Chapter 9
Exploring Inheritance

9.1 Introduction

In this chapter we look more deeply at the topic of inheritance, the basic concepts of which were covered in Chap. 3. Inheritance can be done in two ways: by subclassing existing classes and by implementing interfaces. The major goal of inheritance is reuse, but the two approaches achieve this goal in different ways.

We begin this chapter by exploring the idea of subclassing. Using inheritance effectively is not always a straightforward exercise. We show several examples that illustrate distinct circumstances where this approach can be taken.

Like most tools, subclassing must be done with care, or we may end up with an unstable system. In Sect. 9.3 we present some of the considerations while subclassing and an alternative to this approach. An elegant test for deciding whether subclassing is appropriate in a given context is enunciated in the Liskov substitution principle (LSP), which we study in this context.

Section 9.4 discusses the technique of implementing interfaces, with particular reference to Java. While many cases that use this approach are straightforward enough, one does encounter some tricky situations.

Section 9.5 revisits the case-study for the library system and enhances it to include multiple kinds of items. A 'process-oriented' enhancement is presented and critiqued in Sect. 9.6 before presenting an object-oriented solution incorporating inheritance.

Introducing inheritance and replacing a single class with a hierarchy could complicate some issues such as exception handling and functionality enhancement. These are addressed in Sect. 9.7.

Some object-oriented languages permit a class to subclass multiple classes. While this can be useful in some situations, the technique is also quite complicated. The approach and its pros and cons are covered in Sect. 9.8, followed by discussion and suggestions for further reading.

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_9

9.2 Applications of Inheritance

This section presents varied applications of inheritance illustrating the different circumstances in which this powerful technique can be used. Although the programming language rules concerning inheritance are not overly complicated as long as multiple inheritance is not involved, the design process can sometimes be tricky. A great deal of insight into how the system would evolve is essential for clean and effective use of inheritance.

9.2.1 Restricting Behaviours and Properties

One circumstance in which inheritance can be applied is when a class has characteristics that are a restriction of the characteristics of some other class. Suppose we have two classes, `Rectangle` and `Square`, to represent rectangles and squares. Every square is a rectangle in which length is equal to breadth, i.e., *the property that length be equal to breadth restricts the number of rectangles that qualify to be classified as squares.* Thus, `Square` is obtained from `Rectangle` by restricting a property; note that we are not attaching any more functionality to squares than rectangles.

As a second example, suppose that we create a graphical user interface with many types of widgets, including labels. Suppose we have the requirement that the text in all labels be coloured blue. In this case, it is convenient to have a subclass that simply sets the colour to blue. The subclass of JLabel is given below.

```
import java.awt.colour;
import javax.swing.JLabel;
public class SpecialLabel extends JLabel {
  public SpecialLabel(String text) {
    super(text);
    setForeground(colour.blue);
  }
}
```

In this case, the behaviour of `SpecialLabel` is restricted in that it always displays a blue foreground.

Although the above examples make perfect sense from an abstract view point, the reader should note that a well-established principle of inheritance shows that inheritance might not be justified. Please see Sects. 9.3.4 and 9.9 and Exercise 6.

9.2.2 Abstract Superclass

Sometimes the only purpose of having a superclass is to extract the common attributes and methods of potential subclasses, thus maximising reuse. No objects of the superclass itself are allowed, thus necessitating that the superclass be abstract. In such

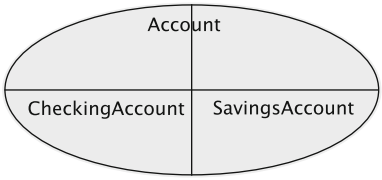

Fig. 9.1 Partitioning a set of objects

cases we have a set of **subclasses that partition the universe** of objects in the superclass.

As an example, consider accounts in a bank. An account is a general concept, and, perhaps, in some bank, all accounts are checking accounts or savings accounts. The bank allows only opening of checking or savings accounts. Therefore, Account is a class that helps us build software more quickly by providing some of the functionality that is common to all types of accounts. The partitioning is indicated in the Venn diagram in Fig. 9.1, which shows that the set of all accounts is partitioned into savings and checking accounts.

9.2.3 Adding Features

In Sect. 9.2.1, we have seen that sometimes classes are extended to restrict the behaviour of the superclass. Somewhat quite opposite to this, we may extend an ancestor class, by adding new features, to get the descendant. Consider a class named `DataStream` that serves as a reader of data. Imagine a situation where we need a class that has the added property of 'reReadability,' i.e., reading some input again. To achieve this, we add new functionality, viz., the ability of 'unreading' so that a character read from the stream can be pushed back. This is shown in Fig. 9.2. Another example of this would be a class for 'Moving Vehicle' that can be defined by extending an existing 'Vehicle' and adding the attribute 'speed.'

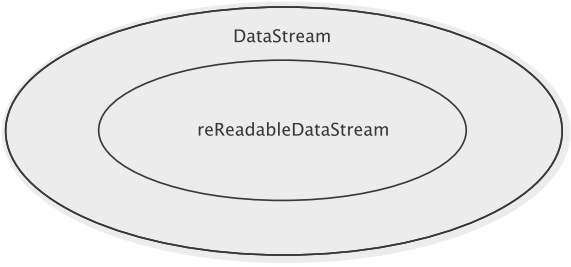

Fig. 9.2 Adding more features

9.2.4 Hiding Features of the Superclass

Sometimes we want to restrict behaviour by suppressing some functionality of the superclass. Such a kind of restriction is discussed in the following example, where some of the features are eliminated in the subclass.

Let `List` be a class that allows the creation of a list in which objects can be added anywhere: in the front, at the tail, or at any position in-between. It is easy to get a class `Queue` that allows adding only at the tail and removing from the front. All other add and remove methods, which allow adding at or removing from other positions, should be disallowed. This can be accomplished in C++ via private[1] inheritance, as shown below:

```
#include <iostream.h>
#include <stdlib.h>

class List {
  private:
    // data structures
  public:
    List() {
      // initialize data structures
    }
    bool add(int index, int value) {
    // code to add at the specified position and return true or false
    }
    bool add(int value) {
    // code to add at the end and return true or false
    }
    int remove(int index) {
    // code to delete the object at the specified position and
    // return true or false
    }
    int remove() {
    // code to delete the object at the front and return true or false
    }
};
class Queue: private List {
  public:
    int dequeue() {
      return List::remove();
    }
    bool enqueue(int value) {
      return List::add(value);
   }
};
```

Such an application has also been referred to in the literature as **Structural Inheritance** because the features inherited from the superclass (`List`) provide the structure needed for implementing the subclass (`Queue`). This kind of an application does have its critics due to the fact that the 'is-a' relationship between the ancestor and the descendant is not preserved.

[1] In private inheritance, all the non-private superclass attributes become private attributes of the subclass.

9.2.5 Combining Structural and Type Inheritance

We can also have situations where two kinds of inheritance are applied to define a class that suits our application. The most common of such situation is one where one superclass provides the necessary **structure** and another one, usually an interface, defines the **function**. A *binary search tree*, for instance, can be seen as a class that extends *binary tree* (structure inheritance) and implements the `OrderedList` interface (which defines the function of the binary search tree). The `OrderedList` operations are *implemented* using the methods provided by the class representing the binary tree, giving the name **implementation inheritance** to this usage.

9.3 Inheritance: Some Limitations and Caveats

Although it facilitates reuse, inheritance by subclassing is not always the best strategy to construct new classes even if there is justification for doing so on the surface. Among the reasons:

1. Subclassing could result in deep hierarchies, which usually makes it quite difficult to understand the code.
2. In systems that do not support multiple inheritance, subclassing is not always feasible.
3. It may be necessary to hide selected features of the superclass. For example, if we extend the class `java.util.LinkedList` to implement a queue, all of the methods of the superclass will be exposed, which may compromise integrity. Facilities such as the `renames` clause in the language Eiffel facilitate this. Explicitly hiding a superclass's field/method is also possible in C++.
4. Combining inheritance with genericity may result in complications due to implementation issues with a particular language. As we saw in Chap. 8, the erasure property employed by Java results in some inconsistencies when inheritance and genericity are combined.
5. The derived class's type may not be a true subtype of the superclass's type.

We elaborate on each of the above aspects in the following subsections.

9.3.1 Deep Hierarchies

When subclassing, we obviously add one more to the length of the hierarchy. The fields and methods available for use in the derived class include all of the inherited fields and methods and the ones added to the class itself. For example, the class `JFormattedTextField` in the package `javax.swing` has a hierarchy of depth 7, assuming that `java.lang.Object` is at depth 1. Table in Fig. 9.3 shows how the number of fields and methods increase for this specific hierarchy.

Class name	Number of Fields	Number of Methods
java.awt.Component	186	291
java.awt.Container	230	417
javax.swing.JComponent	376	594
javax.swing.text.JTextComponent	440	698
javax.swing.JTextField	479	729
javax.swing.JFormattedTextField	513	757

Fig. 9.3 Complexity increase with hierarchy depth

It is a challenging task to remember the interactions between the methods. Clearly, it is advisable to keep the hierarchy to a reasonable depth.

9.3.2 Lack of Multiple Inheritance

In certain situations, it may be desirable to create a subclass from multiple classes. However, some languages such as Java allow subclassing of at most one class. In such circumstances, we cannot limit ourselves to subclassing, but adopt other approaches in conjunction with it, or abandon subclassing altogether. We discuss this issue in Sect. 9.4.

9.3.3 Changes in the Superclass

While it is not desirable to change the set of methods supported by a class, such changes are sometimes inevitable. (As an example, in the Java class system the class `java.awt.Component` added the public method `setEnabled(boolean)` in JDK 1.1.) Imagine an application system A_1, some classes of which extend a set of classes from some other system A_2. Suppose that A_2 is modified to include a number of useful features. To exploit these enhancements, assume that the corresponding subclasses of A_1 use the new versions.

Although A_1 can now exploit all of the new functionality incorporated into the classes of A_2, there are potential problems as well. To see one possible problem, consider the two classes `B` and `D` given below, where `D` extends `B`.

```
public class B {
  public void m1() {
  }
}

public class D extends B {
  public void m2() {
  }
}
```

Suppose that the following method is now added to `B`

```
public int m2() {
  return 1;
}
```

Now class `D` is illegal because method `m2`'s return type is inconsistent with that of the correspondingly-named method in `B`.

9.3.4 Typing Issues: The Liskov Substitution Principle

One of the rules that is implicit in the use of inheritance is the *Liskov substitution principle (LSP)* which is stated as follows:

Subclasses should be substitutable for their baseclasses.

The concept seems rather obvious, and at first glance, one wonders what the fuss is all about. After all, this is the essence of the *is-a* relationship of inheritance. To see its significance, let us first quote Liskov.

> If for each object O_1 of type S there is an object O_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when O_1 is substituted for O_2 then S is a subtype of T.

The subtleties involved in the definition can be brought out through the following example.

A package provides a class `SolidRectangle` that creates a solid (i.e., all the pixels in the rectangle are filled), axis-parallel (or *isothetic*) rectangle. Each `SolidRectangle` object is defined by two points, which are the ends of one of the diagonals.

Let us define an *upper triangle* of a solid rectangle as the triangle formed by the end-points of one of the diagonals and the corner of the rectangle 'above' the diagonal. See Fig. 9.4, where the upper triangle is shown shaded. The two rectangles on the left have the diagonal connect the top-left and bottom-right corners of the rectangle. Therefore, the upper triangle comprises the top-left, bottom-right and top-right corners of the rectangle. Similarly, in other two cases, the top-left, bottom-left, and top-right corners constitute the upper triangle. The reader can verify that when `corner1.y > corner2.y` (see the top row of Fig. 9.4), the third point is formed using *corner2.x* and *corner1.y*; otherwise, the third point is formed using *corner1.x* and *corner2.y*.

Partial code for the class is shown below. The `getUpperTriangle` method returns the `Triangle` object corresponding to the upper triangle formed using the diagonal connecting `corner1` and `corner2`.

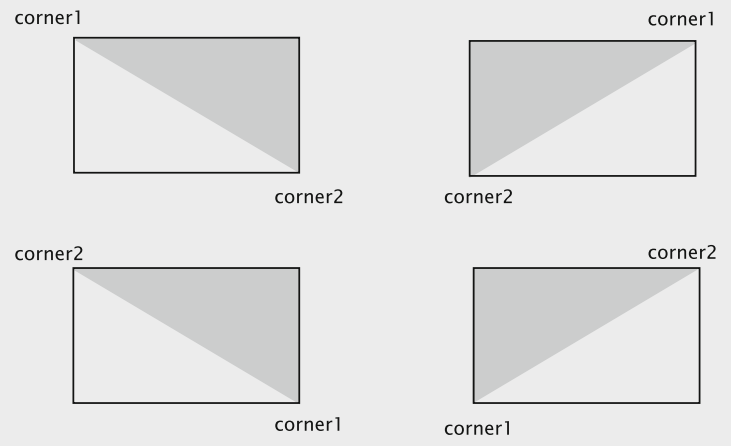

Fig. 9.4 Upper triangle (*shaded*) of a `SolidRectangle`

```
class SolidRectangle {
  private Point corner1;
  private Point corner2;
  public SolidRectangle(Point point1, Point point2) {
    corner1 = point1;
    corner2 = point2;
  }
  public void setCorner1(Point point) {
    corner1 = point;
  }
  public void setCorner2(Point point) {
    corner2 = point;
  }
  public Triangle getUpperTriangle() {
    Point point;
    if ((corner1.x == corner2.x) || (corner1.y == corner2.y)) {
      return null; // degenerate case
    } else {
      if (corner1.y > corner2.y) {
        point = new Point(corner2.x, corner1.y);
      } else {
        point = new Point(corner1.x, corner2.y);
        return (new Triangle(corner1, corner2, point));
      }
    }
  }
}
```

In some situations, it could conceivably be convenient to have a separate class for dealing with an individual pixel, which is just a 1×1 rectangle. This suggests that we could simply extend `SolidRectangle` to accommodate this.

```
class Pixel extends SolidRectangle {
  public Pixel(Point point) {
    super(point, point);
  }
}
```

Note that the set of `Pixel` objects is just a subset of the `SolidRectangle` object with the restriction `corner1 = corner2.` Accordingly, the methods for setting these corners should be redefined in the subclass so that this invariant property is preserved.

```
public void setCorner1(Point point) {
  super.setCorner1(point);
  super.setCorner2(point);
}
public void setCorner2(Point point) {
  super.setCorner1(point);
  super.setCorner2(point);
}
```

The `getUpperTriangle` method does not pose any problem for our `Pixel` class, since it will always return `null`. Our troubles start with existing client classes, whose methods have been using the methods of `SolidRectangle`. Consider the following method in a client class that takes as its input parameter a `SolidRectangle` object.

```
public void clientMethod(SolidRectangle rectangle, Point point) {
  Triangle triangle1;
  //... some code
  rectangle.setCorner1(new point(1, 1));
  rectangle.setCorner2(new point(4, 4));
  triangle1 = rectangle.getUpperTriangle();
  if (triangle1.contains(p)) {
    //... some code
  }
}
```

This code was written by a client using `SolidRectangle`, but unaware of `Pixel`. Since the two corners are set to two distinct points, `triangle1` will not be null, and things will be fine. Now if this method is invoked and a reference to a `Pixel` object is passed as the actual parameter, both the corners will end up being assigned the same point, `triangle1` will be `null`, and we end up with a `NullPointerException`. Note that there is no simple fix; we may just have to find such code in all the methods of the client classes, and either check for null pointers or check the type of the objects stored in the `SolidRectangle` references at runtime.

At every step of the this process our choices seemed logical, but we ended up with an undesirable state of affairs. So it is natural to ask: *what went wrong?* The answer lies in a precise definition the *is-a* relationship: *A pixel object 'is-a' SolidRectangle object if and only if the behaviour of Pixel objects conforms to the behaviour of SolidRectangle objects in* **all** *situations.* The above example shows that the behaviour of `getUpperTriangle` does not exhibit such conformance when invoked in conjunction with the `setCorner` methods. One could argue, perhaps, that the method `getUpperTriangle` is itself poorly designed, but that is really not a choice that we can make now. It is important to keep in mind that we are inheriting from the existing class `SolidRectangle` and we must accept all its methods. To cast this in terms of the formal statement of the LSP, when we substitute `Pixel`

for `SolidRectangle` the behaviour of any program should remain unchanged. What we have shown here is that for the program `clientMethod`, the behaviour changes when such a substitution is made.

The caveat here for the programmer extending a class is therefore that *one must check all behaviours of the class being extended, even in situations where a subset relationship exists between the corresponding 'real-world' entities.*

9.3.5 Addressing the Limitations

The object-oriented approaches to circumvent these limitations are founded on two thumb rules:

- Inherit from abstract types rather than concrete classes.
- Favour composition over inheritance.

In most situations where we have to use inheritance effectively, some combination of these two thumb rules comes in handy. A simple illustrative example is described below.

Consider a situation where a concrete class `C1` exists and we want to create a descendant `C2`. We could do this by having `C2` extend `C1`, but such a solution would suffer from the problems listed earlier. A better approach would be to define an abstract type `C` which is implemented by `C1`. Now `C2` can be defined as another concrete class that extends `C`. In order to reuse the earlier implementation, we have two strategies available. One strategy is to define `C` as an abstract class, factor out the parts of the implementation that are likely to be used and place these in `C`. Each of `C1` and `C2` then put in the details specific to their types. A second strategy would be to define `C2` as a class that implements `C` and *adapts* the implementation provided by `C1`.

9.4 Type Inheritance

So far, we have discussed cases where a class subclassed another to inherit its properties and behaviours. But reuse need not be realised simply via subclassing. It can also be achieved by inheriting behaviour.

To see how this can be achieved, assume that we have a class C that performs some useful function f on all objects of type T. Then, a class D that wishes to utilise this functionality can do so by acquiring the type T. In Java parlance, for example, T might be an interface that class D implements.

9.4.1 A Simple Example

To make these ideas more concrete, let us look at an example. In the `java.util` package, there is a class called `sort` that as the name implies, sorts objects of type `Comparable`. Here is the essential declaration of that method.

```
static T    sort(List<T> list)
```

where `T` must be of type `Comparable`.
`Comparable` is an interface with just one method `compareTo`, which has the following signature.

```
int compareTo(T object)
```

Instances of any class that implements the above interface acquire the ability to be sorted using the `sort` method of `Collections`.

Suppose that we have a class `City` that stores the name, state, and population of cities. To make objects of type `City` comparable, we need to have it implement the `Comparable` interface. Each `City` object maintains a reference to the corresponding `State`. So the `compareTo` method of the former employs the `compareTo` method of the latter to complete its work. Here is the code.

```
public class City implements Comparable {
  private String name;
  private State state;
  private int population;
  public City(String name, State state, int population) {
    this.name = name;
    this.state = state;
    this.population = population;
  }
  public int compareTo(Object object) {
    City city = (City) object;
    int result = 0;
    if ((result = name.compareTo(city.name)) == 0) {
      return state.compareTo(city.state);
    }
    return result;
  }
  public boolean equals(City city) {
    return compareTo(city) == 0;
  }
}
```

The class `State` must also be `Comparable` as shown below.

```
public class State implements Comparable {
  private String name;
  public int compareTo(Object other) {
    State state = (State) other;
    return name.compareTo(state.name);
  }
  // other fields and methods
}
```

Inheriting a property is not always as simple as the above example suggests, and Java has two other interfaces, viz., `Cloneable` and `Runnable`, which exemplify some of the subtleties.

9.4.2 The Cloneable Interface

The property of self-replication comes in handy when we are dealing with objects. Due to the complex interconnections, sending out a reference to the original copy of just one object can result in compromising the integrity of the entire system. As we have seen in our case-study with the library system, a `Member` object can store references to several `Book` objects, each of which could have several references to `Hold` objects. Since each `Hold` object has a reference to a member, one could potentially access (or modify) the information about all the members through a single reference. This problem can be avoided by not sending a reference to the original object, but a reference to a carefully constructed clone.

Implementing such a solution is somewhat more complicated than it appears at first. A naive implementation strategy would be to simply do a bit-wise copy (sometimes called a 'shallow' copy) of all the fields of the user object. If we were to attempt this, consider the problem of making a copy of the holds placed by the user. The `Member` object has a field `booksOnHold` that holds a reference to the collection of holds. A bit-wise copy would simply create another reference to this collection. As a result, the method accessing the clone now has a reference to the original `Hold` objects. Creating a copy therefore requires some knowledge about the object being copied, and this knowledge is available only within the object being copied. As an example, the class `Member` must decide how members are copied. Since `Member` includes a collection of `Hold` objects, that collection must be cloned when its `Member` object is cloned. The rule is that every object should decide for itself how it will be cloned, so that the collection should be able to clone itself and provide the rule for its own cloning.

To add to the difficulty, in general, any given object that holds references to other objects has to rely on those objects to create clones of themselves. If some of those objects cannot be cloned, then the given object must have the ability to decide how the situation is to be handled. Accordingly, the mechanism provided by the language to impart the cloneability property to a class must be versatile enough to accommodate all the desirable possibilities.

The `Object` class comes with a protected `clone` method that does the simple bit-wise copy that we describe above. The `Cloneable` interface in Java imparts the property of cloneability to a class. This interface is empty, and a class may choose not to implement this interface, but may nonetheless have to override the `clone` method in `Object` to ensure that things are done correctly. Finally, we have a `CloneNotSupportedException`, which is thrown to signal that a class's `clone` method should not be invoked. The `clone` method in `Object` is declared to throw this exception, thus allowing all subclasses to throw the exception as well.

Any given class can have one of four possible attitudes toward cloning:

1. *Support clone* In this case the class implements the `Cloneable` interface and declares its `clone` method to throw no exceptions. The author of such a class must ensure that all its contents are `cloneable`.
2. *Conditionally support clone* This situation arises when a class implements `Cloneable`, but cannot guarantee that its contents can be cloned. In this case the `clone` method will throw the `CloneNotSupportedException` if some of its contents are not `Cloneable`.
3. *Not publicly support clone, but allow subclasses to support clone* Such a class does not implement `Cloneable`, but may override the default implementation of `clone` to ensure that it works correctly. This would enable its subclasses to invoke `super.clone()` if the subclasses choose to support clone.
4. *Forbid clone* In this case, the class does not implement `Cloneable` and provides a clone method that always throws the `CloneNotSupported Exception`.

As a first example, let us see how to implement the `Cloneable` interface for the `City` and `State` classes given earlier. The clone in `City` requires the following header.

```
public class City implements Comparable, Cloneable {
```

The `clone` method itself is given below. One thing to note is that the immutable classes do not need to be cloned if our only purpose is to protect the original copy. Any attempt to change an immutable object by modifying the clone will not change the original, even though we have a shallow copy. For instance, if an object is simply a collection of `String` objects, a bit-wise copy will create another object that holds references to the same `String` objects as the original. Attempting to modify a `String` creates a new `String` object and therefore the original object is always preserved.

After calling the `clone` method of the superclass, the `state` field is cloned by calling the `clone` method of the `State` class.

```
public Object clone() {
  City copy = null;
  try {
    super.clone();
    copy.state = (State) state.clone();
  } catch(CloneNotSupportedException cnse) {
    return null;
  }
  return copy;
}
```

The `clone` method of the `State` object is straightforward and we omit giving its code. The reader will notice that shallow cloning works just fine.

As a second example, consider cloning a `Member` object in the library system. To clone a user, the following changes are made to the code:

```
class Member implements Cloneable {
  // other fields and methods
  public Object clone() throws CloneNotSupportedException {
    Member member = (Member) super.clone();
    member.booksBorrowed = new LinkedList();
    member.booksOnHold = new LinkedList();
    member.transactions = new LinkedList();
    for (ListIterator iterator = booksBorrowed.listIterator();
                      iterator.hasNext(); ) {
      member.booksBorrowed.add((Book)((Book) iterator.next()).clone());
    }
    for (ListIterator iterator = booksOnHold.listIterator();
                      iterator.hasNext(); ) {
      member.booksOnHold.add((Hold)((Hold) iterator.next()).clone());
    }
    for (ListIterator iterator = transactions.listIterator();
                      iterator.hasNext(); ) {
      member.transactions.add((Transaction)
               ((Transaction)iterator.next()).clone());
    }
    return member;
  }
}
```

Since the clone method for the class `LinkedList` provides only a shallow copy, it is necessary that we create new instances of `LinkedList` for `booksBorrowed`, `booksOnHold` and `transactions`, clone each item in the original versions of these lists and insert them into the clone.

`Member` is now conditionally supporting clone. The classes `Book`, `Hold` and `Transaction` will also have to support clone if we have to successfully clone `Member`. Although this code appears correct (and does in fact do the cloning correctly) it contains a serious flaw: *invoking the clone method could result in infinite recursion.* This is because the `Hold` object stores a reference to the member who placed the hold! The size of the cloned object also poses a problem: *since the* `Hold` *object also stores a reference to* `Book`*, which in turn stores a reference to the borrower, we could potentially clone all the information in the library.* A simple resolution to these problems could be that the `clone` methods in `Book` and `Hold` do not clone the `Member` fields and also set the `Member` references on cloned copies to `null`.

```
class Hold implements Cloneable {
  // other fields and methods
  public Object clone() throws cloneNotSupportedException {
    Hold hold  = (Hold) super.clone();
    hold.member = null;
    hold.book = (Book) book.clone();
    hold.date = (Calendar) date.clone();
    return hold;
  }
}
```

In this situation we see that cloning is serving a purpose beyond that of preserving the original copy. In general, cloning is a non-trivial exercise and we need to be aware

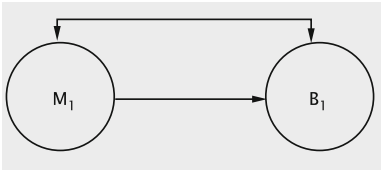

Fig. 9.5 Cloning in the general case: an example

of all the possible complications that can arise when properties like cloneability are inherited by a class.

For an approach that prevents infinite recursion, we could proceed as discussed in the following example. Our approach employs a list of all objects that we have started cloning and the corresponding clone's reference. As we start the cloning process, the list is empty. Suppose we have the situation where member M_1 has borrowed a single book B_1 and we begin the cloning process with M_1 (see Fig. 9.5). It is easy to verify that with no safeguards in place, the cloning will result in infinite recursion. Before cloning the fields of M_1, we add an entry corresponding to M_1 into the list and store the clone's reference. In the process of cloning M_1, we encounter the reference to B_1. Before we clone B_1, we create an entry for it and store the reference to B_1's clone. While cloning the fields of B_1, we encounter M_1 and observe from the list that that we have already started cloning M_1, which means that it should not be cloned again (to prevent infinite recursion). We obtain M_1's clone's reference from the list and store it in B_1's clone.

9.4.3 The Runnable Interface

Software systems often have to employ runtime structures called *concurrent sequential processes (CSPs)*. A simple example of a CSP can be found in the implementation of a bank account. The customer (owner of the account) could be using an ATM to withdraw cash at the same time when a transaction for depositing a check is being processed. Both these actions represent concurrent processes which are accessing a piece of shared data (the account information). In this situation it is vital to ensure that both processes do not try to simultaneously modify the `balance` field in the account; in such a simultaneous access, depending on the order in which the methods for deposit and withdrawal are executed, we could end up with an error.

Java provides a class called `Thread` that can be used for implementing CSPs. In our example, we would have a separate thread for every process that accesses the account object; that is, if a withdrawal and deposit were to be simultaneously carried out, there would be a thread for the withdrawal process and another thread for the deposit. We can then employ *Mutual Exclusion* on the account object, which will ensure that the account is not simultaneously accessed by the two processes. Threads also enable the programmer to implement other operations like suspending

a process, making a process sleep for a specified amount of time, prioritise processes, and enable sharing of resources.

Since `Thread` is a class, we can create a new class that has all the features of `Thread` through inheritance. However, since Java does not support multiple inheritance, this approach does not allow the user to create a class that extends an existing class and also has all the properties of `Thread`. This restriction is overcome by providing the `Runnable` interface as a part of the language. This interface allows the programmer to create a class that has all the properties of `Thread`. The following example illustrates the use of this interface to create a simple 'clock' that prints 'tic' and 'toc' at regular intervals.

```
public class Clock implements Runnable {
  Thread thread = new Thread(this);
  String sound = "tic";
  public void run() {
    try {
      while (true) {
        System.out.println(sound);
        sound = "toc";
        Thread.sleep(1000);
        System.out.println(sound);
        sound = "tic";
        Thread.sleep(1000);
      }
    } catch (InterruptedException ie){
    }
  }
  public Clock() {
    thread.start();
  }
  public static void main (String[] args) {
    new Clock();
  }
}
```

The `run` method contains an infinite loop that alternately prints the words 'tic' and 'toc' with a 1000 ms delay between consecutive words. This process continues until interrupted by the user.

The `Runnable` interface requires that `Clock` implement the `run` method. Inside `Clock` we create a `Thread`, storing this reference in `thread`. A reference to the `Clock` object is passed to `thread`. (The parameter being supplied here is required to be of type `Runnable`.) The `start` method of `Thread` is invoked in the `Clock` constructor, and this method in turn invokes the `run` method of the `Clock` object. In this interface, a `Thread` object is encapsulated along with the class to provide 'thread-like' properties to objects of the class. In essence, what we have done here is to *achieve the benefits of inheritance using* **object composition.**

Object Composition and Inheritance are the two most common mechanisms for reuse in object-oriented design. In both these mechanisms, the implementation of the new class is defined in terms of the functionality of the existing classes. In case of inheritance, since the internal details of the ancestor class are often visible to the descendant, the resulting scenario is referred to as 'white-box reuse'. On the

other hand, object composition requires that the component classes have well-defined interfaces so that they can be used as 'black-boxes'.

Object composition has the obvious advantage of keeping the inheritance hierarchies small, thus reducing the complexity of the system. Also, properties that are not naturally tied to one another are kept separate, so that each class is kept encapsulated and focused on one task. A less obvious advantage is that composition allows us to define the object dynamically at run-time. In the above example, `Clock` has been defined *statically,* since `Thread` will always hold a reference to a `Thread` object. The following example shows how composition can be used to define an object *dynamically.*

```
public class Catalog {
  private List catalogList;
  public Catalog (List list) {
    catalogList = list;
    //other constructor code
  }
  // other fields and methods
}
```

In this case, any object belonging to any class that satisfies the `List` interface can be sent in as a parameter to the constructor. For instance, we could do the following:

```
catalog = new Catalog(new ArrayList());
```

This would create a catalog that uses an `ArrayList` to store the collection of books. Note that such a dynamic definition is possible because `Catalog` is defined by composition; if it had been defined as an extension of, say, `LinkedList`, this would not be possible.

9.5 Making Enhancements to the Library Class

We are now ready to move ahead with the process of employing inheritance and creating hierarchies in our design. Consider a more sophisticated version of the `Library` system that we created in the last chapter. With the advent of technology, our clients now wish to expand their collection to include non-print media. Thus we now have books on tape, CDs, and DVDs in addition to printed books. Also, the library wants to include some periodicals, which are to be handled differently from the other books. For instance, recent periodicals cannot be checked out. For books on tape, CDs and DVDs, we wish to keep track of the duration.

9.5.1 A First Attempt

As a first step in developing our new design, let us ask ourselves the question: *How do these new requirements change the design of our system?* or in more concrete terms, *What new classes/methods need to be added?* and *How do the interactions between*

the existing classes change? To answer these, let us examine how the requirements have changed the way in which the business processes are carried out. Consider the use case for issuing a book. The operations needed are the same: viz., *check issuability, compute a due-date and record the transaction.* We could handle these simply by making changes in the methods of the existing `Book` class.

To simplify the discussion, we restrict ourselves to two types of items that the library lends: books and periodicals. Even with this simplification, we need a mechanism to find out what item (i.e., a book or a periodical) we are dealing with when we process these transactions. One approach could be to add a field `bookType` to the class `Book`, which would tell us what kind of a `book` it is. We would make changes in the method that computes the due-date by switching on the field `bookType`. Periodicals that are less than three months old are not issuable; otherwise, they can be borrowed for a week. Another difference is that periodicals have no authors.

Let us re-write `Book` with these enhancements. New fields are added to hold the `bookType` and `dateAcquired` and we also declare constants to designate the type of the book.

```
private String title;
private String author;
private String id;
private Member borrowedBy;
private List holds = new LinkedList();
private Calendar dueDate;
private int bookType;
private Calendar dateAcquired;
public static final int BOOK = 1;
public static final int PERIODICAL = 2;
```

Since periodicals and books store different values, we need two constructors. To create a periodical, we use the constructor with two parameters because periodicals have no `author` parameter.

```
public Book(String title, String author, String id) {
  this.title = title;
  this.author = author;
  this.id = id;
  this.type = BOOK;
}
public Book(String title, String id) {
  this.title = title;
  this.id = id;
  this.type = PERIODICAL;
  this.dateAcquired = new GregorianCalendar();
  this.dateAcquired.setTimeInMillis(System.currentTimeMillis());
}
```

The user interface should allow the user to specify what kind of item is being added to the library. This will require a conditional that will not ask for an author if the item being added is a periodical.

```
public void addBooks() {
  Book result;
  do {
```

```
    String title = getToken("Enter  title");
    String bookID = getToken("Enter id");
    if (yesOrNo("Is this a  book?")) {
      String author = getToken("Enter author");
      result = library.addBook(title, author, bookID);
    } else {
      result = library.addPeriodical(title, bookID);
    }
    if (result != null) {
      System.out.println(result);
    } else {
      System.out.println("Book could not be added");
    }
    if (!yesOrNo("Add more books?")) {
      break;
    }
  } while (true);
}
```

The method in the UI invokes a different method of `Library` in each case. Accordingly, `Library` provides two methods, one to add periodicals and one to add books.

```
public Book addBook(String title, String author, String id) {
  Book book = new Book(title, author, id);
  if (catalog.insertBook(book)) {
    return (book);
  }
  return null;
}

//new method added for periodical
public Book addPeriodical(String title,  String id) {
  Book book = new Book(title,  id);
  if (catalog.insertBook(book)) {
    return (book);
  }
  return null;
}
```

Let us examine some other methods in `Book`. The process of issuing a book is different from that of a periodical, and that is reflected in the new `issue` method. The `cutoffDate` is computed and compared against `dateAcquired` to decide if the periodical can be issued.

```
public boolean issue(Member member) {
  borrowedBy = member;
  dueDate = new GregorianCalendar();
  dueDate.setTimeInMillis(System.currentTimeMillis());
  switch (bookType) {
    case PERIODICAL:
      Calendar cutoffDate = new GregorianCalendar();
      cutoffDate.setTimeInMillis(System.currentTimeMillis());
      cutoffDate.add(Calendar.MONTH, -3);
      if (cutoffDate.after(dateAcquired)) {
        dueDate.add(Calendar.WEEK_OF_MONTH, 1);
      } else {
        return false;
      }
```

```
        break;
      default:
        dueDate.add(Calendar.MONTH, 1);
        break;
    }
    return true;
  }
```

The `getAuthor` and `toString` methods also differ because of the absence of a specific author for periodicals.

```
  public String getAuthor() {
    if (bookType == PERIODICAL) {
      return "";
    }
    return author;
  }

  public String toString() {
    if (bookType == BOOK) {
      return "title " + title + " author " + author + " id " + id
          + " borrowed by " + borrowedBy;
    } else {
      return "title " + title + " id " + id + " borrowed by " +
          borrowedBy + " Acquired  on " + dateAcquired.getTime().toString();
    }
  }
```

Likewise, all methods that have different behaviour for books and periodicals will be modified, and this behaviour will be decided based on the value stored in `bookType`. The above enhancement is exactly how a procedural design would be modified. The process varies slightly for each type of data, and this variation is accounted for within the same procedural unit by switching on the kind of data.

9.5.2 Drawbacks of the Above Approach

Before embarking on a critique of such an implementation, it is useful to keep in mind the perspective from which we are approaching the issue. We have two fundamental goals:

- The system should be easy to build and test.
- The system should be adaptable.

It is clear from what we are doing that such an approach involves storing more of the complexity of the system in one class (i.e., `Book`) and its methods. This makes the system difficult to build and test. From the point of view of building, the programmer has to deal with increased complexity of the processes. Examples of this can be seen in the methods like `issue` and `getAuthor`, where the programmer has to be aware of two cases and two possible outcomes that are predicated on the properties of the two kinds of items. Testing adds another dimension of complexity. If we have two

categories of objects, each of whose methods handle five different cases, we have a total of ten outcomes to test when we combine them into a single class. As we pack more requirements into a single class, we increase the probability of human error. Another kind of problem is the combinatorial explosion that happens when program segments use a lot of branch statements. Consider, for instance a method with two switch statements, one following the other, each of which has five possible cases. We now have a total of twenty-five possible computational paths when this method executes. This complexity deters the programmer from putting in the assertions needed to catch all exceptional behaviours. As the science of software reliability progresses, it is becoming increasingly clear to researchers that, at least in critical systems, some form of formal verification will be needed. Our focus should therefore be to produce simpler methods.

A second set of problems arise when we apply the adaptability requirement. Changes to business processes, as we well know, are inevitable. In our system these changes can take two forms: (i) *the procedures for performing library operations may change* and (ii) *we may add new categories of items to the library.* The kind of structure that we have above hurts our ability to modify the code in both these situations. When a procedure changes, say we have some new rules for issuing books, we want to ensure that in re-writing the `issue` method we are not messing up the procedure for the periodicals. This also means that when testing the system after the changes, we have to test the system for both books and periodicals. A similar situation develops when we add new categories of items. The existing methods are changed to accommodate one more case, and once again we need the assurance of system behaviour for the new items added as well as the ones that were already in place.

An important (some would argue the most important) guiding tenet of object-oriented design can be summed up in what is referred to as the *open-closed principle (OCP)*:

A module must be open for extension but closed for implementation.

Extension is the process by which new features are added to existing software, and *implementation* is the process that converts an abstract design into concrete code. What this statement implies is that our classes and modules must be written in such a manner that they can be extended, i.e., new features can be added without re-opening the completed implementation, i.e., without the need for modification of the existing code.

It is obvious that OCP is highly desirable, but a deeper understanding is needed when we apply it. Adding new features to software often requires changes to an existing class. Consider, for example, the feature for charging fines that we added in Chap. 8: changes to `Book` and `Member` were inevitable. In such a situation, we want to ensure that classes not directly involved in this (`Catalog`, for instance) are not affected. Exactly which classes will be affected depends on how the responsibilities were assigned for each step of the procedures involved (viz., Return Book). This assignment is a non-trivial task, but the propagation of the effects of change can be contained by proper encapsulation. Sometimes, changes can mostly be handled by

defining a separate class that incorporates the new features and have only a minimal effect on other classes. This new class, however, needs to be related in some way with the existing classes, *without changing their implementation.* In a world without inheritance, such a feat is impossible to accomplish; inheritance allows us to extend an existing class so that more features can be added to the descendants, even though the ancestor class remains closed. However, *merely applying inheritance will not satisfy OCP;* as we shall see next, a thorough understanding of how the implementation will be reused is essential.

9.6 Improving the Design

In keeping with the above arguments, we once again examine the process that we used to add the new kinds of items. Since this is a chapter on inheritance, it is pretty obvious that we shall employ it in some way, but before we plunge in, a couple of issues deserve our attention.

As we discussed above, some extensions necessarily affect other classes, and so we ask: *What classes must change to accommodate the new kinds of items?* To properly answer this, we must go back to the analysis. The system is allowing users to add new kinds of items to the library, so it is safe to assume that the users are aware of their existence and also expect the system to reciprocate this awareness. All business is transacted through the UI, which is therefore required to know about the new items. It is not immediate from the analysis that any of the other classes need to be aware of this, and we shall therefore postpone this decision to a later point.

In our example, we are introducing new kinds of items to the library and would like to encapsulate the resulting changes to the system using separate classes. In this context, it is useful to ask: *Is there a set of guiding principles that can be employed when we introduce inheritance to incorporate the new items?* Such principles can then be used to guide the design of our inheritance hierarchy. In practice, software designers are often confronted with situations where changes not anticipated during design need to be incorporated. A closely related question, therefore, would be: *Is there a systematic procedure that we can employ when introducing inheritance?* We shall now discuss answers to the above questions as we revisit the process of adding periodicals to the library.

9.6.1 Designing the Hierarchy

As discussed above, we now have two classes, `Book` and `Periodical`. The original design did not have the `Periodical` class and therefore we need to decide how it will be related to `Book`. Two obvious choices present themselves (see Fig. 9.6):

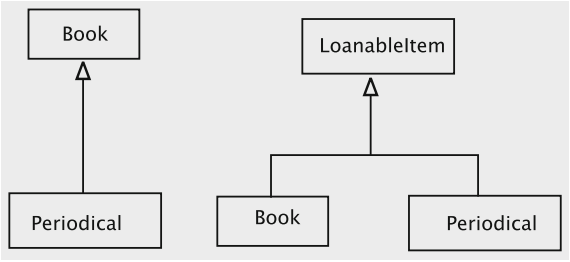

Fig. 9.6 Two hierarchies for library items

- **Option 1**: *Both classes share a common ancestor.* In this hierarchy we have an ancestor class `LoanableItem` with two descendants, `Book` and `Periodical`. The common ancestor is an abstract class that contains the shared attributes. `Catalog` is defined as a collection of `LoanableItem` and the other classes in the system (`Library`, `Member`, etc.) will be redefined to deal with `LoanableItem` (instead of `Book`).
- **Option 2**: *One class inherits from the other.* Since a `Book` class is already present, `Periodical` simply extends `Book` and overrides the necessary attributes. None of the other classes need to be changed.

When designing a new system, the level of complexity appears to be the same in both options. Option 1 requires an additional class, but has the advantage of treating all the items in a uniform manner. On the other hand, when we are adding features to an existing system like the one we have, Option 1 appears to have several disadvantages. The process of creating the new class `LoanableItem` and modifying all the existing classes can be extremely tedious, whereas the 'quick and dirty' approach of Option 2 has the advantage of speed. Over a long period of time, however, such an approach can be very detrimental to a system.

Any system that needs to be up and running for a long time needs *stability*. Simply put, stability of a system is the amount of work that needs to be put in to disturb the existing equilibrium. If left undisturbed, an unstable system may remain stable for a long time, but with very little effort the equilibrium can be disturbed. Extending the notion to a situation where we have several subsystems within a system, consider something like a house, which has several components: a foundation, a wall structure, a roof, and a dish antenna. The roof depends upon the wall structure, which in turn depends on the foundation. The dish antenna is clearly the least stable of all the parts, and therefore it is very undesirable to depend on it (which may be another reason not to become a TV addict!). On the other hand we want the foundation to be very stable, since all the other components are dependent on it. Since the roof depends on the wall structure, we want to ensure that the wall structure is at least as stable as the roof. To summarise, we want to *depend in the direction of stability.* This rule is often referred to as the **stable dependencies principle (SDP)**. In our case study with the Library, we want to ensure that `LoanableItem` is very stable, since the hierarchy depends on it.

In the realm of software, this has some interesting consequences. A 'procedural design' tends to follow a top-down approach: *the structure starts at the top with high-level choices and points down to lower level details.* This suggests that the high-level modules depend on the implementation in the lower level modules. A problem with this approach is that implementations are inherently unstable, and thus we are at odds with the SDP. The object-oriented approach, therefore turns this dependency around so that the design only specifies the abstraction, and the actual implementation ('concretion') satisfies the abstraction. In general, implementations are concrete and therefore inherently unstable. Abstractions do not specify details and thus remain flexible, which makes them more stable when changes have to be incorporated. Thus we can state the following simple thumb rule:

Depend upon abstractions; avoid depending upon concrete implementations.

The design choice that we shall make is therefore to have an abstract class `LoanableItem` on which all the concrete implementations depend. The inheritance structure is translated into code as follows:

```
public abstract class LoanableItem implements Matchable<String> {
  // code common to all types of items that the library lends
}
public class Book extends LoanableItem {
  // code specific to books
}
public class Periodical extends LoanableItem {
  // code specific to periodicals
}
```

In the process of filling in the details for these classes, we will decide how these items are created and added to the collection, what attributes are placed in each class, and how the common code is factored out.

Changes to Other Classes

The main purpose of creating such a hierarchy is to protect all the client classes from changes that occur within the hierarchy. The client classes would now depend on the stable abstraction provided by `LoanableItem` and be completely unaware of the structure below it. All the code in `Library` will now invoke methods through references of type `LoanableItem` and `Catalog` is defined as a collection of `LoanableItem`. It follows then that the abstract class must also implement the `Matchable` interface, as shown in the code above.

9.6.2 Invoking the Constructors

Let us examine the process for adding new items to the library collection. Since the UI knows about the different kinds of items, the method for adding items can query the user about the kind of item, collect the necessary parameters and invoke the method in `Library`. In our earlier implementation we had separate methods

for books and periodicals; this makes the implementation unstable since adding new kinds of items requires adding methods to `Library`. Let us say we can do this with a single method, `addLoanableItem`. The code in the UI would be something like this:

```
private static final int BOOK = 1;      // declaring the constants
private static final int PERIODICAL = 2;
public void addLoanableItems() {
  LoanableItem result;
  do {
    String typeString = getToken("Enter type: "
                                 + BOOK + " for books\n"
                                 + PERIODICAL + " for periodicals\n");
    int type = Integer.parseInt(typeString);
    String title = getToken("Enter title");
    String author = null;
    if (type == BOOK) {
      author = getToken("Enter author");
    }
    String id = getToken("Enter id");
    result = library.addLoanableItem(type, title, author, id);
    if (result != null) {
      System.out.println(result);
    } else {
      System.out.println("Item could not be added");
    }
    if (!yesOrNo("Add more Items?")) {
      break;
    }
  } while (true);
}
```

The method in `Library` must now decide what kind of item to create. This would imply that we have a conditional in `addLoanableItem` that switches on `type`. We no longer have to add methods to `Library` if new kinds of items are desired, but we still need to edit our method to add more clauses to the conditional, which means we cannot reuse the `Library` class directly.

Before proposing solutions, let us take another look at why an inheritance hierarchy is a good idea. We wanted to avoid too much complexity in one class and its methods, so we tried to get rid of conditionals that switched on the type of item by creating a separate subclass for each type, with a common abstract superclass. When invoking the methods on items from this hierarchy, we only refer to the type of the abstract superclass and let dynamic binding take care of the rest. Effectively, we have moved the complexity of the conditional out of the application code and into the interpreter. Dynamic binding works because the system keeps track of the actual concrete subclass of the object even though the reference is stored in a variable declared to have the type of the superclass. When we are invoking constructors, we are yet to create the object; so we cannot rely on dynamic binding to make the choice for us. What this implies is that *the conditionals in the constructor invocation cannot be eliminated.* In other words, conditionals that switch on the input cannot be eliminated using dynamic binding, unlike conditionals that switch on stored values.

(In a sense, creating new objects is like getting new input, and conditionals on the input are essential for any non-trivial program.) The consequence of all this for our design is that the class that chooses the appropriate constructor will undergo a change. Our goal is to protect `Library` from these changes, and some brainstorming gives us three possibilities:

- **Option 1** One possibility is to extend `Library` and redefine `addLoanableItem` whenever new types of items are added.
- **Option 2** A second option is to move the constructor logic into the abstract superclass `LoanableItem`.
- **Option 3** Third, we could develop a new class that takes care of creating the items.

Things to Remember When Creating an Inheritance Hierarchy

Do not rush in too soon. Remember that inheritance is a relationship between well-understood abstractions and the hierarchy usually emerges 'naturally' in our process. This takes time, except in situations where our data has a pre-existing taxonomy. This implies that we have a *clear data abstraction in mind before constructing the hierarchy.*

Allow for future expansion. Keep in mind that we cannot guess how our system might be used; the best way to plan for that is to be generous when allowing for variations. The rules for this are

- *Define methods to be as general as possible at each level of an inheritance hierarchy.* When writing methods, avoid details that are too specifically tailored for the current set of subclasses; the methods should abstract out common functionality so that subclasses can invoke the superclass method to perform some of the task.
- *Be generous in defining data types and storage to avoid difficult changes later on.* For example, you might consider using a variable of type `double` even though your current data may only require a `float` variable.

Make sure the construction is secure. Since we do not know how our system will be used, it is imperative that we do not allow any legal usage to compromise its integrity.

- *Choose the right access modifiers for your attributes* Applying the optimal access levels to members of a class hierarchy makes the hierarchy easier to maintain by allowing you to control how such members will be used. Declare class members with access modifiers that provide the least amount of access feasible.

- *Only expose items that are needed by derived classes* Keeping fields `private` helps descendants and clients by reducing naming conflicts and protects them from using items that may need to be changed at a later stage. Members that are only needed by descendants should be marked as `protected`. This ensures that only the derived classes are dependent on these members, which makes it easier to update these members during development.
- *The functionality provided by the methods of the base class should not depend on features that can be overridden* Make sure that base class methods do not depend on features that can be changed by inheriting classes.

The first choice is easily dismissed; when we extend `Library`, all the classes that depend on it must change. `Library` is a facade and we can therefore expect several other modules to depend on it, which implies that the stability of this module is critical. The other two options share a common underlying principle of designing for change:

> To protect the stability of a module, move the aspects that are likely to change to a different module.

Option 2 suggests that we move this to the class `LoanableItem`. This might seem like a logical assignment of responsibilities, since the constructor invocation is in some way related to the inheritance hierarchy. A closer scrutiny reveals, however, that this would essentially defeat the purpose of introducing inheritance. The abstract superclass is designed to be a *stable abstraction*, that protects the client classes from changes in the hierarchy. It follows that `LoanableItem` should be designed to be unaware of the structure of the hierarchy that lies under it. This leaves us with Option 3, requiring that we create a new module to encapsulate the changes that occur to the logic for invoking constructors. Not surprisingly, this is in fact the standard approach for dealing with this commonly occurring problem.

Implementing a Simple Factory

A **Factory** is typically employed when we want to make a system independent of how its products are created, composed, and represented. In this case, we would like to make the `Library` independent of the process of creating the items. The factory provides a method that can be invoked for creating a new object and thus encapsulates the logic for invocation of constructors. The code for `LoanableItemFactory` is shown below.

```
public class LoanableItemFactory {
  private static final int BOOK = 1;
  private static final int PERIODICAL = 2;
  private static LoanableItemFactory lFactory;
  private LoanableItemFactory() {
  }
```

```
  public static LoanableItemFactory instance() {
    if (lFactory == null) {
      return (lFactory = new LoanableItemFactory());
    } else {
      return lFactory;
    }
  }
  public LoanableItem createLoanableItem(int type, String title,
                                         String author, String id) {
    switch (type) {
      case BOOK:
        return new Book(title, author, id);
      case PERIODICAL:
        return new Periodical(title, id);
      default:
        return null;
    }
  }
}
```

The above code defines `LoanableItemFactory` as a singleton that creates objects of type `LoanableItem`. The method `addLoanableItem` in `Library` is modified as follows:

```
public LoanableItem addLoanableItem(int type, String title,
                                    String author, String id) {
  LoanableItemFactory factory =  LoanableItemFactory.instance();
  LoanableItem item = factory.createLoanableItem(type, title,
                                                 author, id);
  if (item != null) {
    if (catalog.insertLoanableItem(item)) {
      return item;
    }
  }
  return null;
}
```

9.6.3 Distributing the Responsibilities

Next we turn to the task of distributing the attributes and the responsibilities across the hierarchy. This is perhaps the most difficult part of designing the hierarchy and requires considerable experience on part of the software designers. It is useful to keep in mind that we are implementing in a manner that allows the classes in the business logic subsystem to be unaware of the structure of the hierarchy itself. This means that any method that is invoked by code in these classes must be a method of `LoanableItem`. We may also have to store the fields and assign access modifiers based on these considerations. With all this in mind, we start with the following minimum set of attributes for our abstract class.

```
public abstract class LoanableItem implements Serializable,
                                                Matchable<String> {
  private String title;
  private String id;
  protected Member borrowedBy;
  protected Calendar dueDate;
  public boolean matches(String other) {
    return (this.id.equals(id));
  }
  public String getTitle() {
    return title;
  }
  public String getId() {
    return id;
  }
  public Member getBorrower() {
    return borrowedBy;
  }
  public String getDueDate() {
    return (dueDate.getTime().toString());
  }
  // other fields and methods
}
```

The fields `title` and `id` are to be immutable and are therefore defined as `private`. The other fields have been declared `protected` so that they may be accessed by the descendants.

Consider a method like `getAuthor`. Periodicals do not have an author, which suggests that the method can be treated as a specialisation for `Book` and left out of the abstract class. However, it is conceivable that a class may wish to invoke the `getAuthor` on some item without knowing its type. In a situation where we are refactoring to replace `Book` with `LoanableItem`, we may have a client class with a method that has a parameter of type `Book`. We would like this code to behave correctly after refactoring, and so it is desirable that the method be included in `LoanableItem`. On the other hand, it is important that a client class does not assign an author for a periodical. In our case this is easily solved since we do not have a method for setting the `author` field; otherwise, we would have to define a default empty method, `setAuthor`, in the abstract class.

We expect that the methods for processing holds will be similar for all items; these are therefore fully implemented in the abstract class to facilitate reuse. We would like to allow descendants to override them as necessary, which means that the list `holds` has to be a protected attribute. All these additions to `LoanableItem` are shown below.

```
protected List holds = new LinkedList();
protected String author;
public String getAuthor() {
  return "";
}
public Iterator getHolds() {
  return holds.iterator();
}
public void placeHold(Hold hold) {
```

```
    holds.add(hold);
  }
  public void removeHold(String memberId) {
    for (ListIterator iterator = holds.listIterator();
                      iterator.hasNext(); ) {
      Hold hold = (Hold) iterator.next();
      String id = hold.getMember().getId();
      if (id.equals(memberId)) {
        iterator.remove();
      }
    }
  }
  public Hold getNextHold() {
    for (ListIterator iterator = holds.listIterator();
                      iterator.hasNext(); ) {
      Hold hold = (Hold) iterator.next();
      iterator.remove();
      if (hold.isValid()) {
        return hold;
      }
    }
    return null;
  }
  public boolean hasHold(){
      ListIterator iterator = holds.listIterator();
      if (iterator.hasNext()) {
         return true;
      }
      return false;
  }
```

9.6.4 Factoring Responsibilities Across the Hierarchy

The attributes listed above are the ones we selected for the common ancestor. As noted earlier, descendants can override these as needed. Next we examine the responsibilities that are handled in a shared manner between the ancestor and the descendants. Typically, these methods have some common code that can be factored out and placed in the common ancestor and other code specific to each type that is implemented in the descendants.

The first one we examine is the constructor. This is relatively simple in Java, since a constructor for any subclass must first invoke the superclass constructor. The constructor for `LoanableItem` is `protected` and stores the values of the common fields `title` and `id`.

```
protected LoanableItem(String title, String id) {
  this.title = title;
  this.id = id;
}
```

The constructor for `Book` has to set the value for `author`.

```
public Book(String title, String author, String id) {
  super(title, id);
  this.author = author;
}
```

The constructor for `Periodical` needs to store the date of acquisition, and a private field is defined for that. The date itself can be generated using the system clock.

```
private Calendar dateAcquired;
public Periodical(String title, String id) {
  super(title, id);
  this.dateAcquired = new GregorianCalendar();
  dateAcquired.setTimeInMillis(System.currentTimeMillis());
}
```

Next we consider methods like `toString`. Part of this responsibility can be handled by the superclass methods, and the subclass methods simply append the additional information. In the code shown below, the method `LoanableItem` concatenates the fields `title, id`, and `borrowedBy`.

```
public String toString() {
  return " title " + title +  " id " + id + " borrowed by " + borrowedBy;
}
```

Both subclasses append their types to the string returned by the superclass method. In addition, `Book` appends the `author` field, and `Periodical` appends `dateAcquired`.

```
public String toString() {
  return "Book " +  " author " + author + super.toString();
}

public String toString() {
  return "Periodical " + super.toString() + "\n  Acquired On "
         + dateAcquired.getTime().toString();
}
```

Some methods can have more involved cooperation across the hierarchy. Let us examine the method for issuing an item, which involves checking issuability, assigning the item to a `Member` object, and generating and storing the due date. Both checking of issuability and generation of due date involve rules specific to the items. The only common activity is that of assigning the item to a `Member`, which can be factored out. In Java, the process of due date generation can be simplified if we assign the current date as due date (Step 1) and then add the period of loan (Step 2). Step 1 can also be factored out, giving us the following `issue` method for `LoanableItem`.

```
public boolean issue(Member member){
  if (borrowedBy != null) {
    return false;
  }
  dueDate = new GregorianCalendar();
  dueDate.setTimeInMillis(System.currentTimeMillis());
  borrowedBy = member;
  return true;
}
```

`Book` does not have any additional rules for issuability, so it simply invokes the superclass method and adds the loan period, i.e., one month, if the superclass method returns `true`.

```
public boolean issue(Member member) {
  if (super.issue(member)) {
    dueDate.add(Calendar.MONTH, 1); //add loan period
    return true;
  } else {
    return false;
  }
}
```

The method in `Periodical` must first ensure that the periodical is at least three months old before it invokes the superclass method. The loan period of one week is added if everything checks out.

```
public boolean issue(Member member) {
  Calendar cutoffDate = new GregorianCalendar();
  cutoffDate.setTimeInMillis(System.currentTimeMillis());
  cutoffDate.add(Calendar.MONTH, -3);
  if (cutoffDate.after(dateAcquired)) {
    if (super.issue(member)){
      dueDate.add(Calendar.WEEK_OF_MONTH, 1);
      return true;
    }
  }
  return false;
}
```

9.7 Consequences of Introducing Inheritance

From our discussion so far, it should be fairly clear that inheritance provides a lot of benefits to the software development process. In an earlier section, we have discussed some caveats to be followed. In addition to these, inheritance introduces some other problems because of our attempt to ensure that changes that occur within the hierarchy do not affect classes outside the hierarchy. One example of this that we have encountered is the problem of invoking constructors, which we solved with the use of a *factory*. The other solutions follow a similar pattern, in that they create some external structure that in some way parallels the structure in the hierarchy. A couple of such situations are dealt with here.

Introducing an Inheritance Hierarchy Through Refactoring

We sometimes encounter situations in legacy systems where an inheritance hierarchy has to be introduced in order to clean up the existing code. One has to be especially careful when attempting such an exercise since the dependencies involved can be quite complex. A well-designed, systematic procedure can significantly reduce the chances of errors. The following steps can serve as a guide.

Replace Conditional with Polymorphism

If you have a conditional that chooses different behaviour depending on some feature of the object, move each leg of the conditional to an overriding method in a (possibly newly defined) subclass and make the original method abstract.

The steps involved in applying this rule are as follows:

- Identify a conditional statement in a method that changes its behaviour based on the value stored in a particular field. In a large class, it is quite likely that there will be several methods where variation in behaviour is obtained by switching on the same field.
- If the conditional statement is part of a larger method, the conditional may have to be extracted using the EXTRACT METHOD rule (Chap. 8). If such extraction is not easily done, the class may have to be re-examined more closely.
- Define an inheritance hierarchy where the subclasses reflect the variations in the field on which we are switching.
- Create a subclass method that overrides the conditional statement method. Copy one leg of the conditional into each of the subclass methods. and adjust the code so that it fits.
- Remove the conditional from the superclass method and make it abstract. If appropriate, remove the field on which the switching was done.

Note that once the switching is removed, we may no longer need the field to track the variation in the type of the object.

9.7.1 Exception Handling

The following is the standard rule for throwing exceptions when we employ inheritance:

> A subclass method that overrides a method of a superclass may not throw an exception that is not thrown by the superclass method.

This may seem puzzling at first glance—after all a subclass can add new features—but a closer look reveals that this rule is really a consequence of the LSP (see exercises). Of course, such a violation could never be achieved in Java since it can be detected statically at compile time.

There are several situations, however, where we would like to create a subclass and obtain more specific information in the case of exceptional behaviour. As an example, consider a class that processes a stream of data.

```
public class StreamProcessor {
  // fields and constructors not shown
  public void processStream() throws IOException {
  // code not shown
  }
  // other methods not shown
}
```

The method `processStream()` opens a stream and does some elementary processing of the data and creates an output file. In the course of writing the data, exceptions may arise, which cause the method to throw `IOException`. A subclass is expected to override this method.

Now consider a subclass of the above, `FileProcessor`, which uses data from a file. Since a file is a specific kind of data stream, this would be a valid use of inheritance. An exceptional situation arises when the file does not exist, and it is advantageous for users of the subclass to clearly know the reason for the exception.

The subclass with the overriding method is given below.

```
public class FileProcessor {
  String fileName;
  // other fields and constructors not shown
  public void process() throws NoSuchFileException, IOException {
    BufferedReader reader = new BufferedReader(new FileReader(fileName));
    // process the file
  }
  // other methods not shown
}
```

For reasons described earlier, the above code will not compile. The way to get around this is to create an **exception hierarchy**.

```
class NoSuchFileException extends IOException {
  // fields and methods as needed
}
```

Now our client class can be written to deal with the `NoSuchFileException` as needed and can also ignore the classification by writing a handler for just the `IOException`.

9.7.2 Adding New Functionality to a Hierarchy

Replacing a class with a hierarchy can pose additional problems when new functionality has to be added. Consider a situation where a client (end user) wants a list of

books in the library printed in a certain format. The client chooses the format, and therefore it may have considerable variation. Since the system output is not fully specified at the beginning, this would have to be handled differently from the other features like adding members or checking out books. Such a feature is typically provided by asking the user to encapsulate the format as an object or a process, which can then be invoked from within `Library`. In a situation where we have only one `Book` class, `Library` may accomplish this with a method like the one shown below.

```
public void bookReport(BookFormat format) {
  for (ListIterator iterator = catalog.listIterator(); iterator.hasNext();){
    Book book = (Book) iterator.next();
    format.print(book);
  }
}
```

Essentially, the `print` method in `BookFormat` specifies a printing strategy. `BookFormat` itself is defined as an interface.

```
public interface BookFormat {
  public void print(Book book);
}
```

Each client that wishes to use this feature must first define a class that implements `BookFormat`. The class can be configured with several other attributes that decide the output stream and the `print` method can print the details of the book in the required format. The method in `Library` invokes the `print` method once for each `book`.

A solution like the one above suffers from two drawbacks:

1. The structure is tailored too specifically for one kind of operation. In our case, this is for printing the books. A client may instead want an operation that checks which books are issued on a particular day. The method `bookReport` suggests that it can only be used for generating reports.
2. The structure is tailored only for one class and cannot accommodate a hierarchy like `LoanableItem`. In case of a hierarchy, we would like to specify different operations for each subclass. In the solution above, the type of parameter to `print` is fixed as `Book`. We could change that to `LoanableItem`, but the method to be called will not be determined dynamically, which would cause the system to treat books and periodicals identically and is therefore not satisfactory.

The standard solution for dealing with this is to use the **visitor pattern.** The intent of this pattern is *to represent an operation to be performed on the elements of an object structure, and is employed to define a new operation without changing the classes of the elements on which it operates.*

In the situation above, we have a new `print` operation to be performed on the items in the `Catalog` object, which are all of type `LoanableItem`; we would like `Library` to provide a functionality that allows the user to apply the `print` operation to all the items in the `catalog` without exposing `Library` to the details of the hierarchy. Note that if we had separate catalogs for each kind of item, this

discussion would be moot. However, if we defined `Library` to have separate collections for each kind of item, we have a design where the facade is not protected from changes to the specifications. This would result in the kind of instability that we are trying to avoid.

The solution for this once again follows the principle of *encapsulating change.* We create a separate structure that accommodates the changes and shields the classes that must be kept stable. This is similar to what we did with `LoanableItemFactory`, but we have an additional complication here. Our solution will require `Library` to provide some method like `bookReport`, which must invoke the correct `print` method without knowing what kind of item we are dealing with. In the method presented earlier, we knew that all items were of type `Book` and we could safely perform the cast. With the hierarchy, all we know is that the object returned by the iterator is of type `LoanableItem`. We therefore need some additional machinery to ensure that the correct method is invoked. This differs from the situation where we were invoking constructors, since in that case the kind of item to be created was explicitly specified in `type`.

The solution we develop with the visitor pattern has three components:

- *A* **visitor interface** *that encapsulates the variability in the object structure.* In our example the variability in the kinds of items stored in `Catalog` is exactly the variability in the `LoanableItem` hierarchy. Our interface, aptly named `LoanableItemVisitor`, therefore parallels the hierarchy by defining a method for each for kind of item.
- *An* **accept** *method in each visitee.* In our example, the visitees are the concrete classes in the hierarchy. Each of these must have a method with the signature `public void accept(LoanableItemVisitor)`
- *A concrete client class for the required functionality that implements the visitor interface.* In our example with `print`, `BookFormat` would be replaced with this concrete client that implements `LoanableItemVisitor`.

The code for the interface is shown below. Note that we provide a method with a parameter of type `LoanableItem` even though no such concrete item can be created. This is to ensure that we have a 'catch-all' that will take care of extreme situations where we have added new items to the hierarchy but not updated the visitor interface.

```
public interface LoanableItemVisitor {
  public void visit(LoanableItem loanableItem);
  public void visit(Book book);
  public void visit(Periodical periodical);
}
```

Every concrete class has an `accept` method as shown below.

```
public void accept(LoanableItemVisitor visitor) {
  visitor.visit(this);
}
```

The concrete class ItemFormat takes the place of BookFormat and looks something like this.

```
class ItemFormat implements LoanableItemVisitor {
  public void visit (Book book) {
    // code to print a book
  }
  public void visit (Periodical periodical) {
    // code to print a periodical
  }
  public void visit (LoanableItem item){
    System.out.println("Unspecified item");
  }
}
```

The UI provides some method that will allow the user to incorporate this functionality. In our case, we want to provide the functionality to print items, and the client is required to specify the details in ItemFormat, as shown above. The method in UI simply invokes the method in Library, as shown below:

```
public void printItems() {
  library.processItems(new ItemFormat());
}
```

The advantage of having an interface is that we can write a single method in Library which takes care of all such visitors that perform some operation on the items in the catalog. As shown below, we can define any number of operations to be done on the items in the catalog by invoking the processItems method.

```
public void processItems(LoanableItemVisitor visitor) {
  for (Iterator iterator = catalog.getLoanableItems();
                           iterator.hasNext();) {
    LoanableItem item  = (LoanableItem) iterator.next();
    item.accept(visitor);
  }
}
```

The method processItems invokes the accept on each item in the Catalog object, passing as a parameter the concrete object that implements LoanableItemVisitor, viz., the instance of ItemFormat that was created in user interface. As we have seen above, the accept method invokes the visit method on the visitor passing itself as a parameter. The most appropriate signature is matched to decide which method is to be invoked, i.e., when we invoke visit from Book, the system will invoke the method visit(Book) in the ItemFormat object. The two step process (called 'double-dispatch') thus obviates the need to know the class of each catalog item in processItems. These details are illustrated in Fig. 9.7.

To ensure that this approach provides the correct response for each item in the hierarchy, note that the LoanableItemVisitor interface must be updated whenever a new kind of item is added. However, we would like to ensure that the system does not crash in the event that this update is overlooked or if we choose to retain the old interface. Consider for example, that another kind of item, say DVD, has been added and we have a new class VideoItem that extends LoanableItem; we

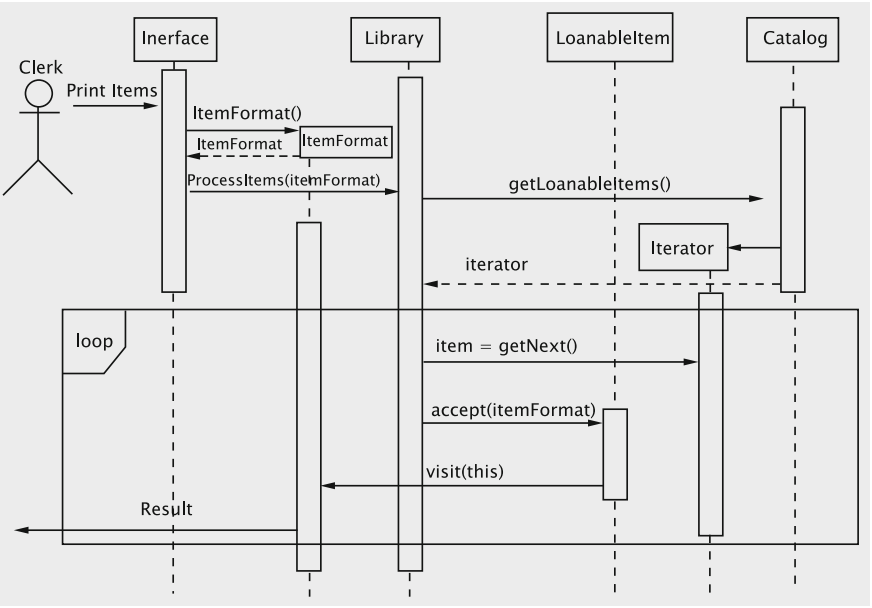

Fig. 9.7 Control flow for the visitor pattern

could be in a situation where the `catalog` contains instances of `VideoItem`, but `LoanableItemVisitor` has not been updated. In this case, the method `visit(LoanableItem)` gets invoked. This may result in a less than ideal system response, since a `VideoItem` object would be treated as a `LoanableItem`, but the system will continue to run and other harmful consequences (such as loss of data in case of a crash) will be avoided. If the `visit` method with the `LoanableItem` parameter did not exist, we would get a run-time error.

9.8 Multiple Inheritance

So far, we have seen situations where a class inherits from only one other class. The term **Multiple Inheritance** is used to describe the ability of a class to subclass multiple classes. Let us consider two examples.

1. A mobile home serves as a home, but could also be driven from location to location. Therefore, it has properties of both a home (it will have bedrooms, kitchen, etc.) and a car (the unit has an engine and can be driven like a car). If we have classes `Car` and `Home`, then the class `MobileHome` can be constructed by utilising the implementations of both of these existing classes.

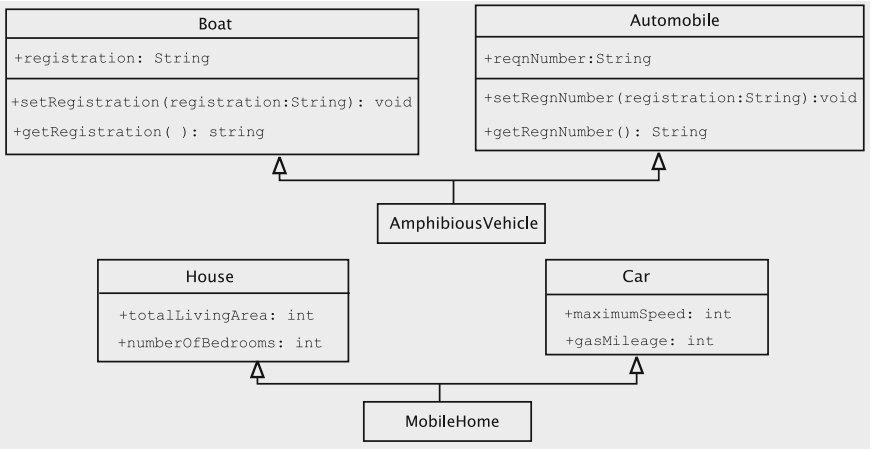

Fig. 9.8 Examples of multiple inheritance

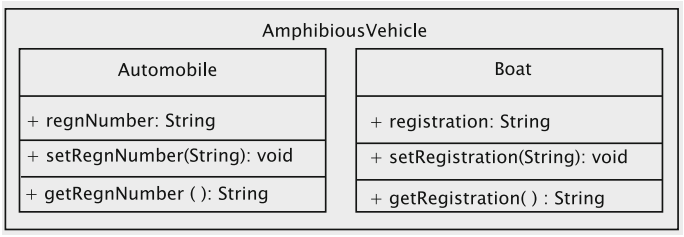

Fig. 9.9 Conceptual view of an AmphibiousVehicle

2. An amphibious vehicle can run on both land and water. It will, therefore, have properties of both an automobile and a boat. With classes `Automobile` and `Boat` available, we can create a class that subclasses both.

Figure 9.8, illustrates these examples using UML diagrams.

In all these examples, the descendant inherits properties from both the ancestors. A `Mobilehome` inherits features such as `maximumSpeed` and `gasMileage` from `Car` and features like `totalLivingArea` and `numberOfBedrooms` from `House`. Programming languages typically allow for multiple inheritance by allowing a class to extend more than one class. This ability does pose some new challenges as we shall see later in this section.

Since the class `AmphibiousVehicle` extends `Automobile` and `Boat`, an instance of an `AmphibiousVehicle` can be viewed as containing both an instance of `Boat` and an instance of `Automobile` (Fig. 9.9).
Consider the following code[2]:

[2]Since Java does not support multiple inheritance in this form, this code does not conform to Java syntax.

```
public class Boat {
   private String registration;
   public Boat(String registration) {
     this.registration = registration;
   }
   public void setRegistration(String registration) {
     this.registration = registration;
   }
   public String getRegistration() {
     return registration;
   }
}

public class Automobile {
   private String regnNumber;
   public Boat(String registration) {
      this.regnNumber = registration;
   }
   public void setRegnNumber(String registration) {
     this.regnNumber = registration;
   }
   public String getRegnNumber() {
     return regnNumber;
   }
}

public class AmphibiousVehicle extends Boat, Automobile {
   public AmphibiousVehicle(String registration) {
      Boat(registration);
      Automobile(registration);
   }
}
```

Both `Automobile` and `Boat` have a field to store the registration number and `AmphibiousVehicle` inherits the field from both these classes, as shown in Fig. 9.9. Since these different names essentially capture the same attribute, we have some ambiguity. Consider the following situation:

```
Automobile automobile;
Boat boat;
AmphibiousVehicle vehicle;
```

An `AmphibiousVehicle` object can be stored in an `Automobile` reference or a `Boat` reference.

```
vehicle =  new AmphibiousVehicle("001");
automobile = vehicle;
boat = vehicle;
```

There appear to be several ways in which the registration number of this object can be accessed: `vehicle.registration`, `vehicle.regnNumber`, `automobile.regnNumber` or `boat.registration`. This multiplicity of field names can make the code hard to read. In addition, we now have a possible polymorphic assignment of the kind:

```
boat = (AmphibiousVehicle) automobile;
```

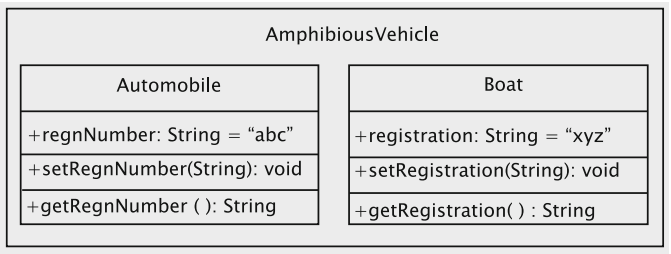

Fig. 9.10 AmphibiousVehicle showing assignments

This would not be possible under single inheritance, since `Car` and `Boat` belong to different hierarchies.

In creating such a hierarchy, it is therefore important to keep the semantics of the attributes in mind. The public methods present a more serious problem. Consider the code

```
vehicle.setRegistration("xyz");
// some code
vehicle.setRegnNumber("abc");
```

This results in the situation shown in Fig. 9.10.

The entities `vehicle.registration` and `vehicle.regnNumber` will now contain different values, causing inconsistencies.

9.8.1 Mechanisms for Resolving Conflicts

Any language that provides a mechanism for multiple inheritance must also provide means for resolving these conflicts. For the example above, let us assume that the designer chooses to store the registration of the `AmphibiousVehicle` in the `Automobile` object, but would like to use the method names `setRegistration` and `getRegistration`. The methods `setRegnNumber` and `getRegnNumber` inherited from `automobile` must now be 'un-inherited' so that there is no ambiguity. One option is to declare the unwanted methods and fields as `abstract` in the descendant class. The class `AmphibiousVehicle` would now be something like this[3]:

```
public class AmphibiousVehicle extends Boat, Automobile {
  public AmphibiousVehicle(String string) {
    Automobile(string);
  }
  public abstract setRegnNumber(String string);
```

[3]We would like to remind the reader that the Java-like code we have given below is not valid in the Java language.

```
  public abstract getRegnNumber(String string);
  public void setRegistration(String string) {
    Automobile.setRegnNumber(string);
  }
  public String getRegistration() {
    return Automobile.getRegnNumber();
  }
}
```

Note that we are not explicitly invoking the constructor for `Boat`. The default constructor is invoked and consequently, there is no copy of the registration being stored in the `Boat` object. The methods `setRegistration` and `getRegistration` have been suitably redefined to access the fields of the `Automobile` object.

9.8.2 Repeated Inheritance

Since multiple inheritance could generate a hierarchy that is not a simple tree, we can end up with a situation where a descendant can be reached from an ancestor by following two different paths, giving rise to what is referred to as the *'diamond of repeated inheritance'* (Fig. 9.11). Such a structure results when we have a class `Vehicle` which serves as an ancestor for both `Automobile` and `Boat`.

```
public class Vehicle {
  private String registration;
  public Vehicle(String string) {
    registration = string;
  }
  public void setRegistration(String string) {
    registration = string;
  }
  public string getRegistration() {
    return registration;
  }
}

public class Automobile extends Vehicle {
  private int maximumSpeed;
  public Automobile(String string, int speed) {
```

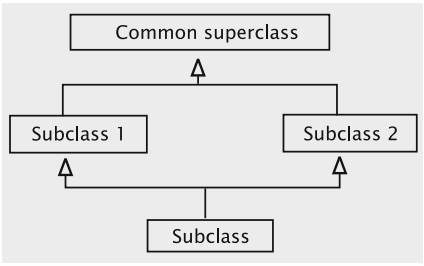

Fig. 9.11 Diamond of repeated inheritance

```
    Vehicle(string);
    maximumSpeed = speed;
  }
}

public class Boat extends Vehicle {
  private int maximumKnots;
  public Boat(String string, int knots) {
    Vehicle(string);
    maximumKnots = knots;
  }
}

class AmphibiousVehicle extends Boat, Automobile {
  public AmphibiousVehicle(String string, int speed, int knots) {
    Boat(string, knots);
    Automobile(string, speed);
  }
}
```

This is a more serious problem than what we faced when a field was duplicated. The constructor for `AmphibiousVehicle` must invoke constructors `Automobile` and `Boat`, both of which invoke the constructor for `Vehicle`. As a result, we have a situation where there are two copies of `registration`. Note that the registration information is actually being stored in a private field of `Vehicle` and the only way to access it is through the methods of `Vehicle`. If the accessor or modifier of `AmphibiousVehicle` is invoked, it is not clear which copy is being modified. The author of `AmphibiousVehicle` has to be aware of these issues and should override these methods to ensure that both copies are updated.

When we are dealing with large hierarchies, it is not always possible for the author of the subclass (such as `AmphibiousVehicle`) to detect the problem. In such a situation, the programming language must ensure that the repeated ancestor is not created twice. C++, for instance, uses the following solution: *The inheritance relationship between* `Vehicle` *and its immediate descendants should be declared as* **virtual** *(or 'shareable').* This means that the space occupied by the two `Vehicle` objects must be shared and as a result the compiler flags an error when the constructor is invoked twice. The constructor in `AmphibiousVehicle` is then required to explicitly invoke all three constructors. In our 'java-like' syntax, the constructor for `AmphibiousVehicle` is as follows:

```
public AmphibiousVehicle(String string, int speed, int knots) {
  Vehicle(string);
  Boat(string, knots);
  Automobile(string, speed);
}
```

When the `Vehicle` constructor is invoked, a `Vehicle` object is created, and the same space is shared by the `Automobile` and `Boat` objects. Since only one copy of the `Vehicle` attributes is maintained, we have no inconsistency.

When the calls to constructors propagate up the hierarchy, calls to 'virtual ancestors' are ignored. In our example, when the constructor for `Boat` (or `Automobile`) is invoked, the call to the `Vehicle` constructor is ignored because `Boat` (or

Automobile) is defined a *virtual* descendant of Vehicle and the Vehicle object has already been created. Since the inheritance hierarchy is statically determined, such an approach is feasible.

The above code suffers from two problems:

1. It requires that AmphibiousVehicle be cognizant of the entire hierarchy. If that class's constructor misses any one of the superclass constructor calls and the corresponding class does not have a default constructor, the compiler flags an error.
   On the other hand, if any default constructors exist, the code may end up being buggy. For instance, the code

```
public AmphibiousVehicle(String string, int speed, int knots) {
  Boat(string, knots);
  Automobile(string, speed);
}
```

   would not generate any compiler errors if Vehicle had a default constructor. In this situation, the registration field in Vehicle will be initialised to the default value instead of the specified parameter, string.
2. This approach is not general enough since it cannot work in situations where there is an existing hierarchy and the inheritances are not virtual.

In the example with the AmphibiousVehicle we saw that repeated inheritance can cause a constructor to be invoked twice. We were able to resolve the consistency problem by redefining the methods and attributes. However in situations where invoking the constructor has a more 'visible' effect, this could pose a more serious problem. Consider the following example where Window has two descendants—MenuWindow, which is a window with a menu, and BorderWindow, which is a window with a border. The fourth class, MenuAndBorderWindow, completes the diamond by inheriting from both MenuWindow and BorderWindow.

Consider a situation where we have a method display for displaying the Window. The display method in BorderWindow first invokes the ancestor's display method (which displays the window) and then invokes its own method that displays the border. Likewise, the display method in MenuWindow first invokes the ancestor's display method (which displays the window) and then invokes its own method that displays the menu. How should we deal with the display method of MenuAndBorderWindow? If we invoke the display methods of both the immediate superclasses, we end up in a situation where the window will be displayed twice.

```
public class MenuAndBorderWindow extends MenuWindow, BorderWindow {
// fields and other methods not shown
  public void display() {
    MenuWindow.display();
    BorderWindow.display();
  }
}

public class MenuWindow extends Window {
```

```
// fields and other methods not shown
  public void display() {
    Window.display();
    // code for displaying menu goes here
  }
}

public class BorderWindow extends Window {
// fields and other methods not shown
  public void display() {
    Window.display();
    // code for displaying border goes here
  }
}
```

In general, there is no simple solution to such problems. The software designer has to be aware of these issues and exercise the necessary caution. The above problem, for instance, could be resolved by having `MenuAndBorderWindow` inherit from all three superclasses. Its `display` would then first invoke the method in `Window` and then invoke methods from `MenuWindow` and `BorderWindow` that display the menu and the border respectively. (We are assuming here that we can invoke the methods for displaying these; this would be another example of a situation where the protected access mode would come in handy.)

```
public class MenuAndBorderWindow extends MenuWindow, BorderWindow, Window {
  // fields and other methods not shown
  public void display() {
    Window.display();
    BorderWindow.showBorder();
    MenuWindow.showMenu();
  }
}

public class MenuWindow extends Window {
  // fields and other methods not shown
  public void display() {
    Window.display();
    this.showMenu();
  }
  protected void showMenu() {
    // code for displaying the menu
  }
}

public class BorderWindow extends Window {
  // fields and other methods not shown
  public void display() {
    Window.display();
    this.showBorder();
  }
  protected void showBorder() {
    // code for displaying the border
  }
}
```

9.8.3 Multiple Inheritance in Java

Java does not support real multiple inheritance in the sense that a class can inherit an implementation from one other class only. To deal with the situation where a class has to inherit attributes from more than one class, the only option is to create the class as a subclass of one of the classes and implement the rest. For example, assume that we would like to create a class `C` that ideally extends classes `C1` and `C2` which implement interfaces `I1` and `I2` respectively. Then, the code for `C` would be

```
public class C extends C1 implements I1, I2 {
  // code
}
```

Since an interface can be viewed as a type, a class that extends another class and implements an interface can be viewed as a sub-type of both the ancestor class and the interface. This gives the flavour of multiple inheritance to the language. In the above example, objects of type `C` are also of type `I2`.

9.9 Discussion and Further Reading

This chapter has explored the uses of inheritance, how to introduce it, and what are some of the consequences of inheritance. Software systems are usually complex, and it is always a difficult task to characterise them completely. Inheritance poses an added challenge since it allows an existing system to change. Attempts have been made by researchers to define taxonomies to understand both the nature of inheritance and to identify changes in the object-oriented systems [1, 2].

The question of when and how to introduce inheritance can be a tricky question. Beginners often tend to follow the lead of textbook examples and introduce inheritance upon finding common fields between classes. This is not only wasteful, but can also lead to problems. Inheritance is a relationship between well-understood abstractions, and should be introduced only when a need for it can be clearly justified based on the principles of object-oriented design. Exceptions are made to this rule only when the classes are being used to model categories of objects in the natural world and there already exists a well-defined taxonomy of these categories.

The *open-closed principle* is perhaps the most applicable design rule to justify introducing inheritance. Note that there is no inherent contradiction in this principle, since the words ‘open’ and ‘closed’ apply to different objectives. The modules should be open for further extension, and closed for clients that are depending on it. The client modules are thus assured that any changes introduced into the system later will not necessitate any modifications. It is important to remember here that a class should not be closed too soon. A class must represent a *coherent data abstraction,* i.e., provide a coherent set of services to potential clients. Closing a class too soon and then frequently extending it because the data abstraction was incompletely defined should be construed as an abuse of inheritance.

Our case-study with the library system can also be viewed as an exercise in designing a schema for an object-oriented database. Use of inheritance can be tricky when we create a database. We need to ensure, for instance, that inheritance conflicts (which can arise with multiple inheritance and overriding) are avoided. In addition to these correctness issues, compactness of the schema is also a consideration [3, 4]. Object-oriented databases are commonly mapped to relational databases for efficient storage, and mapping objects to relations can be tricky when inheritance is involved [5].

The *Liskov substitution principle* is a compact reminder of the most basic invariant an inheritance relationship must satisfy. Barbara Liskov's original article appeared as a joint paper with J. Wing in 1994, but the principle was re-formulated more succinctly [6] as follows:

> Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S, where S is a subtype of T.

Thus, Liskov and Wing's notion of 'subtype' is based on the notion of substitutability; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness).

In most situations where the LSP is violated, we find that an ad-hoc relationship has been introduced to use existing code. In our example in this chapter, the fact the relationship between `SolidRectangle` and `Pixel` was an ad-hoc one and not a well-designed inheritance structure is underscored by the fact that the only way to fix the problem is to modify `clientMethod`. This means that the original modules (i.e., `SolidRectangle`) that were considered closed have been in some way re-opened by introducing inheritance. Thus we have indirectly violated OCP.

As an example of another, perhaps more surprising, example of LSP violation, the reader should look at Exercise 6.

To fix the bugs created by LSP violations, the only option we have is to check every piece of client code and put in conditionals that employ run-time type identification or use exception handlers. This is clearly not feasible. LSP violations are effective reminders of two of the design principles introduced in this chapter, viz., *favour composition over inheritance* and *depend upon abstractions.* If the class `Pixel` had been implemented by *adapting* `SolidRectangle`, we would not face any problem with client methods. Our implementation also violates the dependency rule, since `SolidRectangle` is a concrete implementation.

9.9.1 Design Patterns that Facilitate Inheritance

In this chapter we have introduced two patterns: *factory* and *visitor*. Factories are *creational patterns* that are generally employed for creating objects without specifying the exact class of object. There are some variations on this, and what we have discussed here is perhaps the simplest form in which it can be used. The **factory method**

pattern is employed in situations where we have two independent hierarchies, and the concrete classes in the first hierarchy create objects belonging to classes in the second hierarchy. However, the exact kind of object of the second hierarchy to be created is determined using the input provided at run-time. In such a situation, the logic for invocation of the constructor is encapsulated in a separate method in the abstract superclass of the first hierarchy. The **abstract factory** pattern is used in situations where we have several parallel concrete hierarchies, and the system has to be configurable with any one of them. For instance, we could have a hierarchy of *paint objects* for generating paint objects of several colours. The same colours are used for several situations, viz., interior, exterior, furniture, etc. However, the paint object that must be used has a different implementation for each situation. In such a case, we would have an `AbstractcolourFactory` that would have descendants like `InteriorcolourFactory`, `ExteriorcolourFactory`, etc. The `AbstractcolourFactory` would specify the methods (e.g., `makeRedPaint()`) for creating abstract paint objects, and the concrete descendants would implement these methods to provide concrete paint objects for the given situation (the `makeRedPaint` method in `ExteriorcolourFactory` would create `ExteriorRedPaint`). The client class could then be adapted for any painting situation (interior, furniture, etc.) by configuring it with the appropriate concrete factory. The methods in the chosen factory would then be used to generate the concrete paint objects needed for the situation.

The visitor pattern can be used for any general collection of objects, not necessarily constituting a hierarchy. All that we need is that there should be a matching signature for the class of every object in the collection. The `Object` class can be used as a 'catch-all' to prevent run-time errors. Using this pattern increases the cost of execution due to the additional method call. That can be prevented if the language provided the feature of 'double dispatch,' i.e., the parameter type and the concrete class are both matched when invoking dynamic binding. This feature was provided in smalltalk, but has not found favour with other language designers due to the high cost of method calls.

9.9.2 Performance of Object-Oriented Systems

An issue that is often raised with object-oriented systems is that of poor run-time performance. There are several reasons for this and solutions have been proposed; it would be beyond the scope of this text to go into these in any detail. In the context of inheritance, however, we shall look into one of these issues: **the overhead caused by dynamic binding**.

When we subclass the conditional behaviour by introducing an inheritance hierarchy, we rely on dynamic binding to ensure that the correct version of the method is called. This decision has to be made at run-time, as opposed to method calls whose target can be statically determined during compilation. Normally, with every variable in the system, some sort of type information will be stored. In an object-oriented

system, due to polymorphism, the actual type of the object whose reference we store in the given variable can change dynamically. The standard way to implement dynamic binding is to have a table of method addresses for each class. Whenever a method is invoked on a variable, the type of object the variable refers to is looked up. The actual type of the reference is used to select the appropriate table, and the method name is used to index the table to determine the address of the method to be invoked. Thus, dynamic binding introduces some additional overhead for every method invocation.

Projects

1. Implement the classes `Account`, `CheckingAccount` and `Savings Account` we outlined in Sect. 9.2.2.
2. Consider the classes `DataStream` and `ReReadableDataStream` in Sect. 9.2.3. Show how to implement the two classes. Remember that `ReReadableDataStream` must work regardless of the source from which `DataStream` reads.
3. Implement the `Cloneable` interface for the `Book`, `Member`, and `Hold` classes without having to set any fields to the `null` value.

9.10 Exercises

1. Extend the `LoanableItem` hierarchy to create new classes for CDs, DVDs and books on tape. CDs and DVDs have several common characteristics. Would it be appropriate for these two classes to inherit from a common superclass? Why?
2. As mentioned in the chapter, a subclass method that overrides a method of a superclass may throw subclasses of exceptions that are thrown by the superclass method, but it cannot throw exceptions that are not thrown by the overridden method. Why?
3. A university registration system has a class `Student` that tracks student information. When a student's GPA falls below a certain level, he/she is placed on academic probation. Would you model this by creating a subclass `WeakStudent` that extends `Student`?
4. Keeping in mind that a circle is a special kind of ellipse in which the two foci coincide, create a scenario in which an LSP violation can occur when a class `Ellipse` is extended to define `Circle`.
5. In Chap. 7, we defined some methods of `Library` to return error codes, whereas others return references to objects. In a more sophisticated system, it is often necessary to return an object that contains both the result code and other information that the UI can display upon request. Define a class `Result` and a hierarchy of subclasses that will take care of this for all the methods in `Library`.
6. Consider the following classes and explain whether LSP is violated in the `main` method.

```
class Rectangle {
  private int width;
  private int height;
  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }
  public void setHeight(int height) {
    this.height = height;
  }
  public void setWidth(int width) {
    this.width = width;
  }
  public int getHeight() {
    return height;
  }
  public int getWidth() {
    return width;
  }
}
public class Square extends Rectangle {
  public Square(int side) {
    super(side, side);
  }
  public void setWidth(int side) {
    super.setWidth(side);
    super.setHeight(side);
  }
  public void setHeight(int side) {
    super.setWidth(side);
    super.setHeight(side);
  }
  public static void main(String[] s) {
    Rectangle r = new Rectangle(10, 10);
    r.setWidth(5);
    r.setHeight(6);
    if (r.getWidth() * r.getHeight() != 30) {
       System.out.println("Error");
    }
  }
}
```

7. (Case-studies) Examine the projects presented at the end of Chap. 6 and identify possible situations where we could get variability in the behaviour of an object. Which variabilities will you model using inheritance? Defend your choices based on object-oriented design principles.

References

1. B.M.P. Clarke, P. Gibson, Using a taxonomy tool to identify changes in object-oriented software, in *7th European Conference on Software Maintenance and Reengineering*, Benevento, Italy, 26–28 March 2003
2. X. Girod, Conception par objects—MECANO: une methode et un environnement de construction d'application par objects. Ph.D. thesis, University of Joseph Fourier Grenoble I, Grenoble, June 1991

3. A. Formica, H.D. Gröger, M. Missikoff, Object-oriented database schema analysis and inheritance processing: a graph-theoretic approach. Data Knowl. Eng. **24**(2), 157–181 (1997)
4. A. Formica, H.D. Gröger, M. Missikoff, An efficient method for checking objectoriented database schema correctness. ACM Trans. Database Syst. **23**(3), 334–369 (1998)
5. S.W. Ambler, *Building Object Applications that Work* (Cambridge University Press, Cambridge, 1998)
6. B.H. Liskov, J.M. Wing, Behavioural subtyping using invariants and constraints, in *Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches*, ed. by H. Brown, J. Derick (Cambridge University press, Cambridge, 2001), pp. 254–280