Chapter 3
Relationships Between Classes

In the previous chapter we studied classes and objects as the two building blocks of object-oriented systems. The structure of a software system is defined by the way in which these building blocks relate with one another and the behaviour of the system is defined by the manner in which the objects interact with one another. Therefore, in order to construct a software system, we need mechanisms that create connections between these building blocks. In this chapter we introduce the basic types of relationships between classes (and objects) that make the connections.

The simplest and most general kind of relationships is **association**, which simply indicates that the objects of the two classes are related in some non-hierarchical way. There are almost no other restrictions on how an association can be formed, although we shall see throughout this text the good design practices that ought to be followed when creating associations.

When two or more classes have a hierarchical relationship based on generalisation, it is referred to as **inheritance**. Classes connected by inheritance share some commonalities and therefore, this kind of relationship is more restrictive than association.

The third kind of relationship we see is **genericity**. This is more restrictive than inheritance due to the fact that the only variations permitted across related classes are those that can be captured by **type parametrisation**, i.e., providing parameters of differing types when creating an instance of the generic entity.

In the rest of this chapter we elaborate on each of these, discussing the basic principles and examining situations where they can be applied. Since these mechanisms are basic to OOAD, they will all be revisited in later chapters when dealing with real examples of more complex systems.

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_3

3.1 Association

An association is formally defined as a relation among two or more classes describing a group of links with common structure and semantics. An association implies that an object of one class is making use of an object of another class and is indicated simply by a solid line connecting the two class icons. In the previous chapter we defined a class `Student` that keeps track of information about the courses that the student has registered for. This information is represented as shown in Fig. 3.1. In our example, `Student` objects may make use of `Course` objects when transcripts are generated, when tuition is computed or when a course is dropped. The link to the course provides the student object with the necessary information.

An association does not imply that there is always a link between all objects of one class and all objects of the other. As one would expect, in our example, a link is formed between a `Student` object and a `Course` object only when the operation that links them is completed, i.e., the student represented by the `Student` object registers for that particular course. However, an association does imply that there is a persistent, identifiable connection between two classes. If class A is associated with class B, it means that given an object of class A, you can always find an object of class B, or you can find that no B object has been assigned to the association yet. But in either case there is always an identifiable path from A to B. Associations thus represent conceptual relationships between classes as well as physical relationships between the objects of these classes.

In terms of implementation, what the above implies is that class A must provide a mechanism using the constructs of the chosen programming language to form a link. This could take several forms, for example,

- Class A stores a key (or several keys) that uniquely identifies an object of class B.
- Class A stores a reference(s) to object(s) of class B.
- Class A has a reference to an object of class C, which, in turn is associated with a unique instance of class B.

The first two of these create a direct association, whereas the third one is an indirect association. The mechanism chosen may depend on the requirements that the system has to satisfy (for instance, the kinds of queries that need to be answered) and also on how the rest of system is designed. In our example, when a student registers for a course, he/she actually enrolls in a specific section of the course. The mechanism to make this connection may simply be that the `Student` object stores a reference to the `section` object. Each section is associated with a unique course, completing the picture (see Fig. 3.2).

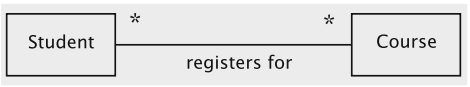

Fig. 3.1 Association between classes

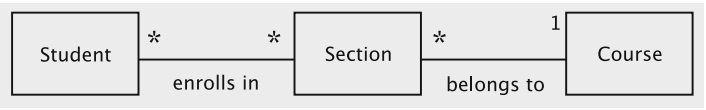

Fig. 3.2 Association involving three classes

An association is assumed to be **bi-directional** unless we place a directional arrow on the connecting line to indicate otherwise. The association usually has a descriptive **name**. The name often implies a direction, but in most cases this can be inverted. Our figure says student *enrolls in* a section, which *belongs to* a course, but this could be stated as a course *has* sections that *enroll* students. The diagram is usually drawn to read the link or association from left to right or top to bottom.

The entities at the ends of the association usually have assigned **roles**, which may have names. We could have an association named 'employs' that connects a class representing a `Business` to a class representing a `Person` employed by the business. Here `Business` plays the role of of the *employer* and `Person` has the role of *employee*.

3.1.1 Characteristics of Associations

Since associations represent very general relationships, they allow for variation in the nature of the connection. The common variation is the arity of the connection, i.e., how many objects of class A can be linked to one object of class B and vice-versa. Another variation involves whether there is some form of containment involved in the relationship. In other cases there is some specific kind of information that is added to the system whenever a link is made between objects. These characteristics are usually represented in UML by annotating the connection between classes. Some of these are discussed below.

Arity of Relationships

The **arity** of a relationship could be **one–one**, **one–many**, or **many–many**. An example of a one–one relationship could be between a `User-Interface` class accepting user input and a `Display-Window` class displaying information. A multi-user system, however, can interact with several users in parallel. Each interaction has a dedicated `Display-Window` object and all these objects are deployed through the common `User-Interface`. This is an example of a one–many relationship. From our example, a course may have several sections but each section is associated with only one course, thus creating a one(course)–many(section) connection. A student can enroll in several sections and each section can have several students enrolled. This would be an example of a many–many relationship.

Fig. 3.3 Composition across classes

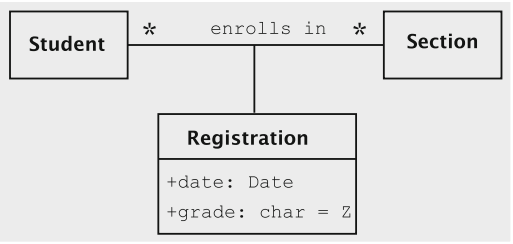

Fig. 3.4 Using an association class

Containment Relationships

Aggregation is a kind of association where the object of class A is 'made up of' objects of class B. This suggests some kind of a whole–part relationship between A and B. Most experts have downplayed the importance of this kind of association as not something that deserves to be embellished in any way. However, **composite aggregation**, also known as **composition**, has been recognised as being significant. Composition implies that each instance of the part belongs to only one instance of the whole, and that the part cannot exist except as part of the whole. Composition is indicated with a filled-in diamond and is usually not named since some form of whole–part relationship is assumed. In Fig. 3.3, a vertex cannot exist unless it is a part of a triangle. If the triangle object is destroyed, so are the individual vertices.

Association Classes

An association usually results in some information being added to the system since it adds a path connecting two objects. In some situations we add some information that qualifies the nature and describes the properties of the relationship. Outside the context of the association, this information does not have any relevance to either of the objects involved. In such cases we treat the association itself as a class. An example of this is shown in Fig. 3.4. When a student enrolls in a section, a registration record is created to store the date of registration and a grade. Such a record does not make sense if a particular student does not enroll in a given section.

FAQs About Forming Associations

What does an association represent?
An association normally represents something that will be stored as part of the data and reflects all links between objects of two classes that may ever exist. It describes a relationship that will exist between instances at run time and has an example.

When can we call a relationship an association?
In UML class diagrams, associations should be shown if a class possesses, controls, is connected to, is related to, is a part of, has as parts, is a member of, or has as

members some other class in the system. As association should *not* be used to denote relationships that: (i) can be drawn as a hierarchy, (ii) stems from a dependency alone, (iii) or relationships whose links will not survive beyond the execution of any particular operation.

How is an association represented?
An association shows how two classes are related to each other and this relationship should be made clear. It is denoted by a line connecting the two classes, with sufficient annotation to make the relationship clear and unambiguous. This annotation includes a name, the arity, roles and any association classes. In particular, if the annotation includes neither an association name nor a role name, the default name 'has' is applied.

3.2 Inheritance

There are situations when two classes have not only a great deal of similarity, but also significant differences. The classes may be similar enough that association does not capture the similarity, and differ too much so that the idea of genericity cannot be profitably employed. Suppose that C_1 and C_2 are two such classes. We then extract the common aspects of C_1 and C_2 and create a class, say, B, to implement that functionality. The classes C_1 and C_2 could then be smaller, containing only properties and methods that are unique to them. This idea is called **inheritance**—C_1 and C_2 are said to **inherit** from B. B is said to be the **baseclass** or **superclass**, and C_1 and C_2 are termed *derived classes* or **subclasses**. The superclasses are generalisations or **abstractions**: we move toward a more general type, an 'upward' movement, and subclasses denote *specialisation* toward a more specific class—a 'downward' movement. The class structure then provides a *hierarchy*.

Inheritance can be defined as the mechanism provided in programming languages to achieve the idea of vertical generalisation outlined above. Formally, an inheritance is a relationship characterised by an **ancestor** and a **descendant** represented using UML notation as in Fig. 3.5. Here, the baseclass is the ancestor and the derived classes are the descendants. We draw one box per class and draw an arrow from the derived classes to the baseclass.

3.2.1 An Example of a Hierarchy

Consider a company that sells various products such as television sets and books. Obvious differences between the products imply that they have different attributes to be tracked and that we need two classes, `Television` and `Book`. One way to accomplish this task is to create a class for television sets, say, `Television`, and a second class, for books, say, `Book`. However, in many situations the company would

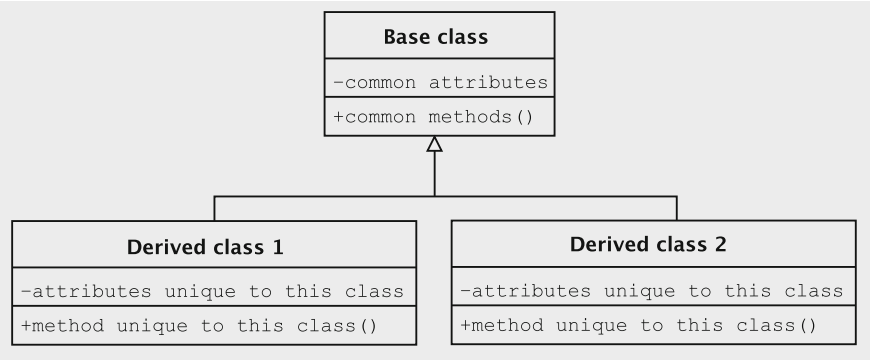

Fig. 3.5 Basic idea of inheritance

like to think of books and televisions as simply products. For instance, the company needs to keep track of sales, profits (or losses), etc., for all products. Now, add to the above situation more products, say, CDs, DVDs, cassette players, pens, etc. Each may warrant a separate class, but, as just discussed, they all have common properties and behaviours and to the company, they are all products.

What we see is an example of a situation where two classes have a great deal of similarities, but also substantial differences. The need to view different entities such as televisions and books as products suggests that we may benefit by having a new type, `Product`, introduced into the system. Since there is a fair amount of common functionality between the two products, we would like `Product` to be a class that implements the commonality found in `Television` and `Book`.

In Java, we do this as follows. We start off with a class that captures the essential properties and methods common to all products.

```
public class Product {
  // functionality for a product
}
```

The above class may have attributes such as number of units sold and unit price. It also will have constructors and methods for recording sales, computing profits, and so on.

We are now ready to create a class that represents a single TV set. For this, we note that a television is a product and that we would like to utilise the functionality that we just implemented for products. In Java, we do this as below:

```
public class Television extends Product {
  // functionality that is unique for televisions
  // modifications
}
```

Informally speaking, the `Television` class inherits all of the properties and methods from the class `Product`. All we have done is add properties and methods unique to televisions, which will not, for obvious reasons, be implemented in `Product`.

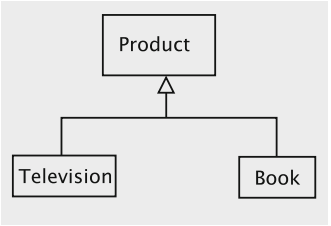

Fig. 3.6 Inheriting from product

In a similar manner, we implement the class `Book`.

```
public class Book extends Product {
  // functionality that is unique for books
  // modifications
}
```

The relationships between the three classes is depicted in Fig. 3.6.

Class Structure

Our purpose in this section is to describe how inheritance works. We do not worry about the details of the functionality, and so we do not describe the use cases. Moreover, due to necessity, we give a simplistic view of the application.

First, let us consider the two entities, television and book, in isolation without worrying about the relationships between them. The functionalities required of the two classes, `Television` and `Book`, are given in Fig. 3.7.

Now, notice the similarities and differences between the two classes: both classes, since they represent products, carry the fields `quantitySold` and `price` with their obvious meanings. The method `sale()` in both classes is invoked whenever one unit (a book or a TV set) is sold. The meaning of the `setPrice()` method should be obvious.

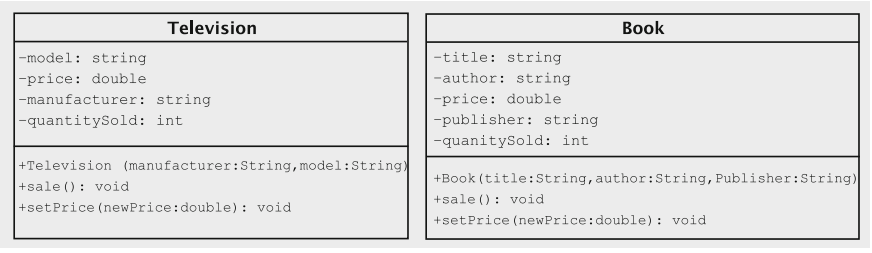

Fig. 3.7 An example of similar classes

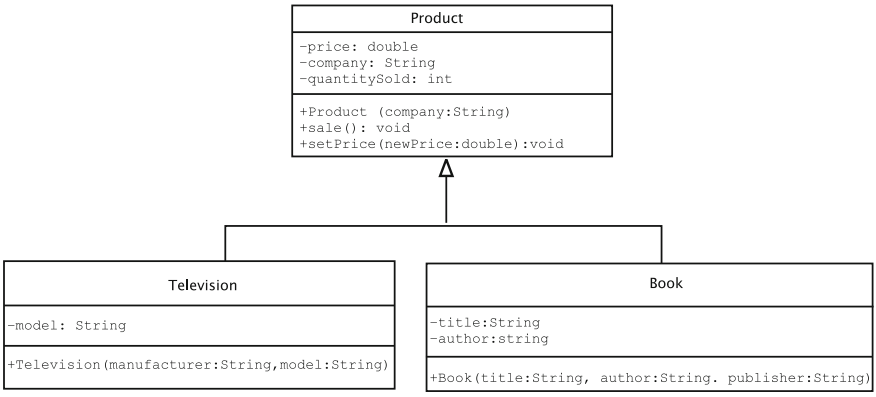

Fig. 3.8 Inheriting from product

The two classes are somewhat different in other respects: `Book` has attributes `title` and `author` whereas `Television` class has the attribute `brand`. The `manufacturer` attribute is named differently from, but not dissimilar to, `publisher`.

Here is where the power of the object-oriented paradigm comes into play. It allows the development of a baseclass or superclass that reflects the commonalities of the two classes and then *extends* or sub classes this base class to arrive at the functionalities we discussed before. A UML diagram that shows the arrangement is shown in Fig. 3.8. The class `Product` keeps track of the common attributes of `Book` and `Television` and implements the methods necessary to act on these attributes. `Television` and `Book` are now constructed as subclasses of `Product`; they will both inherit the functionalities of `Product` so that they are now capable of keeping track of sales of these two products.

The code for `Product`, given below, is fairly simple. The variable `company` stores the manufacturer of the product. Otherwise, there are no special features to be discussed.

```
public class Product {
  private String company;
  private double price;
  private int quantitySold;

  public Product(String company, double price) {
    this.company = company;
    this.price = price;
  }
  public void sell() {
    quantitySold++;
  }
  public void setPrice(double newPrice) {
    price = newPrice;
  }
```

```
  public String toString() {
    return "Company:" + company +  "price:" +
      price + "quantity sold" + quantitySold;
  }
}
```

Let us now construct `Television`, which extends `Product`. Any object of type `Television`, the subclass, can be thought of as having two parts: one that is formed from `Television` itself and the other from `Product`, the superclass. Thus, this object has four fields in it, `model`, `quantitySold`, `price`, and `company`. Often, the code within the subclass is responsible for managing the fields within it and the code in the superclass deals with the fields in the superclass.

Recall that objects are initialised via code in the constructor. When inheritance is involved, the part of the object represented by the superclass must be initialised *before* the fields of the subclass are given values; this is so because the subclass is built from the superclass and thus the former may have fields that depend on the fact that the superclass's attributes are initialised. An analogy may help: when a house is built, the roof is put in only after the walls have been erected, which happens only after the foundation has been laid.

To create a `Television` object, we to invoke a constructor of that class as below, where we pass the brand name, manufacturer name, and price.

```
Television set = new Television("RX3032", "Modern Electronics", 230.0);
```

Thus the constructor of `Television` must be defined as below.

```
public Television(String model, String manufacturer, double price) {
   // code to initialise the Product part of a television
   // code to initialise the television part of the object
}
```

We already have a piece of code that initialises fields of a `Product` object: the constructor of `Product`. So all we need to do is call that constructor! This is accomplished by the statement

```
super(/* appropriate parameters go here*/)
```

The call `super` with proper parameters always invokes the superclass's constructor. The superclass' constructor call can be invoked only as the very first statement from the code within a constructor of a subclass; it cannot be placed after some code or placed in methods.

In this example, the parameters to be passed would be the manufacturer's name and price. The code for the constructor is then

```
public Television(String model, String manufacturer, double price) {
  super(manufacturer, price);
  // store the model name
}
```

`super` is a keyword in Java which denotes superclass. Invocation of the superclass's constructor is done using this keyword followed by the required parameters in parentheses.

The fields of the superclass are initialised before fields in the subclass. What this means in the context of object creation is that the constructor of `Television` can begin its work only after the constructor of the superclass, `Product`, has completed execution. Of course, when you wish to create a `Television` object you need to invoke that class's constructor, but the first thing the constructor `Television` does (and must do) is invoke the constructor of `Product` with the appropriate parameters: the name of the company that manufactured the set and the price.

The result of `super(manufacturer, price)` is, therefore, to invoke `Product`'s constructor, which initialises `company` and `price` and then returns. The `Television` class then gives a value to the `model` field and returns to the invoker.

As is to be expected, the class `Television` needs a field for storing the model name. We thus have a more complete piece of code for this class as given below.

```
public class Television extends Product {
  private String model;
  public Television(String model, String manufacturer, double price) {
    super(manufacturer, price);
    this.model = model;
  }
  public String toString() {
    return super.toString() + "model:" + model;
  }
}
```

The `toString()` method of `Television` works by first calling the `toString()` method of `Product`, which returns a string representation of `Product` and concatenates to it the model name.

3.2.2 Inheriting from an Interface

A specialised kind of inheritance is one where a class inherits from an interface. Recollect that in Chap. 2 we had defined an interface as a collection of methods that can be implemented by a class. An interface has been likened to a contract signed by the class implementing the interface. In the context of this chapter, it should be pointed out that implementing the interface can also be viewed as a form of inheritance, where the implementing class inherits an abstract set of properties from the interface.

Java recognises an interface as a type (as do several other object-oriented languages), which means that objects that belong to classes that implement a given interface also belong to the type represented by the interface. Likewise, we can declare an identifier as belonging to the type of the interface and we can then use it to access the objects of any class that implements the interface.

```
public interface I {
  // details of I
}

public class  A implements I {
  //code for A
}

public class  B implements I {
  //code for B
}

I i1 = new A(); // i1 holds an A

I i2 =  new B(); // i2 holds a B
```

In the UML notation, this kind of a relationship between the interface and the implementing class is termed **realisation** and is represented by a dotted line with a large open arrowhead that points to the interface as shown in Fig. 3.8.

3.2.3 Polymorphism and Dynamic Binding

Consider a university application that contains, among others, three classes that form a hierarchy as shown in Fig. 3.9. A student can be either an undergraduate student or a graduate student. Just as in real life where we would think of an undergraduate or a graduate student as a student, in the object-oriented paradigm also, we consider an `UndergraduateStudent` object or a `GraduateStudent` object to be of the type `Student`. Therefore, we can write

```
Student student1 = new UndergraduateStudent();
Student student2 = new GraduateStudent();
```

This is a powerful idea. We can now write methods that accept a `Student` and pass either an `UndergraduateStudent` or a `GraduateStudent` object to it as below.

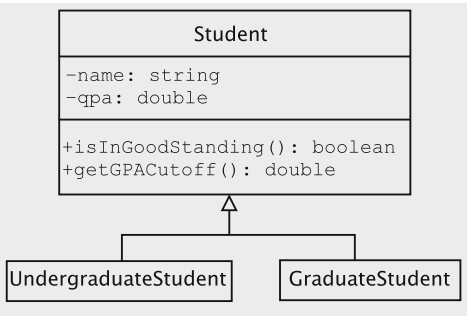

Fig. 3.9 Student hierarchy

```
public void storeStudent(Student student) {
  // code to store the Student object
}
```

We can then create `UndergraduateStudent` and `GraduateStudent` objects and pass them to the above method.

```
storeStudent(new UndergraduateStudent());
storeStudent(new GraduateStudent());
```

Once again, in real life, we usually do not think of a graduate student as an undergraduate student or vice-versa. In the same way, we cannot write the following code in Java.

```
UndergraduateStudent student1 = new GraduateStudent(); // wrong
GraduateStudent student2 = new UndergraduateStudent(); // wrong
```

Since we allow `Student` references to point to both `UndergraduateStudent` and `GraduateStudent` objects, we can see that some, but not all `Student` references may point to objects of type `UndergraduateStudent`; similarly, some `Student` references may refer to objects of type `GraduateStudent`. Thus, we cannot write,

```
Student student1;
student1 = new UndergraduateStudent();
GraduateStudent student2 = student1; // wrong!
```

The compiler will flag that the code is incorrect.

But, the following code, intuitively correct, is flagged by the compiler as incorrect.

```
Student student1;
student1 = new GraduateStudent();
GraduateStudent student2 = student1; // compiler generates a syntax error.
```

The reason for this error is that the compiler *does not execute the code* to realise that `student1` is actually referring to an object of type `GraduateStudent`. It is trying to protect the programmer from the absurd situation that could occur if `student1` held a reference to an `UndergraduateStudent` object. It is the responsibility of the programmer to tell the compiler that `student1` actually points to a `GraduateStudent` object. This is done through a cast as shown below.

```
Student student1;
student1 = new GraduateStudent();
GraduateStudent student2 = (GraduateStudent) student1; //  O.K. Code works.
```

To reiterate, while casting a reference to a specialised type, the programmer must ensure that the cast will work correctly; the compiler will happily allow the code to pass syntax check, but a carelessly-written code will crash when executed. See the following code.

```
Student student1;
student1 = new UndergraduateStudent();
GraduateStudent student2 = (GraduateStudent) student1; // crashes
```

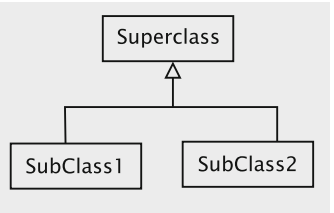

Fig. 3.10 Illustrating polymorphic assignment

`Student1` does not point to a `GraduateStudent` object in the last line, so the system's attempt to cast the instance of `UndergraduateStudent` to an instance of `GraduateStudent` will fail and the code will crash.[1]

The general rules are as follows. Refer to Fig. 3.10.

1. Any object of type `SubClass1` or `SubClass2` can be stored in a reference of type `SuperClass`.
2. No object of type `SubClass1` (`SubClass2`) can be stored in a reference of type `SubClass2` (`SubClass1`).
3. A reference of type `SuperClass` can be cast as a reference of type `SubClass1` or `SubClass2`.

Assignments of the above kind are termed *polymorphic*. A reference is able to point to objects of different types as long as the actual types of these objects *conform* to the type of reference. The above rules informally give the notion of **conformance**.

It is instructive to compare assignments and casts given above with the rules for assignments and casts of variables of primitive types. Some type conversions, for example, from `int` to `float`, do not need any casting; `float` variables have a wider range than `int` variables. Some others, `double` to `int` being an instance, are fine with casting; however, the programmer must be prepared for loss of precision. And the rest—any casts from (to) `boolean` to (from) any other type—are always disallowed.

We have so far seen examples of polymorphic assignments. In one of these, we store a reference to an object of the class `GraduateStudent` in an entity whose declared type is `Student`. This is equivalent to taking a bunch of bananas and storing them in a box labelled 'fruit'. The declared contents of the box (as given by the label) is fruit, just as the declared type of entity `student1` in the LHS of the assignment is `Student`. By doing this, we have lost some information since we can no longer find out what kind of fruit we have in the box without examining its contents. Likewise, when we operate on the entity `student1`, all we can assume is that it contains a reference to a `Student` object. Thus there is a loss of information in this kind of assignment.

The second kind of polymorphic assignment is one where we moved a reference from an entity whose declared type is `Student` to an entity whose declared type is `GraduateStudent`. (This would amount to taking the bananas out of the box

[1] Technically, the system throws an exception, a topic that will be covered in detail in Chap. 4. In this case, an instance of the class ClassCastException is thrown open.

labelled 'fruit' and putting them in the box labelled 'bananas'; we do this only if we are sure that box did have bananas.) As we saw with our cast and exception, this can only be done after ensuring that the entity being used is of type `GraduateStudent`. This is therefore an operation that 'recovers' information lost in assignments of the previous kind.

What we conclude from this is that using polymorphism does result in a loss of information at run time. Why, then, do we use this? The answer lies in **dynamic binding**. This ability allows us to invoke methods on members of a class hierarchy without knowing what kind of specific object we are dealing with. To make a rough analogy with the real world, this would be like a manager in a supermarket asking an assistant to put the fruits on display (this is analogous to applying the 'display' method to the 'fruit' object). The assistant looks at the fruit and applies the correct display technique (assuming he wants to keep his job). Here the manager is like a client class invoking the 'display' method and the assistant plays the role of the system and applies dynamic binding.

To get a concrete understanding of how dynamic binding works, let us revisit the example of the `Student` hierarchy. The code for `Student` may be written as follows.

```
public abstract class Student {
  private String name;
  private double gpa;
  // more fields
  public Student(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public boolean isInGoodStanding() {
    return (gpa >= getGPACutoff());
  }
  public abstract double getGPACutoff();
  // more methods
}
```

In practice, a `Student` class will be far more complicated; we have omitted a large body of code that would otherwise be present there. The `String` field `name` is, as may be guessed, for remembering the name of the student. As you can see, the name of the `Student` gets initialised in the constructor. The grade point average (GPA) is stored in the `double` field `gpa`. As students take classes and complete them, they will get grades, which will be used in computing the GPA. None of that code is shown in this class.

We assume that periodically, perhaps at the end of each semester or quarter, the university will check students to see if they are in 'good standing'. Typically, it would mean ensuring that the student is progressing in a satisfactory manner. We assume that for a student good standing means that the student's GPA meets a certain minimum requirement. The minimum GPA expected of students may change depending on whether the student is an undergraduate or a graduate student. The method `getGPACutoff()` returns the minimum GPA a student must have to be

in good standing. We will assume that this value is 2.0 and 3.0 for undergraduate and graduate students respectively. Note that the method is declared abstract in the `Student` class.

Let us now focus on the code for `UndergraduateStudent`, which is given below.

```
public class UndergraduateStudent extends Student {
  public UndergraduateStudent(String name) {
    super(name);
  }
  public double getGPACutoff() {
    return 2.0;
  }
}
```

The constructor gets the name of the student as its only parameter and calls the superclass's constructor to store it. Since this is a non-abstract class, the `getGPACutoff` method which returns the minimum GPA is implemented.

All of the public and protected[2] methods of a superclass are inherited in the two subclasses. So, the method `isInGoodStanding` can be instantiated on an instance of `UndergraduateStudent` as well. Thus the following code is valid.

```
UndergraduateStudent student = new UndergraduateStudent("Tom");
// code to manipulate student
if (student.isInGoodStanding()) {
   // code
} else {
   // code
}
```

When the method is called, the `isInGoodStanding` method in the superclass `Student` will be invoked.

Finally, we have the code for the class graduate students. The constructor for the class is quite similar to the one for the `UndergraduateStudent` class. To make the class non-abstract, this class, too, should have an implementation of `getGPACutoff`. In addition, we assume that to be in good standing graduate students must meet the requirements imposed on all students and, in addition, they cannot have more than a certain number of courses in which they get a grade below, say, B.

What we would like is a redefinition or **overriding** of the method `isInGood Standing`. Overriding is done by defining a method in a subclass with the same name, return type, and parameters as a method in the superclass so that the subclass's definition takes precedence over the superclass's method. Thus the code for the `isInGoodStanding` method is now different. See below.

```
public class GraduateStudent extends Student {
  public GraduateStudent(String name) {
    super(name);
  }
  public double getGPACutoff() {
```

[2]Protected access will be explained shortly.

```
    return 3.0;
  }
  public boolean isInGoodStanding() {
    return super.isInGoodStanding() && checkOutCourses();
  }
  public boolean checkOutCourses() {
    // implementation not shown
  }
}
```

Now, suppose we have the following code.

```
GraduateStudent student = new GraduateStudent("Dick");
// code to manipulate student
if (student.isInGoodStanding()) {
   // code
} else {
   // code
}
```

In this case, the call to `isInGoodStanding` results in a call to the code defined in the `GraduateStudent` class. This in turn invokes the code in the `Student` class and makes further checks using the locally declared method `checkOutCourses` to arrive at a decision.

Recall the `StudentArrayList` class we defined in Sect. 2.4 which stores `Student` objects. The method to add a `Student` in this class looked as follows:

```
public void add(Student student) {
  // code
}
```

Since a `Student` reference may point to a `UndergraduateStudent` or a `GraduateStudent` object, we can pass objects of either type to the `add` method and have them stored in the list. For example, the code

```
StudentArrayList students = new StudentArrayList();
UndergraduateStudent student1 = new UndergraduateStudent("Tom");
GraduateStudent student2 = new GraduateStudent("Dick");
students.add(student1);
students.add(student2);
```

stores both objects in the list `students`.

Suppose the class also had a method to get a `Student` object stored at a certain index as below.

```
public Student getStudentAt(int index) {
  // Return the Student object at position index.
  // If index is invalid, return null.
}
```

Let us focus on the following code that traverses the list and checks whether the students are in good standing.

```
for (int index = 0; index < students.size(); index++) {
  if (students.getStudentAt(index).isInGoodStanding()) {
    System.out.println(students.get(index).getName()
                       + "is in good standing");
  } else {
    System.out.println(students.getStudentAt(index).getName()
                       + "is not in good standing");
  }
}
```

We assume that students Tom, an undergraduate student, and Dick, a graduate student, are in the list as per the code given a little earlier. The loop will iterate twice, first accessing the object corresponding to Tom and then getting the object for Dick. In both cases, the `isInGoodStanding` method will be called.

What is interesting about the execution is that the system will determine at run time the method to call, and this decision is based on the actual type of the object. In the case of the first object, we have an instance of `UndergraduateStudent`, and since there is no definition of the `isInGoodStanding` method in that class, the system will search for the method in the superclass, `Student`, and execute that. But when the loop iterates next, the system gets an instance of `GraduateStudent`, and since there is a definition of the `isInGoodStanding` method in that class, the overriding definition will be called.

This is a general rule: whenever a method call is encountered, the system will find out the actual type of object referred to by the reference and see if there is a definition for the method in the corresponding class. If so, it will call that method. Otherwise, the search proceeds to the superclass and the process gets repeated. The actual code to be executed is bound dynamically; hence this process is called dynamic binding.

The above code shows the power of dynamic binding. In our calls to `isInGood Standing`, we were unaware of the type of objects. Simply by examining the code that calls the method, we cannot tell which definition of the `isInGoodStanding` method will be invoked, i.e., *dynamic binding gives us the ability to hide this detail in the inheritance hierarchy*.

3.2.4 Protected Fields and Methods

Consider the hierarchy as shown in Fig. 3.11. `ClosedFigure` has an attribute `area` which stores the area of a `ClosedFigure` object. Since the classes `Polygon` and `ClosedCurve` are kinds of `ClosedFigure`, we would like to make this attribute available to them. This implies that the attribute cannot be private; on the other hand making it public could lead to inappropriate usage by other clients. The solution to this is found in the `protected` access specifier. Loosely speaking, what this means is that this field can be accessed by `ClosedFigure` and its descendants as shown below.

```
public class ClosedFigure extends Figure {
  protected  double area;
  //other fields and methods
```

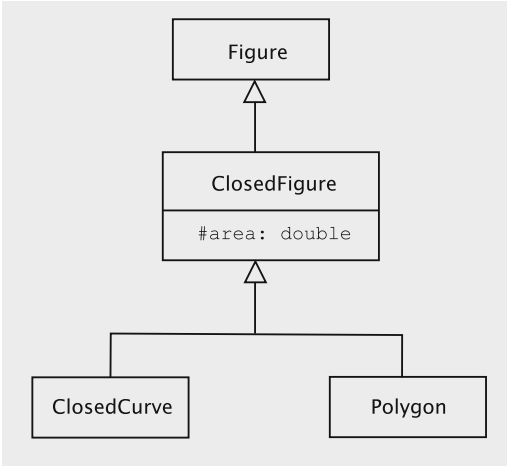

Fig. 3.11 Figure hierarchy

```
}

public class Polygon extends ClosedFigure {
  public void InsertVertex(Point p, int i) {
    // code to insert vertex at position i
    area = computeArea();
  }
  private double computeArea() {
    //code to compute the area
  }
}
```

Declaring it protected ensures that the field is available to the descendants but cannot be accessed by code that resides outside the hierarchy rooted at `ClosedFigure`.

The above example is a simple one since the class `Polygon` is modifying the field of a `Polygon` object. Consider the following situation.

```
public class ClosedCurve {
  // other fields and methods
  public void areaManipulator(Polygon p) {
    p.area = 0.0;
  }
}
```

Here the class `ClosedCurve` is modifying the area of a polygon. Our loose definition says that `area` is visible to `ClosedCurve` which would make this valid. However, `ClosedCurve`, is a sibling of `Polygon` and is therefore not a party to the design constraints of `Polygon`, and providing such access could compromise the integrity of our code. In fact, an unscrupulous client could easily do the following:

```
class BackDoor extends ClosedFigure {
  public void setArea(double area, ClosedFigure someClosedFigure) {
    someClosedFigure.area = area;
  }
}
```

We therefore need the following stricter definition of protected access.

The code residing in a class A may access a protected attribute of an object of class B only if B is at least of type A, i.e., B belongs to the hierarchy rooted at A.

With this definition, methods such as `setArea` in `BackDoor` would violate the protected access (since `ClosedFigure` is not a subclass of `BackDoor`) and can be caught at compile time. The compiler will not raise an objection if `someClosedFigure` is cast as `BackDoor` as shown below.

```
((BackDoor) someClosedFigure).area = area;
```

If `someClosedFigure` contained a reference to a `Polygon` object, the cast would fail at runtime preventing the access to the protected field.

3.2.5 The `Object` Class

Java has a special class called `Object` from which every class inherits. In other words, `Object` is a superclass of every class in Java and is at the root of class hierarchy. From our knowledge of polymorphic assignments, we can see that a variable of type `Object` can store the reference to an object of any other type. The following code is thus legal.

```
Object anyObject;
anyObject = new Student();
anyObject = new Integer(4);
anyObject = "Some string";
```

In the above, the variable `anyObject` first stores a `Student` object, then an `Integer` object, and finally a `String` object.

3.3 Genericity

Genericity is a mechanism for creating entities that vary only in the types of their parameters, and this notion can be associated with any entity (class or method) that requires parameters of some specific types. As we have seen before, in the definition of any entity, the types of involved parameters are specified. In case of a method, we specify the types of arguments and the return type. In case of a class, the types of arguments to the constructor(s), the return types and argument types of the methods are all specified. In any instance of the entity, the actual types of all these parameters must conform to the corresponding types specified in the definition. When we specify a generic entity, the types of the parameters are replaced by placeholders, which are called *generic parameters*. The entity is therefore *not fully specified* and cannot be used as such to instantiate any concrete objects. At the time of creating artifacts

(objects, if our generic entity was a class), these placeholders must be replaced by actual types.

To understand the usefulness of genericity, consider the following implementation of a stack:

```
public class Stack {
  private class StackNode {
    Object data;
    StackNode next;
    // rest of the class not shown
  }
  public void push(Object data) {
    // implementation not shown
  }
  public Object pop() {
    // implementation not shown
  }
  // rest of the class not shown
}
```

Elements of the stack are stored in the `data` field of `StackNode`. Notice that `data` is of type `Object`, which means that any type of data can be stored in it.

We create a stack and store an `Integer` object in it.

```
Stack myIntStack = new Stack(); // line 1
myIntStack.push(new Integer(5)); // line 2
Integer x = (Integer) myIntStack.pop(); //line 3
```

This implementation has some drawbacks. In `line 2`, there is nothing that prevents us from pushing arbitrary objects into the stack. The following code, for instance, is perfectly valid.

```
Stack myIntStack = new Stack();
myIntStack.push("A string");
```

The reason for this is that the `Stack` class creates a stack of `Object` and will, therefore, accept any object as an argument for `push`. The second drawback follows from the same cause; the following code will generate an error.

```
Stack myIntStack = new Stack();
myIntStack.push("A string");
Integer x = (Integer) myIntStack.pop(); // erroneous cast
```

We could write extra code that handles the errors due to the erroneous cast, but it does not make for readable code. On the other hand, we could write a separate `Stack` class for every kind of stack that we need, but then we are unable to reuse our code.

Generics provides us with a way out of this dilemma. A generic `Stack` class would be defined something like this:

```
public class Stack<E> {
  //code for fields and constructors
   public void push(E item) {
     // code to push item into stack
   }
   public E pop() {
```

```
        // code to push item into stack
      }
  }
```

A `Stack` that stores only `Integer` objects can now be defined as

```
Stack<Integer>myIntStack = new Stack<Integer>();
```

The statement

```
myIntStack.push("A string");
```

will trigger an error message from the compiler, which expects that the parameter to the `push` method of `myIntStack` will be a subtype of `Integer`.

3.4 Discussion and Further Reading

In this chapter we have discussed how classes in an object-oriented system relate to one another. Association is the simplest and most general of these. Although this chapter touches on several aspects of associations, a more detailed study of UML notation and some of the finer points of using associations would be needed before embarking on a serious project. UML notation provides a mechanism for another kind of relationship between classes, called a **dependency**. A dependency occurs when a client class has knowledge of some aspect of a supplier class and a change in the supplier class could affect the client. A detailed treatment of class relationships and other related issues can be found in [1].

A thorough knowledge of inheritance is vital to anyone engaging in OOAD. While the notion of a class helps us implement abstract data types, it is inheritance that makes the object-oriented paradigm so powerful. Inheriting from a superclass makes it possible not only to reuse existing code in the superclass, but also to view instances of all subclasses as members of the superclass type. Polymorphic assignments combined with dynamic binding of methods makes it possible to allow uniform processing of objects without having to worry about their exact types.

Dynamic binding is implemented using a table of method pointers that give the address of the methods in the class. When a method is overridden, the table in the extending class points to the new definition of the method. For an easily understandable treatment of this approach, the reader may consult Eckel [2].

There is some overhead associated with dynamic binding. In C++, the programmer can specify that a method is *virtual*, which means that dynamic binding will be used during method invocation. Methods not defined as virtual will be called using the declared type of the reference used in the call. This helps the programmer avoid the overhead associated with dynamic binding in method calls that do not really need the power of dynamic binding. In C++ parlance, all Java methods are virtual.

In Java, it is important to note that dynamic binding is not tied to subclassing. It is also applicable in the context of interfaces. For instance, consider the situation where `Student` is not a class, but an interface.

```
public interface Student {
  public boolean isInGoodStanding();
  public abstract double getGPACutoff();
  public String getName();
  // more methods
}
```

Let us assume that the above interface is implemented by the classes `UndergraduateStudent` and `GraduateStudent`. The implementation is simple enough, so we do not show the code for it; the only major difference now is that since there is no subclassing, the `isInGoodStanding()` of `GraduateStudent` cannot issue the call `super.isInGoodStanding()` but must compute it locally.

Now, the code given earlier and reproduced below, works via dynamic binding.

```
for (int index = 0; index < students.size(); index++) {
  if (students.getStudentAt(index).isInGoodStanding()) {
    System.out.println(students.get(index).getName()
                       + "is in good standing");
  } else {
    System.out.println(students.get(index).getName()
                     + "is not in good standing");
  }
}
```

Genericity is a very restrictive relationship that can exist between classes and is not particularly associated with OOAD. However, it is available in most object-oriented languages and must be used judiciously to facilitate reuse.

3.4.1 A Generalised Notion of Conformance

Most high-level languages perform some kind of type-checking when an assignment is done. This checking is used to ascertain that the type of entity returned by the expression on the left-hand side (LHS) of the assignment can indeed be stored in the type of entity referenced on the RHS. In other words, we say that the type of entity returned by the expression on the left-hand side (LHS) of the assignment conforms to the type of entity referenced on the RHS. If conformance is not there, some kind of casting is required, but the results of the casts cannot be guaranteed by a compiler since they depend on run-time behaviour.

In the context of inheritance, we have seen that a subclass conforms to the type of the superclass. When we add genericity to the mix, and the expression on the LHS evaluates to an instance of a generically defined entity; the corresponding generic parameters of the LHS and RHS must also be in conformance. This check would have to be performed recursively since the parameters could themselves be generically derived [3]. Given the following definitions,

```
public class Polygon {
  // code for Polygon
}
```

```
public class Triangle extends Polygon {
  // code for Triangle
}

public class Square extends Polygon {
  // code for Square
}
```

the generic types `Stack<Square>` and `Stack<Triangle>` conform to `Stack<Polygon>`. However, an assignment of the kind shown below is flagged by a Java compiler.

```
Stack<Square> ssq = new Stack<Square>();
Stack<Polygon> sp = ssq; // Compiler Error!
```

The reason for this appears to be that generics being a later introduction to Java, interoperability with legacy code was required. This was achieved by a mechanism called *erasure*, which resulted in all generic type information being erased at compile time. This implies that if the above statement was not flagged as an error, there is no way that the system could prevent the pushing of a triangle on a stack of squares.

```
Stack<Square> ssq = new Stack<Square>();
Stack<Polygon> sp = ssq;
sp.push(new Triangle()); // no way to detect this
```

Some languages allow for *dynamic casts* which is one way that this situation can be handled. In C++, for instance, the following code would compile, but generate a run-time error [4].

```
Stack<Triangle> * TStack = new  Stack<Triangle>();
Stack <Polygon> * PStack;
PStack = dynamic_cast<Stack <Polygon> *> (TStack); // valid, types conform
Square * s1 = new Square();
Polygon * p1 = dynamic_cast<Polygon*>(s1);
PStack->push(*p1); // run-time error
```

The system keeps track of the fact that `PStack` is a pointer to a `Stack<Triangle>` and that `*p1` is in fact a `Square`.

Projects

1. Implement the interface `Extendable` in Programming Project 3 in Chap. 3 with a class named `AbstractBuffer`. This class stores an array of `chars` whose initial capacity is passed via a constructor.
   The class must have two fields, both `protected`; one stores the `char` array and the other stores the number of elements actually filled in the array.
   Do not implement either of the interface methods. So the class is declared `abstract`.
   This class must also implement the `toString()` method to correctly bring back a `String` representation of the `char` array.
   Next, implement `SimpleBuffer` so that it extends `AbstractBuffer` and actually implements the interface methods correctly. As before, it has a constructor that accepts the size of the array.

2. Consider the interface `Shape` given below.

```
public interface Shape {
  public double getArea();
  public double getPerimeter();
  public void draw();
}
```

Design and code two classes, `Rectangle` and `Circle`, that implement `Shape`. Put as many common attributes and methods as possible in an abstract class from which `Rectangle` and `Circle` inherit. Ensure that your code is modular. For drawing a shape, simply print the shape type and other information associated with the object.
Next, implement the following interface using any strategy you like. The interface maintains a collection of shapes. The draw method draws every shape in the collection.

```
public interface Shapes {
  public void add(Shape shape);
  public void draw();
}
```

Then, test your implementation by writing a driver that creates some `Shape` objects, puts them in the collection and draws them.
Finally, draw the UML diagram for the classes and interfaces you developed for this exercise.

3. The following interface specifies a data source which consists of a number of x-values and the corresponding set of y-values. The method `getNumberOfPoints` returns the number of x-values for which there is a corresponding y-value. `getX` (`getY`) returns the x-value (y-value) for a specific index ($0 \leq$ `index` $<$ `getNumberOfPoints`).

```
public interface DataSource {
  public int getNumberOfPoints();
  public int getX(int index);
  public int getY(int index);
}
```

The next interface is for a chart that can be used to display a specific data source.

```
public interface Chart {
  public void setDataSource(DataSource sourse);
  public void display();
}
```

A user will create a `DataSource` object, put some values in it, create a `Chart` object, use the former as the data source for the latter and then call `display` to display the data.
Here is a possible use. Note that `MyDataSource` and `LineChart` are implementations of `DataSource` and `Chart` respectively.

```
DataSource source = new MyDataSource();
Char chart = new LineChart();
chart.setDataSource(source);
chart.display();
```

Implement the interface `DataSource` in a class `MyDataSource`. Have methods in it to store x and y values.
Provide two implementations of `Chart`: `LineChart` and `BarChart`. For displaying the chart, simply print out the x and y values and the type of chart being printed. If needed, put the common functionality in an abstract superclass.
Draw the UML diagram for your design.

4. Implement three classes:
`BinaryTreeNode`, `BinaryTree` and `BinarySearchTree`.
The first class implements the functionality of a node in a binary tree, the second is an abstract class that has methods for visiting the tree, computing its height, etc., and the third class extends the second to implement the functionality of a binary search tree.

3.5 Exercises

1. Trace the following code and write that the program prints

```
public class A {
  protected int i;
  public void modify(int x) {
    i = x + 8;
    System.out.println("A: i is" + i);
  }
  public int getI() {
    System.out.println("A: i is" + i);
    return i;
  }
}
public class B extends A {
  protected int j;
  public void modify(int x) {
    System.out.println("B: x is" + x);
    super.modify(x);
    j = x + 2;
    System.out.println("B: j is" + j);
  }
  public int getI() {
    System.out.println("B: j is" + j);
    return super.getI() + j;
  }
}
public class UseB {
  public static void main(String[] s) {
    A a1 = new A();
    a1.modify(4);
    System.out.println(a1.getI());
    B b1 = new B();
    b1.modify(5);
    System.out.println(b1.getI());
    a1 = b1;
    a1.modify(6);
    System.out.println(a1.getI());
  }
}
```

2. Consider the class `Rectangle` in Programming Exercise 2. Extend it to implement a square.
3. A manager at a small zoo instructs the zoo-keeper to 'feed the animals'. Explain how a proper completion of this task by the zoo-keeper implies that the zoo operations are implicitly employing the concepts of inheritance, polymorphism and dynamic binding. (Hint: defining a class `Animal` with method `feed` could prove helpful.)

References

1. C. Larman, *Applying UML and Patterns* (Prentice Hall PTR, New Jersey, 1998)
2. B. Eckel, *Thinking in C++ Volume 1 (2nd Edition)* (Prentice Hall, New Jersey, 2000)
3. B. Meyer, *Object-Oriented Software Construction* (Prentice Hall, New Jersey, 1997)
4. P. Anderson, G. Anderson, *Navigating C++ and Object-Oriented Design* (Prentice Hall, New Jersey, 1998)