Chapter 5. Object-oriented Analysis and Design

Table of Contents

Objectives

At the end of this chapter you should be able to:

- Create class models of systems that can be encapsulated in one diagram.
- Simplify a model using generalisation and abstract classes.
- Describe how a more complex model can be specified, and how modelling software maintains design integrity in such a complex system.

- Read and describe class models created by other designers.
- Be able to relate class diagrams to equivalent program outlines.
- Specify an object model using CRC cards.
- Describe how objects change over time.

Introduction

This chapter discusses object-oriented modelling methods, that is, the representation of a software system in terms of classes and their inter-relationships. Class modelling is the most fundamental aspect of object-oriented analysis and design, and its mastery is crucial for anyone who intends to use object-oriented techniques. Class modelling is useful to both the analysis and design disciplines: in analysis, classes and the relationships between them will specify the problem. In design, it specifies the software.

Class modelling is philosophically complex, but practically straightforward. However, it rests on a few important philosophical concepts, and it is vital that these concepts are mastered before doing any practical work. When students fail to make progress with real design exercises, it is commonly because they have not developed a proper understanding of the basic principles. These principles cannot be learned by rote, they must be understood through exercise and application.

Class modelling is used throughout object-oriented development, from analysis through to implementation and testing. The class model is the “skeleton” on which the “flesh” of program code is constructed. During the development of a system the class model is modified and refined, and often several different models will be used for the same software system, each model seeing the system form a different standpoint, as will be discussed further below.

Modelling standpoints

Class modelling is a flexible technique with a number of applications. In this it should be contrasted with use case modelling; although the purpose of use case modelling is not specified in the UML, in practice it is usually employed as part of a requirements elicitation exercise (as discussed in previous chapters). Class modelling, on the other hand, is used throughout an object-oriented analysis and design process, from requirements engineering through to programming.

The designer should have in mind at each stage of development what the class model is being used for. Neglect of this principle leads to poor design work. If you believe you have produced a class model that represents “the system” (whatever the system happens to be), you are quite likely mistaken. A single class model at best represents a particular *standpoint* on the system.

Design standpoints represent the different uses we make of modelling. We can identify three obvious standpoints.

- A *logical design standpoint*: the model shows how a system should function logically, or how its components interact in a logical sense.
- A *specification standpoint*: the model shows how the system is broken into manageable components, and how those components interact.
- An *implementation standpoint*: the model shows how the system is structured, or how it will be structured in the future.

Although these standpoints are not completely distinct, they do embody differences in how the software system should be modelled. Specifically, you should see that analysis models will be more of a logical standpoint, while design models will, ultimately, be an implementation standpoint.

As design progresses the standpoint of the designers will change. Initially — especially if the system to be developed is replacing some other system or manual process — the designers will be concerned mostly with how the system is currently structured. This existing structure may well not be the most efficient or logical, but it is necessary to understand the operation and context of the existing system before designing a replacement. Later, in the design process, the designers will be concerned with the logical structure of the system. That is, regardless of how an existing system may be implemented, the designers consider what the simplest, most effective, logical organisation of the new system is. Still later, the designers will need to refine the model into a design that can be implemented. Here they will be concerned with specification and implementation standpoints.

The UML provides a set of modelling symbols and terminology and defines the concepts that they embody. The first job of the novice designer is to learn this notation and come to understand the concepts. The UML does not specify how these concepts are to be employed. In other words, it does not provide a different set of symbols for logical and implementation standpoints; it is up to the designer to ensure that his or her intentions are clear.

Classes and objects

This section discusses the nature of classes and objects. We begin with the philosophical background of classification and move onto descriptions of objects and classes that will be useful for software and system design.

Classification

The idea of *classification* is not a new one; philosophers have been studying the concept for at least two thousand years. We classify things all the time: we classify people by gender or by age; we classify motor vehicles by size, passenger capacity, and so on; we classify businesses into publicly and privately owned entities, amongst others. When we classify something we are saying that it has properties in common with others of its type. The idea is, at heart, a simple one. We assume that if we know something about the class, then that fact is true of all the members of the class as well. In writing a computer program that manipulates, say, customers' bank accounts, we need to be concerned first of all with the things that all bank accounts will have in common. We do not want to be concerned with individual customers unless there is something about them that is different to the class as a whole. This is a profoundly important and simplifying principle.

By being members of a class it is assumed that an object shares the general properties of the class. For example, all members of the class “human being” have certain properties in common; although there are a great many different human beings in the world, they are sufficiently similar that there should be no difficulty distinguishing between any human being and, for example, a cheese sandwich.

Classification in object-oriented design

In object-oriented design, we classify all the objects that a system should know about. That is, we specifying classes that collectively describe all the behaviour of the system. Although we speak of object-oriented design, we are mostly interested in *classes*, rather than individual *objects*. **Class modelling** is the process of assigning classes, and describing their inter-relationships.

Class modelling works in software design because it allows complex systems to be described in such a way that the complexity becomes manageable. And all interesting (and most useful) software systems will be complex.

The “things” that a software system will process, recognise and serve are, by definition, objects. Users of the system, keyboards, screens, bank accounts, stock items, printouts, and so on, are all objects. By classifying these objects, that is, by creating and managing classes, the designers of the system impose some control on the complexity of the system. Although a complex system may manage many millions of objects, it will frequently manage orders of magnitude fewer classes, perhaps fewer than one hundred.

A definition of *class* and *object*

For the purposes of object-oriented modelling, an object can be defined as follows:

> An **object** is a self-contained entity with well-defined, recognisable properties (*attributes*) and behaviour (*operations*). It can interact with other objects.

This leads to the following definition of a class:

> A **class** embodies the properties and behaviour that a collection of objects have in common.

Examples of classes

Having presented this philosophical discussion, you may be expecting classes themselves to be philosophical. In fact, they are usually not. Suppose we are designing a system to manage the work of a lending library. Some of the classes that will need to be considered are:

- Book
- Member
- Reader
- Borrower
- Loan
- Fine
- Barcode
- Return date

Of course, on more detailed inspection these may turn out to be inappropriate, and no doubt many other classes will have to be considered, but these initial ideas are appropriate classes because:

- they are self-contained with well-defined behaviour and properties
- each will have some instances (objects) that have a great deal in common
- it is obvious that each has an important role to play in the system

A further key point is that some classes may in fact have sub-classes. For example, “Reader” and “Borrower” may in fact be types of “Member”.

Some thoughts on the relationship between classes and objects

Here are some concepts to consider when thinking about classes and objects:

- In English we use the words “is a” to mean several different things. For example, in “Rover is a dog” we are saying that Rover is an instance of class dog. When we say “A dog is a mammal” we are saying that “dog” is a sub-classes of “mammal”, that is, all members of class dog share properties with members of class mammal. Therefore, in describing the relationship between classes and objects, you should take some trouble to use unambiguous terms like “is an instance of” and “is a subclass of”.

- A class may have any number of instances, including one and zero. Classes can be defined that have zero members in practice and in principle. Student are often particularly reluctant to use classes that have only one instance. The “Internet”, for example, is the one and only instance of class “Internet”. Again, this may seem odd, but *class modelling* is about *classes*, not *individuals*.

Concrete and conceptual classes

Previously we looked at some examples of classes that might be appropriate for a library computer system. You should be able to recognise an important sub-grouping of these classes. Classes such as “Book” and “Member” are physical or concrete classes; they correspond to “real” things in the physical world. However, the classes “Loan” and “Return date” are different “types” of class. They do not correspond to real, physical entities at all. We will refer to classes of this sort as “conceptual” classes. Many student mistakenly use the term “abstract” here; this should be avoided because “abstract class” has a quite different meaning, which will be explained later.

Some components of a software system have both concrete and conceptual representation. Consider the “Book” class: The conceptual book “War and Peace” by Tolstoy exists independently of the paper and binding of a real physical book; indeed the conceptual book may be said to exist if the only text in existence was in a computer's memory. In other words, in a sense the book exists outside of its physical representation. At the same time, a lending library loans physical, not conceptual, books. They are real entities made of paper, with barcodes for scanning, and so on.

This is not purely an academic distinction. Large, complex systems (particularly object-oriented databases) may be impossible to implement unless this subtle distinction is understood. For example, if the library's system only represents physical books, it has to store all information about each book in each instance of the “book” class. But if the library has, say, ten copies of each book, then most of this information is duplicated in each object, which is a waste of memory and can cause data inconsistencies. This problem always arises when there is a distinction between physical and conceptual representations of the same thing, and when there are multiple, similar physical instances of the conceptual item.

This is not an issue on which the novice designer should lose sleep; the majority of systems can be designed correctly even if the issue is not understood. However, it will improve the depth of your understanding if you take the trouble to ensure that the sense of this section is understood.

Note

In an implementation, conceptual and physical classes will have different names to one another. For instance, the conceptual class “Book” might be named “BookInformation”, while the physical class may retain the name “Book”.

Attributes

Attributes are the properties of a class; on the whole they are things that can be measured or observed. For example, an important attribute of the class Animal is “number of legs”. Different animals have different numbers of legs, but all animals have the property “number of legs”. Even a snake has a number of legs: it just happens to be zero. However, it would make no sense to talk about a plant's having “zero legs”. The attribute “number of legs” is not a valid attribute of plants.

Attributes are used rather vaguely in classification in general. For example, biologists distinguish between insects and spiders because insects have the attribute “has six legs” and spiders have the attribute “has eight legs”. There is a numerical difference here. However, one of the ways that biologists distinguish between reptiles and mammals is that reptiles are cold-blooded and mammals are warm-blooded. There is no numerical difference in this case.

In class modelling the nature of attributes is formalised. We can say that:

> An **attribute** of an object is a property that has a name, a value and a type. The name and type must be identical for all instances of a class, but the value may be different.

For example, suppose the class “Book” in our library example has attributes “title”, “publisher”, and “date of publication” (there will, no doubt, be others, but these will do for now). The attribute “title” has a name (“title”), it has a type (probably “text” or “string”). Objects of class “Book” will have values for this attribute, for example, “War and Peace”. The attribute “date of publication” has a name, and its type is “Date”. Again, objects of class “Book” will be published on different dates, but all objects will have a value for the attribute.

The important point here is that, although all instances of class “Book” have different values for these attributes, they all have the attribute, and they are all the same type. The date of publication of a book will always be a date, and never a height. The title will always be a piece of text and never a colour, and so on.

In some cases the values of attributes distinguishes one object from another. For example, if Rover is a brown dog and Fido is a black dog, we may reasonably say that their “colour” attributes are different. There is no chance of confusing Rover for Fido. However, common sense indicates that if we see two brown dogs, we cannot assume they are the same dog: there are many brown dogs in the world. Indeed, even if all the attributes of two objects are identical, this does not make the objects identical.

There is an interesting philosophical issue behind this; many students have argued (sometime fiercely) in classes that if we really knew all the attributes of a particular class, and two objects had all those attributes in common, then they would of necessity be the same object. For example, two brown dogs may be indistinguishable by colour, but the are distinguishable by position. If two putative objects occupy the same point in space, then they must be identical.

This is all well and good, but irrelevant for this subject. Whatever the philosophical contentions, the principle which object-oriented designers work to is that:

> Two objects are not equal, or identical, just because they have identical attributes.
> Objects are only **identical** to themselves, or things that refer to themselves.

For example, the person Mary Smith is identical to herself, and if Mary Smith is the world-record holder for (say) javelin throwing, then the object “Mary Smith” is identical to the object “world record holder for javelin throwing”.

Ignorance of this principle leads to very subtle problems when designs are translated into programs. For example, consider the following portion of a Java program:

```
String response = "HELLO".toLowerCase();
    if (response == "hello")
    {
    System.out.println("They are identical.");
    }
    else
    {
    System.out.println("They are NOT identical.");
    }
```

This Java snippet creates an object of class String and sets it equal to the lower-case text "hello". The program then tests whether the “response” is equal to the object "hello". Since both objects have the value “hello”, we might expect the program to consider them equal. But in fact the program will inform us that the two objects are not equal; the object "hello" and the object “response” have identical attributes, but they are not equal to each other. The correct way to test whether two String objects have equal attributes in Java is to write something like:

```
String response = "HELLO".toLowerCase();
    if (response.equals("hello"))
    {
    System.out.println("They are equal.");
    }
    else
    {
```

```
System.out.println("They are NOT equal.");
}
```

The **equals** method tests two objects to see if their attributes are equal. The **==** operator, on the other hand, tests whether the objects themselves are, in fact, the same object.

It is a key principle of object-oriented design that attributes should be simple. The technical term is **atomic**, meaning “not capable of division into smaller components”. If an attribute appears to have attributes of its own, then it is not an attribute at all, and is probably an object in its own right. You will find that programmers, of necessity, do not follow this rule. This is because it is standard practice to use classes to extend the functionality of a programming language. In Java, for example, numbers — such as 12 and 4.7 — are atomic, but text strings are represented as objects. In this case it is often necessary to make objects attributes of other objects.

Also, although “has six legs” may be an attribute of insects to a biologist, it will never be an attribute of any class in an object-oriented design. This is because it is not precise enough. A designer would say that the class “insect” has an attribute “number of legs” whose value, by default, is 6. We would write this in a formal way (as will be discussed later):

```
numerOfLegs:integer(6)
```

Having said that, in the initial stages of design, it is inadvisable to be too precise about attributes. In particular, usually the names are established first before the types and values. In some cases the types of attributes will not be determined until writing the program code; however, at this point it must be specified: many programming languages require the types of attributes to be fully specified.

In addition, during design there is no reason why the types of attributes have to match the types that are available to a particular programming language. For example, real numbers (that is numbers with fractional parts) are represented in Java by the types “float” and “double”. These are technical terms with no meaning outside of certain programming languages. In design there is nothing to prevent the use of attribute types “number”, rather than “float”, simply because it is easier for non-specialists to follow. The translation into programming terms can be made later in the development if necessary.

It is important to understand that an object's attributes belong to the object in some way, that is, they tend to be “private”. One object cannot automatically obtain, or change, the attributes of another. This is just another way of saying the objects are self-contained. Part of the process of object-oriented design is determining which of an object's attributes may be read and written by other objects, and which will remain private.

Terminology note

Other terms for “attribute” include “instance variable”, “member variable” and “member data”, and may differ between programming languages (Common Lisp, for instance, uses the term *slot*). Technically an “instance variable” is not exactly the same as an attribute, but the distinction is not important enough to worry about at this stage. Java programmers tend to use the term “instance variable” rather than “attribute”, but “attribute” is preferred by most designers and CASE tools.

Operations

In the last section we considered attributes, which say something about what a class is. *Operations* say what a class does (or can have done to it). When you have fully specified the attributes and operations of a class, then you have specified that class completely (some experts would say that one must specify the class's constraints as well, but that is largely beyond the scope of this course).

For example, the “Clock” class may have the operation “display time”. This is something that a clock does. It is not an attribute of class “clock”, because it does not tell us anything about objects of class clock that we do not already know.

At the design stage, we tend to assume that all of a class's operations are accessible to other classes, that is, classes can communicate by invoking each others operations. This is called “message passing”. As design progresses, we usually find that new operations have to be added that are private to that class, because they are only concerned with internal operation of the class. Remember that classes should be self-contained, and should expose as little as possible of their internal workings.

In the same way that attributes are formalised in object-oriented design, beyond what is true of classification in general, operations are also formalised. An operation has a name, a parameter list, and a return value, much like a function. The parameter list is (essentially) the data elements that will be supplied to the operation. The return value is a piece of data that can be returned to the object that invoked the operation. These issues are not very important in the early stages of design (we may only know the name of the operation), but become important as the design is translated into a program. Note that an operation has access to the object's attributes, and can read or write them, but normally has no access to the attributes of other objects.

Terminology

Other terms for “operation” include “member function”, “class function”, and “method”. Java programmers tend to use the term “method”.

Dynamic behaviour and state

A class model, taken alone, is a description of the static structure of a system; this means that it shows the system in a way that is independent of time. In the library system discussed earlier, the individual books may come and go, and individual members for that matter, but the classes “Book” and “Member” persist. To say that a lending libraries has books and members is true, independent of time.

Of course, in reality the system will change over time. This change is reflected in the values of the attributes of objects, and the numbers of instances of each class. The library may begin with 1000 books, and ten years later it may have 10 000 books; but there will never be a time when the class “Book” is not a valid one for the library. Similarly, the addresses and even the names of the members will change. This will be reflected in the values of the attributes of the instances of class “Member”.

The sum total of information carried by the attributes of an object is called its **state**. If we know the state of an object, then we know everything there is to know about it. This last statement is true by definition. If in describing an object the designer finds that its attributes are not sufficient to specify its state completely, then they have overlooked some attributes.

If the state of an object is given by the values of all its attributes, then the state of the system if given by the states of all its objects. In other words, the values of all the attributes of all the objects completely specify the state of the system at any given time.

If state is so important, why can it not be represented on a class model? The reason is that there are far better ways to represent changes in state. The UML provides state transition modelling for this purpose. In a full design it may be necessary to provide state transition diagrams for some or all classes, to describe their state changes in detail. The pattern of state change over time is called the **dynamic behaviour** of a system, and modelling this is called **dynamic modelling**. As well as state transition modelling, the UML provides *object interaction diagrams*, *activity diagrams*, *sequence diagrams* and *communication diagrams* for dynamic modelling. One can, of course, describe the dynamic behaviour of classes in plain language as well, and one probably should.

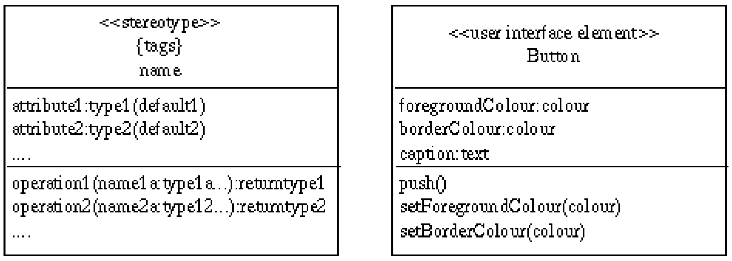
Only objects have state; classes *do not* have a state. This is because a class specifies properties that are true of all objects of that class. However, a class can have a *default* state. This is the set of attribute values that an object of that class will have by default, that is, if not provided with any other information.

UML notation and conventions

Symbol

The diagram below shows the UML symbol for a class formally (left) and as an example (right). The formal detail looks quite daunting, but you will see that most of the information is optional, and in practice a class symbol is quite straightforward.

Figure 5.1. The UML symbol for a class


<<stereotype>>
{tags}
name
attribute1:type1(default1)
attribute2:type2(default2)
....
operation1(name1a:type1a...):returntype1
operation2(name2a:type12...):returntype2
....
<<user interface element>>
Button
foregroundColour:colour
borderColour:colour
caption:text
push()
setForegroundColour(colour)
setBorderColour(colour)

The class symbol has three compartments. The top compartment shows the name of the class, any tags it has, and any stereotype it has. The stereotype of a class is a higher-level category to which it belongs. Stereotypes are used to provide additional structure to a model, and to make it easier to follow. Common stereotypes include «use case» and «actor». Most classes, in practice, will not need stereotypes. The designer is at liberty to create new stereotypes to organise the work, if required. The only tag that we will consider in this module is {abstract}, which will be discussed later.

The middle compartment contains the class's attributes. Each attribute can specify a name, a type and a default value. Only the name is compulsory.

The bottom compartment contains the class's operations. These can be specified in full, with parameter lists and return values, but very often only the name is given. To distinguish operation names from attribute names, it is conventional to write an operation name with pair of brackets after it, like this: push(). If the operation has a parameter list, then it can be written in the brackets, but, again, this is not compulsory.

The example shows a possible class diagram for a “Button” class. By button we mean a little rectangle that appears on the screen, and responds to mouse clicks, as with a computer GUI rather than a physical button. In this example, the button has the stereotype «user interface elements». It is intended that this stereotype be applied to all classes that represent user interface elements, like list boxes, menus, windows, and so on. Again, this is not compulsory, but it helps structure the model. There are no tags associated with this class, so none are shown.

The “Button” class has a foreground colour and a border colour, and these can be changed. So we have defined two attributes to store the colours. These attributes have type “colour”. Most programming languages do not have an inbuilt “colour” data type, and at some point the programmer will have to translate this type name into something that the programming language will understand. This is not an issue for the early stage of design. It is clear what “colour” means, and this is the important factor.

The operations of the class are “push()”, meaning simulate what happens when a button is pushed (probably the screen appearance will change), and two operations to change the colours. Why can another class simply not change the attributes to change the colour? It is vital to understand that a class should be, as far as possible, self-contained. It is up to the class to decide whether its colours can be changed or not, and by which other objects. The operations to change that colour will probably change the values of the colour attributes, but it is up to the implementer of the class to decide how this will happen. Another class does not need to know the internal operation of the “Button” class.

The operations that set the colour each have a parameter list with one parameter: the colour to be used.

You will see that the example class is very much simpler than the formal specification. In practice, it is common for classes to start of with very little detail (no parameters, no types), and to be “filled in” as design progresses.

A note on software

The class diagrams were created using a commercial package, *Select SSADM*. Various software packages may support the UML notation to a greater or lesser degree, and so diagrams made using different packages will differ from one another.

Naming conventions

The UML makes certain recommendations about how names of things are to be written, with varying degrees of force. We recommend that you follow the UML naming conventions as if they were all compulsory, because most professional designers do, and you will want your work to be understandable by your colleagues. The naming conventions are also in line with the Java naming conventions, and should be familiar.

- Names of classes are written with an initial capital letter (like this: Button, not like this: button) and have no spaces. To show where a space would be, the next letter is capitalised (like this: BarcodeReader, not like this: Barcode Reader).
- Class names are singular, not plural. It is incorrect to name a class “Books”. Although there may be many book objects, there is only one book class.
- Names of attributes and operations start with a lower-case letter, but may have capital letters to indicate where the spaces would be, if the name would normally have spaces.
- Names of operations are followed by brackets () to distinguish them from names of attributes

The prohibition against spaces arises because most programming languages do not allow spaces in their naming convention, and it is modern practice for the final model diagrams to be as close to a program as possible. Some, but not all, CASE packages will enforce these conventions.

Finding classes

The customers of a software engineering project will not think of their work in terms of classes; it is the job of the developers to determine suitable classes with which to structure their work. This job requires experience and intuition, and invariably improves with practice. Here are a few general hints:

- The names of classes are usually nouns. A good place to start is to take a written description of the system and select all the nouns. You will undoubtedly have to remove some classes and add others, but this method provides a place to start.
- It is better to have too many classes to begin with, rather than too few. You can always remove some if you find that they are redundant
- Classes should, as far as possible, correspond to entities that the clients would understand. Computer terms rarely make good class names. You will find that as development progresses you will need to add classes to represent implementation factors, but you should not start off with these. Classes should make sense to non-technical people, at least in the early stages.
- If something has sub-types, or it is a type of something else, it is probably a class
- If something is important, but does not appear to be a class, it might be an attribute of some other class

As you add detail to a design you will discover new classes, and you can add these to the model. You will also often find that you can simplify a design by adding classes, as will be shown.

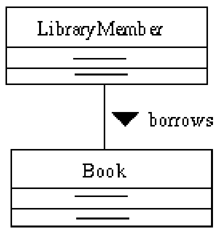
At all stages of development you should challenge your class model and be prepared to modify it if necessary. It is important to remember that iterative, incremental and evolutionary methods will have you update not only requirements, but the software design (and hence class model) for multiple development iterations. This is especially important for agile process methods.

Relationships between classes

Finding and describing classes is the first stage of class modelling. The next stage is to determine, and show, how classes are related.

In the simplest terms, a relationship between two classes is shown by a line (as it is in an entity relation diagram):

Figure 5.2. A relationship between two classes



In general such a relationship is referred to as a *link*, *association*, or a *collaboration*.

In the diagram above, the classes are shown with horizontal dashes where the operations and attributes normally go; this is an indication that there are some operations and attributes, but they are not shown on this diagram. It is also permissible to show a class simply as a rectangle, but then the reader does not know whether the class has attributes and associations and they are not shown, or if it does not have any at all.

The line connecting the two classes has an arrow to indicate the *direction* of the association. Objects of class “LibraryMember” borrow objects of class “Book”, not the other way around. This is fairly obvious from the context in this example, and the arrow could have been omitted, but if you are working on a system which is less obvious it does well to include the arrows. Note that links should have names, although you will not always want to show the name on the diagram (to reduce clutter). If you cannot assign a sensible name to a link, it is probably not a real link, or the classes at each end are incorrect. Normally a link should be able to have a relatively simple, short name, which will probably be a verb.

The UML is strict about the positions of arrows. It would be incorrect to draw the arrow in the figure above on the line. Arrows on lines have a different meaning altogether.

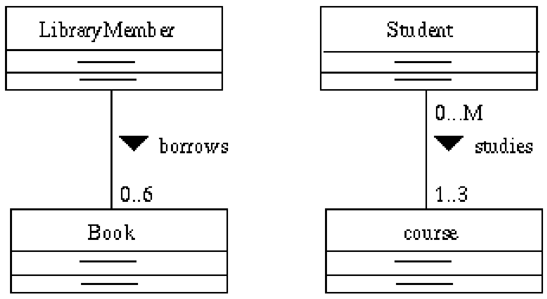
The links shown above is an example of a “named association”; it is the most general kind of link. The UML also recognises a number of special-purpose links, which will be described later.

Specifying relationships in detail

If a link is drawn as a simple line, it is assumed to be a *one-to-one association*. In the previous example this would mean that each member of the library can borrow exactly one book, and each book can be borrowed by exactly one member. This is probably not correct. It is good practice to indicate on each end of the link its **multiplicity**, that is, the number of objects associated with the link. It is particularly important to distinguish between “1” and “many” in multiplicities.

We could re-draw the previous example like the figure on the left:

Figure 5.3. Indicating multiplicity



This indicates that at any given time a “LibraryMember” is involved with a “borrows” association with an object of class Book, that is, a library member can borrow up to six books. At the same time, a book can only be on loan to one member.

The figure on the right is an example of a *many-to-many relationship*. A student can be studying one, two, or three courses at a particular time. Each course can be studied by an indeterminate number of students. The symbols “M” stands for “many”, and means “some unspecified number”. The asterisk (*) can also be used to stand for “many”. Note that the UML distinguishes a *many* that may include zero from a many that does not. In the “students” example, the number of students enrolled on a course varies from zero up to many, not from one up to many.

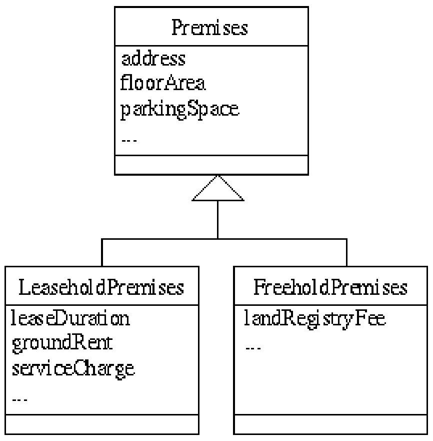
The multiplicity symbols are an example of what the UML specification calls **end ornamentation**. End ornaments are symbols that can be used on the ends of associations to enhance their meanings. Another common end ornament is the **navigability arrow**. This is an arrow drawn on (not alongside) the line and indicates the direction in which the class structure can be traversed for information.

Inheritance

We indicated earlier that object-oriented modelling recognised some specialised types of associations. The first, and most important, of these is the generalisation-specialisation association. You will have encountered this previously in when studying use case modelling; it is now time to examine this relationship in more detail.

As a reminder, the symbol for a generalisation-specialisation relationship is shown below:

Figure 5.4. Generalisation-specialisation represented in the UML



The arrow points towards the most general class. In this example, we are considering part of a system to be used by an estate agency for assisting in finding suitable premises for small businesses to purchase. The class “Premises” represents any premises. Its important attributes are the address, floor size, type of use permitted, amount of parking space, price expected, and so on, but these are not all shown in the diagram for reasons of lack of space (ellipses are used to indicate this). There are two important sub-classes of premises: leasehold premises, and freehold premises. These have all the same properties of “Premises”, and some additional ones. For example, leasehold premises will naturally have an address and a size, but they will also have a lease duration, a ground rent, and some others. Similarly, freehold premises will have properties that are not applicable to leasehold premises, or to premises in general.

It would be an error to show the attributes “address”, “floor area”, and so on, in “LeaseholdPremises” and “FreeholdPremises”. These classes will automatically have these properties, as they will inherit them from “Premises”. The same mechanism applies to operations: any operations of “Premises” will apply also to its sub-classes and should not be shown again in the diagram except if the sub-classes do different things in these operations.

There may also be sub-classes of “LeaseholdPremises” and “FreeholdPremises”. These will inherit the properties of all their super-classes. So classes can form a complex hierarchy. When a sub-class inherits properties from a class other than its own base class, this is called **indirect inheritance**.

Terminology

A variety of different terminology is used to denote different parts of a generalisation-specialisation relationship. Using the above figure as an example, we could say:

- “Premises” is a generalisation of “LeaseholdPremises” and “FreeholdPremises”
- “Premises” is the base class of “LeaseholdPremises” and “FreeholdPremises”
- “Premises” is the super-class of “LeaseholdPremises” and “FreeholdPremises”
- “LeaseholdPremises” and “FreeholdPremises” are sub-classes of “Premises”
- “LeaseholdPremises” and “FreeholdPremises” are specialisations of “Premises”

Multiple inheritance

It is possible for a class to be a sub-class of more than one base class.

For example, “Radio” and “Television” are sub-classes of “ElectronicDevice”. “Television” and “Car” are sub-classes of “TaxableProperty”. “Television” inherits properties of “ElectronicDevice” and of “TaxableProperty”. “Radios” are not taxable, and therefore cannot be a sub-class of “TaxableProperty”. “Cars” are not electronic. This is a clear example of multiple inheritance. This example is quite artificial, to illustrate a case where it is difficult to avoid multiple inheritance. In many cases it will be possible to restructure a model to avoid it. Note that some programming languages (including Java) do not allow multiple inheritance, so many designers avoid it as well.

Abstract classes

An **abstract** class is one that can have no *direct* instances.

An abstract class has no direct instances logically, or by definition. But a class is not abstract merely because it happens to have no instances. For example, the class “PersonWhoCanRunAMileInLessThanThreeMinutes” happens to have no instances at present, but there is no logical reason why this should be the case. No doubt careful use of performance-enhancing drugs and twenty years of selective breeding could cause this class to be instantiated. For a class to be abstract it must be logically impossible, not physically impossible, that it be instantiated.

In addition, a class is not abstract merely because it does not correspond to a real-world entity. This mistake is commonly made by novices; it is not unusual to see beginners label classes like

"BankAccount" or "Transaction" as abstract because they represent non-physical things. The correct term for these classes, as mentioned earlier, is *conceptual*.

As an example, consider a computer system that manages information about certain products that a business manufactures. The bulk of information about a product (e.g., price, delivery time, warranty duration) is encapsulated in a class called "Product". However, we may choose to create classes to represent the specific products we produce, and have them as sub-classes of "Product". In this case, by definition there are no direct instances of class "Product", but there will be instances of the sub-classes of "Product". Because "Product" is never directly instantiated, it is an abstract class.

What is gained by doing this? In short we have gained a measure of simplicity. By gathering most of the important information about the products into a single class ("Product") we have removed the need to duplicate this information in the different sub-classes of "Product".

Making "Product" abstract has two important benefits. First, the reader of the model will know immediately that "Product" has a particular role in the system, that of providing structure and simplification. Second, the compiler that generates the final program will know that there can be no direct instances of class "Product". This means that (i) it does not have to use memory to store the details of the instances, because there are none, and (ii) it is able to prevent the programmer making some trivial mistakes. Consider this definition in Java:

```
abstract class Product
    {
    //... details go here
    }
```

If the program attempts to create an object of this class later, it can only be because the programmer has made an error. The compiler will flag any such attempt as a compile time error.

An abstract class can have abstract operations. These are operations that are specified, but not implemented. It is up to the subclasses to provide a full implementation of the abstract method.

An abstract class with no-subclasses is more than likely useless. If you find this on your models you have probably made a mistake.

It is possible to indicate that a class is abstract in a class diagram by adding the {abstract} tag beneath the class's name.

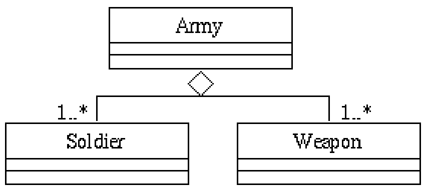
Aggregation and composition

All object-oriented designers use generalisation-specialisation relationships all the time. They are vital to good design. The relationships we will consider in this section are defined by the UML, but it is not generally agreed that they are a special kind of association at all.

Aggregation

Aggregation is a general term for any whole-part relationship between objects. Its symbol is a white diamond, like this:

Figure 5.5. The representation of aggregation in the UML

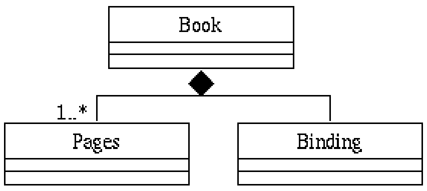


This diagram can be read as “an Army is an aggregate of one or more soldiers and one or more weapons”. The aggregation is expressing a loose whole-part relationship between the army and its constituents, where the constituents are not necessarily members of the aggregate. For example, if the army as an entity ceased to exist there would still be soldiers (they would simply be unemployed). This topic has lead to some very heated debate. In practice the distinction between a computer system which works, and a system which does not, is highly unlikely to hinge on whether the designers used aggregations or named associations. An aggregation can usually be replaced by an association called “has” without much loss of clarity.

Composition

It is not clear that composition is more expressive than a simple association labelled “contains”. **Composition** expresses a whole-part relationship where the “parts” are definitely contained within the “whole”.

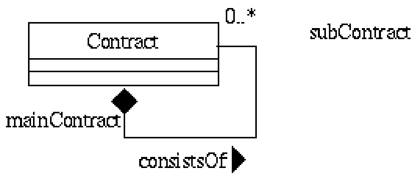
Figure 5.6. The representation of composition in the UML



This example can be read as “a book consists of one or more pages, and one binding”. Note the distinctions between this and the previous example. First, the binding and pages physically comprise the book; it is impossible for the binding and pages to be in one place and book in another. Second, destruction of the book would imply destruction of the binding and pages.

Self-association and roles

So far we have shown how classes can enter into associations with other classes. However, it is allowable — and often essential — to model a class as being associated with itself. For example, suppose we are designing a system for managing contracts, and we want to show that a contract can consist of a number of small sub-contracts. Sub-contracts have exactly the same properties as contracts. We could represent this as follows:



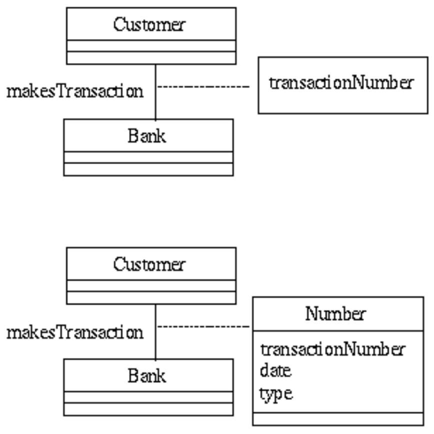
This example says that one object of class “Contract” consists of zero or more objects of class “Contract”. To clarify the relationship, it is customary to use role names. Where an association name describes the overall nature of the association, a role name is attached to the end of an association, and describes the role played by the object at that end. If the example above, one object plays the role of “main contract” while zero or more others play the role of “sub-contract”. Because role names are so important in these cases, some CASE tools will report an error if they are omitted.

Link classes and link attributes

In many cases each instance of an association has properties of its own. It is sometimes helpful to show this in the model for increased expressiveness. In addition, it gives us a way to show that multiple classes are involved in the same association.

A link attribute simply assigns a value to a particular instance of the link, that is, for each pair of objects that are associated there is a particular value of the attribute. For example, in the top diagram of the figure shown below:

Figure 5.7. Link attributes



“transactionNumber” is a link attribute. This model says that the bank and its customers enter into transactions, and each transaction has a particular number.

However, the association may be too complex to be represented by a single attribute. In this case we can use a link class. With a link class there is one instance of the class for each pair of objects that are associated. The link class can have any number of attributes, and can quite legitimately enter into relationships of its own with other classes in the system. In the example above, the link class “Number” defines each transaction carried out by the bank with its customers. It has attributes to indicate the nature and date of the transaction, as well as the number. Link classes can also have sub-classes. For example, it may be useful to indicate that there are different types of transaction by means of sub-classes, rather than simply having an attribute called “type”.

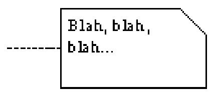
Link classes are a very powerful and expressive tool, but do take some experience to able to use effectively.

Constraints and notes

Notes

As in use case modelling, it is often very helpful to be able to annotate a diagram to show particular features. The notation is exactly the same:

Figure 5.8. The notation for notes



You can, and should, apply notes wherever you think they will be helpful to the reader, but they should not be viewed as a substitute for proper documentation. A class model is not complete until all classes, attributes, operations, and associations, have been documented. Notes are useful for pointing out specific details on diagrams.

Constraints

Constraints are additional information about a class that are not associated with any particular attribute or operation. A typical constraint might say, for example, that one attribute will have a value which depends on some other attribute. It is customary to write simple constraints below the class in braces, {like this}.

Constraints are becoming increasingly important in object-oriented modelling, as there is an increasing interest in formalising the technique. A scheme called the *object constraint language* is under development to support formalised constraints, but this is beyond the scope of this course.

Class-Responsibility-Collaborator cards

Class-responsibility-collaborator cards (CRC cards) are not a part of the UML specification, but they are a useful tool for organising classes during analysis and design.

A CRC card is a physical card representing a single class. Each card lists the class's name, attributes and methods (its *responsibilities*), and class associations (*collaborations*). The collection of these CRC cards is the *CRC model*.

Using CRC cards is a straightforward addition to object-oriented analysis and design:

1. Identify the classes.

2. List responsibilities.

3. List collaborators.

CRC cards can be used during analysis and design while classes are being discovered in order to keep track of them.

CRC cards have various benefits, which you might notice makes them very amenable to iterative and incremental process models, especially agile ones:

- They are **portable**: because CRC cards are physical objects, they can be used anywhere. Importantly, they can easily be used during group meetings.

- They are **tangible**: the participants in a meeting can all easily examine the cards, and hence examine the system.

- They have a **limited size**: because of their physicality, CRC cards can only hold a limited amount of information. This makes them useful to restricting object-oriented analysis and design from becoming too low-level.

Class *responsibilities* are the class's attributes and methods. Clearly, they represent the class's state and behaviour. *Collaborators* represent the associations the class has with other classes.

CRC cards are useful when the development of classes need to be divided between software engineers, as the cards can be physically handed over to them. A useful time to do this is when classes are being reviewed, for, say, determining whether they are appropriate in a design.

From model to program

During object-oriented development, the models will tend to become more detailed, and there will be a shift in standpoint from a logical to an implementation view. Developers using the waterfall or similar process models feel that there should be a strong correspondence between the final model and the program code it results in. However, iterative and agile methods often eschew heavy amounts of documentation, and realise that models (including object models) may not always accurately reflect the code, since both will always be changing as the code-base develops, and as the customers' requirements develop over time.

For agile methodologies it is important to remember that documentation is secondary to the software being produced. Slavishly attempting to specify every last component of the software in a detailed object model is antithetical to agile process methods, and there are many published studies showing that such complete, upfront designs are both inadequate (since requirements, and hence designs, continuously change) and detrimental to software engineering as a whole.

Dynamic behaviour

The object model which we have just been discussing is a *static* model of classes and their relationships. It does not show how the classes call on one another to perform the functions required in the software system.

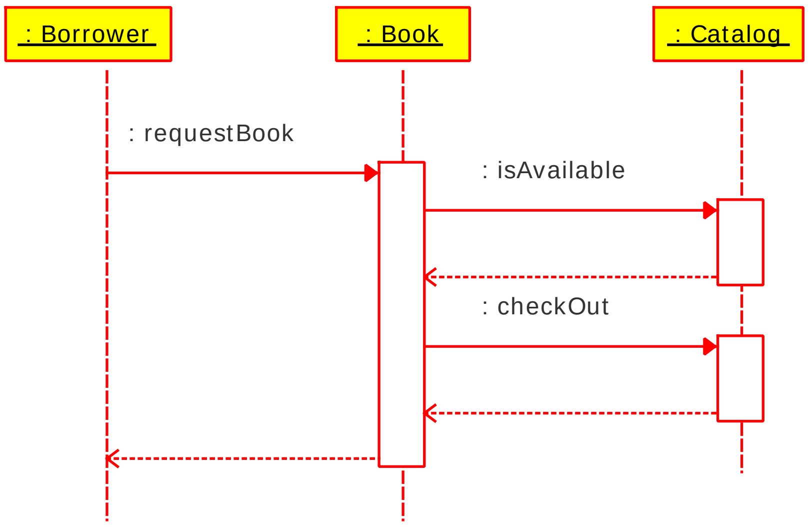
These *dynamic* aspects of the object model are usually represented in other diagram types of the UML. We will be briefly discuss *interaction diagrams*.

Interaction diagrams

Interaction diagrams show how objects interact with one another. Specifically, they show which objects are currently executing, and what methods they are calling on other objects.

There are various kinds of interaction diagrams that can be employed in the UML. We will examine one specific type, the *sequence diagram*. Sequence diagrams show how objects interact with each other over *time*. In other words, sequence diagrams show the sequence of interactions between objects.

Figure 5.9. Sequence diagram notation in the UML



The figure above shows a sequence diagram of how books are checked out of a library. Notice how the objects are arranged horizontally along the top of the diagram, and are represented as rectangle boxes, just as classes are. It is important to notice the colon before the class name in these boxes — this colon tells us that the box represents not a class, but an instance of the class. If the instance should be given a specific name, then the name appears before the colon. For example, “artOfWar : Book” represents an object called “artOfWar”, of class “Book”. This is exactly the same as when specifying attributes in classes (see the section called “Attributes” above).

It is important to realise that it is because sequence diagrams represent the interactions a software system undergoes over time that it is interested in objects rather than classes. A running software system consists of instances of classes, and not of the classes themselves.

Time is represented vertically, with earlier events happening closer to the top. This means that sequence diagrams should be read from top to bottom. Each object also has a vertical **life-line** that shows us the object's period of existence. Interactions are represented by solid and dotted lines, each of which has an arrow and a label. Solid lines show *method calls*, and point from the life-line of the object making the call towards the object which will execute the method. The rectangles along the life-line represent the time over which an object is executing a function. Dotted lines represent a return of control from the object executing a method to and object which had originally called it.

Note that the method names should be operations belonging to the object to which the arrow is pointing. “Borrower” calls “requestBook”, an operation belonging to an instance of class “Book”.

A useful way of using sequence diagrams is to outline difficult use cases as a sequence diagram: this will highlight what objects are required in developing the use case, as well as what actions each object will perform in order to complete the use case. Further, a sequence diagram can then be thought of as showing how the system reacts to *events*, the events being the input from the actor in the use case.

The UML also prompts us to use sequence diagrams in this way: remember that actors in use cases are themselves classes. We can easily use instance of these classes (of these actors) as valid objects in sequence diagrams.

Summary

Class modelling is the key modelling technique for object-oriented designers. If you are in a position to say that you understand class modelling, then you can reasonably claim that you understand object-oriented design. Of course, an understanding of other object-oriented techniques (e.g., use case modelling) will also be helpful.

Class modelling centres on a small number of philosophical points, which must be understood extremely well. There is not a great deal of factual material to remember, but these key points are crucial. These are the most important issues:

- The distinction between objects and classes
- The nature of attributes and operations
- The use and specification of associations, especially generalisation

In constructing a class model, the first step must be to obtain a working list of candidate classes. As design progresses, these classes will be refined and some will be deleted. At an even later stage, new classes will be introduced to support more detailed, implementation-related concepts.

A class model is refined by finding and specifying associations. Doing this helps to determine whether the initial choice of classes is a sound one, and how to improve it.

Ultimately the class model will be transformed into a computer program.

Review

Questions

Review Question 1

Which of the following entities are likely to be of interest when considering system design from a “logical” point of view, and which from an “implementation” point of view? Which are relevant to all standpoints? Clerk; Screen; Mouse (the input device, not the rodent); File; List; Ledger; Service technician; Database; Customer;

A discussion of this question can be found at the end of this chapter.

Review Question 2

Fill in the gaps with an appropriate word or phrase, or select the correct one of the indicated choices:

A class encapsulates the behaviour and properties of a number of [__________]. The members of a class are also referred to as [__________]. The minimum number of instances of a class is [__________], the maximum is [__________]. In object-oriented design, an object [can/cannot] be be a instance of more than one class, and an object [can/cannot] change which class it is an instance of.

A discussion of this question can be found at the end of this chapter.

Review Question 3

Suppose you are designing a computer system to carry out stock control in a large book warehouse. The system should store extensive details of each book, including at least the following information: authors, title, publisher, date of publication, size, number of pages, binding format, shelf location, re-order level, purchase price, sale price. Which of these properties of books do you think are relevant to the *conceptual* class “Book”, and what properties to the *concrete* class “Book”? This problem is somewhat more difficult than it first appears; see the answer for further discussion.

A discussion of this question found at the end of this chapter.

Review Question 4

Fill in the blanks with an appropriate word or phrase:

Attributes define the properties of a class. In a class specification an attribute has a [__________] and a [__________] (e.g., number, text). In each object of that class the attribute will also have a [__________]. This may be different for each object, but no instance of the class will lack the attribute entirely. For example if I say that a particular vehicle has 12 wheels, I am implying that vehicles have a attribute whose name is “[__________]”, whose type is “[__________]” and whose value in the particular case is [12]. The minimum number of attributes that a class may have is [__________]. The maximum number is unlimited, but in practice it is difficult to manage a class with more than about 30 attributes. Java programmers tend to use the term [__________] in preference to attribute, but these are essentially the same thing.

A discussion of this question can be found at the end of this chapter.

Review Question 5

Draw a UML class symbol that has the following specification. The class is called “Customer”. It has four attributes: name (which is text), address (also text), balance (a number), and credit rating (type currently unknown). It has one operation: “printStatement()”, which has no parameters or return value.

A discussion of this question can be found at the end of this chapter.

Review Question 6

In the early stages of development it is usually better to identify too many classes than too few. You can then rationalise you choice of classes by removing, modifying and merging them. The following list is of classes that may be considered for deletion or absorption into a new class. Suggest why this might be the case for each example.

- Fred Bloggs, a customer of the bank
- Colour
- Hard disk
- Central processing unit

- Linked list
- Red
- Test procedure
- Print
- List of customers

For as many of these examples as you can, suggest an application where it is not a bad choice of class.

A discussion of this question can be found at the end of this chapter.

Review Question 7

Draw the class diagram for the radio/televsion/car example given in the text, showing where multiple inheritance occurs.

A discussion of this question can be found at the end of this chapter.

Review Question 8

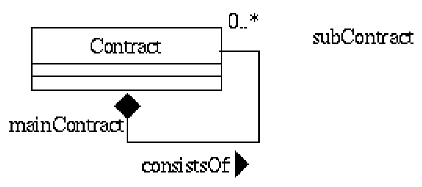
Construct a class model to represent the following information. The customer is a company that specialise in the supply of specialist musical recordings, of the type that are difficult to obtain through mainstream record shops. The customer would like to make it possible for their clients to order their products on-line using a Web browser. The on-line system is to provide full details of each musical recording, and will be integrated into an automated stock control system. In this way their client will be able to tell immediately if the items required are in stock. The customers describes the details of their stock as follows:

> We stock about 10,000 different musical recordings, usually with 1-10 of each item in stock at any given time. With more popular items we will stock ten or more copies of the same recording, perhaps in different formats.
>
> We stock recordings in different formats: cassette tapes, audio CDs and audio DVDs. The publishers we buy from will normally supply a recording in more than one format. For example, usually we will get CDs and cassettes of a given recording from the same publisher.
>
> We buy our stock from music publishers. For each publisher we record the name and address, and the name and e-mail address of our contact person there. When we place an order we need to know the publisher's catalogue number, which is generally different for different formats.
>
> When a customer looks at our Web site, we want to be able to give extensive details for all our recordings. For example, we will want to display the title, performer, composer, date and venue of recording, copyright holders and some general information. In addition, we want to display the titles and durations of all the musical tracks. This is complicated by the fact that the CD, DVD and cassette formats generally don't have the same tracks. Because a DVD is longer than a CD, it will usually have a few extra tracks. A cassette tape usually has even fewer tracks than the CD.

A discussion of this question can be found at the end of this chapter.

Review Question 9

Redraw the contract-subcontract relationship shown below so that it does not use a self-association. Do you think your model is more or less expressive in this form?



A discussion of this question can be found at the end of this chapter.

Review Question 10

What is meant by the statement “objects are not classes, but instances of classes. However, classes can be considered objects”? In other words, in what sense is a class also an object? (This is a tricky question if you are not absolutely clear on the distinction between “classes” and “objects”).

A discussion of this question can be found at the end of this chapter.

Review Question 11

Write a simple, one-line definition of each of the following key terms. Try to do this without referring back to the notes if possible. For each term, give an example.

- Attribute
- Operation
- Self-association
- Generalisation
- Polymorphism
- Link class
- Multiplicity
- Stereotype

A discussion of this question can be found at the end of this chapter.

Review Question 12

Which of the following are true? Give an example to support your decision.

- If Z is a subclass of Y, and Y is a subclass of X, then Z is a subclass of X
- If Z is associated with Y by an association A, and Y is associated with X by an association A, then Z is associated with X by an association A
- If Z is associated with Y by an association A and by an association B, then A and B are different names for the same association
- If each object of class Z is associated with ten objects of class Y, and each object of class Y is associated with ten objects of class X, then there are 100 objects of class X for each object of class Z

A discussion of this question can be found at the end of this chapter.

Review Question 13

What is the difference between class modelling, and entity-relationship modelling? In what ways are classes and entities similar? In what ways are they different? (If you have not completed the part of the course that deals with entity-relationship modelling, you may find this question difficult).

A discussion of this question can be found at the end of this chapter.

Review Question 14

Consider these three classes: Rectangle, Circle, and Line, which are subclasses of a general Shape class. These classes have operations drawRectangle(), drawCircle(), and drawLine() respectively. Each of these methods cause the appropriate shape to be drawn. Is this system exhibiting polymorphism or not?

A discussion of this question can be found at the end of this chapter.

Review Question 15

What is the difference between a link class and a link attribute? Gives examples of each.

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Probably “Clerk”, “Customer”, “Ledger” and “Service technician” are important in a logical standpoint. None of these are likely to transform directly into parts of a computer program. On the other hand, “File”, “List” and “Database” probably will. “Screen” and “Mouse” are less easy to categorise. In some cases these will in fact simply be alternative representations of the human beings that will interact with a system. For example, a clerk may use a keyboard and a mouse; in some systems it is the fact that these particular devices are in use that is important, but in most it will be the person and that person's role that are important.

Discussion of Review Question 2

A class encapsulates the behaviour and properties of a number of [objects, entities]. The members of a class are also referred to as [instances]. The minimum number of instances of a class is [zero], the maximum is [unlimited, infinite]. In object-oriented design, an object [cannot] be be a instance of more than one class, and an object [cannot] change which class it is an instance of.

Discussion of Review Question 3

Some things are clearly properties of the conceptual class. For a book, things like the title, author, date of publication and publisher are probably conceptual. They will be independent of any physical realisation of that book. For example, there are many copies of Tolstoy's “War and Peace” in existence, in many different formats, but all share the same fundamental properties.

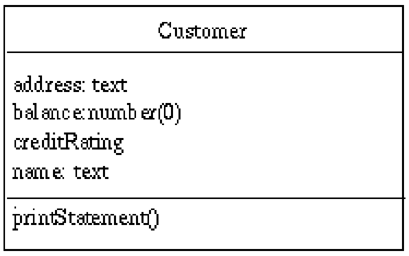
In a library, the shelf location and bar-code number are properties of the physical book.

Less straightforward cases are properties like the format (size, shape, binding) of the book. These are clearly “concrete” properties. However, there are many different copies of each of these formats. Are these individual copies “physical” and the formats themselves “conceptual”? This is not a straightforward question. It may be in a very complex book handling system that we have to institute a number of different classes to represent book information.

Discussion of Review Question 4

Attributes define the properties of a class. In a class specification an attribute has a [name] and a [type] (e.g., number, text). In each object of that class the attribute will also have a [value]. This may be different for each object, but no instance of the class will lack the attribute entirely. For example if I say that a particular vehicle has 12 wheels, I am implying that vehicles have a attribute whose name is “[number of wheels]”, whose type is “[number, integer]” and whose value in the particular case is [12]. The minimum number of attributes that a class may have is [zero]. The maximum number is unlimited, but in practice it is difficult to manage a class with more than about 30 attributes. Java programmers tend to use the term [instance variable] in preference to attribute, but these are essentially the same thing.

Discussion of Review Question 5

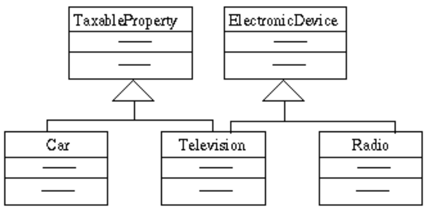


Discussion of Review Question 6

Bad classes:

- Fred Bloggs, a customer of the bank: this is potentially a bad class because Fred Bloggs is an object (instance), perhaps of class Customer. It is unlikely that there will be a whole class of Fred Bloggses.
- Colour: colour is likely to be an attribute of something. If you have studied Java programming you will probably come across a class called Colour. However, Java does this as an extension of the language, not because Colour represents anything in a model. However, in a program for, for example, colour mixing in printing machinery, perhaps Colour is important enough to be a class in its own right.
- Hard disk: this is a bad class because it is concerned with the internal operation of the computer. In modelling, we want to avoid detail of this depth. “Document” and perhaps “Folder” may be reasonable modelling concepts. However, in an application to design computers, HardDisk may be a sensible class.
- Central processing unit: has all the same problems as the example above
- Linked list: if you didn't know what a linked list was, then that is as good a reason as any for getting rid of it. Even if you did, it is still too technical to use in modelling
- Red: is probably the value of an attribute
- Test procedure: this is probably an operation, or perhaps a use case
- Print: this is almost certainly an operation (or a Printer class perhaps?)
- List of customers: too much detail. “Customer” is a sensible class name. The fact that there are more than one customer should be represented by the multiplicities of associations with other classes (see later).

Discussion of Review Question 7

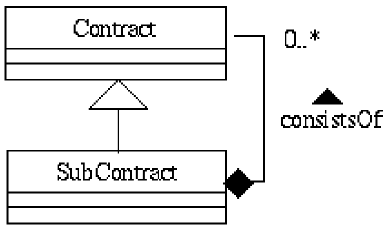


This is a somewhat contrived example, and is unlikely to occur in a real application. However, multiple inheritance does occur in real applications all the time. It is probably more important in programming, rather than modelling as such. For example, a popular style of programming called "mix-in" programming makes extensive use of multiple inheritance. In mix-in programming a large number of very simple, general-purpose classes are defined, and these are inherited in various combinations by the major classes. Because something is important in programming, this does not necessarily mean it should be used in modelling. This topic is an important and controversial one.

Discussion of Review Question 8

There is no single right answer to this question. You should be prepared to discuss you solution with your tutor/classmates (hint: a good answer will probably contain between 5 and 10 classes).

Discussion of Review Question 9



In this alternative formulation, there is no self-association, but a SubContract is shown as a sub-class of Contract. The representation that used a self-association did not make this class/sub-class relationship explicit; indeed it may not really be true. If one is prepared to accept the implicit assumption that SubContract is a sub-class of Contract, then the diagram shown above does effectively remove the self-association. This may be a reasonable approach for someone who really find self-associations difficult to follow. So perhaps some improvement in expressiveness has been gained, but at the expense of a slight loss of accuracy.

Discussion of Review Question 10

Suppose a class model has three classes, called ClassA, ClassB and ClassC. Objects of these classes will share properties (this is the definition of being an instance of a class, after all). However, the classes themselves will also have properties in common. For example, they will all the attribute "name". Its value is "ClassA" for ClassA, "ClassB" for ClassB, etc. All classes will have the property of being able to enter into relationships with other classes. All classes will have the attribute "numberOfAttributes". Thus we can say that there is a class called "Class", which embodies the behaviour of all classes. Each actual class is an instance of class "Class".

This is not simply an academic point. Being able to express a modelling notation in terms of the modelling notation itself is a very powerful way to ensure that the modelling system is consistent. If you study the UML specification documents, you will find that all UML diagram rules are themselves specified in UML notation. This is called meta-modelling

Discussion of Review Question 11

- Attribute: a property that objects of a class have, that can be expressed in terms of a name, a type and a value. For example, class "Vehicle" has attribute "numberOfWheels". Different vehicles have different values of this attribute.
- Operation: a well-defined unit of behaviour of an object of a class. For example, class Vehicle has operation "startEngine()".
- Self-association: an association between one or more objects of a class, and one or more objects of the same class. For example, each object of class "LivingThing" is involved in a one-to-many

association called “eats” with other objects of the same class (technically I suppose that some living things don't eat other living things, but I hope you get the general idea).

- Generalisation: the property that a class has of representing behaviour and attributes of a number of subclasses. For example, class Vehicle generalises classes Car, Bus and Tram, etc.
- Polymorphism: differences in behaviour between different sub-classes of the same base class. For example, all sub-classes of Vehicle have the operation “startEngine()”, but the mechanism of starting the engine is different in each case.
- Link class: a class that specifies the details of an association between two other objects. For example, the relationship between a seller and a buyer may be complex enough to need a class (e.g., SalesTransaction) to represent its details.
- Multiplicity: the number of objects that take part in each end of an association. For example, the multiplicity of the relationship between Driver and Vehicle is 1 to (1-*), meaning a driver can drive one or more vehicles.
- Stereotype: a category to which a class belongs, e.g., “Actor”, or “User interface element”.

Discussion of Review Question 12

- *If Z is a subclass of Y, and Y is a subclass of X, then Z is a subclass of X.* This is correct, and a fundamental principle of object orientation. If mammal is a type of animal, and dog is a type of mammal, then dog is a type of animal. Can you think of a dog that is a mammal but not an animal?
- *If Z is associated with Y by an association A, and Y is associated with X by an association A, then Z is associated with X by an association A.* This is false. For example, lions eat zebra and zebra eat grass. But lions don't eat grass. The association is not carried across the classes. This is one of the things that distinguishes generalisation/specialisation from other relationships.
- *If Z is associated with Y by an association A and by an association B, then A and B are different names for the same association.* This is also false. Two objects can be associated in completely different ways. For example, customers deposit money in a bank account; customers withdrawn money from a bank account. “Deposit” and “withdraw” are two totally different associations.
- *If each object of class Z is associated with ten objects of class Y, and each object of class Y is associated with ten objects of class X, then there are 100 objects of class X for each object of class Z.* This is correct. For example, if a program has ten screen windows, and each window has ten buttons, then there will be 100 buttons for each program.

Discussion of Review Question 13

Class diagrams can be considered to be highly-developed and regularised entity relationship models (ERMs). Class modelling provides a definite notation and meaning for concepts that are weakly-defined in ERMs. For example, an relationship called “is-a” can be used in an ERM to show that one thing is a type of another thing, or that one thing is an instance of another thing. Moreover, when an “is-a” relationship is used, it does not necessarily denote any inheritance of properties. In ERMs properties are usually represented as entities as well, so this would be difficult to show. The same applies to relationships that are often written as “has-a”. This is an aggregation, but the different types of aggregation are not as well defined in ERM as they are in class modelling.

For all these differences, ERMs and class diagrams are more similar than they are different; both attempt to show the static structure of system in terms of the “things” (entities; classes) of which it is comprised.

Discussion of Review Question 14

The system as described is not polymorphic. There is a difference in behaviour between the different subclasses of Shape, and this difference is in an operation that is philosophically common to the three classes. That is, drawCircle(), drawRectangle(), and drawLine() are obviously drawing operations. However, there is no corresponding “drawing” operation in the base class “Shape”. The system could not invoke the drawing operation in Shape, with a view to the correct sub-class being drawn. To be polymorphic, the base class needs to have an operation called “draw()” that is abstract, that is, does nothing. The sub-classes all provide their own implementations of “draw()”. Now if there is an object of an sub-class of Shape (it doesn't matter which), another object can cause it to be drawn by invoking the “draw()” operation. The presence of “draw” as an abstract operation in Shape guarantees that all sub-classes of shape will respond to a “draw” operation. This is a powerful simplifying principle.

Discussion of Review Question 15

Both link classes and link attributes provide additional information about associations than can be carried by a simple association name. A link attribute carries only a single piece of information and, being and attribute, cannot interact directly with other objects. A link class is a fully-fledged class and can take part in class relationships of its own. It can carry as much information as required. Link attributes are simpler to read and simpler to implement, but are not as versatile. For example, an employer and an employee may interact in a way defined by a contract. A simple link attribute might be a contract number. This would identify different contractual agreements between the same entities. However, implementing Contract as a class would allow a lot of extra information to be represented. Of course, it is only worth doing this if it improves the expressiveness or accuracy of the model.