Chapter 6

Brahma Dathan Sarnath Ramnath

May 13, 2019

Contents

6 Implementation **5**
6.1 Creation of the Entity Classes: Member and Book 5
6.1.1 Transferring Data Among Business Processes 6
6.1.2 The `UserInterface` Class 8
6.1.3 Structure of `Library` 9
6.1.4 Adding Books 9
6.2 Issuing Books 11
6.3 Placing a Hold 12
6.4 Returning Books 14
6.5 Process Holds 15
6.6 Removing a Hold 16
6.7 Removing Books 16
6.8 Renewing Books 17
6.9 Displaying Transactions 19
6.9.1 Filtered Iterator 20
6.10 Extra Functionality 22
6.11 Saving and Retrieving Data 22
6.12 Discussion and Further Reading 24
6.12.1 Conceptual, Software, and Implementation Classes 26
6.12.2 Building a Commercially Acceptable System 26

Chapter 6

Implementation

Having designed the system, we proceed to the implementation stage. In this step, we use the class structures produced by the design to implement a system that behaves in the manner specified by the model. As the coding is being done, the programmer should follow good coding and testing practices, etc. Although we touch up on some of these principles here, we do not spend a significant amount of time on them, since these are concepts common to all software implementations. Our implementation will be done in Java. Any new language concepts that need elaboration are dealt with in the context where we employ them.

To a good extent, we follow the blueprints arrived at the design stage. (Otherwise, there would be little point having a design stage.) However, we will employ support mechanisms provided by the implementation language (Java) to enhance the software structure. There are also situations where programming language restrictions may force us to tweak the design somewhat. We wish to emphasize that there will be no deviations that make the implementation too divergent from design and that all such tweaks will be well justified.

The implementation is about 1500 lines of Java code, without counting documentation. Obviously, we cannot explain every line of code, nor is it necessary. What we will do in this chapter is explain code that cannot be easily inferred from the design itself. As a result, while we may make passing remarks on the following topics that are indicated in the design, we will not spend any real estate explaining them.

1. Constructors that simply copy the parameter values to fields and/or initialize some fields. An example would be the constructor of `Book`.
2. Constants (Java `final` variables) whose identifiers are self-documenting.
3. Getter and setter methods.
4. Overrides of `Object` methods: `toString()`, `hashCode()`, and `equals()`.

6.1 Creation of the Entity Classes: Member and Book

Both `Member` and `Book` objects are instantiated from the facade class, `Library`. While the `Book` class constructor is straightforward, the `Member` class's constructor has a slight complexity. Recall that the system needs to generate a unique id for every member. Let us see how this is accomplished.

`Member` maintains the field

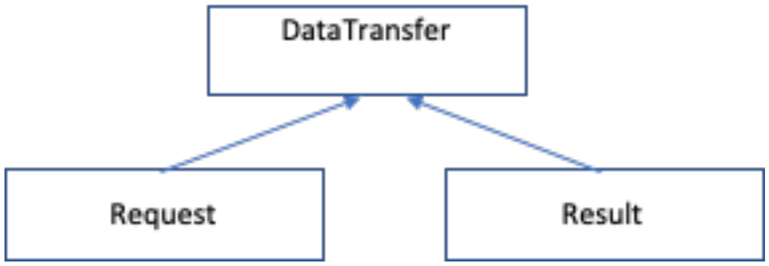

Figure 6.1: The DataTransfer class has fields to store most of the Book and Member fields, among other things. Some fields like the due date is never part of the request, so such fields are not in DataTransfer, but declared in Result. Also some fields like duration for a hold is part of a request, but not in the result.

```
private static int idCounter;
```

The `id` field in `Member` stores the unique member id. In its constructor, `Member` assigns to the field `id`, a `String` value concatenated with the value in `idCounter`. `MEMBER_STRING` is currently "M" to denote that the id is a member id. The static field `idCounter` is incremented every time a new `Member` object is created.

```
id = MEMBER_STRING + ++idCounter;
```

Let us now see how `Library` creates `Member`. Refer to Figure 5.2. `UserInterface` calls the `addMember()` of `Library` to create a `Member` object. The method has three parameters: `name`, `address`, and `phone`, all of type `String`. This is slightly problematic from the viewpoint of coding `UserInterface`. If the parameters are passed out of order, the mistake will not be caught by the compiler and this will be a bug.

6.1.1 Transferring Data Among Business Processes

To facilitate data transfer among different objects, we introduce the **Data Transfer** object. This is an object originally meant to facilitate data transfer between remotely located processes. Although the entire library system, as implemented here, resides in a single Java Virtual Machine, in practice, the user interface could be a browser and the business logic would be located in a remote server. This approach also helps reduce bugs.

A data transfer object (DTO) simply carries data that needs to be transferred between processes and does not contain any business logic. The information transferred between `UserInterface` and `Library` can be viewed as either request data coming from `UserInterface` to `Library` or results of these requests traveling from `Library` to `UserInterface`. Thus, we can organize the data shipped between the two classes into a simple hierarchy shown in Figure 6.1.

The instance fields of the `DataTransfer` class are given below. The meaning of each field should be obvious.

```
    private String bookId;
    private String bookTitle;
    private String bookAuthor;
    private String bookBorrower;
    private String bookDueDate;
    private String memberId;
    private String memberName;
    private String memberAddress;
    private String memberPhone;
```

The class has getters and setters for the fields.

The user interface classes use a `Request` object to ship parameters for a specific operation to the facade. A `Result` object is used to return the result of a facade operation. `Request` is a singleton class.

To add a member, `UserInterface` assembles the relevant parameters as below. The `getName()` method returns a sequence of characters (with possible spaces in it), after prompting the user with the `String` parameter.

```
    Request.instance().setMemberName(getName("Enter member name"));
    Request.instance().setMemberAddress(getName("Enter address"));
    Request.instance().setMemberPhone(getName("Enter phone"));
```

The `Request` object is used to send the parameters to `Library`'s `addMember()` method, which returns a `Result` object.

```
   Result result = library.addMember(Request.instance());
```

The `Result` class has the `int` field `resultCode`, which stores a value specifying the end-result of an operation carried out by `Library`. Constants (final variables) declared in the class are used instead of hard-coded numbers to specify the results. Thus, after obtaining the result of requesting member addition, `UserInterface` proceeds to interpret the result of the operation and display an appropriate result. If an operation is successful, `resultCode` is always `Result.OPERATION_COMPLETED`.

```
        Result result = library.addMember(Request.instance());
        if (result.getResultCode() != Result.OPERATION_COMPLETED) {
            System.out.println("Could not add member");
        } else {
            System.out.println(result.getMemberName()
                 + "'s id is " + result.getMemberId());
        }
```

`Library`'s `addMember()` extracts each of the three values supplied, before following the sequence of steps specified in the sequence diagram. It then assembles the result and returns it to `UserInterface`. For all methods that return a `Result` object, the code for the relevant `Library` method follows the structure given below.

```
    create a Result object
    follow the steps in the corresponding sequence diagram
         modifying the Result object as and when needed
    return the Result object
```

Here is the code for `addMember()` in `Library`.

```
    public Result addMember(Request request) {
        Result result = new Result();
        Member member = new Member(request.getMemberName(),
                  request.getMemberAddress(), request.getMemberPhone());
        if (members.insertMember(member)) {
            result.setResultCode(Result.OPERATION_COMPLETED);
            result.setMemberFields(member);
            return result;
        }
        result.setResultCode(Result.OPERATION_FAILED);
        return result;
    }
```

The following fields and their corresponding getters and setters are declared in `Request`.

```
private int holdDuration;
private Calendar date;
```

The following method sets all book-related fields. Note that if the book is not issued, `book.getBorrower()` is `null` and both `bookBorrower` and `bookDueDate` are set to `"none"`. As a result, classes in the user interface will not see a `null` value for any field.

```
public void setBookFields(Book book) {
    if (book.getBorrower() != null) {
        bookBorrower = book.getBorrower().getId();
        bookDueDate = book.getDueDate();
    } else {
        bookBorrower = "none";
        bookDueDate = "none";
    }
    bookId = book.getId();
    bookTitle = book.getTitle();
    bookAuthor = book.getAuthor();
}
```

6.1.2 The `UserInterface` Class

This is a Singleton class. In its `main()` method, `UserIntarface` creates its only instance and then calls the `process()` method, which is in a loop: in each iteration of the loop, the code reads a user command and calls the appropriate method to assemble the `Request` object and invoke the corresponding`Library` method before finally displaying the result.

```
public void process() {
  int command;
  help();
  while ((command = getCommand()) != EXIT) {
    switch (command) {
      case ADD_MEMBER:    addMember();
                          break;
      case ADD_BOOKS:     addBooks();
                          break;
      case ISSUE_BOOKS:   issueBooks();
                          break;
      // several lines of code not shown
      case HELP:          help();
                          break;
    }
  }
}
```

The `help` method displays all the options with the corresponding numeric choices. In addition to the methods for each of the menu items, `UserInterface` also has methods `getToken()`, `getName()`, `getNumber()`, `getDate()`, and `getCommand()` for reading different types of data.

An examination of the sequence diagrams shows the need to query the user in multiple situations for a "Yes" or "No" answer to different questions. For this, we have also coded a method named `yesOrNo()`, which has like all other methods to read data, a `String` parameter to prompt the user.

6.1.3 Structure of `Library`

This is also a singleton class. It has methods named as in its class diagram in Figure 5.17. Although the relevant parameters are as specified in the class diagram, they are always passed in a `Request` object as described earlier. The return values, except for `getTransaction()` and `getBooks()`, are packed into a `Result` object. The flow of every method follows the corresponding sequence diagram closely.

6.1.4 Adding Books

The code for adding books follows the sequence diagram in Figure 5.3, but as in many other business processes, it uses the `Request` and `Result` objects; The way these two objects are used closely follows the code to add a member.

The `addBooks()` method of `UserInterface` has a loop to prompt for and read in book information and add them to the library.

As Figures 5.2 and 5.3 show, after a `Book` or `Member` object is created, it should be remembered in the `Catalog` object or `MemberList`, respectively. Let us examine the considerations involved in implementing these two collections.

1. Rather than start implementing the collections from the "first principles" (such as arrays or dynamic memory), we should utilize the language facilities to the extent possible. Let us look at the `Collection` interface is in the package `java.util`. It provides basic functionality with methods such as `add()`, `remove()`, `iterator()`, and `size()`. Many of the sub-interfaces and their implementing classes such as `LinkedList` provide even richer sets of methods. However, this interface does not support functionality of the nature shown in Figure 5.15. For example, there is no way to remove a book with a call such as `removeBook(bookId)`. The supported functionality such as `myBooks.remove(book)` does not meet our requirements. This warrants the creation of a custom collection.

2. The collections should not be accessible from classes outside the business logic. In fact, the only class that needs to access the collections is `Library`, so it is best to declare the collections as private inner classes of `Library`.

The implementation of `Catalog` and `MemberList` follows the design shown in Figure 5.15 and they are both object adapters that use `java.util.LinkedList` as an adaptee. We now show the implementation of `Catalog`. The code for `MemberList` is quite similar.

```
private class Catalog implements Iterable<Book>, Serializable {
  private List<Book> books = new LinkedList<Book>();
```

The class supports an itarator to the collection. The need for implementing `Serilizable` will be explained later.

Adding a book is accomplished by adding it to the `books` object.

```
        public boolean insertBook(Book book) {
            books.add(book);
            return true;
        }
```

Given a book id, the corresponding book is searched for in `books` as shown below.

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

Removal of a book first involves a search to retrieve the object and subsequent removal from `books`. Note that a `Book` object is passed to the `remove()` method of `List`. That method removes the `Book` object based on the `equals()` method implemented in `Book`. That considers two `Book` objects to be equal if they have the same book id.

```
    public boolean removeBook(String bookId) {
        Book book = search(bookId);
        if (book == null) {
            return false;
        } else {
            return books.remove(book);
        }
    }
```

Implementing the `iterator()` method is simple.

```
    public Iterator<Book> iterator() {
        return books.iterator();
    }
```

6.2 Issuing Books

`UserInterface` begins the process by getting the member's id and verifies that it is correct.

```
 public void issueBooks() {
    Request.instance().setMemberId(getToken("Enter member id"));
    Result result = library.searchMembership(Request.instance());
    if (result.getResultCode() !=
        Result.OPERATION_COMPLETED) {
        System.out.println("No member with id " +
            Request.instance().getMemberId());
        return;
    }
```

if the member id is valid, `UserInterface` invokes the `issueBook()` method of `Library` repeatedly for each book to be issued. Ir remembers the member's id throughout the process.

```
    do {
        Request.instance().setBookId(getToken("Enter book id"));
        result = library.issueBook(Request.instance());
        if (result.getResultCode() == Result.OPERATION_COMPLETED) {
            System.out.println("Book " +
              result.getBookTitle() + " issued to " +
              result.getMemberName() + " is due on " + result.getBookDueDate());
        } else {
            System.out.println("Book could not be issued");
        }
    } while (yesOrNo("Issue more books?"));
```

Some of the representative and interesting lines of `issueBook()` in `Library` are given below. The code verifies that the member id is valid for each invocation. The `Result` object has been initialized with `"none"` in every field, so if the memebr could not be located, the method simply sets `resultCode` to NO_SUCH_MEMBER and returns the result. Otherwise, the member-related fields are set to the appropriate values. The book fields are set if and only of the book could be issued.

```
Member member = members.search(request.getMemberId());
if (member == null) {
    result.setResultCode(Result.NO_SUCH_MEMBER);
    return result;
}
result.setMemberFields(member);
if (!(book.issue(member) && member.issue(book))) {
    result.setResultCode(Result.OPERATION_FAILED);
} else {
    result.setResultCode(Result.OPERATION_COMPLETED);
    result.setBookFields(book);
}
```

The `issue()` method in `Book` simply records the fact that the book is issued to the correct member. It generates a due date by adding one month to the date of issue. Note the extra code needed to ensure that the book is due by 11:59:59 on the due date. We needed to cater to the quirks of the `GregorianCalendar` class to get this properly set.

```
    public boolean issue(Member member) {
        borrowedBy = member;
        dueDate = new GregorianCalendar();
        dueDate.set(Calendar.HOUR, 0);
        dueDate.set(Calendar.MINUTE, 0);
        dueDate.set(Calendar.SECOND, 0);
        dueDate.add(Calendar.HOUR, 11);
        dueDate.add(Calendar.MINUTE, 59);
        dueDate.add(Calendar.SECOND, 59);
        dueDate.add(Calendar.MONTH, 1);
        return true;
    }
```

We list below the implementation of `issue()` in `Member`. The two collections `booksBorrowed` and `transactions` are implemented using the JDK class `LinkedList`.

```
    private List<Book> booksBorrowed = new LinkedList<Book>();
    private List<Transaction> transactions = new LinkedList<Transaction>();
    public boolean issue(Book book) {
        if (booksBorrowed.add(book)) {
            transactions.add(new Transaction("Issued", book.getTitle()));
            return true;
        }
        return false;
    }
```

The collection of `Transaction` objects is maintained as a `LinkedList`. We will have more to say about this in Section 6.9.

6.3 Placing a Hold

To support a member place a hold on a book, the book id, member id, and duration are stored in the `Request` object and sent to `Library`. After verifying the existence of the member and book, `Library` executes the following code, which essentially follows the sequence diagram in Figure 5.6.

```
        Calendar date = new GregorianCalendar();
        date.add(Calendar.DATE, request.getHoldDuration());
        Hold hold = new Hold(member, book, date);
        book.placeHold(hold);
        member.placeHold(hold);
        result.setResultCode(Result.OPERATION_COMPLETED);
        result.setBookFields(book);
        return result;
```

The implementation of `Hold` is quite simple, with the code a direct java translation of the class diagram in Figure 5.16. The constructor, getters and setters deserve no explanation. The only method with a modicum of complexity is the following, where hold checks whether it is still valid by comparing the date and time stored in it against the current date and time. `System.currentTimeMillis()` returns the number of millisecond between the current instant in time and some reference time say, T. Similarly, `date.getTimeInMillis()` is the number of milliseconds between the the time stored in `date` (the last valid date of the hold) and the same reference time T. If the former is smaller than the latter, clearly the hold is valid.

```
public boolean isValid() {
        return (System.currentTimeMillis() <
            date.getTimeInMillis());
}
```

Here is how the `placeHold()` method in `Member` is implemented. The method creates a `Transaction` object and adds it to the collection in `Member` and then stores the `Hold` object in a collection that represents the list of holds placed by this member.

```
    public void placeHold(Hold hold) {
        transactions.add(new Transaction("Hold placed", hold.getBook().getTitle()));
        booksOnHold.addHold(hold);
    }
```

The question we should address here is how the collection of holds for a member should be implemented. In general, to choose an appropriate data structure for a data type, we should consider all of the operations acting upon the ADT. An important method to operate on the collection is removal of a hold. For efficiency, we store a collection of `Hold` objects in both `Book` and `Member`. Consider the following issue. Given a `Book` object `book`, `Member` should remove the `Hold` object corresponding to `book`. Although the Java `Collection` interface has a `remove()` method, it is not very convenient to use it in this context, because the method would need a `Hold` object, which we do not have. To clarify, we need to search for the `Hold` object in the collection associated with a `Member` object, looking for a specific book id. This warrants the creation of a collection class, which we name `HoldList`.

`HoldList` for a book should order the `Hold` objects in the chronological order of their insertion times. It is implemented as an object adapter, using `LinkedList` as an adaptee.

```
    private List<Hold> holds = new LinkedList<Hold>();
```

The collection is a queue. So a `Hold` objects is added at the end of the list.

```
    public boolean addHold(Hold hold) {
        holds.add(hold);
        return true;
    }
```

A `Hold` object related to a given book is removed as below. Note the use of `ListIterator`, which allows removal during iteration.

```
    public Hold removeHoldOnBook(String bookId) {
        for (ListIterator<Hold> iterator = holds.listIterator(); iterator.hasNext();) {
            Hold hold = iterator.next();
            String id = hold.getBook().getId();
            if (id.equals(bookId)) {
                iterator.remove();
                return hold;
            }
        }
        return null;
    }
```

Another method `removeHoldOnMember()` removes holds on a specific member. The code is almost identical, except for the obvious changes needed for matching based on member id.

When a book is returned, the library must check if there is a valid hold on the book. This is carried out by searching the holds collection related to the book. All invalid holds are removed and the first valid hold is located. The code uses `ListIterator` to facilitate removal during iteration.

```
    public Hold getNextValidHold() {
        for (ListIterator<Hold> iterator = holds.listIterator(); iterator.hasNext();) {
            Hold hold = iterator.next();
            if (hold.isValid()) {
                return hold;
            } else {
                iterator.remove();
            }
        }
        return null;
    }
```

The following method checks if there is a valid hold on the book.

```
     public boolean hasHold() {
        return holds.getNextValidHold() != null;
    }
```

6.4 Returning Books

To return books, `UserInterface` has a loop to read and send a book id to the `returnBook()` method of `Library` and display one of several different types of messages, depending on the result.

```
   public void returnBooks() {
        do {
            Request.instance().setBookId(getToken("Enter book id"));
            Result result = library.returnBook(Request.instance());
            switch (result.getResultCode()) {
```

```
            // different cases
            case Result.BOOK_HAS_HOLD:
                System.out.println("Book " + result.getBookTitle() + "has a hold");
                break;
            }
            if (!yesOrNo("Return more books?")) {
                break;
            }
        } while (true);
    }
```

Note that if there is a hold on the book, `returnBook()` of `Library` returns the result code `BOOK_HAS_HOLD`, and `UserInterface` displays an appropriate message on the console. This action requires a follow up, which is discussed in Section 6.5.

After ensuring that the book id valid and that it was actually issued, `Library` executes the following code.

```
    if (!(member.returnBook(book))) {
        result.setResultCode(Result.OPERATION_FAILED);
        return result;
    }
    if (book.hasHold()) {
        result.setResultCode(Result.BOOK_HAS_HOLD);
        return result;
    }
    result.setResultCode(Result.OPERATION_COMPLETED);
    result.setBookFields(book);
    result.setMemberFields(member);
    return result;
```

The `hasHold()` method of `Book` returns `true` if and only if there is a valid hold on the book.

The `returnBook()` of `Member` removes the book from its list of borrowed books and generates a `Transaction` object.

```
    if (booksBorrowed.remove(book)) {
        transactions.add(new Transaction("Returned", book.getTitle()));
        return true;
    }
```

6.5 Process Holds

The `procesHolds()` method in `UserInterface` has a `do` loop that repeatedly reads book ids and calls the `processHold()` method of `Library`, whose code is given below.

```
    public Result processHold(Request request) {
        Result result = new Result();
        Book book = catalog.search(request.getBookId());
        if (book == null) {
            result.setResultCode(Result.BOOK_NOT_FOUND);
            return result;
        }
        Hold hold = book.getNextHold();
        if (hold == null) {
            result.setResultCode(Result.NO_HOLD_FOUND);
            return result;
        }
        hold.getMember().removeHold(request.getBookId());
        hold.getBook().removeHold(hold.getMember().getId());
        result.setResultCode(Result.OPERATION_COMPLETED);
        result.setBookFields(book);
        result.setMemberFields(hold.getMember());
        return result;
    }
```

After checking that the book id is valid, the above code gets the next valid `Hold` object on the book, and it could be that there is none. If there is a valid hold, the corresponding `Hold` object is removed from both `Member` and `Book`.

6.6 Removing a Hold

Given a book id and a member id for a hold, `Library`'s `removeHold()` method validates the ids and removes the `Hold` object from both `Member` and `Hold`. Note the calls to `removeHold()` of `Member` and `Book` in the same `if` statement.

```
    if (member.removeHold(request.getBookId()) &&
        book.removeHold(request.getMemberId())) {
        result.setResultCode(Result.OPERATION_COMPLETED);
    } else {
        result.setResultCode(Result.NO_HOLD_FOUND);
    }
```

6.7 Removing Books

To remove a book with a given book id, `Library` checks for the existence of the book and whether it is borrowed or has a hold. The book is removed if it has no holds and is not borrowed. The outline of the code is given below.

```
    Book book = catalog.search(request.getBookId());
    if (book == null)
       // return error
    if (book.hasHold())
       // return error
    if (book.getBorrower() != null)
        // return error
    if (catalog.removeBook(request.getBookId())) {
        result.setResultCode(Result.OPERATION_COMPLETED);
        return result;
    }
    // return error
```

6.8 Renewing Books

The logic for renewing a book, shown in Figure 5.11, is quite complicated. It requires `UserInterface` to get information about every book issued to the member and prompt the user to repeatedly for each book to see if it should be renewed.

A crucial question here concerns the mechanism for getting the information about all checked out books to `UserInterface`. One option is to return a list of `Book` objects or an `Iterator<Book>`. This approach is risky, because `UserInterface` could misuse the `Book` reference to fulfill any nefarious intentions. To overcome this, we could create a `ReadOnlyBook` object as we discussed in Section 5.4. This will work, but the construction is rather elaborate, and, in general, this will mean multiple classes for every type of class we wish to expose.

A little easier approach is to supply an `Iterator<Result>` object to `UserInterface`. We arrange matters so that each `Result` object so returned will contain a copy of the fields of the `Book` object. This is what `Library` does to allow iteration on issued book information for a member.

```
    Member member = members.search(request.getMemberId());
    if member is valid {
        anIterator = Iterator<Result> using the Iterator
               member.getBooksIssued();
        return anIterator
    }
```

We create a new `Iterator<Result>` object using the `Iterator<Book>` object returned by `member.getBooksIssued()`. This is done as follows.

We create a new class called `SafeIBookterator`, which requires an iterator of `Book` objects. Suppose the original `Iterator<Book>` reference is `iterator`. Then `iterator.next()` returns a `Book` object. We create a `Result` object and copy the fields of `Book` using the `setBookFields()` of `Result`.

```
    Result result = new Result();
    public Result next() {
        if (iterator.hasNext()) {
            Book book = iterator.next();
            result.setBookFields(book);
        } else {
            throw new NoSuchElementException("No such element");
        }
        return result;
    }
```

The `hasNext()` method simply returns whatever `iterator.hasNext()` returns.

This would be the essence of the implementation of `SafeBookIterator`.

We could employ the same idea to create a safe iterator for `Member` or for any other class. But a more compact structure would have a single generic `SafeIterator` class, declared as follows.

```
public class SafeIterator<T> implements Iterator<Result> {
// implementation
}
```

We can then declare a `SafeIterator<Book>` reference, `SafeIterator<Member>`, etc., to create safe iterators for all classes we wish,

But there are some more hoops we need to jump before we have working code. The biggest hurdle is the line

```
result.setBookFields(book);
```

The code explicitly uses the method `setBookFields()` of `Result`.

For a `Member`, the code would have been

```
result.setMemberFields(member);
```

How do we ensure that the correct method is employed, depending on the type? Here is where polymorphism comes to our rescue. We declare the following inner class in `SafeIterator`.

```
    public abstract static class Type {
        public abstract void copy(Result result, Object object);

        public static class SafeBook extends Type {
            public void copy(Result result, Object object) {
                Book book = (Book) object;
                result.setBookFields(book);
            }
        }

        public static class SafeMember extends Type {
            public void copy(Result result, Object object) {
                Member member = (Member) object;
                result.setMemberFields(member);
            }
        }
    }
```

We also provide the following fields.

```
    private Iterator<T> iterator;
    private Type type;
    private Result result = new Result();
    public static final Type BOOK = new SafeBook();
    public static final Type MEMBER = new SafeMember();
```

`Library` would instantiate `SafeIterator` as below.

```
     return new SafeIterator<Book>(member.getBooksIssued(), SafeIterator.BOOK);
```

The `next()` method would now become

```
    public Result next() {
        if (iterator.hasNext()) {
            type.copy(result, iterator.next());
        } else {
            throw new NoSuchElementException("No such element");
        }
        return result;
    }
```

So the appropriate method to copy the `Book` or `Member` object would be invoked.

6.9 Displaying Transactions

As shown in Figure 5.16, the `Transaction` class has three fields: the type of the transaction, the book title, and the date of the transaction. We keep the collection of all transactions associated with a member in the `Member` class itself. The only query we have regarding transactions is display them for a specific member on a given date.

As a result, the only operations we need to have on the collection is add new transactions and iterate over existing transactions, which means that a standard Java collection class will suffice.

Let us give some thought on how we might actually display transactions that fall on a given date, or for that matter, satisfy any other criteria. We could iterate over the collection and select all qualifying transactions, put them in a temporary collection, and supply an iterator to that collection to `UserInterface`. As we will see, we take a slightly different approach that is a little more elegant and efficient.

`Library` provides a query that returns an `Iterator` of all the transactions of a member on a given date, and this is implemented by passing the query to the appropriate `Member` object. Even if the member id is invalid, `UserInterface` gets an iterator, but there will not be any objects to ierate on. As a result, the code in `UserInterface` is straightforward.

```
    public void getTransactions() {
        Request.instance().setMemberId(getToken("Enter member id"));
        Request.instance().setDate(getDate("Please enter the date "
           + "for which you want records as mm/dd/yy"));
        Iterator<Transaction> result =
             library.getTransactions(Request.instance());
        while (result.hasNext()) {
            Transaction transaction = (Transaction) result.next();
            System.out.println(transaction.getType() + "   " +
                 transaction.getTitle() + "\n");
        }
        System.out.println("\n  End of transactions \n");
    }
```

`Library` verifies that the member id is valid. If it is not, it returns a dummy iterator.

```
    public Iterator<Transaction> getTransactions(Request request) {
        Member member = members.search(request.getMemberId());
        if (member == null) {
            return new LinkedList<Transaction>().iterator();
        }
        return member.getTransactionsOnDate(request.getDate());
    }
```

The `getTransactionsOnDate()` in tt Member has a declared return type of `Iterator<Transaction>`. But its actual type is `FilteredIterator`.

6.9.1 Filtered Iterator

A filtered iterator behaves like an iterator, but ignores all objects that do not satisfy a certain property. The class header and fields are given below.

```
public class FilteredIterator implements Iterator<Transaction> {
    private Transaction item;
    private Predicate<Transaction> predicate;
    private Iterator<Transaction> iterator;
```

`Predicate` is a generic Java interface with a single method named `test()`. For a `Transaction`, the signature of this method can be thought of as

```
    boolean test(Transaction t)
```

That is, given a `Transaction` reference `t`, `test()` must determine whether the transaction meets a certain condition. When the `FilteredIterator` object is created, it must be supplied two things:

1. An iterator of `Transaction` objects. The idea is that `FilteredIterator` will only select some of the objects. Essentially, `FilteredIterator` is an object adapter of `Iterator<Transaction>`.
2. The condition for selecting a transaction. This will be a `Predicate<Transaction>` reference.

The constructor header is

```
    public FilteredIterator(Iterator<Transaction> iterator,
        Predicate<Transaction> predicate) {
```

Conceptually, what the filtered iterator does is not very difficult. It uses the `next()` method of the adaptee to retrieve each `Transaction` object. It supplies each `Transaction` object as argument to the `Predicate` object's `test()` method. If the method returns `true`, that is the next `Transaction` object to be returned from `next()`, so it is stored in the field `item`. All of this is accomplished in the method `getNextItem()`. Note that when there are no more items, `item` is set to `null`.

```
    private void getNextItem() {
        while (iterator.hasNext()) {
            item = iterator.next();
            if (predicate.test(item)) {
                return;
            }
        }
        item = null;
    }
```

The constructor calls `getNextItem()` to initialize `item`.

```
    this.predicate = predicate;
    this.iterator = iterator;
    getNextItem();
```

If `item` is not `null`, it is an indication that there is at least one more transaction. So here is the `hasNext()` method.

```
    public boolean hasNext() {
        return item != null;
    }
```

The next item to be returned is in `item`. But before returning it, the `next()` method must find the next qualifying transaction and store it in `item`.

```
    public Transaction next() {
        if (!hasNext()) {
            throw new NoSuchElementException("No such element");
        }
        Transaction returnValue = item;
        getNextItem();
        return returnValue;
    }
```

To instantiate a `FilteredIterator` object, `Member` supplies an iterator to the `Transaction` objects and a `Predicate` object. The first one is simply `transactions.iterator()`. To create the `Predicate` object, the following code would work.

```
    private class DatePredicate implements Predicate<Transaction> {
       private Calendar date;
        public DatePredicate(Calendar date) {
            this.date = date;
        }
        public boolean test(Transaction transaction) {
            return transaction.onDate(date);
        }
    }
```

The method `getTransactionsOnDate()` in `Member` would be

```
    public Iterator<Transaction> getTransactionsOnDate(Calendar date) {
        return new FilteredIterator(transactions.iterator(), new DatePredicate(date));
    }
```

However, instead of creating a separate class to implement `Predicate` and instantiating it, we can simply supply the essential code in the implementation of the `test()` method. Given the parameter `transaction`, what the method does is evaluate `transaction.onDate(date)` and return the result. This is expressed in a more compact form called a `lambda function`.

```
    transaction -> transaction.onDate(date)
```

We can simply pass the above as the second argument. Java will interpret it to mean an instantiation of an anonymous object that implements the `Predicate` interface.

The final code for the method is thus

```
    public Iterator<Transaction> getTransactionsOnDate(Calendar date) {
        return new FilteredIterator(transactions.iterator(),
             transaction -> transaction.onDate(date));
    }
```

6.10 Extra Functionality

We provide two additional operations not covered in analysis or design. One of them lists all books and the other displays all members. `Library` uses `SafeIterator` to return an `Iterator<Result>` object for both cases.

6.11 Saving and Retrieving Data

Although a full-fledged application would possibly employ a database management system to manage data on a long-term basis, that topic is beyond the scope of this book. Instead, to help us use the system at least on a small scale, we use Java serialization and file management support to store and retrieve data from disk.

Recall that every class that needs to be serialized should implement the `Serializable` interfac, which has no methods. It is simply a marker to identify it as a class to be serialized when needed. Thus every class that contains data, that is, all classes except `UserInterface` and the two `Iterator` classes, implements this interface. For example, we have declarations such as

```
public class Member implements Serializable {
...
public class Book implements Serializable
...
public class Hold implements Serializable {
...
```

When the command to save data is invoked, `UserInterface` calls the method `save()` in `Library`.

```
     private void save() {
        if (library.save()) {
            System.out.println(" The library has been successfully saved in the file LibraryDa
        } else {
            System.out.println(" There has been an error in saving \n");
        }
    }
```

`Library` contains references to the two main collections: the catalog and the list of members. Both `Catalog` and `MemberList` contain just one field each, for storing the data. For example,

```
    private List<Book> books = new LinkedList<Book>();
```

in `Catalog` stores references to all `Book` objects. `LinkedList` implements `Serializable`, so if we attempt to serialize `Catalog`, Java will serialize `books` and thus succeed in serializing `Catalog`. This takes care of serializing all `Book` objects and all objects referred to by these `Book` objects, which include all `Member` and `Hold` objects stored in them. Serializing the `Hold` objects will also serialize all `Member` objects who have placed a hold. If we then serialize `MemberList`, it will serialize all members not already serialized: the ones who haven't borrowed or are holding books.

Thus serializing `Catalog` and `MemberList` will serialize almost all data in the library system. Java does not serialize any static fields, so the field `idCounter` in `Member` will not be serialized. To store this, we need to explicitly serialize the value in that field.

The code in `Library` to serialize the data and store it in a disk file is given below.

```
     public static boolean save() {
        try {
            FileOutputStream file = new FileOutputStream("LibraryData");
            ObjectOutputStream output = new ObjectOutputStream(file);
            output.writeObject(library);
            Member.save(output);
            file.close();
            return true;
        } catch (IOException ioe) {
            ioe.printStackTrace();
            return false;
        }
    }
```

The `save()` method in `Member` is

```
    public static void save(ObjectOutputStream output) throws IOException {
        output.writeObject(idCounter);
    }
```

The file `LibraryData` can be retrieved and deserialzied using the method `retrieve()` in `Library`. The method calls `Member`'s `retrieve()` method to restore the static field `idCounter`.

```
     public static Library retrieve() {
        try {
            FileInputStream file = new FileInputStream("LibraryData");
            ObjectInputStream input = new ObjectInputStream(file);
            library = (Library) input.readObject();
            Member.retrieve(input);
            return library;
        } catch (IOException ioe) {
            ioe.printStackTrace();
            return null;
        } catch (ClassNotFoundException cnfe) {
            cnfe.printStackTrace();
            return null;
        }
    }
```

The method `retrieve()` in `Member` is

```
    public static void retrieve(ObjectInputStream input) throws IOException, ClassNotFoundExce
        idCounter = (int) input.readObject();
    }
```

As is evident from the pieces of code shown above, the implementation follows the blueprint established by design. We made use of language features (like `LinkedList`, lambda functions, FilteredIterator), design patterns (adapter, singleton), and imparted a certain degree of code safety by using the concepts of Data Transfer Objects, safe iterators, and private inner classes.

6.12 Discussion and Further Reading

Converting the model into a working design is by far the most complex part of the software design process. Although there are only a few principles of good object-oriented design that the designer should be aware of, the manner in which these should be applied in a given situation can be quite challenging to a beginner. Indeed, the only way these can be mastered is through repeated application and critical examination of the designs produced. It is also extremely useful to study peer-reviewed designs of software systems that have been published in sources of repute, and discussing design issues with more experienced colleagues. In this chapter we have attempted to capture some of this complexity through an example, and also tried to raise and deal with the questions that trouble the typical beginner.

The sequence of topics so far suggests that the design would progress linearly from analysis to design to implementation. In reality, what usually happens is more like an iterative process. In the analysis phase, some classes and methods may get left out; worse yet, we may not even have spelled out all the functional requirements. These shortcomings could show up at various points along the way, and we may have to loop through this process (or a part of this process) more than once, until we have an acceptable design. It is also instructive to remember that we are not by any means prescribing a definitive method that is to be used at all times, or even coming up with the perfect design for our simple library system. As stated before, our goal is to provide a condensed, but complete, overview of the object-oriented design process through an example. At the end of the previous chapter three student projects were presented. To maximize benefit, the reader is encouraged to apply the concepts

Memory Management in Object-Oriented Systems

Proliferation of objects contributes in large part to the degradation of performance in object-oriented systems, which means that objects must be removed from the system in an expedient manner as soon as they have served their purpose. Objects are typically allocated in the process memory space known as the *heap.* In Java, memory allocated to an object in the heap is not reclaimed until all references to the object are set to `null`. Some languages such as C++ allows and requires the user to employ a specific operator in order to free up the space allocated to an object.

The availability of automatic reclamation of storage in Java is often touted as a boon, and it indeed is: it ensures that there are no *dangling references* or *memory leaks* in the traditional sense of these two terms. But it does not absolve the programmer from his/her responsibilities to ensure proper memory management. The reader must be aware that memory shortage and data integrity issues, which are respectively the consequences of memory leaks and dangling pointers, may manifest themselves because of design and coding errors.

The problem of memory shortage may still arise in a Java program because we may forget to set to `null` every reference to an object that should be deleted: the language's garbage collection mechanism must be given a chance to kick in, and that won't happen without some cooperation from the application code. Removing objects can be a tricky exercise and to ensure reliable performance, a systematic process is needed for removing the unwanted ones. As systems become more complex, we have more intricate relationships between the objects, which, in turn, make the unwanted objects harder to detect. In our example, `Book` and `Member` objects are relatively stable and introduced into the system in a fairly controlled manner. `Hold` objects, on the other hand, are more ephemeral and can be easily added and removed,which means that there is a potential for their numbers to explode. In the library system, we suggest that this be fixed by removing invalid holds periodically.

Dangling pointers, which imply invalid object references, could ultimately lead to illegal data access and failure. Careless design and development may result in the very same fate in a Java program. While deleting the reference to an object from one part of the system, we must be careful to ensure that any remaining references to the object from other parts of the system will not lead to inconsistencies. When deleting an object from a collection, we typically obtain a reference to the object by searching the container. If there are references to a deleted object stored in other active objects, we may up with *mutual inconsistency.* For instance, assume that we remove a book b from the catalog by deleting the reference to the appropriate `Book` object from `Catalog`. Furthermore, suppose that b has a hold h on it. This could lead to the situation where we obtain the reference to the `Book` object (corresponding to b) from the `Hold` object (corresponding to h) and use b's ID at a later point to search the catalog; obviously, this search will lead to an unexpected failure! There are two possible solutions to overcome this problem: *(i) delete the corresponding* `Hold` *object* while removing the book from the catalog or *(ii) remove the reference from* `Catalog` *only if there are no holds and the book is not currently checked out.* In our implementation, we have chosen the second solution.

Figure 6.2: Memory Management Considerations

to one or more of these projects as he/she reads through the material. From our experience, we have seen that students find this practice very beneficial.

6.12.1 Conceptual, Software, and Implementation Classes

Finding the classes is a critical step in the object-oriented methodology. In the course of the analysis-design-implementation process, the idea of what constitutes a class goes through some subtle shifts.

In the analysis phase, we found the **conceptual** classes. These correspond to real-world concepts or things, and present us with a conceptual or essential perspective. These are derived from and used to satisfy the system requirements at a conceptual level. At this level, for instance, we can identify a piece of information that needs to be recognized as an entity and make it a class; we can talk of an association between classes without any thought to how this will be realized.

As we go further into the design process and construct the sequence diagrams, we need to deal with the issue of these conceptual classes will be manifested in the software, i.e., we are now dealing with **software** classes. These can be implemented with typical programming languages, and we need to identify methods and parameters that will be involved. We have to finalize which entities will be individual classes, which ones will be merged, and how associations will be captured.

The last step is the **implementation** class, which is a class created using a specific programming language such as Java or C++. This step nails down all the remaining details: identification and implementation of helper methods, nitty-gritty details of using software libraries, names of fields and variables, etc.

The process of going from conceptual to implementation classes is a progression from an abstract system to a concrete one and, as we have seen, classes may be added or removed at each step. For instance, `Transaction` and `MemberIdServer` were added as software and implementation class respectively, whereas the conceptual class `Borrows` was dropped.

6.12.2 Building a Commercially Acceptable System

The reader having familiarity with software systems may be left with the feeling that our example is too much of a "toy" system, and our assumptions are too simplistic. This criticism is not unjustified, but should be tempered by the fact that our objective has been to present an example that can give the learner a "big-picture" of the entire design process, without letting the complexity overwhelm the beginner.

Non-Functional Requirements

A realistic system would have several non-functional requirements. Giving a fair treatment to these is beyond the scope of the book. Some issues like portability are automatically resolved since Java is interpreted and is thus platform independent. Response time (run-time performance) is a sticking point for object-oriented applications. We can examine this in a context where design choice affects performance, and this is addressed briefly in a later case-study.

Functional Requirements

It can be argued that for a system to be accepted commercially it must provide a sufficiently large set of services, and if our design methodologies are not adequate to handle that complexity, then they are of questionable value. We would like to point out the following:

- *Additional features can be easily added.* Some of these will be added in the next chapter. Our decision to exclude several such features has been made based on pedagogical considerations.
- *Allowing for variability among kinds of books/members.* This variability is typically incorporated by using inheritance. To explain the basic design process, inheritance is not essential. However using inheritance in design requires an understanding of several related issues, and we shall in fact present these issues and extend our library system in Chapter 9.
- *Having a more sophisticated interface.* Once again, we might want a system that allows members to login and perform operations through a GUI. This would only involve the interface and not the business logic. In Chapter 10, we shall see how a GUI can be modeled as a multi-panel interactive system, and how such features can be incorporated.
- *Allowing remote access.* Now-a-days most systems of this kind allow remote access to the server. Chapter 12 looks how such features can be introduced through the use of distributed objects.

It should be noted that in practice several of the non-functional requirements would actually be provided by a database. What we have done with the use case model, the sequence diagrams and the class diagrams is in fact an object-oriented schema, which can be used to create an application that runs on an object-oriented database system. Such a system would not only address issues of performance and portability but also take care of issues like persistence, which can be done more efficiently using relations rather than reading and writing the objects. Details of this are beyond the scope of this text.

Projects

1. Complete the designs for the case-study exercises from the previous chapter.

Exercises

1. Consider a situation where a library wants to add a feature that enables the librarian to print out a list of all the books that have been checked out at a given point in time. Construct a sequence diagram for this use case.
2. Explain the rationale for separating the user interface from the business logic.
3. Suppose the due-date for a book depends not only on the date the book is issued, but also on factors such as member type (assume that there are multiple types of membership), number of books already issued to the member, and any fines owed by the member. Which class should then be assigned the responsibility to compute the due date and why?

4. (Discussion) There is fairly tight coupling in our system between the `Book`, `Member` and `Hold` classes. Code in `Book` could inadvertently modify the fields of a `Member` object. One way to handle this is to replace the `Member` reference with just the member's ID. What changes would we have to make in the rest of the classes to accommodate this? What are the pros and cons of such an approach?

5. Continuing with the previous question, the `Hold` object stores references to the `Book` and `Member` objects. This may not be necessary. What specific information does `Book` (`Member`) require from `Hold`? Define an interface that contains the relevant methods to retrieve this information. What are the pros and cons of an implementation where `Hold` implements these interfaces, over the design presented in this chapter?

6. (Keeping mutables safe.) Suggest a simple scheme for creating a new class `SafeMember` that would allow us to export a reference to a `Member`. The classes outside the system should be unaware of this additional class, and access the reference like a reference to a `Member` object. However, the reference would not allow the integrity of the data to be compromised.

7. Without modifying any of the classes other than `Library`, write a method in `Library` that deletes all invalid holds for all members.

Index

Memory Management, 25