Chapter 5
Design with Patterns

Contents

5.1 Indirect Communication: Publisher-Subscriber
5.1.1 Control Flow
5.1.2 Pub-Sub Pattern Initialization
5.1.3
5.1.4
5.1.5

5.2 More Patterns
5.2.1 Command
5.2.2 Decorator
5.2.3 State
5.2.4 Proxy

5.3 Concurrent Programming
5.3.1 Threads
5.3.2 Exclusive Resource Access—Exclusion Synchronization
5.3.3 Cooperation between Threads—Condition Synchronization
5.3.4
5.2.3

5.4 Broker and Distributed Computing
5.4.1 Broker Pattern
5.4.2 Java Remote Method Invocation (RMI)
5.4.3
5.4.4

5.5 Information Security
5.5.1 Symmetric and Public-Key Cryptosystems
5.5.2 Cryptographic Algorithms
5.5.3 Authentication
5.5.4

5.6 Summary and Bibliographical Notes

Problems

"It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change."
—Charles Darwin

"Man has a limited biological capacity for change. When this capacity is overwhelmed, the capacity is in future shock."
—Alvin Toffler

Design patterns are convenient solutions for software design problems commonly employed by expert developers. The power of design patterns derives from reusing proven solution "recipes" from similar problems. In other words, patterns are *codifying* practice rather than *prescribing* practice, or, they are capturing the existing best practices, rather than inventing untried procedures. Patterns are used primarily to *improve existing designs or code by rearranging it according to a "pattern."* By reusing a pattern, the developer gains *efficiency*, by avoiding a lengthy process of trials and errors in search of a solution, and *predictability* because this solution is known to work for a given problem.

Design patterns can be of different complexities and for different purposes. In terms of complexity, the design pattern may be as simple as a naming convention for object methods in the JavaBeans specification (see Chapter 7) or can be a complex description of interactions between the multiple classes, some of which will be reviewed in this chapter. In terms of the purpose, a pattern may be intended to facilitate component-based development and reusability, such as in the JavaBeans specification, or its purpose may be to prescribe the rules for responsibility assignment to the objects in a system, as with the design principles described in Section 2.5.

As pointed earlier, finding effective representation(s) is a recurring theme of software engineering. By condensing many structural and behavioral aspects of the design into a few simple concepts, patterns make it easier for team members to discuss the design. As with any symbolic language, one of the greatest benefits of patterns is in *chunking* the design knowledge. Once team members are familiar with the pattern terminology, the use of this terminology shifts

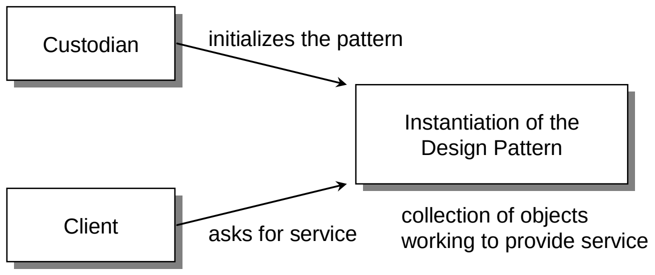

Custodian
initializes the pattern
Instantiation of the Design Pattern
Client
asks for service
collection of objects working to provide service

Figure 5-1: The players in a design pattern usage.

the focus to higher-level design concerns. No time is spent in describing the mechanics of the object collaborations because they are condensed into a single pattern name.

This chapter reviews some of the most popular design patterns that will be particularly useful in the rest of the text. What follows is a somewhat broad and liberal interpretation of design patterns. The focus is rather on the techniques of solving specific problems; nonetheless, the "patterns" described below do fit the definition of patterns as recurring solutions. These patterns are conceptual tools that facilitate the development of flexible and adaptive applications as well as reusable software components.

Two important observations are in order. First, finding a name that in one or few words conveys the meaning of a design pattern is very difficult. A similar difficulty is experienced by user interface designers when trying to find graphical icons that convey the meaning of user interface operations. Hence, the reader may find the same or similar software construct under different names by different authors. For example, The *Publisher-Subscriber* design pattern, described in Section 5.1, is most commonly called *Observer* [Gamma *et al.*, 1995], but [Larman, 2005] calls it *Publish-Subscribe*. I prefer the latter because I believe that it conveys better the meaning of the underlying software construct[1]. Second, there may be slight variations in what different authors label with the same name. The difference may be due to the particular programming language idiosyncrasies or due to evolution of the pattern over time.

Common players in a design pattern usage are shown in Figure 5-1. A Custodian object assembles and sets up a pattern and cleans up after the pattern's operation is completed. A client object (can be the same software object as the custodian) needs and uses the services of the pattern. The design patterns reviewed below generally follow this usage "pattern."

5.1 Indirect Communication: Publisher-Subscriber

[1] The *Publish-Subscribe* moniker has a broader use than presented here and the interested reader should consult [Eugster *et al*. 2003].

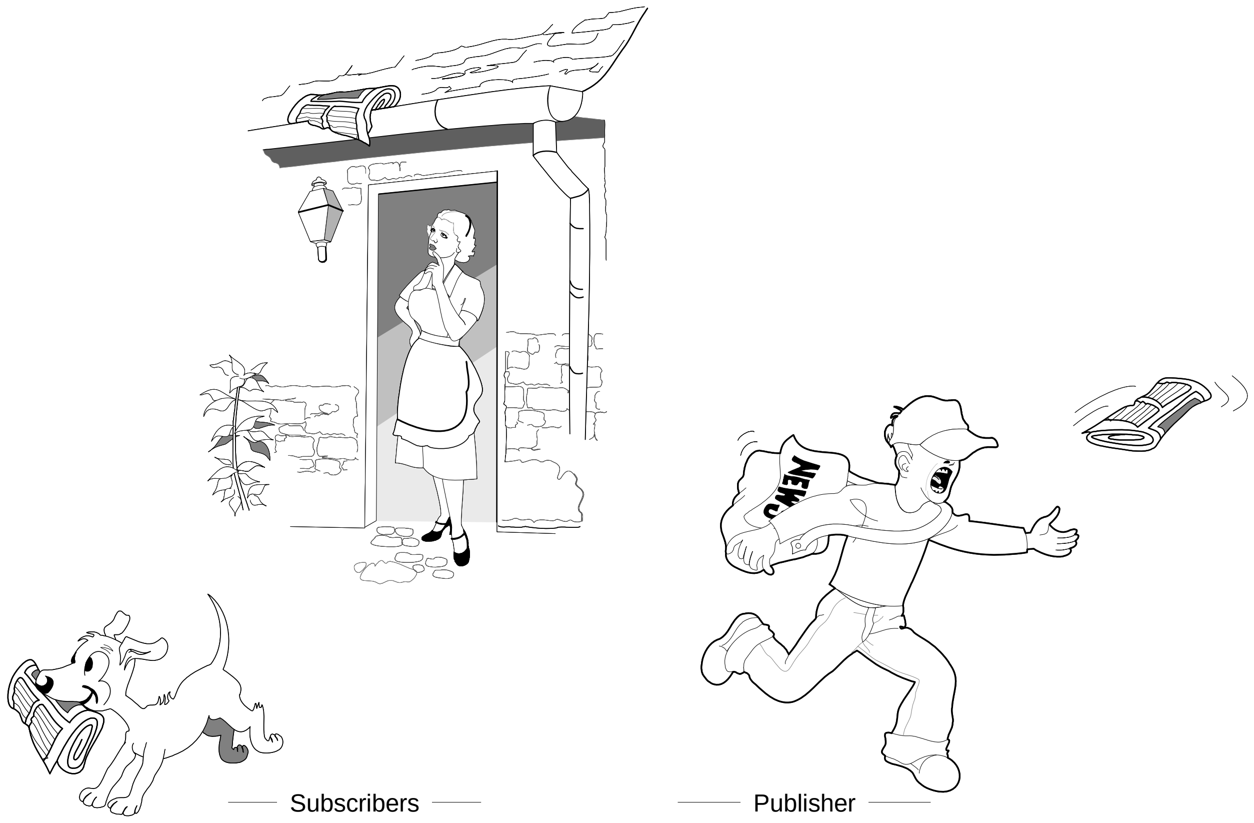


Figure 5-2: The concept of indirect communication in a Publisher/Subscriber system.

“If you find a good solution and become attached to it, the solution may become your next problem.”
—Robert Anthony

“More ideas to choose from mean more complexity … and more opportunities to choose wrongly.”
—Vikram Pandit

Publisher-subscriber design pattern (see Figure 5-2) is used to implement *indirect communication* between software objects. Indirect communication is usually used when an object cannot or does not want to know the identity of the object whose method it calls. Another reason may be that it does not want to know what the effect of the call will be. The most popular use of the pub-sub pattern is in building reusable software components.

1) Enables building reusable components

2) Facilitates separation of the business logic (responsibilities, concerns) of objects

Centralized vs. decentralized execution/program-control method—spreads responsibilities for better balancing. Decentralized control does not necessarily imply concurrent threads of execution.

The problem with building reusable components can be illustrated on our case-study example. Let us assume that we want to reuse the KeyChecker object in an extended version of our case-study application, one that sounds alarm if someone is tampering with the lock. We need to modify the method `unlock()` not only to send message to LockCtrl but also to AlarmCtrl, or to introduce a new method. In either case, we must change the object code, meaning that the object is not reusable as-is.

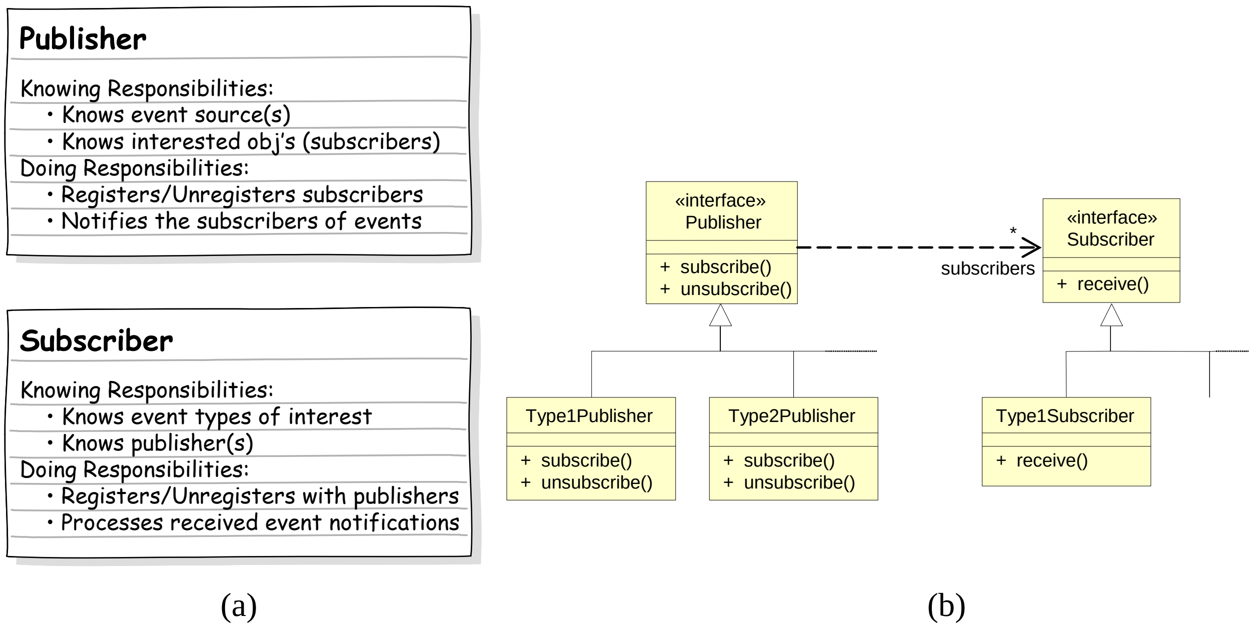
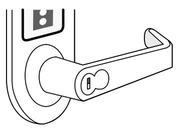


Figure 5-3: Publisher/Subscriber objects employee cards (a), and the class diagram of their collaborations (b).

Information source acquires information in some way and we assume that this information is important for other objects to do the work they are designed for. Once the source acquires information (becomes "information expert"), it is logical to expect it to pass this information to others and initiate their work. However, this tacitly implies that the source object "knows" what the doer object should do next. This knowledge is encoded in the source object as an "IF-THEN-ELSE" rule and must be modified every time the doer code is modified (as seen earlier in Section 2.5).

Request- vs. event-based communication, Figure 5-4: In the former case, an object makes an explicit request, whereas in the latter, the object expresses interest ahead of time and later gets notified by the information source. In a way, the source is making a method request on the object. Notice also that "request-based" is also synchronous type of communication, whereas event based is asynchronous.



Another way to design the KeyChecker object is to make it become a publisher of events as follows. We need to define two class interfaces: Publisher and Subscriber (see Figure 5-3). The first one, Publisher, allows any object to subscribe for information that it is the source of. The second, Subscriber, has a method, here called `receive()`, to let the Publisher publish the data of interest.

Listing 5-1: Publish-Subscribe class interfaces.

```
public interface Subscriber {
    public void receive(Content content);
}
```

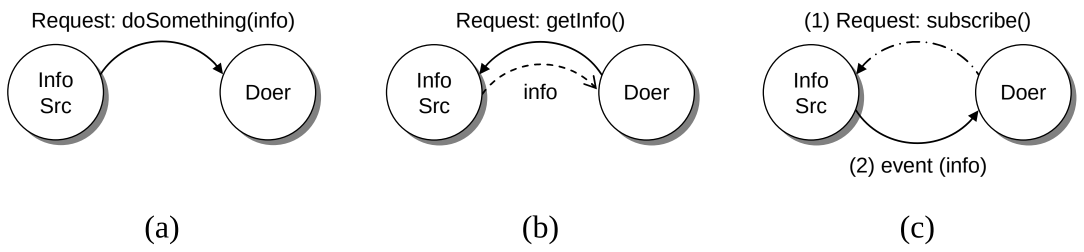


Figure 5-4: Request- vs. event-based communication among objects. (a) Direct request—information source controls the activity of the doer. (b) Direct request—the doer controls its own activity, information source is only for lookup, but doer must know when is the information ready and available. (c) Indirect request—the doer controls its own activity and does not need to worry when the information is ready and available—it gets prompted by the information source.

```
import java.util.ArrayList;
public class Content {
    public Publisher source_;
    public ArrayList data_;

    public Content(Publisher src, ArrayList dat) {
        source_ = src;
        data_ = (ArrayList) dat.clone(); // for write safety...
    }                      // ...avoid aliasing and create a new copy
}
```

```
public interface Publisher {
    public subscribe(Subscriber subscriber);
    public unsubscribe(Subscriber subscriber);
}
```

A `Content` object contains only data, no business logic, and is meant to transfer data from `Publisher` to `Subscriber`. The actual classes then implement those two interfaces. In our example, the key Checker object would then implement the `Publisher`, while DeviceCtrl would implement the `Subscriber`.

Listing 5-2: Refactored the case-study code of using the Publisher-Subscriber design pattern. Here, the class `DeviceCtrl` implements the `Subscriber` interface and the class `Checker` implements the `Publisher` interface.

```
public class DeviceCtrl implements Subscriber {
    protected LightBulb bulb_;
    protected PhotoSObs sensor_;

    public DeviceCtrl(Publisher keyChecker, PhotoSObs sensor, ... ) {
        sensor_ = sensor;
        keyChecker.subscribe(this);
        ...
    }
    public void receive(Content content) {
        if (content.source_ instanceof Checker) {
            if ( ((String)content.data_).equals("valid") ) {
                // check the time of day; if daylight, do nothing
                if (!sensor_.isDaylight()) bulb_.setLit(true);
            }
        } else (check for another source of the event ...) {
            ...
        }
    }
}
```

```
import java.util.ArrayList;
import java.util.Iterator;

public class Checker implements Publisher {
    protected KeyStorage validKeys_;
    protected ArrayList subscribers_ = new ArrayList();

    public Checker( ... ) { }

    public subscribe(Subscriber subscriber) {
        subscribers_.add(subscriber); // could check whether this
    }                                 // subscriber already subscribed

    public unsubscribe(Subscriber subscriber) {
        int idx = subscribers_.indexOf(subscriber);
        if (idx != -1) { subscribers_.remove(idx); }
    }

    public void checkKey(Key user_key) {
        boolean valid = false;
        ...  // verify the user key against the "validKeys_" database

        // notify the subscribers
        Content cnt = new Content(this, new ArrayList());

        if (valid) {  // authorized user
            cnt.data.add("valid");
        } else {      // the lock is being tampered with
            cnt.data.add("invalid");
        }
        cnt.data.add(key);

        for (Iterator e = subscribers_.iterator(); e.hasNext(); ) {
            ((Subscriber) e.next()).receive(cnt);
        }
    }
}
```

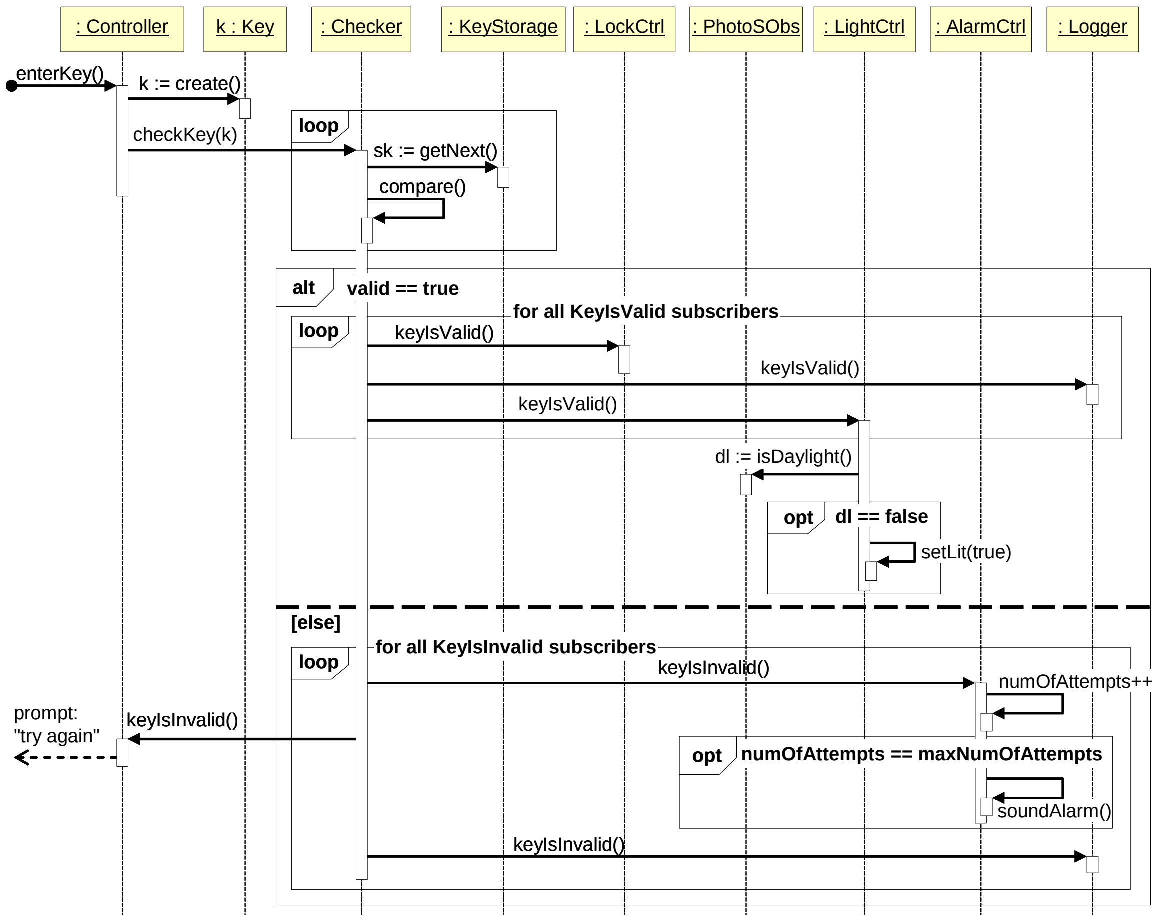


Figure 5-5: Sequence diagram for publish-subscribe version of the use case "Unlock." Compare this with Figure 2-27.

A Subscriber may be subscribed to several sources of data and each source may provide several types of content. Thus, the Subscriber must determine the source and the content type before it takes any action. If a Subscriber gets subscribed to many sources which publish different content, the Subscriber code may become quite complex and difficult to manage. The Subscriber would contain many `if()`or `switch()` statements to account for different options. A more object-oriented solution for this is to use class *polymorphism*—instead of having one Subscriber, we should have several Subscribers, each specialized for a particular source. The Subscribers may also have more than one `receive()` method, each specialized for a particular data type. Here is an example. We could implement a Switch by inheriting from the generic `Subscriber` interface defined above, or we can define new interfaces specialized for our problem domain.

Listing 5-3: Subscriber interfaces for "key-is-valid" and "key-is-invalid" events.

```
public interface KeyIsValidSubscriber {
    public void keyIsValid(LockEvent event); // receive() method
}

public interface KeyIsInvalidSubscriber {
    public void keyIsInvalid(LockEvent event); // receive() method
}
```

The new design for the Unlock use case is shown in Figure 5-5, and the corresponding code might look as shown next. Notice that here the attribute `numOfAttempts` belongs to the `AlarmCtrl`, unlike the first implementation in Listing 2-2 (Section 2.7), where it belonged to the `Controller`. Notice also that the `Controller` is a `KeyIsInvalidSubscriber` so it can prompt the user to enter a new key if the previous attempt was unsuccessful.

Listing 5-4: A variation of the Publisher-Subscriber design from Listing 5-2 using the subscriber interfaces from Listing 5-3.

```
public class Checker implements LockPublisher {
    protected KeyStorage validKeys_;
    protected ArrayList keyValidSubscribers_ = new ArrayList();
    protected ArrayList keyInvalidSubscribers_ = new ArrayList();

    public Checker(KeyStorage ks) { validKeys_ = ks; }

    public void subscribeKeyIsValid(KeyIsValidSubscriber sub) {
        keyValidSubscribers_.add(sub);
    }

    public void subscribeKeyIsInvalid(KeyIsInvalidSubscriber sub) {
        keyInvalidSubscribers_.add(sub);
    }

    public void checkKey(Key user_key) {
        boolean valid = false;
        ...  // verify the key against the database

        // notify the subscribers
        LockEvent evt = new LockEvent(this, new ArrayList());
        evt.data.add(key);

        if (valid) {
            for (Iterator e = keyValidSubscribers_.iterator();
                 e.hasNext(); ) {
                ((KeyIsValidSubscriber) e.next()).keyIsValid(evt);
            }
        } else { // the lock is being tampered with
            for (Iterator e = keyInvalidSubscribers_.iterator();
                e.hasNext(); ) {
                ((KeyIsInvalidSubscriber) e.next()).keyIsInvalid(evt);
            }
        }
    }
}
```

```
public class DeviceCtrl implements KeyIsValidSubscriber {
    protected LightBulb bulb_;
    protected PhotoSObs photoObserver_;

    public DeviceCtrl(LockPublisher keyChecker, PhotoSObs sensor, .. )
    {
        photoObserver_ = sensor;
        keyChecker.subscribeKeyIsValid(this);
        ...
    }

    public void keyIsValid(LockEvent event) {
        if (!photoObserver_.isDaylight())  bulb_.setLit(true);
    }
}
```

```
public class AlarmCtrl implements KeyIsInvalidSubscriber {
    public static final long maxNumOfAttempts_ = 3;
    public static final long interAttemptInterval_ =300000; //millisec
    protected long numOfAttempts_ = 0;
    protected long lastTimeAtempt_ = 0;

    public AlarmCtrl(LockPublisher keyChecker, ...) {
        keyChecker.subscribeKeyIsInvalid(this);
        ...
    }

    public void keyIsInvalid(LockEvent event) {
        long currTime = System.currentTimeMillis();
        if ((currTime – lastTimeAttempt_) < interAttemptInterval_) {
            if (++numOfAttempts_ >= maxNumOfAttempts_) {
                soundAlarm();
                numOfAttempts_ = 0; // reset for the next user
            }
        } else {  // this must be a new user's first mistake ...
            numOfAttempts_ = 1;
        }
        lastTimeAttempt_ = currTime;
    }
}
```

It is of note that what we just did with the original design for the Unlock use case can be considered refactoring. In software engineering, the term *refactoring* is often used to describe modifying the design and/or implementation of a software module without changing its external behavior, and is sometimes informally referred to as "cleaning it up." Refactoring is often practiced as part of the software development cycle: developers alternate between adding new tests and functionality and refactoring the code to improve its internal consistency and clarity. In our case, the design from Figure 2-27 has been transformed to the design in Figure 5-5.

There is a tradeoff between the number of `receive()` methods and the `switch()` statements. On one hand, having a long `switch()` statement complicates the Subscriber's code and makes it difficult to maintain and reuse. On the other hand, having too many `receive()` statements results in a long class interface, difficult to read and represent graphically.

5.1.1 Applications of Publisher-Subscriber

The Publisher-Subscriber design pattern is used in the Java AWT and Swing toolkits for notification of the GUI interface components about user generated events. (This pattern in Java is known as *Source-Listener* or *delegation event model*, see Chapter 7.)

One of the main reasons for software components is easy visualization in integrated development environments (IDEs), so the developer can visually assemble the components. The components are represented as “integrated circuits” in analogy to hardware design, and different `receive()` / `subscribe()` methods represent “pins” on the circuit. If a component has too many pins, it becomes difficult to visualize, and generates too many “wires” in the “blueprint.” The situation is similar to determining the right number of pins on an integrated circuit. (See more about software components in Chapter 7.)

Here I reiterate the key benefits of using the pub-sub design pattern and indirect communication in general:

- The components do not need to know each other’s identity. For example, in the sample code given in Listing 1-1 (Section 1.4.2), LockCtrl maintains a reference to a LightCtrl object.
- The component’s business logic is contained within the component alone. In the same example, LockCtrl explicitly invokes the LightCtrl’s method `setLit()`, meaning that it minds LightCtrl’s business. In the worst case, even the checking of the time-of-day may be delegated to LockCtrl in order to decide when to turn the light on.

Both of the above form the basis for component reusability, because making a component independent of others makes it reusable. The pub-sub pattern is the most basic pattern for reusable software components as will be discussed in Chapter 7.

In the “ideal” case, all objects could be made self-contained and thus reusable by applying the pub-sub design pattern. However, there are penalties to pay. As visible from the examples above, indirect communication requires much more code, which results in increased demand for memory and decreased performance. Thus, if it is not likely that a component will need to be reused or if performance is critical, direct communication should be applied and the pub-sub pattern should be avoided.

When to apply the pub-sub pattern? The answer depends on whether you anticipate that the component is likely to be reused in future projects. If yes, apply pub-sub. You should understand that decoupled objects are independent, therefore reusable and easier to understand, while highly interleaved objects provide fast inter-object communication and compact code. Decoupled objects are better suited for global understanding, whereas interleaved objects are better suited for local understanding. Of course, in a large system, global understanding matters more.

5.1.2 Control Flow

Figure 5-6 highlights the difference in control flow for direct and indirect communication types. In the former case, the control is centralized and all flows emanate from the Controller. In the latter case, the control is decentralized, and it is passed as a token around, cascading from object to object. These diagrams also show the *dynamic* (behavioral) *architecture* of the system.

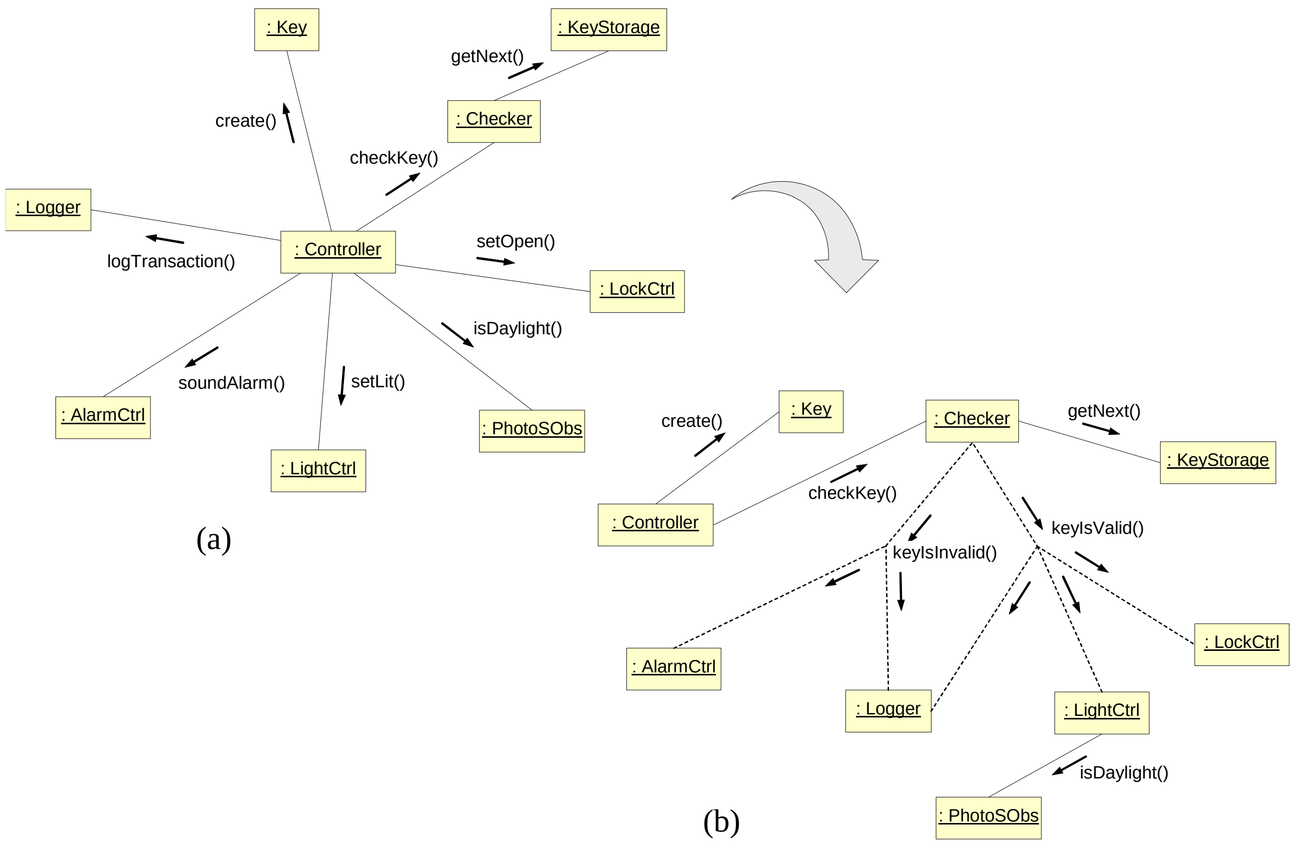


Figure 5-6: Flow control without (a) and with the Pub-Sub pattern (b). Notice that these UML *communication diagrams* are redrawn from Figure 2-27 and Figure 5-5, respectively.

Although in Figure 5-6(b) it appears as if the Checker plays a central role, this is not so because it is not "aware" of being assigned such a role, i.e., unlike the Controller from Figure 5-6(a), this Checker does not encode the requisite knowledge to play such a role. The outgoing method calls are shown in dashed lines to indicate that these are indirect calls, through the Subscriber interface.

Whatever the rules of behavior are stored in one Controller or distributed (cascading) around in many objects, the *output* (seen from outside of the system) is the same. Organization (internal function) matters only if it simplifies the software maintenance and upgrading.

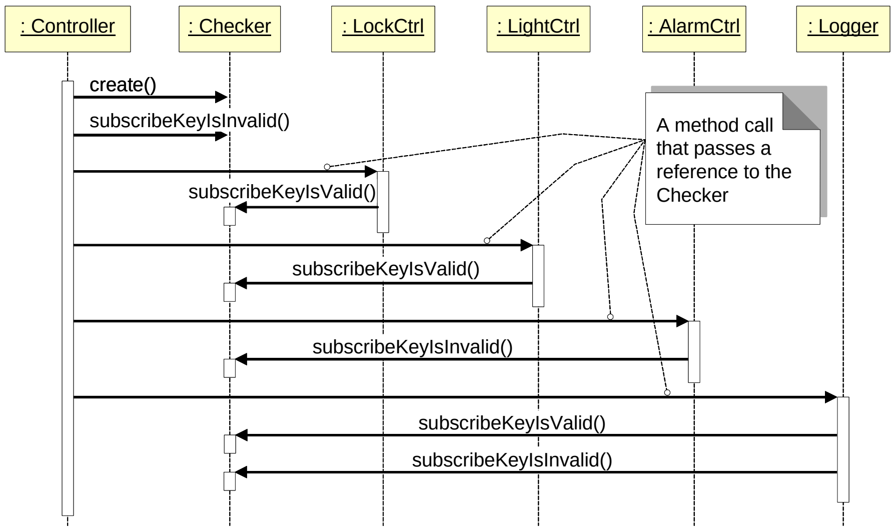


Figure 5-7: Initialization of the pub-sub for the lock control example.

5.1.3 Pub-Sub Pattern Initialization

Note that the "setup" part of the pattern, example shown in Figure 5-7, plays a major, but often ignored, role in the pattern. It essentially represents the master plan of solving the problem using the publish-subscribe pattern and indirect communication.

Most programs are not equipped to split hard problems into parts and then use divide-and-conquer methods. Few programs, too, represent their goals, except perhaps as comments in their source codes. However, a class of programs, called General Problem Solver (GPS), was developed in 1960s by Allen Newel, Herbert Simon, and collaborators, which did have explicit goals and subgoals and solved some significant problems [Newel & Simon, 1962].

I propose that goal representation in object-oriented programs be implemented in the setup part of the program, which then can act at any time during the execution (not only at the initialization) to "rewire" the object relationships.

5.2 More Patterns

Publisher-Subscriber belongs to the category of behavioral design patterns. *Behavioral patterns* separate the interdependent behavior of objects from the objects themselves, or stated differently, they separate functionality from the object to which the functionality applies. This promotes reuse, because different types of functionality can be applied to the same object, as needed. Here I review *Command* as another behavioral pattern.

Another category is structural patterns. An example structural pattern reviewed later is *Proxy*.

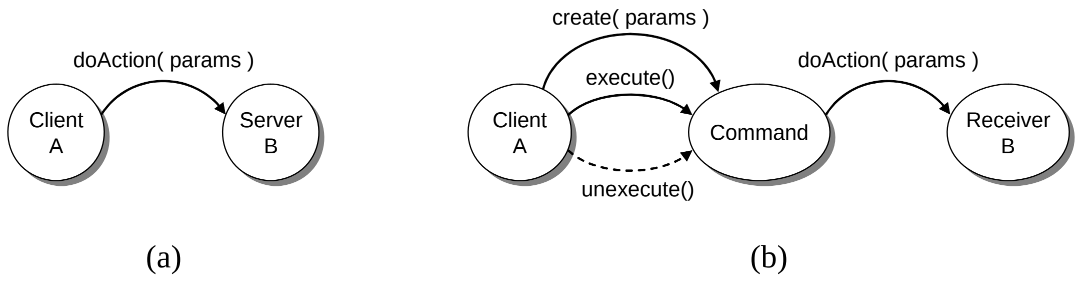


Figure 5-8: Command pattern interposes Command (and other) objects between a client and a server object. Complex actions about rolling back and forward the execution history are delegated to the Command, away from the client object.

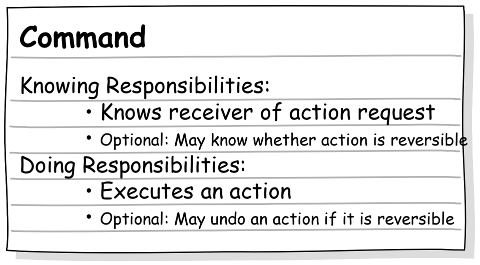
A common drawback of design patterns, particularly behavioral patterns, is that we are replacing what would be a single method call with many method calls. This results in performance penalties, which in certain cases may not be acceptable. However, in most cases the benefits of good design outweigh the performance drawbacks.

5.2.1 Command

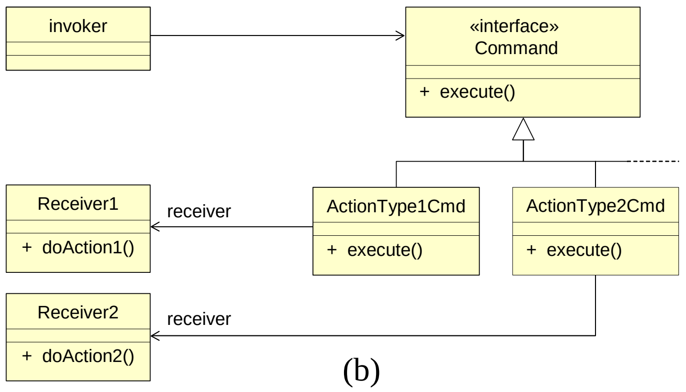
Objects invoke methods on other objects as depicted in Figure 1-22, which is abstracted in Figure 5-8(a). The need for the Command pattern arises if the invoking object (client) needs to reverse the effect of a previous method invocation. Another reason is the ability to trace the course of the system operation. For example, we may need to keep track of financial transactions for legal or auditing reasons. The purpose of the Command patter is to delegate the functionality associated with rolling back the server object's state and logging the history of the system operation away from the client object to the Command object, see Figure 5-8(b).

Instead of directly invoking a method on the Receiver (server object), the client object appoints a Command for this task. The Command pattern (Figure 5-9) encapsulates an action or processing task into an object thus increasing flexibility in calling for a service. Command *represents operations as classes* and is used whenever a method call alone is not sufficient. The Command object is the central player in the Command pattern, but as with most patterns, it needs other objects to assist with accomplishing the task. At runtime, a control is passed to the `execute()` method of a non-abstract-class object derived from Command.

Figure 5-9(c) shows a sequence diagram on how to create and execute a command. In addition to executing requests, we may need to be able to trace the course of the system operation. For example, we may need to keep track of financial transactions for legal or auditing reasons. CommandHistory maintains history log of Commands in linear sequence of their execution.


Command
Knowing Responsibilities:
• Knows receiver of action request
• Optional: May know whether action is reversible
Doing Responsibilities:
• Executes an action
• Optional: May undo an action if it is reversible

(a)



(b)

(c)

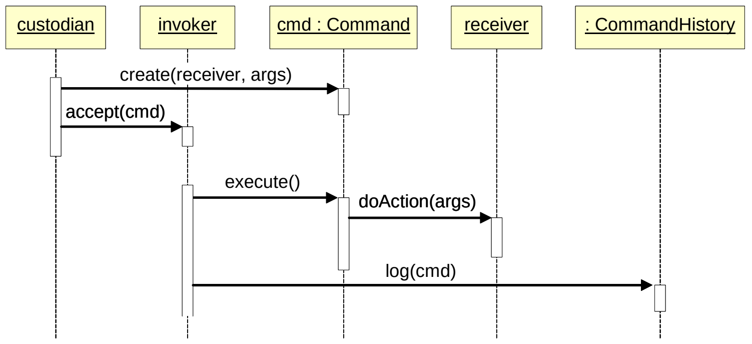
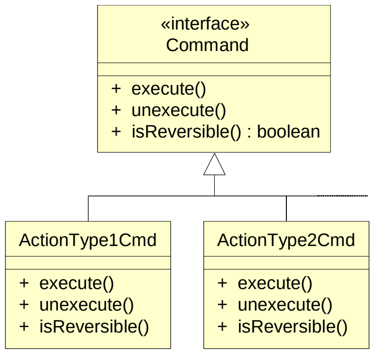


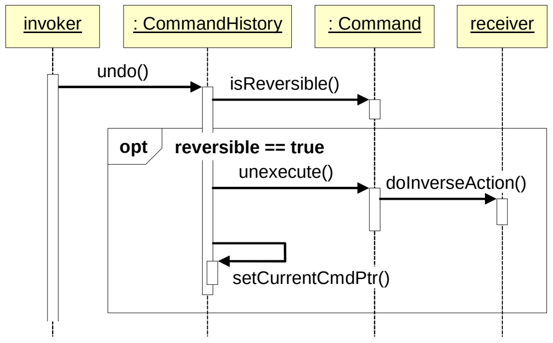
Figure 5-9: (a) Command object employee card. (b) The Command design pattern (class diagram). The base Command class is an interface implemented by concrete commands. (c) Interaction diagram for creating and executing a command.

It is common to use Command pattern in operating across the Internet. For example, suppose that client code needs to make a function call on an object of a class residing on a remote server. It is not possible for the client code to make an ordinary method call on this object because the remote object cannot appear in the usual compile-execute process. It is also difficult to employ remote method invocation (Section 5.4.2) here because we often cannot program the client and server at the same time, or they may be programmed by different parties. Instead, the call is made from the client by pointing the browser to the file containing the servlet (a server-side software component). The servlet then calls its method `service(HttpServletRequest, HttpServletResponse)`. The object `HttpServletRequest` includes all the information that a method invocation requires, such as the argument values, obtained from the "environment" variables at standardized global locations. The object `HttpServletResponse` carries the result of invoking `service()`. This technique embodies the basic idea of the Command design pattern. (See also Listing 5-5.)

Web services allow a similar runtime function discovery and invocation, as will be seen in Chapter 8.



(a)



(b)

Figure 5-10: (a) Class diagram for commands that can be undone. (b) Interaction diagram for undoing a (reversible) command. Compare to Figure 5-9.

Undo/Redo

The Command pattern may optionally be able to support *rollback* of user's actions in an elegant fashion. Anyone who uses computers appreciates the value of being able to undo their recent actions. Of course, this feature assumes that a command's effect can be reversed. In this case, the Command interface would have two more operations (Figure 5-10(a)): `isReversible()` to allow the invoker to find out whether this command can be undone; and `unexecute()` to undo the effects of a previous `execute()` operation.

Figure 5-10(b) shows a sequence diagram on how to undo/redo a command, assuming that it is undoable. Observe also that CommandHistory should decrement its pointer of the current command every time a command is undone and increments it every time a command is redone. An additional requirement on CommandHistory is to manage properly the undo/redo caches. For example, if the user backs up along the undo queue and then executes a new command, the whole redo cache should be flushed. Similarly, upon a context switching, both undo/redo caches should be flushed. Obviously, this does not provide for long-term archiving of the commands; if that is required, the archive should be maintained independently of the undo/redo caches.

In physical world, actions are never reversible (because of the laws of thermodynamics). Even an approximate reversibility may not be realistic to expect. Consider a simple light switch. One might thing that turning the switch off is exactly opposite of turning it on. Therefore, we could implement a request to turn the switch off as an undo operation of the command to turn the switch on. Unfortunately, this may not be true. For example, beyond the inability to recover the energy lost during the period that the switch was on, it may also happen that the light bulb is burnt. Obviously, this cannot be undone (unless the system has a means of automatically replacing a burnt light bulb with a new one ☺).

In digital world, if the previous state is stored or is easy to compute, then the command can be undone. Even here we need to beware of potential error accumulation. If a number is repeatedly divided and then multiplied by another number, rounding errors or limited number of bits for number representation may yield a different number than the one we started with.

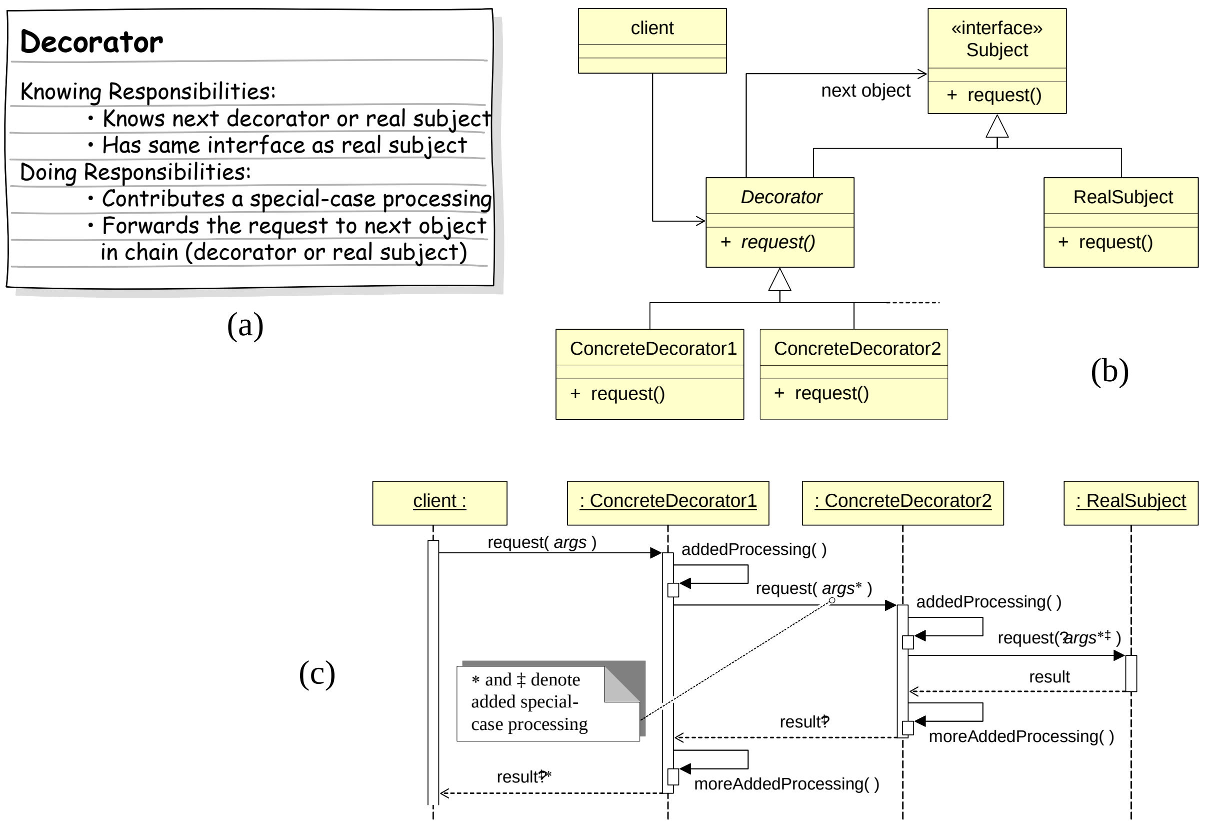


Figure 5-11: (a) Decorator object employee card. (b) The Decorator design pattern (class diagram). (c) Interaction diagram for the Decorator pattern.

5.2.2 Decorator

The Decorator pattern is used to add non-essential behavior to key objects in a software design.

The embellished class (or, decoratee) is wrapped up by an arbitrary number of Decorator classes, which provide special-case behaviors (embellishments).

Figure 5-11

Notice that the Decorator is an abstract class (the class and method names are italicized). The reason for this choice is to collect the common things from all different decorators into a base decorator class. In this case, the Decorator class will contain a reference to the next decorator. The decorators are linked in a chain. The client has a reference to the start of the chain and the chain is terminated by the real subject. Figure 5-11(c) illustrates how a request from the client propagates forward through the chain until it reaches the real subject, and how the result propagates back.

To decide whether you need to introduce Decorator, look for special-case behaviors (embellishment logic) in your design.

Consider the following example, where we wish to implement the code that will allow the user to configure the settings for controlling the household devices when the doors are unlocked or locked. The corresponding user interface is shown in Figure 2-2 (Section 2.2). Figure 5-12 and

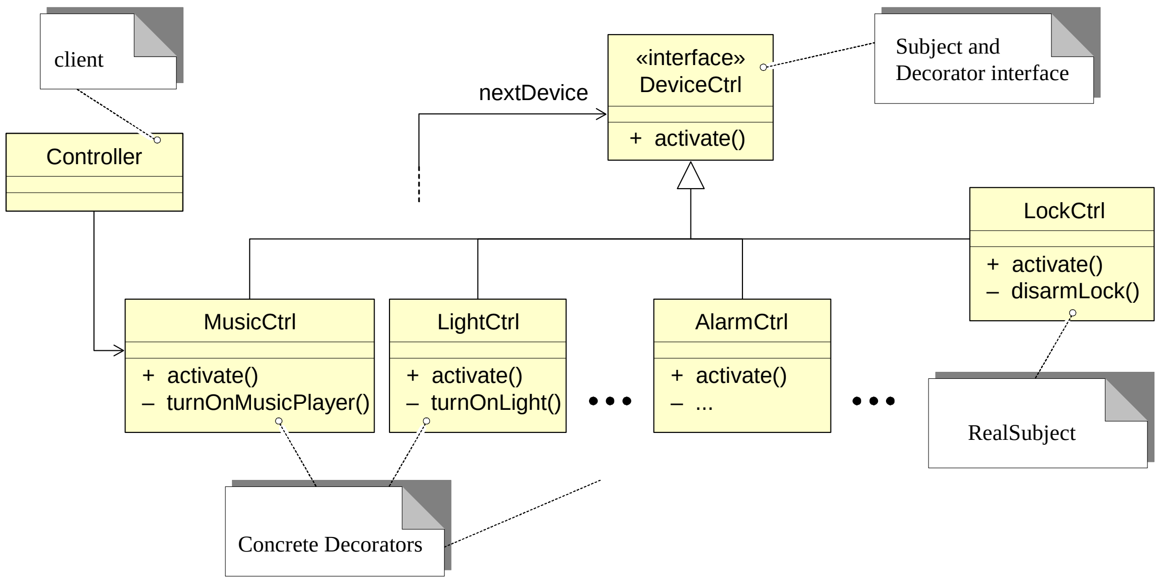


Figure 5-12: Example Decorator class diagram, for implementing the interface in Figure 2-2.

Figure 5-13 show UML diagrams that use the Decorator design pattern in solving this problem. Notice the slight differences in the class diagrams in Figure 5-11(b) and Figure 5-12. As already pointed out, the actual pattern implementation will not always strictly adhere to its generic prototype.

In this example, the decorating functionalities could be added before or after the main function, which is to activate the lock control. For example, in Figure 5-13 the decorating operation `LightCtrl.turnOnLight()` is added before `LockCtrl.activate()`, but `MusicCtrl.turnOnMusicPlayer()` is added after it. In this case all of these operations are commutative and can be executed in any order. This may not always be the case with the decorating functionalities.

5.2.3 State

The State design pattern is usually used when an object's behavior depends on its state in a complex way. In this case, the *state* determines a *mode* of operation. Recall that the *state* of a software object is represented by the current values of its attributes. The State pattern externalizes the relevant attributes into a State object, and this State object has the responsibility of managing the state transitions of the original object. The original object is called "Context" and its attributes are externalized into a State object (Figure 5-14).

A familiar example of object's state determining its mode of operation includes tools in document editors. Desktop computers normally have only keyboard and mouse as interaction devices. To enable different manipulations of document objects, the document needs to be put in a proper state or mode of operation. That is why we select a proper "tool" in a toolbar before performing a manipulation. The selected tool sets the document state. Consider an example of a graphics editor, such as Microsoft PowerPoint. When the user clicks the mouse pointer on a graphical object and drags the mouse, what will happen depends on the currently selected tool. The default

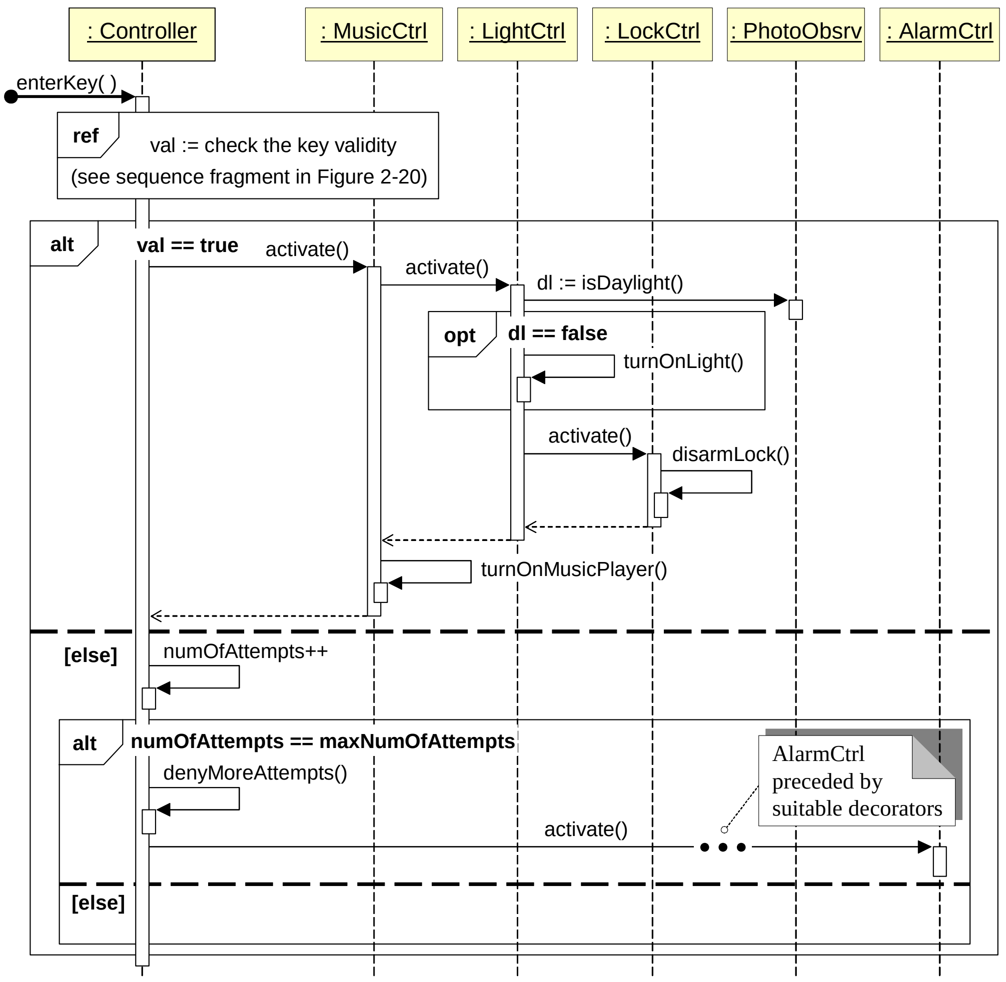


Figure 5-13: Decorator sequence diagram for the class diagram in Figure 5-12.

tool will relocate the object to a new location; the rotation tool will rotate the object for an angle proportional to the distance the mouse is dragged over; etc. Notice that the same action (mouse click and drag) causes different behaviors, depending on the document state (i.e., the currently selected tool).

The State pattern is also useful when an object implements complex conditional logic for changing its state (i.e., the values of this object's attributes). We say that the object is *transitioning from one state* (one set of attribute values) *to another state* (another set of attribute values). To simplify the state transitioning, we define a State interface and different classes that implement this interface correspond to different states of the Context object (Figure 5-14(b)).

Each concrete State class implements the behavior of the Context associated with the state implemented by this State class. The behavior includes calculating the new state of the Context. Because specific attribute values are encapsulated in different concrete states, the current State class just determines the next state and returns it to the Context. Let us assume that the UML state diagram for the Context class is represented by the example in Figure 5-14(c). As shown in Figure 5-14(d), when the Context receives a method call `request()` to handle an event, it calls the method `handle()` on its `currentState`. The current state processes the event and

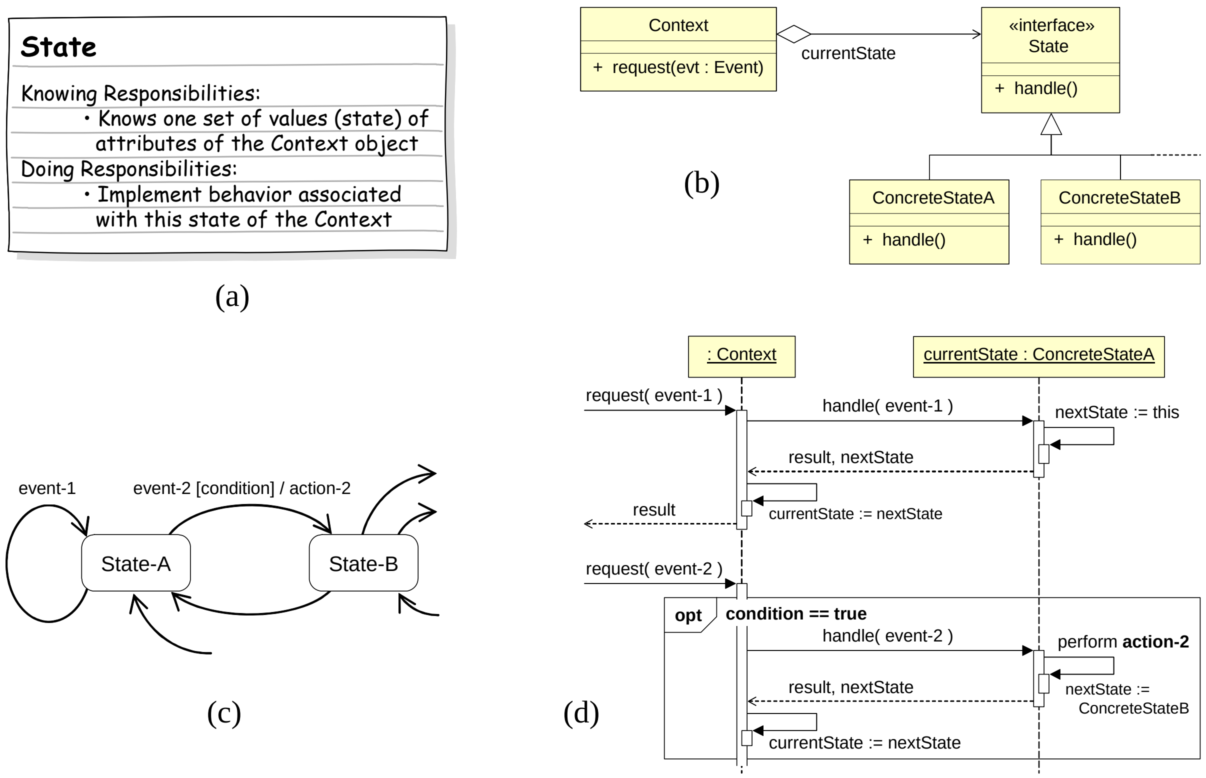
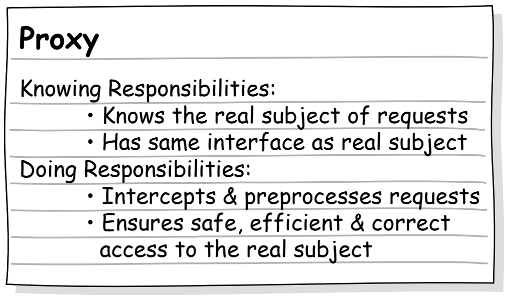


Figure 5-14: (a) State object employee card. (b) The State design pattern (class diagram). (c) Example state diagram for the Context object. (d) Interaction diagram for the state diagram in (c).

performs any action associated with the current state transition. Finally, it returns the next state to the caller Context object. The Context sets this next state as the current state and the next `request` will be `handled` by the new current state.

5.2.4 Proxy

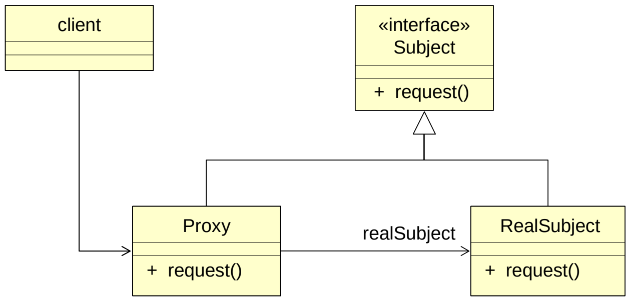
The **Proxy pattern** is used to manage or control access to an object. Proxy is needed when the *logistics* of accessing the subject's services is overly complex and comparable or greater in size than that of client's primary responsibility. In such cases, we introduce a helper object (called "proxy") for management of the subject invocation. A Proxy object is a surrogate that acts as a stand-in for the actual subject, and controls or enhances the access to it (Figure 5-15). The proxy object forwards requests to the subject when appropriate, depending on whether the constraint of the proxy is satisfied.


Proxy

Knowing Responsibilities:
- Knows the real subject of requests
- Has same interface as real subject

Doing Responsibilities:
- Intercepts & preprocesses requests
- Ensures safe, efficient & correct access to the real subject

(a)



(b)

(c)

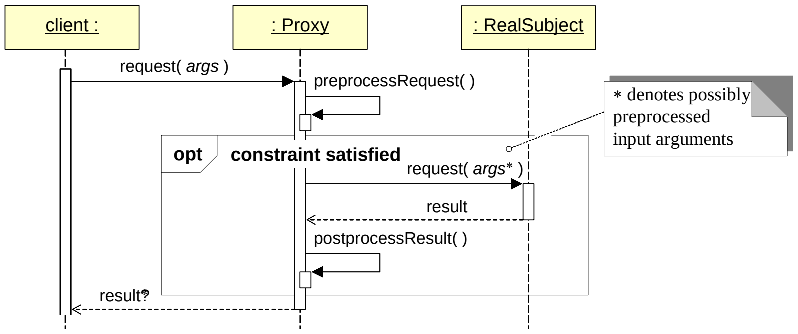


Figure 5-15: (a) Proxy object employee card. (b) The Proxy design pattern (class diagram). (c) Interaction diagram for the Proxy pattern.

The causes of access complexity and the associated constraints include:

- The subject is located in a remote address space, e.g., on a remote host, in which case the invocation (sending messages to it) requires following complex networking protocols. *Solution*: use the **Remote Proxy** pattern for crossing the barrier between different memory spaces
- Different access policies constrain the access to the subject. Security policies require that access is provided only to the authorized clients, filtering out others. Safety policies may impose an upper limit on the number of simultaneous accesses to the subject.
Solution: use the **Protection Proxy** pattern for additional housekeeping
- Deferred instantiation of the subject, to speed up the performance (provided that its full functionality may not be immediately necessary). For example, a graphics editor can be started faster if the graphical elements outside the initial view are not loaded until they are needed; only if and when the user changes the viewpoint, the missing graphics will be loaded. Graphical proxies make this process transparent for the rest of the program.
Solution: use the **Virtual Proxy** pattern for optimization in object creation

In essence we could say that proxy allows client objects to cross a barrier to server objects (or, “subjects”). The barrier may be physical (such as network between the client and server computers) or imposed (such as security policies to prevent unauthorized access). As a result, the client cannot or should not access the server by a simple method call as when the barrier does not exist. The additional functionality needed to cross the barrier is extraneous to the client’s business logic. The proxy object abstracts the details of the logistics of accessing the subject’s services

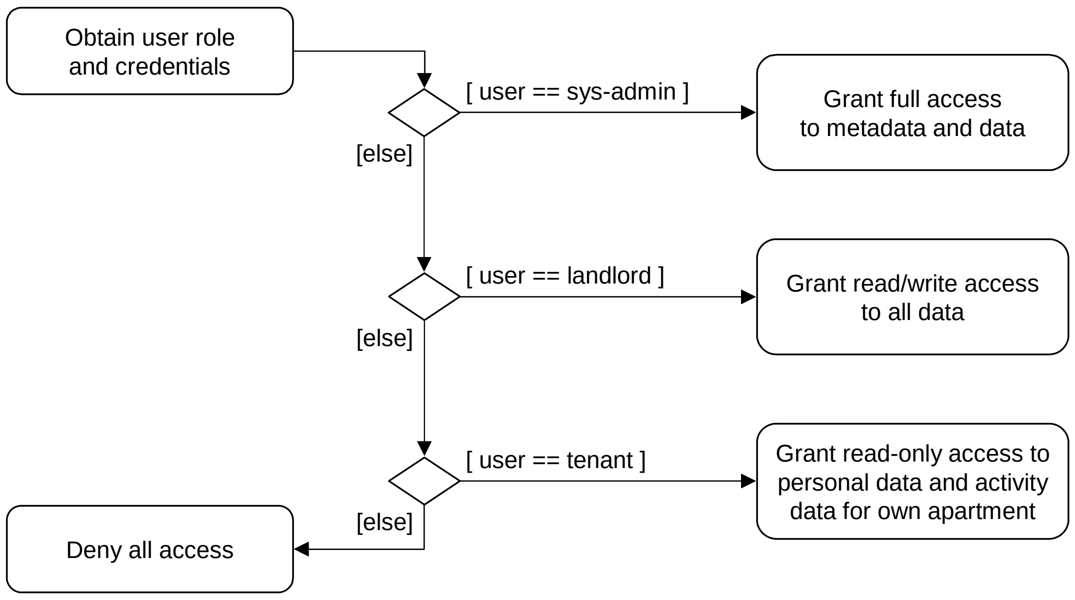

Obtain user role and credentials
[user == sys-admin]
Grant full access to metadata and data
[else]
[user == landlord]
Grant read/write access to all data
[else]
[user == tenant]
Grant read-only access to personal data and activity data for own apartment
[else]
Deny all access

Figure 5-16: Conditional logic for controlling access to the database of the secure home access system.

across different barriers. It does this transparently, so the client has an illusion it is directly communicating with the subject, and does not know that there is a barrier in the middle.

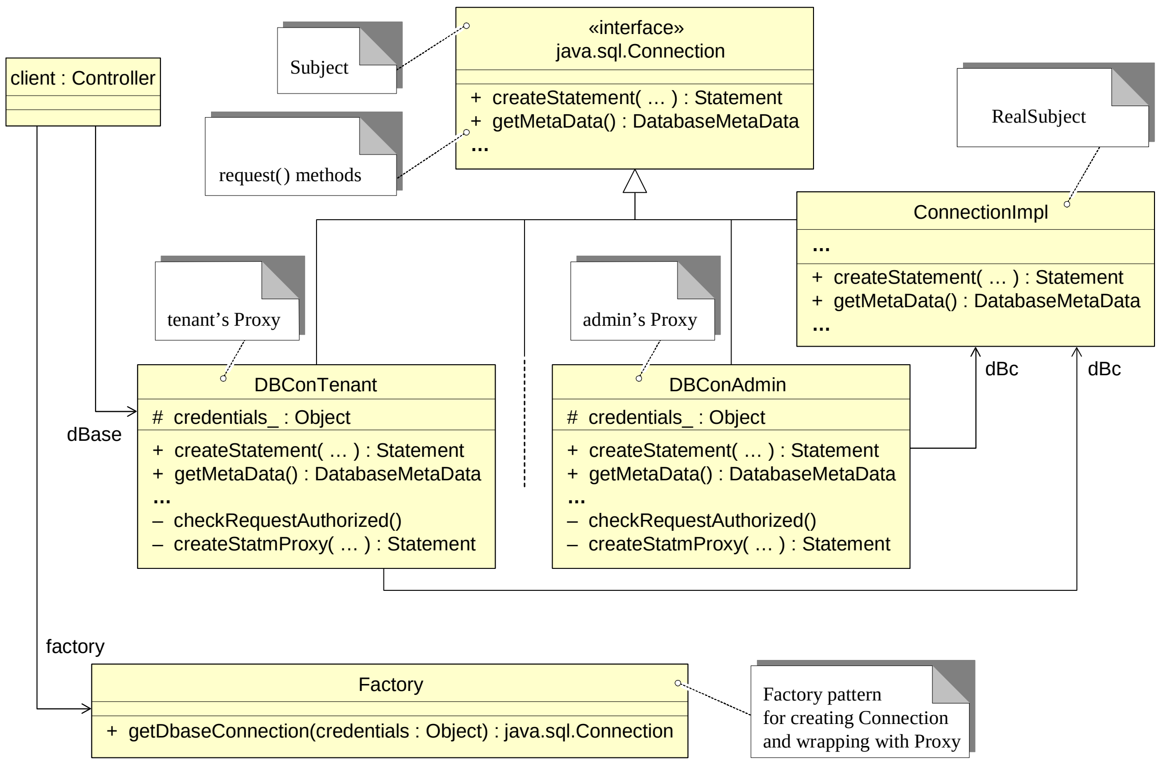
Proxy offers the same interface (set of methods and their signatures) as the real subject and ensures correct access to the real subject. For this reason, the proxy maintains a reference to the real subject (Figure 5-15(b)). Because of the identical interface, the client does not need to change its calling behavior and syntax from that which it would use if there were no barrier involved.

The Remote Proxy pattern will be incorporated into a more complex Broker pattern (Section 5.4). The rest of this section provides more detail on the Protection Proxy.

Protection Proxy

The Protection Proxy pattern can be used to implement different policies to constrain the access to the subject. For example, a security policy may require that a defined service should be seen differently by clients with different privileges. This pattern helps us customize the access, instead of using conditional logic to control the service access. In other words, it is applicable where a subset of capabilities or partial capability should be made available to different actors, based on their roles and privileges.

For example, consider our case study system for secure home access. The sequence diagram for use case UC-5: Inspect Access History is shown in Figure 2-26. Before the Controller calls the method `accessList := retrieve(params : string)` on Database Connection, the system should check that this user is authorized to access the requested data. (This fragment is not shown in Figure 2-26.) Figure 5-16 depicts the Boolean logic for controlling the access to the data in the system database. One way to implement this scheme is to write one large conditional IF-THEN-ELSE statement. This approach would lead to a complex code that is difficult to understand and extend if new policies or roles need to be considered (e.g., the Maintenance



Subject
«interface»
java.sql.Connection
+ createStatement(…) : Statement
+ getMetaData() : DatabaseMetaData
…
client : Controller
request() methods
RealSubject
ConnectionImpl
…
+ createStatement(…) : Statement
+ getMetaData() : DatabaseMetaData
…
tenant's Proxy
admin's Proxy
DBConTenant
# credentials_ : Object
+ createStatement(…) : Statement
+ getMetaData() : DatabaseMetaData
…
– checkRequestAuthorized()
– createStatmProxy(…) : Statement
DBConAdmin
# credentials_ : Object
+ createStatement(…) : Statement
+ getMetaData() : DatabaseMetaData
…
– checkRequestAuthorized()
– createStatmProxy(…) : Statement
dBase
dBc
dBc
factory
Factory
+ getDbaseConnection(credentials : Object) : java.sql.Connection
Factory pattern for creating Connection and wrapping with Proxy

Figure 5-17: Class diagram for the example proxy for enforcing authorized database access. See the interactions in Figure 5-18. (Compare to Figure 5-15(b) for generic Proxy pattern.)

actor). In addition, it serves as a distraction from the main task of the client or server objects. This is where protection proxy enters the picture in and takes on the authorization responsibility.

Figure 5-17 shows how Protection Proxy is implemented in the example of safe database access. Each different proxy type specifies a set of legal messages from client to subject that are appropriate for the current user's access rights. If a message is not legal, the proxy will not forward it to the real subject (the database connection object `ConnectionImpl`); instead, the proxy will send an error message back to the caller (i.e., the client).

In this example, the Factory object acts as a custodian that sets up the Proxy pattern (see Figure 5-17 and Figure 5-18).

It turns out that in this example we need two types of proxies: (a) proxies that implement the database connection interface, such as `java.sql.Connection` if Java is used; and (b) proxies that implement the SQL statement interface, such as `java.sql.Statement` if Java is used. The connection proxy guards access to the database metadata, while the statement proxy guards access to the database data. The partial class diagram in Figure 5-17 shows only the connection-proxy classes, and Figure 5-18 mentions the statement proxy only in the last method call `createStatmProxy()`, by which the database proxy (`DBConTenant`) creates a statement proxy and returns it.

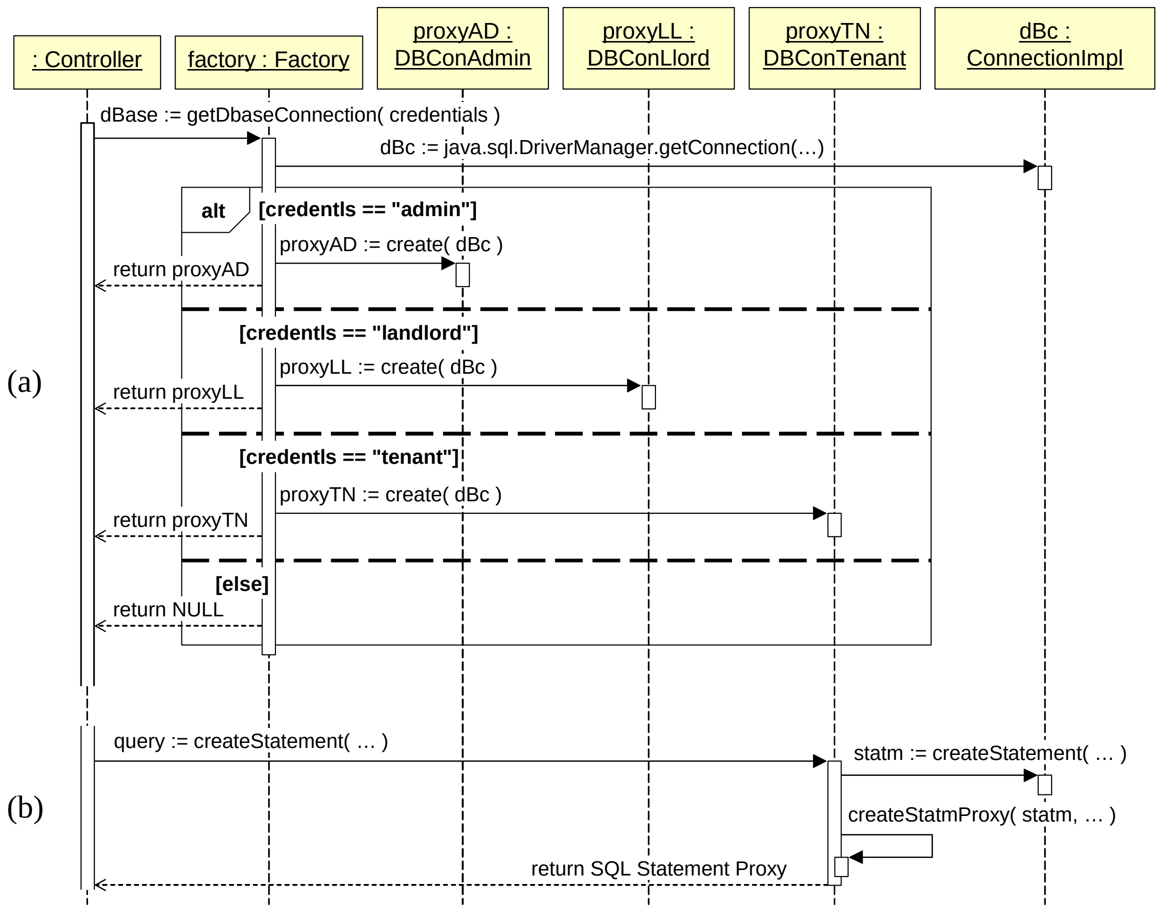


Figure 5-18: Example of Protection Proxy setup (a) and use (b) that solves the access-control problem from Figure 5-16. (See the corresponding class diagram in Figure 5-17.)

Figure 5-18

Listing 5-5: Implementation of the Protection Proxy that provides safe access to the database in the secure home access system.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class WebDataAccessServlet extends HttpServlet {
    private String        // database access parameters
        driverClassName = "com.mysql.jdbc.Driver",
        dbURL = "jdbc:mysql://localhost/homeaccessrecords",
        dbUserID = null,
        dbPassword = null;
    private Connection dBase = null;
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        ...

        dbUserID = config.getInitParameter("userID");
        dbPassword = config.getInitParameter("password");
        Factory factory = new Factory(driverClassName, dbURL);
        dBase = factory.getDbaseConnection(dbUserID, dbPassword);
    }

    public void service(
        HttpServletRequest req, HttpServletResponse resp
    ) throws ServletException, java.io.IOException {
        Statement statm = dBase.createStatement();
        // process the request and prepare ...
        String sql = // ... an SQL statement from the user's request
        boolean ok = statm.execute(sql);

        ResultSet result = statm.getResultSet();
        // print the result into the response (resp argument)
    }
}
```

```
public class Factory {
    protected String dbURL_;
    protected Connection dBc_ = null;

    public Factory(String driverClassName, String dbURL) {
        // load the database driver class   (the Driver class creates
        Class.forName(driverClassName);  //    an instance of itself)
        dbURL_ = dbURL;
    }

    public Connection getDbaseConnection(
        String dbUserID, String dbPassword
    ) {
        dBc_ = DriverManager.getConnection(
            dbURL_, dbUserID, dbPassword
        );

        Connection proxy = null;

        int userType = getUserType(dbUserID, dbPassword);
        switch (userType) {
            case 1:        // dbUserID is a system administrator
                proxy = new DBConAdmin(dBc_, dbUserID, dbPassword);
            case 2:        // dbUserID is a landlord
                proxy = new DBConLlord(dBc_, dbUserID, dbPassword);
            case 3:        // dbUserID is a tenant
                proxy = new DBConTenant(dBc_, dbUserID, dbPassword);
            default:        // dbUserID cannot be identified
                proxy = null;
        }
        return proxy;
    }
}
```

```
// Protection Proxy class for the actual java.sql.Connection
public class DBConTenant implements Connection {
    protected Connection dBc_ = null;
    protected String
        dbUserID = null,
        dbPassword = null;;

    public DBConTenant(
        Connection dBc, String dbUserID, String dbPassword
    ) {
        ...
    }

    public Statement createStatement() {
        statm = dBc_.createStatement();

        return createStatmProxy(statm, credentials_);
    }

    private Statement createStatmProxy(
        Statement statm, credentials_
    ) {
        // create a proxy of java.sql.Statement that is appropriate
        // for a user of the type "tenant"
    }
}
```

One may wonder if we should similarly use Protection Proxy to control the access to the locks in a building, so the landlord has access to all apartments and a tenant only to own apartment. When considering the merits of this approach, the developer first needs to compare it to a straightforward conditional statement and see which approach would create a more complex implementation.

The above example illustrated the use of Proxy to implement security policies for authorized data access. Another example involves safety policies to limit the number of simultaneous accesses. For example, to avoid inconsistent reads/writes, the policy may allow at most one client at a time to access the subject, which in effect serializes the access to the subject. This constraint is implemented by passing a token among clients—only the client in possession of the token can access the subject, by presenting the token when requesting access.

The Protection Proxy pattern is structurally identical to the Decorator pattern (compare Figure 5-11 and Figure 5-15). We can also create a chain of Proxies, same as with the Decorators (Figure 5-11(c)). The key difference is in the intent: Protection Proxy protects an object (e.g., from unauthorized access) while Decorator adds special-case behavior to an object.

SIDEBAR 5.1: Structure and Intention of Design Patterns

♦ The reader may have noticed that many design patterns look similar to one another. For example, Proxy is *structurally* almost identical to Decorator. The difference between them is in their *intention*—what they are used for. The intention of Decorator is to *add* functionality, while the intention of Proxy is to *subtract* functionality, particularly for Protection Proxy.

5.3 Concurrent Programming

"The test of a first-rate intelligence is the ability to hold two opposed ideas in mind at the same time and still retain the ability to function." —F. Scott Fitzgerald

The benefits of concurrent programming include better use of multiple processors and easier programming of reactive (event-driven) applications. In event-driven applications, such as graphical user interfaces, the user expects a quick response from the system. If the (single-processor) system processes all requests sequentially, then it will respond with significant delays and most of the requestors will be unhappy. A common technique is to employ *time-sharing* or time slicing—a *single processor dedicates a small amount of time for each task*, so all of them move forward collectively by taking turns on the processor. Although none of the tasks progresses as fast as it would if it were alone, none of them has to wait as long as it could have if the processing were performed sequentially. The task executions are really sequential but *interleaved* with each other, so they virtually appear as concurrent. In the discussion below I ignore the difference between real concurrency, when the system has multiple processors, and virtual concurrency on a single-processor system. From the user's viewpoint, there is no logical or functional difference between these two options—the user would only see difference in the length of execution time.

Computer *process* is, roughly speaking, a task being executed by a processor. A task is defined by a temporally ordered sequence of instructions (program code) for the processor. In general, a process consists of:

- Memory, which contains executable program code and/or associated data
- Operating system resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows terminology)
- Security attributes, such as the identity of process owner and the process's set of privileges
- Processor state, such as the content of registers, physical memory addresses, etc. The state is stored in the actual registers when the process is executing, and in memory otherwise

Threads are similar to processes, in that both represent a single sequence of instructions executed in parallel with other sequences, either by time slicing on a single processor or multiprocessing. A *process* is an entirely independent program, carries considerable state information, and interacts with other processes only through system-provided inter-process communication mechanisms. Conversely, a thread directly shares the state variables with other threads that are part of the same

5.5 Information Security

"There is nothing special about security; it's just part of getting the job done." —Rob Short

Information security is a nonfunctional property of the system, it is an *emergent* property. Owing to different types of information use, there are two main security disciplines. *Communication security* is concerned with protecting information when it is being transported between different systems. *Computer security* is related to protecting information within a single system, where it can be stored, accessed, and processed. Although both disciplines must work in accord to successfully protect information, information transport faces greater challenges and so communication security has received greater attention. Accordingly, this review is mainly concerned with communication security. Notice that both communication- and computer security must be complemented with physical (building) security as well as personnel security. Security should be thought of as a chain that is as strong as its weakest link.

The main objectives of information security are:

- *Confidentiality*: ensuring that information is not disclosed or revealed to unauthorized persons
- *Integrity*: ensuring consistency of data, in particular, preventing unauthorized creation, modification, or destruction of data
- *Availability*: ensuring that legitimate users are not unduly denied access to resources, including information resources, computing resources, and communication resources
- *Authorized use*: ensuring that resources are not used by unauthorized persons or in unauthorized ways

To achieve these objectives, we institute various *safeguards*, such as concealing (*encryption*) confidential information so that its meaning is hidden from spying eyes; and *key management* which involves secure distribution and handling of the "keys" used for encryption. Usually, the complexity of one is inversely proportional to that of the other—we can afford relatively simple encryption algorithm with a sophisticated key management method.

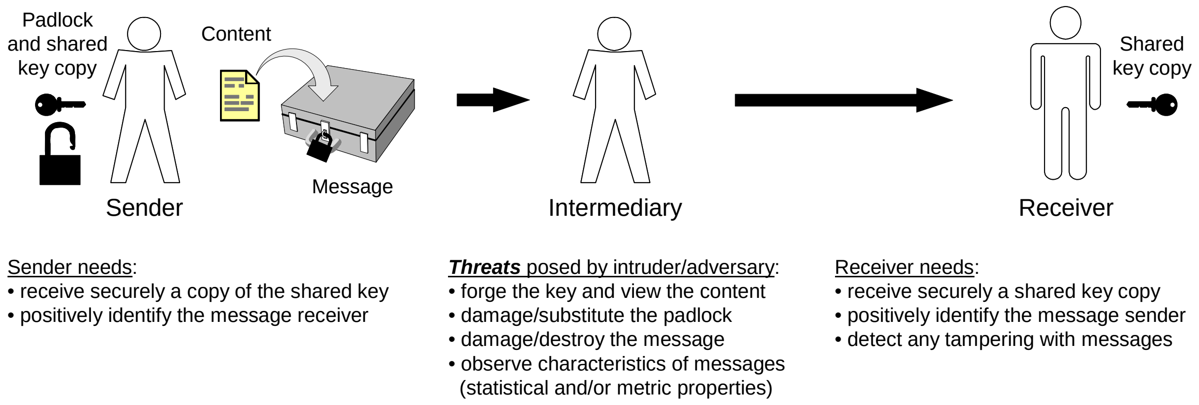

Padlock and shared key copy
Content
Message
Sender
Intermediary
Receiver
Shared key copy
Sender needs:
• receive securely a copy of the shared key
• positively identify the message receiver
Threats posed by intruder/adversary:
• forge the key and view the content
• damage/substitute the padlock
• damage/destroy the message
• observe characteristics of messages (statistical and/or metric properties)
Receiver needs:
• receive securely a shared key copy
• positively identify the message sender
• detect any tampering with messages

Figure 5-31: Communication security problem: Sender needs to transport a confidential document to Receiver over an untrusted intermediary.

Figure 5-31 illustrates the problem of transmitting a confidential message by analogy with transporting a physical document via untrusted carrier. The figure also lists the security needs of the communicating parties and the potential threats posed by intruders. The sender secures the briefcase, and then sends it on. The receiver must use a correct key in order to unlock the briefcase and read the document. Analogously, a sending computer encrypts the original data using an *encryption algorithm* to make it unintelligible to any intruder. The data in the original form is known as ***plaintext*** or ***cleartext***. The encrypted message is known as ***ciphertext***. Without a corresponding "decoder," the transmitted information (ciphertext) would remain scrambled and be unusable. The receiving computer must regenerate the original plaintext from the ciphertext with the correct *decryption algorithm* in order to read it. This pair of data transformations, encryption and decryption, together forms a ***cryptosystem***.

There are two basic types of cryptosystems: (*i*) *symmetric* systems, where both parties use the *same* (secret) key in encryption and decryption transformations; and, (*ii*) *public-key* systems, also known as *asymmetric* systems, where the parties use two related keys, one of which is secret and the other one can be publicly disclosed. I first review the logistics of how the two types of cryptosystems work, while leaving the details of encryption algorithms for the next section.

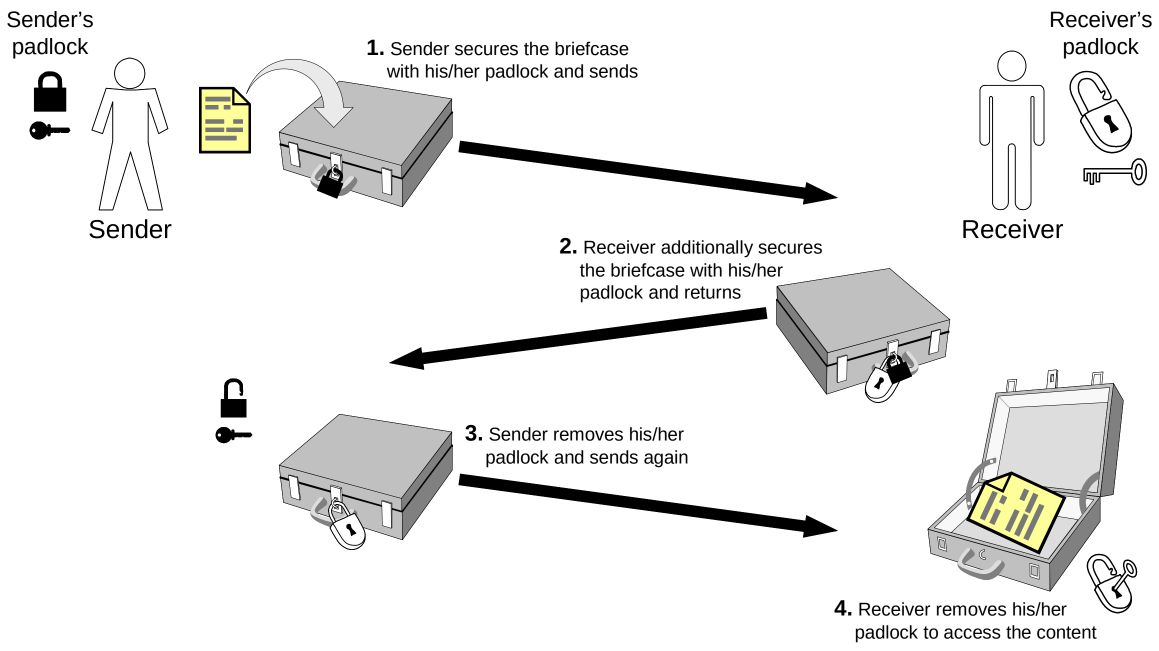

Sender's padlock
Sender
1. Sender secures the briefcase with his/her padlock and sends
Receiver's padlock
Receiver
2. Receiver additionally secures the briefcase with his/her padlock and returns
3. Sender removes his/her padlock and sends again
4. Receiver removes his/her padlock to access the content

Figure 5-32: Secure transmission via untrustworthy carrier. Note that both sender and receiver keep their own keys with them all the time—the keys are never exchanged.

5.5.1 Symmetric and Public-Key Cryptosystems

In symmetric cryptosystems, both parties use the *same* key in encryption and decryption transformations. The key must remain secret and this, of course, implies trust between the two parties. This is how cryptography traditionally works and prior to the late 1970s, these were the only algorithms available.

The system works as illustrated in Figure 5-31. In order to ensure the secrecy of the shared key, the parties need to meet prior to communication. In this case, the fact that only the parties involved know the secret key implicitly *identifies* one to the other.

Using encryption involves two basic steps: *encoding* a message, and *decoding* it again. More formally, a code takes a message M and produces a coded form $f(M)$. Decoding the message requires an inverse function f^{-1}, such that $f^{-1}(f(M)) = M$. For most codes, anyone who knows how to perform the first step also knows how to perform the second, and it would be unthinkable to release to the adversary the method whereby a message can be turned into code. Merely by "undoing" the encoding procedures, the adversary would be able to break all subsequent coded messages.

In the 1970s Ralph Merkle, Whitfield Diffie, and Martin Hellman realized that this need not be so. The weasel word above is "merely." Suppose that the encoding procedure is very hard to undo. Then it does no harm to release its details. This led them to the idea of a *trapdoor function*. We call f a trapdoor function if f is easy to compute, but f^{-1} is very hard, indeed impossible for practical purposes. A trapdoor function in this sense is not a very practical code, because the legitimate user finds it just as hard to decode the message as the adversary does. The final twist is

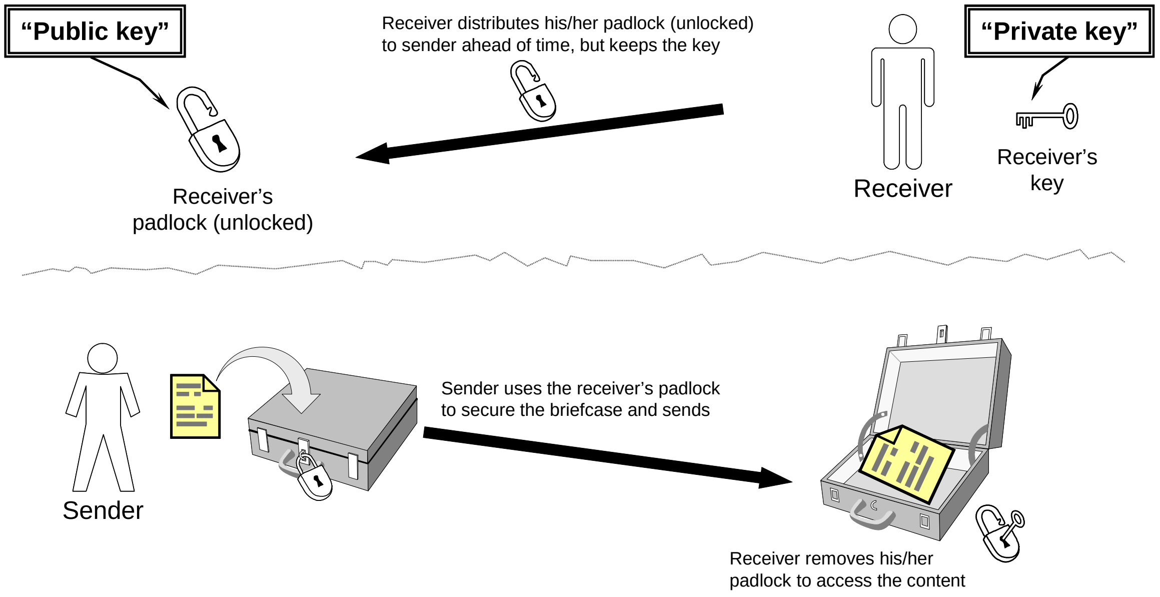

"Public key"
Receiver distributes his/her padlock (unlocked) to sender ahead of time, but keeps the key
"Private key"
Receiver's padlock (unlocked)
Receiver
Receiver's key
Sender
Sender uses the receiver's padlock to secure the briefcase and sends
Receiver removes his/her padlock to access the content

Figure 5-33: Public-key cryptosystem simplifies the procedure from Figure 5-32.

to define f in such a way that a single extra piece of information makes the computation of f^{-1} easy. This is the only bit of information that must be kept secret.

This alternative approach is known as *public-key cryptosystems*. To understand how it works, it is helpful to examine the analogy illustrated in Figure 5-32. The process has three steps. In the first step, the sender secures the briefcase with his or her padlock and sends. Second, upon receiving the briefcase, the receiver secures it additionally with their own padlock and returns. Notice that the briefcase is now doubly secured. Finally, the sender removes his padlock and re-sends. Hence, sender manages to send a confidential document to the receiver without needing the receiver's key or surrendering his or her own key.

There is an inefficiency of sending the briefcase back and forth, which can be avoided as illustrated in Figure 5-33. Here we can skip steps 1 and 2 if the receiver distributed his/her padlock (unlocked, of course!) ahead of time. When the sender needs to send a document, i.e., message, he/she simply uses the receiver's padlock to secure the briefcase and sends. Notice that, once the briefcase is secured, nobody else but receiver can open it, not even the sender. Next I describe how these concepts can be implemented for electronic messages.

5.5.2 Cryptographic Algorithms

Encryption has three aims: keeping data confidential; authenticating who sends data; and, ensuring data has not been tampered with. All cryptographic algorithms involve substituting one thing for another, which means taking a piece of plaintext and then computing and substituting the corresponding ciphertext to create the encrypted message.

Symmetric Cryptography

The Advanced Encryption Standard has a fixed block size of 128 bits and a key size of 128, 192, and 256 bits.

Public-Key Cryptography

As stated above, f is a trapdoor function if f is easy to compute, but f^{-1} is very hard or impossible for practical purposes. An example of such difficulty is factorizing a given number n into prime numbers. An encryption algorithm that depends on this was invented by Ron Rivest, Adi Shamir, and Leonard Adelman (RSA system) in 1978. Another example algorithm, designed by Taher El Gamal in 1984, depends on the difficulty of the discrete logarithm problem.

In the RSA system, the *receiver* does as follows:

1. Randomly select two large prime numbers p and q, which always must be kept secret.
2. Select an integer number E, known as the *public exponent*, such that $(p - 1)$ and E have no common divisors, and $(q - 1)$ and E have no common divisors.
3. Determine the product $n = p \cdot q$, known as *public modulus*.
4. Determine the *private exponent*, D, such that $(E \cdot D - 1)$ is exactly divisible by both $(p - 1)$ and $(q - 1)$. In other words, given E, we choose D such that the integer remainder when $E \cdot D$ is divided by $(p - 1)\cdot(q - 1)$ is 1.
5. Release publicly the **public key**, which is the pair of numbers n and E, $K^{+} = (n, E)$. Keep secret the **private key**, $K^{-} = (n, D)$.

Because a digital message is a sequence of digits, break it into blocks which, when considered as numbers, are each less than n. Then it is enough to encode block by block.

Encryption works so that the sender substitutes each plaintext block B by the ciphertext $C = B^{E} \% n$, where % symbolizes the modulus operation. (The *modulus* of two integer numbers x and y, denoted as $x \% y$, is the integer remainder when x is divided by y.)

Then the encrypted message C (ciphertext) is transmitted. To decode, the receiver uses the *decoding key D*, to compute $B = C^{D} \% n$, that is, to obtain the plaintext B from the ciphertext C.

Example 5.2 RSA cryptographic system

As a simple example of RSA, suppose the receiver chooses $p = 5$ and $q = 7$. Obviously, these are too small numbers to provide any security, but they make the presentation manageable. Next, the receiver chooses $E = 5$, because 5 and $(5 - 1)\cdot(7 - 1)$ have no common factors. Also, $n = p \cdot q = 35$. Finally, the receiver chooses $D = 29$, because $\frac{E \cdot D - 1}{(p-1)\cdot(q-1)} = \frac{5 \cdot 29 - 1}{4 \cdot 6} = \frac{144}{24} = 6$, i.e., they are exactly divisible. The receiver's public key is $K^{+} = (n, E) = (35, 5)$, which is made public. The private key $K^{-} = (n, D) = (35, 29)$ is kept secret.

Now, suppose that the sender wants to send the plaintext "hello world." The following table shows the encoding of "hello." I assign to letters a numeric representation, so that a=1, b=2, …, y=25, and z=26, and I assume that block B is one letter long. In an actual implementation, letters are represented as binary numbers, and the blocks B are not necessarily aligned with letters, so that plaintext "l" will not always be represented as ciphertext "12."

Plaintext letter	Plaintext numeric representation	B^E	Ciphertext B^E % n
h	8	$8^5 = 32768$	8^5 % 35 = 8
e	5	$5^5 = 3125$	5^5 % 35 = 10
l	12	$12^5 = 248832$	12^5 % 35 = 17
l	12	248832	17
o	15	$15^5 = 759375$	15^5 % 35 = 15

The sender transmits this ciphertext to the receiver: 8, 10, 17, 17, 15. Upon the receipt, the receiver decrypts the message as shown in the following table.

Ciphertext	C^D	$B = C^D$ % n	Plaintext letter
8	$8^{29} = 154742504910672534362390528$	8^{29} % 35 = 8	h
10	100000000000000000000000000000	5	e
17	481968572106750915091411825223071697	12	l
17	481968572106750915091411825223071697	12	l
15	12783403948858939111232757568359375	15	o

We can see that even this toy example produces some extremely large numbers.

The point is that while the adversary knows *n* and *E*, he or she does not know *p* and *q*, so they cannot work out $(p - 1)\cdot(q - 1)$ and thereby find *D*. The designer of the code, on the other hand, knows *p* and *q* because those are what he started from. So does any legitimate receiver: the designer will have told them. The security of this system depends on exactly one thing: the notoriously difficulty of factorizing a given number *n* into primes. For example, given $n = 2^{67} - 1$ it took three years working on Sundays for F. N. Cole to find by hand in 1903 *p* and *q* for $n = p\cdot q$ = 193707721 × 761838257287. It would be waste of time (and often combinatorially self-defeating) for the program to grind through all possible options.

Encryption strength is measured by the length of its "key," which is expressed in bits. The larger the key, the greater the strength of the encryption. Using 112 computers, a graduate student decrypted one 40-bit encrypted message in a little over 7 days. In contrast, data encrypted with a 128-bit key is 309,485,009,821,345,068,724,781,056 times stronger than data encrypted with a 40-bit key. RSA Laboratories recommends that the product of *p* and *q* be on the order of 1024 bits long for corporate use and 768 bits for use with less valuable information.

5.5.3 Authentication

5.5.4 Program Security

A virus is malicious code carried from one computer to another by some medium—often an "infected" file. Any operating system that allows third-party programs to run can theoretically run viruses. Some operating systems are more secure than others; earlier versions of Microsoft

Windows did not even provide something as simple as maintain memory space separation. Once on a computer, a virus is executed when its carrier file is "opened" in some meaningful way by software on that system. When the virus executes, it does something unwanted, such as causing software on the host system to send more copies of infected files to other computers over the network, infecting more files, and so on. In other words, a virus typically maximizes its likelihood of being passed on, making itself contagious.

Viral behavior relies on security vulnerabilities that exist in software running on the host system. For example, in the past, viruses have often exploited security vulnerabilities in Microsoft Office macro scripting capabilities. Macro viruses are no longer among the most common virus types. Many viruses take advantage of Trident, the rendering engine behind Internet Explorer and Windows Explorer that is also used by almost every piece of Microsoft software. Windows viruses often take advantage of image-rendering libraries, SQL Server's underlying database engine, and other components of a complete Windows operating system environment as well.

Viruses are typically addressed by antivirus software vendors. These vendors produce virus definitions used by their antivirus software to recognize viruses on the system. Once a specific virus is detected, the software attempts to quarantine or remove the virus—or at least inform the user of the infection so that some action may be taken to protect the system from the virus.

This method of protection relies on knowledge of the existence of a virus, however, which means that most of the time a virus against which you are protected has, by definition, already infected someone else's computer and done its damage. The question you should be asking yourself at this point is how long it will be until you are the lucky soul who gets to be the discoverer of a new virus by way of getting infected by it.

It's worse than that, though. Each virus exploits a vulnerability — but they don't all have to exploit different vulnerabilities. In fact, it's common for hundreds or even thousands of viruses to be circulating "in the wild" that, between them, only exploit a handful of vulnerabilities. This is because the vulnerabilities exist in the software and are not addressed by virus definitions produced by antivirus software vendors.

These antivirus software vendors' definitions match the signature of a given virus — and if they're really well-designed might even match similar, but slightly altered, variations on the virus design. Sufficiently modified viruses that exploit the same vulnerability are safe from recognition through the use of virus definitions, however. You can have a photo of a known bank robber on the cork bulletin board at the bank so your tellers will be able to recognize him if he comes in — but that won't change the fact that if his modus operandi is effective, others can use the same tactics to steal a lot of money.

By the same principle, another virus can exploit the same vulnerability without being recognized by a virus definition, as long as the vulnerability itself isn't addressed by the vendor of the vulnerable software. This is a key difference between open source operating system projects and Microsoft Windows: Microsoft leaves dealing with viruses to the antivirus software vendors, but open source operating system projects generally fix such vulnerabilities immediately when they're discovered.

Thus, the main reason you don't tend to need antivirus software on an open source system, unless running a mail server or other software that relays potentially virus-laden files between other systems, isn't that nobody's targeting your open source OS; it's that any time someone targets it, chances are good that the vulnerability the virus attempts to exploit has been closed up — even if it's a brand-new virus that nobody has ever seen before. Any half-baked script-kiddie has the potential to produce a new virus that will slip past antivirus software vendor virus definitions, but in the open source software world one tends to need to discover a whole new vulnerability to exploit before the "good guys" discover and patch it.

Viruses need not simply be a "fact of life" for anyone using a computer. Antivirus software is basically just a dirty hack used to fill a gap in your system's defenses left by the negligence of software vendors who are unwilling to invest the resources to correct certain classes of security vulnerabilities.

The truth about viruses is simple, but it's not pleasant. The truth is that you're being taken to the cleaners — and until enough software users realize this, and do something about it, the software vendors will continue to leave you in this vulnerable state where additional money must be paid regularly to achieve what protection you can get from a dirty hack that simply isn't as effective as solving the problem at the source would be.

However, we should not forget that security comes at a cost.

In theory, application programs are supposed to access hardware of the computer only through the interfaces provided by the operating system. But many application programmers who dealt with small computer operating systems of the 1970s and early 1980s often bypassed the OS, particularly in dealing with the video display. Programs that directly wrote bytes into video display memory run faster than programs that didn't. Indeed, for some applications—such as those that needed to display graphics on the video display—the operating system was totally inadequate.

What many programmers liked most about MS-DOS was that it "stayed out of the way" and let programmers write programs as fast as the hardware allowed. For this reason, popular software that ran on the IBM PC often relied upon idiosyncrasies of the IBM PC hardware.

5.6 Summary and Bibliographical Notes

Design patterns are heuristics for structuring the software modules and their interactions that are proven in practice. They yield in design for change, so the change of the computing environment has as minimal and as local effect on the code as possible.

Key Points:

- Pattern use must be need-driven: use a pattern only when you need it to improve your software design, not because it can be used, or you simply like hitting nails with your new hammer.

- Using the Broker pattern, a client object invokes methods of a remote server object, passing arguments and receiving a return value with each call, using syntax similar to local method calls. Each side requires a proxy that interacts with the system's runtime.

There are many known design patterns and I have reviewed above only few of the major ones. The text that most contributed to the popularity of patterns is [Gamma *et al*., 1995]. Many books are available, perhaps the best known are [Gamma *et al*., 1995] and [Buschmann *et al*., 1996]. The reader can also find a great amount of useful information on the web. In particular, a great deal of information is available in Hillside.net's Patterns Library: http://hillside.net/patterns/ .

R. J. Wirfs-Brock, "Refreshing patterns," *IEEE Software*, vol. 23, no. 3, pp. 45-47, May/June 2006.

Section 5.1: Indirect Communication: Publisher-Subscriber

Section 5.2: More Patterns

Section 5.3: Concurrent Programming

Concurrent systems are a large research and practice filed and here I provide only the introductory basics. Concurrency methods are usually not covered under design patterns and it is only for the convenience sake that here they appear in the section on software design patterns. I avoided delving into the intricacies of Java threads—by no means is this a reference manual for Java threads. Concurrent programming in Java is extensively covered in [Lea, 2000] and a short review is available in [Sandén, 2004].

[Whiddett, 1987]

Pthreads tutorial: http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pthreads.html

Pthreads tutorial from CS6210 (by Phillip Hutto):
http://www.cc.gatech.edu/classes/AY2000/cs6210_spring/pthreads_tutorial.htm

Section 5.4: Broker and Distributed Computing

The *Broker* design pattern is described in [Buschmann *et al*., 1996; Völter *et al*., 2005].

Java RMI:

Sun Developer Network (SDN) jGuru: "Remote Method Invocation (RMI)," Sun Microsystems, Inc., Online at: http://java.sun.com/developer/onlineTraining/rmi/RMI.html

http://www.javaworld.com/javaworld/jw-04-2005/jw-0404-rmi.html

http://www.developer.com/java/ent/article.php/10933_3455311_1

Although Java RMI works only if both client and server processes are coded in the Java programming language, there are other systems, such as CORBA (Common Object Request Broker Architecture), which work with arbitrary programming languages, including Java. A readable appraisal of the state of affairs with CORBA is available in [Henning, 2006].

Section 5.5: Information Security

In an increasingly networked world, all computer users are at risk of having their personally identifying information and private access data intercepted. Even if information is not stolen, computing resources may be misused for criminal activities facilitated by unauthorized access to others' computer systems.

Kerckhoffs' Principle states that a cryptosystem should remain secure even if everything about it other than the key is public knowledge. The security of a system's design is in no way dependent upon the secrecy of the design, in and of itself. Because system designs can be intercepted, stolen, sold, independently derived, reverse engineered by observations of the system's behavior, or just leaked by incompetent custodians, the secrecy of its design can never really be assumed to be secure itself. Hence, the "security through obscurity" security model by attempting to keep system design secret Open source movement even advocates widespread access to the design of a system because more people can review the system's design and detect potential problems. Transparency ensures that the security problems tend to arise more quickly, and to be addressed more quickly. Although an increased likelihood of security provides no guarantees of success, it is beneficial nonetheless.

There is an entire class of software, known as "fuzzers," that is used to quickly detect potential security weaknesses by feeding abusive input at a target application and observing its behavior under that stress. These are the tools that malicious security crackers use all the time to find ways to exploit software systems. Therefore, it is not necessary to have access to software design (or its source code) to be able to detect its security vulnerabilities. This should not be surprising, given that software defects are rarely found by looking at source code. (Recall the software testing techniques from Section 2.7.) Where access to source code becomes much more important is when trying to determine *why* a particular weakness exists, and how to remove it. One might conclude, then, that the open source transparency does not contribute as much to detecting security problems as it does to fixing them.

Cryptography [Menezes *et al.*, 1997], which is available for download, entirely, online at http://www.cacr.math.uwaterloo.ca/hac/.

ICS 54: History of Public-Key Cryptography:
http://www.ics.uci.edu/~ics54/doc/security/pkhistory.html

http://www.netip.com/articles/keith/diffie-helman.htm

http://www.rsasecurity.com/rsalabs/node.asp?id=2248

http://www.scramdisk.clara.net/pgpfaq.html

http://postdiluvian.org/~seven/diffie.html

http://www.sans.org/rr/whitepapers/vpns/751.php

http://www.fors.com/eoug97/papers/0356.htm

Class iaik.security.dh.DHKeyAgreement

http://www.cs.utexas.edu/users/chris/cs378/f98/resources/iaikdocs/iaik.security.dh.DHKeyAgreement.html

Bill Steele, "'Fabric' would tighten the weave of online security," *Cornell Chronicle* (09/30/10): Fabric's programming language, which is based on Java, builds in security as the program is written. Myers says most of what Fabric does is transparent to the programmer.
http://www.news.cornell.edu/stories/Sept10/Fabric.html

P. Dourish, R. E. Grinter, J. Delgado de la Flor, and M. Joseph, "Security in the wild: user strategies for managing security as an everyday, practical problem," *Personal and Ubiquitous Computing (ACM/Springer)*, vol. 8, no. 6, pp. 391-401, November 2004.

M. J. Ranum, "Security: The root of the problem," *ACM Queue (Special Issue: Surviving Network Attacks)*, vol. 2, no. 4, pp. 44-49, June 2004.

H. H. Thompson and R. Ford, "Perfect storm: The insider, naivety, and hostility," *ACM Queue (Special Issue: Surviving Network Attacks)*, vol. 2, no. 4, pp. 58-65, June 2004.
introducing trust and its pervasiveness in information technology

Microsoft offers integrated hardware-level security such as data execution prevention, kernel patch protection and its free Security Essentials software:
http://www.microsoft.com/security_essentials/

Microsoft's 'PassPort' Out, Federation Services In
In 2004 Microsoft issued any official pronouncements on "TrustBridge," its collection of federated identity-management technologies slated to go head-to-head with competing technologies backed by the Liberty Alliance.
http://www.eweek.com/c/a/Windows/Microsofts-Passport-Out-Federated-Services-In/

Problems

Problem 5.1

Problem 5.2

Consider the online auction site described in Problem 2.31 (Chapter 2). Suppose you want to employ the Publish-Subscribe (also known as Observer) design pattern in your design solution for Problem 2.31. Which classes should implement the Publisher interface? Which classes should implement the Subscriber interface? Explain your answer. (Note: You can introduce new classes or additional methods on the existing classes if you feel it necessary for solution.)

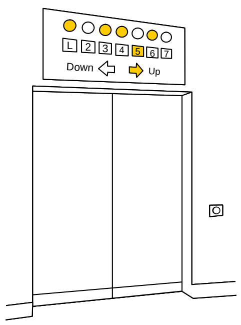
Problem 5.3

In the patient-monitoring scenario of Problem 2.35 (Chapter 2), assume that multiple recipients must be notified about the patient condition. Suppose that your software is to use the Publish-Subscribe design pattern. Identify the key software objects and draw a UML interaction diagram to represent how software objects in the system could accomplish the notification problem.

Problem 5.4

Problem 5.5

Problem 5.6: Elevator Control

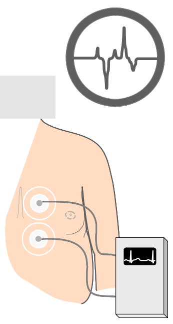


Consider the elevator control problem defined in Problem 3.7 (Chapter 3). Your task is to determine whether the Publisher-Subscriber design pattern can be applied in this design. Explain clearly your answer. If the answer is yes, identify which classes are suitable for the publisher role and which ones are suitable for the subscriber role. Explain your choices, list the events generated by the Publishers, and state explicitly for each Subscriber to which events it is subscribed to.

Problem 5.7

Problem 5.8

Problem 5.9



Consider the automatic patient monitoring system described in Problem 2.35. Carefully examine the draft UML sequence diagram in Figure 2-45. Check if the given design already uses some patterns and explain your claim. Identify as many opportunities as you can to improve the design by applying design patterns. Consider how an unnecessary application of some design patterns would make this design worse. Draw UML sequence diagrams or write pseudo-code to describe the proposed design. Always describe your motivation for adopting or rejecting design modifications.

Problem 5.10

Consider the system for inventory management grocery supermarket from Problem 2.15. Suppose you are provided with an initial software design as follows. This design is based on a basic version of the inventory system, but the reader should be aware of extensions that are discussed in the solution of Problem 2.15(c) and Problem 2.16. The software consists of the following classes:

ReaderIface:

> This class receives messages from RFID readers that specific tags moved in or out of coverage.

DBaseConn:

> This class provides a connection to a relational database that contains data about shelf stock and inventory tasks. The database contains several tables, including `ProductsInfo[key = tagID]`, `PendingTasks[key = userID]`, `CompletedTasks`, and `Statistics[key = infoType]` for various information types, such as the count of erroneous messages from RFID readers and the count of reminders sent for individual pending tasks.

Dispatcher:

> This class manages inventory tasks by opening new tasks when needed and generates notifications to the concerned store employees.

Monitor:

> This class periodically keeps track of potentially overdue tasks. It retrieves the list of pending tasks from the database and generates reminders to the concerned store employees.

Messenger:

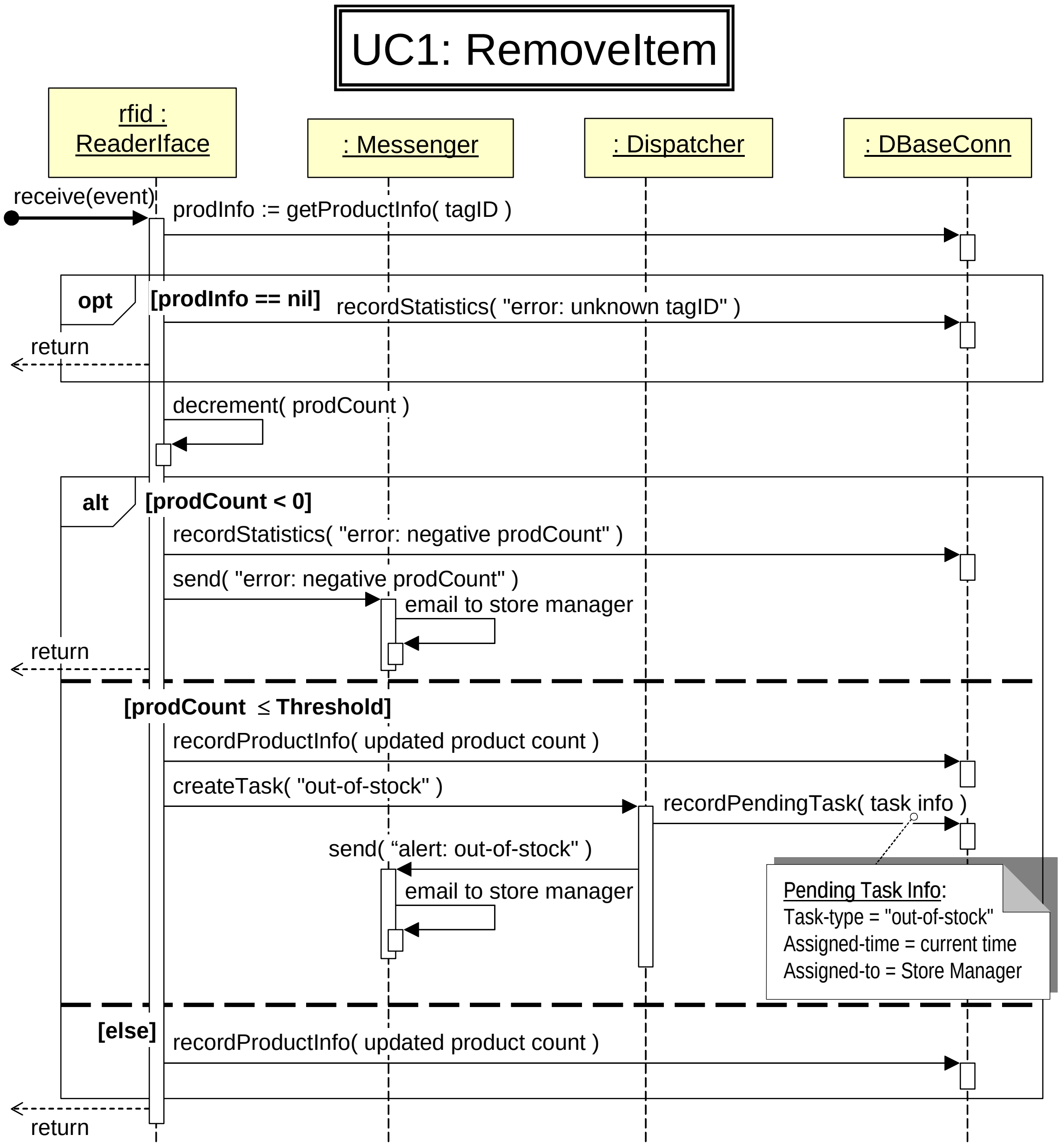
> This class sends email notifications to the concerned store employees. (The notifications are generated by other classes.)

Assume that email notifications are used as a supplementary tool, but the system must keep an internal record of sent notifications and pending tasks, so it can take appropriate actions.

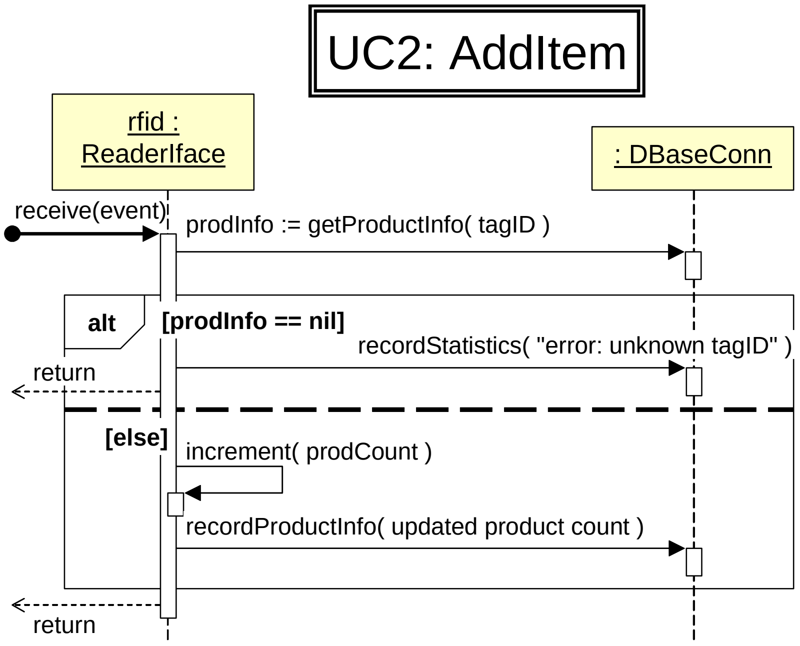
Notice that the current design has a single timer for the whole system. The software designer noticed that sending notifications for overdue tasks does not need to be exactly timed in this system. Delays up to a certain period (e.g., hour or even day) are tolerable. Maintaining many timers would be overkill and would significantly slow down the system. It would not be able to do important activities, such as processing RFID events in a timely fashion. Therefore, the software is designed so that, when a new pending task is created, there is no explicit activation of an associated timer. Instead, the task is simply added to the list of pending tasks. The Monitor object periodically retrieves this list and checks for overdue tasks, as seen below in the design for the use case UC-5 SendReminder.

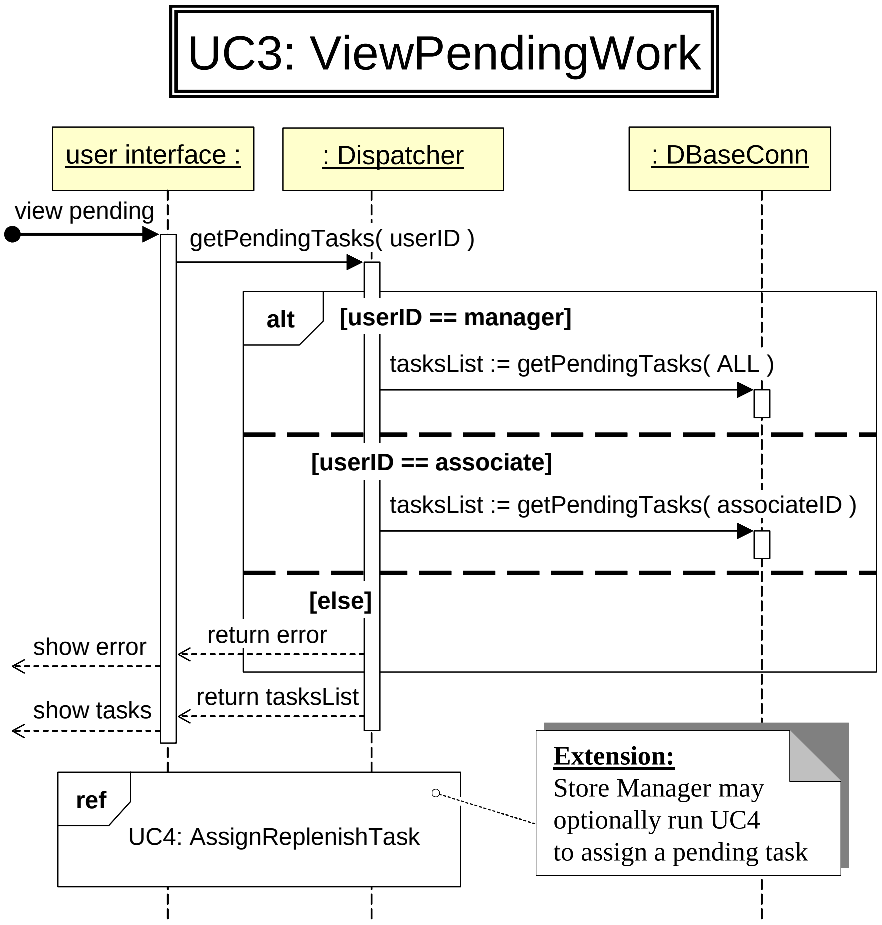
Another simplification is to check only for "out-of-stock" events and not for "low-stock" events. If the customer demands that "low-stock" events be included, then the design of the software-to-be will become somewhat more complex.

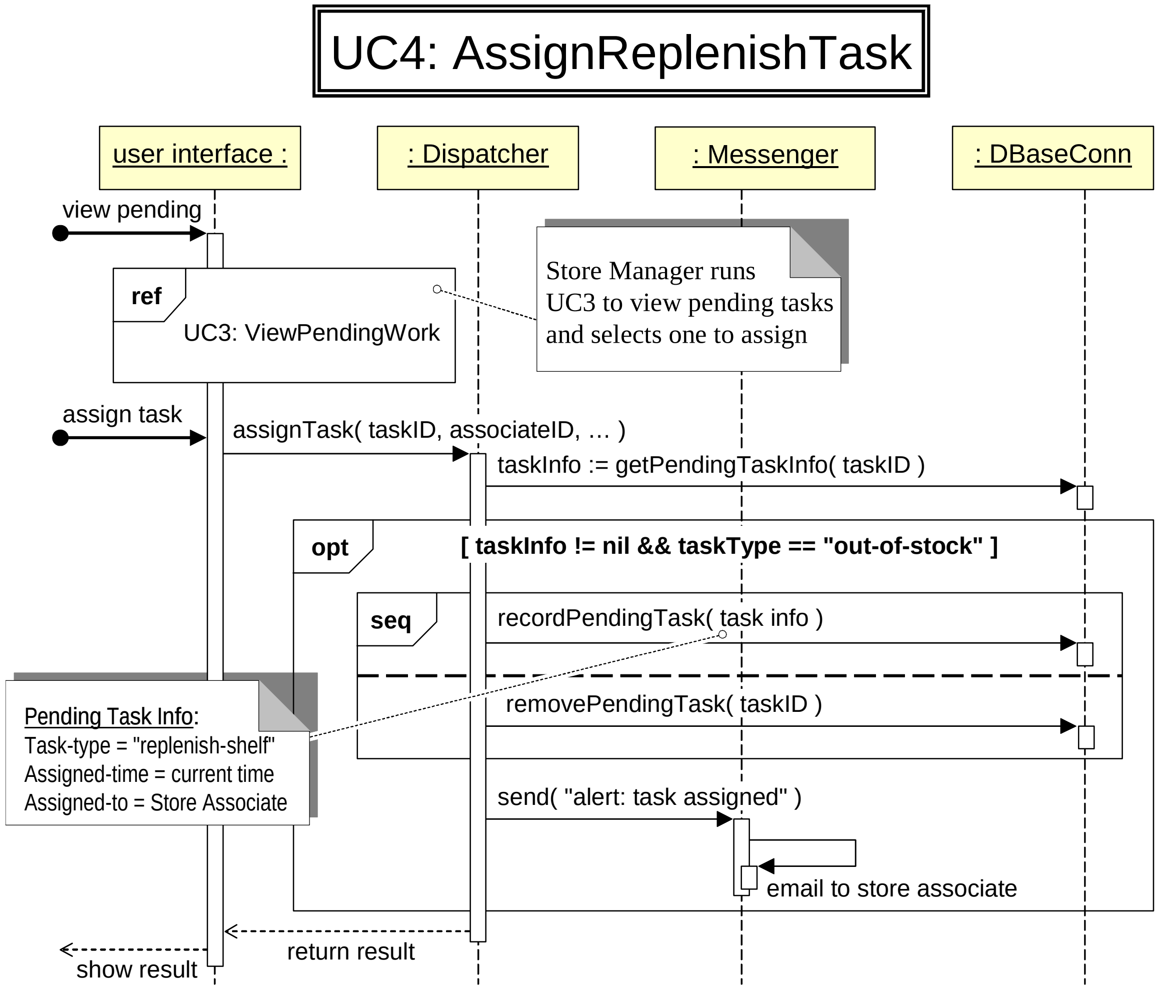
The UML sequence diagrams for all the use cases are shown in the following figures. Notice that use cases UC-3, UC-4, and UC-6 «include» UC-7: Login (user authentication), which is not shown to avoid clutter.


UC1: RemoveItem
rfid : ReaderIface
: Messenger
: Dispatcher
: DBaseConn
receive(event)
prodInfo := getProductInfo(tagID)
opt
[prodInfo == nil]
recordStatistics("error: unknown tagID")
return
decrement(prodCount)
alt
[prodCount < 0]
recordStatistics("error: negative prodCount")
send("error: negative prodCount")
email to store manager
return
[prodCount ≤ Threshold]
recordProductInfo(updated product count)
createTask("out-of-stock")
recordPendingTask(task info)
send("alert: out-of-stock")
email to store manager
Pending Task Info:
Task-type = "out-of-stock"
Assigned-time = current time
Assigned-to = Store Manager
[else]
recordProductInfo(updated product count)
return

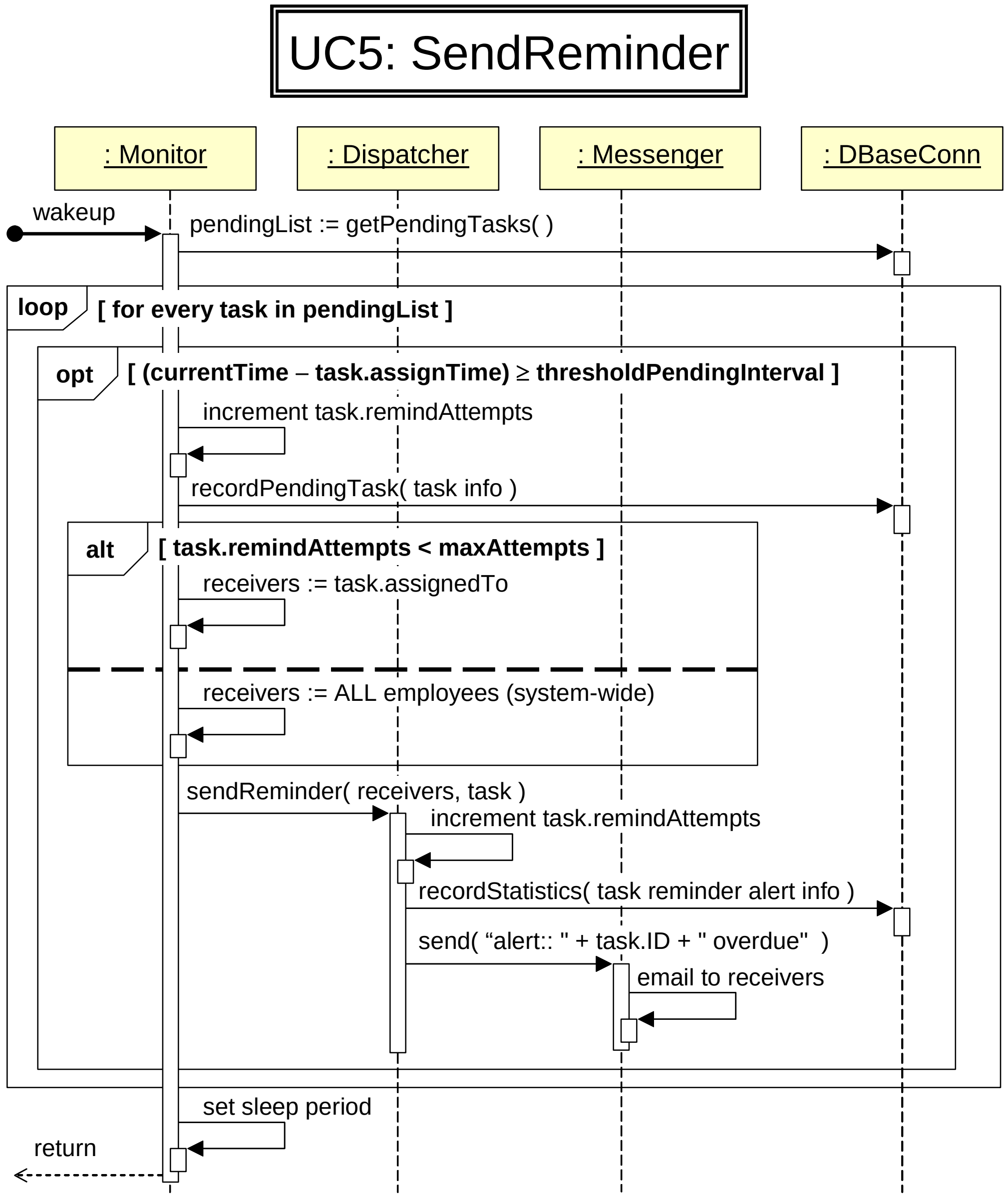
In the design for UC-1, the system may check if a pending task for the given product already exists in the database; if yes, it should not generate a new pending task for the same product.


UC2: AddItem
rfid : ReaderIface
: DBaseConn
receive(event)
prodInfo := getProductInfo(tagID)
alt
[prodInfo == nil]
recordStatistics("error: unknown tagID")
return
[else]
increment(prodCount)
recordProductInfo(updated product count)
return


UC3: ViewPendingWork
user interface :
: Dispatcher
: DBaseConn
view pending
getPendingTasks(userID)
alt
[userID == manager]
tasksList := getPendingTasks(ALL)
[userID == associate]
tasksList := getPendingTasks(associateID)
[else]
return error
show error
return tasksList
show tasks
ref
UC4: AssignReplenishTask
Extension:
Store Manager may optionally run UC4 to assign a pending task


UC4: AssignReplenishTask
user interface :
: Dispatcher
: Messenger
: DBaseConn
view pending
ref
UC3: ViewPendingWork
Store Manager runs UC3 to view pending tasks and selects one to assign
assign task
assignTask(taskID, associateID, …)
taskInfo := getPendingTaskInfo(taskID)
opt
[taskInfo != nil && taskType == "out-of-stock"]
seq
recordPendingTask(task info)
removePendingTask(taskID)
Pending Task Info:
Task-type = "replenish-shelf"
Assigned-time = current time
Assigned-to = Store Associate
send("alert: task assigned")
email to store associate
return result
show result

The [seq] interaction fragment specifies that the interactions contained within the fragment box must occur exactly in the given order. The reason for using this constraint in UC-4 is that the system may crash while the task is being converted from unassigned to pending. If `removePendingTask()` were called first, and the system crashed after it but before `recordPendingTask()`, then all information about this task would be lost! Depending on the database implementation, it may be possible to perform these operations as atomic for they update the same table in the database. To deal with crash scenarios where a task ends up in both tables, the Monitor object in UC-5 SendReminder should be extended to perform a database clean-up after a crash. It should remove those tasks from the `PendingTasks` table that are marked both as unassigned and pending.

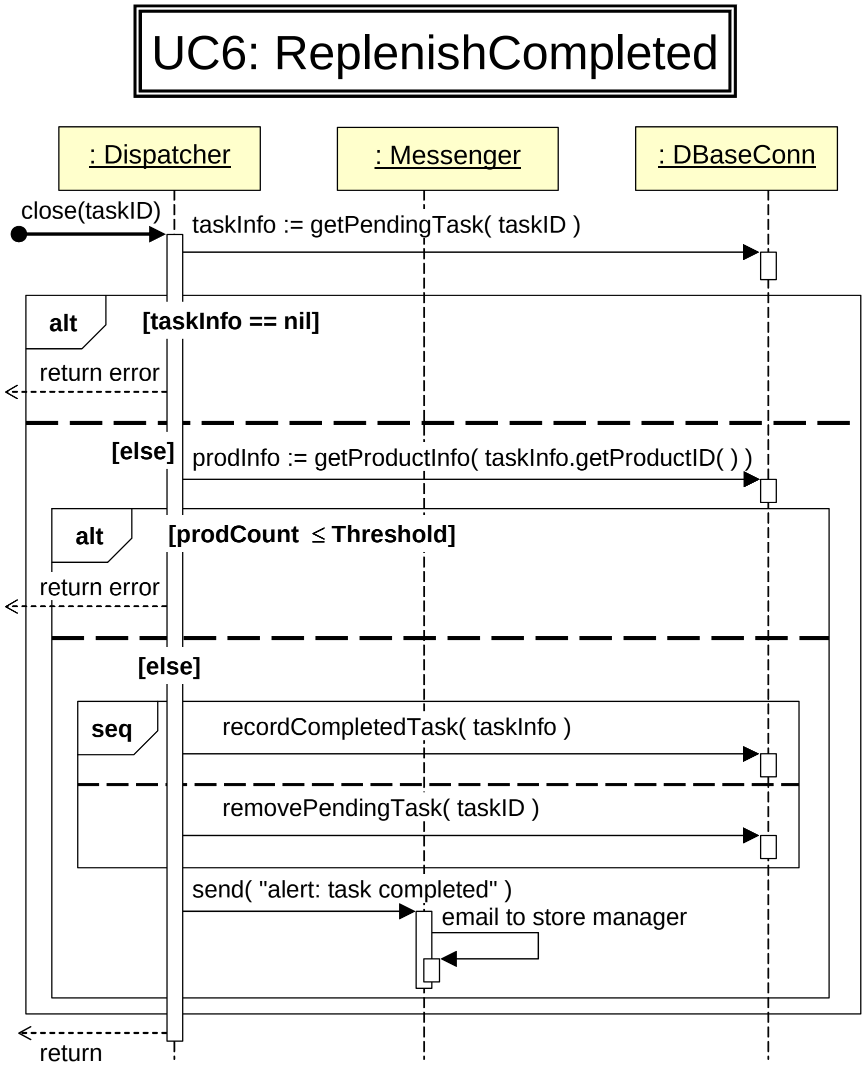

UC5: SendReminder
: Monitor
: Dispatcher
: Messenger
: DBaseConn
wakeup
pendingList := getPendingTasks( )
loop
[for every task in pendingList]
opt
[(currentTime – task.assignTime) ≥ thresholdPendingInterval]
increment task.remindAttempts
recordPendingTask(task info)
alt
[task.remindAttempts < maxAttempts]
receivers := task.assignedTo
receivers := ALL employees (system-wide)
sendReminder(receivers, task)
increment task.remindAttempts
recordStatistics(task reminder alert info)
send(“alert:: " + task.ID + " overdue")
email to receivers
set sleep period
return

Note that the Monitor discards the list of pending tasks before going to sleep, so it starts every cycle with a fresh list of pending tasks, retrieved from the database, because our assumption is that the database contains the most current information (possibly updated by other objects).

By examining the design for the use case UC-5 SendReminder, we see that the Monitor has to do a lot of subtractions and comparisons every time it wakes up, but this can be done at leisure because seconds or minutes are not critical for this activity. The computing power should be better used for other use cases. Of course, we must ensure that the Monitor still works fast enough not to introduce delays on the order of hours or days during each cycle!

In addition, we need to handle the case where the time values are not incrementing constantly upwards, such as when a full 24 hours passes and a new day starts, the time resets to zero. In Java, using `java.lang.System.currentTimeMillis()` returns the current time in milliseconds as a long integer.

In the solution of Problem 2.16 we discussed using adaptive timeout calculation to adjust the frequency of reminders for busy periods. Another option is to have shorter sleep periods, but during each wakeup, process only part of the list of pending tasks, and leave the rest for subsequent wakeups. Then cycle again from the head of the list. This way, the reminders will be spread over time and not all reminders will be generated at once (avoid generating one “bulk” notification each period).



The logic of UC-6 is that it first retrieves the task, checks if such a task exists, makes sure it is really done, and finally marks it as completed. The [seq] interaction fragment specifies that the interactions contained within the fragment box must occur exactly in the given order. Similar to UC-4 AssignReplenishTask, this constraint is needed in case the system crashes while the task is being closed. If `removePendingTask()` were called first, and the system crashed after it but before `recordCompletedTask()`, then all information about this task would be lost! These operations cannot be performed as atomic, because they work on different tables in the database. To deal with crash scenarios where a task ends up in both tables, the Monitor object in UC-5 should be modified to perform a database clean-up after a crash. It should remove those tasks from the `PendingTasks` table that are already in the `CompletedTasks` table.

Notice also that the Monitor runs in a separate thread, so while UC-6 is in the process of closing a task, the Monitor may send an unnecessary reminder about this task (in UC-5).

Carefully examine the existing design and identify as many opportunities as you can to improve the design by applying design patterns. Note that the existing design ignores the issue of concurrency, but we will leave the multithreading issue aside for now and focus only on the patterns that improve the quality of software design. (The concurrency issues will be considered later in Problem 5.20.)

(a) If you introduce a pattern, first provide arguments why the existing design may be problematic.

(b) Provide as much details as possible about how the pattern will be implemented and how the new design will work (draw UML sequence diagrams or write pseudo-code).

(c) Explain how the pattern improved the design (i.e., what are the expected benefits compared to the original design).

If considering future evolution and extensions of the system when proposing a modification, then describe explicitly what new features will likely be added and how the existing design would be inadequate to cope with resulting changes. Then introduce a design pattern and explain how the modified version is better.

If you believe that the existing design (or some parts of it) is sufficiently good then explain how the application of some design patterns would make the design worse. Use concrete examples and UML diagrams or pseudo-code to illustrate and refer to specific qualities of software design.

Problem 5.11

Problem 5.12

Problem 5.13

Problem 5.14

Problem 5.15

In Section 5.3, it was stated that the standard Java idiom for condition synchronization is the statement:

```
while (condition) sharedObject.wait();
```

(a) Is it correct to substitute the `yield()` method call for `wait()`? Explain your answer and discuss any issues arising from the substitution.

(b) Suppose that `if` substitutes for `while`, so we have:

```
        if (condition) sharedObject.wait()
```

Is this correct? Explain your answer.

Problem 5.16

Parking lot occupancy monitoring, see Figure 5-34. Consider a parking lot with the total number of spaces equal to `capacity`. There is a single barrier gate with two poles, one for the entrance and the other for the exit. A computer in the barrier gate runs a single program which controls both poles. The program counts the current number of free spaces, denoted by `occupancy`, such that

$$0 \leq \texttt{occupancy} \leq \texttt{capacity}$$

When a new car enters, the occupancy is incremented by one; conversely, when a car exits, the occupancy is decremented by one. If `occupancy` equals `capacity`, the red light should turn on to indicate that the parking is full.

In order to be able to serve an entering and an exiting patron in parallel, you should design a system which runs in two threads. `EnterThread` controls the entrance gate and `ExitThread` controls the exit gate. The threads share the `occupancy` counter so to correctly indicate the parking-full state. Complete the UML sequence diagram in Figure 5-35 that shows how the two threads update the shared variable, i.e., `occupancy`.

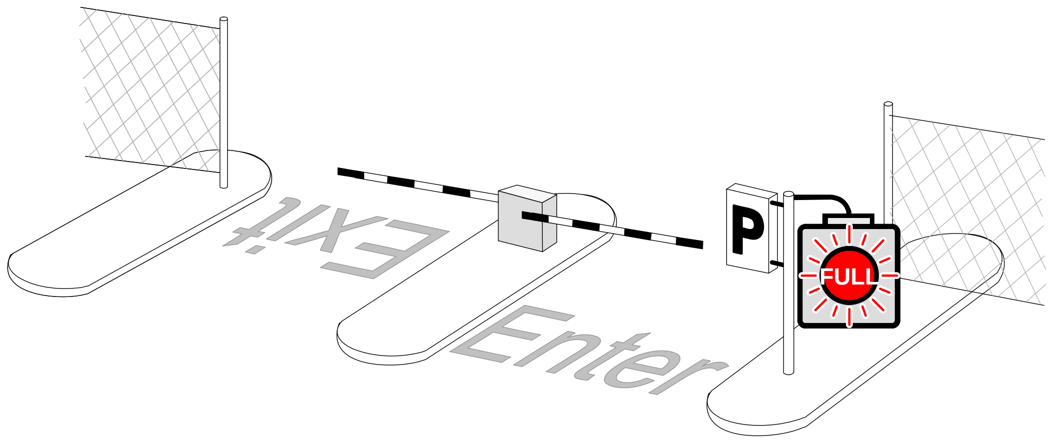


Figure 5-34: Parking lot occupancy monitoring, Problem 5.16.

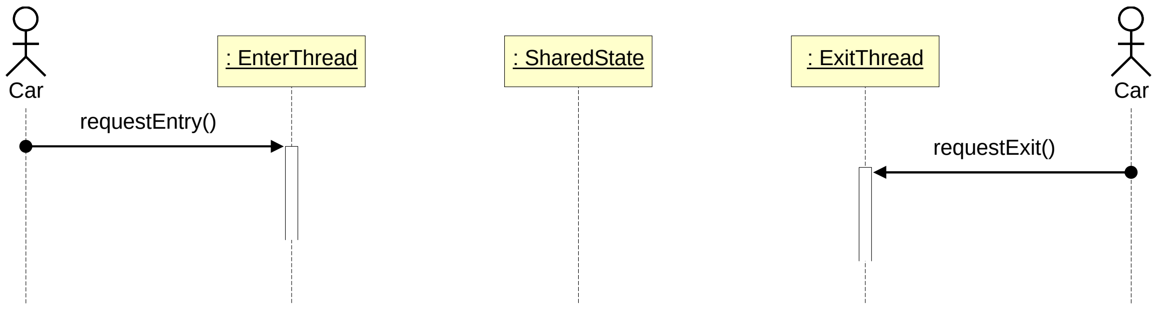


Figure 5-35: UML diagram template for parking lot occupancy monitoring, Problem 5.16.

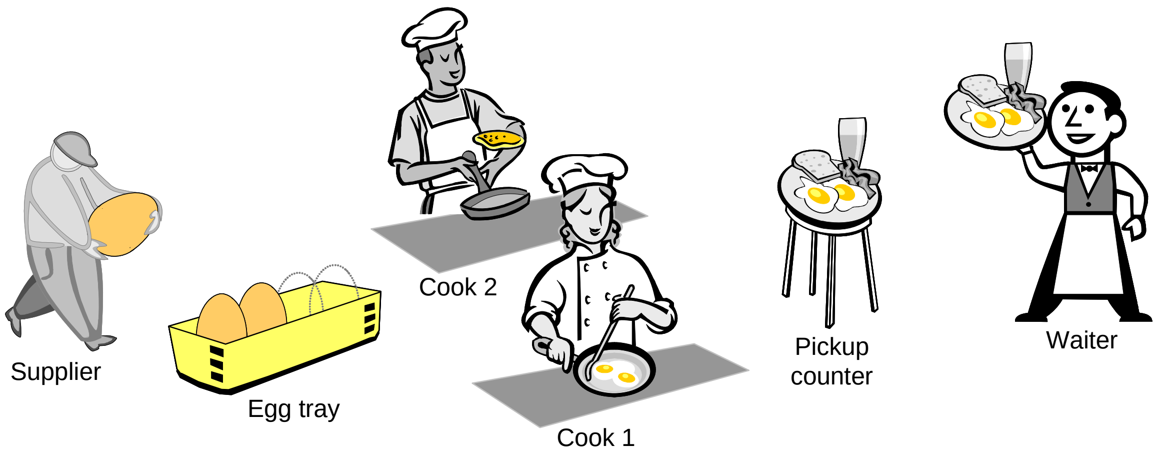


Figure 5-36: Concurrency problem in a restaurant scenario, Problem 5.17.

Hint: Your key concern is to maintain the consistent shared state (`occupancy`) and indicate when the parking-full sign should be posted. Extraneous actions, such as issuing the ticket for an entering patron and processing the payment for an exiting patron, should not be paid attention—only make a high-level remark where appropriate.

Problem 5.17

Consider a restaurant scenario shown in Figure 5-36. You are to write a simulation in Java such that each person runs in a different thread. Assume that each person takes different amount of time to complete their task. The egg tray and the pickup counter have limited capacities, N_{eggs} and N_{plates}, respectively. The supplier stocks the egg tray but must wait if there are no free slots. Likewise, the cooks must hold the prepared meal if the pickup counter is full.

Problem 5.18

A *priority inversion* occurs when a higher-priority thread is waiting for a lower-priority thread to finish processing of a critical region that is shared by both. Although higher-priority threads normally preempt lower-priority threads, this is not possible when both share the same critical region. While the higher-priority thread is waiting, a third thread, whose priority is between the first two, but it does not share the critical region, preempts the low-priority thread. Now the higher-priority thread is waiting for more than one lower-priority thread. Search the literature and describe precisely a possible mechanism to avoid priority inversion.

Problem 5.19

Assume that the patient device described in Problem 2.3 (at the end of Chapter 2) runs in a multi-threaded mode, where different threads acquire and process data from different sensors. (See also Problem 2.35 and its solution on the back of this book.) What do you believe is the optimal number of threads? When designing this system, what kind of race conditions or other concurrency issues can you think of? Propose a specific solution for each issue that you identify (draw UML sequence diagrams or write pseudo-code).

Problem 5.20

Consider the supermarket inventory management system from Problem 5.10. A first observation is that the existing design ignores the issue of concurrency—there will be many users simultaneously removing items, and/or several associates may be simultaneously restocking the shelves. Also, it is possible that several employees may simultaneously wish to view pending tasks, assign replenishment tasks, or report replenishment completed. Clearly, it is necessary to introduce multithreading even if the present system will never be extended with new features. Modify the existing design and introduce multithreading.

Problem 5.21

Problem 5.22

Use Java RMI to implement a *distributed* Publisher-Subscriber design pattern.

Requirements: The publisher and subscribers are to be run on different machines. The naming server should be used for rendezvous only; after the first query to the naming server, the publisher should cache the contact information locally.

Handle sudden (unannounced) departures of subscribers by implementing a heartbeat protocol.

Problem 5.23

Suppose that you are designing an online grocery store. The only supported payment method will be using credit cards. The information exchanges between the parties are shown in Figure 5-37. After making the selection of items for purchase, the customer will be prompted to enter information about his/her credit card account. The grocery store (merchant) should obtain this information and relay it to the bank for the transaction authorization.

In order to provide secure communication, you should design a *public-key cryptosystem* as follows. All messages between the involved parties must be encrypted for confidentiality, so that only the appropriate parties can read the messages. Even the information about the purchased items, payment amount, and the outcome of credit-card authorization request should be kept confidential. Only the initial catalog information is not confidential.

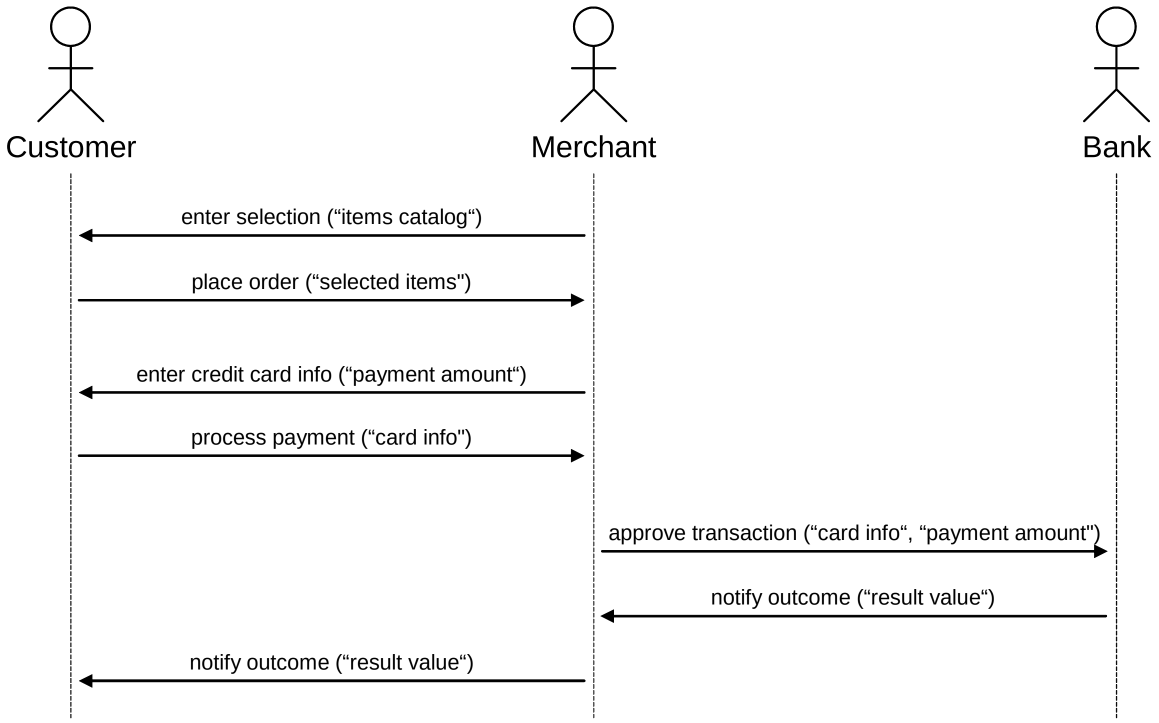

Customer
Merchant
Bank
enter selection (“items catalog“)
place order (“selected items")
enter credit card info (“payment amount“)
process payment (“card info")
approve transaction (“card info“, “payment amount")
notify outcome (“result value“)
notify outcome (“result value“)

Figure 5-37: Information exchanges between the relevant parties. The quoted variables in the parentheses represent the parameters that are passed on when the operation is invoked.

The credit card information must be encrypted by the customer so that only the bank can read it—the merchant should relay it without being able to view the credit card information. For the sake of simplicity, assume that all credit cards are issued by a single bank.

The message from the bank containing binary decision (“approved” or “rejected”) will be sent to the merchant, who will forward it securely to the customer. *Both* the merchant and customer should be able to read it.

Answer the following questions about the cryptosystem that is to be developed:

(a) What is the (minimum) total number of public-private key pairs (K_i^+ , K_i^-) that must be issued? In other words, which actors need to possess a key pair, or perhaps some actors need more than one pair?

(b) For each key pair *i*, specify which actor should issue this pair, to whom the public key K_i^+ should be distributed, and at what time (prior to each shopping session or once for multiple sessions). Provide an explanation for your answer!

(c) For each key pair *i*, show which actor holds the public key K_i^+ and which actor holds the private key K_i^- .

(d) For every message in Figure 5-37, show exactly which key K_i^+ / K_i^- should be used in the encryption of the message and which key should be used in its decryption.

Problem 5.24

In the Model-View-Controller design pattern, discuss the merits of having Model subscribe to the Controller using the Publish-Subscribe design pattern? Argue whether Controller should subscribe to the View?