Chapter 7
Design and Implementation

Having done an analysis of the requirements, we proceed to the design stage. In this step, we use the class structure produced by the analysis to design a system that behaves in the manner specified by the model. The main UML tool that we employ here is the sequence diagram. In a sequence diagram, the designer specifies the details of how the behaviour specified in the model will be realised. This process requires the system's actions to be broken down into specific tasks, and the responsibility for these tasks to be assigned to the various players in the system. In the course of assigning these responsibilities, we determine the public methods of each class, and also describe the function performed by each method. Since the stage after design is implementation, which is coding, testing, and debugging, it is imperative that we have a full understanding of how the required functionality will be realised through code. The designer thus breaks down the system into smaller units and provides enough information so that a programmer can code and test each unit separately.

After the design is complete, we proceed to the implementation stage. As the coding is being done, the programmer should follow good coding and testing practices. We do not emphasise these principles here, since these are concepts common to any software design methodology. Our implementation will be done in Java. Any new language concepts that need elaboration are dealt with in the context where we employ them.

7.1 Design

During the design process, a number of questions need to be answered:

1. On what platform(s) (hardware and software) will the system run? For example, will the system be developed for just one platform, say, Windows running on 386-type processors? Or will we be developing for other platforms such as Unix?

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_7

2. What languages and programming paradigms will be used for implementation? Often, the choice of the language will be dictated by the expertise the company has. But sometimes the functionality will also heavily influence the choice of the language. For example, a business application may be developed using an object-oriented language such as Java or C++, but an artificial intelligence application may be programmed in LISP or Prolog. (In this chapter, we are assuming an object-oriented system.)
3. What user interfaces will the system provide? These include GUI screens, printouts, and other devices (for example, library cards).
4. What classes and interfaces need to be coded? What are their responsibilities?
5. How is data stored on a permanent basis? What medium will be used? What model will be used for data storage?
6. What happens if there is a failure? Ideally, we would like to prevent data loss and corruption. What mechanisms are needed for realising this?
7. Will the system use multiple computers? If so, what are the issues related to data and code distribution?
8. What kind of protection mechanisms will the system use?

Since our focus in this book is on software design and development using the object-oriented paradigm using the Java programming language, we will not be distracted by considerations of the exact platform on which the system will run. Our major focus throughout the book is the identification of the software structure: *the classes and interfaces that make up the system.* Although we discuss User Interface (UI) design and long-term storage issues, we do not address protection and recovery mechanisms since the development of these is largely orthogonal to the issues that we are attempting to address. In general, systems typically employ some combination of application software, firewalls, database management system support, manual procedures, etc., to provide the necessary mechanisms for protection, concurrency control and recovery. The choices made when designing solutions for these issues should have little or no impact on the design of the application software itself.

7.1.1 Major Subsystems

The first step in our design process is to identify the major subsystems. We can view the library system as composed of two major subsystems:

1. **Business logic** This part deals with input data processing, data creation, queries, and data updates. This module will also be responsible for interacting with external storage, storing and retrieving data.
2. **User interface** This subsystem interacts with the user, accepting and outputting information.

It is important to design the system such that the above parts are separated from each other so that they can be varied independently. That way, we get good cohesion within each module. Our focus in this chapter is mainly on the design and implementation of the business logic. At the end of the chapter, we put together a rudimentary UI. We also implement a mechanism for storing and retrieving data by interacting with external storage devices. While the UI and external storage management modules are adequate to carry out functional testing of our system, a more sophisticated design (and implementation) would be in order for a full-blown system.

7.1.2 Creating the Software Classes

The next step is to create the **software classes**. During the analysis, after defining the use case model, we came up with a set of conceptual classes and a conceptual class diagram for the entire system. As mentioned earlier, these come from a conceptual or essential perspective. The software classes are more 'concrete' in that they correspond to the software components that make up the system. In this phase there are two major activities.

1. Come up with a set of classes.
2. Assign responsibilities to the classes and determine the necessary data structures and methods.

In general, it is unlikely that we can come up with a design simply by doing these activities exactly once. Several iterations may be needed and classes may need to be added, split, combined, or eliminated.

As we are having just a rudimentary text-based interface, the UI subsystem will consist of a single class, aptly named `UserInterface`. The classes for the business logic module will be the ones instrumental in implementing the system requirements described in the use case model. In our analysis, we came up with a set of *conceptual* classes and relationships. It is, therefore, reasonable that as a 'first guess' for the required software classes for the business logic, we pick these conceptual classes. A closer scrutiny of these is now in order.

1. **Member and Book** These are central concepts. Each `Member` object comprises several attributes such as name and address, stays in the system for a long period of time and performs a number of useful functions. Books stay part of the library over a long time and we can do a number of useful actions on them. We need to instantiate books and members quite often. Clearly, both are classes that require representation in software.
2. **Library** Do we really need to make a class for this? To answer the question, let us ask what the real library—not a possible object—has. It keeps track of books and members. When a member thinks of a library, he/she thinks of borrowing and returning books, placing and removing holds, i.e., the *functionality* provided by the library. To model a library with software, we need to mimic this functionality, which we did by creating a use case model. The use case behaviour is what is exhibited by the UI, and to meet the required specifications, the UI must perform

some other computations that involve the module that implements the business logic. One of the important principles of object-oriented design is that every computation must be represented as an application of a method on a given object, which is then treated as the current object for the computation. All the computation required of the business logic module must be executed on some current object; that object is a `Library`. This requires that `Library` be a class in its own right, and the operations required of the business logic module correspond to the methods of this class.

Although details of its functionality remain to be determined by examining the use cases, with some thought we can come up with two important aspects of the `Library` class. As we have seen in Chap. 6, the `Library` instance must keep track of the members of the library as well as the books, which obviously imply maintenance of two collection objects. The functionality of these two collections is again to be determined, but it is likely that we need two different classes, `MemberList` and `Catalog`, which may be alike in certain respects.[1] These two collections last as long as the library itself, and we make modifications to them very frequently. The actions that we perform are not supported by programming languages although there may be some support in the associated packages such as the list classes in the Java Development Kit. All these would suggest that they be classes. However, we create them just once. As we know from Chap. 5, a class that has just one instance is called a *singleton.* Both `MemberList` and `Catalog` are singletons.

3. **Borrows** This class represents the one-to-many relationship between members and books. *In typical one-to-many relationships, the association class can be efficiently implemented as a part of the two classes at the two ends.* To verify this for our situation, for every pair of member *m* and book *b* such that *m* has borrowed *b*, the corresponding objects simply need to maintain a reference to each other. Since a member may borrow multiple books, this arrangement entails the maintenance of a list of `Book` objects in `Member`, but since there is only a single borrower for a book, each `Book` object needs to store a reference to only one instance of `Member`. Further examining the role played by the information in `Borrows`, we see that when a book is checked out, the due date can be stored in `Book`. In general, this means that all attributes that are unique to the relationship may be captured by storing information at the 'many' end of the relationship. When the book is returned, the references between the corresponding `Member` and `Book` objects as well as the due date stored in `Book` can be 'erased.'

   This arrangement efficiently supports queries arising in almost any situation: a user wanting to find out when her books are due, a staff member wanting to know the list of books borrowed by a member, or an anxious user asking the librarian when he can expect the book on which he placed a hold. In all these situations we have operations related to some `Member` and `Book` objects.

[1]Although we use the name `MemberList`, we do not imply that this class has to be organised as a list.

4. **Holds** Unlike `Borrows`, this class denotes a many-to-many relationship between the `Member` and `Book` classes. *In typical many-to-many relationships, implementation of the association without using an additional class is unlikely to be clean and efficient.* To attempt to do this without an additional class in the case of holds, we would need to maintain within each `Member` object references to all `Book` instances for which there is a hold, and keep 'reverse' references from the `Book` objects to the `Member` objects. This is, however, incomplete because we also need to maintain for each hold the number of days for which it is valid. But there is no satisfactory way of associating this attribute with the references. We could have queries like a user wanting a list of all of his holds that expire within 30 days. The reader can verify that implementations without involving an additional class will be messy and inefficient.

   It is, therefore, appropriate that we have a class for this relationship and make the `Hold` object accessible to the instances of `Member` and `Book`.

As we look at ways to implement the use cases, it often happens that we eliminate some of these classes, discover more, and determine the attributes and methods for all of the concrete classes.

7.1.3 Assigning Responsibilities to the Classes

Having decided on an adequate set of software classes, our next task is to assign responsibilities to these. Since the ultimate purpose of these classes is to enable the system to meet the responsibilities specified in the use case, we shall work with these system responsibilities to find the class responsibilities. The next step is, therefore, to spell out the details of how the system meets its responsibilities by devolving these down to the software classes, and the UML tool that we employ to describe this devolution is the sequence diagram.

It should be noted that the sequence diagram is only a concise, visual way of *representing* the devolution, and we need to make our design choices *before* we start drawing our arrows. For each system response listed in the right-hand column of the use case tables, we need to specify the following:

- The sequence in which the operations will occur.
- How each operation will be carried out.

For the first item above, we need a complete algorithm; the second item describes which classes will be involved in each step of the algorithm and how the classes will be engaged. In specifying the second item, we spell out detailed definitions of the classes: the methods that need to be invoked and the parameters that should be passed to these methods. The first item specifies what is done in each step; since each step is a method call, we are specifying what each method is supposed to accomplish. In the course of figuring out how the method computes what is needed, we make other design choices. In the end, all of these things come together to give us a complete system.

Register Member

The sequence diagram for the use case for registering a member is shown in Fig. 7.1. The clerk issues a request to the system to add a new member. The system responds by asking for the data about the new member. This interaction occurs between the library staff member and the `UserInterface` instance. The clerk enters the requested data, which the `UserInterface` accepts.

Obviously, at this stage the system has all the data it needs to create a new `Member` object. The role of the UI is to interact with the user and not to perform business logic. So if the UI were to assume all responsibility for creating a `Member` object and adding that object to the `Library` instance, the consequence will be unnecessary and unwanted coupling between the business logic module and the UI class. We would like to retain the ability to develop the UI knowing as little as possible about the application classes. For this purpose, it is ideal to have a method, viz., `addMember`, within `Library` to perform the task of creating a `Member` and storing it in `MemberList`. All that `UserInterface` needs to do is pass the three pieces of information—name, address, and phone number of the applicant—as parameters to the `addMember` method, which then assumes full responsibility for creating and adding the new member.

Let us see details of the `addMember` method. The algorithm here consists of three steps:

1. Create a `Member` object.
2. Add the `Member` object to the list of members.
3. Return the result of the operation.

To carry out the first two steps, we have two options:

Option 1 Invoke the `Member` constructor from within the `addMember` method of `Library`. The constructor returns a reference to the `Member` object and an operation, `insertMember`, is invoked on `MemberList` to add the new member.

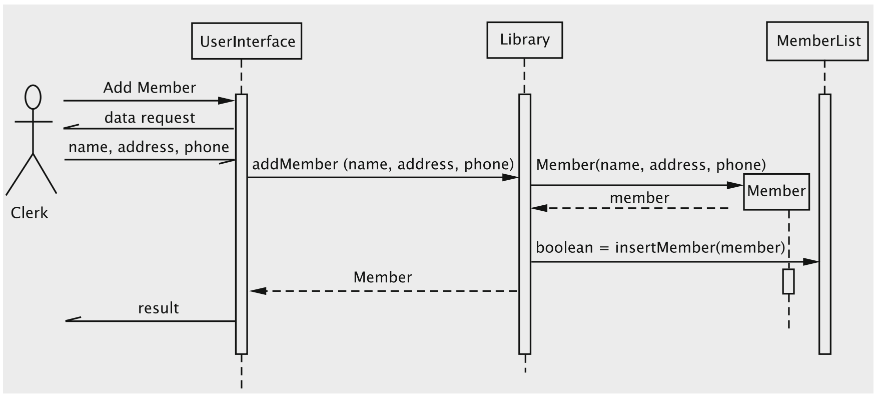


Fig. 7.1 Sequence diagram for adding a new member

Option 2 Invoke an `addNewMember` method on `MemberList` and pass as parameters all the data about the new member. `MemberList` creates the `Member` object and adds it to the collection.

Let us examine what the purpose of the `MemberList` class is: *to serve as a container for storing a large number of members, adding new ones, removing existing ones, and performing search operations.* The container should not, therefore, concern itself with details of a member, especially, its attributes. If we choose Option 2, `addNewMember` must take in as parameters the details of a member (name, address, and phone) so that it can call the constructor of the `Member` class. This introduces unnecessary coupling between `MemberList` and `Member`. As a result, if changes are later made to the `Member` constructor, these will also affect `MemberList`, even though the intended functions of `MemberList` do not warrant these changes.

Therefore, we prefer Option 1 to implement the `addMember` method.

The last step is to return the result so that `UserInterface` can adequately inform the actor about the success of the operation. The requirements for this are spelled out in Step 5 in Table 6.1, which reads: '(The system) informs the clerk if the member was added and outputs the member's name, address, phone, and id.' This can be achieved if `Library` returns a reference to the `Member` object that was created. If the reference is `null`, the UI informs the actor that the operation was unsuccessful; otherwise, the necessary information is accessed from the `Member` object and reported.

Add Books

The next sequence diagram that we show is for the Add Books use case. This use case allows the insertion of an arbitrary number of books into the system. In this case, when the request is made by the actor, the system enters a loop. Since the loop involves interacting repeatedly with the actor, the loop control mechanism is in the UI itself. The first operation is to get the data about the book to be added. The algorithm here consists of the following steps: (i) create a `Book` object, (ii) add the `Book` object to the catalog and (iii) return the result of the operation. This is handled in a manner similar to the previous use case.

The UI returns the result and continues until the actor indicates an exit. This repetition is shown diagrammatically by a special rectangle that is marked `loop`. All activities within the rectangle are repeated until the clerk indicates that there are no more books to be entered (Fig. 7.2).

In the last two sequence diagrams, note that `Library`, `MemberList` and `Catalog` are in the top row. Placing the entity in the top row indicates that it is in existence at the beginning of the process. This contrasts with the entities corresponding to `Member` and `Book`, which do not exist at the start, but are created by invoking constructors. This is indicated by placing the boxes labelled 'Member' and 'Book' respectively at the end of the arrow representing the call to the constructor. This box is at a lower level, to signify the later point in time when the entity comes into existence.

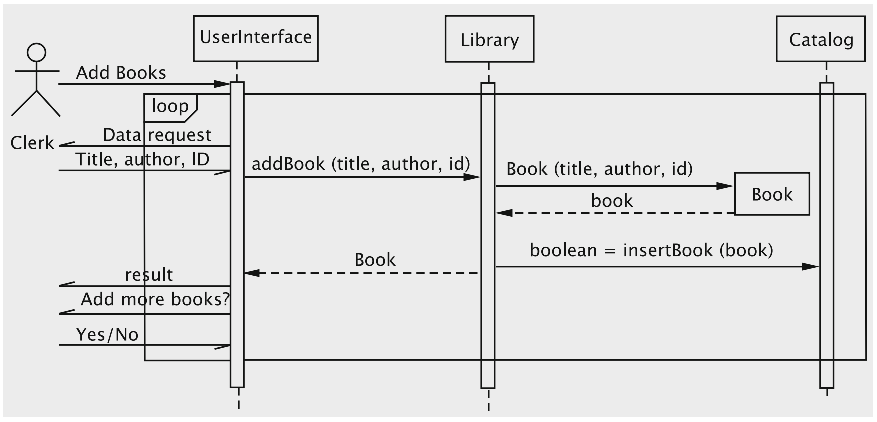


Fig. 7.2 Sequence diagram for adding books

Issue Books

The sequence diagram for the Issue Books use case is given next (Fig. 7.3). When a book is to be checked out, the clerk interacts with the UI to input the user's ID. The system has to first check the validity of the user. This is accomplished by invoking the method `searchMembership` on the Library.

Two options suggest themselves for implementing the search:

- **Option 1** Get an enumeration of all `Member` objects from `MemberList`, get the ID from each and compare with the target ID.
- **Option 2** Delegate the entire responsibility to `MemberList`.

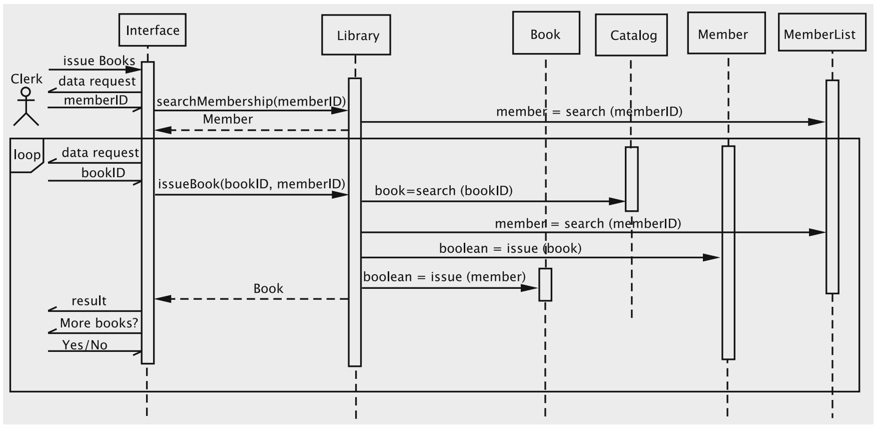


Fig. 7.3 Sequence diagram for issuing books

Option 1 places too much detail of the implementation in Library, which is undesirable. Option 2 is more attractive because search is a natural operation that is performed on a container. The flip-side with the second option is that in a naive implementation, MemberList will now become aware of implementation details of Member (that memberID is a unique identifier, etc) causing some unwanted coupling between (Member) and the container class (MemberList). This coupling is not a serious concern because it can be removed using generics as we shall see in the next chapter.

UserInterface receives a reference to the Member object from Library and then queries the actor for the ID of the book. In Library, we are providing a method that issues a single book to a user. UserInterface invokes this method repeatedly in order to issue several books to the user, each time passing the member's ID and the book's ID as parameters. Once again, searching for the Book object is delegated to Catalog. Next, the Book and Member objects are updated to indicate that the book is checked out to the member (and that the member is in possession of the book). Notice that the Library class orchestrates the whole show and also acts as a go-between for all operations that the UserInterface requests from the business logic module.

It may be tempting for a beginner to directly access the Member object from UserInterface, pass the book's ID as a parameter and thereby initiate the issuing process. To understand why this is a bad idea, imagine that at later time, the business logic associated with issuing a book changes. This change could potentially force changes in the UserInterface class, i.e., *classes outside the core library subsystem are affected.* As a general rule, we avoid exposing details of business logic implementation to the UI. Likewise, one may be tempted to send bookID to Member and handle all the details within Member; this would mean that Member searches Catalog, creating a dependency between these classes. These other approaches, therefore, expose system details to the UI and create tight coupling between the classes, thus hurting reuse.

Another question we need to address is this: *Where should the responsibility for generating the due-date lie?* In our simple system, the due-date is simply one month from the date of issue, and it is not determined by other factors such as member privileges. Consequently computing the due-date is a simple operation that can be done in any of the objects, but since we are storing the due-date as a field in Book, we will assign this responsibility to Book.

As before, we must decide the return type of the method issueBook. The use case requires of the system that it generates a due-date. The system displays the book title and due-date and asks if there are any more books. This can be easily done by returning a reference to the Book object. The operation is reported as unsuccessful if the reference is null.

Return Books

The Return Book use case is implemented in Fig. 7.4 as a sequence diagram. For each book returned, the returnBook method of the Library class obtains the

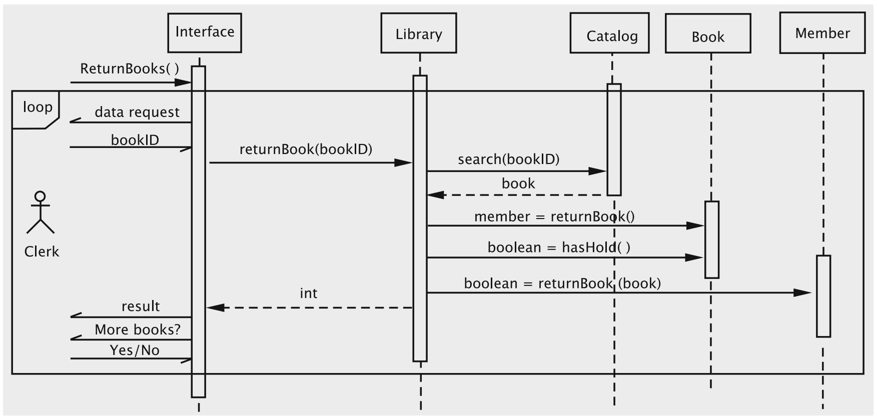


Fig. 7.4 Sequence diagram for returning books

corresponding `Book` object from `Catalog`. The `returnBook` method is invoked using this `Book` object, and this method returns the `Member` object corresponding to the member who had borrowed the book. The `returnBook` method of the `Member` object is now called to record that the book has been returned. This operation has three possible outcomes that the use case requires the system to distinguish (Step 5 in Table 6.5):

1. *The book's ID was invalid*, which would result in the operation being unsuccessful;
2. *the operation was successful*;
3. *The operation was successful and there is a hold on the book*. The value returned by `returnBook` must enable `UserInterface` to make the distinction between these. This is done by having `Library` return a result code, which could simply be one of three suitably named integer constants.

Remove Books

The diagram in Fig. 7.5 shows the sequence diagram for removing books from the collection. Here, as discussed in the use case, we remove only those books that are not checked out and do not have a hold. This logic for deciding whether the book is removable is in the `removeBook` method in `Library`. This method checks each property of the book in question and if all properties are satisfied, the `remove` method in `Catalog` is invoked, which then removes the book. The square brackets before the invocation of `remove` contain the condition 'can delete book', indicating that the book is deleted only if this condition is met. `Library` returns a specific code for each possible outcome, which `UserInterface` translates into an appropriate message.

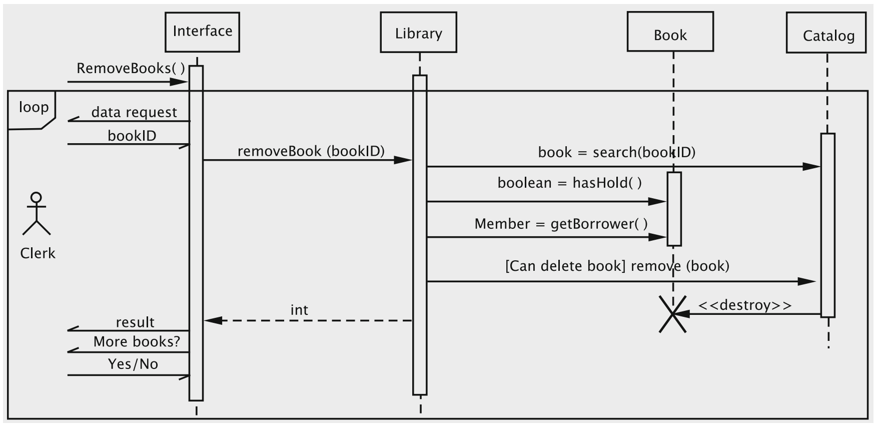


Fig. 7.5 Sequence diagram for removing books

Member Transactions

Following the earlier examples, it is no surprise that the end-user (clerk) interacts with the `Library` class to print out the transactions of a given member. From the descriptions given so far, the reader should have gained enough skill to interpret most of the sequence diagram in Fig. 7.6.

The `Member` class stores the necessary information about the transactions, but the UI would be the one to decide the format. It would, therefore, be desirable to provide the information to the UI as a collection of objects, each object containing the information about a particular transaction. This can be done by defining a class `Transaction` that stores the type of transaction (issue, return, place, or remove hold), the date, and the title of the book involved. `Member` stores a list of transactions, and the method `getTransactions` returns an enumeration (`Iterator`) of the

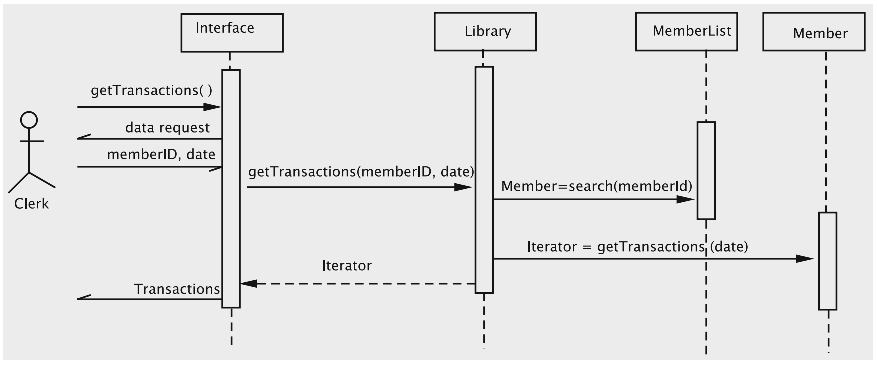


Fig. 7.6 Sequence diagram for printing a member's transactions

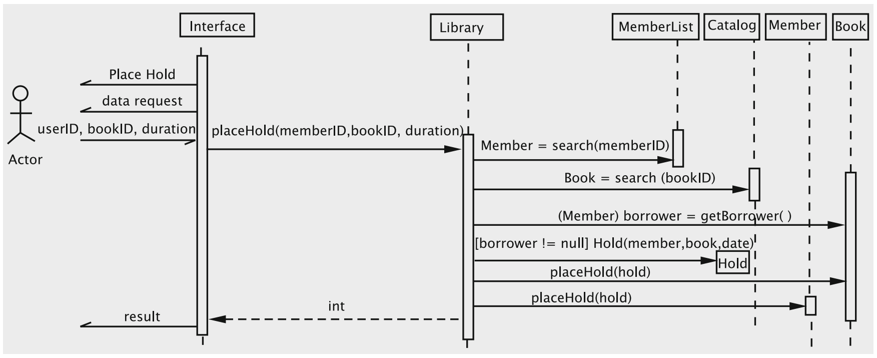


Fig. 7.7 Sequence diagram for placing a hold

`Transaction` objects whose date matches the one specified. `Library` returns this to the UI, which extracts and displays the needed information.

Place Hold

As discussed earlier, we create a separate `Hold` class for representing the holds placed by members. Each `Hold` object stores references to a `Member` object and a `Book` object, and the date when the hold expires (see Fig. 7.7).

When a clerk issues request to the library to place a hold on behalf of a member for a certain book, the `Library` object itself creates an instance of `Hold` and makes both the `Book` and `Member` instances involved to store references to it. The UI is informed of the outcome by a result code.

It is instructive to consider what alternate implementations may be used for storing the holds. One possibility is that both `Book` and `Member` create their own individualised `Hold` objects, with a `BookHold` class storing the date and a reference to `Member` and `MemberHold` storing the date and a reference to `Book`. Such a solution is less preferable because it creates additional classes, and if not carefully implemented, could also lead to inconsistency due to multiple copies of the date.

Cohesion and Coupling
In deciding the issues of how specific details of the implementation are carried out, we have to keep in mind the twin issues of cohesion and coupling. We must have *good cohesion* among all the entities that are grouped together or placed within a subsystem. Simultaneously, entities within the group must be *loosely coupled.*

In our example, when issuing a book, we have chosen to implement the system so that the Library calls the issue methods of `Book` and `Member`. Contrast this with a situation where `Book` calls the `issue` method of `Member`; in such a situation, the code in `Book` depends on the method names of `Member`, which causes tight coupling between these two classes. Instead, we have chosen a solution where each of these classes is somewhat tightly coupled with `Library`, but there is very loose coupling between any other pair of classes. This means that when the system has to adapt to changes in any class, this can be done by modifying `Library` only. `Library`, therefore, serves as ‘glue’ that holds the system together and simultaneously acts an interlocutor between the entities in the library system.

We have also consciously chosen to separate the design of the business module from the UI through which the actors will interact with the system. This is to ensure good cohesion within the system’s ‘back-end’.

A related question that we face at a lower level is that of how responsibilities are being assigned. We ask this question when a class is being designed. Responsibilities are assigned to classes based on the fields that the class has. These responsibilities turn into the methods of the class. The principle that we are following here can be tersely summarised in an Italian saying (attributed to Bertrand Meyer), *‘The shoemaker must not look past the sandal’*. In other words, the only responsibilities assigned to an object/class should be the ones that are relevant to the data abstraction that the class represents. This, in turn, ensures that we avoid unnecessary coupling between classes.

Process Holds

The input here is only the ID for the book, from which we get the next hold that has not expired. In this process, the book would quite likely find some holds that are not valid. These holds should obviously be removed from the system and the responsibility for this clean up is assigned to the `getNextHold()` method in `Book`. The Library gets a reference to the `Member` object from `Hold` (see Fig. 7.8) and returns this to the UI.

Remove Hold

The sequence diagram is given in Fig. 7.9. A request is issued to `Library` via the method `removeHold`. `Library` retrieves the corresponding `Member` and `Book` objects using `MemberList` and `Catalog` and then invokes the `removeHold` method on these objects to delete their references to the `Hold` object.

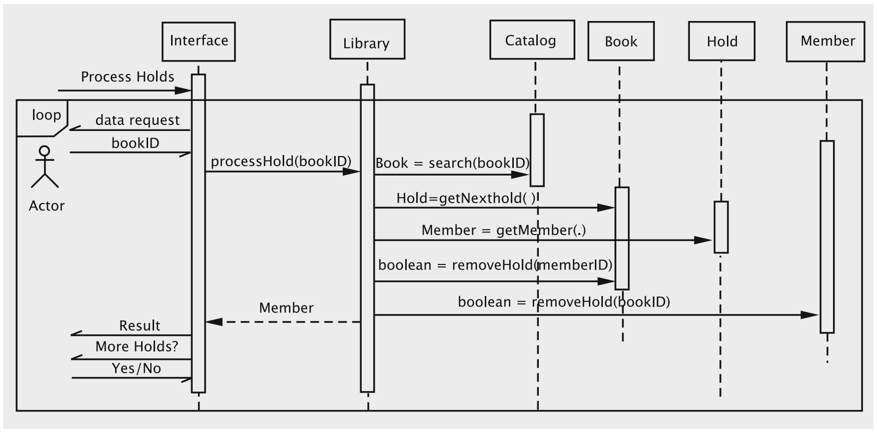


Fig. 7.8 Sequence diagram for processing holds

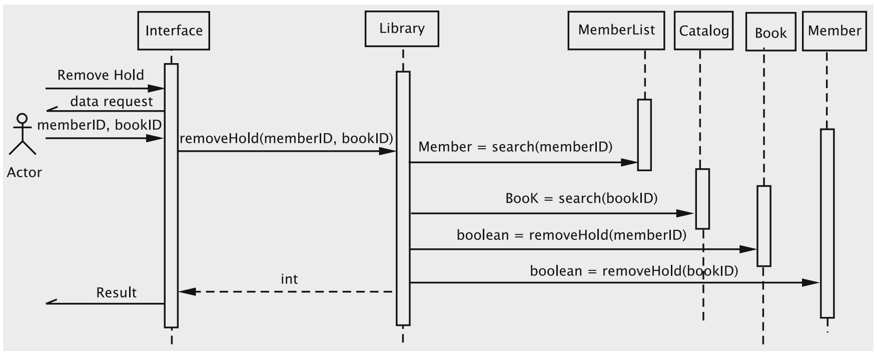


Fig. 7.9 Sequence diagram for removing a hold

Renew Books

Figure 7.10 details the implementation for renewing books. This process involves interactively updating the information on several members of a collection. We can accomplish this by allowing `UserInterface` to get an enumeration (`Iterator`) of the items in the collection, getting responses on each from the user and invoking the methods on the library to update the information.

The `Library` class thus provides a set of methods for the UI and serves as a single point of entry to and exit from the business logic module. This is a useful approach in many situations, so it is given a special name: **Facade.** All updates are done by invoking methods on the facade and not by directly manipulating the objects in the enumeration. Such direct manipulation would place some of the business logic in the UI and also hurt reuse as we have observed earlier.

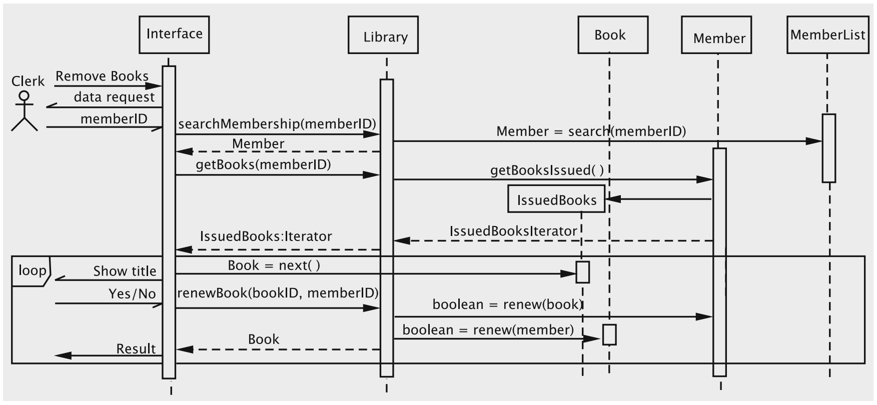


Fig. 7.10 Sequence diagram for renewing books

7.1.4 Class Diagrams

Hopefully, at this stage, we have come up with all the software classes. To review:

1. `Library`
2. `MemberList`
3. `Catalog`
4. `Member`
5. `Book`
6. `Hold`
7. `Transaction`

The relationships between these classes is shown in Fig. 7.11. Note that `Hold` is not shown as an association class, but an independent class that connects `Member` and `Book`. The new class `Transaction` is added to record transactions; this has a dependency on `Book` since it stores the title of the book.

By inspecting the sequence diagrams, we can collect the methods of each of these classes, and draw a class diagram for each. In specifying the types of attributes, we have to make language-specific choices; in the process of doing this we transition from the software classes to the **implementation** classes.

We first examine the methods and then arrive at the attributes by examining the methods.

Class Diagram for Library

The methods are simply a collection of methods with their parameters as given in the sequence diagrams. However, we have specified their return types, which were not clearly specified in the sequence diagrams. Whenever something is added to the system such as a member or a book or a hold, some information about the added object is returned, so that the clerk can verify that the data was correctly recorded.

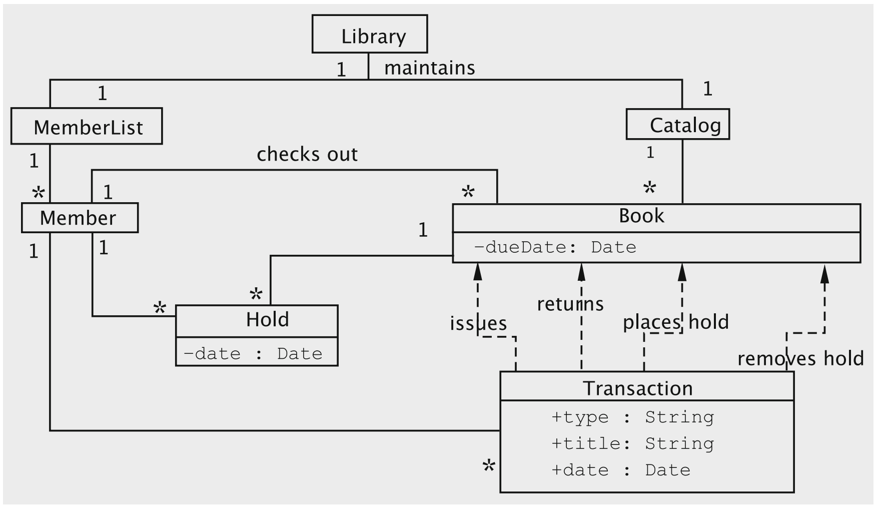
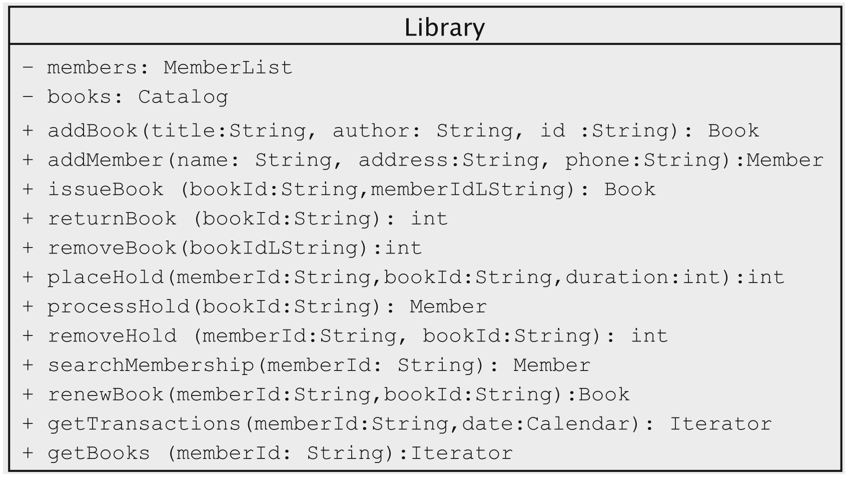


Fig. 7.11 Relationships between the software classes



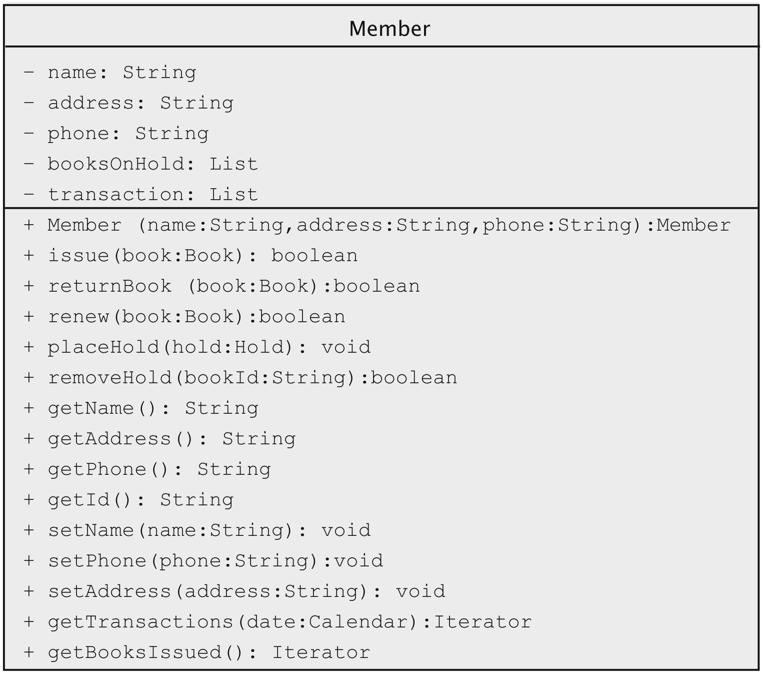
```
Library
- members: MemberList
- books: Catalog
+ addBook(title:String, author: String, id :String): Book
+ addMember(name: String, address:String, phone:String):Member
+ issueBook (bookId:String,memberIdLString): Book
+ returnBook (bookId:String): int
+ removeBook(bookIdLString):int
+ placeHold(memberId:String,bookId:String,duration:int):int
+ processHold(bookId:String): Member
+ removeHold (memberId:String, bookId:String): int
+ searchMembership(memberId: String): Member
+ renewBook(memberId:String,bookId:String):Book
+ getTransactions(memberId:String,date:Calendar): Iterator
+ getBooks (memberId: String):Iterator
```

Fig. 7.12 Class diagram for Library

We have already seen that the class must maintain references to `Catalog` and `MemberList`. See Fig. 7.12 for the class diagram.

Class Diagram for Member

Once again, we get our methods and attributes by examining the sequence diagrams. In our design, we make the `Member` class generate the member ID. We need a mechanism to ensure that no two members get the same ID, i.e., there has to be some



```
Member
- name: String
- address: String
- phone: String
- booksOnHold: List
- transaction: List
+ Member (name:String,address:String,phone:String):Member
+ issue(book:Book): boolean
+ returnBook (book:Book):boolean
+ renew(book:Book):boolean
+ placeHold(hold:Hold): void
+ removeHold(bookId:String):boolean
+ getName(): String
+ getAddress(): String
+ getPhone(): String
+ getId(): String
+ setName(name:String): void
+ setPhone(phone:String):void
+ setAddress(address:String): void
+ getTransactions(date:Calendar):Iterator
+ getBooksIssued(): Iterator
```

Fig. 7.13 Class diagram for Member

central place where we keep track of how ids are generated. It would be tempting to do this in the `Library` class, but the right solution would be to make it a static method in the `Member` class. This gives us decentralised control and places responsibilities close to the data. The class diagram is given in Fig. 7.13.

Class Diagram for Book

The approach to developing the class diagram for `Book` parallels that of the approach for the `Member` class. As in the other cases, we now add the attributes. However, there are no setters for the `Book` class because we don't expect to change anything in a `Book` object (see Fig. 7.14).

Class Diagram for Catalog

Typical operations on a list would be add, remove, and search for objects. Proceeding as in the case for the `Library` class, we obtain the methods shown in Fig. 7.15.

The only attribute that we come up with is a `List` object that stores `Book` objects. The reader will also notice the method `getBooks`, whose return type is `Iterator`. This enables the `Library` to get an enumeration of all the books so that any specialised operations that have to be applied to the collection are facilitated.

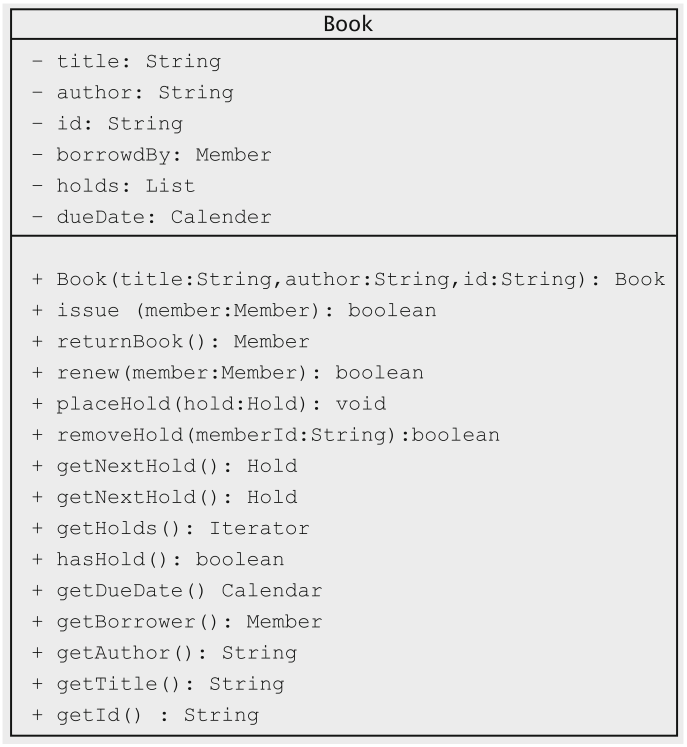
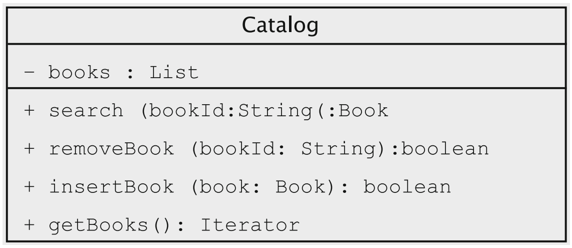


Fig. 7.14 Class diagram for the Book class

Fig. 7.15 Class diagram for the Catalog class


Catalog
- books : List
+ search (bookId:String(:Book
+ removeBook (bookId: String):boolean
+ insertBook (book: Book): boolean
+ getBooks(): Iterator

Class Diagram for MemberList

The derivation of this is fairly straightforward after developing the `Catalog` class and is shown in Fig. 7.16. Since we never asked for the functionality of removing a member, there is no such method in the class. We need an attribute of type `List` to store the members.

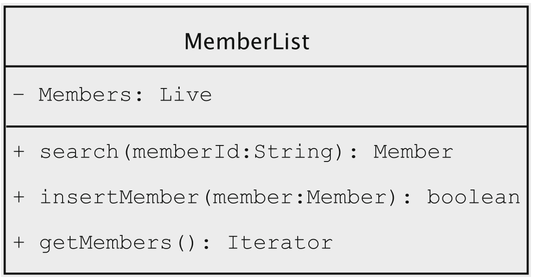

MemberList
- Members: Live
+ search(memberId:String): Member
+ insertMember(member:Member): boolean
+ getMembers(): Iterator

Fig. 7.16 Class diagram for the MemberList class

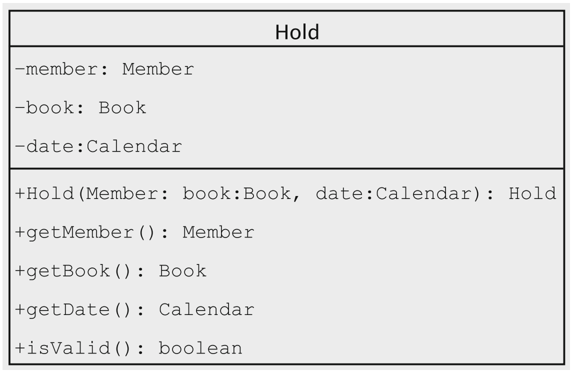


Fig. 7.17 Class diagram for Hold

Class Diagram for Hold

Besides the accessors, `getMember`, `getBook`, and `getDate`, the class diagram for `Hold` (Fig. 7.17) shows the `isValid` method, which checks whether a certain hold is still valid.

Exporting and Importing Objects
The classes that we have implemented for the business logic form an object-oriented system, which can be accessed and modified through the methods of `Library`. When dealing with object-oriented systems, one must keep in mind that there are often several references to one object, stored in multiple locations. For instance, a reference to every `Member` object is stored in `MemberList`, but when the member checks out a book, the `Book` object also holds a reference. In a lot of situations it is convenient to have a query return a reference to an object. This multiplicity of references means that we need to observe some caveats to ensure that data integrity is not compromised. In the context of importing and exporting references through the facade, the following deserve mention.

- *Do not export references to mutable objects.* All the objects that we are creating in the library system are **mutable**, i.e., the values stored in their fields can be changed. Within the system, objects store references to each other (`Book` and `Member` in our case study) and this is unavoidable. Our worries start with situations like the implementation we have for Issue Books, in which a reference to a `Member` object is being returned to `UserInterface`. Here a reference to a mutable object is being exported from the library subsystem, and in general we do not have any control over how this reference could be (mis)used. In a system that has to be deployed for widespread use, this is a serious matter, and some mechanism must be employed to make sure that the security and integrity of the system are not compromised. Several mechanisms have been proposed and we can create simple ones by defining additional classes (see exercises).
- *The system must not import a reference to an* **internal** *object.* Objects of type `Book` and `Member` belong to the system and their methods are invoked to perform various operations. To ensure integrity, it is essential that these methods behave exactly in the expected manner, i.e., *the objects involved belong to the classes we have defined and not any malicious descendants.* This means that our library system cannot accept as a parameter a reference to a `Book` object. This can be seen in the sequence diagram for Renew Books. The UI has the references to the `Book` and `Member` objects, but the Library does not accept these as parameters for `renewBook`. Working with the ID may mean an additional overhead to search for the object reference using the ID, but it certifies that when the `renew` methods are invoked, these are on objects that belong to the system.

Class Diagram for Transaction

The class diagram is shown in Fig. 7.18. Note that we have to store the date for each transaction, i.e., we need to choose an appropriate type for this attribute. Java's `util` package has a class `Calendar` that provides the needed functionality.

7.1.5 User Interface

As discussed earlier, our UI provides a menu with the following options:

1 Add a member
2 Add books
3 Issue books
4 Return books
5 Renew books

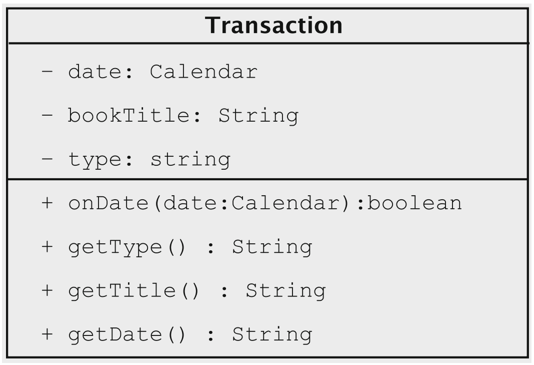


Fig. 7.18 Class diagram for Transaction

6 Remove books
7 Place a hold on a book
8 Remove a hold on a book
9 Process holds
10 Print a member's transactions on a given date
11 Save data for long-term storage
12 Retrieve data from storage
0 Exit
13 Help

Initially, the system will display a menu. The user can enter a number from 0 through 13 indicating the operation. (The options 0 and 13 will be used to exit the system and display the help screen respectively.) Parameters required for the operation will be prompted. The result of the operation is then displayed.

All input/output will be via simple text interface.

7.1.6 Data Storage

Ultimately, most applications will need to store data on a long-term basis. In a full-blown system, data is usually stored in a database, and this data is managed by a database management system. To avoid digressing, however, we will adopt a simple approach to store data on a long-term basis. Recall that we had decided to include the following commands in our UI.

1. A command to save the data on a long-term basis.
2. A command to load data from a long-term storage device.

When the first command is executed, we will copy all of the data onto secondary storage. Similarly, when the second command is executed, the data stored on the storage device is copied to recreate the object.

7.2 Implementing Our Design

In this phase, we code, test, and debug the classes that implement the business logic (`Library`, `Book`, etc.) and `UserInterface`. An important issue in the implementation is the communication via the return values between the different classes: in particular between `Library` and `UserInterface`; `Library` has several methods that return `int` values, and these values must be interpreted by the UI.[2] A separate named constant is declared for each of these outcomes as shown below.

```
public static final int BOOK_NOT_FOUND  = 1;
public static final int BOOK_NOT_ISSUED  = 2;
// etc.
```

These are declared in `Library`.

7.2.1 Setting Up the Interface

We are now ready to complete our development by writing the code. The main program resides in the class `UserInterface`. When the main program is executed, an instance of the UserInterface is created (a singleton).

```
public static void main(String[] s) {
  UserInterface.instance().process();
}

public static UserInterface instance() {
  if (userInterface == null) {
    return userInterface = new UserInterface();
  } else {
    return userInterface;
  }
}
```

The private constructor checks whether a serialized version of the `Library` object exists. (We assume that it is stored in a file called ‘LibraryData’.) The `File` class in Java is a convenient mechanism to check the existence of files. The user is given an option to retrieve any serialized version of the `Library` object. (We will explain later how the problem of safely combining serialization and singletons is tackled.) In any case, `UserInterface` gets an instance of `Library`.

```
private UserInterface() {
  File file = new File("LibraryData");
  if (file.exists() && file.canRead()) {
    if (yesOrNo("Saved data exists. Use it?")) {
      retrieve();
    }
```

[2] The implementation has additional methods to aid testing: methods to display books, members, etc. We do not discuss these methods here.

```
    }
    library = Library.instance();
  }
```

Following this, the `process` method of `UserInterface` is executed, which initialises a loop that provides the user with a list of options. This code snippet is given below.

```
  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`, `getNumber`, `getDate`, and `getCommand` for reading the user input. 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 `yesOrNo` with a `String` parameter to prompt the user. We can now follow our sequence diagrams to implement the methods. Some of these are explained below.

7.2.2 Adding New Books

The `addBooks` method in `UserInterface` is shown below:

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

The loop is set up in `UserInterface`, all the input is collected, and the `addBook` method in `Library` is invoked. Following the sequence diagram, this method is implemented in `Library` as follows:

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

In the above code, the constructor for `Book` is invoked and the new book is added to the catalog. The `Catalog` (which is also a singleton) is an adapter for the `LinkedList` class, so all it does is to invoke the `add` method in Java's `LinkedList` class, as shown below.

```
public class Catalog {
  private List books = new LinkedList();
  // some code not shown
  public boolean insertBook(Book book) {
    return books.add(book);
  }
}
```

7.2.3 Issuing Books

Once again, `UserInterface` gets the member's ID and sets up the loop. Here, `UserInterface` remembers the member's ID throughout the process. The `issue Book` method of `Library` is repeatedly invoked and the response to the actor is generated based on the value returned by each invocation.

```
public void issueBooks() {
  Book result;
  String memberID = getToken("Enter member id");
  if (library.searchMembership(memberID) == null) {
    System.out.println("No such member");
    return;
  }
  do {
    String bookID = getToken("Enter book id");
    result = library.issueBook(memberID, bookID);
    if (result != null){
      System.out.println(result.getTitle()+ "    " +  result.getDueDate());
    } else {
        System.out.println("Book could not be issued");
    }
    if (!yesOrNo("Issue more books?")) {
      break;
    }
  } while (true);
}
```

The `issueBook` method in `Library` does the necessary processing and returns a reference to the issued book.

```
  public Book issueBook(String memberId, String bookId) {
    Book book = catalog.search(bookId);
    if (book == null) {
      return(null);
    }
    if (book.getBorrower() != null) {
      return(null);
    }
    Member member = memberList.search(memberId);
    if (member == null) {
      return(null);
    }
    if (!(book.issue(member) && member.issue(book))) {
      return null;
    }
    return(book);
  }
```

The `issue` methods in `Book` and `Member` record the fact that the book is being issued. The method in `Book` generates a due date for our simple library by adding one month to the date of issue.

```
  public boolean issue(Member member) {
    borrowedBy = member;
    dueDate = new GregorianCalendar();
    dueDate.setTimeInMillis(System.currentTimeMillis());
    dueDate.add(Calendar.MONTH, 1);
    return true;
  }
```

`Member` is also keeping track of all the transactions (issues and returns) that the member has completed. This is done by defining the class `Transaction`.

```
  import java.util.*;
  import java.io.*;
  public class Transaction implements Serializable {
    private String type;
    private String title;
    private Calendar date;
    public Transaction (String type, String title) {
      this.type = type;
      this.title = title;
      date = new GregorianCalendar();
      date.setTimeInMillis(System.currentTimeMillis());
    }
    public boolean onDate(Calendar date) {
      return ((date.get(Calendar.YEAR) == this.date.get(Calendar.YEAR)) &&
              (date.get(Calendar.MONTH) == this.date.get(Calendar.MONTH)) &&
              (date.get(Calendar.DATE) == this.date.get(Calendar.DATE)));
    }
```

```
      public String getType() {
        return type;
      }
      public String getTitle() {
        return title;
      }
      public String getDate() {
        return date.get(Calendar.MONTH) + "/" + date.get(Calendar.DATE) + "/"
                                              + date.get(Calendar.YEAR);
      }
      public String toString(){
        return (type + "   " + title);
      }
    }
```

With each book issued, a record is created and added to the list of transactions, as shown in the following code snippet from `Member`.

```
private List booksBorrowed = new LinkedList();
private List booksOnHold = new LinkedList();
private List transactions = new LinkedList();

public boolean issue(Book book) {
  if (booksBorrowed.add(book)){
    transactions.add(new Transaction ("Book issued ", book.getTitle()));
    return true;
  }
  return false;
}
```

7.2.4 Printing Transactions

`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. The method `getTransactions` in `Member` filters the transactions based on the date and returns an `Iterator` of the filtered collection.

```
public Iterator getTransactions(Calendar date) {
  List result = new LinkedList();
  for (Iterator iterator = transactions.iterator(); iterator.hasNext(); ) {
    Transaction transaction = (Transaction) iterator.next();
    if (transaction.onDate(date)) {
      result.add(transaction);
    }
  }
  return (result.iterator());
}
```

`Library` returns `null` when the member is not in `MemberList`; otherwise an iterator to the filtered collection is returned. The UI extracts the necessary information and displays it in the preferred format.

```
public void getTransactions() {
  Iterator result;
  String memberID = getToken("Enter member id");
```

```
      Calendar date  = getDate("Please enter the date for which you want " +
                             "records as mm/dd/yy");
      result = library.getTransactions(memberID,date);
      if (result == null) {
        System.out.println("Invalid Member ID");
      } else {
        while(result.hasNext()) {
          Transaction transaction = (Transaction) result.next();
          System.out.println(transaction.getType() + "    "    +
                             transaction.getTitle() + "\n");
        }
        System.out.println("\n  There are no more transactions \n" );
      }
    }
```

7.2.5 Placing and Processing Holds

When placing a hold, the information about the hold is passed to `Library`, which checks the validity of the information and creates a `Hold` object. In our implementation, the `Member` and `Book` objects store the reference to the `Hold` object. The `placeHold` method in both `Book` and `Member` simply appends the new hold to the list. (The code for `Book` is shown below.)

```
    private List holds = new LinkedList();
    public void placeHold(Hold hold) {
      holds.add(hold);
    }
```

One problem with this simple solution is that unwanted holds can stay in the system forever. To prevent this, we may want to delete all invalid holds periodically, perhaps just before the system is saved to disk. This is left as an exercise.

The list `booksOnHold` in `Member` keeps a collection of all the active holds the member has placed. In the `Member` class we also generate a transaction whenever a hold is placed.

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

To process a hold, `Library` invokes the `getNextHold` method in `Book`, which returns the first valid hold.

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

The `Hold` class is shown below. There are no modifiers for the attributes, since a hold cannot be changed once it has been placed. The method `isValid()` checks if the hold is still valid.

```
public class Hold implements Serializable {
  private Book book;
  private Member member;
  private Calendar date;
  public Hold(Member member, Book book, int duration) {
    this.book = book;
    this.member = member;
    date = new GregorianCalendar();
    date.setTimeInMillis(System.currentTimeMillis());
    date.add(Calendar.DATE, duration);
  }
  public Member getMember() {
    return member;
  }
  public Book getBook() {
    return book;
  }
  public Calendar getDate() {
    return date;
  }
  public boolean isValid() {
    return (System.currentTimeMillis() < date.getTimeInMillis());
  }
}
```

Once the reference to the `Hold` object has been found in the Book, the hold is removed from the book and from the corresponding member as well. The book's ID is passed to the `removeHold` method in `Member`, which is shown below.

```
public boolean removeHold(String bookId) {
  boolean removed = false;
  for (ListIterator iterator = booksOnHold.listIterator();
                               iterator.hasNext(); ) {
    Hold hold = (Hold) iterator.next();
    String id = hold.getBook().getId();
    if (id.equals(bookId)) {
      transactions.add(new Transaction ("Hold Removed ",
                                    hold.getBook().getTitle()));
      removed = true;
      iterator.remove();
    }
  }
  return removed;
}
```

As is evident from the pieces of code shown above, the translation from sequence diagrams to code is a fairly straightforward task. This is what one should expect. In fact, the sequence diagrams in software design perform a role analogous to blueprint in engineering design. Once the sequence diagrams are complete, there is very little left to explain or discuss. If it were otherwise, that would reflect poorly on the process being followed.

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++ allow and require 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 will not 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 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.

7.2.6 Storing and Retrieving the Library Object

Java Serialization

Our approach to long-term storage of the library data uses the Java *serialization* mechanism. In Chap. 4 we saw that the methods readObject() and writeObject (Object) in ObjectInputStream and ObjectOutputStream respectively can be used to read and write objects and that this can be easily done for simple cases by having the corresponding class implement the Serializable interface.

In our current example, Book and Hold can be serialized by simply declaring them to be Serializable. This is because they contain instance fields each of which is defined to be Serializable. (The reader can verify this by examining the documentation of the Java classes we use, such as GregorianCalendar and LinkedList and the definition of Book and Hold.) Member, MemberList, Catalog, and Library need more work because they all have static fields in them. The default serialization mechanism in Java does not store static fields.

Storing the Data

What should we do to store the entire data? To answer this question, observe that Library has references to both the Catalog and MemberList objects, which, in turn, have references to the Book and Member objects respectively; the Hold objects are referred to by the Book objects and the Member objects. Thus, if we simply store the Library object, all of the data will be stored. As in our earlier use cases, we would like to keep these details out of the UI, and so UserInterface has a save method that simply invokes a save method on the Library object.

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

The `save` method in `Library` could simply write the `Library` object to a file named 'LibraryData' and return `true` if nothing goes wrong, as shown below.

```
FileOutputStream file = new FileOutputStream("LibraryData");
ObjectOutputStream output = new ObjectOutputStream(file);
output.writeObject(library);
return true;
```

Likewise, we could read the stored data with a method that reverses the above process by opening `LibraryData` and reading the contents into `library`. This simple approach may well suffice for a small 'in-house' system, which is used and maintained by a dedicated programmer, but to have a system that is more suitable for wider usage, some issues need attention.

Maintaining the Singleton Property

The process of retrieving the data has some subtle complications associated with it. The `Library`, `MemberList` and `Catalog` objects are singletons: they cannot have more than one instance. Using the serialization mechanism, *it is now possible to serialize an object and then deserialize it to get a second instance.* For example, see the following pseudocode.

```
Library library = Library.instance();
Serialize library onto a disk file "library1";
Library library2 = Deserialized version of "library1";
Update library (add a member);
Update library2 (delete a book);
```

The first three lines of the pseudo-code are shown pictorially in Fig. 7.19. What has happened is that some user of the `Library` object initially obtained an instance of the `Library` object: essential and valid. In the second line, the user makes a copy of the object on disk: this is also perfectly legal and necessary. What follows in the third step is the problem. The user is now able to deserialize the object and obtain a second copy. The two copies can then diverge via independent updates as in the last two lines.

To understand what the essential problem is, recall that the intent of the singleton pattern is to *ensure that a class has only one instance and provide a global point of access to it.* We now have two mechanisms that can create instances of a class: (i) *constructors* and (ii) *deserialization.* The first mechanism was controlled by making constructors private and requiring all instantiations to got through the `instance` method. We now need a way of restricting the creation mechanism of deserialization.

Fortunately, due to the manner in which the reading of objects takes place in Java, this is not a complicated task. The default `readObject` method can be overridden to ignore retrieval if a copy already exists in memory. This way, no other class such as `UserInterface` will be able to do direct deserialization.

```
private void readObject(java.io.ObjectInputStream input) {
  try {
    input.defaultReadObject();
    if (library == null) {
```

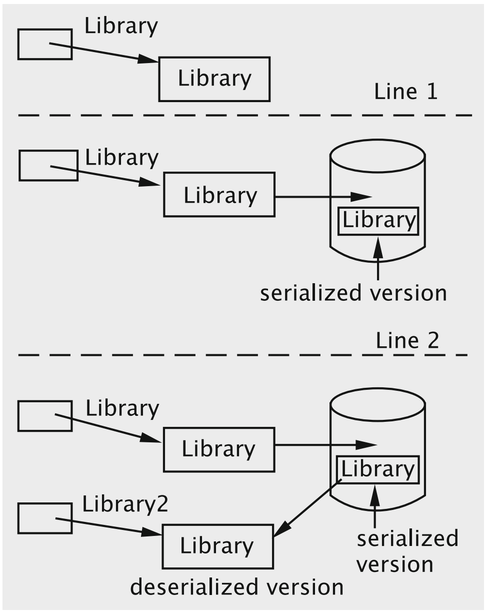


Fig. 7.19 A pitfall in using serialization with a singleton

```
          library = (Library) input.readObject();
        } else {
          input.readObject();
        }
      } catch(IOException ioe) {
        ioe.printStackTrace();
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
```

If there is no memory-resident copy of the `Library` object, the `retrieve` method reads the disk copy; otherwise, it returns the copy in memory. In case of an unexpected error, it returns `null`.

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

As discussed earlier, when we read (or write) a Library object, the Catalog and MemberList objects are automatically read (or written). However, since these are singletons, we will need to implement readObject for these classes in an analogous manner.

Dealing with Static Fields in Non-singletons

The above modifications take care of preserving the singleton classes, but the static fields in non-singletons pose a different challenge. Since the static field idCounter in Member stores the value that is used to generate the ID for each new member, this value must be saved along with the library. Since static fields are not serialized, this value will have to be explicitly written in the writeObject method of Member. The flip-side to this is that we will store a separate copy with each object, and as a result whenever a Member object is read, we are assigning a new value to idCounter, which makes our implementation very unstable. One simple solution to this is to circumvent the problem by encapsulating the static field as a separate class. The singleton MemberIdServer, shown below, holds the idCounter and also increments it each time getId is invoked.

```
class MemberIdServer implements Serializable {
  private  int idCounter;
  private static MemberIdServer server;
  private MemberIdServer() {
    idCounter = 1;
  }
  public static MemberIdServer instance() {
    if (server == null) {
      return (server = new MemberIdServer());
    } else {
      return server;
    }
  }
  public int getId() {
    return idCounter++;
  }
  // other code not shown
}
```

The methods for readObject and writeObject are defined as before to throw exceptions if the instance exists. Note that unlike the other objects which can all be reached directly or indirectly from references stored in Library, the instance of MemberIdServer does not have a stored reference in any other object. This raises the issue how this object will be serialized. The approach we adopt is to (de)serialize it in the file along with the the library object. The UI invokes the save method, which writes the instances of Library and MemberIdServer to the file LibraryData.

```
public static boolean save() {
  try {
    FileOutputStream file = new FileOutputStream("LibraryData");
    ObjectOutputStream output = new ObjectOutputStream(file);
    output.writeObject(library);
```

```
      output.writeObject(MemberIdServer.instance());
      return true;
    } catch(IOException ioe) {
      ioe.printStackTrace();
      return false;
    }
  }
```

The `retrieve` method reads the instance of `Library` and then invokes the `retrieve` method of `MemberIdServer`. These are defined as `static` methods since no instance of the singleton can exist if it has to be retrieved. The method in `Library` is shown below; the method in `MemberIdServer` invokes `readObject` on the input stream after the library has been deserialized.

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

7.3 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 maximise benefit, the reader is encouraged to apply the concepts 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.

7.3.1 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 recognised as an entity and make it a class; we can talk of an association between classes without any thought to how this will be realised.

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 finalise 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, the nitty-gritty 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.

7.3.2 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; 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 Chap. 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 Chap. 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.

7.3.3 The Facade Pattern

Earlier on, we discussed our preference for keeping the interface away from the complexity of the business logic implementation. This was done by having a `Library` class that provided a set of methods for the interface and thus served as a single point of entry to and exit from the business logic module. In the language of design patterns, what we created is known as a **facade.**

The structure of the facade is shown in Fig. 7.20. The primary motivation behind using a facade is to reduce the complexity by minimising communication and dependencies between a subsystem and its clients (Fig. 7.21). The facade not only shields the client from the complexity but also enables loose coupling between the subsystem and its clients. Facades are not typically designed to prevent the client from accessing the components within the subsystem.

Perhaps the most ubiquitous example of the use of facade is in designing the interface to an operating system. The system provides various menus through which users may invoke the standard operations of the operating system, thus shielding the user from its complexity. The interface does not prevent users from writing a script to customise operations, which gives them access to the components of the system. Common software packages also employ facades; a compiler is a good example. While the user may have direct access to components like the lexical analyser and the parser, the complexity of the system can be avoided by directly invoking the commands to compile a file.

Using a Facade
Where do we employ this? A situation in which we have:

1. A system with several individual classes, each with its own set of public methods.
2. An external entity interacting with the system requires knowledge of the public methods of several classes.

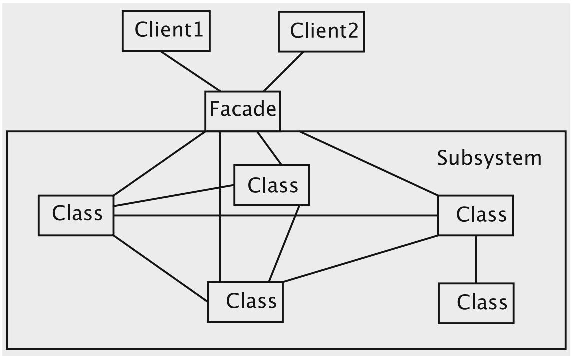


Fig. 7.20 Structure diagram for facade

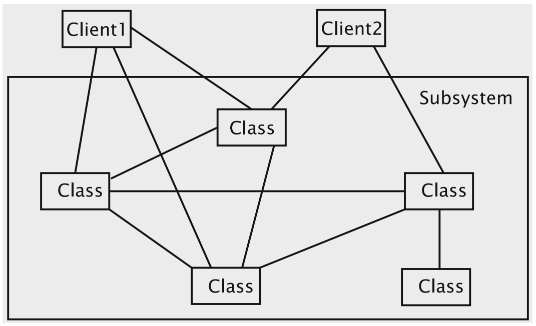


Fig. 7.21 Interactions with a subsystem without a facade

What problem are we facing? A lot of the details of the system have to be revealed to the external entity, which hurts modularity and abstraction. Coupling becomes tight, since a change to any one class in the system requires changes to all entities that interact with the system.

How have we solved it? Facade acts as a single class that:

1. Provides a single point of entry through which external entities can interact with the system without hurting abstraction.
2. Adapts to changes in individual classes in a manner such that external entities are unaffected, as long as the functionality of the system remains unchanged.

How have we employed it? The Library class acts as a facade through which the user interface communicates with the system. Library is aware of all the other classes and the methods that they provide. The methods in Library employ the functionality provided by the other classes to complete the tasks required of the system.

A facade enables the subsystem to have private/protected components. Such components are available only to the subsystem and its descendants, which prevents clients from accessing these just to get around the facade. When operations that involve private components have to be invoked, the client is forced to go through the facade. One apparent downside is that a facade is a largely 'custom-written' class that cannot be reused. However, the actual coding is quite simple, and the advantage gained by simplifying the interactions between other entities is worth this effort.

7.3.4 Implementing Singletons

Implementing a singleton correctly is not a trivial matter. In Chap. 5 we overcame the difficulties with creating a singleton hierarchy. In this chapter we have dealt with the issue of serialization. These solutions are very language specific and a careful study of the language features is needed when moving from the software classes to the implementation classes.

There do not appear to be any 'standard mechanisms' in the literature for handling implementation issues. Most languages provide a general collection of features that can be adapted for a variety of purposes. We have used the implementation of `readObject` and `writeObject` in Java to ensure that our purpose is served. Java also provides other methods like `readResolve` and `writeReplace` to override the effects of serialization and deserialization. The `Externalisable` interface can be employed when the serialization has to be fully customised.

7.3.5 Further Reading

The book by Meyer [1] devotes an entire chapter to the problem of class design and makes valuable reading. As we discussed earlier in the book, the notion of design patterns captures the idea that many design situations are similar in nature and a knowledge of the solution to these problems can make a designer more productive. The reader is encouraged to read the book by Gamma et al. [2] to get an exposure to the common design patterns. There are hundreds of other lesser patterns and a catalog of these can be found in [3, 4].

For a sophisticated introduction to the Unified Modeling Language, refer to [5]. The more enthusiastic reader is referred to the Object Management Group's UML Specification available online via www.omg.org.

To understand how an object-oriented schema fits in with a database, we refer the reader to [6].

Projects

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

7.4 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.

References

1. B. Meyer, *Object-Oriented Software Construction* (Prentice Hall, New York, 1997)
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, Reading, 1994)
3. L. Rising, *The Pattern Almanac* (Addison-Wesley, Boston, 2000)
4. J.M. Vlissides, J.O. Coplien, N.L. Kerth, *Pattern Languages of Program Design 2 (Software Patterns Series)* (Addison-Wesley, Reading, 1999)
5. M. Fowler, K. Scott, *UML Distilled* (Addison-Wesley Longman, Reading, 1997)
6. A. Chaudhri, R. Zicari, *Succeeding with Object Databases: A Practical Look at Today's Implementations with Java and XML* (Wiley, New York, 2000)