# Chapter 8
# How 'Object-Oriented' Is Our Design?

## 8.1 Introduction

In the course of the last two chapters, we have seen that the design process involves making several choices. This is quite typical of any engineering design, and should come as no surprise. For instance, when a bridge is designed, an engineer is starting with an architect's plan and is making choices about the kind of materials needed. While doing this, the engineer is typically guided by well-formulated design rules.

Given the multitude of choices that we face during the object-oriented design process, it is only natural to ask if there is a set of rules that can help us make the correct decisions. More specifically, we would like some way of answering two questions a designer often grapples with:

1. *Have I made the right decision in assigning responsibilities?*
and
2. *In case I make a mistake, how can I detect it early and correct it?*

In this chapter we demonstrate via examples how an awareness of a concept known as **refactoring** can help answer the above questions. Refactoring is defined simply as the process of improving the internal structure (design and code) of a piece of software without altering the module's external behavior. The process may be applied to a system in production, or we can use this process just as effectively during development. Practitioners have developed a set of rules that can be used systematically to refactor code. Some of these rules serve as means for detecting where modifications are needed, and it is not surprising that they can often be turned into guidelines for good software practice. The rules are relatively simple and the changes we make are usually small, so it is usually the case that not much goes wrong while refactoring. Familiarity with these rules can help a beginner make decisions about how to assign responsibilities, when to introduce inheritance, etc.

It should be noted that there is a vast amount of knowledge on the subject of refactoring, and our treatment of it in this book merely scratches the surface. Nonetheless, it is useful to see this as an integral part of the object-oriented design process.



B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_8

## 8.2 A First Example of Refactoring

Our first example illustrates how the two refactoring rules, EXTRACT METHOD and MOVE METHOD, are applied during system development. To serve as the platform for using these rules, we impose some new requirements to the library system we designed and implemented in Chap. 7. After constructing an initial design and implementing the code, we refine the solution using refactoring rules.

In Sect. 8.2.1 we describe the new requirements and come up with an implementation. Refactoring is done in Sect. 8.2.2.

### *8.2.1 A Library that Charges Fines: Initial Solution*

Consider the situation where the library decides to cut down on truancy by imposing fines. When an overdue book is returned, the librarian would like to know the amount of fine and send out a notice to the user regarding the fine payable. The system should therefore compute the fines and display the relevant information. The resulting changes in the business process are captured in the use case in Table 8.1.

**Table 8.1** *Use-case* Book Return with Fines

| Actions performed by the actor | Responses from the system |
|---|---|
| 1. The member arrives at the return counter with a set of books and gives the clerk the books | |
| 2. The clerk issues a request to return books | |
| | 3. The system asks for the identifier of the book |
| 4. The clerk enters the book identifier | |
| | 5. If the identifier is valid, the system marks that the book has been returned and informs the clerk if there is a hold placed on the book; otherwise (that is, in case of an invalid id), it notifies the clerk that the identifier is not valid. If there is a fine involved, the system computes the amount of fine using *Rule 5* and adds it to the user's account and information about the member is displayed. It then asks if the clerk wants to process the return of another book |
| 6. If there is a hold on the book, the clerk sets it aside. He/she then informs the system if there are more books to be returned | |
| | 7. If the answer is in the affirmative, the system goes to Step 3. Otherwise, it exits |

This use case for Book Return with Fines is similar to what we had earlier, with one addition—the amount of fine owed is computed whenever a book is returned. Obviously, the `Member` class needs to be changed to track the amount of fine owed. Also, notice that the use case does not say anything about actually collecting fines from a member and updating the corresponding `Member` object after the fine is paid. These are left as exercises.

We have the following formula for computing the fine:

**Rule 5** *New books (less than a year old) are charged $0.25 for the first day and $0.10 for every subsequent day. Older books are charged $0.15 cents for the first day and $0.05 for every subsequent day. If a book has a hold on it, the amount of fine is doubled.*

Before we construct the modified sequence diagram, we have to decide where the amount of fine owed will be computed. There are three possible options: `Library`, `Book`, and `Member`. We can make a case for each option: `Book` would be appropriate since it is the return of the book that incurs a fine; `Member` is where the fine is stored and is therefore the place it could be computed; since both `Book` and `Member` are involved in this, `Library` is perhaps the best place to do the computation. We decide (somewhat arbitrarily) that `Library` is the place where the fine is computed. The new sequence diagram for returning books is shown in Fig. 8.1.

The `returnBook` method in Library must now check if a fine is involved: if so, it computes the fine and updates the corresponding `Member` object by calling the `Member's addFine` method, so that the fine is accumulated. The books title is also passed so that a transaction can be created to keep a record of the fine.

The `returnBook` method returns a code that indicates if a fine was involved, so that the interface can alert the library clerk. For this purpose, two new return codes are introduced:

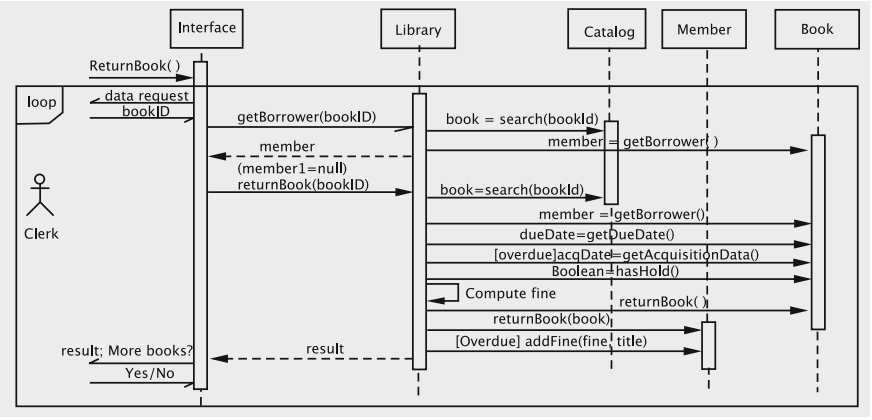


**Fig. 8.1** Returning a book and checking for fines

- `BOOK_HAS_FINE`, which is returned when the book has a fine, but no holds.
- `BOOK_HAS_HOLD_FINE`, which is returned when the book has both a fine and at least one hold.

The assumption is that the code in the user interface will take appropriate action to notify the clerk in the above circumstances.

A somewhat knotty problem concerns the display of member information, as required in Step 5 of the use case, when there is a fine involved. Since the `returnBook` method in `Library` simply returns an integer value to `UserInterface`, the latter does not have the necessary information to display. We have a couple of options:

- **Option 1** Modify the `returnBook` method to return more information. The information could be sent as an object with multiple fields, or simply as a string with all the data concatenated.
- **Option 2** Allow the user interface to manage the output by providing additional query operations. These operations would require adding more methods to the `Library` class.

Implementing the first option requires that we either create a new class to send the result or assemble a string that will have to be parsed in the interface. Neither of these is a good idea since both result in additional coupling between the UI and the back end. On the other hand, adding another query is a very natural thing to do, since our back end is in fact a database. In our situation, we need a new method that returns the borrower of a given book. This query can be invoked by `UserInterface` at the start of the process, so that it has all the necessary information. This is truly a natural extension of the code development process since the query could conceivably be used in multiple situations, not just when a book is returned. For instance, the library may want to know who the borrower of a book is because its due date has been well past.

We have to make some other changes to our implementation. `Book` now has an `acquisitionDate` field and an associated accessor. The private methods `yearApart` (which checks if two given dates are at least one year apart) and `daysElapsedSince` are added to `Library`.

The resulting code is given below.

```
public int returnBook(String bookId) {
  // search for book and its borrower
  Book book = catalog.search(bookId);
  if (book == null) {
    return(BOOK_NOT_FOUND);
  }
  Member member = null;
  if ((member = book.getBorrower()) == null) {
    return(BOOK_NOT_ISSUED);
  }
```

```
    //compute fines
    double fine = 0.0;
    Calendar dueDate = book.getDueDate();
    if (System.currentTimeMillis() > dueDate.getTimeInMillis()) {
      Calendar acquisitionDate = book.getAcquisitionDate();
      if (yearApart(acquisitionDate, dueDate)) {
        fine = 0.15 + 0.05 * daysElapsedSince(dueDate);
      } else {
        fine = 0.25 + 0.1 * daysElapsedSince(dueDate);
      }
      if (book.hasHold()) {
        fine *= 2;
      }
    }
    // final steps
    if (!(member.returnBook(book))) {
       return(OPERATION_FAILED);
    }
    if (fine > 0.0) {
       member.addFine(fine, book.getTitle());
       if (book.hasHold()) {
         return(BOOK_HAS_HOLD_FINE);
       } else {
         return(BOOK_HAS_FINE);
      }
    }
    if (book.hasHold()) {
       return(BOOK_HAS_HOLD);
    }
    return(OPERATION_COMPLETED);
  }
  private boolean yearApart(Calendar date1, Calendar date2) {
    return ((date2.getTimeInMillis() - date1.getTimeInMillis())
                                       / 86400000) > 365;
  }
  private int daysElapsedSince(Calendar date) {
    return (int) ((System.currentTimeMillis() - date.getTimeIn
      Millis())
                                                / 86400000);
  }
```

From the formula for computing fines, we see that we need to compare the due date and the current date and a fine is imposed if the latter is later than the former. To see how to determine which of these two dates is larger, notice the two lines after the comment `//compute fines`. The static method

`System.currentTimeMillis()` gives the number of milliseconds elapsed since January 1, 1970 and `dueDate.getTimeInMillis()` is the number of milliseconds elapsed since January 1, 1970 for the book's due date. A simple comparison then affords the result.

The method `yearApart()` is used to check if the book's due date is a year or more than its acquisition date. Invoking the method `getTimeInMillis` on the two dates does the trick.

After the fine is computed, the corresponding `Member` object is updated by calling the `addFine` method, so the fine is accumulated. The book's title is also passed, so a transaction can be created to keep a record of the fine.

### *8.2.2 Refactoring the Solution*

Having come up with an initial design and its implementation, we must carefully consider the two questions we said we must ask of the solution: whether the responsibilities have been properly assigned and if mistakes have been made, how to detect and correct them.

We begin with making some observations about the new method:

- It is bigger than before.
- It has more detail.

The second observation is particularly alarming. One of the broad goals we have in object-oriented design is to make each method simple so that unit testing is facilitated. Sometimes longer methods are unavoidable; but excessive amount of detail is usually a more serious indicator that we are making a mistake. Let us first revisit the whole algorithm for returning a book. Here are the steps:

1. Get the reference to the book.
2. Get the reference to the member.
3. Get the due date.
4. Get the acquisition date.
5. Compute fines.
6. Record that the member has returned the book.
7. Add fines to member.
8. Check if there is a hold.
9. Return a result. (If there is a hold, fine, etc.)

Each of these steps except 5 is an application of a single method, which is computed on some object, viz., `catalog`, `book`, or `member`. In the code corresponding to Step 5, we see that we are dealing with a lot of detail about how the fine is being computed. Modular design principles suggest that such details be abstracted out.

The above observation leads us to our first refactoring rule, Extract Method. Considerations involved in applying this rule and the steps for carrying it out are detailed in Fig. 8.2.

EXTRACT METHOD RULE

*If you have a code fragment that can be grouped together, turn the fragment into a method and assign it a name that explains the purpose of the method.*

It is easy to recognise these fragments from the comments added by the programmer. These comments, which typically take the form of a verb phrase, also suggest how the extracted method should be named. If a code fragment does not appear to have a simple name, it is often unlikely to be a good candidate for extraction into a method. Another indicator is the number of local variables that are modified; if the code fragment modifies only one variable, this strengthens the case for extraction. If a large number of variables are modified, the code fragment should probably be left in place.

The steps involved in applying this rule are as follows.

- Identify a code fragment and copy it into a method named for the intention of that code fragment.
- In the extracted code, locate the references to variables local to the original method and pass these as parameters to the new method.
- For all temporary variables that are used in the fragment, declare corresponding variables in the new method.
- Determine the local variable that is modified by the extracted code and set its type as the return type of the new method.
- Replace the code fragment in the original code with a call to the new method and store the value returned in the local variable identified in the previous step.

**Fig. 8.2** Extract method

Note that the fragment that we want extracted is preceded by the comment 'compute fines'. This suggests how the extracted method should be named. We now have the following version of the method.

```
public int returnBook(String bookId) {
  // search for book and its borrower (not shown)

  fine = computeFine(book);

  // final steps not shown
}

public double computeFine(Book book) {
  double fine = 0.0;
  Calendar dueDate = book.getDueDate();
  if (System.currentTimeMillis() > dueDate.getTimeInMillis()) {
```

```
      Calendar acquisitionDate = book.getAcquisitionDate();
      if (yearApart(acquisitionDate, dueDate)) {
        fine = 0.15 + 0.05 * (daysElapsedSince(dueDate) - 1);
      } else {
       fine = 0.25 + 0.1 * (daysElapsedSince(dueDate) -1);
      }
      if (book.hasHold()) {
        fine *= 2;
      }
    }
    return fine;
  }
```

The method `returnBook` looks much cleaner now. All it is doing is getting the relevant information by applying appropriate methods and then compiling all the results.

Let us take a closer look at the method that we have extracted. The logic employed by `computeFine` involves examining the fields of `Book` and making decisions based on the values stored in these fields. To get these values, the method repeatedly invokes the accessor methods of book. One of the rules of good object-oriented design is called the Law of Inversion, which says that

> If your routines exchange too many data, put your routines in your data.

What this means is that our focus should be more on the data and less on the process. In a process-oriented design, we do not think adversely about importing all the data elements into the function that implements the process. In a data-centered approach, the parts of the process that are close to one data element are encapsulated as methods and placed into the class corresponding to that data element. The computation for the encapsulated part of the process is then carried out by calling the method on the data element.

The above design principle leads us to the next refactoring rule, MOVE METHOD. The `computeFine` method is moved from `Library` to `Book` using the principles set forth in Fig. 8.3.

After applying the MOVE METHOD rule, have the following code:

```
public int returnBook(String bookId) {
  // search for book and its borrower (not shown)

  fine = book.computeFine();

  // final steps not shown
}
```

MOVE METHOD RULE

If we have a method that is using more features of another class than the class on which it is defined, then the method needs to be moved to the class whose features it is using the most.

This rule is a manifestation of the process of assigning responsibilities to the appropriate class and is perhaps the most frequently applied rule in refactoring. When a method uses too many features of another class, we have a situation where the classes are either collaborating too much or are too tightly coupled. It is not always the case that such a problem will be resolved by moving a method. Sometimes, other patterns may have to be applied that allows objects to communicate without getting too entangled in each other's methods. The simplest and most obvious situation is when a method accesses several fields of another class and almost all its computation is done on these fields.

The steps involved in applying this rule are as follows:

- Make a list of all features used by the method in question.
- Identify the *target* class for the move, i.e, the class whose features are most frequently employed in the computation.
- Examine other features that are not in the most frequently used class and decide if those features need to be moved to the target class as well.
- It could happen that the features from the source that are being moved to the target are being used by other methods in the source. The possibility that these methods also need to be moved should be taken into consideration. It is sometimes easier to move a set of methods and fields instead of a single method.
- Declare the method(s) and field(s) in the target class, and move the code to the new method. Make the necessary adjustments so that the code works in the target class. This would involve changing the names of the features being used.
- Change the code in the source class to reflect the movement of the fields and methods.

As is evident from this description, moving a collection of methods and fields can affect several methods of the source class. Care must be taken to ensure that the new code reflects the changes. When this rule is applied in the presence of inheritance, we have to exercise an additional caveat: *If super-classes and sub-classes of the source class have also declared the method, then the method cannot be moved unless the polymorphism can also be expressed in the target class.*

**Fig. 8.3** Move method

The computeFine method in Book is as follows:

```
public double computeFine() {
  double fine = 0.0;
  if (System.currentTimeMillis() > dueDate.getTimeInMillis()) {
    if (yearApart(acquisitionDate, dueDate)) {
       fine = 0.15 + 0.05 * (daysElapsedSince(dueDate) - 1);
    } else {
      fine = 0.25 + 0.1 * (daysElapsedSince(dueDate) -1 );
    }
    if (hasHold()) {
      fine *= 2;
     }
  }
  return fine;
}
private boolean yearApart(Calendar date1, Calendar date2) {
  return ((date2.getTimeInMillis() - date1.getTimeInMillis())
   / 86400000)
                          > 365;
}
private int daysElapsedSince(Calendar date) {
  return (int) ((System.currentTimeMillis()
                - date.getTimeInMillis()) / 86400000);
}
```

Note that we have moved the methods yearApart and daysElapsedSince as well to Book. This process has helped resolve our dilemma about where the fine should be computed. In the initial stages of design, we need not go through the entire process of refactoring to correct our errors. Nonetheless, beginners may often find themselves in a quandary as to where the responsibilities for a certain task should be placed. The exercise of refactoring code helps to formalise some of the basic principles of object-oriented design so that such errors can be caught early in the design process and suitably corrected.

## 8.3 A Second Look at RemoveBooks

Now that we have an idea of the kinds of issues that we have to watch out for, let us take a second look at the code that we have written to find suitable candidates for refactoring. The sequence diagram for Remove Books looks interesting, since it bears some resemblance to Return Books. Once again, we begin by describing the overall algorithm being followed.

1. Get the reference to the book object from `Catalog`.
2. Check if the book can be removed. We cannot remove a book if it has holds or if it is checked out.
3. If the book is not removable, return the appropriate error code.
4. If the book is removable, remove the reference to the book from `Catalog` and return the appropriate code.
5. We reach this step only if there was a problem removing the book from the catalog. In this case, return an error code.

The second step is the one that is not being carried out by a single method call and, therefore, is our focus for further investigation. Here is the code, with some comments inserted.

```
public int removeBook(String bookId) {
  // Step 1: Get reference to book.
  Book book = catalog.search(bookId);
  if (book == null) {
    return(BOOK_NOT_FOUND);
  }

  // Step 2: Check if book is removable
  if (book.hasHold()) {
    return(BOOK_HAS_HOLD);
  }
  if (book.getBorrower() != null) {
    return(BOOK_ISSUED);
  }

  // Step 3: Attempt the actual removal.
  if (catalog.removeBook(bookId)) {
    return (OPERATION_COMPLETED);
  }

  // Step 4: This error should not happen.
  return (OPERATION_FAILED);
}
```

In Step 2, we have a situation similar to the previous example in that the information stored in `Book` is being used to make a decision in `Library`, with the difference that in this case, we see very little computation being carried out. Our decision, however, should not be based on this fact alone (in a more complicated example, we could have several other reasons for not deleting a book, viz., we may have some 'rare books' that should not be removed, etc.), but should consider where the responsibility for this computation is best assigned. The repeated access to the fields of book suggests that this computation be moved out. As before, we can apply the EXTRACT METHOD and

MOVE METHOD rules in succession. We have the following situation after applying EXTRACT METHOD:

```
public int removeBook(String bookId) {
  // Step 1: Same as before
  // Step 2: Check if book is removable

  int returnCode = checkRemovability(book);
  if (returnCode != OPERATION_COMPLETED) {
     return returnCode;
  }

  // Remaining steps same as before
}
private int checkRemovability(Book book) {
  if (book.hasHold()) {
    return (BOOK_HAS_HOLD);
  }
  if (book.getBorrower() != null) {
    return (BOOK_ISSUED);
  }
  return OPERATION_COMPLETED;
}
```

Since `checkRemovability` uses attributes of `Book`, it appears that we must apply the MOVE METHOD rule. After moving this method to `Book` we get the following end product.

```
public int removeBook(String bookId) {
  // Step 1: Same as before

  // Step 2: Check if book is removable
  int returnCode = book.checkRemovability();
  if (returnCode != OPERATION_COMPLETED) {
    return returnCode;
  }

  // Remaining steps same as before
}
```

In Book we have to add the new method, taking care to change "book" to "this." The constants belong to Library, so they need to be qualified.

```
public int checkRemovability() {
  if (hasHold()) {
    return (Library.BOOK_HAS_HOLD);
  }
  if (borrowedBy != null) {
    return (Library.BOOK_ISSUED);
  }
  return Library.OPERATION_COMPLETED;
}
```

Let us now pause and take stock of what we have accomplished. We started with one method in Library with two conditional statements that invoked methods from Book. We now have a new version that uses named constants that are defined in Library. This increases the coupling between Book and Library. The benefit provided by these changes is questionable, and on the flip-side, we have added to the complexity of our code. In such a situation it is perhaps better not to modify the original code.

It is important to note here that the control flow in the extracted code for Return Book has a single point of entry and a single point of exit, which makes it well-suited for applying the refactoring rules. The multiple exit points in the extracted code of Remove Book prevent us from reaping significant gains by refactoring.

## 8.4 Using Generics to Refactor Duplicated Code

As a means to reduce system complexity and development and maintenance effort, it is important to look for opportunities where the number of classes in a system is kept as small as possible, subject, of course, to good object-oriented design principles. Prospects for merging two or more classes arise if they have similar functionality, although they may differ in relatively minor aspects. In the library system, for instance, MemberList and Catalog are strikingly similar in what they do; of course, there are some differences: one stores books and the other is a collection of members, and Catalog has a method to remove books, but no such functionality exists in MemberList.

In this section, we show how generics may be used in situations such as the above to factor out some of the commonalities in a manner that most of the complexity is located in one module. As an example of the use of generics, we develop a generic class called ItemList, which can be used to store books or members.

### 8.4.1 A Closer Look at the Collection Classes

We somehow need to tell the system that `ItemList` should be capable of storing either books or members. For this, the type of element to be stored in the `ItemList` object is passed as a parameter to the class name itself as given in the following class declaration.

```
public class ItemList<T> implements Serializable {
// generic code
}
```

The reader may recall from Chap. 3 that the idea is that `T`, a parameter to the class name, stands for an arbitrary type and objects of that type will be stored in the collection.

To implement the methods of this generic collection class, we utilise the logic used in corresponding methods of `Catalog`. (As we noted before, `Catalog` is slightly more general than `MemberList` because the former contains a method to remove items from the collection.)

We need to modify the places where references to specific types occur with generic type names. Specifically, we need to replace the data definition such as

```
private List books = new LinkedList();
```

with

```
private List<T> elements = new LinkedList<T>();
```

Next, we focus on `search`. Here is the code from `Catalog`:

```
public Book search(String bookId) {
for (Iterator iterator = books.iterator();iterator.hasNext();){
    Book book = (Book) iterator.next();
    if (book.getId().equals(bookId)) {
      return book;
    }
  }
  return null;
}
```

There are two problems with the code.

1. In each iteration, the id value of an object in the catalog is checked against the given book id. This constitutes fairly tight coupling between `Book` and `Catalog`; the parameter to `search` is of type `String`, so we build into `Catalog` the

information that id is of type `String`. The iterator's return type is cast as a `Book`. It also assumes the existence of a method called `getId()`. If we want to use generics and factor out the common code, this coupling has to be eliminated.
2. In the code, note that two books are considered equal (i.e., identical) if their `id` fields are equal. If the coupling between `Catalog` and `Book` is to be removed, the decision as to which field(s) should be used in the comparison should be made by `Book`, and not left for the collection class. This suggests that the code for deciding how to match the incoming object against the `Book` object must be extracted and moved to `Book`.

### A Caveat on Using the Equals Method

We have seen that the responsibility for checking whether a specific book's id is equal to that of some given id should be delegated to the `Book` class itself. At first sight it would appear that we can use the `equals` method to carry out this task. However, a careful look at the definition of this method as described in the Java documentation reveals that this method is unsuitable for use in the present context. Carefully read the following, which is taken from the Java online documentation.

```
The equals method implements an equivalence relation on
non-null object references:

• It is reflexive: for any non-null reference value x,
  x.equals(x) should return true.
• It is symmetric: for any non-null reference values x
  and y, x.equals(y) should return true if and only if
  y.equals(x) returns true.
```

Suppose that we write the equals method in `Book` as below.

```
public boolean equals(Object object) {
  String id = (String) object;
  return (this.id.equals(id));
}
```

As we demonstrate below, the relation implemented by the method is asymmetric. Clearly, there is an implication that the `equals` method expects a `String` object. The `equals` method of the `String` class, however, will not produce the same result when a `Book` object is passed as its parameter. Trace the following piece of code:

```
String id = "id1";
Book book1 = new Book("title1", "author1", id);
System.out.println(book1.equals(id)); // call 1
System.out.println(id.equals(book1)); // call 2
```

Although invocation of the `equals` method on `book1` (commented as `call 1`) results in a `true` output, calling the method on the corresponding `String` object (commented as `call 2`) returns value `false`.

In other words, the `equals` method as implemented above will not result in an equivalence relation. We should, therefore, refrain ourselves from using that method as the vehicle for comparison. Failure to do so could ultimately result in subtle bugs that are likely to be quite difficult to catch: imagine the plight of a programmer who trusts that the above implementation of `equals` follows the requirements set forth in the Java documentation and codes as in `call 2` above!

### A Different Approach

To rectify the situation, we implement a method called `matches` in `Book`, which does not impose the equivalence relation requirement. To start with, we implement the code as shown next:

```
public boolean matches(String bookId) {
  return this.id.equals(bookId);
}
```

The `search` method in `Catalog` is modified as follows:

```
public Book search(String value) {
  for (Book element: elements) {
    if (element.matches(value)) {
      return element;
    }
  }
  return null;
}
```

Next, we eliminate type dependence. For this, we replace the type name `Book` with the generic type `T`. (Recall that `T` is the parameter to the class.)

```
public T search(String value) {
  for (T element: elements) {
    if (element.matches(value)) {
      return element;
    }
  }
  return null;
}
```

This change begs the question: *What if* `id` *were to be of a type other than* `String`*?* This additional type dependence is eliminated by introducing a second generic parameter. The class `ItemList` is now defined as:

```
public class ItemList<T, K> implements Serializable {
  // generic code
}
```

`K` represents the type of key on which the container matches items. The `search` is now written as:

```
public T search(K value) {
  for (T element: elements) {
    if (element.matches(value)) {
      return element;
    }
  }
  return null;
}
```

Similar modifications can be made to the other methods. The changes are fairly straightforward.

```
public boolean removeItem(K value) {
  T element = search(value);
  if (element == null) {
    return false;
  } else {
    return elements.remove(element);
  }
}
public boolean insertItem(T item) {
  elements.add(item);
  return true;
}
public Iterator<T> getItems() {
  return elements.iterator();
}
```

While this solution is satisfactory for our limited case, in a more general situation one may wish to use `ItemList` to create other collection classes. If we were to replace `T` by some user-defined class `C`, the code would fail to compile if the method `matches` was not defined for class `C`. In other words, to create instantiations of `ItemList`, we require that `T` satisfy a specific property, i.e., have a method named

matches. This property is named Matchable and is extracted as an interface that T must implement.

```
public interface Matchable<K> {
  public boolean matches(K other);
}
```

The Book and Member classes are modified as below.

```
public class Member implements Serializable, Matchable<String>
{  // fields and other methods
  public boolean matches(String id) {
    return this.id.equals(id);
  }
}

public class Book implements Serializable, Matchable<String> {
  // fields and other methods
  public boolean matches(String id) {
    return this.id.equals(id);
  }
}
```

Finally, ItemList is defined as:

```
public class ItemList<T extends Matchable<K>, K> implements
  Serializable {
   // generic code
}
```

### 8.4.2 *Instantiating* `Catalog` *and* `MemberList`

With the code developed so far, we can create a new catalog as below.

```
ItemList<Book, String> catalog = new ItemList<Book, String>();
```

A similar code can be used to create a collection for members.

However, from the viewpoint of robustness, this approach is unsatisfactory. There can be multiple catalogs and member lists because the constructor can be invoked from the outside. In other words, the class is not a singleton.

Ideally, we would like to put within ItemList<T, K> a static method that returns an ItemList<T, K> object with the correct parameter. The code should look like the following.

```
private static ItemList<T, K> itemList;
private ItemList() {
}
public static ItemList<T,K> instance() {
  if (itemList == null) {
    itemList = new ItemList<T,K>();
  }
  return itemList;;
}
```

Unfortunately, the above code is not legal. Because of the way Java implements generics, the type name `T` is *erased* from the class definition at compilation time and is not available during execution. Therefore, there can be no useful checks against the type name `T`. (It would appear that the implementation details are driving the rules of the language, and not vice-versa!)

`Catalog` is now declared as an extension of `ItemList<Book, String>`.

```
public class Catalog extends ItemList<Book, String> {
}
```

Now, every `public` and `protected` method of `ItemList<T extends Matchable<K>, K>` is inherited by `Catalog`.

`MemberList` is coded in a similar fashion.

We now have two choices for naming the methods of `Catalog` and `MemberList`:

1. We could create methods such as `removeBook` and `insertBook` inside `Catalog` and similarly-named methods in `MemberList`. Thus, instead of having methods with names such as `removeItem` and `insertItem`, we end up with the old method names: `removeBook`, `insertBook`, etc.

   ```
   public boolean removeBook(String value) {
     return super.removeItem(value);
   }
   public boolean insertBook(Book item) {
     return super.insertItem(item);
   }
   public Iterator<Book> getItems() {
     return super.getItems();
   }
   ```

   This means that `Catalog` is a *class adapter,* i.e., it is a subclass of `ItemList<T extends Matchable<K>, K>` and implements a different interface by suitably calling the methods of the superclass.
2. We simply live with the new names `insertItem` and `removeItem`, and then modify the `Library` class to adjust to these changes.

While refactoring a module or a set of modules within a system, it is clearly preferable to ensure that the changes do not require modifications in the rest of the system. In our case, if we choose Option 2, `Library` needs to be updated, which would mean that we should choose Option 1. The number of places in `Library` that refers to these methods is small, so a case could be made for Option 2. In general that is not advisable, however, because there could be many modules with numerous locations that could be affected.

Making `Catalog` a singleton is not hard. See the following code.

```
private static Catalog catalog;
private Catalog() throws Exception {
}
public static Catalog instance() {
  try {
    if (catalog == null) {
      return catalog = new Catalog();
    }
  } catch(Exception e) {
    return null;
  }
  return catalog;
}
```

## 8.5 Discussion and Further Reading

In this chapter our main focus was to show the importance of being aware of the refactoring rules and how these can in fact lead us to making good choices in the way we assign responsibilities. In practice, being faithful to the refactoring process also results in software that is easier to maintain and understand.

The book by Fowler [1] is the reference for much of the material in this chapter. Among other things, the book emphasises the importance of the role that refactoring can play in keeping a system from falling into decay. While the benefits of refactoring are many, there are also a few caveats one should follow to avoid going overboard, and there are also situations and systems whose characteristics make refactoring difficult. The reader would be well advised to engage in a deeper study of this process before attempting a wider application.

Fowler points out that refactoring, when added to the design process, has the capacity to present us with an alternative to the conventional 'up-front' design which views the development of the design as a blueprint and considers coding to be just a process of going through the mechanics of implementation. While this up-front approach is certainly the one recommended by most textbooks, the process can be tempered by refactoring. Instead of getting the design down to the last detail and then coding it, we work with a loosely defined design, start the coding and 'firm-up'

(and correct) the design with some refactoring as we go through the implementation process. This process may be better description of what happens in practice and has the added advantage of giving the designers some flexibility in the choices that they make.

## 8.6 Exercises

1. Critically examine the design decisions that you have made in the three student projects at the end of Chap. 7 in the light of the information and ideas contained in this chapter. What changes would you like to make?
2. What changes do you need to make to the `Member` class to track the amount of fine owed?
3. Try to implement `ItemList<T extends Matchable<K>, K>` as a singleton. What are the difficulties you encounter?
4. Suppose that we do not specify `Matchable` as a generic interface. What changes will you make? What drawbacks do you foresee?
5. Compile the source files for the classes given for the generic implementation. Make modifications so that all compiler warning messages disappear.
6. In Chap. 7, we pointed out that using 'magic numbers' is poor programming practice, and we replaced them with named constants. This is listed in the literature as a standard refactoring rule [1], *Replace magic number with symbolic constant*, which involves the following steps:

   (a) Declare a constant and set it to the value of the magic number.
   (b) Find all occurrences of the magic number.
   (c) See if magic number matches the usage of the constant; if yes, replace the magic number by the constant.
   (d) Compile and test; code should work exactly as before.

   It has been noted that using named constants does not solve all problems since these can still be interpreted as numbers. A much safer approach in Java is to use the `enum` construct.
   Develop a refactoring process to replace named constants with enums and apply this to refactor the code developed for the Library so that the result codes returned by `Library` are all contained in a single `enum` named `LibraryResults`. (Hint: this will involve finding references to these named constants in all situations, which include variable declarations and return types of methods.)
7. Modify the library system so that it actually collects the fine owed by a user at the time he/she checks out books.

## Reference

1. M. Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, New York, 1999)