CONTENTS

1 INTRODUCTION TO PATTERNS *1*

Why Patterns? *1*
Why Design Patterns? *2*
Why Java? *5*
Why UML? *6*
Why a Workbook? *6*
The Organization of This Book *7*
Welcome to Oozinoz! *8*
Source Code Disclaimer *9*
Summary *9*

6 BRIDGE *65*

A Classic Example of BRIDGE: Drivers *65*
Refactoring to BRIDGE *70*
A Bridge Using the List Interface *73*
Summary *74*

22 STATE *225*

Modeling States *225*
Refactoring to STATE *229*
Making States Constant *234*
Summary *235*

B APPENDIX B: SOLUTIONS *359*

1 INTRODUCTION TO PATTERNS

THIS BOOK is for developers who know Java and who have had some exposure to the book *Design Patterns* (Gamma et al. 1995). The premise of this book is that you want to

- Deepen your understanding of the patterns that *Design Patterns* describes
- Build confidence in your ability to recognize these patterns
- Strengthen your ability to apply these patterns in your own Java programs

Why Patterns?

A **pattern** is a way of doing something, or a way of pursuing an intent. This idea applies to cooking, making fireworks, developing software, and to any other craft. In any craft that is mature or that is starting to mature, you can find common, effective methods for achieving aims and solving problems in various contexts. The community of people who practice a craft usually invent jargon that helps them talk about their craft. This jargon often refers to patterns, or standardized ways of achieving certain aims. Writers document these patterns, helping to standardize the jargon. Writers also ensure that the accumulated wisdom of a craft is available to future generations of practitioners.

Christopher Alexander was one of the first writers to encapsulate a craft's best practices by documenting its patterns. His work relates to architecture—of buildings, not software. *A Pattern Language: Towns, Buildings,*

Construction (Alexander, Ishikawa, and Silverstein 1977) provides patterns for architecting successful buildings and towns. Alexander's writing is powerful and has influenced the software community, partially because of the way he looks at intent.

You might state the intent of architectural patterns as "to design buildings." But Alexander makes it clear that the intent of architectural patterns is to serve and to inspire the people who will occupy buildings and towns. Alexander's work showed that patterns are an excellent way to capture and to convey the wisdom of a craft. He also established that properly perceiving and documenting the intent of a craft is a critical, philosophical, and elusive challenge.

The software community has resonated with Alexander's approach and has created many books that document patterns of software development. These books record best practices for software process, software analysis, and high-level and class-level design. Table 1.1 lists books that record best practices in various aspects of software development. This list of books is not comprehensive, and new books appear every year. If you are choosing a book about patterns to read you should spend some time reading reviews of available books and try to select the book that will help you the most.

Why Design Patterns?

A **design pattern** is a pattern—a way to pursue an intent—that uses classes and their methods in an object-oriented language. Developers often start thinking about design after learning a programming language and writing code for a while. You might notice that someone else's code seems simpler and works better than yours does, and you might wonder how that person achieves this simplicity. Design patterns are a level up from code and typically show how to achieve a goal, using one to ten classes. Other people have figured out how to program effectively in object-oriented languages. If you want to become a powerful Java programmer, you should study design patterns, especially those in *Design Patterns*.

TABLE 1.1: Books Conveying Software Development Wisdom in the Form of Patterns

PATTERN CATEGORY	TITLE	AUTHORS/EDITORS
SOFTWARE PROCESS	*Process Patterns: Building Large-Scale Systems Using Object Technology*	Scott W. Ambler
	More Process Patterns: Delivering Large-Scale Systems Using Object Technology	Scott W. Ambler
OBJECT MODELING	*Analysis Patterns: Reusable Object Models*	Martin Fowler
	Object Models: Strategies, Patterns and Applications	Peter Coad Mark Mayfield David North
ARCHITECTURE	*CORBA Design Patterns*	Thomas J. Mowbray Raphael C. Malveau
	Core J2EE™ Patterns: Best Practices and Design Strategies	Deepak Alur John Crupi Dan Malks
	Pattern-Oriented Software Architecture, Volume 1: A System of Patterns	Frank Buschmann Regine Meunier Hans Rohnert Peter Sommerlad Michael Stal
	Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects	Douglas Schmidt Michael Stal Hans Rohnert Frank Buschmann
DESIGN	*AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*	William J. Brown Raphael C. Malveau Hays W. McCormick III Thomas J. Mowbray
	Applying UML and Patterns, Second Edition	Craig Larman
	Concurrent Programming in Java™, Second Edition: Design Principles and Patterns	Doug Lea
	Design Patterns	Erich Gamma Richard Helm Ralph Johnson John Vlissides
	Design Patterns for Object-Oriented Software Development	Wolfgang Pree
	Pattern Hatching: Design Patterns Applied	John Vlissides
	SanFranciso™ Design Patterns	James Carey Brent Carlson Tim Graser

TABLE 1.1: Books Conveying Software Development Wisdom in the Form of Patterns, continued

PATTERN CATEGORY	TITLE	AUTHORS/EDITORS
SMALLTALK ORIENTED	*The Design Patterns Smalltalk Companion*	Sherman R. Alpert Kyle Brown Bobby Woolf
	Smalltalk Best Practice Patterns	Kent Beck
JAVA ORIENTED	*Java™ Design Patterns: A Tutorial*	James W. Cooper
	Patterns in Java™, Volume 1	Mark Grand
COMPENDIA	*The Pattern Almanac 2000*	Linda Rising
	Pattern Languages of Program Design	James O. Coplien Douglas C. Schmidt
	Pattern Languages of Program Design 2	John M. Vlissides James O. Coplien Norman Kerth
	Pattern Languages of Program Design 3	Robert C. Martin Dirk Riehle Frank Buschmann
	Pattern Languages of Program Design 4	Neil Harrison Brian Foote Hans Rohnert

Design Patterns describes 23 design patterns—that is, 23 ways of pursuing an intent, using classes and objects in an object-oriented language. These are probably not absolutely the most useful 23 design patterns to know. On the other hand, these patterns are probably among the 100 most useful patterns. Unfortunately, no set of criteria establishes the value of a pattern, and so the identity of the other 77 patterns in the top 100 is a mystery. Fortunately, the authors of *Design Patterns* chose well, and the patterns they document are certainly worth learning.

GoF

You may have noted the potential confusion between design patterns the topic and *Design Patterns* the book. To distinguish between the topic and the book title, many speakers and some writers refer to the book as the "Gang of Four" book or the "GoF" book, referring to the number of its authors. In print, this distinction is not so confusing. Accordingly, this book avoids using the term "GoF."

Why Java?

This book gives its examples in Java because Java is popular and important and will probably be the basis of future generations of computer languages. The popularity of a language is recursive. Developers invest their learning cycles in technology that they believe will last for at least a few years. The more popular a technology becomes, the more people want to learn it, and the more popular it becomes. This can lead to **hype**, or overexcitement about a technology's potential value. But Java is more than hype.

At a superficial level, Java is important because it is popular, but Java is also popular because it is a stride forward in computer languages. Java is a **consolidation language**, having absorbed the strengths and discarded the weaknesses of its predecessors. This consolidation has fueled Java's popularity and helps ensure that future languages will evolve from Java rather than depart radically from it. Your investment in Java will almost surely yield value in any language that supplants Java.

The patterns in *Design Patterns* apply to Java because, like Smalltalk and C++, Java follows a class/instance paradigm. Java is much more similar to Smalltalk and C++ than it is to, say, Prolog or Self. Although competing paradigms are important, the class/instance paradigm appears to be the most practical next step in applied computing. This book uses Java because of Java's popularity and because Java appears to lie along the evolutionary path of languages that we will use for decades ahead.

Why UML?

Where challenges have solutions in code, this book uses Java. But many of the challenges ask you to draw a diagram of how classes, packages, and other elements relate. You can use any notation you like, but this book uses **Unified Modeling Language** (**UML**) notation. Even if you are familiar with UML, it is a good idea to have a reference handy. Two good choices are *The UML User Guide* (Booch, Rumbaugh, and Jacobson 1999), and *UML Distilled* (Fowler with Scott 2000). The bare minimum of UML knowledge you need for this book is provided in Appendix C, UML at a Glance, page 441.

Why a Workbook?

No matter how much you read about doing something, you won't feel as though you know it until you do it. This is true partially because until you exercise the knowledge you gain from a book, you won't encounter subtleties, and you won't grapple with alternative approaches. You won't feel confident about design patterns until you apply them to some real challenges.

The problem with learning through experience is that you can do a lot of damage as you learn. You can't apply patterns in production code before you are confident in your own skills. But you need to start applying patterns to gain confidence. What a conundrum! The solution is to practice on example problems where mistakes are valuable but painless.

Each chapter in this **workbook** begins with a short introduction and then sets up a series of challenges for you to solve. After you come up with a solution, you can compare your solution to one given in Appendix B, Solutions, starting on page 359. The solution in the book may take a different slant from your solution or may provide you with some other insight.

You probably can't go overboard in how hard you work to come up with answers to the challenges in this book. If you consult other books, work with a colleague, and write sample code to check out your solution, terrific! You will never regret investing your time and energy in learning how to apply design patterns.

A danger lurks in the solutions that this book provides. If you flip to the solution immediately after reading a challenge, you will not gain much from this book. The solutions in this book can do you more harm than good if you don't first create your own solutions.

The Organization of This Book

There are many ways to organize and to categorize patterns. You might organize them according to similarities in structure, or you might follow the order in *Design Patterns*. But the most important aspect of any pattern is its intent, that is, the potential value of applying the pattern. This book organizes the 23 patterns of *Design Patterns* according to their intent.

Having decided to organize patterns by intent raises the question of how to categorize intent. This book adopts the notion that the intent of a design pattern is usually easily expressed as the need to go beyond the ordinary facilities that are built into Java. For example, Java has plentiful support for defining the interfaces that a class implements. But if you want to adapt a class's interface to meet the needs of a client, you need to apply the ADAPTER pattern. The intent of the ADAPTER pattern goes beyond the interface facilities built into Java.

This book places design pattern intent in five categories, as follows:

- Interfaces
- Responsibility
- Construction
- Operations
- Extensions

These five categories account for five parts of this book. Each part begins with a chapter that discusses and presents challenges related to features built into Java. For example, Part I, Interface Patterns, begins with a chapter on ordinary Java interfaces. That chapter will challenge your understanding of the Java interface construct, especially in comparison to abstract classes. The remaining chapters of Part I address patterns whose primary intent involves the definition of an interface—the set of methods

TABLE 1.2: Categorization of Patterns by Intent

INTENT	PATTERNS
INTERFACES	ADAPTER, FACADE, COMPOSITE, BRIDGE
RESPONSIBILITY	SINGLETON, OBSERVER, MEDIATOR, PROXY, CHAIN OF RESPONSIBILITY, FLYWEIGHT
CONSTRUCTION	BUILDER, FACTORY METHOD, ABSTRACT FACTORY, PROTOTYPE, MEMENTO
OPERATIONS	TEMPLATE METHOD, STATE, STRATEGY, COMMAND, INTERPRETER
EXTENSIONS	DECORATOR, ITERATOR, VISITOR

that a client can call from a service provider. Each of these patterns addresses a need that cannot be addressed solely with Java interfaces.

Categorizing patterns by intent does not mean that each pattern support only one type of intent. A pattern that supports more than one type of intent appears as a full chapter in the first part to which it applies and gets a brief mention in subsequent sections. Table 1.2 shows the categorization behind the organization of this book.

I hope that you will question the categorization in Table 1.2. Do you agree that SINGLETON is about responsibility, not construction? Do you think that COMPOSITE is an interface pattern? Categorizing patterns is somewhat subjective. But I hope that you will agree that thinking about the intent behind patterns and thinking about how you will apply patterns are very useful exercises.

Welcome to Oozinoz!

The challenges in this book all cite examples from **Oozinoz**, a fictional company that manufactures and sells fireworks and puts on fireworks displays. (Oozinoz takes its name from the sounds heard at Oozinoz exhibitions.) The current code base at Oozinoz is pretty well designed, but many challenges remain for you to make the code stronger by applying design patterns.

Source Code Disclaimer

The source code used in this book is available at `www.oozinoz.com`. The code is free. You may use it as you wish, with the sole restriction that you may not claim that you wrote it. On the other hand, neither I nor the publisher of this book warrant the code to be useful for any particular purpose. If you use the `oozinoz` code, I hope that you will thoroughly test that it works properly with your application. And if you find a defect in my code, please let me know! I can be contacted at Steve.Metsker@acm.org.

Summary

Patterns are distillations of accumulated wisdom, providing a standard jargon and naming the concepts that experienced practitioners apply. The patterns in *Design Patterns* are among the most useful class-level patterns and are certainly worth learning. This book complements *Design Patterns*, providing challenges to exercise your understanding of the patterns. This book uses Java in its examples and challenges because of Java's popularity and its future prospects. By working through the challenges in this book, you will learn to recognize and to apply a large portion of the accumulated wisdom of the software community.

6
BRIDGE

THE BRIDGE PATTERN focuses on the implementation of an **abstraction**. *Design Patterns* (Gamma et al. 1995) uses the word *abstraction* to refer to a class that relies on a set of abstract operations, where several implementations of the set of abstract operations are possible.

The ordinary way to implement an abstraction is to create a class hierarchy, with an abstract class at the top that defines the abstract operations; each subclass in the hierarchy provides a different implementation of the set of abstract operations. This approach becomes insufficient when you need to subclass the hierarchy for another reason. It can also happen that the abstract operations need to be defined and implemented in advance of abstractions that will use them.

You can create a *bridge* by moving the set of abstract operations to an interface, so that an abstraction will depend on an implementation of the interface. The intent of the BRIDGE pattern is to decouple an abstraction from the implementation of its abstract operations, so that the abstraction and its implementation can vary independently.

A Classic Example of BRIDGE: Drivers

A common use of BRIDGE occurs when an application uses a driver. A **driver** is an object that operates a computer system or an external device according to a well-specified interface. Applications that use drivers are *abstractions*: The effect of running the application depends on which driver is in place. Each driver is an instance of the ADAPTER pattern, providing the interface a client expects, using the services of a class with a

different interface. An overall design that uses drivers is an instance of BRIDGE. The design separates application development from the development of the drivers that implement the abstract operations on which the applications rely.

An everyday example of applications using drivers to operate computer systems appears in database access. Database connectivity in Java usually depends on **JDBC**. ("JDBC" is a trademarked name, not an acronym.) A good resource that explains how to apply JDBC is *JDBC Database Access with Java™* (Hamilton, Cattell, and Fisher 1997). In short, JDBC is an **application programming interface** (**API**) for executing **structured query language** (**SQL**) statements. Classes that implement the interface are JDBC drivers, and applications that rely on these drivers are abstractions that can work with any database for which a JDBC driver exists. The JDBC architecture decouples an abstraction from its implementation so that the two can vary independently—an excellent example of BRIDGE.

To use a JDBC driver, you load it, connect to a database, and create a `Statement` object:

```
Class.forName(driverName);
Connection c =
    DriverManager.getConnection(url, user, passwd);
Statement s = c.createStatement();
```

A discussion of how the `DriverManager` class works is outside the scope of a discussion on BRIDGE. The point here is that the variable `s` is a `Statement` object, capable of issuing SQL queries that return result sets:

```
ResultSet r = s.executeQuery(
    "select name, apogee from firework");
while(r.next())
{
    String name = r.getString("name");
    int apogee = r.getInt("apogee");
    System.out.println(name + ", " + apogee);
}
```

CHALLENGE 6.1

Figure 6.1 shows a UML sequence diagram that illustrates the message flow in a typical JDBC application. Fill in the missing type names and the missing message name in this illustration.

A solution appears on page 371.

Figure 6.1 is a **sequence diagram**. You can use a **class diagram** to show the relationship of the application to the driver it applies. Figure 6.2 shows the general form of an abstraction and its implementation in an application of the BRIDGE pattern. This figure brings out the separation of an abstraction and its implementation. The effect of calling the `operation()` method depends on which implementation of the `Implementor` interface is in place.

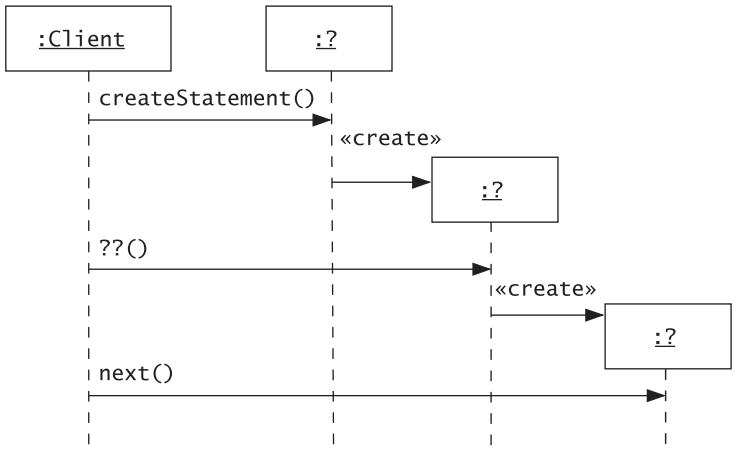

FIGURE 6.1: This diagram shows most of the typical message flow in a JDBC application.

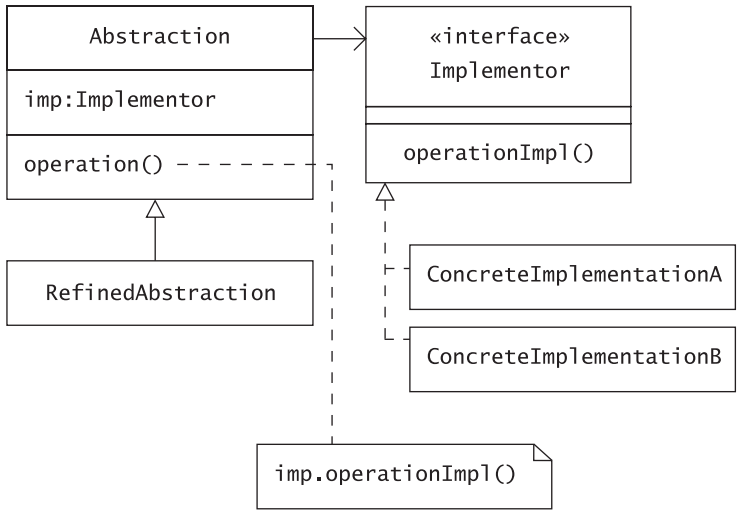

FIGURE 6.2: A BRIDGE structure moves the abstract operations that an abstraction relies on into a separate interface.

It can be challenging to see how the classes in a JDBC application align with the BRIDGE structure that Figure 6.2 shows. The questions to answer are: Which classes are abstractions? Which classes are "implementations"?

Consider the `JDBCAdapter` class that Sun Microsystems packages in the JDK as a demonstration of how to display data from a database table. This class implements the `TableModel` interface as shown in Figure 6.3.

The `JDBCAdapter` class constructor establishes a database connection and creates a `Statement` object. The adapter passes queries to the `Statement` object and makes the results available in a GUI component.

The `JDBCAdapter` class assumes that a client will call `executeQuery()` before making calls to extract data from the query. The results of not providing a query are somewhat unpredictable. Noticing this, suppose that you decide to subclass `JDBCAdapter` with an `OzJDBCAdapter` class that throws a `NoQuery` exception if a client forgets to issue a query before asking for data.

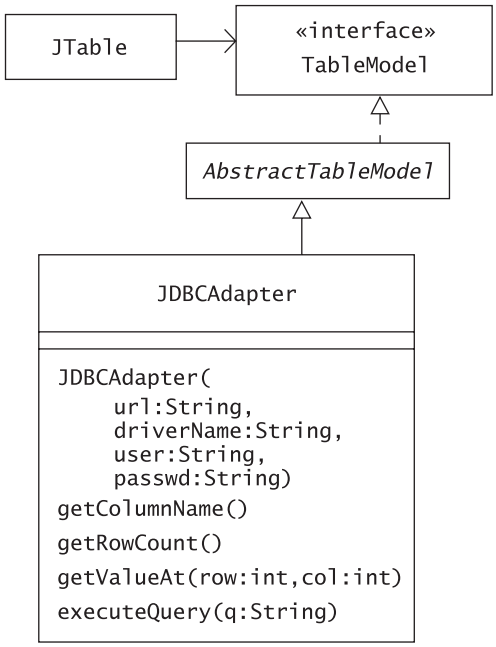

FIGURE 6.3: The `JDBCAdapter` class adapts information in a database table to appear in Swing `JTable` component.

CHALLENGE 6.2

Draw a diagram that shows the relationships among the classes `JDBCAdapter`, `OzJDBCAdapter`, `NoQuery`, and `Statement`.

A solution appears on page 371.

The JDBC architecture clearly divides the roles of the driver writer and the application writer. In some cases, this division will not exist in advance, even if you are using drivers. You may be able to set up drivers as subclasses of an abstract superclass, with each subclass driving a different subsystem. In such a case, you can be forced to set up a BRIDGE when you need more flexibility.

Refactoring to BRIDGE

Sometimes, you have to refactor an abstract class into a BRIDGE to gain more flexibility in the abstraction. Consider the `MachineController` hierarchy at Oozinoz. Other than database drivers, the most common application of drivers is to operate external devices. These devices may be peripherals, such as printers and pen plotters, or any machine capable of communicating with computers. Applications at Oozinoz control most of the machines on the factory floor with the drivers, or controllers, that Figure 6.4 shows.

FIGURE 6.4: Instances of classes in the `MachineController` hierarchy drive machines in the Oozinoz factory.

The `MachineController` class defines the interface for an object that controls a machine. Subclasses provide different implementations of the methods that drive a machine. Most of the methods that `MachineController` declares are abstract, but it does have concrete methods that depend on its **abstract methods**. For example, the `inputFull()` method is concrete, but its effects depend on how subclasses implement `getQueueMax()`:

```
public boolean inputFull()
{
    return getQueueMax() >= getQueue().size();
}
```

The `MachineController` class is an abstraction whose implementation occurs in its subclasses, and we can consider separating the abstraction from its implementation. The motivation to separate an abstraction from its implementation usually occurs when a new reason for subclassing emerges. For example, suppose that you want to differentiate normal controllers from testing controllers. A testing controller has new methods that test or stress machines, such as an `overflow()` method that dispatches material to a machine until the input queue overflows and the machines signals an alarm. Figure 6.5 shows a design that provides for testing classes.

Introducing a `TestController` class turns out to force you to create three new classes, as test controllers for star presses and shell assemblers implement `getQueueMax()` differently. This is a problem because you'd like to be able to add new classes individually. The problem arises because the hierarchy is factored according to two principles: Each machine needs its own controller, and different types of controllers are used for normal operation and for testing. You really need a class for every combination of machine type and controller type—a maintenance nightmare. You can solve the problem by separating the `MachineController` abstraction from its implementation.

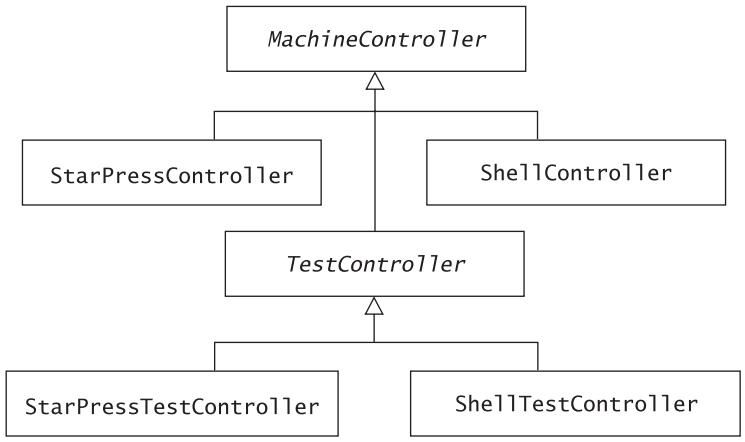

FIGURE 6.5: In this factoring, subclasses of `MachineController` exist for various types of machines and for various types of controllers.

To refactor a hierarchy with an abstract class at its top into a bridge, do the following.

1. Move the abstract operations in the superclass into an interface.
2. Define implementation classes that provide different implementations of the interface.
3. Redefine the remaining operations in the abstract class as operations on an instance of the new interface.

CHALLENGE 6.3

Figure 6.6 shows the `MachineController` hierarchy refactored into a bridge. Fill in the missing labels.

A solution appears on page 373.

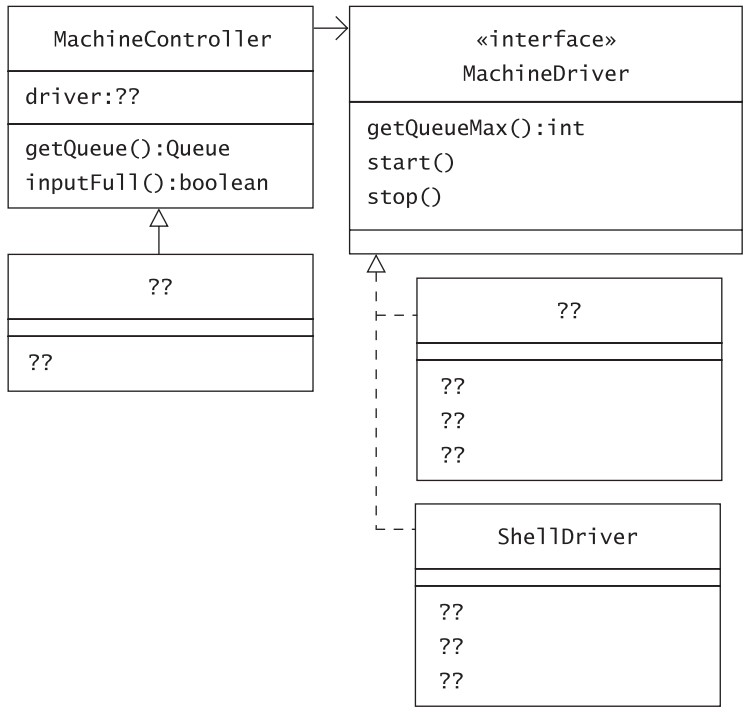

FIGURE 6.6: This diagram, when complete, will show the results of refactoring `MachineController` to apply BRIDGE.

After this type of refactoring, you can usually change the declaration of the abstract superclass to be a **concrete class**. Note, however, that the class is still an *abstraction* in that its operations depend on the implementation of abstract operations defined in the interface.

A Bridge Using the List Interface

Drivers offer good examples of the BRIDGE pattern, but you can apply BRIDGE in other situations, too. Whenever you separate an abstraction—a class that relies on a set of abstract operations—from the implementation of its abstract operations, you have applied BRIDGE.

Consider the `Queue` class in use at Oozinoz. When using a list to model a conveyor, you have discovered that it is easy to get confused about which end of the list applies to which end of a conveyor. To avoid this confusion, Oozinoz developers always define the ends of the conveyor as the conveyor's tail and head and define that the belt moves from the tail toward the head. Placing material on the tail of a conveyor *enqueues* the material on the conveyor; removing material from the head *dequeues* the material from the conveyor. Figure 6.7 shows the `Queue` class. The `add()` method adds an object to the tail of a list, and a call to `remove(0)` removes the head of a list.

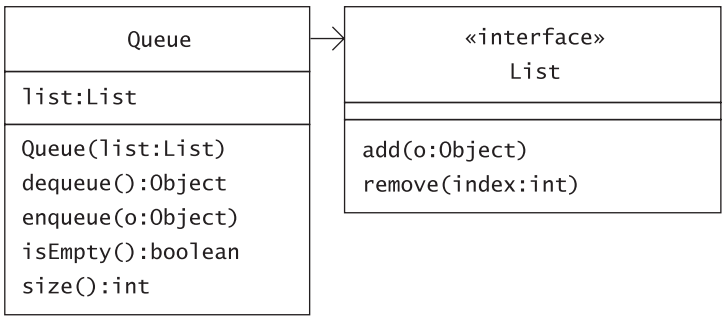

FIGURE 6.7: The `Queue` class defines operations that rely on an abstract `List` object.

CHALLENGE 6.4

Write the code for `dequeue()` and `enqueue()`.

A solution appears on page 373.

Summary

A common example of BRIDGE occurs in drivers. An application that uses a driver is an abstraction—the choice of driver determines what happens when the application runs. The interface between applications and drivers lets you vary either side independently. You can create new applications that use the drivers without changing the drivers. You can also add new drivers without changing existing applications. That is the intent of BRIDGE.

You may encounter abstractions that are not decoupled from their implementation but rather are arranged as an abstract class whose concrete subclasses provide various implementations. In such a case, you can apply BRIDGE if you want to factor the hierarchy along another line. Moving the implementations out lets each implementation become a new class that implements a standard interface. This lets you add new subclasses to the abstraction, regardless of which implementation of the interface you will use. This move also lets you add new implementations of the interface without affecting the abstraction hierarchy. The BRIDGE pattern decouples an abstraction from its implementation so that the two can vary independently.

22
STATE

THE **STATE** of an object is a combination of the current values of its attributes. When you call a `set-` method, you typically change an object's state, and an object can change its own state as its methods execute.

In some cases, an object's state can be a prominent aspect of its behavior, such as when modeling transactions and machines. Logic that depends on the object's state may spread through many of the class's methods. To counter this spread, you can move state-specific behavior into a group of classes, with each class representing a different state. This lets you avoid having deep or complex `if` statements, relying instead on polymorphism to execute the right implementation of an operation. The intent of the STATE pattern is to distribute state-specific logic across classes that represent an object's state.

Modeling States

When you model an object whose state is important, you may find that you have a variable that tracks how the object should behave, depending on its state. This variable may appear in complex, cascading `if` statements that focus on how to react to the events that an object can experience. One problem with this approach to modeling state is that `if` statements can become complex. Another problem is that when you adjust how you model the state, you often have to adjust `if` statements in several methods. The STATE pattern offers a cleaner, simpler approach, using a distributed operation.

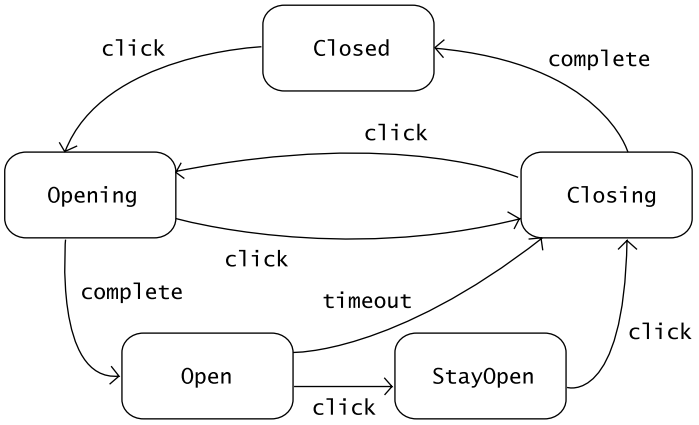

FIGURE 22.1: A carousel door provides one-touch control with a single button that changes the door's state.

Consider the Oozinoz software that models the state of a carousel door. A **carousel** is a large, smart rack that accepts material through a doorway and stores the material according to a bar code ID on the material. The door operates with a single button. If the door is closed, clicking the button makes the door start opening. If you click again before the door opens fully, the door will begin closing. If you let the door open all the way, it will automatically begin closing after a 2-second timeout. You can prevent this by clicking again when the door is open. Figure 22.1 shows the states and transitions of the carousel's door.

The diagram in Figure 22.1 is a UML *state machine*. Such diagrams can be much more informative than a corresponding textual description.

CHALLENGE 22.1

Suppose that you open the door and place a material bin in the doorway. Is there a way to make the door begin closing without waiting for it to time out?

A solution appears on page 415.

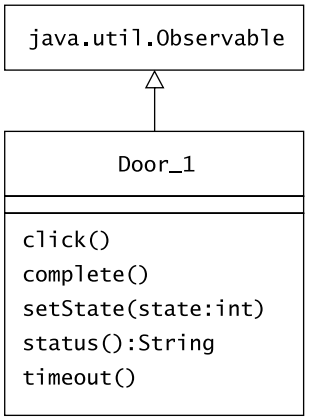

FIGURE 22.2: The `Door_1` class models a carousel door, relying on state change events sent by the carousel machine.

You can supply a `Door_1` object that the carousel software will update with state changes in the carousel. (The underscore in the class name is a hint that we will soon refactor this class.) Figure 22.2 shows the `Door_1` class.

The `Door_1` class subclasses `Observable` so that clients, such as a GUI, can observe a door. The class definition declares its superclass and establishes the states that a door can enter:

```
package com.oozinoz.carousel;
public class Door_1 extends Observable
{
    public static final int CLOSED   = -1;
    public static final int OPENING  = -2;
    public static final int OPEN     = -3;
    public static final int CLOSING  = -4;
    public static final int STAYOPEN = -5;

    private int state = CLOSED;
    //...
}
```

Not surprisingly, a textual description of the state of a door depends on the door's state:

```
public String status()
{
    switch (state)
    {
```

```
            case OPENING :
                return "Opening";
            case OPEN :
                return "Open";
            case CLOSING :
                return "Closing";
            case STAYOPEN :
                return "StayOpen";
            default :
                return "Closed";
        }
    }
```

When a user clicks the carousel's one-touch button, the carousel generates a call to a `Door` object's `click()` method. The `Door_1` code for a state transition mimics the information in Figure 22.1:

```
public void click()
{
    if (state == CLOSED)
    {
        setState(OPENING);
    }
    else if (state == OPENING || state == STAYOPEN)
    {
        setState(CLOSING);
    }
    else if (state == OPEN)
    {
        setState(STAYOPEN);
    }
    else if (state == CLOSING)
    {
        setState(OPENING);
    }
}
```

The `setState()` method of the `Door_1` class notifies observers of the door's change:

```
private void setState(int state)
{
    this.state = state;
    setChanged();
    notifyObservers();
}
```

CHALLENGE 22.2

Write the code for the `complete()` and `timeout()` methods of the `Door_1` class.

Solutions appear on page 415.

Refactoring to STATE

The code for `Door_1` is somewhat complex because the use of the `state` variable is spread throughout the class. In addition, you might find it difficult to compare the state transition methods, particularly `click()`, with the state machine in Figure 22.1. The STATE pattern can help you to simplify this code. To apply STATE in this example, make each state of the door a separate class, as Figure 22.3 shows.

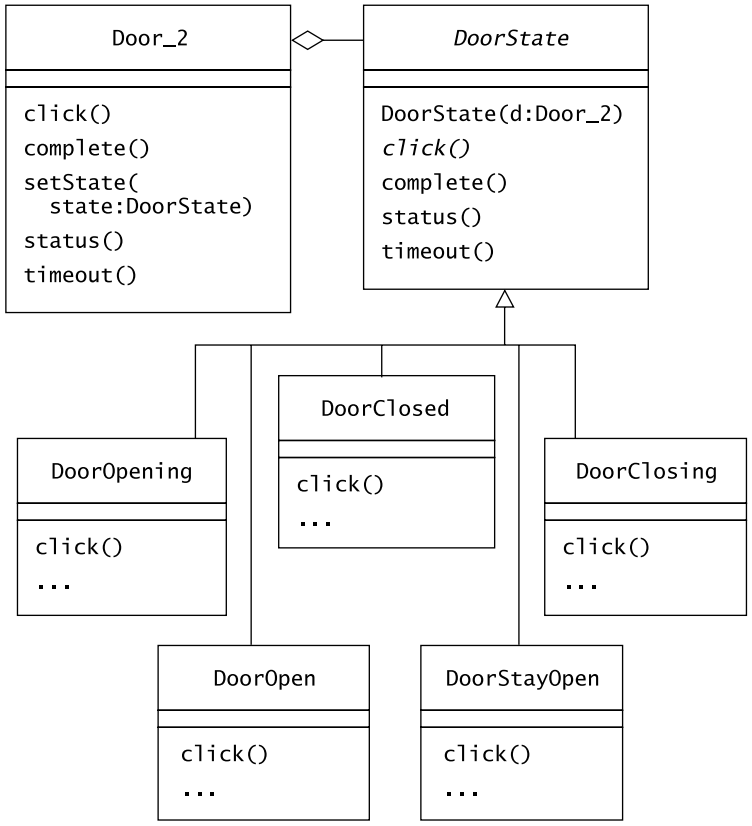

FIGURE 22.3: This diagram shows a door's states as classes in an arrangement that mirrors the door's state machine.

The refactoring that Figure 22.3 shows uses the `Door_2` class to contain the context of the state machine. A **context** is an object that contains information that is environmental and relevant to a group of other objects. In particular, STATE uses a context object to record which instance of `DoorState` is the current state.

The `DoorState` class constructor requires a `Door_2` object. Subclasses of `DoorState` use this object to communicate changes in state back to the door. By giving these classes an attribute that ties them to a specific `Door` object, this design requires that a `DoorState` object be referenced by a single `Door` object. In turn, this requires the `Door` class to define its states as local variables:

```
package com.oozinoz.carousel;
public class Door_2 extends Observable
{
    public final DoorState CLOSED   = new DoorClosed(this);
    public final DoorState OPENING  = new DoorOpening(this);
    public final DoorState OPEN     = new DoorOpen(this);
    public final DoorState CLOSING  = new DoorClosing(this);
    public final DoorState STAYOPEN = new DoorStayOpen(this);
    //
    private DoorState state = CLOSED;
    // ...
}
```

The abstract `DoorState` class requires subclasses to implement `click()`. This is consistent with the state machine, in which every state has a `click()` transition. The `DoorState` class stubs out other transitions, so that subclasses can override or ignore irrelevant messages:

```
public abstract class DoorState
{
    protected Door_2 door;

    public DoorState(Door_2 door)
    {
        this.door = door;
    }

    public abstract void click();

    public void complete()
    {
    }

    public String status()
    {
        String s = getClass().getName();
        return s.substring(s.lastIndexOf('.') + 1);
    }

    public void timeout()
    {
    }
}
```

Note that the `status()` method works for all the states and is much simpler than its predecessor before refactoring.

CHALLENGE 22.3

The new `status()` method returns a slightly different description of a door's state. What's the difference?

Solutions appear on page 415.

The new design doesn't change the role of a `Door` object in receiving state changes from the carousel. But now the `Door_2` object simply passes these changes to its current `state` object:

```
package com.oozinoz.carousel;
public class Door_2 extends Observable
    {
    // ... (DoorState variables)

    public void click()
    {
        state.click();
    }

    public void complete()
    {
        state.complete();
    }

    protected void setState(DoorState state)
    {
        this.state = state;
        setChanged();
        notifyObservers();
    }

    public String status()
    {
        return state.status();
    }
```

```
    public void timeout()
    {
        state.timeout();
    }
}
```

The `click()`, `complete()`, `status()`, and `timeout()` methods show the pure polymorphism of this approach. Each of these methods is still a kind of switch. In each case, the operation is fixed, but the class of the receiver—the class of `state`—varies. The rule of polymorphism is that the method that executes depends on the operation signature and the class of the receiver. What happens when you call `click()`? The answer depends on the door's state. The code still effectively performs a switch, but by relying on polymorphism, the code is simpler than before.

The `setState()` method in the `Door_2` class is now used by subclasses of `DoorState`. These subclasses closely resemble their counterparts in the state machine in Figure 22.1. For example, the code for `DoorOpen` handles calls to `click()` and `timeout()`, the two transitions from the `Open` state in the state machine:

```
package com.oozinoz.carousel;
public class DoorOpen extends DoorState
{
    public DoorOpen(Door_2 door)
    {
        super(door);
    }

    public void click()
    {
        door.setState(door.STAYOPEN);
    }

    public void timeout()
    {
        door.setState(door.CLOSING);
    }
}
```

CHALLENGE 22.4

Write the code for `DoorClosing.java`.

Solutions appear on page 415.

The new design leads to much simpler code, but you might feel a bit dissatisfied that the "constants" that the `Door` class uses are in fact local variables.

Making States Constant

Suppose that you decide to move the state constants to a `DoorConstants` interface. For this approach to work, you have to eliminate the `Door` instance variable from the `DoorState` class. You can drop this variable by changing the `click()`, `complete()`, and `timeout()` transition methods in the `DoorState` class to require a `Door` object as an input parameter. In this design, a `Door` object that receives, for example, a `click()` method from the carousel forwards this call as `state.click(this)`.

CHALLENGE 22.5

Complete the class diagram in Figure 22.4 to show a design that moves the door states to an interface.

A solution appears on page 416.

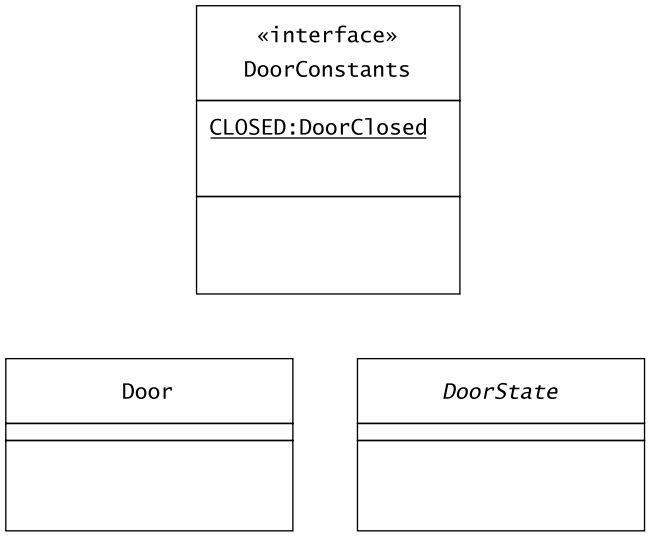

FIGURE 22.4: Complete this diagram to shows door states as constants defined by the `DoorConstants` interface.

Summary

Generally speaking, the state of an object depends on the collective value of the object's instance variables. In some cases, an object's state can be a prominent aspect of a class's behavior. This often occurs when an object is modeling a real-world entity whose state is important. In such a situation, complex logic that depends on the object's state may appear in many methods. You can simplify such code by moving state-specific behavior to a hierarchy of state objects. This lets each state class contain the behavior for one state in the domain. It also allows the state classes to correspond directly to states in a state machine.

To handle transitions between states, you can let states retain a context reference that contains a set of states. Alternatively, you can pass around, in state transition calls, the object whose state is changing. Regardless of how you manage state transitions, the STATE pattern simplifies your code by distributing an operation across a collection of classes that represent an object's various states.

APPENDIX B: SOLUTIONS

BRIDGE (Chapter 6)

SOLUTION 6.1 *(from page 67)*

Figure B.5 shows a completed sequence diagram that illustrates the normal steps in an application that uses a JDBC driver.

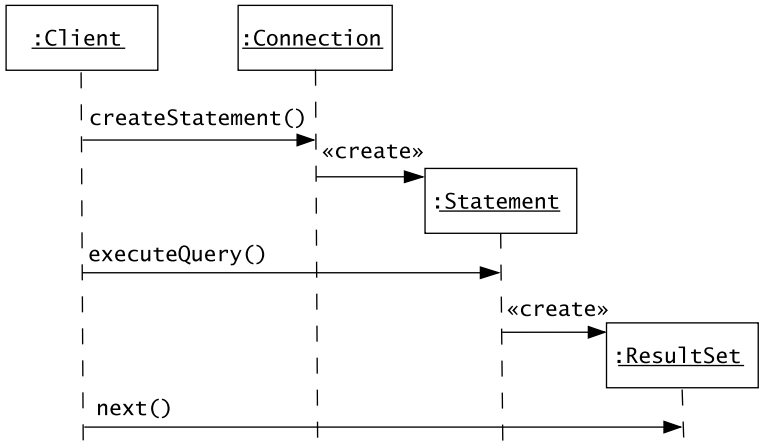

FIGURE B.5: A typical JDBC application creates a `Statement` object and uses it to execute one or more queries.

SOLUTION 6.2 *(from page 69)*

You have many choices about how to illustrate the class relationships in this example. Figure B.6 shows one answer. Your figure should show the relationship between `JDBCAdapter` and `Statement`. In UML, the open arrowhead on the association between `JDBCAdapter` and `Statement` indicates that a `JDBCAdapter` object can use or navigate to a `Statement` object, but a `Statement` object cannot use a `JDBCAdapter` object.

Note that you cannot show the class that implements `Statement`, as the JDBC architecture isolates you from knowing how a driver writer has named this class.

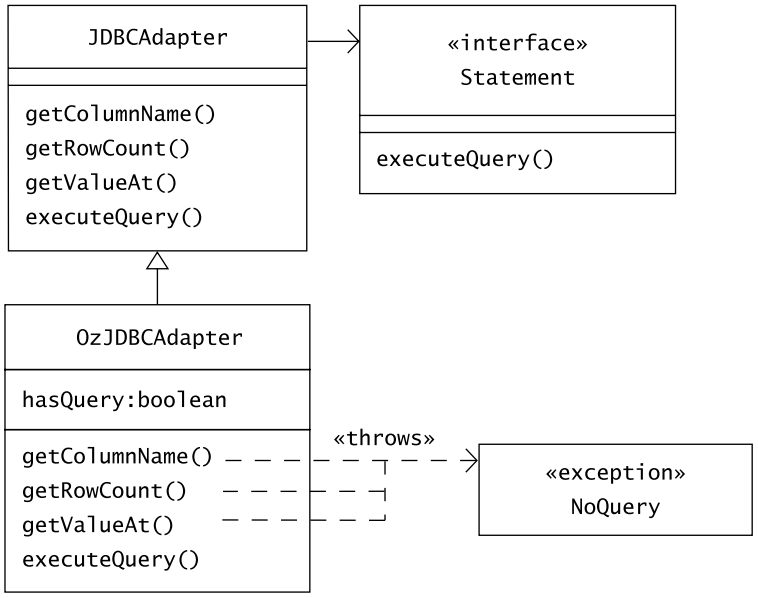

FIGURE B.6: An `OzJDBCAdapter` object will throw a `NoQuery` exception if the adapter is asked for data before a query has made data available.

An illustration of the `OzJDBCAdapter` class should show the methods it overrides. Figure B.6 also indicates that `OzJDBCAdapter` objects use a `hasQuery` Boolean to keep track of whether they have received a query.

SOLUTION 6.3 *(from page 72)*

Figure B.7 shows a solution.

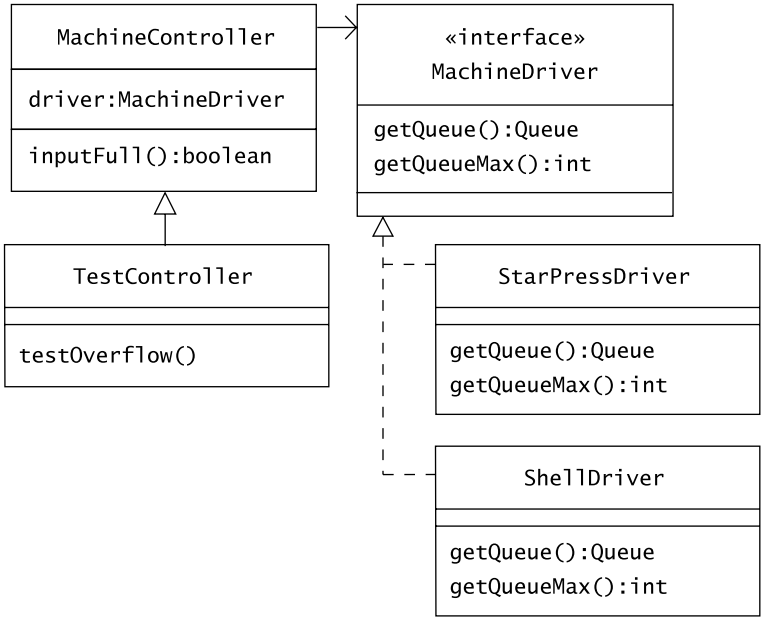

FIGURE B.7: The abstraction—a hierarchy of types of controllers—is separated from implementations of the abstract `driver` object that the abstraction uses.

SOLUTION 6.4 *(from page 74)*

Figure B.8 shows one solution.

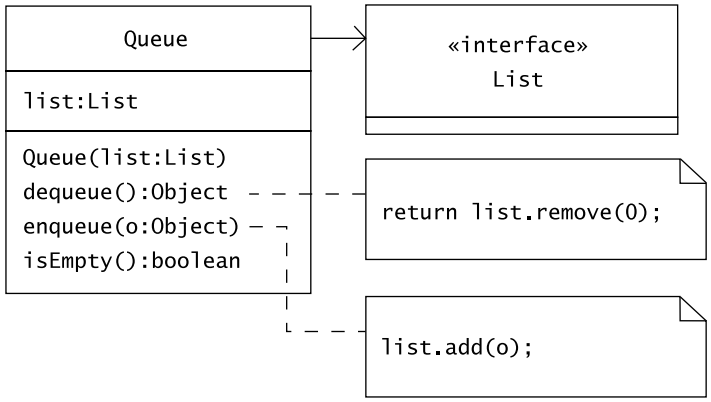

FIGURE B.8: The line forms at the rear: Objects enqueue at the tail of the list; dequeued objects come from the "head" of the list, at index 0.

STATE (Chapter 22)

SOLUTION 22.1 *(from page 226)*

As the state machine shows, when the door is open, a click will take the door to the `StayOpen` state, and a second click will start the door closing.

SOLUTION 22.2 *(from page 229)*

Your code should look something like:

```
public void complete()
{
    if (state == OPENING)
    {
        setState(OPEN);
    }
    else if (state == CLOSING)
    {
        setState(CLOSED);
    }
}

public void timeout()
{
    setState(CLOSING);
}
```

SOLUTION 22.3 *(from page 232)*

The code for `status()` relies on the state's class name and will report, for example, the status of an open door as "`DoorOpen`", instead of "`Open`". You can, of course, trim off the "`Door`" prefix, if you like.

SOLUTION 22.4 *(from page 234)*

Your code should look something like:

```
package com.oozinoz.carousel;
public class DoorClosing extends DoorState
{
    public DoorClosing(Door_2 door)
    {
```

```
        super(door);
    }

    public void click()
    {
        door.setState(door.OPENING);
    }

    public void complete()
    {
        door.setState(door.CLOSED);
    }
}
```

SOLUTION 22.5 *(from page 234)*

Figure B.25 shows a reasonable diagram. The code for this refactoring is in the `com.oozinoz.carousel_sol` package.

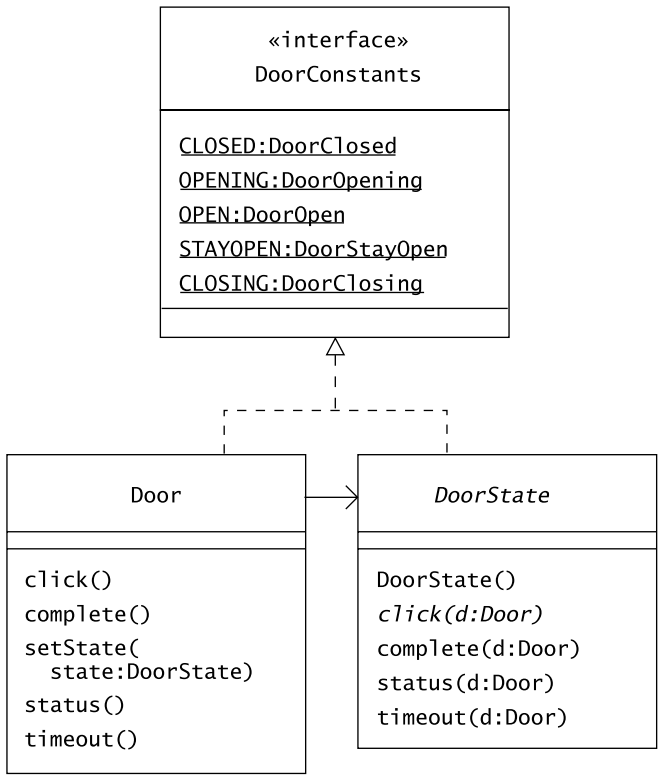

FIGURE B.25: This diagram shows the `Door` and `DoorState` classes sharing constants that `DoorConstants` defines.

The code for DoorConstants is:

```
package com.oozinoz.carousel_sol;
public interface DoorConstants
{
    DoorState CLOSED   = new DoorClosed();
    DoorState OPENING  = new DoorOpening();
    DoorState OPEN     = new DoorOpen();
    DoorState CLOSING  = new DoorClosing();
    DoorState STAYOPEN = new DoorStayOpen();
}
```

The Door class now retains a single state object:

```
package com.oozinoz.carousel_sol;
import java.util.*;
public class Door extends Observable
implements DoorConstants
{
    private DoorState state = CLOSED;

    //....
}
```

The Door object passes itself as an argument when it forwards state transition calls to its state variable. This lets the receiving state update the Door object's state with setState(). For example, Door.click() is:

```
public void click()
{
    state.click(this);
}
```

And DoorClosing.click(door:Door) is:

```
public void click(Door door)
{
    door.setState(OPENING);
}
```