Chapter 5

Brahma Dathan Sarnath Ramnath

January 2, 2008

Contents

5 Designing a System **5**

5.1 Initiating the Design Process 5

5.1.1 The Major Subsystems 6

5.1.2 Identifying the Software Classes 7

5.2 Assigning Responsibilities to the Classes 10

5.2.1 Actions that Add New Entities (Populating the System) 11

5.2.2 Adding/Removing the Relationship Between Existing Entities 15

5.2.3 Actions that Remove Objects 20

5.2.4 Actions that Involve Queries 26

5.3 Designing the Classes 30

5.3.1 Designing the `Member` and `Book` classes 32

5.3.2 Building the Collection Classes 34

5.3.3 Designing `Hold` and `Transaction` classes 35

5.3.4 Designing the `Library` class 36

5.3.5 Constructing the User Interface 36

5.4 Designing for Safety and Security 36

5.4.1 Ensuring that the Data is not Inconsistent 37

5.4.2 Preventing Unauthorized Objects from Making Data Updates 38

5.4.3 Preventing Unauthorized Access through Exported Objects 39

5.5 Evaluating Quality of the Software Design 42

5.5.1 A New Requirement: Charging Fines for Overdue Books 43

5.5.2 The Initial Design 44

5.5.3 Evaluating and Improving the Solution 45

5.6 Discussion and Further Reading 50

5.6.1 Conceptual Classes and Software Classes 50

5.6.2 The Single Responsibility Principle 52

5.6.3 Recognizing Bad Smells and Design Flaws in Object-oriented Programs 53

5.6.4 Building a Commercially Acceptable System 53

5.6.5 Further Reading 54

Chapter 5

Designing a System

During the analysis, we constructed a conceptual model and a behavioral model for the proposed system. In the design step, we use the class structure defined in the conceptual model to design a system that behaves in the manner specified by the behavioral model. The main UML tool that we employ here is the sequence diagram. In a sequence diagram, the designer details how the behavior specified in the model will be realized. 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 (classes and objects) 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 realized 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.

As we shall see in this chapter, the design process involves making several choices. This is quite typical of any engineering design, and should come as no surprise. For instance, when a structure is designed, an engineer may be starting with an architect's plan and making choices about how various aspects of the plan are going to be realized; in doing this, the engineer is typically guided by principles that have been formulated through years of experience. Likewise, we have principles that guide the process of designing object-oriented software. A natural question that arises is: *Can we review the completed design and determine if the principles were applied correctly?* Fortunately, there are some things that can guide us in this. Experienced practitioners have identified *code smells* and *design smells* that help us identify parts of the completed design that do not square with best practice. The principles of *Refactoring* can prove to be useful in this context. Refactoring is most commonly used to improve an existing system, and refactoring rules guide us through this process. However, we can also apply these rules as we design the system, thereby avoiding mistakes that can be expensive to correct after deployment. In the later part of this chapter we examine a situation where reviewing a completed design helps us to refactor and improve it.

5.1 Initiating the Design Process

In the broadest sense, the design process requires a number of questions 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?

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.

3. What will the software architecture be, i.e. the major subsystems and their relationships?

4. What user interfaces will the system provide? These include GUI screens, printouts, and other devices (for example, library cards).

5. What classes and interfaces need to be coded? What are their responsibilities?

6. How is data stored on a permanent basis? What medium will be used? What model will be used for data storage?

7. What happens if there is a failure? Ideally, we would like to prevent data loss and corruption. What mechanisms are needed for realizing this?

8. Will the system use multiple computers? If so, what are the issues related to data and code distribution?

9. What kind of protection mechanisms will the system use?

However, our focus in this book is on object-oriented design and development using the the Java programming language. Hence 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 major subsystems, and 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.

5.1.1 The 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. **Back End.** 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 they can be varied independently. That way, we get good cohesion within each subsystem. Our focus in this chapter is mainly on the design and implementation of the back end.

We put together a rudimentary, text-based UI, that enables to test the system. Later on, 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.

5.1.2 Identifying the Software Classes

The next step is to identify and 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. Let us examine each of the conceptual classes and decide which conceptual classes have corresponding software classes.

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 behavior 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.

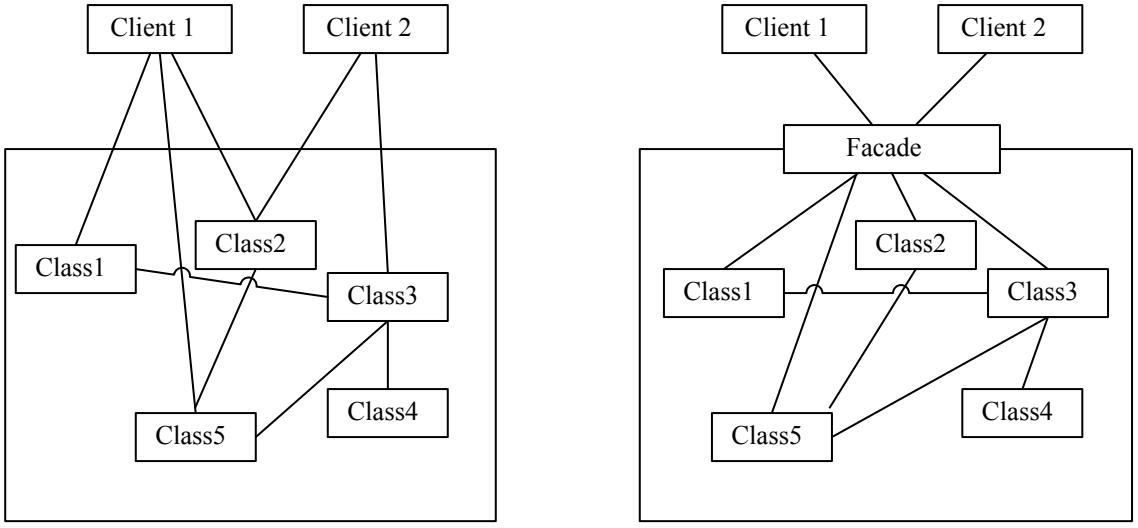
   One of the important principles of object-oriented design is that every computation must be represented as an application of a method on a given object, which is then treated as the current object for the computation. All the computation required of the business logic module must be executed on some 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.

   We therefore create a `Library` class that provides a set of methods for the interface, and serves 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.**

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

The Facade Pattern

The Facade pattern provides a unified interface to a subsystem, allowing external objects to use the subsystem without any knowledge of the details. It is a **Structural** pattern, since it provides a simple way to realize the relationship between external objects and the subsystem. The relationship here is that the external object is invoking the functionality provided by the subsystem. The subsystem has several individual classes, each with its own set of public methods. Without the facade, the external object would require knowledge of the public methods of each individual class. The Facade shields the client from this requirement, thus enabling loose coupling between the subsystem and its clients. Facades are typically designed to prevent the client from accessing the components within the subsystem.
The facade provides a single point of entry through which external objects can interact with a subsystem, enabling abstraction. It also allows the entire system to adapt to changes in individual classes within the subsystem, as long as the functionality of the subsystem remains unchanged. The figure below shows how the relationship between the external(client) objects and the subsystem changes when we have a facade.



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 the 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 customize operations, which gives them access to the components of the system.
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.

Figure 5.1: The Facade Pattern. Contrast the two different ways in which the clients interact with the system. In the diagram in the right, the two clients access the Facade to utilize a subset of the functionality provided by the five classes.

`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.

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`.

Finding Other Classes The conceptual classes are a rich source for the software classes, However, they do not provide a complete set of software classes. We can find other classes using our experience with programming and software creation, and also by thinking about how the system will be implemented.

From the analysis, we see two important aspects of the `Library` class: the `Library` instance must keep track of the members of the library as well as the books, which obviously imply maintenance of two collections. 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.

At this point we do not see the need for any more classes. 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.

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

5.2 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 cases, 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.

Returning the Result The detailed use cases tell us what result the system returns for each business process. Presenting these results is the responsibility of the UI, but the necessary information has to be returned by the `Library` class. As we get into the design, we need to specify what will be the type of the result in each business process. In some cases, we can return an object that contains all the information needed by the UI to display the output. In other cases, we may need to display a message that can be returned as a string, or return a result code (an integer) and allow the UI to decide what kind of message should be displayed. The two principles that guide us in these decisions are:

- Allow the UI the freedom to design the input and output.
- Avoid coupling between the UI and the back end.

If the result is, say, a string, the UI has little choice except to display it; on the other hand, if the result is a complex object that the UI has to decode, it could cause unwanted coupling. Since we have conflicting constraints, we will try and make the best decision, individually for each case.

Saving and Retrieving Data The business processes for *saving the data* and *retrieving the data* are not discussed here, since these operations depend heavily on the choice of programming language for implementation. The design for these is quite simple, and the difficulty lies in the implementation. We therefore defer the discussion of these to the next chapter.

5.2.1 Actions that Add New Entities (Populating the System)

The simplest of the operations for setting up a database is that of populating it with all the basic entities. Our system has only two kinds of basic entities: *member* and *book*. We therefore start with designing the operations for adding a member and for adding books.

Register Member

The sequence diagram for the use case for registering a member is shown in Figure 5.2. 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 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 a `Member` object and an operation, `insertMember()`, is invoked on `MemberList` to add the new member.

- **Option 2:** Invoke an `addNewMember()` method on `MemberList` and pass as parameters all the data about the new member. `MemberList` creates a `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 this option, `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. A second issue crops up in the generation of the `memberID`. Once again, we have two options:

- **Option 1:** Generate the ID in `Library` and pass in through the constructor.
- **Option 2:** Generate the ID in `Member`, through the constructor.

Here, it is important to note that we need a mechanism to ensure that no two members get the same ID, i.e., there has to be some central place where we keep track of how ids are generated. This makes it tempting to do this in the `Library` class, but is a poor design choice since this reduces the cohesiveness of the design by adding irrelevant responsibilities to the facade. The difficulty with doing it in `Member` is that we need to remember the history of previous IDs to avoid duplication. This difficulty can be overcome by using static fields and static methods. These details depend on the language, and are laid out when we implement the design.

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. The downside to this choice is that `Library` cannot communicate specific reasons for an unsuccessful operation.

Add Books

The next sequence diagram (Figure 5.3) 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 displays 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.

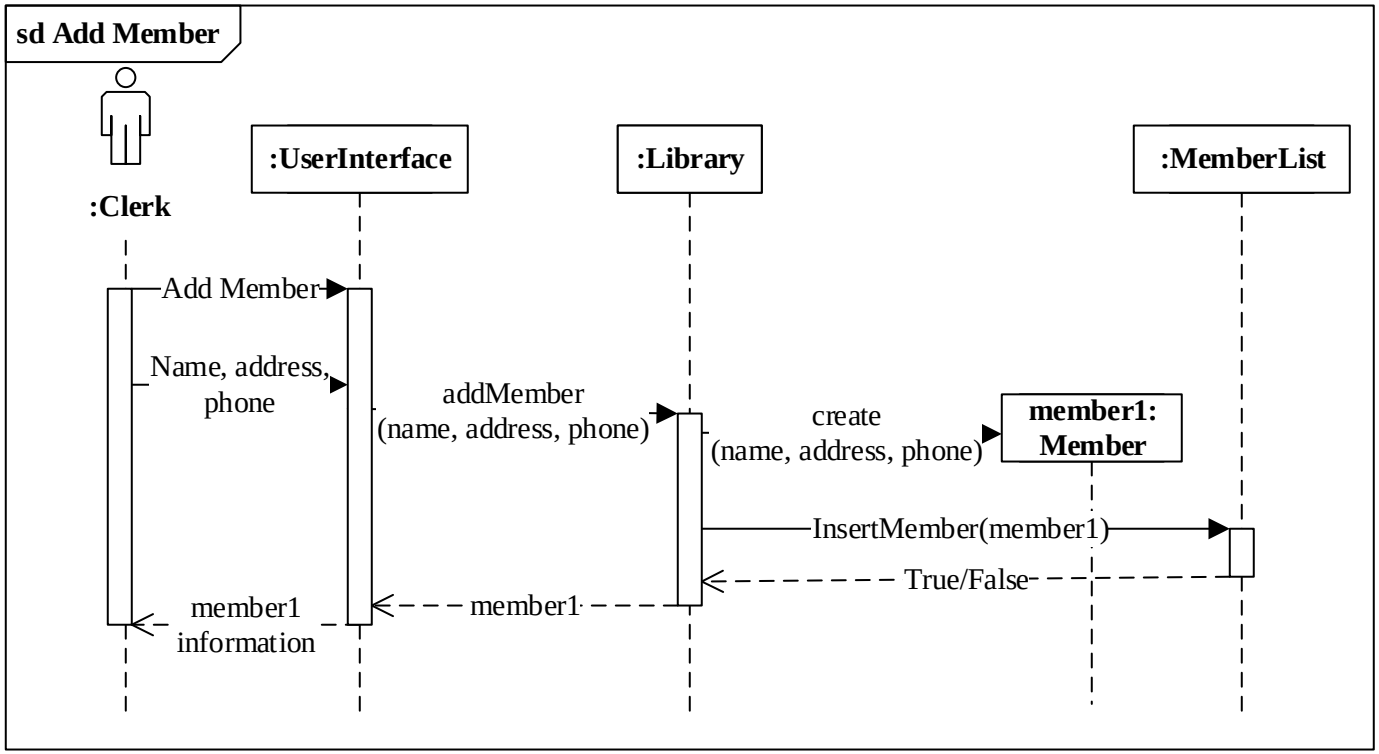


Figure 5.2: Sequence diagram for adding a new member. The `Library` invokes the constructor to create the object `member1`, and then adds the object to the `MemberList`.

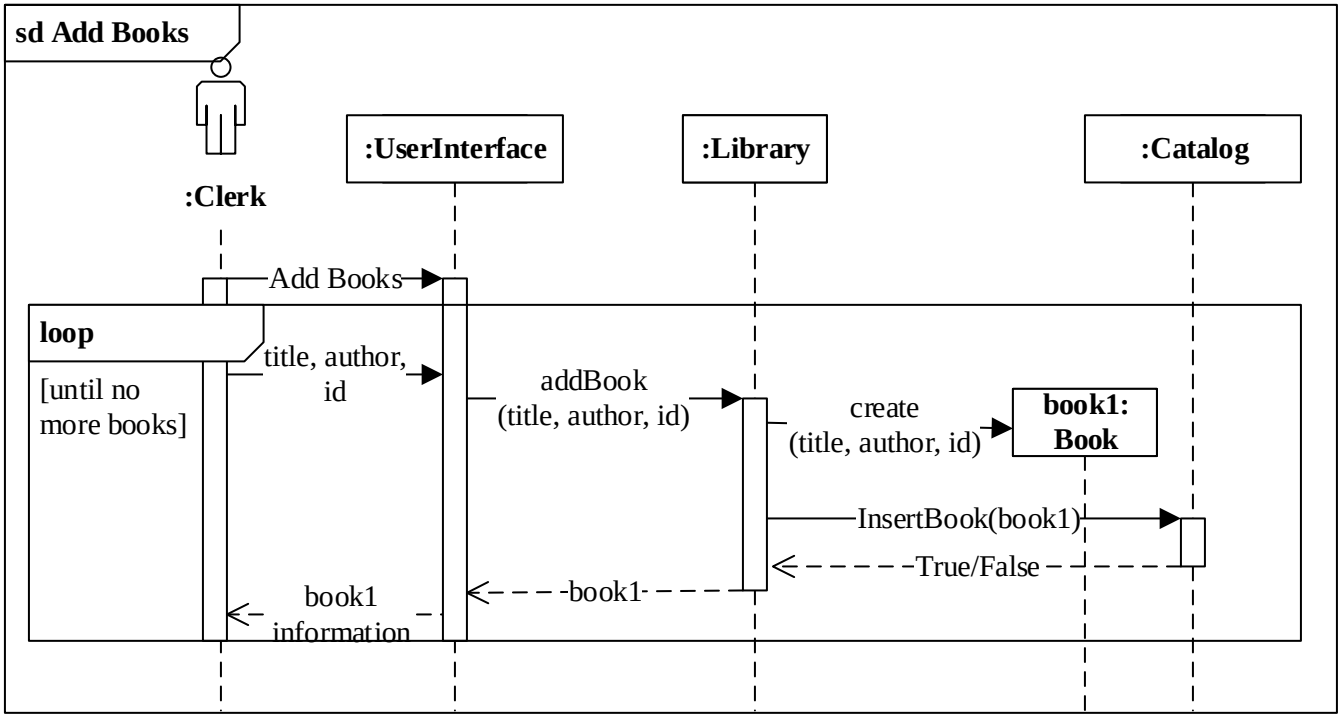


Figure 5.3: Sequence diagram for adding books. For each book, `Library` invokes the constructor, and adds the `Book` object to `Catalog`.

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 `member1` and `book1`, which do not exist at the start, but are created by invoking constructors. This is indicated by placing these boxes 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.

5.2.2 Adding/Removing the Relationship Between Existing Entities

The operation for checking out a book requires that the system record the book as being issued to the member and the member as having possession of the book. This is accomplished by modifying both the `Book` object and the `Member` object to reflect this relationship. Consequently, the operation for returning a book must modify the `Book` object and the `Member` object to indicate that the relationship no longer exists. Likewise, when a hold is placed, the system must record that the member has placed a hold on the book, and that the book is being held by the member. In all these situations, different parts of the operation are done in different objects. `Library` provides the glue code that ties all these operations together.

Issue Books

The sequence diagram for the Issue Books use case is given next. From the detailed use case, we see two steps:

1. Get the user's ID and verify that it is valid.
2. Issue the books, one at a time, to the user.

To enable these steps, `Library` provides two methods: `searchMember()`, and `issueBook()`. Since there are several books, the `issueBook()` method will be invoked several times, from within a loop.

Two options suggest themselves for implementing `searchMember()` method in `Library`:

- **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`.

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 concern because it can be removed using generic parameters as we shall see later in the text.

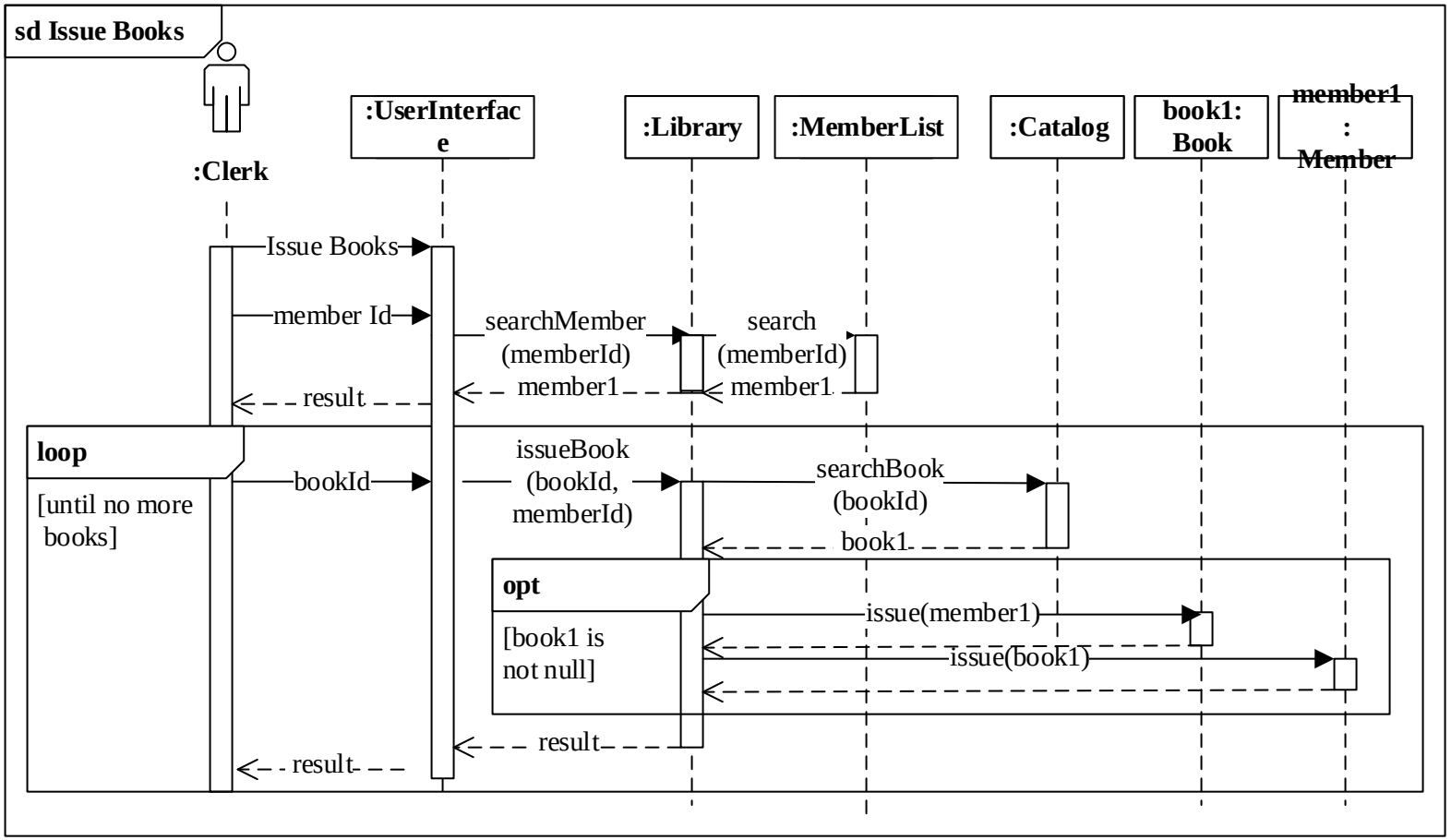


Figure 5.4: Sequence diagram for issuing books. The UI first verifies the member, then invokes the `issue()` method for each book.

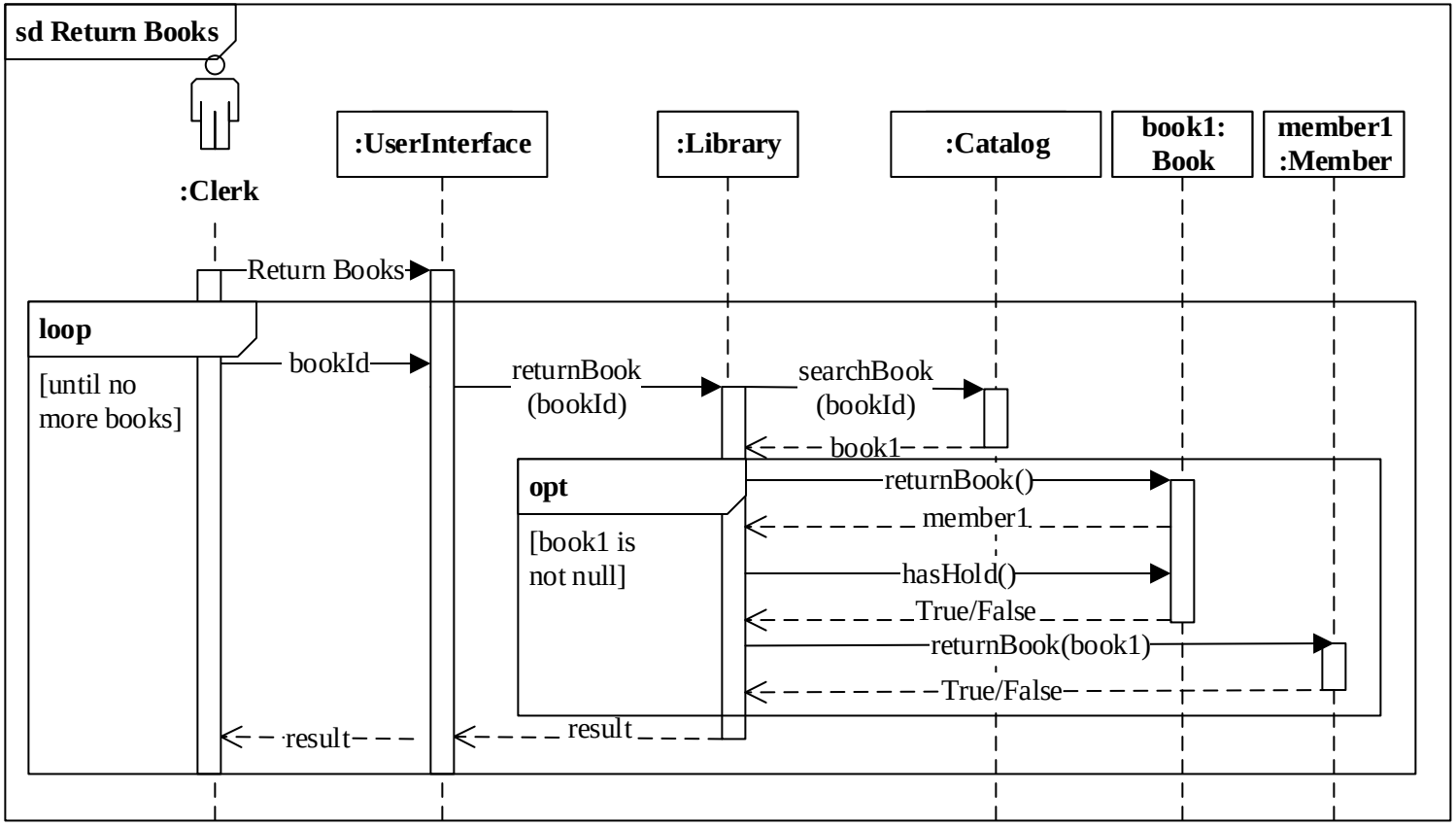


Figure 5.5: Sequence diagram for returning books: clerk can return several books issued to various members; both `Book` and `Member` are updated to reflect the return.

The `issueBook()` method involves the following steps:

1. Verify that the book ID is valid.
2. If the book is issuable:
    - record that the book is being issued to the member;
    - record that the member has possession of the book;
    - generate and record a due date for returning the book.
3. Return the result.

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.

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, and since we are storing the due-date as a field in `Book`, we can fold this detail into the `issue()` method of `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 sequence diagram for the Return Books use case is given next (Figure 5.5). From the detailed use case, we see that for each book a return operation is performed. To enable this, the `Library` provides the method `returnBook()`, which performs the following steps:

1. Verify that the book ID is valid.
2. If the book ID is valid:
    - record that the book has no borrower;
    - record that the member has returned the book;
    - check if the book has any holds.
3. Return the result.

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 `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 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 summarized 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.

For each book returned, we obtain the corresponding `book` object from `Catalog`. The `returnBook()` method is invoked on this object, and it returns the `member` object corresponding to the member who had borrowed the book. The `returnBook()` method is then invoked on the `member` object to record that the book has been returned. This business process has three possible outcomes that the use case requires the system to distinguish (Step 5 in the use case Return Book):

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 result returned by `returnBook()` must enable `UserInterface` make the distinction between these. This can be done by having `Library` return an integer result code.

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. The use case shows that the system accepts the book ID and the

member ID and places the hold. To enable this, `Library` provides the method `placeHold()`, which performs the following steps:

1. Verify that the book ID is valid.
2. Verify that the member ID is valid.
3. If both IDs are valid:
   - create the `Hold` object;
   - store a reference to the `Hold` object in the `member`;
   - store a reference to the `Hold` object in the `book`.
4. Return the result.

To store the reference to hold object, the `placeHold()` method is provided in both the `Book` class and the `Member` class. It is instructive to consider what alternative structures may be used for tracking the holds. One possibility is that both `Book` and `Member` create their own individualized `Hold` objects (`BookHold` and `MemberHold`), with the `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.

5.2.3 Actions that Remove Objects

Actions that remove objects can be tricky due to the fact that multiple objects may be holding references to the object removed. Verifying that these references are removed is critical, and this verification requires knowledge of all the associations between the classes.

Remove Hold

Removing a hold removes a relationship between a book and a user. However, since we have created a separate `Hold` object, we need to observe the precautions associated with removing objects. A request is issued to `Library` via the method `removeHold()`. As presented in the detailed use case, the user specifies the book ID and the member Id. We need to ensure that all references to the `Hold` object are removed before the object is removed. We know from our design that the only objects that ever store references to a `Hold` object are the corresponding `Book` and `Member` objects; furthermore, these are stored at the time the `Hold` object is constructed, and no additional references to the `Hold` object are created during its lifetime.

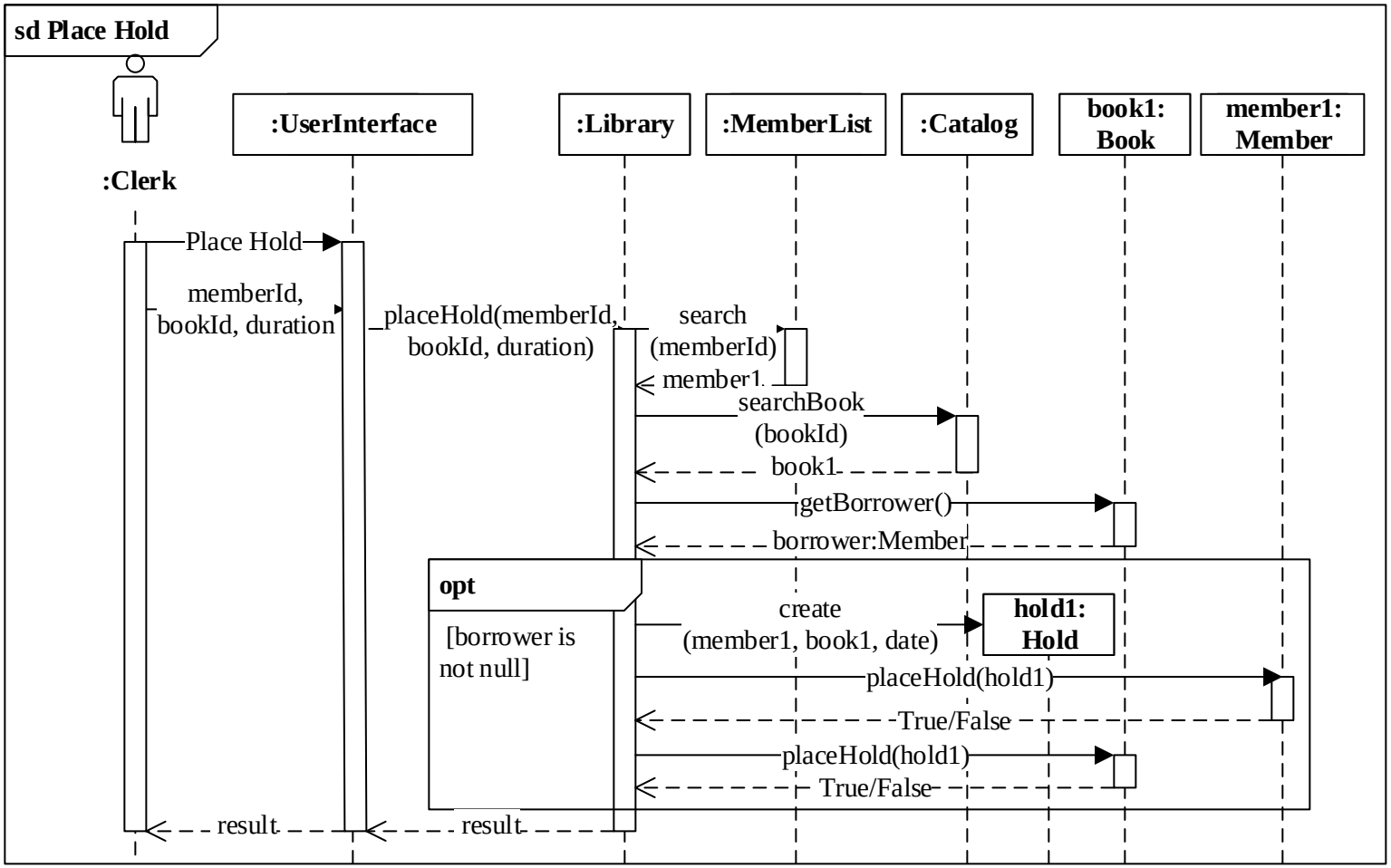


Figure 5.6: Sequence diagram for placing a hold: we create the object `hold1`, that connects `book1` and `member1`.

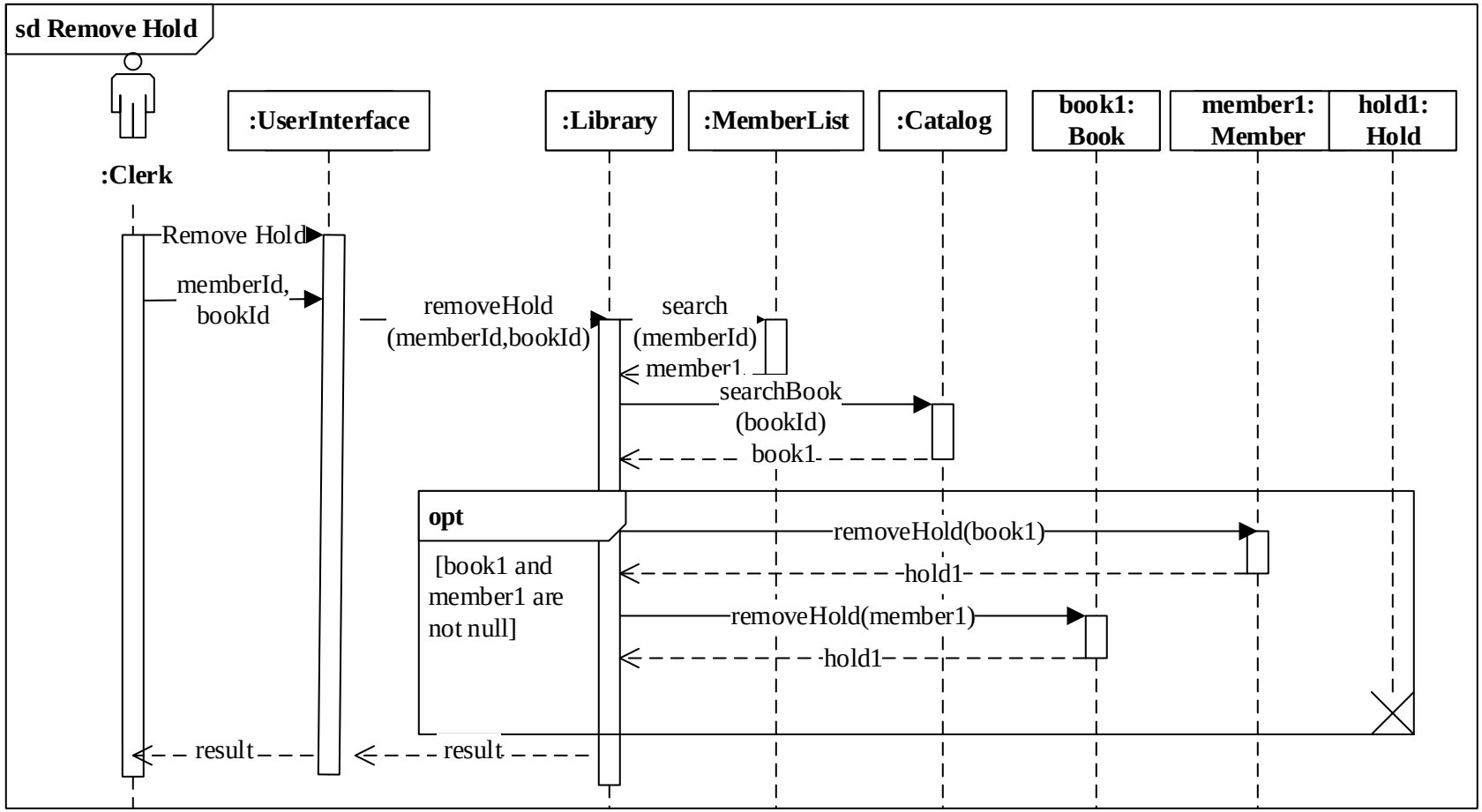


Figure 5.7: Sequence diagram for removing a hold: references to `hold1` must be removed from `book1` and `member1`.

The `removeHold()` method in `Library` therefore performs the following steps:

1. Retrieve the `Book` object using the book ID.
2. Retrieve the `Member` object using the member ID.
3. If both objects are successfully retrieved:
   - inform the `Member` object to remove the hold on the specified book;
   - inform the `Book` object to remove the hold by the specified member;
   - delete the `Hold` object.
4. Return the result.

To enable this, a `removeHold()` method is provided in both the `Book` class and the `Member` class. The sequence diagram is given in Figure 5.7.

Remove Books

Removing a `Book` object is complicated by the fact that during its lifetime, several objects store and delete references to it. However, from our structure we know that the only objects that can store references to a `Book` object are the `Catalog`, any one `Member` object, and multiple `Hold` objects. Accordingly, as discussed in the use case, we remove only those books that are not checked out and do not have a hold.
The `removeBook()` method in `Library` therefore performs the following steps:

1. Retrieve the `Book` object using the book ID.
2. If the object is successfully retrieved:
   - check if anyone has borrowed the book;
   - check if there are any holds on the book.
3. If there are no holds and no borrower, remove the book.
4. Return the result.

To enable this, the `getBorrower()` and `hasHold()` methods are provided in the `Book` class, and a `remove()` method is provided in `Catalog`. The diagram in Figure 5.8 shows the sequence diagram for removing books from the collection. The square brackets before the invocation of `remove()` contain the condition "book1 can be removed," indicating that the `Book` object is deleted only if this condition is met. `Library` returns a specific code for each possible outcome, which `UserInterface` translates into an appropriate message.

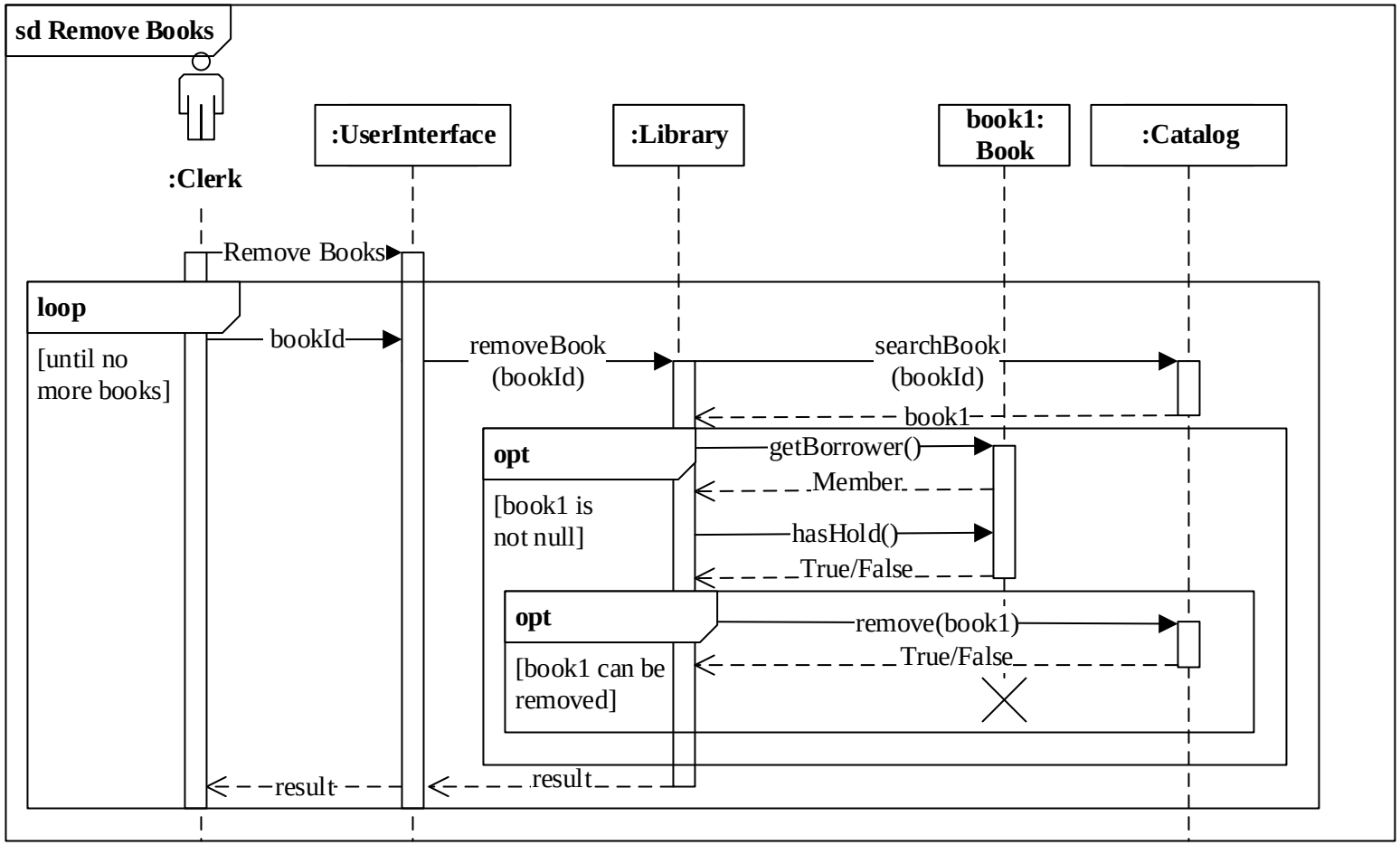


Figure 5.8: Sequence diagram for removing books: we ensure that no references to `book1` exist in any other object

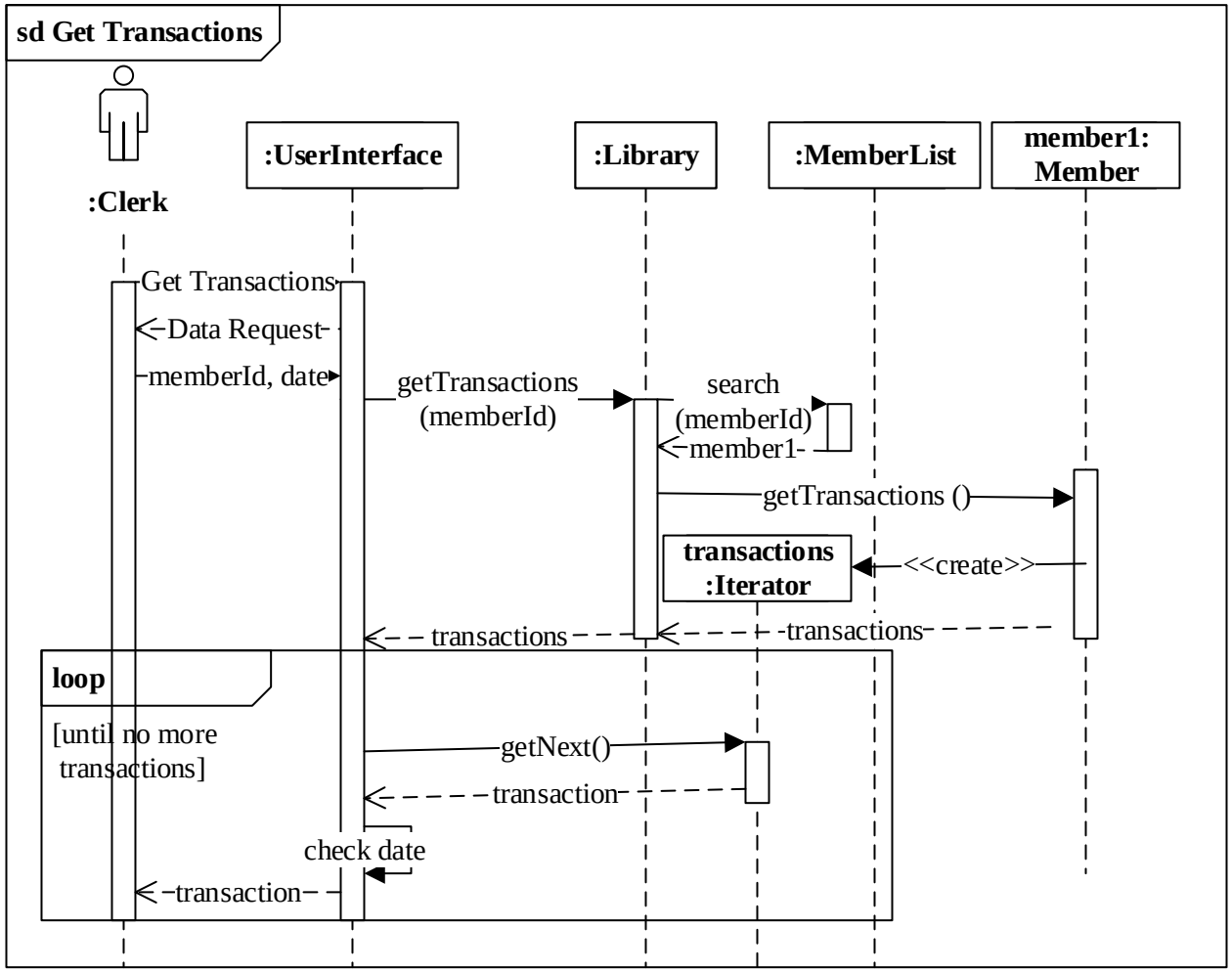


Figure 5.9: Sequence diagram for printing a member's transactions on a given date: the relevant transactions are extracted from `Iterator` in `UserInterface`

5.2.4 Actions that Involve Queries

Some actions require some data to be retrieved and displayed to the user. The simplest situation is that where the retrieved data is simply displayed. In more complex situations, additional actions may be needed depending on the results of the query.

Member Transactions

As described in the use case, the user specifies the member ID and a date, and prints the information about all the transactions by the corresponding member on the specified date. To access information about all transactions would be difficult if there no record of operations done by a member. Accordingly, we create a `Transaction` class and maintain a list of `transaction` objects for each member, and for all the relevant business processes, we generate the appropriate `transaction` object and store it in the list for the member. This query can be easily answered by accessing this list. There are two steps that `UserInterface` performs:

1. Query the system to get the transactions.
2. Display the transactions.

We have two options for structuring this process:

- **Option 1:** Provide a method `getTransactions()` in `Library`, that returns a list of transactions for the given member, on the given date only. The UI simply displays all the items in the list in the required format.
- **Option 2:** Provide a method `getTransactions()` in `Library`, that returns the entire list of transactions for the given member. The UI filters the list, and displays only transactions on the specified date in the required format.

Option 1 is attractive because it reduces the coupling between the UI and the back end. The UI does not have to be involved in any details of how the answer to the query is generated, and deals only with displaying the result. Option 2 is attractive because we can easily accommodate variations in the query. Say, we want all the transactions since the specified date. This change can be easily done in the code in the UI, and other classes are not affected. The question then arises: *Under what circumstances is it acceptable to allow this coupling between the UI and the backend?*

To answer this, we need to look at the specifics of the situation. We are maintaining a record of each transaction. A simple way to record transactions would be a descriptive string and a date, and the filtering process would only involve accessing the date. Hence, the UI does not have to know any details of what is happening in the back end, and the process would not increase coupling between the UI and the back end. We can therefore go with Option 2, which offers us greater flexibility for later changes.

The `UserInterface` therefore does the following:

1. Query the system to get all the transactions for the specified member.
2. Display the transactions for the specified date in the appropriate format.

The method `getTransactions()` in `Library` does the following:

1. Verify that the member ID is valid.
2. If valid, retrieve the list of transactions for the member.
3. Return the result.

The sequence diagram in Figure 5.9 describes the details. 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 `Transaction` objects. `Library` returns this to the UI, which extracts and displays the needed information.

Process Holds

The operation here is to retrieve the next valid hold, delete the hold object and display the relevant information. This operation is not strictly a query, but performs one when the next hold is retrieved. The input here is only the ID for the book.

The `processHold()` method in `Library` therefore performs the following steps:

1. Retrieve the `book` object using the book ID.
2. If the `book` exists, get the next valid hold.
3. If there is a hold:
   - get the associated member object;
   - remove the hold from the book;
   - remove the hold from the member.
4. Return the result.

To enable this, we add the method `getNextHold()` to the `Book` class, and the method `getMember()` to the `Hold` class. The diagram in Figure 5.10 shows the sequence diagram for processing holds. Since we process several books in this use case, a loop is placed around the steps described above.

Renew Books

As described in the use case, the system displays the books checked out to the member and allows the actor to specify the ones that need to be renewed. There are two steps to this process:

1. Query the system to get the books.
2. Display the books and perform the requested operation on each book.

This operation is not strictly a query, but performs a query to get a list of all the books issued to a user.

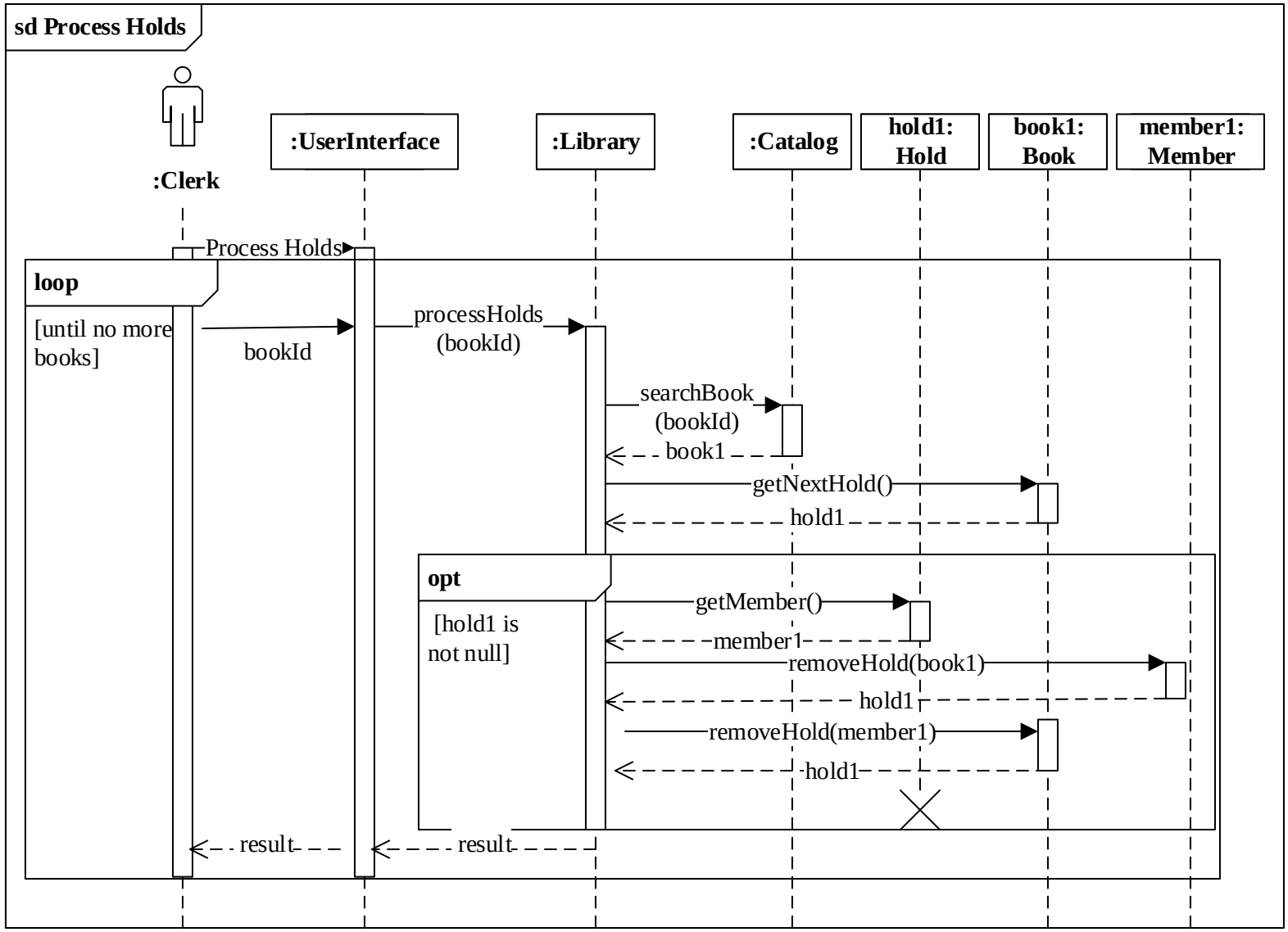


Figure 5.10: Sequence diagram for processing holds: the next valid hold is identified and processed.

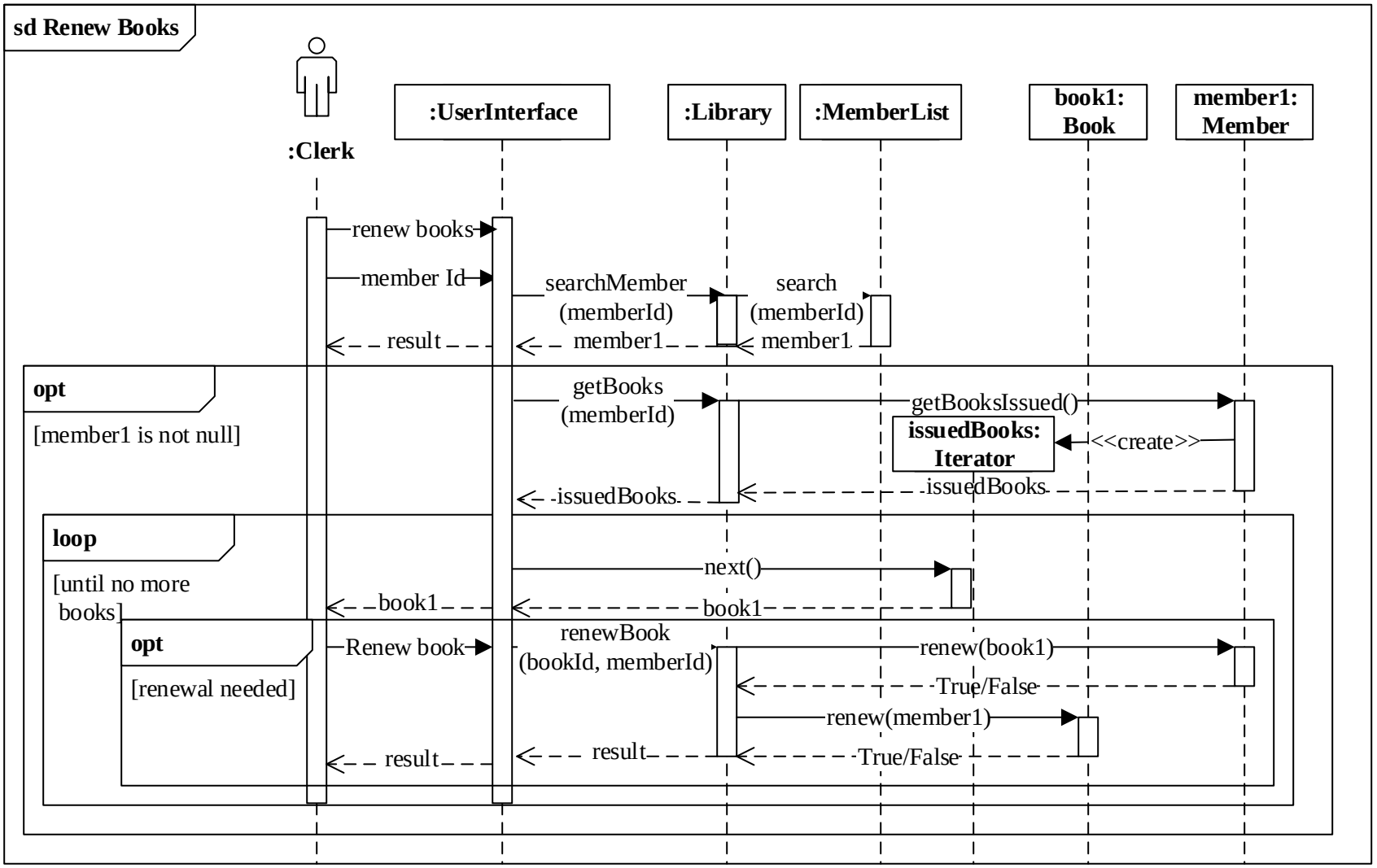


Figure 5.11: Sequence diagram for renewing books: the `UserInterface` queries the actor about each book.

We provide a query operation, `getBooks()` in `Library`, to get the list of books issued to a member, and a `renewbook()` method in `Library` to renew a single book. The UI uses these methods to carry out both the steps described above. A beginner may wonder why we do not provide a method `renewBooks()` in `Library`, that is invoked by the UI, and renews all the books. This approach may seem more compact, since it can handle both the steps described above. This is not an option, because after getting the list of books, the `renewBooks()` method in `Library` has to seek the user input on each book. This makes a method in `Library` responsible for user interaction, which is unacceptable.

The `renewBooks()` method in the `UserInterface` therefore gets the member ID and performs the following steps:

1. Query `Library` to get the list of books issued to the member.
2. For each `book` object do the following:
    - display the `book` details and ask if user wants to renew;
    - if yes, call the `renewBook()` method in `Library`;
    - display the result.

The `renewBook()` method in `Library` does the following:

1. Retrieve the `book` and `member` objects using the book ID and member ID.
2. If the objects are successfully retrieved:
    - call the `renew()` method on the `book` object;
    - if renewed, call the `renew()` method on the `member` object;
3. Return the result.

Figure 5.11 shows the sequence diagram for renewing books.

5.3 Designing the Classes

From the previous section, we can see that we have the following software classes:

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

The relationships between these classes are shown in Figure 5.12. Note that `Hold` is not shown as an association class, but as 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.

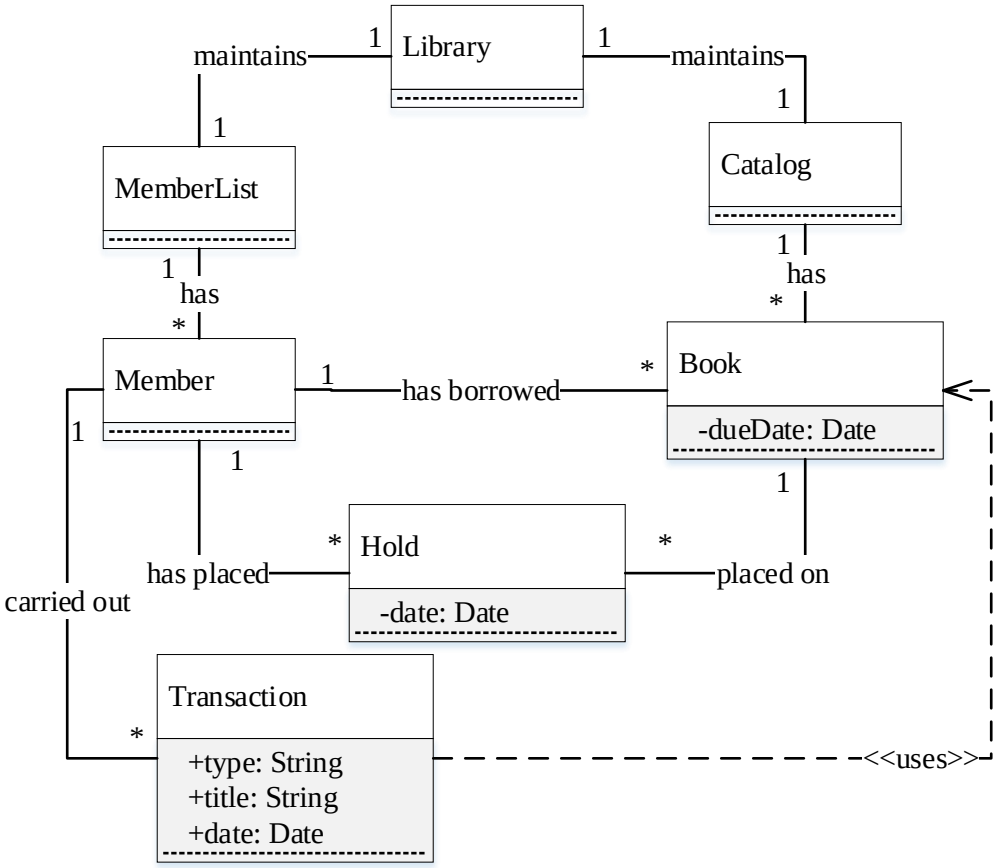


Figure 5.12: Software Classes for the Library System: Figure represents all the entities identified as Software Classes. Some fields (such as `dueDate` for `Book`) represent design decisions and are included. Other details are deferred to the diagrams for individual classes.

Sequence Diagram	Method(s) to add	Description
Add Member	constructor	Parameters are name, address, phone number; generates the ID
Issue Books	issue()	Stores a reference to the given book object in the member object; records the transaction
Place Hold	placeHold()	Stores a reference to the given hold object in the member object; records the transaction
Return Books	returnBook()	Removes the reference to the given book from the member object; records the transaction
Process Holds	removeHold()	Removes the reference to the given hold from the member object; records the transaction
Remove Hold	removeHold()	Removes the reference to the given hold from the member object; records the transaction
Renew Books	getBooksIssued() renew()	Returns a list of all the books currently issued to the member Verifies that a reference to the given book is in the member object; records the transaction
Get Transactions	getTransactions()	Returns a list of all the transactions of the member

Figure 5.13: Collecting the methods the `Member` class: Each sequence diagram tells us what methods have to be invoked on the objects of the `Member` class, and what task the method should perform.

5.3.1 Designing the `Member` and `Book` classes

In the course of the design, we have seen that the details of the operations are performed in the `member` and `book` objects. Getting these classes specified correctly is therefore critical.

The `Member` Class

Let us examine the `Member` class. From Figure 5.12, we see that a `Member` object may store several references to `Book` objects, several references to `Hold` objects, and several references to `Transaction` objects. Accordingly, the `Member` class has a list of books, a list of holds, and a list of transactions. In addition, we store the personal data for the member. To decide the list of methods for the `Member` class, we do the following:

1. Provide accessors and modifiers for all the appropriate data fields.
2. Provide the methods needed to implement the designs described in our sequence diagrams.

Accessors are provided for all the data fields. The methods should have the appropriate return type. For instance, `name, address, id`, etc., are strings. Hence the return type for `getName(), getAddress(), getId()`, etc., is `String`. We provide modifiers for `name (setName())` and `address (setAddress())`, but not for `id`, since a member ID is generated by the system. Note that in specifying the types of attributes, we may have to make language-specific choices; some of these decisions may therefore be deferred to the implementation stage.

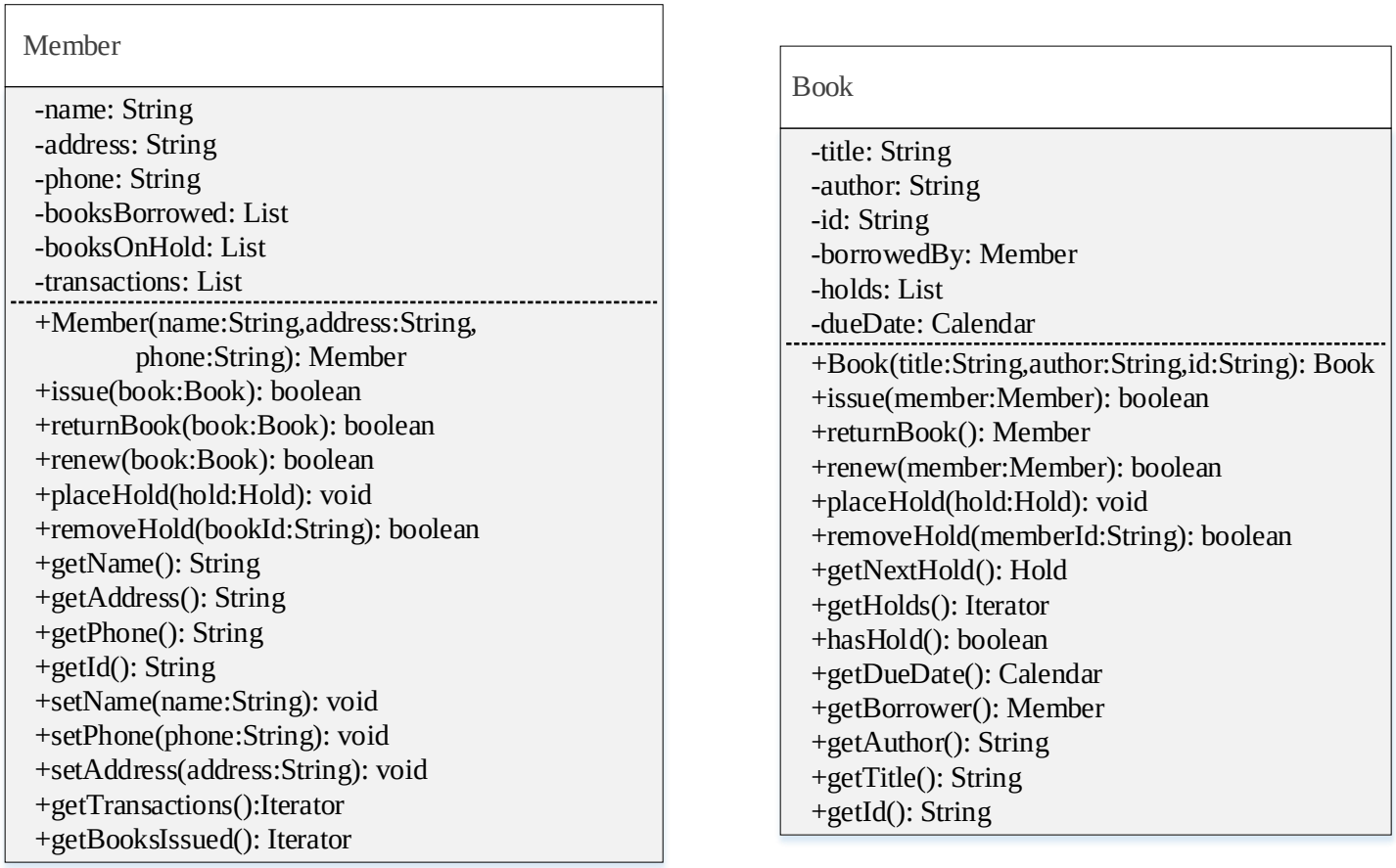


Figure 5.14: Class Diagram for the `Member` and `Book` classes: The fields are derived from the requirements and from the design process. In addition to the necessary accessor and modifier methods for the fields, we include the methods required by the sequence diagrams.

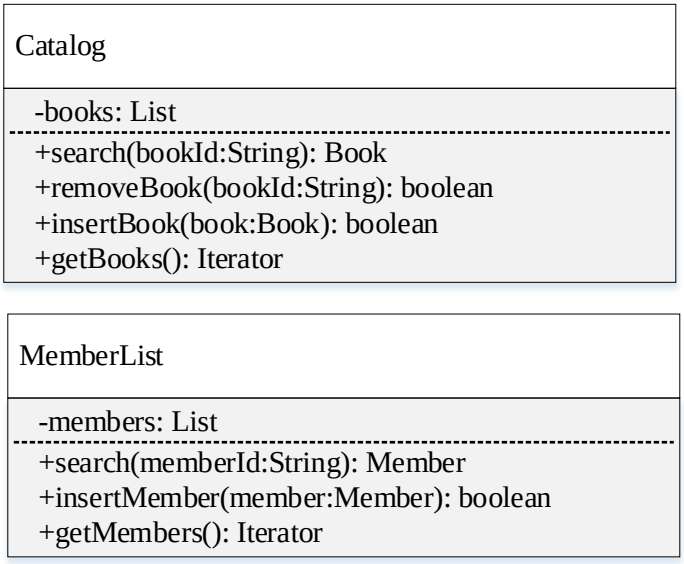


Figure 5.15: Class Diagram for `Catalog` and `MemberList`: The collection classes comprise of a `List` and methods for list operations.

Let us now examine all the sequence diagrams and see what methods are needed. This can be succinctly done, as shown in Table 5.13. Using the information about the `Member` attributes and the information in this table, we can generate a class diagram for `Member`.

The `Book` Class

The approach to developing the class diagram for `Book` parallels that of the `Member` class. A table similar to 5.13 is created from the sequence diagrams (see exercises). Fields are identified from the requirements and the appropriate accessor methods are added. However, there are no setters for the `Book` class because we don't expect to change anything in a `Book` object. The class diagrams for `Book` and `Member` given in Figure 5.14.

5.3.2 Building the Collection Classes

All the collection classes would be built around an existing `List` class. Typical operations on a list would be add, remove, and search for objects. Our collection classes are `Catalog` and `MemberList`, each of which store a reference to an object of type `List`. These `List` objects store references to `book` objects and `member` objects respectively. In `Catalog`, we have method `getBooks`, whose return type is `Iterator`. This enables `Library` to get an enumeration of all the books so that any specialized operations that have to be applied to the collection are facilitated. Since the requirements did not ask for the functionality of removing a member, there is no `remove()` method in the `MemberList` class. The resulting class diagrams are shown in 5.15

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 to 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.

5.3.3 Designing `Hold` and `Transaction` classes

These classes are created for a single purpose and are quite simple. The `Hold` class is defined for the purpose of capturing a relationship between `Book` and `Member`. Besides the accessors, `getMember(), getBook()`, and `getDate()`, the `Hold` class needs a method `isValid()` method, which checks whether a certain hold is still valid.

The `Transaction` class needs a date, which could itself be a class. Since languages may provide this (Java's `util` package has a class `Calendar`) for the time being we will designate

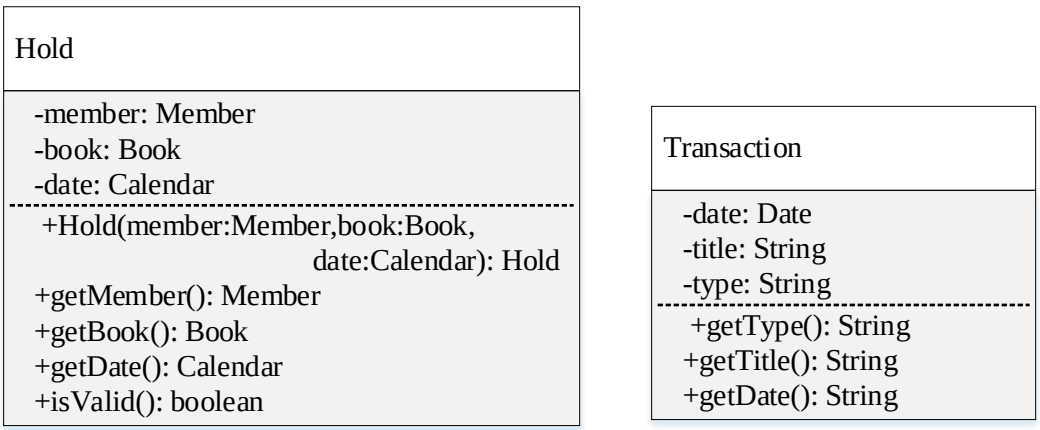


Figure 5.16: Class Diagrams for `Hold` and `Transaction`: These classes are created for specific purposes, and need to store only the relevant data and methods.

the class as `Date`. We also need the title of the associated book and a descriptive string. Accessors are provided for each field. Since this object is used only to record the details of a transaction, no mutators are provided. The class diagrams are shown in Figure 5.16.

5.3.4 Designing the `Library` class

The methods are simply a collection of the 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.

We have already seen that the class must maintain references to `Catalog` and `MemberList`.

See Figure 5.17 for the class diagram.

5.3.5 Constructing the User Interface

As we discussed earlier, all input/output will be via a simple text interface. The system will present a menu with user choices, and will accept input from, and display output to the user. All of this will be accomplished through an imperative program, and is therefore not of interest for an object-oriented design. The details are deferred to the implementation phase.

5.4 Designing for Safety and Security

Building adequate safety and security into a system is a complex task, and is usually handled at several levels. At the level of our design, it is important to know what kind of vulnerabilities can exist, and what design choices must be made to secure the system. Distinguishing between safety and security is tricky: safety generally refers to accidental threats, and security refers to intentional threats. However, if a system is unsafe due to poor construction, this could lead to the presence of vulnerabilities that can be exploited by malicious actors. Being aware of these vulnerabilities enables us to build some protections into the software during the design.

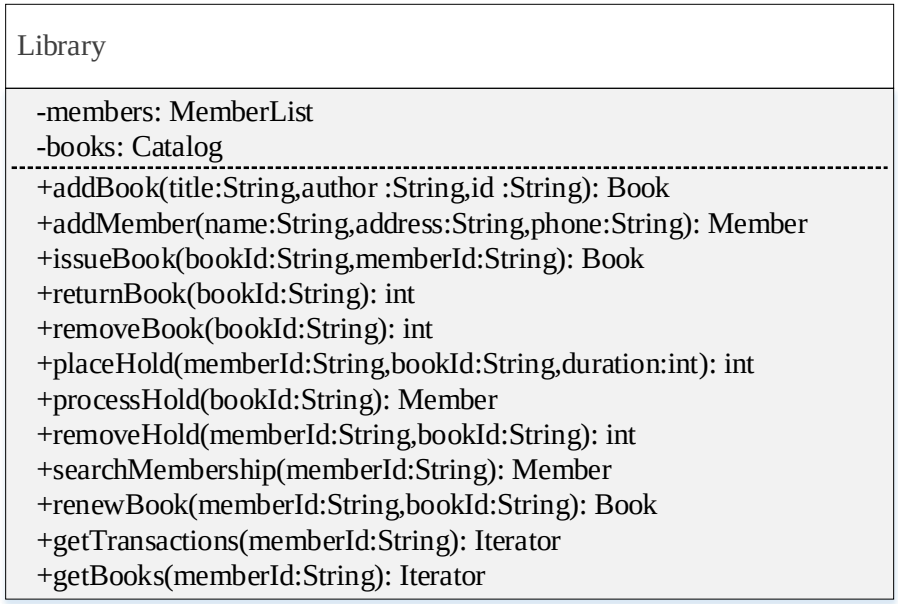


Figure 5.17: Class diagram for `Library`: Methods are provided to ensure that all the business processes can be carried out. Fields are provided for the collections.

For a system like the library, let us examine three situations that can compromise the safety and security of our system.

1. An error during the data update causes the data to become inconsistent.
2. Data updates are performed through imported objects not created within the system.
3. The data being exported is accessed by unauthorized objects.

Each of these illustrates a different example of how the system ends up in an undesirable state. We shall examine each of these in detail below.

5.4.1 Ensuring that the Data is not Inconsistent

In any object-oriented system, all the data items (objects and classes) must satisfy certain **integrity constraints**. For example, in the library system, if a `Member` object `m1` is shown as having checked out `b1`, then `b1` must indicate `m1` as its borrower. It is in general hard, if not impossible, to list all integrity constraints in a collection of data items; but bug-free application programs preserve them. That is, every application program that reads a consistent set of data items, leaves the data items in a consistent state, when it completes.

In the context of this text book, we consider three ways in which data integrity constraints could be violated.

1. The data could end up in an inconsistent state due to system errors. Suppose in the course of issuing a book, the system manages to mark a book as borrowed by a member, but as it attempts to mark the member as having borrowed the book, encounters an error. The system would be in an inconsistent state.

2. Untrusted, malicious code could get unauthorized access to insufficiently-protected data. For example, classes in the `ui` package could obtain a reference to the `Catalog` object. Or they could find a way to access the database stored on external storage. After gaining access to the catalog, the user interface program could wreak havoc deleting books as it pleases, making incorrect updates, and so on.

3. The design calls for the business logic to provide references of business objects to untrusted code, which could result in a security leak. In our design, for example, `Library` returns a `Member` reference in the course of issuing a book. This reference could then be used to make updates to that object (mark the member as having not checked out a book, for instance) or any objects accessible via that object (for example, access and make changes to `Hold` objects accessible through the `Member` reference).

Cases 2 and 3 are discussed in the next two subsections. In case 1, we can apply various practices like formal verification and validation, testing and debugging until the system is found to preserve the integrity constraints. In addition, integrity can be lost when the underlying hardware or the operating system behave in unexpected ways. These are handled in different ways, depending on the implementation:.

- The data is completely stored in memory as in our design. The error could happen only if there is a system error such as a hardware issue or some kind of resource (such as memory) unavailability. In these cases, the program crashes and we lose all updates done since the last save to disk. On restarting the program, we revert to the data saved on disk, which is in a consistent state. By saving data after every transaction, we can avoid anomalies such as sending a customer home with a bunch of books, without having a record of that in the library data.

- The data is backed by a database system. In this case, the database management system provides support for rolling back the system to a consistent state.

It also helps ensure that concurrent updates to the database do not result in inconsistency and program or hardware failures do not result in lost data or data consistency issues. We may lose incomplete transactions, but the database itself will be consistent.

5.4.2 Preventing Unauthorized Objects from Making Data Updates

In our design, all the operations are carried out through actions taken by the objects we have created within the Library system. In such a situation, the methods discussed in the previous section can be employed to ensure consistency, but for this to be effective, we also need to ensure that no action can be taken by any object not created within our system. For example, how do we ensure that `UserInterface` is unable to get a hold of the `Catalog` object?

Providing a degree of safety to the objects would require actions at different levels: proper design, employment of relevant language features, following proper manual and automated procedures, and so on. Since our scope is limited to design and implementation, our focus would be on these. The reader is cautioned that our approaches are presented as examples of what can be done, and do not cover all possible safety issues.

Actions we Take During Implementation

Secure coding practices play a big role in ensuring safety. These practices range from very basic things like using good variable names, comments and structured programming. They are often dependent on the programmer's familiarity with the idioms of the language being used. Knowledge of the language libraries and the manner in which design patterns are implemented also plays a big role.

We have decided to use Java for implementation, so we will need to see some of the Java features to use in this connection. This will be dealt with in a following chapter of this text. The reader may wish to explore facilities supported in other languages, should they choose to use a different language.

Good Safety Practices During Design

Applying good class design principles and design patterns play a big role in ensuring safety. Software vulnerabilities caused by violating these cannot be corrected in later stages of development. For our specific situation, we look at two concepts that have been applied

Using the Design Patterns to restrict access. Facade, Iterator enable us to hide details

Not importing object references into the system If we allow this, it is easy to breach security.

5.4.3 Preventing Unauthorized Access through Exported Objects

As we shall see in the next chapter, Java provides some mechanisms to control access at the class level or at the method level. Resorting to language support alone may not be feasible in all circumstances for the following reasons.

1. The implementation language may not always support a mechanism to regulate access.

2. Even in a language that allows some convenient mechanism, there may be circumstances where the mechanism cannot be applied. For example, if a method `m1` is declared `public`, it is impossible to narrow `m1`'s access to, say, `protected` while overriding.

3. Access at implementation level lacks flexibility. For example, if a method is declared `protected`, all subclasses see that as a protected method. The `protected` declaration alone is insufficient to discriminate between the subclasses.

We now describe a design approach that allows us to restrict access to selected methods. We illustrate our design using the library system.

The basic idea is to implement multiple versions of classes for which we should allow different levels of access. Let us focus on two methods.

1. One of them gets the title of a book. Let us assume that we wish to allow all objects that have access to a `Book` object access to its title as well.

2. The second one marks the book as returned. We consider this to be used in a controlled manner, and do not trust every holder of a `Book` reference to judiciously use this method.

Clearly, providing the same `Book` reference to all objects is not a solution. We need two types of references, one of which permits full access to both methods and the other allowing to get the title but preventing book returns. But both must be of type `Book` as well. We can accomplish this by creating two classes, which we name `FullBook` and `ReadOnlyBook`. The idea is that `FullBook` supports both read (getting the title) and write (allowing book returns), whereas `ReadOnlyBook` supports the read operation, but not the write. When we export a `Book` reference to untrusted code, we supply a reference whose actual type is `ReadOnlyBook`. We supply a `FullBook` reference to trusted code.

Let us clearly state the major design issues.

1. Any code that used to receive a `Book` reference should also be able to accept a `ReadOnlyBook` reference. Of course, the code may not successfully execute the "write" methods anymore.
2. Similarly, any code that used to receive a `Book` reference should be able to now take a `FullBook` reference.
3. No matter what changes are made to the book object, the "read" methods should see the most recent update.
4. If the book object is deleted, neither reference should be valid.

For every `FullBook` object, we create exactly one `ReadOnlyBook` object. This can be done at the time of creation of `FullBook`, in the `Book` constructor. The first two issues can easily be satisfied by having `Book` as a superclass and both `FullBook` and `ReadOnlyBook` as its subclasses. We could have all code that originally belonged to `Book` be put in `FullBook`, so that code would be used for all updates. Then we make `ReadOnlyBook` an object adapter that uses `FullBook` as an object adaptee. This ensures that any data retrieved via a `ReadOnlyBook` reference returns the same value as a `FullBook` reference and satisfies the third condition.

Now, for meeting condition 4, we proceed as follows.

1. `Catalog` declares a list of references to `Book` objects, but the actual type of the reference is `FullBook`.
2. In the constructor of `FullBook`, we create a `ReadOnlyOBook` object and have the `FullBook` (`ReadOnlyBook`) object store a reference to the `ReadOnlyBook` (`FullBook`) object.
3. Whenever a book is removed, we proceed as follows.
   (a) First, locate the reference to the `FullBook` object in `Catalog`.
   (b) Obtain the reference to `ReadOnlyBook`. Set their mutual references to `null`.

Data encapsulation can be enhanced by arranging to have `ReadOnlyBook` as an inner class of `FullBook`, if the language permits inner classes. In a language like Java, we can also have `ReadOnlyBook` declared a private inner class, so no other object needs to know about it.

The design is flexible enough to support multiple "views" of `Book`, if the situation warrants it. For instance, we could have a version of `Book` that supports renewal, but not removal or one that supports removal, but not renewal. For every combination of requirements, we simply create a new class, all of them object adapters of `FullBook`, with the `FullBook` object maintaining a reference to each type of object. All these objects can be instantiated in the constructor of `FullBook`. The design is depicted in Figure 5.18.

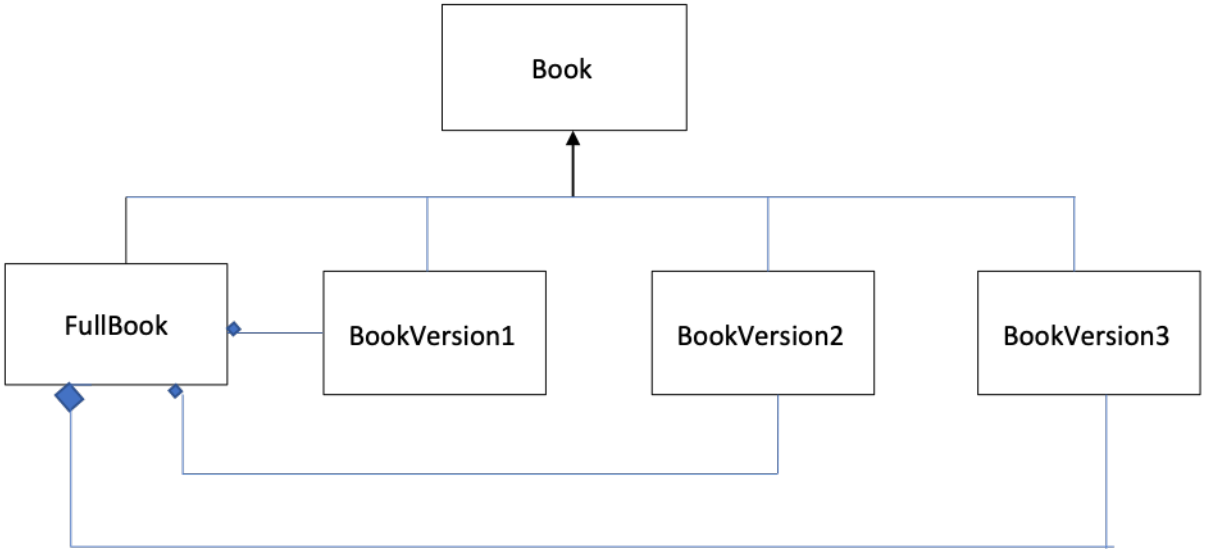


Figure 5.18: A single "entity" can be implemented with multiple rights. Here, `FullBook`, `BookVersion1`, `BookVersion2`,and `BookVersion3` all afford different access privileges. It is assumed that `FullBook` allows "full" rights, whereas the other three have limited access to some of the functionalities of a book. All of them inherit from `Book`, an abstract class. `BookVersion1`, `BookVersion2`,and `BookVersion3` work like object adapters with `FullBook` as the adaptee, but `FullBook` maintains references to each of these three adapters (unlike in the adapter pattern). Note that `FullBook`'s relationship with each of the other sibling classes is shown as composition, so deletion of `FullBook` also implies deletion of the other three.

The Proxy Pattern
A description of what the pattern is and a bit of discussion on Protection Proxy. Under construction; see Wikipedia page on the Proxy Pattern.

Table 5.1: The Proxy Pattern

5.5 Evaluating Quality of the Software Design

In the process of assigning responsibilities, we explored all the choices we could think of, applied the rules of good design, and picked the best. Using this process exclusively may not be wise, since we may fail to see the impact of these design choices on the system as a whole. In this context, it is useful to examine the resulting structure and look for anomalies. Two questions that naturally arise are:

- *How can we recognize poor design choices in a system?*

- *Is there a systematic process for fixing the bad choices*

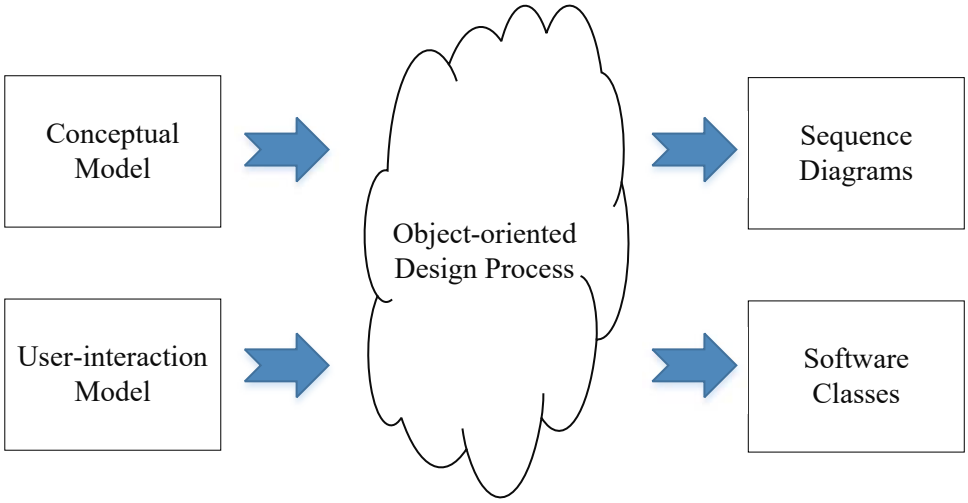
Code smells and **design smells** are terms that are used to refer to structures that appear in a system when design principles are violated. Familiarity with these helps us recognize when a poor design choice has been made. If we look for these smells as we are making the design decisions, correcting them is easier.

Code smells can be classified based on the level at which they are recognized and eliminated. At the highest level, are the smells that are recognized by looking at the entire system. An example of this is the smell *Duplicated Code* which refers to a situation where identical or very similar code is present in multiple locations. At the next level, are the smells identified by looking at an entire class, and its relationship with other classes. The smell *Inappropriate Intimacy* refers to a situation where a class depends on the implementation details of another. When a class uses the features of another class excessively, the resulting smell is called *Feature Envy.* Some smells are recognized at the method level, which is the lowest level at which we recognize smells. An example of these is the smell *Long Method*, which is characterized by a method, function or procedure that has become too long.

Refactoring is defined as the process of improving the internal structure (design and code) of a piece of software without altering the module's external behavior. The process is often applied to a system in production, but we can use this process just as effectively during design and development. Practitioners have developed a set of rules that can be used systematically to refactor code. Once we have identified a bad smell in the code, we apply the appropriate refactoring rules to remove the smell. It is easier to understand these concepts and process through an example, and to that end, we add a new requirement to one of the business processes in our library system.

The Object-Oriented Design Process

The object-oriented design process we have followed in this chapter can be summarized in the figure below:



The inputs to this process are the conceptual model, defined by the conceptual classes and their relationships, and the behavioral model, detailed in the use cases. These artifacts are the result of the object-oriented analysis that we have described earlier in the text.
We start the object-oriented design process by examining the conceptual classes and making decisions on what software classes will be needed. Next, we look at each use case, and decide how the system functionality specified in the use case is to be realized through the software classes. This realization is described through sequence diagrams. It should be noted that this is not a linear process. For instance, as we construct the sequence diagrams, we may discover the need for additional software classes.
Once we have sequence diagrams, we know all the software classes and the methods required in each. This completes the design process.

5.5.1 A New Requirement: Charging Fines for Overdue Books

Consider the situation where the library decides to cut down on truancy by imposing fines. When an overdue book is returned, the librarian would like to know the amount of fine and send out a notice to the user regarding the fine payable. The system should therefore compute the fines and display the relevant information. The resulting changes in the business process are captured in the use case in Figure 5.19.

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

We have the following business rule for computing the fine:

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

Figure 5.19: Use-case Book Return with Fines: If a book is overdue, a fine is calculated and charged to the user.

Rule 7: New books (less than a year old) are charged $0.25 for the first day and $0.10 for every subsequent day. Older books are charged $0.15 cents for the first day and $0.05 for every subsequent day. If a book has a hold on it, the amount of fine is doubled.

5.5.2 The Initial Design

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

Having made this decision, let us take a look at pseudocode for the new `returnBook()`

method in `Library`. As a result of these decisions, `Book` now has an `acquisitionDate` field and an associated accessor. The `returnBook()` method in `Library` involves the following steps:

1. Search the catalog for the book; if not found, exit.
2. From the book, get the member who issued the book; if none, exit.
3. From the book, get the date of acquisition and the due date.
4. Compute the fine(if any) as follows:
   - if (current date is past the book's due date) compute fine as follows:
     - if (365 days have elapsed since the book's acquistion date)
       fine = 0.15 + 0.05 × (days elapsed since due date);
     - otherwise, fine = 0.25 + 0.1 × (days elapsed since due date);
   - if (book has any holds) double the amount of fine.
5. Record that the member has returned the book.
6. If there was a fine, record the information in the member.
7. Check if the book has any holds.
8. Return the results.

The resulting sequence diagram for returning books is shown in Figure 5.20. The `returnBook()` method in `Library` must now check if a fine is involved: if so, it computes the fine and updates the corresponding `member` object by invoking the `addFine()` method. The book's title is also passed, so a transaction can be created to keep a record of the fine.

The `returnBook()` method returns a code that indicates if a fine was involved, so the interface can alert the library clerk. The assumption is that the code in the user interface will take appropriate action to notify the clerk in the above circumstances.

5.5.3 Evaluating and Improving the Solution

In the process of evaluation, we perform two tasks: *recognize poor design* and *identify a process for fixing it.* We begin by noting that our process takes eight steps, which is large in comparison to the designs we had for the other processes. In the jargon of code smells, this is called a Long Method, and we look for the possibility of refactoring it.

When methods or classes become too large, we try to remove parts that do not fit very well with rest of the method or class. The simplest kind of removal is one where we identify a part of a method that can be pulled out as a subprogram. This leads us to our first refactoring rule: Extract Method. Considerations involved in applying this rule and the steps for carrying it out are detailed in Figure 5.21.

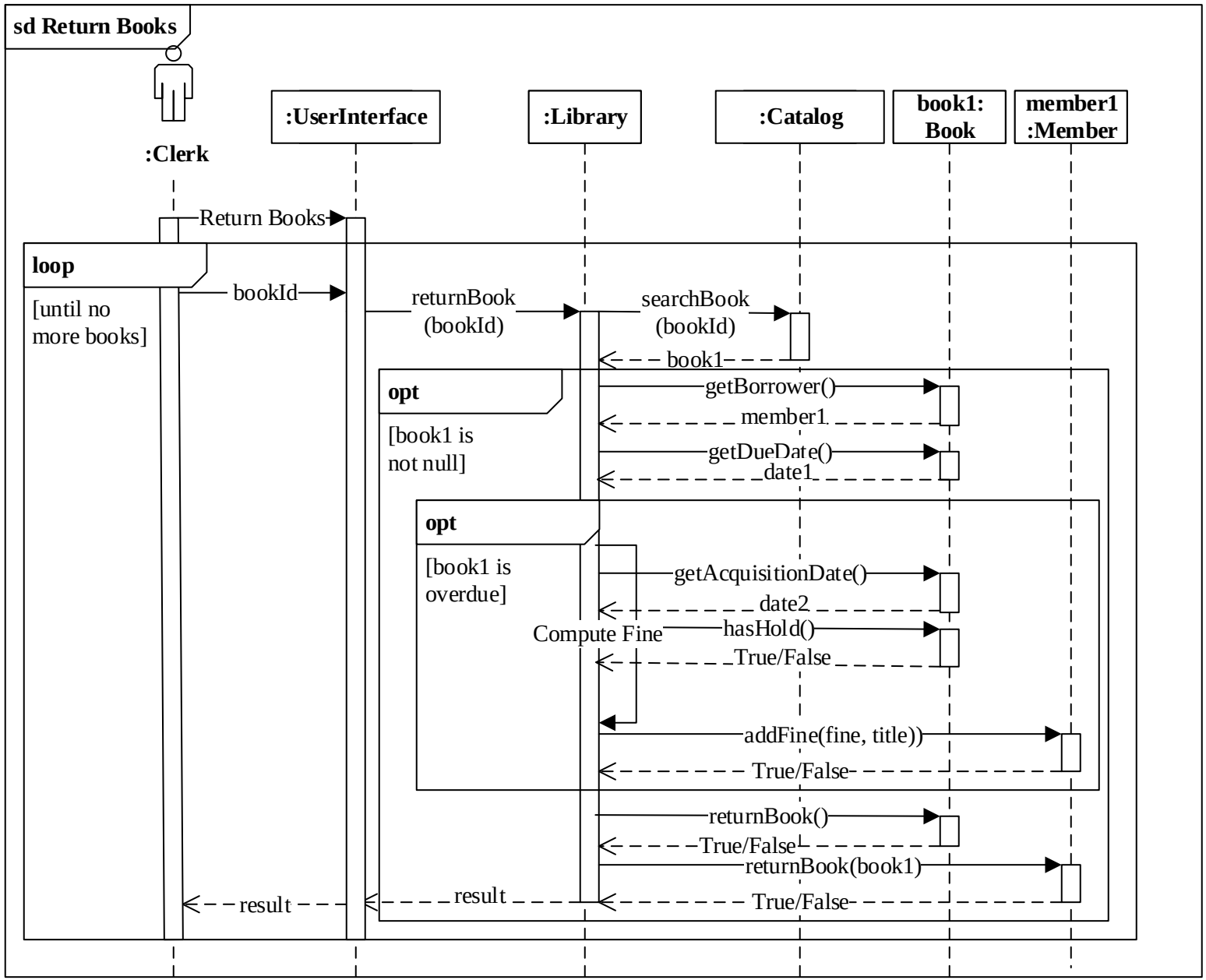


Figure 5.20: Returning a book and checking for fines. If `book` is overdue, `Library` gets the relevant data, computes the fine, and adds the fine to `member`

The first step of the process is key: identifying what can be extracted. In our design for the business process, notice that each of these steps, *except Step 4*, is an application of a single method, which is computed on some object (`catalog`, `book`, or `member`). Step 4, bycontrast, deals with a lot of detail about how the fine is to be computed. This kind of detail is unusual for a method in a Facade, and is causing a significant increase in the size and complexity of the method for returning a book. Hence this step is a good candidate for extraction.

The next part of the process is to figure out how Step 4 will be extracted. The data used in Step 4 is accessed from `book` in Step 3; this data will have to be passed as parameters to the new method. However, we can also see that the data accessed in Step 3 is used only in Step 4; it therefore makes sense to move both these steps and make the `book` object a parameter.

Extract Method Rule *If you have a code fragment that can be grouped together, turn the fragment into a method, and assign it a name that explains the purpose of the method.*
It is easy to recognize these fragments from the comments added by the programmer. These comments, which typically take the form of a verb phrase, also suggest how the extracted method should be named. If a code fragment does not appear to have a simple name, it is often unlikely to be a good candidate for extraction into a method. Another indicator is the number of local variables that are modified; if the code fragment modifies only one variable, this strengthens the case for extraction. If a large number of variables are modified, the code fragment should probably be left in place.
The steps involved in applying this rule are as follows:

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

Figure 5.21: Principles and process for extracting a method from a longer method.

The two steps together perform the task of computing the amount of time. We therefore extract this code, and create the method `computeFine()` with `Book` as a parameter. This method will return the amount of fine. The new `returnBook()` method is as follows:

1. Search the catalog for the book; if not found, exit.

2. From the book, get the member who issued the book; if none, exit.
3. Call the `computeFine()` method to compute the fine.
4. Record that the member has returned the book.
5. If there was a fine, record the information in the member.
6. Check if the book has any holds.
7. Return the results.

The method `returnBook()` looks much cleaner now. All it is doing is getting the relevant information by invoking appropriate methods and then compiling all the results. The `computeFine()` method does the following:

1. Get the due date of the book.
2. Get the date of acquisition of the book.
3. Compute the fine(if any) as follows:
   - if the current date is past the book's due date compute fine as follows:
     - if 365 days have elapsed since the book's acquistion date
       fine = 0.15 + 0.05 × (days elapsed since due date);
     - otherwise, fine = 0.25 + 0.1 × (days elapsed since due date);
   - If the book has any holds double the amount of fine.
4. Return the amount of fine.

Let us take a closer look at the method that we have extracted. The logic employed by `computeFine()` involves examining the fields of `Book` and making decisions based on the values stored in these fields. To get these values, the method repeatedly invokes the accessor methods of book. In the jargon of code smells, this is referred to as Inappropriate Intimacy between the `Library` class and the `Book` class. This leads to tighter coupling between the classes.

One of the rules of good object-oriented design is called the Law of Inversion, which says that

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

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

The above design principle leads us to the next refactoring rule, Move Method. The `computeFine()` method is moved from `Library` to `Book` using the principles set forth in Figure 5.22. This process has helped resolve our dilemma about where the fine should be computed. The resulting sequence diagram is shown in 5.23.

Move Method Rule If we have a method that is using more features of another class than the class on which it is defined, then the method needs to be moved to the class whose features it is using the most.
This rule is a manifestation of the process of assigning responsibilities to the appropriate class and is perhaps the most frequently applied rule in refactoring. When a method uses too many features of another class, we have a situation where the classes are either collaborating too much or are too tightly coupled. It is not always the case that such a problem will be resolved by moving a method. Sometimes, other patterns may have to be applied that allows objects to communicate without getting too entangled in each other's methods. The simplest and most obvious situation is when a method accesses several fields of another class and almost all its computation is done on these fields.
The steps involved in applying this rule are as follows:

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

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

Figure 5.22: Principles and practices for moving a method from one class to another.

In the initial stages of design, we need not go through the entire process of refactoring to correct our errors. Nonetheless, beginners may often find themselves in a quandary as to where the responsibilities for a certain task should be placed. The exercise of refactoring helps to formalize some of the basic principles of object-oriented design so that such errors can be caught early in the design process and suitably corrected.

5.6 Discussion and Further Reading

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 may appear troublesome. 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 at first. Indeed, the only way these can be mastered is through repeated application and critical examination of the designs produced. Perhaps needless to say, it is also extremely useful to discuss design issues with more experienced colleagues.

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 several student projects were presented. To maximize 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.

5.6.1 Conceptual Classes and Software Classes

Finding the classes is a critical step in the object-oriented methodology. 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.

In the design phase we deal *software* classes, i.e., classes which can be built using typical programming languages. We need to deal with the following issues:

- how conceptual classes will be manifested in the software
- what additional classes will be needed to build such a system in a typical programming language

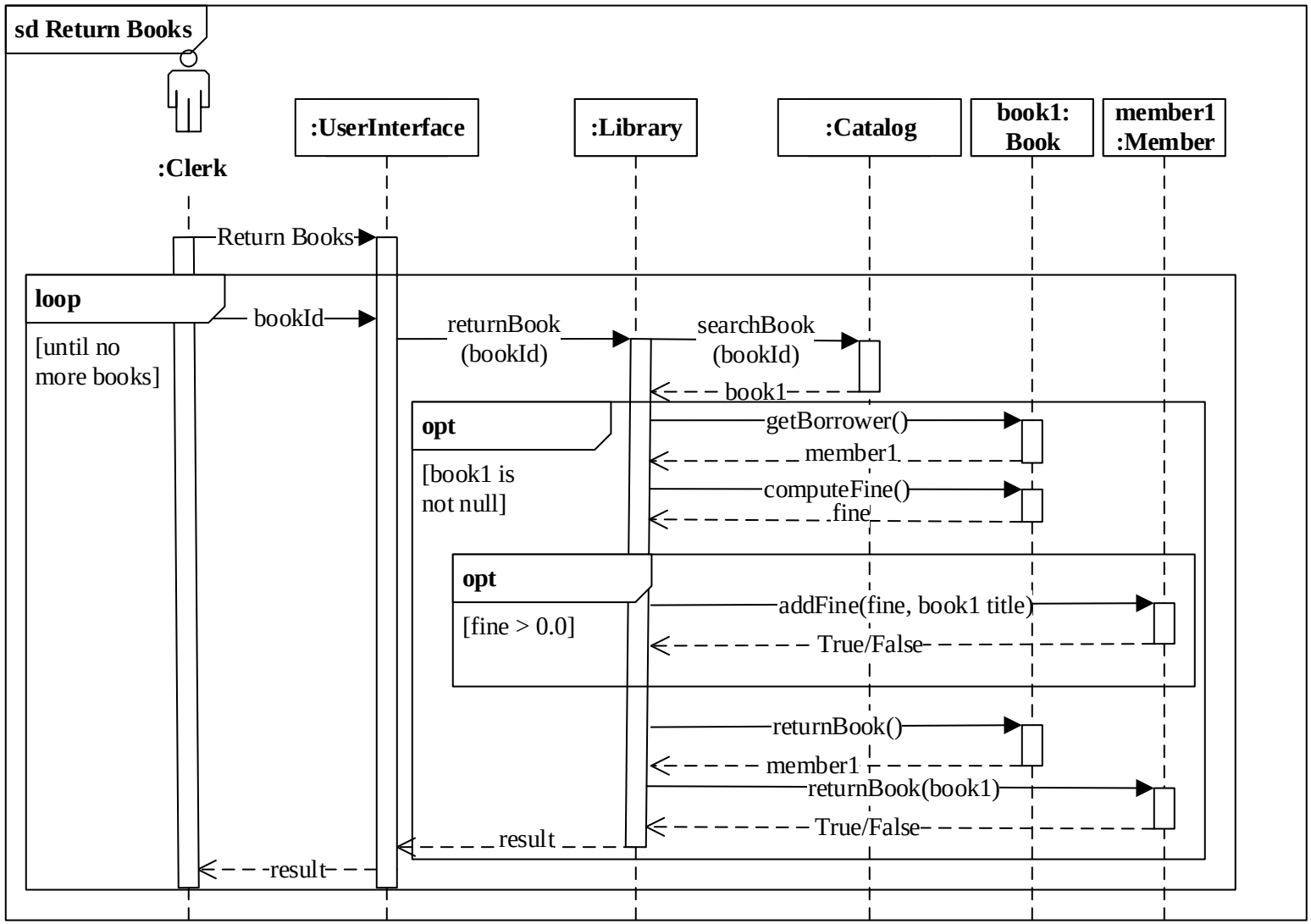


Figure 5.23: Refactored process for returning a book and checking for fines. `Library` invokes a method provided by `Book` for computing fine, and then invokes a method in `Member` to add the fine.

We identify the conceptual classes that will become software classes, the conceptual classes that are dropped, and the additional classes to be added. For the software classes we identify all the fields, and the methods and their parameters. In the library case study, we added the collection classes `Catalog` and `MemberList`, and a `Transaction` class as new software classes. The conceptual class `Borrows` was not included in the set of software classes. Methods were identified when we created the sequence diagrams.

5.6.2 The Single Responsibility Principle

The Single Responsibility Principle captures a lot of the the essence of the ideas of cohesion and coupling. It was coined by Robert C. Martin [5] in 2005, stated as follows:

Each software module should have one and only one reason to change.

He revisited it in a 2014 blog post, attributing its inception to a 1972 article [8] by David L. Parnas, wherein the author compared two different strategies for decomposing and separating the logic in a simple algorithm. One of the conclusions of Parnas' article is as follows:

> We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

Starting with this, Martin leads up to the Single Responsibility Principle, restating it as:

Gather together the things that change for the same reasons. Separate those things that change for different reasons.

The above approach looks at a construction of a module as a process of grouping together the related responsibilities. In the software development process used in this text, the software modules are identified based on the conceptual model that came from our analysis. We are therefore looking at responsibilities from the other end: *given a responsibility, identify the best module.*

Most of the design decisions made in this chapter can be justified from this point of view. The chosen module in each case, is the one that does not violate the Single Responsibility Principle; in case there is a responsibility that does not fit well into any module, we either add another module(class), or revisit our conceptual model. Consider for instance, our initial choice of software classes, where we decided to add modules for `Catalog` and `MemberList`. If we had decided instead, to store references to the collections within `Library`, we can identify two different reasons for changing `Library`:

- modifications to operations provided by the `Library` subsystem
- using a different data structure for storing the collections

Clearly, this violates the Single Responsibility Principle, necessitating the addition of separate modules for the collections.

5.6.3 Recognizing Bad Smells and Design Flaws in Object-oriented Programs

Software design problems have been studied under different terms, such as code smells, flaws, non-compliance to design principles, violation of heuristics, excessive metric values, and anti-patterns. They can be identified in many different stages of the software lifecycle, and can usually be removed by applying an appropriate refactoring. Several aspects of quality, such as maintainability, comprehensibility, and reusability, are often improved as consequence of correcting the flaws.

Attempts have been made to classify the code smells based on the nature of their appearance. Since there are a large number of smells, such a classification can help identify them in different settings. The Long Method smell has been grouped along with some others as a "bloater," i.e., something that causes a method or class to become unnecessarily large. Smells like Inappropriate Intimacy and Feature Envy are classified as "couplers," due to their tendency to increase coupling. The classification can also help identify the kind of refactoring: bloaters require part of the method or class to be extracted, and couplers require part(s) of the class to be moved elsewhere.

Some attempts have been made to automate the process of finding and correcting design flaws. JDeodorant is an Eclipse plug-in that identifies design problems in software, known as bad smells, and resolves them by applying appropriate refactorings. JDeodorant employs a variety of novel methods and techniques in order to identify code smells and suggest the appropriate refactorings that resolve them. For the moment, the tool identifies five kinds of bad smells, namely Feature Envy, Type Checking, Long Method, God Class, and Duplicated Code.

5.6.4 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.* 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 a later chapter.

- *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. We shall see how a GUI can be modeled as a multi-panel interactive system, and how such features can be incorporated later on.

- *Allowing remote access.* Now-a-days most systems of this kind allow remote access to the server. Technologies can be incorporated into the UI to accomplish this, but are beyond the scope of the this text.

It should be noted that in practice several of the non-functional requirements would actually be provided by a database management system. 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.

5.6.5 Further Reading

The book by Meyer [7] 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. [3] to get exposure to the common design patterns. There are hundreds of other lesser patterns and a catalog of these can be found in [9, 10].

The book by Fowler [2] is the main reference for refactoring. Among other things, the book emphasizes the importance of the role that refactoring can play in keeping a system from falling into decay. While the benefits of refactoring are many, there are also a few caveats one should follow to avoid going overboard, and there are also situations and systems whose characteristics make refactoring difficult. The reader would be well advised to engage in a deeper study of this process before attempting a wider application.

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

happens in practice and has the added advantage of giving the designers some flexibility in the choices that they make.

The notions of code smells and design smells were first introduced by Kent Beck in 1990's. The book by Martin [6] lists several bad smells and case studies explaining how these can be detected and removed. The research in [4] attempts a classification of code smells, and [1] discusses how bad smells evolve throughout the software lifecycle.

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. Create a table similar to 5.13 showing all the methods for the `Book` class.

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

9. (Discussion) Instead of having `renew()` methods, would it be simpler to return the book and issue it again? What problems might this cause?

10. Consider the use case in a university registration system where a student drops a course. The amount of refund to be credited to the student can depend on several factors:

    - a student can always drop a course for a full refund within 24 hours of registration
    - the amount of refund depends on the current date and the starting date of the course

    In which class should the responsibility of computing the amount of refund be placed, given that both the student and section are involved in this process? Write the detailed steps in the process assuming that the `RegistrationSystem` (the facade) is responsible. Look for bad smells and any possible refactoring(s) to remove them.

11. Consider a use case in a warehouse system where a customer is returning an item to the warehouse. The amount of refund could depend on the following:

    - warehouse return policy and stocking fee for the item
    - the number of days since the purchase

    In which class should the responsibility of computing the amount of refund be placed, given that both the product and the transaction may be involved in this process? Write the detailed steps in the process assuming that the `WarehouseSystem` (the facade) is responsible. Look for bad smells and any possible refactoring(s) to remove them.

12. Consider a use case in an airline reservation system where a customer is canceling a reservation. The amount of refund could depend on the following:

    - full refund for any cancellation within 24 hours
    - amount of refund depend on the kind of reservation
    - nature of refund (credit towards next reservation vs. cash) depends on the kind of reservation

    In which class should the responsibility of computing the amount of refund be placed, given that result depends on the kind of reservation but may also affect the passenger? Write the detailed steps in the process assuming that the `ReservationSystem` (the facade) is responsible. Look for bad smells and any possible refactoring(s) to remove them.

13. When we added the a book or a member to the library system, the responsibility for invoking the constructor was assigned to `Library` and not to the respective collection classes. Explain why this is consistent with the Single Responsibility Principle.

14. When we implemented the `searchMember()` method in `Library`, we rejected the idea of getting an enumeration of the members for the collection and doing the search operation in `Library`. Explain how this approach would be a violation of the Single Responsibility Principle.

15. Re-examine the Member Transactions query in the light of the Single Responsibility Principle. We used the fact that query variations could be easily accommodated, to support using Option 2. The query variation can be viewed as a reason for change. Which classes are affected by the change in the query, under each of the two options? Is this consistent with the requirement that "each software module should have one and only one reason to change."

Bibliography

[1] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of code smells in object-oriented systems. *Innov. Syst. Softw. Eng.*, 10(1):3–18, Mar. 2014.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, 1994.

[4] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 381–384, Sept 2003.

[5] R. C. Martin. The principles of ood. 2005.

[6] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[7] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[8] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[9] L. Rising. *The Pattern Almanac.* Addison-Wesley, 2000.

[10] J. M. Vlissides, J. O. Coplien, and N. L. Kerth. *Pattern Languages of Program Design 2 (Software Patterns Series).* Addison-Wesley, 1999.

Index

Class
    Software Class, 7
Cohesion, 19
Coupling, 19

Design, 5
    Issues, 5
    Subsystems, 6
Design Patterns
    Facade, 7

Facade Pattern, 7

Law of Inversion, 48

Mutable Object, 35

Refactoring, 5
    Extract Method, 47
    Move Method, 49

Sequence diagrams, 10
Software class, 7
SRP, Single Responsibility Principle, 52

UML
    Sequence Diagrams, 10