Chapter 13
The Unified Modelling Language

Throughout this book we have used several kinds of UML diagrams to describe various aspects of the system under consideration. UML is a complex language and it is hard to grasp all the details at the beginning. In this chapter, we re-visit UML to gain a broader perspective that will enable us to use it more effectively. Also, we take a closer look at the UML diagrams that were not dealt with in-depth in the earlier chapters.

The OMG specification states:

> The Unified Modelling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artefacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

The important point to note here is that UML is a "language" for specifying, and not a method or procedure. The UML is used to define a software system, to detail the artefacts in the system, to document and construct—it is the language that the blueprint is written in. The UML may be used in a variety of ways to support a software development methodology (such as the Rational Unified Process), but in itself it does not specify that methodology or process.

UML can be seen as a mechanism for defining the notation and semantics for several kinds of models (or domains). Although the exact domain under which a particular UML diagram falls can be a matter of discussion, we can roughly categorise the diagrams among the following domains:

1. User Interface Model: Describes the boundary and interaction between the system and users. Corresponds in some respects to a requirements model.
2. Interaction or Communication Model: Describes how objects in the system will interact with each other to get work done.

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_13

3. State or Dynamic Model: State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflows the system will implement.
4. Logical or Class Model: Describes the classes and objects that will make up the system.
5. Physical Component Model: Describes the software (and sometimes hardware components) that make up the system.
6. Physical Deployment Model: Describes the physical architecture and the deployment of components on that hardware architecture.

The semantics of these domains are such that each of these domains presents a different view of the system. A system design methodology often uses only a subset of the domains, and exactly which ones are used is usually decided by the people in charge of a project. To make such a decision, it is imperative to know what kind of a model each domain gives us, so that a clear communication occurs between all those involved in the project.

During the various stages of the creation of software, we need different kinds of models. When the system is initially being defined, we rely on the User Interface Model. The UML diagram used for this is the use-case diagram, which shows the interaction between the actors and the various use cases. This was discussed in Chaps. 2 and 6.

The Interaction Model is useful when we want to flesh out the details of the all the required behaviours. The interaction model describes how objects in the system will interact (collaborate) with each other to get work done. In the context of UML, a *collaboration* is a description of a collection of objects that interact to implement some behaviour within a context. Collaboration diagrams are created to represent how these classes collaborate to provide specific behaviours. There are 3 kinds of diagrams that model the collaboration (i.e., interaction and communication) between objects: **sequence diagrams, communication diagrams**, and **timing diagrams**.

Sequence diagrams were dealt with extensively in Chap. 7, where we used them to describe in detail how the software classes would interact to achieve the required functionality for our Library system. We do not address them any further here. **Communication diagrams** are used show the message flow between objects in an application and also imply the basic associations (relationships) between classes. When showing the associations between classes, we could have a *static* description which shows only the classes and the roles played by members of each class. Such a diagram is called a **specification-level communication diagram.** In a *dynamic* description, we show the messages that flow between the instances of the collaborating classes, and hence these are called **instance-level communication diagrams. Timing diagrams** are a type of interaction diagram, where the focus is on timing constraints and are used to explore the behaviours of objects throughout a given period of time. Later in this chapter, communication diagrams and timing diagrams will be used to describe some aspects of the projects discussed in this book.

Statecharts and **activity diagrams** are two ways in which the dynamic model can be captured. Statecharts describe finite state machines that describe the various states of a component of the system; these were dealt with in Chap. 10. Activity diagrams, which represent the flow of control from one activity to another, are discussed next. A more recent addition to the UML family are **interaction overview diagrams**. These focus on the overview of the flow of control of the interactions. They are a variant of the activity diagram, where the nodes are the interactions or interaction occurrences. They are useful to understand how the interaction occurrences come together in defining the system and are therefore a part of the dynamic model of a system.

The logical model and the component models tell us what software entities have to be created. We discussed **class diagrams** in Chap. 7. The additional diagrams for representing these models are **component diagram**, **composite structure diagram, package diagram** and the **object diagram**.

Finally, we discuss the **deployment diagram**, which describes the deployment of the components on the hardware.

13.1 Communication Diagrams

As already stated, communication diagrams *(formerly referred to as collaboration diagrams)* are used show the message flow between objects in an application, implying the basic associations (relationships) between classes.

A **specification-level communication diagram** is a *static* description which shows only the classes and the roles played by members of each class, while an **Instance-level Communication Diagram** is a *dynamic* description which shows the messages that flow between the instances of the collaborating classes. (It should be noted that instance-level diagrams are very similar to sequence diagrams and it would be very unusual if a design of a project provides descriptions using both these kinds of diagrams.)

13.1.1 Specification-Level Communication Diagrams

The model that we created in Chap. 6 consisted of a set of use cases. Each use-case represents a scenario and each class has some role to play in the scenario. As an example, consider the scenario of checking out a book to a user. This scenario is used multiple times in the use case Checkout Books. The entities (software classes) that play a role are the MemberList, Catalog, Member and Book. The role to be played by the objects of a class in a particular scenario is called a *ClassifierRole*. Quite often, particularly in less complex systems, the ClassifierRole for a class cannot be distinguished from the class itself. Thus the ClassifierRole for the Book class is Book, the ClassifierRole for the Member class is Member, etc.

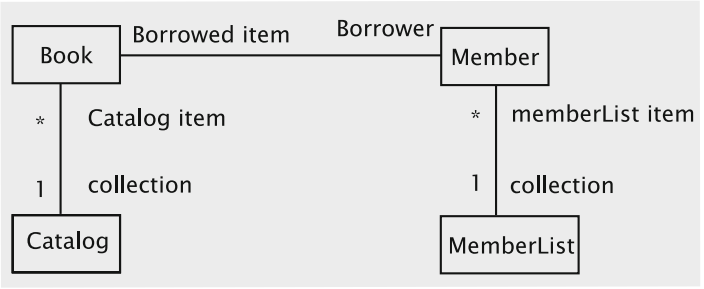

Fig. 13.1 Specification-level diagram for Book Checkout

Within a scenario, the specific role that a class has with respect to another class is called an *AssociationRole.* The association between the two classes is then termed as a *Link*. If two boxes are connected by a link, they form a *collaborating pair.* Each entity in the pair plays a particular role in the link (different from its ClassifierRole); this is the *AssociationRole* of the entity. In Fig. 13.1, there is a link between Book and Catalog. Catalog is merely a collection of Books and so the AssociationRoles are just "collection" and "catalog item" respectively. In the link between Book and Member, the AssociationRoles are "borrowed item" and "borrower" respectively. Thus the class "Book" has the ClassifierRole "Book" with two AssociationRoles, viz., "catalog item" and "borrowed item".

Note that this scenario corresponds to the use-case for checking out books. The diagram explains which classes in the Library subsystem collaborate when a book has to be checked out and defines the nature of the relationship (collaboration) between all pairs of these classes. However, the specification-level diagram does not give us the details of actions taken by each of the objects, i.e., it does not tell us *how* the collaboration occurs.

A second example involving the drawing program from Chap. 11 is shown in Fig. 13.2. This diagram shows how the classes collaborate to create a circle. The GUI transmits the user requests to the `CircleCommand`. The label `/CircleCommand:Command` denotes that `CircleCommand` is an entity of type `Command`. The `CircleCommand` plays the role of the request processor in this association. The `CircleCommand` then acts as a creator to create the `Circle`, which is an entity of type `Item`. Once the Item has come into existence, it has to be drawn. This is shown in the association between the `Circle` and the `UIContext`. In our case, the particular `UIContext` is the `SwingUI`, and this is represented by the label. In the resulting association, the `Circle` is a drawable item and the `UIContext` is a drawing tool. Finally, we have an association between the `UIContext` and the GUI, in which the GUI behaves as the drawing medium.

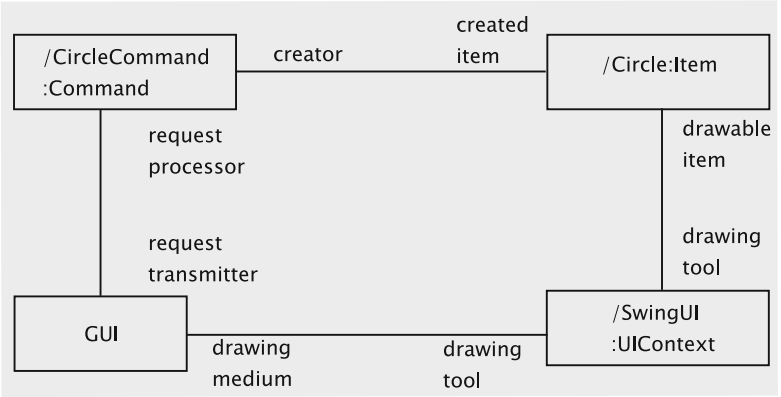

Fig. 13.2 Specification-level diagram for drawing a circle

In our examples, the role played by any participating class (the ClassifierRole) does not change; we can say that the ClassifierRole is synonymous with the class itself. In more complex systems, a class may play multiple ClassifierRoles. In diagrams where two ClassifierRoles are associated with the same class, the ClassifierRoles should be distinct in terms of their requested operations and the values of attributes.

13.1.2 Instance-Level Communication Diagrams

In an instance-level diagram, we show how the instances of the classes collaborate with each other to complete the task. As one would expect, the collaboration takes place by passing messages, i.e., methods of one class being invoked by the other. This in turn implies that the links now have arrows on them; the arrow points from class that invokes the method towards the class whose method is being invoked. The label on each link describes the name of the method, the parameters passed and the result. In addition, we have a unique number next to each message; these numbers indicate the order in which the messages are sent.

Figure 13.3 shows how this would be used to represent the checking out of a book. When the Library receives a request to checkout a book, it first sends a search message to MemberList, which returns a reference to the Member object. Next it sends a search message to Catalog, which returns a reference to the Book object. Next it sends the issueBook message to Member, and finally the issue message to the Book.

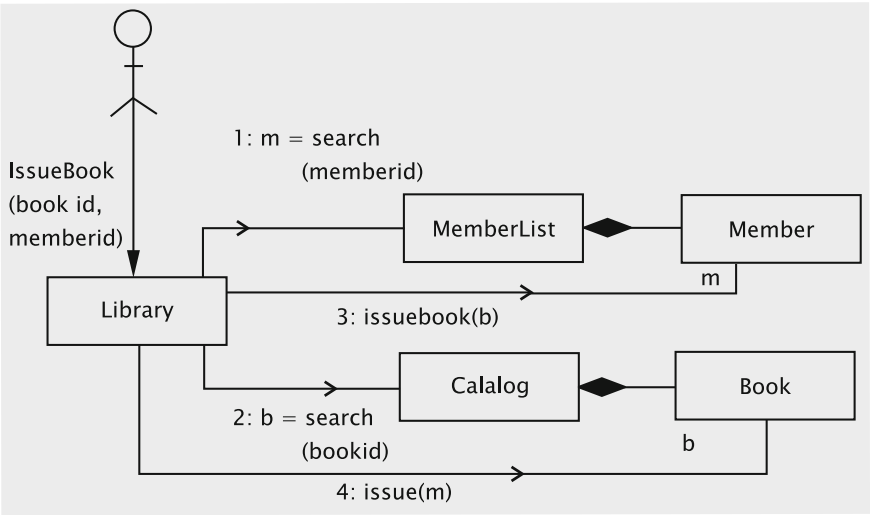

Fig. 13.3 Instance-level diagram for Book Checkout

13.2 Timing Diagrams

UML timing diagrams describe two system characteristics:

- the change in state or value of one or more elements over time
- the interaction between timed events and the time and duration constraints that govern them.

When both the characteristics are juxtaposed we get a picture of how the interaction between the objects is connected to the change in the objects' state. The state change can happen due to external events and trigger interaction with other objects, or the state change may occur due to a message being received from another object. Thus the timing diagram can provide a mechanism for understanding the internal dynamics of the system.

There are two kinds of figures that are used to describe this: the **Value Lifeline** and the **State Lifeline.** The Value Lifeline shows the change in value over time, whereas the State Lifeline shows the change in state.

As a simple example, consider a card-swipe system that provides access through a door. To gain access a user (card-holder) swipes the card. There is a wait period when the system processes the card and then a period when the access is granted. The user is thus in one of three possible states: *Idle, Waiting* and *Has access.* The system can be modelled as being in one of two states: *Idle* and *Activated.* When the card is swiped, the system goes from Idle to the Activated state. When the card is validated, a event is triggered that allows access, and after a short interval a second event is triggered that removes access and the system returns to the Idle state.

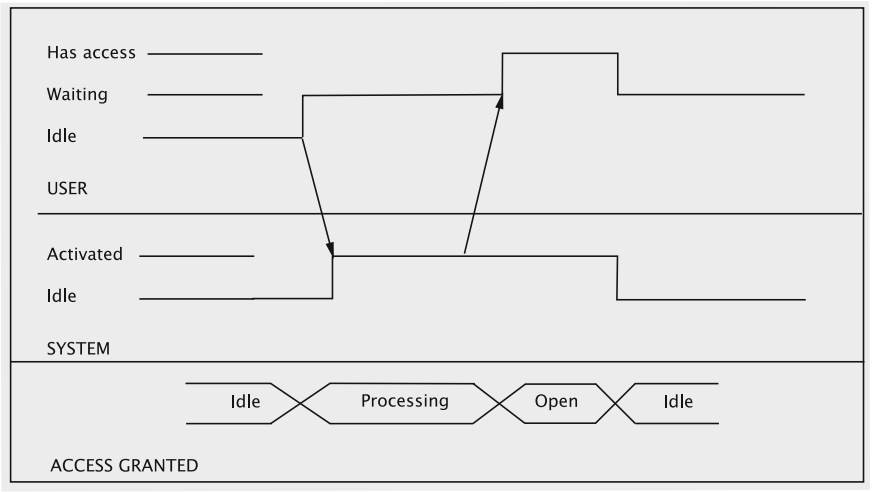

Fig. 13.4 Timing diagram for a card-swipe

All of this can be expressed by stacking the state lifelines for the user and the system as shown in Fig. 13.4. The horizontal segments of these lifelines represent the intervals when the system stays in a particular state and vertical segments denote transitions between states. The third lifeline is a value lifeline which is describing the status of the transaction. Each hexagon corresponds to the system staying in a particular state. The length of the hexagon denotes the length of time the system was in that state; within each hexagon we write the name of the state and any associated values.

In the figure the first event that occurs is that the User swipes the card, and goes from the Idle state to the Waiting state. A message is sent to the System which then enters the Activated state. In this system, we allow for the possibility of a small delay between when the user swipes the card and when the System enters the Activated state; this is indicated by the arrow not being vertical. For the transaction as a whole, however, as soon as the card is swiped, the system enters the Processing state. After some processing, the System sends a message to the user, presumably by unlocking a door, and there is a small associated delay. The transaction enters the Open state only when the User gets the message. After a brief period, we assume that the door will get locked again. At this point both the System and the User become Idle and the transaction is assumed to be complete.

For the Library system (from Chap. 6), consider the following sequence of events:

> *user1 checks out book; user2 places hold, user3 places hold, user1 returns book, user2 is informed, user2 fails to show up in allotted time, user3 is informed, user3 checks out book.*

A Book object can be in one of several states: (i) *on shelf*, (ii) *issued*, (iii) *issued and on-hold*, (iv) *on-hold and available*. Clearly, transition from one state to another is triggered by external events. When the book is on hold, we have an additional parameter—the number of holds. We would prefer a value lifeline to represent the book because it allows us to represent an additional value. With this parameter, we can now reduce to two possible states: *Available*(*holds*) and *Issued*(*holds*). Available(0) would mean the book is on the shelf, whereas Issued(0) would represent the *issued* state.

The user can be in one of 4 states: *Idle, Book, Wait* and *CanGet*. The system (Library system) can be in one of two states: *Idle* and *Activated.* In the Activated state, the system sends messages to the users triggering state changes. The actions taken by the users cause changes in the state of the book and the system.

When we stack the state lifelines for the users and the system and the value lifeline for the book, we get the result shown in Fig. 13.5.

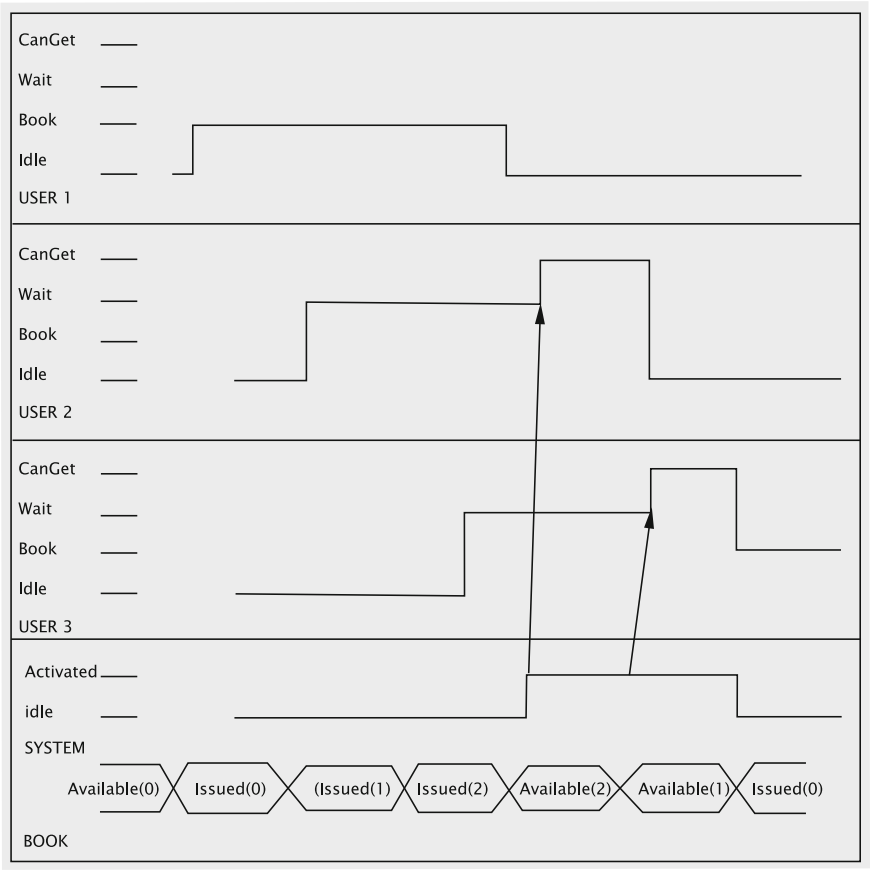

Fig. 13.5 Timing diagram for library transactions

When User1 checks out the book, he/she goes from Idle to Book. When User2 and User3 place holds, they enter the Wait state. The Book object accordingly goes from the Available(0) state to the Issued(2) state as a result of these events. The System remains Idle until the book is returned. At this point Book enters the Available(2) state, and System becomes activated and sends a message to User2. We are assuming some delay in the sending of the message, indicated by the arrow not being vertical. User2 changes state only upon receiving the message. Since User2 does not show up, the System remains Activated and after a while sends a message to User3. If User2 had checked out the book, the System would have gone back to the Idle state and no further messages would have been sent. The System returns to the Idle state concurrently with User3 checking out the book.

This scenario is centred around a Book. A different scenario can be created that is centred around a User. The state of a User can be characterised by the books that they have checked out, the books on which they have placed a hold on and the books that are currently waiting for them at the library. This information will be captured in a value lifeline. Each book on which a hold has been placed is either Issued or Available. If the user checks out the book when it becomes Available, the book gets added to the list of books checked out by the user. If the user does not check out in the allotted time, book is dropped from the list. Constructing such a timing diagram is not of much use since it does not capture the interaction between the entities but merely logs the activities of the User.

For the example of the microwave in Chap. 10, the communication between the GUI, the Microwave and the Clock can be captured in a timing diagram. The Microwave can be in one of 3 states: *Cooking, DoorOpen* and *Idle*. In the GUI, we assume that the cook button is disabled when the door is open; this implies that the GUI can be modeled as being in one of two states: *DoorOpen* and *DoorClosed*. In the DoorOpen state the GUI does send any signals to the Microwave; when the door is closed, a signal is immediately sent to the microwave. (On the other hand, if the cook button is not disabled, we have a model where the GUI is always in the same state and passes all signals to the microwave.) The clock is represented by an alternating sequence of pulses, and a message is sent at the leading edge. Once again, there is a possibility of a small delay in the arrival of the message from the clock, which is indicated by the arrows not being vertical (Fig. 13.6).

The system as a whole is represented using a value lifeline, since this allows us to keep track of the remaining time. *Cook(n)* represents the situation where we have *n* seconds of cooking time left; the `doorClosedIdle` is represented as *Cook(0)*. *DoorOpen* represents the situation where the door is open.

Figure 13.6 shows how these lifelines can be stacked for a small window of time. At the start of the window, the microwave is cooking with two seconds left. The following events occur: *timer runs out, door is opened, door is closed, cook button pressed, door opened after two seconds, door closed.*

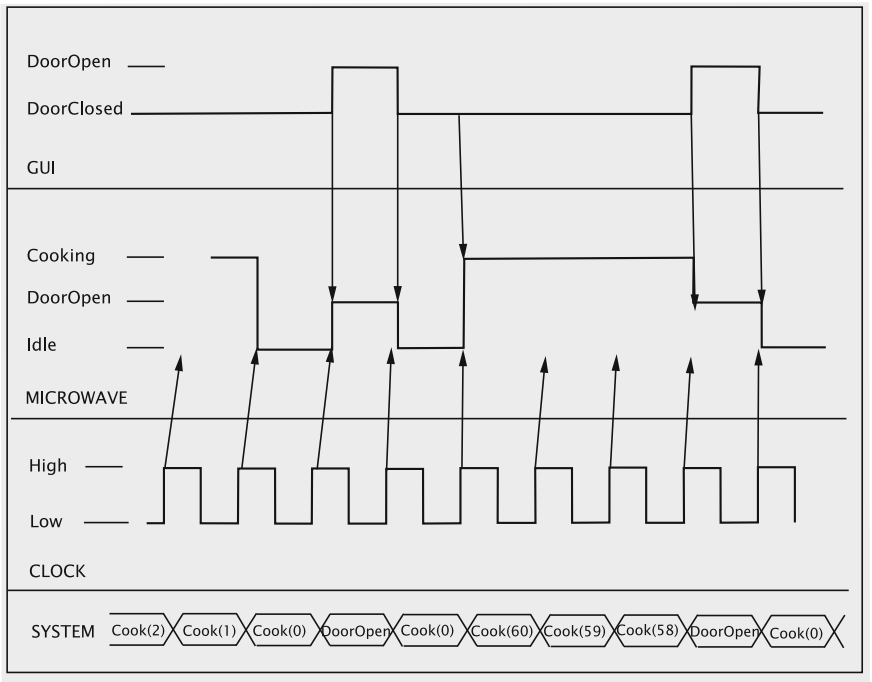

Fig. 13.6 Timing diagram for the microwave

13.3 Activity Diagrams

Activity diagrams show the flow of control from one activity to another. This flow may happen sequentially or concurrently. Each activity is one of the steps in a larger operation; the diagram as a whole describes how the operation is completed.

Figure 13.7 shows an activity diagram for checking out a book from our Library. The activities to be completed are accepting the ID, verifying the ID, accepting the book, checking the issuability of the book, stamping the due date and handing the book to the member. These are the activities listed/implied in the detailed use case for Book Checkout. Note that the sequence diagram for Book Checkout also shows a similar flow of control; *in the case of sequence diagrams, however, the flow of control is from object to object.*

The UML notation also allows us to show concurrent activities using **fork** and **join** operations. Usually, the concurrent activities are performed by different players and it is meaningful to use **swimlanes** to depict these. Each player has a dedicated swimlane as shown in Fig. 13.8. Here we have dropped the branches for `invalid ID` and `unsuccessful checkout` to reduce the clutter. The member is expected to wait

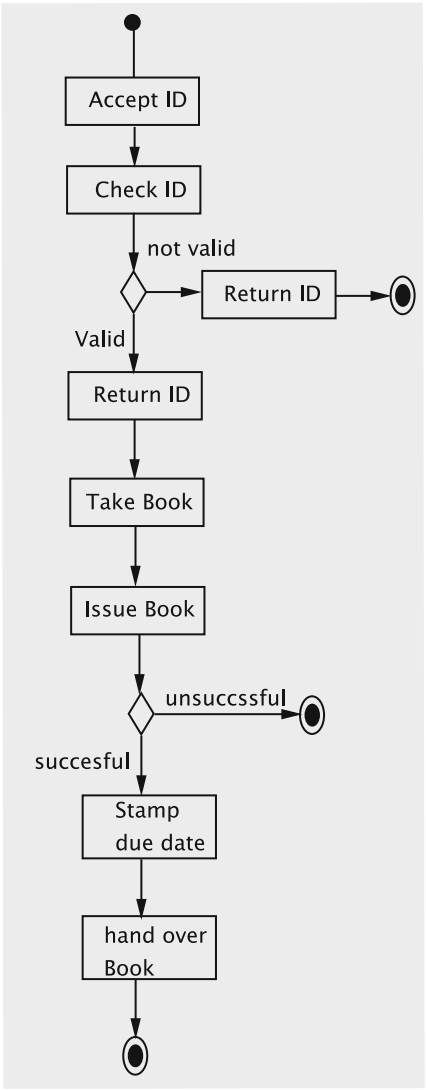

Fig. 13.7 Activity diagram for Book Checkout

when the clerk verifies the ID and also when the book is being checked out. These can therefore be treated as concurrent activities. The entire process begins with the member presenting the ID and this is followed by a fork operation to denote the concurrent activities of the two players. Likewise, the two players must synchronise again when the verification is complete and this synchronisation is represented by a join.

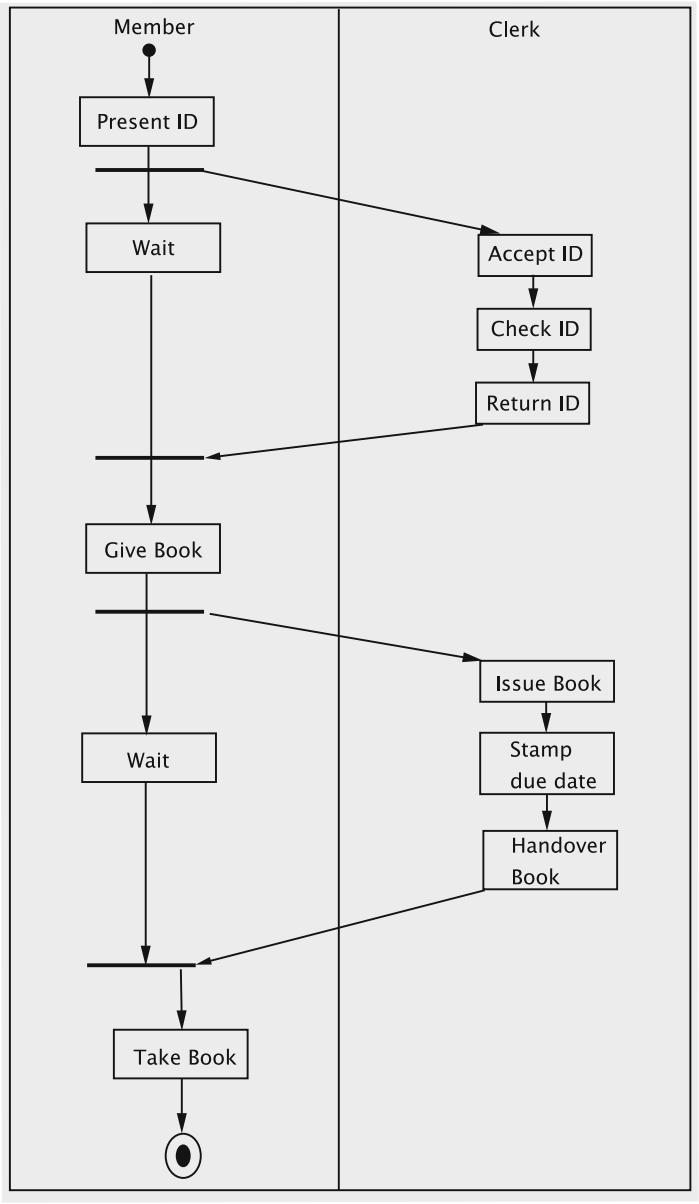

Fig. 13.8 Activity diagram showing swimlanes

13.4 Interaction Overview Diagrams

The interaction overview diagram consists of nodes connected by directed arcs. Each node represents an *interaction occurrence*, which could be represented by a sequence diagram or a communication diagram. Each node has one incoming arc and one outgoing arc. Diamond boxes are used to indicate a branch or a merge; whenever a branch is employed there is an associated Boolean condition.

Consider an extension of our Library case study that allows two additional use cases: *Search for a book by title* that takes a title form the actor and determines if a book with that title is in the catalog, and *Check book availability* that takes a user and a book and determines if the book can be issued to the user at that time. The associated sequence diagrams can be easily constructed and are left as an exercise.

We can now construct an interaction overview diagram that captures a user's attempt to check out a desired book. The details of this are shown in Fig. 13.9. Each of the boxes marked **sd** represent sequence diagrams for the operation named in the

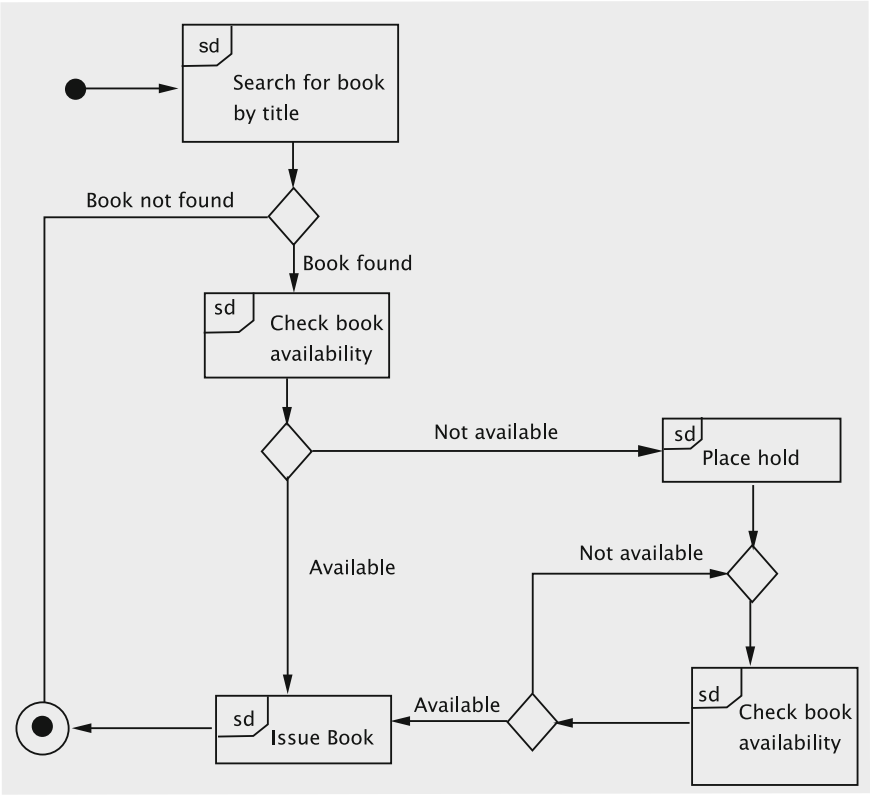

Fig. 13.9 Interaction overview diagram for a Book Checkout

box. (The boxes are sometimes marked **ref** to indicate that it could be any kind of interaction diagram.) If the book is found but not available, a hold is placed and the user waits until the book is available. The wait is represented by a loop, where the user repeatedly checks the book availability.

13.5 Component Diagrams

Simply stated, a component is any module that is a part of a system and has two properties:

1. It encapsulates its state
2. It exhibits behaviour that can be concisely described.

As a simple example of a component, consider the `MemberList` class in the Library system we studied in Chaps. 6 through 9. It obviously encapsulates its state: none of the fields are exposed to the outside world, and an instance of it can be queried or manipulated only via the methods. The behaviour is quite easy to specify: the class behaves as a collection of `Member` objects.

A natural question arises in this context: When can a class be considered a component? The two requirements given above are not sufficient to answer this question. After all, classes usually encapsulate the data and the issue of whether the behaviour can be concisely specified is subjective in nature. To help us narrow the choice, we employ two more constraints. If neither of these applies, the class is not a component.

(i) A class could be considered a component if one could envisage multiple implementations for its functionality.
(ii) A class could be treated as a component is if it can be reused in multiple applications.

In this light, consider the class `Member` in the library system. While there is almost always more than one way to write code, it is a bit of a stretch to imagine significantly different implementations of the class `Member`. Nor is it reasonable to assume that `Member` could be used in a different scenario. Because of these, it seems inappropriate to consider `Member` as a component, whereas `MemberList` could be implemented in clearly different ways such as arrays or linked lists or trees, making it a component. Similarly, `Book` is perhaps just considered a class, whereas `Catalog` is another example of a component.

A component diagram in UML depicts a component and is usually drawn as shown in Fig. 13.10. The name of the component is shown in bold. The little icon in the top-right corner indicates that it is a component as does the keyword «`component`». Either one of these two, the keyword or the icon, may be omitted if desired. In this book though, we will employ both.

For another example of a component, consider the implementation of the drawing program in Chap. 11. In Sect. 11.5, we considered the issue of catering to multiple

Fig. 13.10 Component diagram: Example 1

user interface technologies. The idea is that the interface `UIContext` would capture the variations in the technologies, whereas the hierarchy rooted in `Item` would represent the differences between the items. We could have multiple modules, each implementing the `UIContext` interface. `UIContext` is called a **provided interface** of `SwingUI`. Any implementation of `UIContext` could be used, depending on the technology, making `SwingUI` a component.

Figure 13.11 shows one way of depicting `SwingUI` as a component. The full circle connected to the component represents the provided interface. An alternative representation is given in Fig. 13.12; the diagram shows the class `SwingUI` as a component that implements the interface `UIContext`. The methods implemented by `SwingUI` are documented in this representation.

Consider the Clock class in the implementation of the microwave system of Sect. 10.7. This could be considered a component because it could be used in any application that requires timed signals spaced one second apart; moreover, it is just

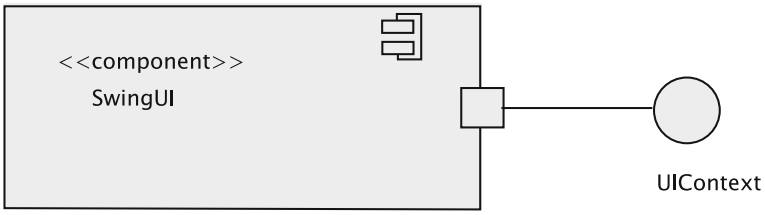

Fig. 13.11 Component diagram: Example 2

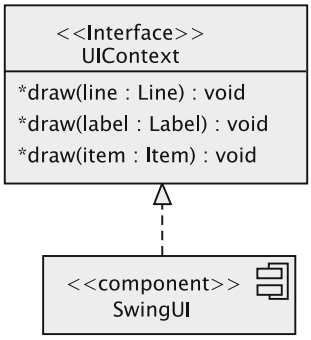

Fig. 13.12 Component diagram: Example 3

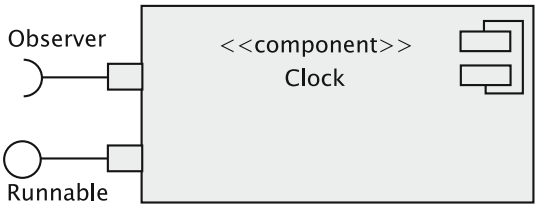

Fig. 13.13 Component diagram: Example 4

one possible implementation of the interface `Runnable`. It sends out a signal every second, but one could easily substitute a different implementation that sends signals at a different frequency. The class has two important properties:

1. It implements the `Runnable` interface, which is a requirement of classes that use the clock. `Runnable` is a provided interface.
2. It requires that any class that uses it implement the `Observer` interface. `Observer` is said to be a **required interface**.

The concepts are depicted in Fig. 13.13. As we discussed earlier, the circle represents the `Runnable` interface implemented by `Clock`. The half-circle with the word `Observer` means that any class that uses `Clock` must implement the `Observer` interface.

For a slightly more involved example, consider the microwave system. We could think of it as comprising three components: `GUIDisplay`, `Clock`, and `Microwave`. `GUIDisplay` implements the functionality of `MicrowaveDisplay`. (In our system, `MicrowaveDisplay` is an abstract class, not an interface, for implementation convenience. Apart from this technical issue, `MicrowaveDisplay` could be considered an interface.) The component `Microwave` could be implemented in a variety of ways as we discussed in Chap. 10. The diagram is given in Fig. 13.14.

Notice that the provided interface of a class becomes the required interface of another. For instance, `MicrowaveDisplay` is a provided interface of `GUIDisplay`, whereas it is a required interface of `Microwave`.

13.5.1 Usage

Component diagrams show certain aspects of relationships that are not apparent in other class diagrams. The relationship between provided and required interfaces helps the reader understand the design better. It is worthwhile noting that although some types may be implemented as abstract classes instead of interfaces, the functionality provided in the abstract class may be so insignificant that such a class could very well be considered as an interface and shown as such in component diagrams. The ultimate purpose is to help the reader understand the design and the spectrum of choices available for design and implementation.

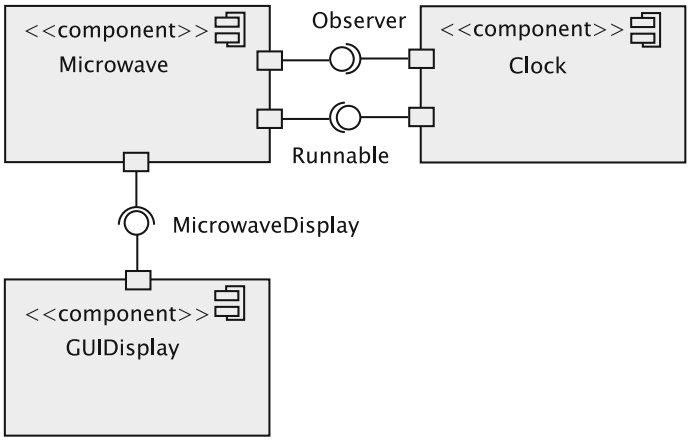

Fig. 13.14 Component diagram: Example 5

13.6 Composite Structure Diagrams

Some classes are complicated enough to warrant a separate form of documentation. For instance, consider a figure such as a rectangle, which is represented by a `Rectangle` object. The object may store four `Line` objects, in addition to other things such as line colour and line type (dashed or continuous, etc.). In such a situation, it is often the case that the `Line` objects are not shared between multiple figures. For example, even if a `Triangle` object exists with one or two lines falling exactly at the same coordinates as the corresponding `Line` objects of a `Rectangle`, there would be separate `Line` objects for the `Triangle` object. In other words, the `Line` objects in the `Rectangle` object are exclusively for the use of the rectangle.

In such a case, we dignify the class by calling it a **composite** and use a composite structure diagram to depict it. While the distinction between a composite and a class may appear fuzzy, the following three rules should help the reader make the determination.

1. The class must have at least one member with a non-primitive type. In this context, it seems appropriate to ignore language-specific definition of primitive types. For example, a language may designate every value as an object; this may include integers, characters, and Booleans. All of these types and strings should be considered primitive types.
2. At least one non-primitive member of the class must be exclusively for the use of the class. When an object is destroyed, either that member should be destroyed as well, or it should be entrusted for the exclusive use of another object.
3. At least one member that satisfies the second criterion should be a "part" of the object. For instance, consider a rectangle; each of the four lines that is an edge of the rectangle is intuitively a "part" of the rectangle object. On the other hand, if we have a player object in a card-playing program store all the cards that have

been played so far, the list itself cannot perhaps be considered a "part" of the player object. The question as to whether a member is a "part" of the object may not be easy to answer, making the criterion somewhat difficult to apply in some situations.

Let us examine a few classes in light of the above guidelines.

```
public class ComplextNumber {
  private double realPart;
  private double imaginaryPart;
  // methods
}
```

The class cannot be considered a composite because the fields are not objects themselves.

Consider the following Java declaration of a `Customer` class.

```
public class Customer {
  private String name;
  private String address;
  private City city;
  private State state;
  private String phone;
  // methods
}
```

The members, `name`, `address`, and `phone` are primitive types, whereas `city` and `state` are not. Therefore, the class meets the first requirement set forth earlier. In general, there will be multiple customers from the same state and even from the same city. Therefore, those two objects are not for the exclusive use of a specific customer. Because of that reason, the class cannot be considered a composite.

Let us look at the `Member` class of the library system.

```
public class Member implements Serializable {
  private static final long serialVersionUID = 1L;
  private String name;
  private String address;
  private String phone;
  private String id;
  private static final String MEMBER_STRING ="M";
  private List booksBorrowed = new LinkedList();
  private List booksOnHold = new LinkedList();
  private List transactions = new LinkedList();
  // methods
}
```

The three list objects are for the exclusive use of the `Member` object. Nonetheless, it is difficult to imagine them being part of a member. Hence, it seems appropriate to reject `Member` as a composite.

Finally, let us consider the following GUI class in the microwave example from Chap. 10.

```
private class SimpleDisplay extends JFrame {
    private JButton doorCloser = new JButton("close door");
    private JButton doorOpener = new JButton("open door");
    private JButton cookButton = new JButton("cook");
    private JLabel doorStatus = new JLabel("Door Closed");
    private JLabel timerValue = new JLabel("          ");
   private JLabel lightStatus = new JLabel("Light Off");
    private JLabel cookingStatus = new JLabel("Not cooking");
    // methods
}
```

The members satisfy all of the three properties. The frame is made up of a number of widgets that live and die with the frame. The members are for the exclusive use of the GUI. It would be appropriate to call this a composite.

A composite may be shown as in Fig. 13.15. The window consists of three buttons and four labels. There are no direct connections between the parts, except that clicking on some buttons may cause certain labels to change their displays and some other buttons to be enabled or disabled.

A more explicit relationship between parts of a composite object is illustrated in Fig. 13.16. We employ a **connector** to indicate a relationship between pairs of `Line` objects that form the `Rectangle` object. Lines `line1` and `line2` share `point1`, a `Point` object. Similar interpretations can be made for the other connectors.

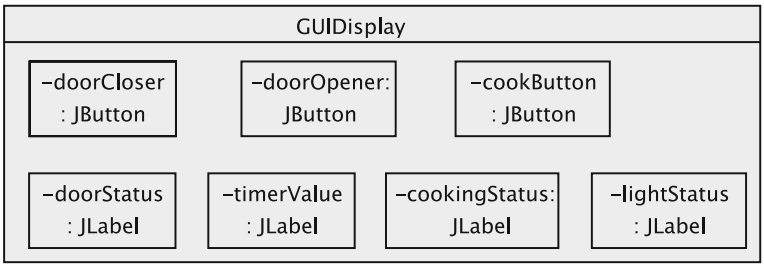

Fig. 13.15 Composite structure diagram: Example 1

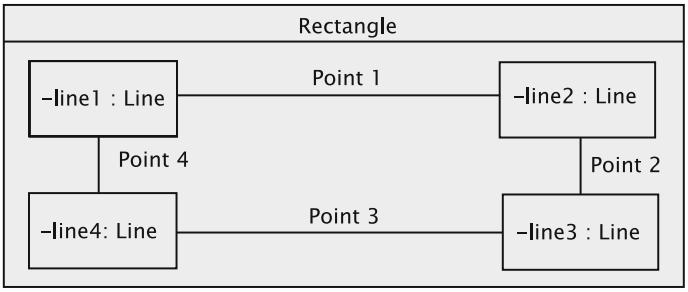

Fig. 13.16 Component structure diagram: Example 2

13.7 Package Diagrams

In many scenarios—not just in computer science—it is necessary to organise large collections. For example, related books are put in the same group in a library or a bookshop. A departmental store has a number of departments each of which hold related items. The utility of such organization is quite apparent.

The situation in computer science, especially in an object-oriented system, is similar. As software becomes more complex, it becomes important to ensure that the modules are properly organised. A **namespace** is an artefact used as a collection of unique names. Entries within two distinct namespaces may have the same name, with the namespace itself used to disambiguate such conflicts.

A **package** is a namespace. It helps developers to develop modules in a non-interfering way. In some situations, it also helps users.

Let us look at a familiar example. Even though the API classes that support the Java language do not form an application system, they are organised into a number of packages. Such an arrangement is beneficial for more than one reason.

1. It avoids conflict in class names. For example, there is a class named `List` associated with GUI construction and another class with the same name as a collection. One way to distinguish them would be to give them different names, which is artificial. Separating them into different packages is more attractive.
2. Classes can be grouped into different categories based on their purpose. This makes it easier for the user to grasp the underlying architecture.
3. Sometimes classes within one group depend in some way on classes in another. This relationship can be more clearly conveyed by separating them into two different name spaces and then describing their relationship. For example, the package `javax.swing` contains components that fire events in the package `javax.swing.event`.

In contrast, when a simple application, say, one with a handful of classes, is being implemented, it is probably not critical to consider a grouping of the classes. For instance, although the library system in Chaps. 6 through 8 was quite illustrative of many design issues, there were only a few classes. All of the functionality was

accessible only to the library staff—there were no conflicting class names, etc. There was no real need to organise the modules in any way.

But consider a slightly more advanced library system. In our new system, users of the library system are divided into three categories:

1. Public (who might become members): They can do the following.
   (a) Apply for membership (via a browser). The library staff will have to prepare a card and probably mail it to the member.
   (b) Check out books.
   (c) Display the catalog in various ways.
   (d) Display information about their account.
   (e) Pay fines using a credit card.
   (f) Update personal information (for example, phone number).
2. Library staff who handle request from a person/member for the following:
   (a) Applying for membership
   (b) Check out books
   (c) Queries on library holdings
   (d) Get information about their account
   (e) Pay fines using a credit card
   (f) Update personal information (for example, phone number)
3. System administrators who deal with the installed software and hardware to:
   (a) add, remove, and update information about library staff, give them access privileges, etc.,
   (b) back up and reload data,
   (c) recover the database in case of a crash.

While we are not going to design the system, it is not hard to see that there will be some modules that are of common utility to one or more of the modules related to the above functionality. We would also need a separate GUI system for each of the above classes of users.

The overall organisation of the system can be described by means of a package diagram. In a typical system, there could be a large package containing more packages nested in it. The nested packages themselves may be large containing even more nested packages. A package is represented by a rectangle with a tab. In a large package, the name of the package is written in the tab. An example is shown in Fig. 13.17. A smaller package (see Fig. 13.18) has its name written inside the larger rectangle rather than in the tab.

For the library system, a *possible* package diagram is shown in Fig. 13.19. (We emphasise *possible* because we have not really designed the system here.) The Library package nests several packages: a pair of GUI and functional packages for the members, staff, and administrators.

The member functionality depends on the staff functionality for completing some of the actions. For instance, if a person fills out and submits an application form

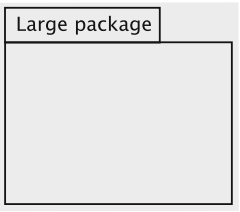

Fig. 13.17 Large package

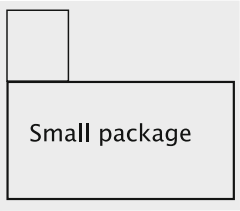

Fig. 13.18 Small package

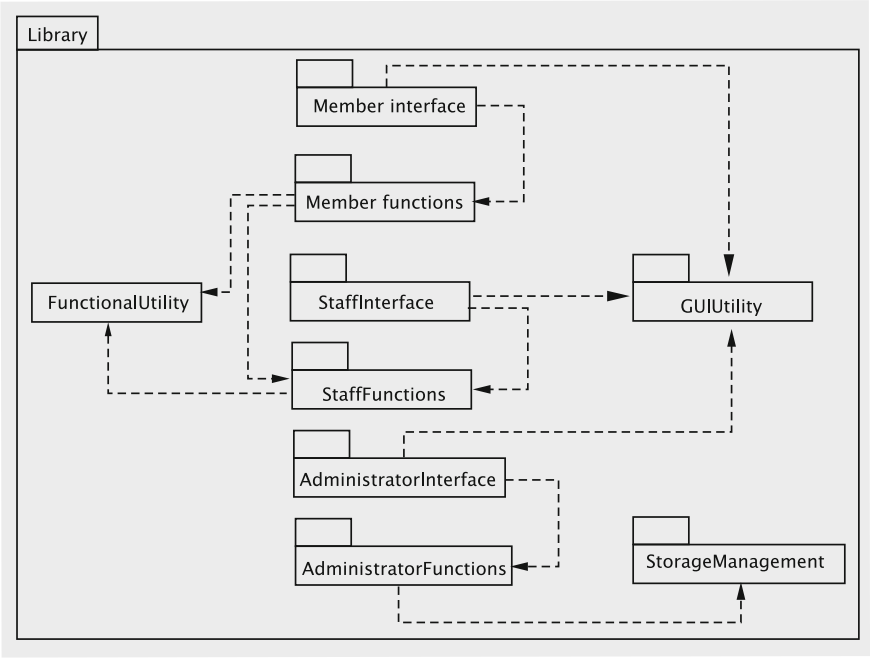

Fig. 13.19 Package diagram

for membership, there must be an invocation of some way for the staff package to generate an identification card; part of the functionality for id card generation may be in the member package.

The relationships between packages are indicated by dashed lines. Since there is some similarity between some of the functionality in the member package and the staff package, it may be appealing to have a common utility package that implements the common needs. For example, the member and staff functionality packages allow credit card payment of fines. The functionality for verifying the card information and actually making the payment would be implemented in the utility package. The class(es) in the member and staff packages that deal with fine payment would make use of this common utility. The utility package could also extract the details of storing and retrieving information from long-term storage. The underlying database may be implemented using the relational model and thus a mapping between the object-oriented model used in the program code and the relational model employed for data storage is inevitable. This could be done by custom software or off-the shelf software, or a combination of the two. The line from the member and staff packages to the utility package represents this dependence of the member and staff packages on the utility package.

The storage package is meant for supporting database backup, reload, and recovery. Much of this may again be a combination of off-the-shelf and custom software. The administrative staff would depend on this software for data maintenance activities.

Note that a dependency line between two packages does not imply a dependency between all pairs of classes in the two packages.

13.8 Object Diagrams

While a class diagram shows the relationships between classes, an object diagram shows a partial or complete view of the objects that exist at a given point in time and the relationships between the objects. An object diagram is more concrete than a class diagram and thus serves as a snapshot of the system, an example that adds to the conceptual understanding of the design. The diagram also serves the designers in identifying the concrete classes.

Obviously, there can be an infinite number of object diagrams, so one must be careful in selecting the proper object diagram to show. The one with a large number of objects would lack clarity and impair understanding; if the number of objects is too small, some of the generalisation that could have been demonstrated may not come through.

A simple example of an object diagram is shown in Fig. 13.20. As in several other examples in this chapter, we draw from the library example. In the system we have two members, whom we refer to as John and Mary. These are two `Member` objects as shown in the picture. The values of the attributes of the two objects are marked.

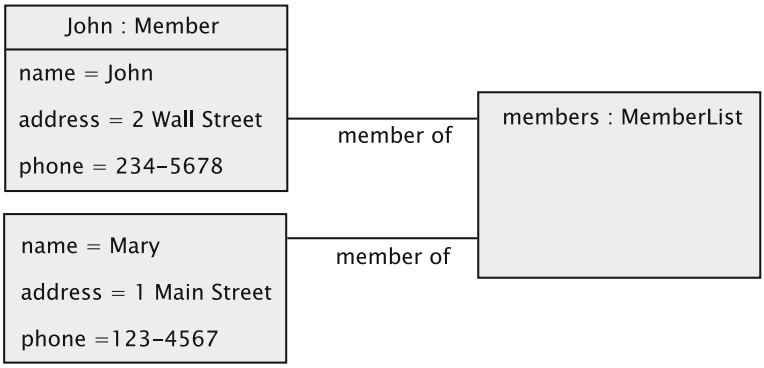

Fig. 13.20 Object diagram: Example 1

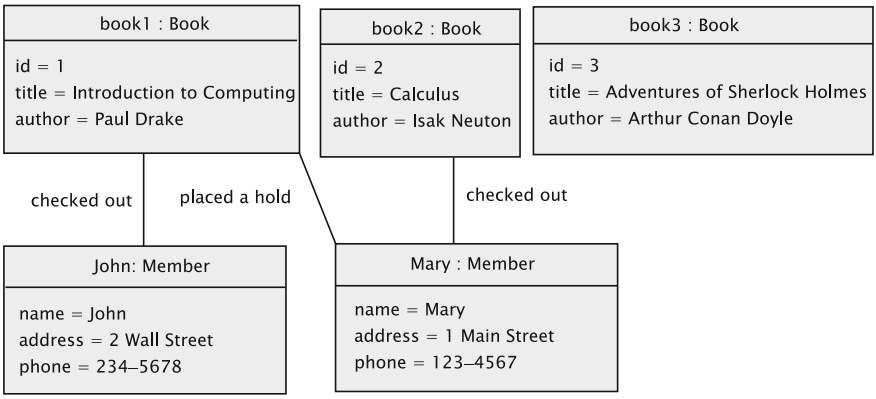

Fig. 13.21 Object diagram: Example 2

The `MemberList` class has two **links** emanating from it to the two objects. The links imply that `MemberList` is a collection, storing references to the two `Member` objects.

A second example is shown in Fig. 13.21. The members are the same as in the previous example. There are three books, two of which are checked out to John and Mary respectively. There are three books, one of which is checked out to John while a second one is borrowed by Mary. Note that we have added labels to the links to explain what the links mean.

13.9 Deployment Diagrams

Although the analysis phase does not specify the hardware or software configuration, it is often the case that the user community will have at least a vague idea of the eventual setup. For example, users may know whether a distributed or centralised

configuration is needed. The configuration will be more clearly defined during the design stage, and it is highly desirable to document the eventual configuration. The Deployment Diagram is used for this purpose.

Unlike most of the other UML diagrams, the deployment diagram depicts the configuration of the finished product. It shows the placement of the hardware and software components, called **artefacts**, in the system. An artefact could be a hardware component such as a network router, software such as operating systems, middleware, applications, database tables, and so on.

Let us consider the library system developed in Chap. 7 being extended to a distributed environment. Assume that the distributed system is implemented using Java Remote Method Invocation (RMI). As discussed in Chap. 12, the system consists of the following components:

1. A server that implements the library functions. This consists of the `Library` and other classes such as `Member, Catalog`, etc. As explained in Chap. 12, `Library` now provides remote methods that can be invoked from other computer systems via remote object invocations.
2. One or more clients that implement the user interface. An example would be the `UserInterface` class developed in Chap. 7, enhanced for RMI.

In this scenario, we have a server machine that hosts the business logic, which includes `Library`, `MemberList`, `MemberIdServer`, and Catalog. In this diagram, we have shown only the singleton instances. Although not depicted, instances of MEMBER, BOOK, and TRANSACTION are also on this machine.

Each clerk station contains an instance of `UserInterface`. There could be many such machines, all of which are connected to the server. The communication is via RMI. The stubs and skeletons associated with RMI are not shown, but are implied by the fact that the interconnection explicitly shows the employment of RMI.

As one can guess, in a deployment diagram we will be showing one box for each artefact (Fig. 13.22). The artefact could be a hardware or software entity. Such a box is called a **node**. Nodes can contain more nodes in them. The node for the clerk

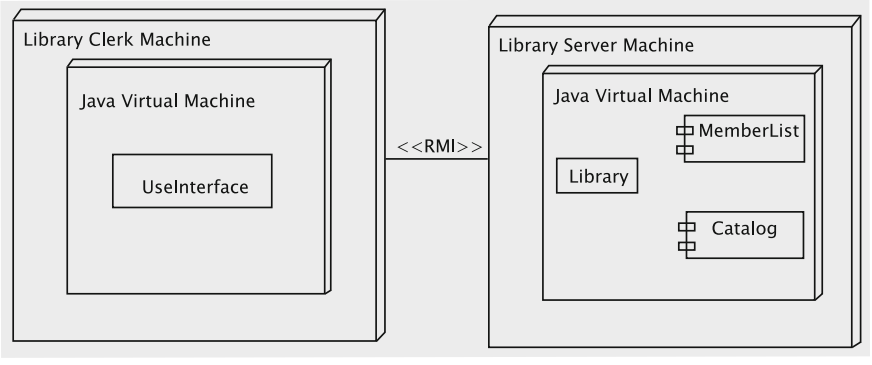

Fig. 13.22 Deployment diagram, Example 1

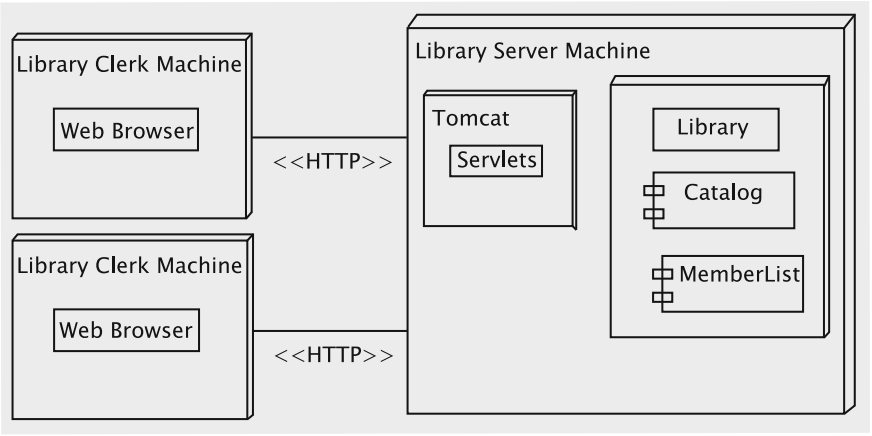

Fig. 13.23 Deployment diagram, Example 2

machine (hardware artefact) contains a node for the Java Virtual Machine (software artefact), which in turn contains the node for `UserInterface`.

A similar interpretation can be made of the server machine as well. Note that `MemberList` and `Catalog` are treated as components and are shown using the corresponding notation.

For a second example, assume that the library is implemented with a web interface. Again, we have done the system development in Chap. 12. The configuration would have two or more machines:

1. Client machines, which could be used by library members or clerks. These machines will run a web browser.
2. A server machine, which consists of the following:

   (a) Tomcat to host the classes for handling requests coming from the user via the browser. These are called servlets.
   (b) Classes associated with the business logic, which includes `Library`, `MemberList`, `MemberIdServer` and `Catalog`.

   These are no different from the classes developed in Chap. 7. The servlets invoke appropriate methods in `Library`.

The deployment diagram for the system is shown in Fig. 13.23.

13.10 Discussion and Further Reading

There is some tendency in some people to overemphasise the importance of UML. UML is a documentation tool, i.e., it serves as a vehicle for conveying our thoughts. UML cannot replace good analysis; however, good analysis and design can be done without using UML.

UML has come in for some criticism, not all of which are undeserved. As the term "unified" suggests, the notation is a compromise on the many prior existing and competing notations for documenting object-oriented systems. People seem to use the notation differently, suggesting confusion, ambiguity, and inconsistency.

The most authentic source on UML is the Object Management Group's web site, http://www.uml.org/. The page http://www.omg.org/gettingstarted/what_is_uml.htm provides an overview. As of the time of writing this document, the latest UML specification was available via http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/. We would like to warn the reader that the document makes for pretty hard reading: until the reader gets some significant experience in using the tool, there is little point in delving into this document in any serious fashion. Instead, we recommend the much friendlier work by Rumbaugh, Jacobson, and Booch [1].

A popular tool for UML modelling is IBM Rational Rose. The OMG web site has references to other tools.

The UML also defines extension mechanisms for extending the UML to meet specialised needs. A UML Extension is formally defined as a set of Stereotypes, TaggedValues, Constraints, and Notation icons that collectively extend and tailor the UML for a specific domain or process. An example of this is the Erickson-Penker Business Process Modelling extension. As the Business Process Model typically has a broader and more inclusive range than just the software system being considered, it also allows the analyst to clearly map what is in the scope of the proposed system and what will be implemented in other ways (e.g. a manual process). In addition to symbols for representing a business process, this custom extension allows the user to represent things like the consumable resources needed by a business process, information needed by the process, etc. Further discussion on this is beyond the scope of this book.

Reference

1. R.W. Sebesta, *Concepts of Programming Languages* (Addison-Wesley, Reading, 2007)