ADDISON WESLEY

USING UML

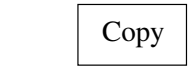
SOFTWARE ENGINEERING WITH OBJECTS AND COMPONENTS

PERDITA STEVENS
WITH ROB POOLEY

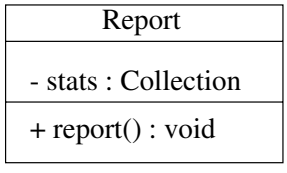
SECOND EDITION

Additional student support
at www.pearson.co.uk/stevens

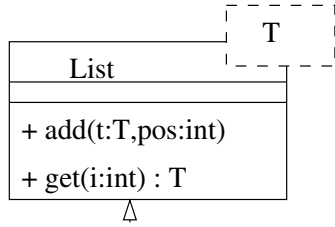
Classes : Chapters 5 and 6

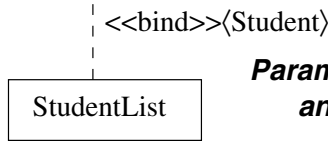


Simple class

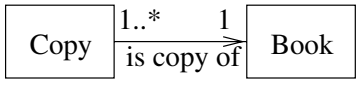


Class with attribute and operation

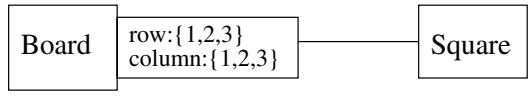




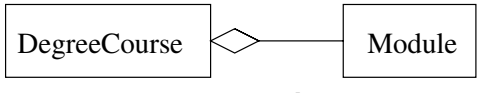
Parameterized class and its uses



Association with multiplicities and navigability

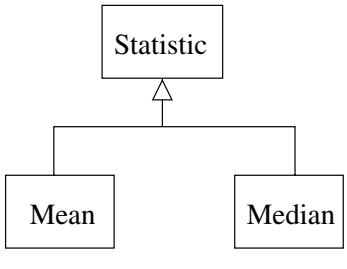


Qualified association

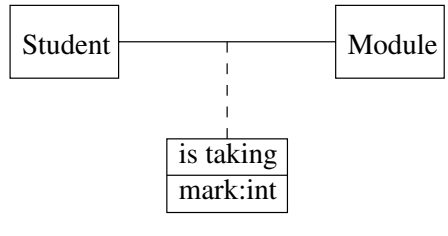


Aggregation

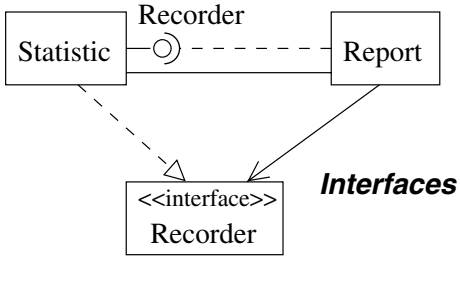
Composition



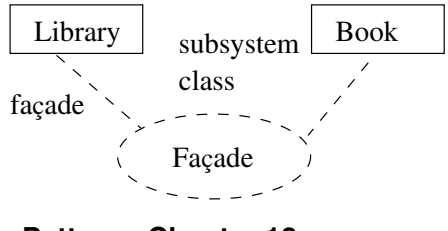
Generalization



Association class



Interfaces



Pattern - Chapter 18

5.1.4 Real-world objects vs their system representation

It is important to remember that objects are really things inside a computer program – that when we talk about 'books' and 'copies', for example, we really mean the representation of these things within our system. The consequences of this are that we must be careful

- not to record information that is definitely irrelevant to our system;
- not to lose sight of the fact that the objects *are* the system!

The latter point is particularly interesting. A classic error for people not yet steeped in OO is to invent a class, often called [Something]System, which implements all the system's interesting behavior. But in OO the whole caboodle is the system – that's the point! It's easy to drift into *monolithic* design in which there is just one object that knows and does everything. This is bad because such designs are very hard to maintain: they tend to have assumptions built into them about how the system will be used. (There is a quite different way of having a class which encapsulates the system: it is often useful, and in some languages obligatory, for there to be a main class, which is instantiated just once in each running instance of the system, and which provides the entry point of the program. Starting the program automatically creates this main object, which in turn creates the other objects in the system. The main object does not, however, have any complex behavior of its own. At the very most, it might forward messages arriving from outside to the appropriate object. This is related to the *Façade pattern*, which we will consider in Chapter 18.)

In Chapter 7 we will return to the question of how participants from outside our computer system (known in UML as *actors*) are represented in our design. This (it will turn out) is an essential part of what we've been discussing here. We've already touched on this in Chapter 3.

Q: Revisit the requirements description for the library system in Chapter 3. Apart from the classes identified for the first iteration, what classes should be in the final system?

5.2 Associations

In the same sense that classes correspond to nouns, associations correspond to verbs. They express the relationship between classes. In the example in Chapter 3, we saw associations like `is a copy of` and `borrows/returns`.

There are instances of associations, just as there are instances of classes. (Instances of classes are called objects; instances of associations are called *links* in UML, though this term is rarely used.) An instance of an association relates a pair[2] of objects.

We can look at associations conceptually or from an implementation point of view. Conceptually, we record an association if there is a real-world association described by a short sentence like 'a library member borrows a book' and the sentence seems relevant to the system at hand.

Class `A` and class `B` are associated if:

- an object of class `A` sends a message to an object of class `B`;

[2] We discuss only binary associations in this book, though in fact UML does have associations of other arities.

- an object of class `A` creates an object of class `B`;
- an object of class `A` has an attribute whose values are objects of class `B` or collections of objects of class `B`;
- an object of class `A` receives a message with an object of class `B` as an argument.

In short, they are associated if some object of class `A` has to know about some object of class `B`. Each link, that is, each instance of the association, relates an object of class `A` and an object of class `B`. For example, the association called `borrows/returns` between `LibraryMember` and `Copy` might have the following links:

- Jo Bloggs `borrows/returns` copy 17 of *The Dilbert Principle*
- Marcus Smith `borrows/returns` copy 1 of *The Dilbert Principle*
- Jo Bloggs `borrows/returns` copy 4 of *Software Reuse*

Discussion Question 33
As an alternative to calling this association `borrows/returns`, we (the authors) could have chosen to have two separate associations, one called `borrows` and the other called `returns`. Indeed if, instead of considering who borrows and returns a copy, we had been considering who is the author of a copy and who owns it, we would have chosen to have two separate associations. What's different about the two situations? Do you agree with our choice?

Discussion Question 34
Think about how the cases of association listed above overlap, and consider whether any of them should be removed. It is arguable, for example, that if an object of class `A` receives a message with an object of class `B` as argument, but does *not* later send that object a message or store it in an attribute, then this should not count as an association. In fact such a situation is often, though not always, bad design anyway. Construct some examples and consider whether you think they are sensible. Do you have an opinion on whether this should count as association?

The secret of good object-oriented design is to end up with a class model which does not distort the conceptual reality of the domain – so that someone who understands the domain will not get unpleasant surprises – but which also permits a sensible implementation of the required functionality. When you develop the initial class model, before you have identified the messages which pass between objects, you necessarily concentrate on the conceptual aspect of the model. Later, when we use interaction models to check our class model, we will be more concerned with whether the model permits a sensible implementation of the required functionality. However, the process isn't that you first develop the conceptual model, and then forget about conceptual relationships to develop the implementation class model. Throughout the development, you aim to develop a model which is good in both conceptual and implementational aspects. Success in this is a large part of what leads to a maintainable system, because such a model tends to be comparatively easy to understand, and therefore comparatively easy to modify sensibly.

In Figure 5.2 we can see how UML represents a general association between two classes, by drawing a line between their icons. This is usually annotated in various ways. It should

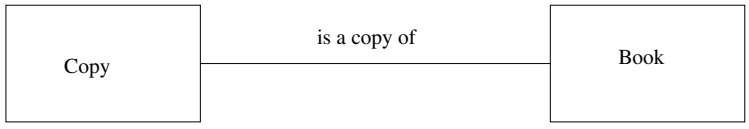


Figure 5.2 Simple association between classes.

normally at least have a label giving it a name, for readability. In Chapter 6 we will discuss the use of arrows on the association line to denote *navigability*: does the book know about the copy, or vice versa, or both? You might like to think about these questions now: some experts feel strongly that it is important to answer such questions early, others disagree.

Discussion Question 35
What are the advantages and disadvantages of deciding about navigability at this stage?

In the earliest stages of the development of a model it is often enough to draw a single line, indicating but not tying down the existence of some coupling. As the design matures, this line may be replaced with several, indicating different sorts of association. Some sorts of association are so common that UML has a predefined way of showing them; others can be defined by the designer as needed. We will look at different forms in Chapter 6.

TECHNICAL UML NOTE

The definition of association that we use in this book is a *dynamic* one: if at runtime objects may exchange a message, there must be a navigable association between their classes. Sometimes it is convenient to take a more restrictive *static* view, in which class `A` only has a navigable association to class `B` if an attribute of `A` contains an object (or collection of objects) of class `B`. UML does not properly specify which definition to use: in practice, it is up to the modeler to decide exactly what the existence of an association means. (Sometimes your choice may be determined by your choice of tool, especially if you use code generation features.) UML2 also includes the notion of a 'Connector' which is a more general way of connecting classes (and other classifiers). If you took a restrictive view of associations, you might also want to use connectors in other circumstances. We will look at a particularly useful kind of connector, an 'assembly connector', in the next chapter.

One annotation which is often used early on is the *multiplicity* of an association. Although it may not always be clear initially and may change with subsequent refinement of the design, this is so fundamental that we will spend some time thinking about it here.

5.2.1 Multiplicities

In the example in Chapter 3, we showed a `1` at the `Book` end of the association `is a copy of`, because every copy (that is, every object of class `Copy`) is associated by `is a copy`

of with just one book (object of class Book). On the other hand, there may be any number of copies of a given book in our system. So the multiplicity on the Copy end is 1..*.

As you see, we can specify:

- **an exact number** simply by writing it;
- **a range of numbers** using two dots between a pair of numbers;
- **an arbitrary, unspecified number** using *

Loosely, you can think of UML's * as an infinity sign, so the multiplicity 1..* expresses that the number of copies can be anything between 1 and infinity. Of course, at any time there will in fact be only a finite number of objects in our whole system, so what this actually says is that there can be any number of copies of a book, provided there's at least one.

Attributes, which we'll discuss in the next section, can have multiplicities too.

Q: Express in UML that a Student takes up to six Modules, where at most 25 Students can be enrolled on each Module.

Q: Consider the various ways in which such an association can be implemented in your programming language.

Discussion Question 36
Express in UML the relationship between a person and his/her shirts. What about the person's shoes? Do you think you have exposed a weakness in UML? Why, or why not?

Discussion Question 37
The number zero can never be a meaningful multiplicity, or can it?

Discussion Question 38
The existence of a multiplicity greater than one is sometimes assumed to mean that objects of that class must exist as a collection of some sort. Is that a safe assumption?

5.3 Attributes and operations

The system that we build will consist of a collection of objects, which interact to fulfill the requirements on the system. We have begun to identify classes and their relationships, but this cannot proceed far without considering the state and behavior of objects of these classes. We need to identify the operations and attributes that each class should have. Some will be obvious; others will emerge as we consider the responsibilities of objects and the interactions between them.

5.3.1 Operations

Most important are the operations of a class, which define the ways in which objects may interact. As we said in Chapter 2, when one object sends a message to another, it is asking the receiver to perform an operation. The receiver will invoke a method to perform the operation; the sender does not know which method will be invoked, since there may be many methods implementing the same operation at different levels of the inheritance hierarchy. The *signature* of an operation gives the selector, the names and types of any formal parameters (arguments) to the operation, and the type of the return value. Here, as usual, a type may be either a basic type or a class. Operations are listed in the bottom, third, compartment of a class icon.

Q: Revisit the example in Chapter 3 and derive operations for all the classes there.

5.3.2 Attributes

The attributes of a class – which, as discussed in Chapter 2, describe the data contained in an object of the class – are listed in the second, middle, compartment of a class icon. Figure 5.3 is a version of the `Book` class icon which shows that each object of class `Book` has a `title`, which is a string, and understands a message with selector `copiesOnShelf`, which takes no argument and returns an integer, as well as `borrow`, which takes as argument an object of class `Copy` and returns no result.

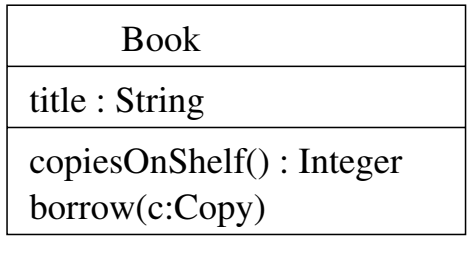


Figure 5.3 A simple class model, with attribute and operation.

Notice that we do not normally include attributes that simply implement the associations shown in the class diagram. For example, we do *not* show that `Book` has an attribute `copies` in which to store the collection of references to the `Copy` objects associated with the `Book`. The final implementation probably will have such an attribute, but to show it would add extra notation to the diagram without adding useful information. (It would also imply a decision about the navigability of the association, which would arguably be premature. Navigability is discussed further in Chapter 6.)

A rule of thumb is that the types of attributes should be either primitive types (integer, string etc.) or classes which do not appear in the class diagram, such as library classes. If the type of an attribute is a class which does appear in the class diagram, it is usually better to record an association between the two classes.

Q: Look for any obvious attributes that might exist for the other classes in the example from Chapter 3.

Views of operations and attributes

Just as with associations, we have a conceptual approach and a pragmatic approach, which we attempt to make consistent.

The conceptual approach involves identifying what data is conceptually associated with an object of this class, and what messages it seems reasonable to expect the object to understand. The latter view can lead you into an anthropomorphic view of objects as having an intelligence of their own – 'If a book could talk, what questions would you expect it to be able to answer?' – which some people find disconcerting, but which can nevertheless be worthwhile!

Pragmatically, we have to check that we have included enough data and behavior for the requirements at hand. To do this, we have to start to consider how the objects of our classes will work together to satisfy the requirements. One very useful technique for doing this is CRC cards, which we describe later in this chapter.

5.4 Generalization

Another important relationship which may exist between classes is *generalization*. For example, `LibraryMember` is a generalization of `MemberOfStaff` because, conceptually, every `MemberOfStaff` is a `LibraryMember`. Everything that every `LibraryMember` can do can certainly be done by every `MemberOfStaff`. So if some part of our system (e.g. the facility to reserve a book) works on an arbitrary `LibraryMember`, it ought to work on an arbitrary `MemberOfStaff` too, since every `MemberOfStaff` *is* a `LibraryMember`. On the other hand, there may be things that don't make sense for every `LibraryMember`, but only for a `MemberOfStaff` (e.g. borrow journal). `MemberOfStaff` is more specialized than `LibraryMember`; or `LibraryMember` is a generalization of `MemberOfStaff`.

In other words, an object of class `MemberOfStaff` should *conform* to the interface given by `LibraryMember`. That is, if some message is acceptable to any `LibraryMember`, it must also be acceptable to any `MemberOfStaff`. `MemberOfStaff`, on the other hand, may understand other, specialized messages which an arbitrary `LibraryMember` might not be able to accept – that is, a `MemberOfStaff`'s interface may be strictly broader than `LibraryMember`'s.

From this we can see that to say that a generalization relationship exists between classes is to make a strong, though informal, statement about the way the objects of the two classes behave.

> An object of a specialized class can be substituted for an object of a more general class in any context which expects a member of the more general class, but not the other way round.

This takes us to a further rule about the design of classes where one is a specialization of the other.

> There must be no conceptual gulf between what objects of the two classes do on receipt of the same message.