed for, however the two are separate, and the UML is often used with other techniques and for other purposes.

Session Objectives

In this session, you will learn:

- The history and motivation for the UML; and
- How to prepare the following UML diagrams: use case diagram, class diagram, state machine diagram, activity diagram, and interaction diagrams.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*Although we do not directly address this ULO, an understanding of object-oriented concepts is required to successfully apply the UML, which is used to express a visual representation of object-oriented applications from various aspects, and a study of UML often helps develop an understanding of the bigger picture of object-oriented development.
- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*
  Although we do not directly address this ULO, the application of UML throughout the object-oriented analysis and design process makes the content of this section a key input to the ability to solve programming problems using object-oriented techniques.
- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*
  Although we do not directly address this ULO, the UML is often used both to provide the specification of an object-oriented application but also to document such applications, a skill we will explore in the assessment of the unit.
- *ULO 4. Construct object-oriented designs and express them using standard UML notation.*
  This session encompasses the coverage of the UML notation in this unit and is clearly critical content to the achievement of this ULO.

Required Reading

There are no required readings for this week's content, however the prescribed textbook (Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014) does include some discussion of UML in the following sections:

- pp10, The UML (Unified Modeling Language);
- pp112, UML Class Diagram for Class Gradebook;
- pp121, Section 4.6: UML Class Diagram with a Property;
- pp145, Sequence Structure in C#;
- pp147, Sections 5.4-5.7, and pp191, Sections 6.4-6.6, includes discussion of the UML activity diagram;
- pp407, Section 11.2, addresses inheritance in UML class diagrams.

Required Tasks

The following tasks are mandatory for this week:

- Task 7.1. Use Case Diagrams
- Task 7.2. Class Diagrams
- Task 7.3. Sequence Diagrams

7.1. Introduction

The history of object-oriented development extends as far aback as the 1960s. The 1980s however was the time when the processes object-oriented analysis and design were developed substantially, resulting in three popular alternative approaches: the Booch Method by Grady Booch, the Object Modeling Technique (OMT) by James Rumbaugh, and Objectory by Ivar Jacobson. In the 1990s, these three authors began working for one company, Rational, which has since been acquired by IBM. At this time, the authors began working on combining their approaches into what is now known as The Unified Process. To allow the models of the Unified Process to be expressed, the authors began work on a standard notation, known as the Unified Modeling Language (UML).

It is important to understand the difference between the Unified Process and the UML. Where the Unified process is a complete methodology for conducting object-oriented analysis and design, the UML is only a standard graphical notation that can be used for the Unified Process, but also for other object-oriented analysis and design methodologies, and/or for other data modeling purposes (modeling databases being a common example). It is important to understand this distinction when examining textbooks on the UML, which often describe the Unified Process at the same time without clearly distinguishing the two.

The UML defines both a graphical notation, for illustrating the analysis and design of a system, and the Object Constraint Language (OCL), which defines a language through which it is possible to indicate constraints/conditions upon a (class) model. OCL is not mandatory however, and the UML allows any text to be used for constraints, thus we do not examine it here.

Although the UML is a very detailed notation for data modeling, there are a number of gaps in the specification. For example, when considering class diagrams we highlight the lack of support for properties (see Section 7.3, below). UML however is extensible, and users of UML are both permitted and encouraged to extend the notation wherever required[1]. The notation that we show below however, unless indicated otherwise, follows the UML specification strictly. Note however that we do not cover every aspect of the UML notation, there are many books dedicated to this purpose, but we cover enough to be adequate for most purposes.

7.2. Use Case Diagrams

UML use case diagrams are useful for capturing the functional requirements of a system, i.e., what functionality does the object-oriented application need to provide? When considering use case diagrams it is necessary to first consider what a use case is, even though the syntax and format for a use case itself is actually not defined by the UML. A use case is a collection of scenarios describing an interaction with the system, either between a user and the system, or between one system and another system. These scenarios are closely related by the goal of that interaction.

For example, a use case could describe a customer attempting to withdraw cash from an ATM, as shown below. In this use case, we can identify the main success scenario, where money is successfully withdrawn and no errors occur. In addition to the main success scenario are a number of extensions, which describe variations to the main success scenario that can occur. Many more extensions could be shown than indicated, e.g., the ATM card jamming or being unreadable, timers expiring for entering input/taking cash, and so on. Complex steps can also be broken down further as necessary.

Withdraw Funds

Level: User Goal

Main Success Scenario:

1. Customer inserts ATM card.
2. Customer enters correct PIN.
3. Customer selects withdrawal option.
4. Customer fills in withdrawal amount.
5. System authorises withdrawal.
6. System dispenses cash.
7. System returns ATM card.

Extensions:

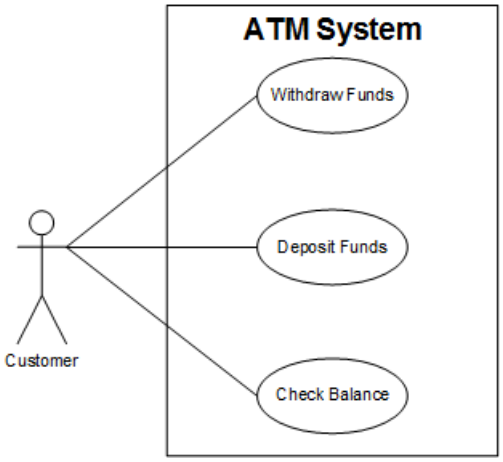
[1] It is also worth noting that some applications, including Microsoft Visio and Microsoft Visual Studio, include diagramming functionality that has similarities to UML, however the notation is not maintained. Given that these diagrams are not strictly UML, they cannot be used for assessment purposes in this unit. Note that Microsoft Visio can still be used either by manually drawing the UML diagrams and/or with the assistance of UML aligned templates/stencils, such as those available from http://softwarestencils.com/uml/index.html

2a: Customer enters incorrect PIN
.1: Customer may re-enter PIN number
5a: System fails to authorise withdrawal
.1: Customer may re-enter withdrawal amount or cancel transaction

The above use case shows the following elements:

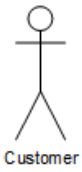
- Withdraw Funds – the title/name of the use case;
- Level – each use case is typically categorised into one of three levels[2]:
  - Summary level – taking several user goals to complete, e.g., booking/planning a trip;
  - User Goal level – a user interaction with a single goal, e.g., book a room;
  - Subfunction – partial satisfaction of a user goal, e.g., searching for a hotel.
- The main success scenario – a list of steps conducted in a successful execution of the scenario;
- Extensions – a list of one or more alternative scenarios, whereby the step number matches one of the steps in the main scenario (e.g., 2a is the first extension/alternative for step 2 of the main success scenario), and steps (.1, .2, etc.) are shown for the extension/alternative to replace the step in the main success scenario.

With an understanding of what a use case is, it is now possible to describe use case diagrams. Use case diagrams are similar to a contents page for use cases, an example of which is as follows:

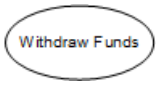


There are three main elements in a use case diagram:

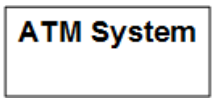
[2] There is also a five level categorisation of use cases as well: cloud level (a very high level summary), kite level (equivalent to summary), sea level (equivalent to user goal), fish level (equivalent to subfunction), and clam level (very low level), however this is more detailed than required for the purposes of this unit.



An actor, which is a role that a user or another system plays in the use case.



The use case, for which the title is shown.



Finally, the system boundary is shown, which identifies the scope and responsibilities of the system being modelled.

A line is then drawn from the actor/s to the use case/s that they are involved in. One can be involved in one or more use cases. Similarly, one use case can be used by/involve one or more actors.

Task 7.1. Use Case Diagrams

Objective: Use cases and use case diagrams are very useful tools for identifying the requirements for a system. Using common knowledge about an application provides a good starting point to begin modelling an object-oriented program.

Consider the development of an application for the electronic submission of assignments. Using Microsoft Visio and/or other software for drawing diagrams, prepare use cases and a use case diagram (or diagrams) for the following goals:

- Creation of a submission box by a lecturer;
- Submission of an assignment by a student; and
- Retrieval of submitted assignments by a lecturer.

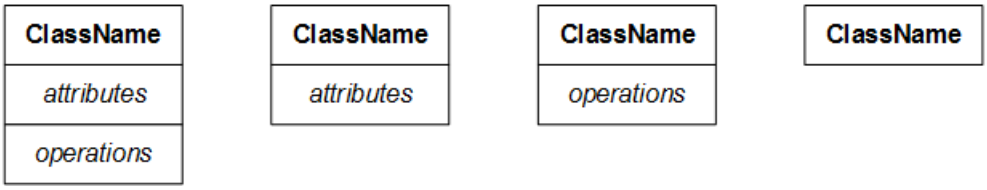
Note: The UML template in recent versions of Microsoft Visio does not follow the UML standard and should not be used for this task. You may choose to draw the diagrams manually or with the assistance of UML aligned templates/stencils, such as those available from http://softwarestencils.com/uml/index.html. Regardless of which software and/or template you choose to use, the diagrams you produce should be similar to those appearing in the workbook.

7.3. Class Diagrams

The UML class diagram is by far the most common UML diagram, and is often the type of diagram referred to when someone mentions “a UML diagram” in conversation. The UML class diagram shows the structure of a system being developed by illustrating the classes and the relationships between the classes in the system[3]. Classes are represented in UML by a box which is divided into three sections: the

[3] Visual Studio is able to generate "class diagrams" for some or all of the files in a programming project. As noted before, these diagrams have similarities with the UML standard, however vary enough to not be acceptable for this unit.

class name (required, in bold), attributes, and operations. If the class is also an abstract class, the class name should also be in italics. The attributes and operations sections are optional, and do not need to be shown. Whether they are shown or not will usually depend on the context of the diagram and/or how advanced the system design is. The four possible structures for a class drawn in UML notation are as follows:



The syntax for attributes is as follows:

[*visibility*]*name*[: *type*][*multiplicity*][= *default_value*][{*property*}]

The following elements are visible in the above syntax:

- *visibility* – indicates the visibility (access modifier) of the attribute using one of four symbols:
  - + for public;
  - # for protected;
  - – for private; or
  - ~ for package (such as an assembly).
- *name* – the name of the attribute;
- *type* – the data type for the attribute (simple type or custom type);
- *multiplicity* (see below) – optional, indicating how many instances the attribute refers to (usually one unless referring to a collection);
- *default_value* – optional, an equals symbol (=) followed by the attribute's default value; and
- *property* – optional, surrounded by braces ('{' and '}'), indicates any additional properties about the attribute, e.g., readOnly.

Importantly, although many of the above elements are indicated as optional, as much information should be specified as possible, e.g., more information will be available towards the end of the design phase than is available at the early part of the analysis phase. Constants are not clearly defined by the UML, however given that constants usually have capitalised names, have a value, and cannot change, the following example demonstrates the syntax commonly used:

+ MATH_PI: double = 3.14159 {readOnly}

The syntax for operations are as follows:

[*visibility*]*name*([*parameter_list*])[: *return_type*][{*properties*}]

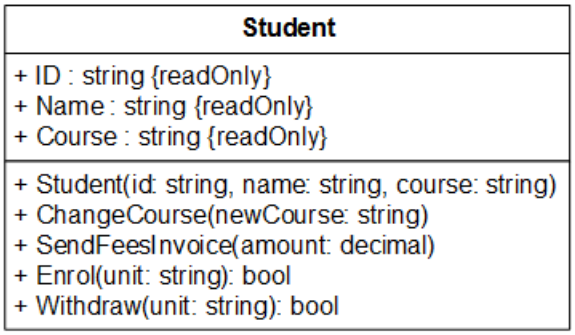
In the above syntax the parenthesis ('(' and ')') are mandatory. As for attributes, static operations are indicated by using an underlining the entire operation. As for classes,

abstract methods are indicated by using italics. The *visibility* and *properties* elements are the same as for attributes. The remaining elements are as follows:

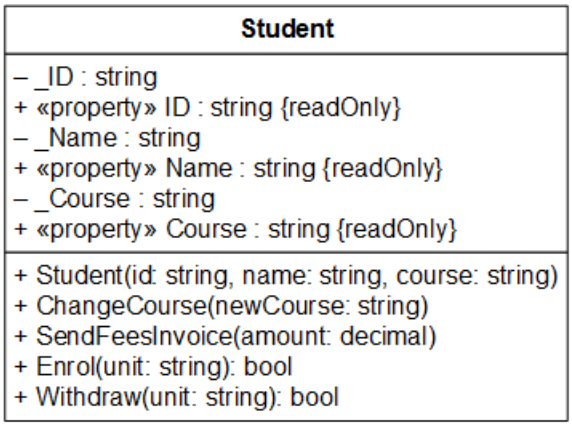
- *name* – the name of the operation;
- *parameter_list* – optional, the parameters to the operation use a similar syntax to attributes, as follows:
  [*direction*]*parameter_name*[: *type*][*multiplicity*][= *default_value*][{*property*}]
  with the addition of the *direction* field which represents input parameters (in), output parameters (out), or reference parameters (inout);
- *return_type* – indicates the data type for any value (simple type, custom type, or blank for no return value);

The UML notation also defines notation for abstract and static classes and members. Abstract classes and members are indicated by using *italics* text for the class name and member name, respectively. Simiarly, static classes and members are indicated by underlining the text for the class name and member name, respectively.

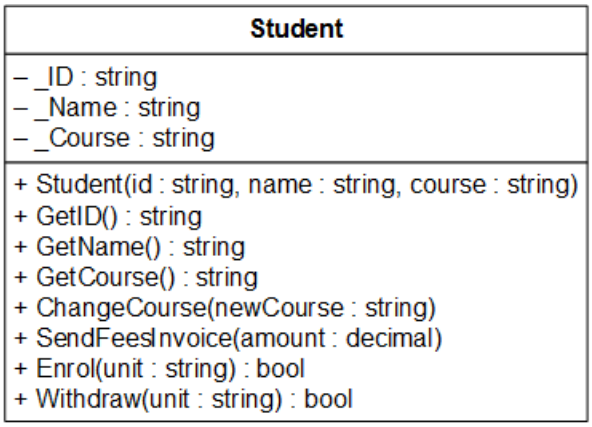
An example class following this notation is as follows:



You will notice in the above figure that the attributes are all indicated as public, although they are marked with the property readonly. If we consider how we implement attributes using C# (see Section 2.5), we should be using a private instance variable and a public property. The UML however is not specific to any particular programming language and does define a notation for properties. However given that the UML is extensible, the authors of the prescribed textbook (Deitel and Deitel) have extended the UML to allow for properties, as follows:

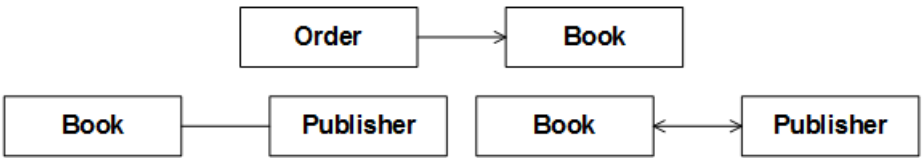


We will use this notation for showing properties. If the programming language does not support properties, accessor and mutator methods are shown the same as other operations, as follows:



Importantly however, UML diagrams do not need to match any particular language. This is quite clear when UML diagrams are prepared as part of the analysis phase of software development – programming language may not have even been considered at this time!

An object-oriented application does not only consist of classes however, we must still consider how to indicate relationships in a UML class diagram. An association is indicated by a straight line drawn between two classes. Navigability for the association is optional, indicated by using a feathered arrowhead. Bi-directional associations are shown by drawing arrowheads at both ends of the line or by not indicating any arrowheads[4]. Examples of association notation are as follows[5]:



The first of these examples is an Order class which has an association with a Book class (an Order of a Book). This is uni-directional association, where you can navigate from the Order object to the Book object, but not vice-versa. The second example is then the relationship between a Book and a Publisher, which is bi-directional, indicated using both a simple straight line and secondly by a double-ended arrow.

Consider the first example more closely. We can indicate the name of the attribute/property for this association by writing the name next to the line. The name should appear at the target of the association, i.e., close to the Book class. We can also indic-

[4] Recall that a line without arrowheads is also used where navigability is not indicated, so be careful using this notation.

[5] Only class names are shown in these example figures for simplicity. Completed class diagrams would usually show classes with attributes and operations panels included.

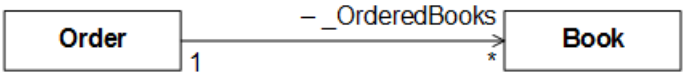
ate ***multiplicity*** (also known as cardinality), which follows the same rules as for attributes, above, as follows:

- Multiplicity is indicated as a range of values separate by two periods, i.e., *start..finish*;
- Either single values can be indicated or an asterisk ('*') can be used to indicate any non-negative number (zero or more when shown on its own);
- If the same value appears for the *start* and *finish*, only one value needs to be shown, i.e., instead of '1..1', just show '1'; and
- Several different multiplicities can be separated by commas.

Examples:

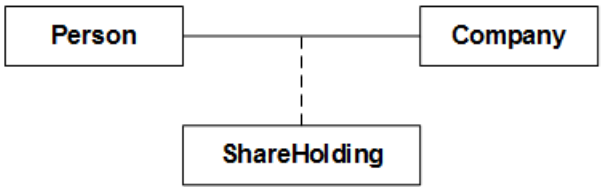
- 1
- 3
- 0..1
- 2..*
- 1,2,4,5..*

By adding the name and multiplicity, we can update our example as follows:



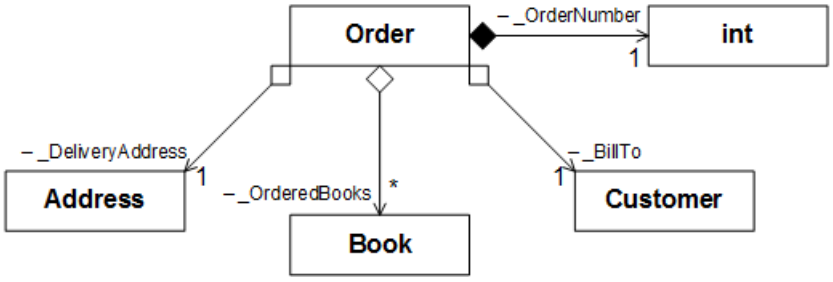
UML class diagrams also provide notation for the concept of an ***association class***. An association class is used for an association relationship when it is not clear whether one or more attributes should be added to the class that defines the client object/s or the class that defines supplier object/s. For example, consider modelling two classes, a Person and a Company class, where the association is used to represent a Person investing in the Company as a shareholder. Placing the number of shares held in the Person class does not make sense - a Person does not have a number of shares without having a reference to the Company. Similarly, a Company would not keep shares divided into such small amounts without having a reference to the (Person) shareholder/s. As such, the number of shares is directly relevant to the association itself, rather than either of the two classes.

Association classes are indicated for association relationships in UML by the addition of a third class (the association class), which is then connected to the line indicating the association by a dashed line, as follows:



Aggregation and composition relationships are then shown through the addition of a diamond next to the aggregate class (the 'whole'). For aggregation, the diamond is

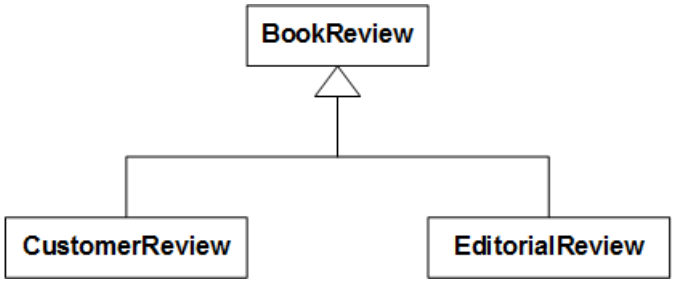
unfilled. For composition, the diamond is filled in. Examples of aggregation and composition relationships are as follows:



There are two points worth highlighting in this example. Firstly, you will notice that no multiplicity is shown next to the aggregate classes. This is because the multiplicity is almost always 0 or 1 and only needs to be shown if it is particularly unusual. Secondly, notice that the integer data type ('int') is shown as a class instead of an attribute here. This demonstrates how data types, including both simple/fundamental and complex/user-defined data types, can be shown in UML class diagrams either as separate classes or as types for attributes. Usually simple data types, e.g., integer types, and other built-in types that developers would be familiar with, e.g., DateTime, would be shown using attribute syntax, where as user-defined custom types would be shown as classes in diagrams. However note that the display of data types can also vary depending on the context, e.g., in a large system the UML class diagram may be divided onto several pages, and a user-defined type may be indicated on one page as a full class (with attributes and operations illustrated), whilst other pages may only show the class name, and finally other pages may only indicate the type using attribute syntax.

Due to a lack of detail in the UML specification, it is very difficult to tell the difference between the semantics for aggregation versus an association. As such, it is often recommended that associations be used in preference to aggregation, unless there is a very specific need for aggregation to be shown. Composition however can be used where there is clear ownership of the objects of one class over another. In C#, the only data types where this occurs are value types, which would usually be indicated as attribute types anyway.

The last class relationship, inheritance, is referred to as generalisation in UML class diagrams. Generalisation is shown similar to an association, with an unfilled triangle appearing next to the base class, as follows:



Note that there are two arrows indicated here that merge together close to the BookReview class. This is not required, and the arrows can be drawn separately if preferred/needed.

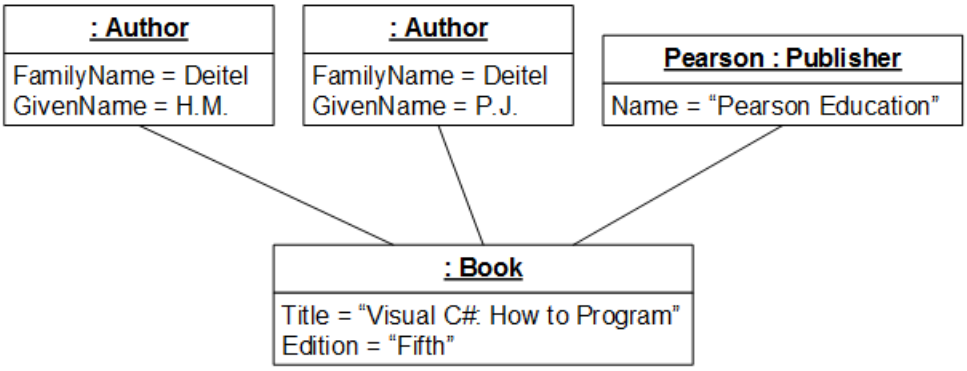
Object Diagrams

UML ***object diagrams***, sometimes also referred to as instance diagrams, are a sub-type of the UML class diagram. The purpose of an object diagram is to clarify particularly complex UML class diagrams by illustrating a snapshot of the resulting objects and the links between them at run-time. Several object diagrams can be provided for the same class diagram to illustrate different scenarios and/or ways in which the classes will be instantiated.

As a sub-type of the UML class diagram, the object diagram shares the same basic notation with the following differences:

- The class name takes the form '*instance_name* : *class_name*', which are then underlined, e.g., John Doe : Person
- Both the instance name and class name are optional, but the colon should always be shown with the class name, i.e., John Doe: Person, John Doe, or : Person;
- Actual values are shown for attributes, rather than classes, and you do not need to show all attributes; and
- Operations are not shown.

The following is an example diagram for the textbook we use in this unit:



Task 7.2. Class Diagrams

Objective: The UML class diagram is by far the most common diagram you will see when working with object-oriented applications and is also used in many other fields such as for modelling the tables in a database. As such, it is critical that you master this diagram's notation.

Consider again the development of an application for electronic assignment submission. Using Microsoft Visio and/or other software for drawing diagrams, prepare a

class diagram that illustrates how the assignment submissions would be recorded in a working system. You should include the following:

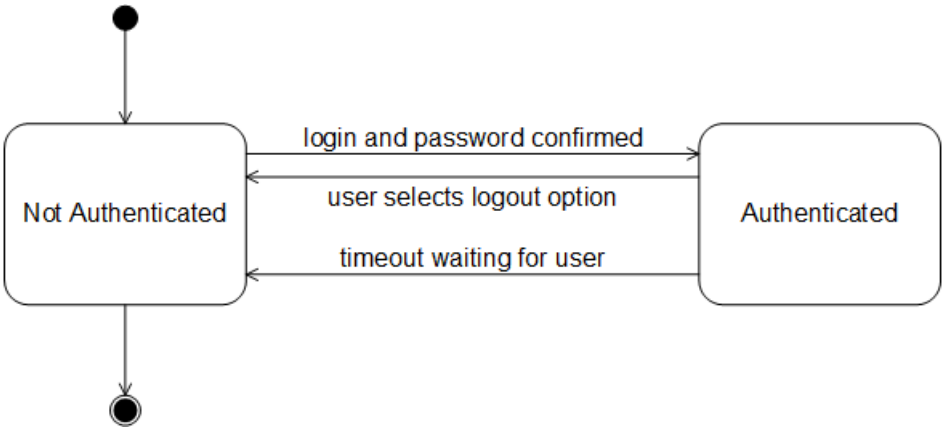
- Person, Student, and Staff classes in an inheritance hierarchy;
- Unit, Assignment, and Submission classes;
- Relationships as appropriate between these classes; and
- Attributes and operations you feel would be appropriate for such as system.

When preparing your class diagram, think very carefully about how the code would be implemented to access the relevant data and consider the navigability that would be required between the various classes to achieve the required functionality.

Note: The purpose of this task is for you to gain experience with preparing UML class diagrams, not to prepare a multi-volume design for a working system. Add necessary detail to your model for it to be relatively realistic, but don't go over-board.

7.4. State Machine Diagrams

Many computer systems can be modelled as simple state machines. For example, consider the Deakin Studies Online (DSO) web application that we use for managing and disseminating information for this unit. In this application we can see two obvious states in the system: not-authenticated and authenticated. We can also identify the transitions that allow us to move between these two states: logging in (transition from not-authenticated to authenticated state), and logging out (transition from authenticated to not-authenticated). The following shows a UML ***state machine diagram*** for such an application:



This figure shows a number of elements:

- States are illustrated as rectangles with rounded corners with the name of the state in the middle of the rectangle;
- Transitions are shown as lines with a feathered arrowhead, with the trigger condition for the state transition appearing next to the line; and
- Two pseudo-states indicate where the state machine begins and ends: the initial pseudo-state is shown as a filled-in circle and the final pseudo-state is the

same, except a further circle that has not been filled in is drawn around the outside edge.

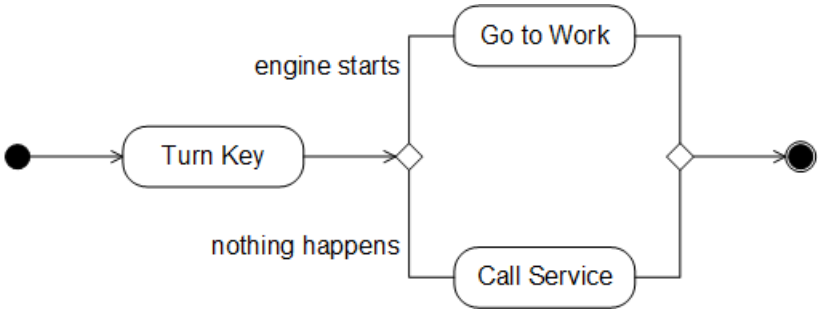
There is much more detail in the UML specification for state machine diagrams than we present here, however the above notation is adequate for our purposes in this unit.

7.5. Activity Diagrams

The UML ***activity diagram*** is very similar to traditional flow charts, except that the possibility of operations being executed in parallel is introduced. Parallelism refers to the ability of certain sequences of code to be executed at the same time without interfering with each other. For example, consider the development of an application managing the fulfilment of orders placed on a web site. The purchased goods can be prepared for dispatch at the same time as the customer's credit card is charged and an invoice prepared. There is also then a need for synchronisation, whereby an ordering is imposed on certain events occurring, e.g., do not send the goods to the customer until they have been charged correctly/until they have paid. For this unit however, we are not concerned with parallelism.

UML activity diagrams can be applied to almost any type of ***behavioural modelling*** including both business processes and processing that occurs in software. Activity diagrams model an activity, which is broken down into one or more actions. Each action typically represents the behaviour or changes to program state caused by a single method invocation in the program, and are drawn in rectangles with rounded corners. The flow of execution that occurs between the actions is illustrated by a line connecting two activities with a feathered arrowhead. To indicate decisions (if statements or loop conditions), decision nodes are added at the point where the decision is made and also at the point where the control flows merge again, represented by unfilled diamonds. The start and end of an activity are indicated by a filled in circle and by a filled in circle surrounded by an unfilled circle, respectively.

The following is an example of a simple activity diagram:



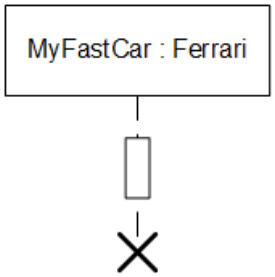
The diagram shows a simple scenario for starting a motor vehicle, where the key is turned in the ignition. If the engine is running, time to go to work. If nothing happens, call for service.

7.6. Interaction Diagrams

UML ***interaction diagrams*** show the exchange of messages that occur between objects in an object-oriented system to implement some behaviour within a context, e.g., the Person turning the key to start the Vehicle's engine. The UML defines two different interaction diagrams: UML ***sequence diagrams*** and UML ***communication diagrams***. Both diagrams show the same basic information, except sequence diagrams emphasise the order of events whereas communication diagrams emphasise the structure and connection of objects in the system. In this document we focus on sequence diagrams. Communication diagrams are also generally considered to be less expressive than sequence diagrams, hence we do not consider them further.

Each sequence diagram shows a number of participants consisting of objects and/or classes (if static methods are used). Each of these participants is drawn as a box with a dashed line appearing underneath it, known as the lifeline and representing the life of the object during the interaction. The name of the participant is shown in the box, using a similar notation to that of object diagrams (see Section 7.3). Unlike the object diagram notation however, the participant name is not in bold or underlined.

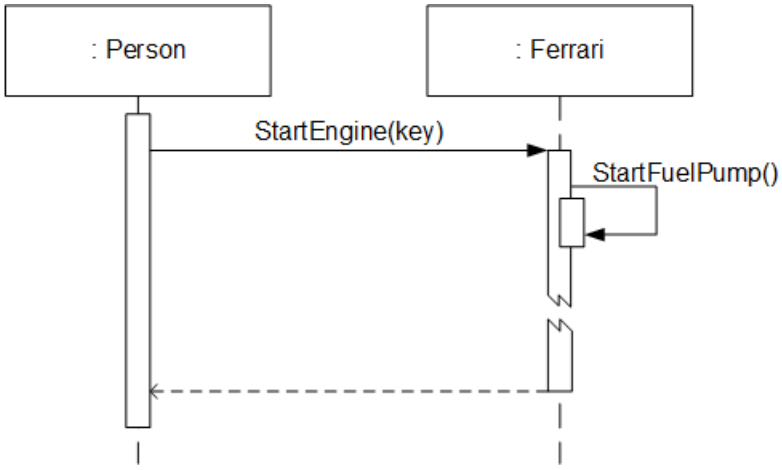
The following is a simple example of a participant in a sequence diagram:



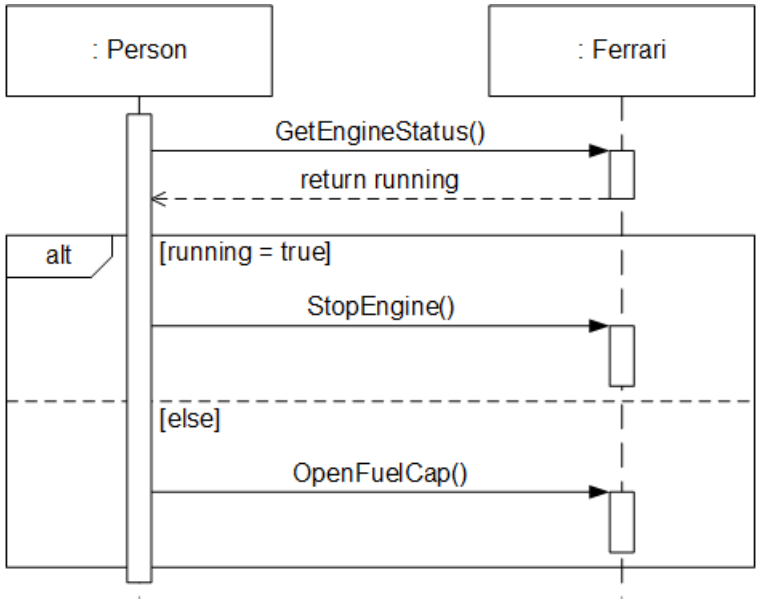
The figure shows an object called MyFastCar of type Ferrari. The clear rectangle that is drawn over the lifeline is known as an activation bar, which indicates when a method of that object is mid-execution. The activation bar is optional according to the UML specification, however it helps to emphasise the behaviour and interaction between the objects so is worth while. If the class/object calls itself. If necessary, the activation bar can be broken to hide unnecessary detail, e.g., if the detail is shown in another diagram. Finally, the destruction of the Ferrari object is shown by a large cross that terminates the lifeline.

Messages exchanged between objects are shown by arrows with a solid arrowhead, pointing to the object receiving the message (the supplier object). The name of the method invoked, and any parameters are shown as annotations to the message. Messages can also be sent to the same object, resulting in an additional activation bar is shown on top of the existing activation bar. The order in which the messages are sent is top to bottom of the diagram, i.e., the first message sent appears at the top of the diagram and the last message appears at the bottom. If the messages are synchronous, e.g., invoking a method, the return of control to the calling object can optionally be indicated by a dashed line with feathered arrowhead. Note however that unless some

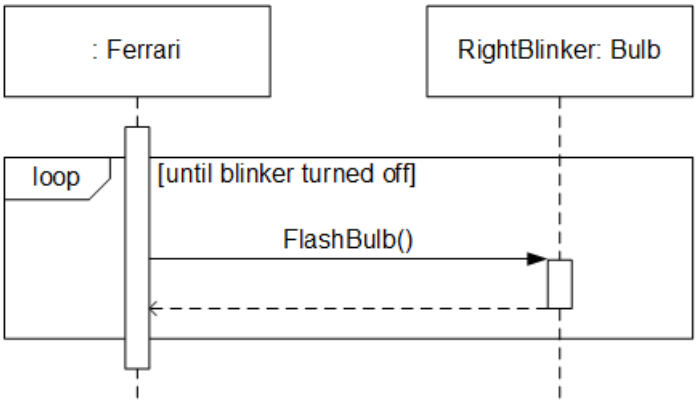
data is being returned these lines are often left out. These concepts are shown in the following figure:



Conditional checks resulting in one methods or another being invoked are indicated as shown in the following figure. An interaction frame (rectangle) is added to the diagram surrounding the alternate code paths with a the word ‘alt’ (meaning alternatives) appearing in the top left corner in a rounded box. Alternate code paths are separated by a dashed line/s. The guarding conditions are then indicated in square brackets ('[' and ']').



Finally, iteration is shown using a similar notation except the word ‘loop’ appears instead of ‘alt’, as follows:



Task 7.3. Sequence Diagrams

Objective: The expression of an architecture for an object-oriented system in a class diagram is only useful if the system can perform its function once built. In this task we test our earlier design expressed in a UML class diagram by preparing a UML sequence diagram that exploits the model.

Using Microsoft Visio and/or other software for drawing diagrams, prepare a UML sequence diagram that works with your earlier design (Task 7.2) to demonstrate how the classes interact when receive an assignment submission.

Note: In completing this task, you may need to modify your earlier design. This is a normal part of developing a design for an object-oriented application – like almost all aspects software development, success can only be achieved through an iterative process, i.e., prepare a design model, then test that model with realistic scenarios, making any necessary changes that result from the testing. Also note that your completed class diagram and sequence diagrams should also still match the use cases you developed in Task 7.1 as well, if not, which one of them is wrong?