1

WHY LEARN DESIGN PATTERNS AND WHY DO SO WITH HELP FROM HARRY POTTER?

Design, in general, involves complex mental processes, most of them poorly understood. However, it is commonly believed that all design entails using previously learned patterns at some level of detail. One can certainly assume that the designers who are celebrated for their work use patterns at a fairly low level of detail. A musical genius like Mozart probably used any previously learned patterns for only the most basic of the sound effects he wanted to create. On the other hand, it would be safe to say that the lesser composers of his time (or, for that matter, of any time) have probably borrowed significantly from the harmonies and the rhythms created by the geniuses.

Whereas the need for the content to be original in the artistic and the literary domains necessitates that the use of existing patterns be kept to a minimum in a new design, exactly the opposite is true for the case of software. In software design, although the need to be original and creative is important in dealing with hitherto unseen problems, of greater importance are the correctness and the robustness of the software produced.

If a new problem in software design is similar to one seen previously (and for which a correct software solution is already known to exist), the previously developed solution is preferred over what would otherwise be a new and creative way of solving the new problem. Using a previously known trusted solution to the problem can only increase the confidence that others place in your software.

And, should it happen that you are dealing with a large complex problem that does not lend itself to any single previously known solution strategy, you'd be expected to be creative in decomposing the problem into subproblems that can be solved with previously known trusted solutions.

In general, when you decompose a large problem, the subproblems that result are likely to be of varying levels of difficulty and detail. It is even possible that your overall decomposition will be hierarchical in which the smallest of the problems can be solved by using the well-known programming idioms in the language you are using for your software development. And, yes, those programming idioms can also be called patterns. However, we will not concern ourselves with such low-level design issues in this book.

On the other hand, this book is about the trusted solutions for what may loosely be referred to as the mid-level problems you are likely to encounter in creating a software

Designing with Objects: Object-Oriented Design Patterns Explained with Stories from Harry Potter,
First Edition. Avinash C. Kak.
© 2015 John Wiley & Sons, Inc. Published 2015 by John Wiley & Sons, Inc.

solution for a large problem. In particular, we will focus on the mid-level problems that can be solved by the twenty-three patterns first proposed by four authors who are now affectionately referred to as the Gang of Four (GoF). The book in which these patterns first appeared is now commonly referred to as the "Bible" of the object-oriented (OO) design patterns. The next section is devoted to this book and its contents.

1.1 THE OO DESIGN PATTERNS "BIBLE" BY GoF

The patterns movement in the software community was started by the much celebrated book "*Design Patterns — Elements of Reusable Object-Oriented Software*" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [1]. Drawing from their collective experience with object-oriented programming, the authors succeeded in crystallizing out twenty-three design patterns that have become, as is now universally acknowledged, the building blocks of much modern object-oriented software. As mentioned in the previous section, these authors are known as the Gang of Four (GoF) and the book frequently referred to as the "OO Patterns Book by GoF."

What is amazing about the GoF book, and also what makes the book timeless, is not only the large variety of programming problems that can be solved by its twenty-three patterns, but also the fact that the authors had the foresight to recognize a host of basic issues in object-oriented design that are likely to endure for all time. To grasp the reality of the moment and to abstract from it new fundamental understandings that can serve us for a long time into the future is no small feat.

Central to most GoF patterns is the interplay between the following four tenets of good object-oriented programming: (1) Programming to the public interfaces declared at the roots of class hierarchies, as opposed to calling directly the methods defined in the implementations of those interfaces. (2) Choosing composition over inheritance if a purely inheritance-based implementation is likely to result in an unmanageable number of classes as you try to figure out the best way to create representations for all the different variants of a generic object. (3) Again choosing composition over inheritance when the flexibility made possible by the former in how the objects relate to one another is more important than the representational efficiency provided by the latter. (4) Exploiting function overriding for runtime adaptation of the behavior of a class to the implementations provided by its subclasses.

Here we will briefly review the intuitive underpinnings of the tenets listed above: When the users of a class hierarchy make sure that their own code calls only the public methods declared in the root interface of the hierarchy, folks whose business it is to provide and maintain the implementation code in the hierarchy acquire the freedom to change that code as long as the interface declarations remain unchanged.

Regarding the second tenet, even though inheritance is enshrined as a cornerstone of object-oriented programming, using it without thought may result in class hierarchies that are much too large. If you try to create a subclass for capturing every small variation from a generic class, you could end up with too many subclasses. Why not take care of the small variations through composition, that is, by endowing your generic class with additional instance variables for the extra degrees of freedom that would allow you to create a larger variety of instances from the class? Since the objects constructed from a class definition are "composed" of the values given to the instance variables — these values may themselves be class type objects — you can see why we use the word "composition" to describe this alternative to inheritance.

Favoring composition over inheritance, as in the third tenet, also makes sense in situations where there is a need to maintain programming flexibility with regard to how the different types of objects relate to one another. Although inheritance makes for efficient representational frameworks (since the common attributes declared in the general classes do not need to be repeated in the specialized classes), the resulting inter-object relationships once created become hardcoded in the code base.

The intuition behind using function overriding for runtime customization of the behavior of a class, as mentioned in the fourth tenet, is just as straightforward. When you put function overriding to use, the code you write for the methods — especially when methods call other supporting methods defined in the same class — becomes much more efficient from the standpoint of being able to represent a *range* of behaviors. The specific behavior you elicit at runtime can then be customized by overriding the supporting methods in the subclasses.

Getting back to the subject of the twenty-three patterns in the GoF book, those patterns were placed by GoF in the following three categories: (1) Creational, (2) Structural, and (3) Behavioral. And, for finer differentiation, each pattern was given a mnemonic name that captures the essence of what that pattern is about.

The Creational Patterns examine issues in designing classes and the methods to construct instances from those classes from the standpoint of a number of considerations that frequently arise in object-oriented programming. Consider, for example, one of the aforementioned tenets of good object-oriented programming: The users of object-oriented software should only program to the interfaces of the class hierarchies. Keeping in mind this tenet, here is an example of a question addressed by the Creational Patterns: Say we have two class hierarchies, one in which we model the domain knowledge and the other in which we have code that, using indirection, can spit out instances of the classes in the first hierarchy. Can we still have the clients of this software adhere to the above mentioned tenet of good object-oriented programming? The Creational Patterns also recognize that calling willy-nilly the constructor of a class can be a dangerous thing to do if the instances to be produced require special computational resources. Included in the lessons that these patterns teach us is how to design a class so that it gives us some control over the instances created from the class — control in the sense of how many of the instances are allowed to exist at the same time.

The Structural Patterns are also about designing classes and constructing instances from the classes, but now the representational issues related to class design are more complex. As a case in point, let's say that the basic representation for a problem domain as captured by a single class must be enriched with arbitrary combinations of certain embellishments. What is the best way to create an overall representation that would allow for efficient production of the instances while incorporating the embellishments in them? Another representational question important to the Structural Patterns is how to design a class hierarchy when the problem domain calls for instantiating large objects that are made up of smaller objects, with all the objects (including the large objects) being of the same fundamental type. The Structural Patterns are also concerned about how to write a new class that must simultaneously adapt itself to the behavior of an old class while providing new services to a client of the old class, about creating different usage views of a complex system of classes for different categories of users, and so on.

The Behavioral Patterns are about eliciting useful runtime behaviors from a set of classes working together. These behaviors are dynamic, in the sense that how a class (or a group of classes working together) responds to a runtime condition (which may be created by user input or by a change in the state of one of the objects) will, in general, depend on the states of all other objects relevant to the condition. A major consequence of this is that the flow of execution for how the classes interact at runtime cannot be predicted in advance — unlike what is the case with the Creational and the Structural Patterns. Dynamic effects exhibited by the Behavioral Patterns include synthesizing at runtime a large behavior from more elementary behaviors provided by support classes, regulating the interaction among a set of classes so that it conforms to a protocol selected at runtime, rolling back the state of an object to what it was at an earlier time, and so on.

1.2 BUT WHAT HAS HARRY POTTER GOT TO DO WITH OO DESIGN PATTERNS?

Although some of the twenty-three design patterns are straightforward and can be understood easily, several require multiple readings from the GoF book before the ideas sink in. And, even after multiple readings, it is not until you have yourself programmed a pattern that you can claim to have fully understood what the GoF authors have tried to convey. The following quote from the preface to the GoF book is perhaps the best indicator of the complexity of several of the patterns:

> "A word of warning and encouragement: Don't worry if you don't understand this book completely on the first reading. *We didn't understand it all on the first writing!*"

The italics are by the author of this book.[1]

It is the complexity of the more difficult patterns that has served as a motivation for what you will find in the rest this book: An attempt to demystify the patterns by explaining them through the stories in Harry Potter. Thanks to the story-telling genius of J. K. Rowling, the richness of how the various characters interact in Harry Potter can be put to use to explain even the most complex object interactions in the OO patterns. The rest of the chapters in this book do exactly that.

A reader who has not read the GoF book might ask as to what mode of explanation was used by the authors of that book for the original presentation of the patterns. Or even, what sorts of explanations have been used in the other books on OO design patterns that have been published since the GoF book. In a majority of the explanations that the reader will see in the existing literature, the authors have attempted to use "real-world" problems in object-oriented software development to both motivate the reader to learn the patterns and to demonstrate their inner workings. But, unfortunately, in practically all cases, the typical space constraints of a book prevent a full airing out of the "real-world" problems.

Therefore, what a reader actually sees for a pattern explanation in practically all other books are just skeletal versions of some real-world problems. For most readers, it is not so easy to relate to the pattern explanations based on those highly abbreviated accounts of the original real-world problems — unless they themselves happen to be working in those problem domains. Let's put it this way: Are we to really believe that someone who spends all his/her time writing code for financial applications would understand all of the nuances

[1]This self-effacing statement by the GoF was also a reflection of their personality that placed a higher value on learning and exploring than on hubris and hype. And that, in turn, brings to mind the quote "Knowledge begins with humility — with wonder and with appreciation for what you don't know. The more you learn, the more you see how much you want and need to learn," that was made by the Bryn Mawr College President Katharine McBride in 1964.

associated with a pattern that is explained with the help of a highly abbreviated version of a problem in text format conversion or in computer-aided design?

This disconnection that a reader experiences with a pattern explanation that is based on a brief made-up version of a real-world problem often becomes even more pronounced for a young student who must simultaneously straddle two worlds: the world in which he/she is still trying to come to terms with the fundamental notions of encapsulation, inheritance, concurrency, polymorphism, and so on, and the real world used for pattern explanations.

So if skeletal versions of the so-called real-world problems are not the best medium for explaining the patterns, how about constructing pattern explanations with generic-sounding names for the classes, for their attributes, and for their methods? For example, for class names, we could use letters like `A`, `B`, `C`, etc., and for method names we could use `fooA()`, `barA()`, `bazA()`, `footB()`, etc. Subsequently, we could describe through method calls what it is that an object constructed from one class does to an object constructed from another class and weave these interactions into a narrative that explains the pattern. The main problem with this approach is that as the number of classes and the number of interactions between the classes increases, it becomes difficult to remember the roles assigned to the different classes and the behaviors programmed into their methods.

One could, of course, give more meaningful names to the classes and their methods, names that are evocative of their purpose. But that frequently leads to either absurd names or absurdly long names. For example, suppose you write an explanation for a pattern that shows how you can adapt a new class to an old class. You could try to refer to the old class as `OldClass` and the new class as `NewClass`. Let's now say your explanation requires you to name multiple such old and new classes with different roles. As an attempt at using "meaningful" names, you could try `OldClassForRoleM`, `NewClassForRoleN`, etc., with the suffixes `RoleM` and `RoleN` replaced by the names of the actual roles of the classes. As you can see, as you increase the number of classes, your explanation will sound boring, clumsy, and dry.

Figures 1.1 and 1.2 are meant to convey the difference in the quality of the explanations that can be created from a class diagram that uses arid symbolic names like

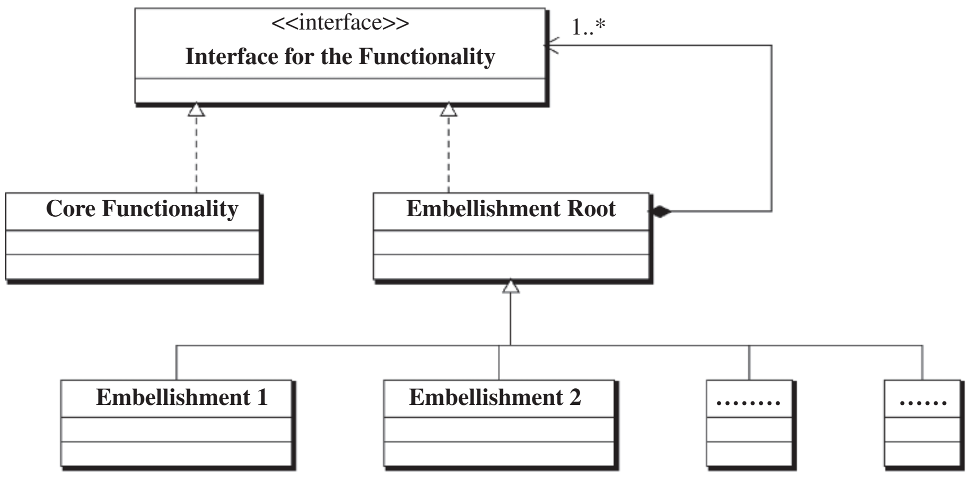

Fig. 1.1

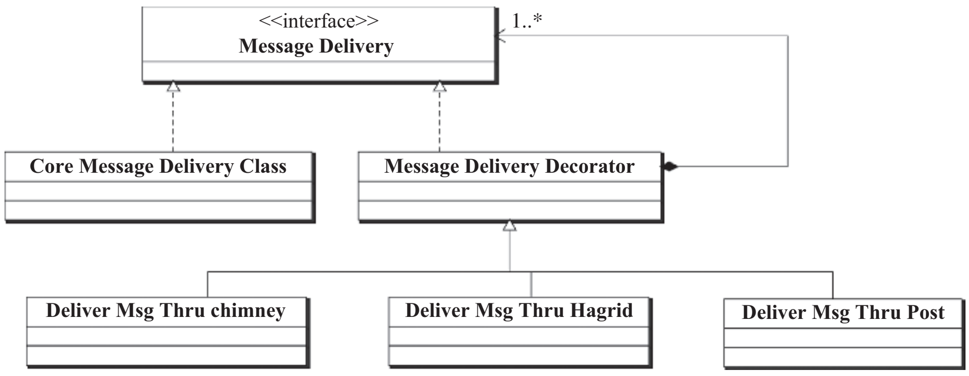

Fig. 1.2

`Core Functionality`, `Embellishment Root`, etc., and a class diagram that uses easier-to-relate-to names such as `Message Delivery`, `Deliver Msg Thru Chimney`, `Deliver Msg Thru Hagrid`, and so on. As the reader will see in Chapter 10, both of these class diagrams can be used to explain the notion of recursive nesting of class embellishments in the Decorator pattern. Figure 1.1 uses boring generic names, whereas Figure 1.2 uses names that are evocative of a hilarious snippet from the first Harry Potter book in which Mr. Dursley does everything in his power to keep the admission letter sent by Hogwarts from reaching Harry. This connection between Figure 1.2 and a most delightful episode related to Harry Potter makes the Decorator pattern both easier to understand and easier to remember. As a measure of Figure 1.2 possessing greater explanatory power over the more generic depiction in Figure 1.1, the former figure, with its richly evocative class names, along with just a couple of words about what the Decorator pattern is all about, is likely to cause a reader to immediately experience an aha moment with regard to understanding the core idea of the pattern.

1.3 IS FAMILIARITY WITH HARRY POTTER A REQUIREMENT FOR UNDERSTANDING THIS BOOK?

Although the book is written for a reader who is already familiar with Harry Potter, that does not imply that the explanations presented would be inaccessible to someone who has not read those books. Starting with the next chapter, every chapter includes a section titled "Harry Potter Story Used to Illustrate the XYZ Pattern" that, while intended primarily for a Harry Potter fan to recall the ideas used in the explanation of that pattern, should nevertheless be understandable to others as a "brief story" in its own right. If a reader who has no desire to read Harry Potter before launching into this book would be indulgent enough to accept this section as a standalone account that is either related to some magical objects or that is about an interaction between certain characters — although by no means an account written in the same compelling style as by J. K. Rowling — he/she should be able to use that account to understand the rest of the explanation of the pattern.

1.4 HOW THE PATTERN EXPLANATIONS ARE ORGANIZED

Each pattern is presented in a separate chapter, and each chapter consists of the following sections:

- An introduction to the key thought in the pattern.
- The intent and the applicability of the pattern.
- A general introduction to the pattern.
- A section that describes which part of the Harry Potter story was used for explaining the pattern. The title of this section is always "Harry Potter Story Used to Illustrate the XYZ Pattern" for the pattern XYZ.
- A section titled "A Top Level View of the Pattern Demonstration" that presents the overall class diagram and the related explanations for the pattern in question. The explanation of the pattern in this section is an extension of the narrative in the preceding "Harry Potter Story" section.
- The "Top Level View" section is followed by multiple sections, each devoted to explaining one class, although, and in some cases, a single section may present multiple classes that are closely related.
- Each chapter ends in a section named "Playing with the Code" that first describes how to compile and execute the Java code presented for the pattern and then talks about how a reader may extend the demonstration code in order to gain additional insights into the pattern.

1.5 THE TERMINOLOGY OF OBJECT-ORIENTED PROGRAMMING

It is possible that a casual reader skipping through the book — especially a reader with minimal prior exposure to object-oriented programming — would be bewildered by some of the terminology used for describing how objects are created and manipulated in software, how they interact with one another, how the clients interact with them, and so on. The terminology definitions shown below are meant for such a reader especially, the idea being that it might help the reader grasp at a high level, if not in detail, the explanations of the patterns in this book.[2]

The definitions shown below are specifically for the case of object-oriented programming in Java — since that is the language used in the code for demonstrating the patterns. Note, however, most of the definitions are language agnostic and apply to object-oriented programming in general.

The words that appear italicized in the definitions are also defined in the Terminology that follows.

abstract class: To add to the entry shown for *class*, a class is abstract if it is not possible to construct *instances* from that class. Abstract classes play a very important role in object-oriented programming by serving as interfaces for class hierarchies (although,

[2]The "definitions" shown have been excerpted from Chapter 3 of Programming with Objects [2].

more commonly, you'd use Java interfaces for that purpose if you are programming in Java) and as "mix-in" classes to lend specialized behaviors to other classes. They can also be of great help in building a *class hierarchy* incrementally. In Java, a class becomes abstract when it is explicitly declared to be so.

abstract method: To add to the entry for *method*, a method can be declared *abstract* if its header includes the keyword `abstract`. When a method is declared abstract, it does not come with any implementation code. You declare a method abstract in a class because you expect the subclasses of that class to provide the implementation code for the method. A class remains abstract as long it has any abstract methods.

access control modifiers: Each member of a class, whether it is a variable or a method, has associated with it an access control property that for Java is one of *private*, *public*, *protected*, and *package*.

The *public* members of a class can be directly accessed anywhere in the source code. Additionally, the public members of a class are inherited by the subclasses.

On the other hand, the *private* members of a class are accessible only within that class. Although the private members of a class are inherited by its subclasses, they cannot be directly accessed in the subclasses.

The access control modifier *protected* is used for a variable or a method in a class if we wish for that member of the class to be visible in only the subclasses of the class across all packages. A protected member acts like a public member within the same package.

When no access control modifier is mentioned for a class member in Java, that implies *package* access control for that member. Such members behave like public members within the same package but private with respect to the code in all other packages.

attribute: See the entry for *instance variable*.

base class: See the entry for *inheritance*.

child class: See the entry for *inheritance*.

class: At a high level of conceptualization, a *class* can be thought of as a category or a type. We may think of "User" as a class and a specific user would then be an *instance* of (or an *object* constructed from) this class. The following constitutes a definition of the class `User` in Java:

```
class User {
    private String name;
    private int age;
}
```

class hierarchy: See the entry for *inheritance*.

class method: See the entry for *method*.

class variable: First read the entry for *instance variable*. In addition to *instance variables*, we may endow a class with one or more *class variables*. Whereas an instance variable is given values on a per-instance basis, the value of a class variable is global with respect to all instances constructed from a class. Class variables are frequently referred to as *static variables*. In Java, a class variable is declared by using the keyword `static`, as in the class definition below that declares a class with two instance variables and one class variable:

```
class SavingsAccount {
    private String name;
    private int age;
    public static double interestRate;
    // ....
}
```

concrete class: A class that is not *abstract* is informally referred to as a *concrete class*.

constructor: A class is generally provided with a *constructor* whose job is to create instances from the class. A constructor sets aside the memory needed for the instance and in that memory sets the values of the instance variables according to the arguments supplied to the constructor (or, in the absence of arguments, by the default values if those are known). Shown below is another definition of the `User` class. What you see in the lines labeled "(A)" through "(D)" is the constructor for the class.

```
class User {
    private String name;
    private int age;

    public User(String str, int yy) {                   //(A)
        name = str;                                     //(B)
        age = yy;                                       //(C)
    }                                                   //(D)
}
```

In general, a class is allowed to have any number of constructors. When a class has multiple constructors, they differ with respect to the number of parameters and the parameter types. The compiler uses *overload resolution* to figure out which constructor to invoke for a given constructor call.

derived class: See the entry for *inheritance*.

extended class: See the entry for *inheritance*.

final: See the entries for *inheritance*, *instance variable*, and *overriding*.

implements: First read the entry for *interface*. A class that provides implementation code for the methods declared in an interface is said to *implement* that interface.

inheritance: Earlier we defined *class* as a category. A subcategory of a more general category can also be defined as a class — in the form of a *subclass* of the more general class. The subclass *inherits* some or all of the attributes and the methods defined for the class that corresponds to the more general category.

Since, in general, a class may be extended into multiple *subclasses*, each such subclass further extended into even more specialized *subclasses*, and so on, we can end up with what is known as a *class hierarchy*.

A subclass is also commonly referred to as a *derived class*, an *extended class*, or a *child class*. And the more general class is commonly referred to as the *base class* or the *superclass*.

The syntax used in Java for defining a subclass is illustrated by the following example:

```
class User {
    private String name;
    private int age;
    public User(String str, int yy) { name = str; age = yy; }
}

class StudentUser extends User {                         //(A)
    private String schoolEnrolled;
    public StudentUser( String nam, int y, String sch ) {
        super(nam, y);                                   //(B)
        schoolEnrolled = sch;
    }
}
```

where `StudentUser` is a subclass of `User`. Note the Java keyword `extends` in line (A). Also note how the *constructor* definition for the subclass calls on the constructor of the *base class* in line (B) through the keyword `super` for the construction of the base-class slice of the subclass object. With class–subclass definitions as shown above, an instance of type `StudentUser` can act like an instance of type `User`. That is, we can construct an instance of type `StudentUser` and assign it to a variable of type `User`. The fact that a subclass-type object can act like a superclass type is referred to as *polymorphism*. Also note that a class declared *final* cannot have subclasses.

instance: Given a class, you may construct a specific *instance* of that class by calling its *constructor*. An *instance* created by a constructor is also frequently referred to as an *object*.

instance method: See the entry for *method*.

instance variable: In the definition of the `User` class in the entries for *class*, *constructor*, and *inheritance*, we refer to `name` and `age` as the *instance variables* of the class. In general, we expect the values of such variables to vary from one instance to another. Again, in general, the value assigned to an *instance variable* can be changed, unless the variable is declared to be *final*, in which case it can only be assigned to once. An *instance variable* may also be given a default value in the class definition.

An *instance variable* is also referred to as *data member*[3] and *attribute*. The UML notation that is described briefly in the next section specifically uses *attribute* when referring to *instance variables*.

instantiation: The word *instantiation* means constructing an *instance* of a class.

interface: A Java *interface* is an *abstract class* that, in its most common usage, declares a set of methods by only mentioning their *signatures*. A Java *interface* must not include any implementation code. Here is an example of an interface from the Java Collections Framework:

```
interface Collection {
    public boolean add( Object a );
    public boolean remove( Object a );
    // other methods
}
```

[3]It's the latter usage, meaning *data member*, that you will find in [2] when referring to the instance variables of a class.

Interfaces in Java are also used for grouping together related constants. When a class inherits from such an interface, the constants appear as if locally defined in the class. Such constants are treated implicitly as `static` and `final`.

method: A class is commonly endowed with behaviors through functions that are normally invoked on the instances constructed from the class. Such functions are most commonly referred to as methods. The definition of `print()` in the `User` class shown below is an example:

```
class User {
    private String firstname;
    private String lastname;
    public User(String str, int yy) { name = str; age = yy; }
    public void print() {
        System.out.print( firstname + " " + lastname );
    }
}
```

Note that when an *instance* is created, the methods in a class are bound to the instance at run time. That is referred to as "dynamic binding" — unless the method is declared to be *static*, in which case it is bound at compile time to the class itself. It's for that reason that a method such as `print()` shown above is also referred to as an *instance method*. A *static method*, on the other hand, is also referred to as a *class method*.

object: First see the entry for *instance*. The term *object* with the lowercase 'o' is not to be confused with *Object* with the uppercase 'O' in Java. Whereas the former can refer to any instance created by calling the constructor of a class, the latter is the name of the superclass of all classes in Java. Therefore, all objects in Java are of type *Object*. Every Java class, user defined or otherwise, inherits implicitly from the root class *Object.*

overloading: Overloading a method name or a constructor name means being able to use the same name with a different number and/or types of parameters. When a class has overloaded constructor or method names, it is the compiler's job to determine which of the constructors or the methods to invoke for a given call. The compiler does this by using what is known as the *overload resolution algorithm*.

overload resolution: See the entry for *overloading*.

overriding: A *subclass* can provide an override implementation for a method defined in the base class. Subsequently, if this method is invoked on a variable whose type is that of the base class but that is actually pointing to an object of type subclass, it is the subclass definition for the method that will be used. A *method* that is declared to be *final* in a base class cannot be overridden in a subclass of the base class.

package: See the entry for *access control modifiers*. *Package* may also refer to a "Java package" that groups together related classes and defines a namespace for them. For example, the basic classes of Java that are made available automatically in a user program are in the `java.lang` package. In general, if you need to use a class from a package, either you must import the package in your program or you must use the fully qualified name for the class.

polymorphism: See the entry for *inheritance*.

private: See the entry for *access control modifiers*.

protected: See the entry for *access control modifiers*.

public: See the entry for *access control modifiers*.

signature: By the *signature* of a method we mean the return type, followed by the name of the method, followed by a list of the data types in its parameter list.[4]

static: See the entries for *class variable* and *class method*.

static method: See the entry for *method*.

subclass: See the entry for *inheritance*.

superclass: See the entry for *inheritance*.

The object-oriented programming terminology listed above is by no means complete. We have only defined the terms that occur most frequently in the rest of the book. Obviously, only a book devoted to just OO, such as Programming with Objects [2], can do full justice to all of the terminology of object-oriented programming.

1.6 THE UML NOTATION USED IN THE CLASS DIAGRAMS

The diagrams used in this book for explaining the patterns are mostly class diagrams that are based on the UML conventions for constructing such diagrams.[5]

In UML, a class is represented by a rectangular box that in its most detailed representation is divided into three parts vertically. The name of the class is written in the uppermost partition of the box, followed by its instance and static variables (referred to as *attributes* in UML) in the middle partition, followed by its methods (called *operations* in UML) in the lowest partition. The name of the class is shown in bold for a concrete class and in italics for an abstract class. Figure 1.3 shows an example of this representation for a class named `Employee`.

In its most common usage, a class diagram shows two relationships between different classes: *generalization* and *association*. A superclass is considered to be a generalization of its subclasses. By the same token, a subclass is considered to be a specialization of its superclass. For example, the class diagram of Figure 1.4 shows with an arrowed solid line the class `Employee` as a superclass, and therefore a generalization, of the class `Manager`. Note that the generalization arrow, with a closed triangle arrowhead, points to the superclass. In Figure 1.4, the class `Manager` would be considered to be a specialization of the class `Employee`. An association, on the other hand, is depicted with a solid line between two

[4]In some languages, the return type may not be considered to be a part of the signature. Another word that is used for the signature of a method is *header*. The header of a method always includes its return type.

[5]UML stands for Unified Modeling Language. It was promulgated by the Object Management Group (OMG), a not-for-profit organization founded by the leading software corporations of the world. UML is a visual language that allows one to create different types of graphical representations of software. These visual representations, in the form of various types of diagrams, greatly facilitate the conceptualization and dissemination of object-oriented designs, especially because the diagrams only need to be drawn with the level of detail that is required for explaining how the classes are meant to be used. The main repository of all information related to UML can be found at [3]. If all that a reader wants is a level of familiarity with UML that would be sufficient to understand the diagrams in this book, he/she is only going to need the information presented in Chapter 14 of [2]. The most relevant sections from that source are reproduced here for the convenience of the reader.

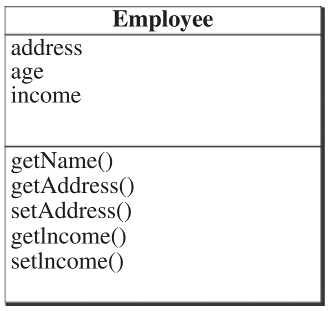

Fig. 1.3

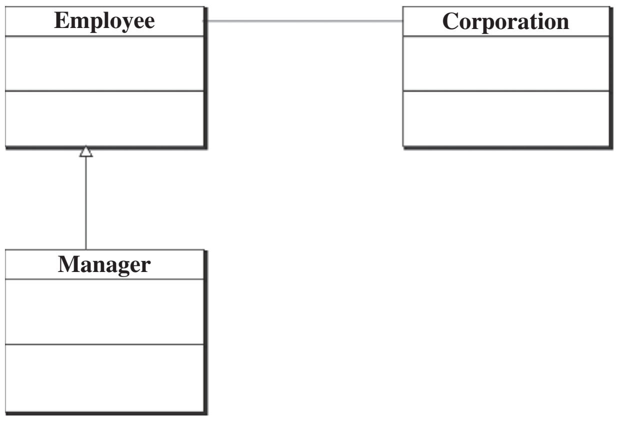

Fig. 1.4

classes, as between `Employee` and `Corporation` in the figure. You show an association link between two classes if the objects constructed from one class use in some capacity — say as instance variables — the objects constructed from the other class.

Other types of relationships between classes that can be depicted in a class diagram are *aggregation* and *composition*. The next two subsections discuss in greater detail the depiction of associations, aggregations, and generalizations in class diagrams.

We placed only the names of the classes in the boxes in the class diagram of Figure 1.4. In general, how much detail one shows for a class depends on the perspective used in drawing the diagram. A class diagram may be drawn using three different perspectives: (i) *conceptual*, (ii) *specification*, and (iii) *implementation*.

At the conceptual level, for each class you include only the bare minimum information needed to convey an overall sense of the main concepts of a problem domain. This is the diagram you are likely to draw when you are just getting started with the design of an OO program. However, even after you have fully developed an OO system, a conceptual level diagram can be useful for communicating to others a coarse-level description of the system. At the specification level, you want to show the interfaces of each class. At this level you'd want to make explicit the class responsibilities, as embodied in the public operations for each class. At the implementation level, you want to show more precisely how a class was (or needs to be) implemented in code. Now you'd include the private and the protected attributes and operations as well.

In the OO literature, one also commonly sees mention of IsA and HasA relationships between classes. The former represents a generalization-specialization relationship and the latter an association, an aggregation, or a composition. The name IsA is supposed to capture relationships such as

```
A Manager IsAn Employee
A CorporateCustomer IsA Customer
```

In such statements, what comes after IsA is a generalization or a super-type of what comes before. On the other hand, statements like

```
An Order HasA Customer
An Orchestra HasA Player
A Window HasA Slider
```

express containment, in the form of an association, an aggregation, or a composition.

1.6.1 Association as a Relationship Between Classes

The class diagram of Figure 1.4 showed an association to display the conceptual link between an object of type `Employee` and an object of type `Corporation`. An example of a more elaborate representation of such an association is shown in Figure 1.5.

In the example depicted, an `Employee` has an instance variable called `employedBy` of type `Corporation`; this instance variable is shown as a label at the head of the arrowed association link from `Employee` to `Corporation`. We can talk about the label `employedBy` as the *role* played by a `Corporation` in an instance of type `Employee`. The arrowhead on the association link from `Employee` to `Corporation` is referred to as the *navigability* arrow. The arrow tells us which of the two objects implements the association. In the example shown, the association with the rolename `employedBy` is implemented in the `Employee` class and therefore "belongs" to instances of type `Employee`. The label '`0..1`' at the `Corporation` end of the association is referred to as the *multiplicity* of the association, which specifies *how many* instances of type `Corporation` in the role `employedBy` may associate with a single instance of type `Employee`. The multiplicity of '`0..1`' means that an `Employee` is employed by no more than one `Corporation`.

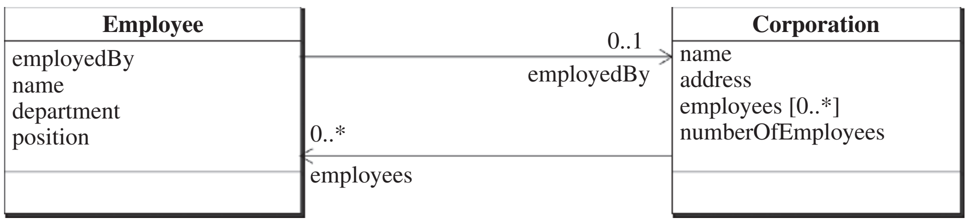

Fig. 1.5

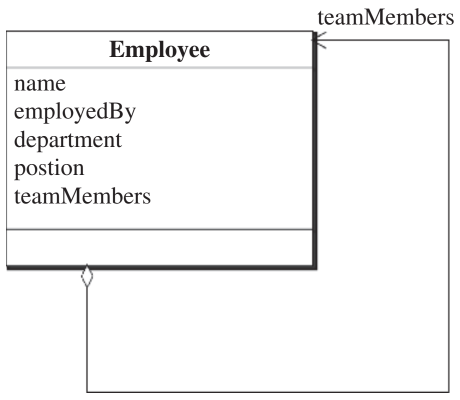

Fig. 1.6

About the association link that goes from `Corporation` to `Employee` in Figure 1.5, the navigability arrow points toward the latter, and the rolename label is `employees` with the multiplicity symbol '`0..*`'. The multiplicity of '`0..*`' means that any number of employees, including zero, is allowed in an instance constructed from `Corporation`. If there was a legal requirement that a corporation possess at least one employee, with no constraints on the upper limit, the multiplicity label associated with the rolename `employees` would change to '`1..*`'. As you might have guessed already, the symbol '`*`' in a multiplicity label means *an indefinite number*.

The two association links in Figure 1.5 could also be shown as a single line between the two classes. If we were to do so for our example, the line would show navigability arrows, rolenames, and multiplicity symbols at both ends. An association with no navigability arrows is considered bidirectional.

Figure 1.5 is an example of a binary association between two different classes. A binary association is also allowed to connect a class to itself, in which case the association is called *reflexive*. Figure 1.6 shows an example of a reflexive association. The next subsection talks about the role of the diamond that you see at the base of the association link in Figure 1.6.

1.6.2 Aggregation and Composition as Relationships Between Classes

The objects connected through an association may or may not exist independently of each other, and it is useful to differentiate between the two cases in a class diagram, especially when an association is a link between a "whole" and its "parts."

When there exist lifetime dependencies between the whole and its parts — in the sense that the parts exist solely for the benefit of the whole — we refer to the relationship between the whole and the parts as a *composition*. That is, we consider the whole to be a composite of its parts. Such an association is depicted with a filled diamond at its base. Even though such a relation can also be depicted as a straightforward association with appropriate navigability arrows and multiplicities, showing it as composition draws attention to the lifetime dependencies between the composite and its parts.

Consider the example in Figure 1.7 where we have used filled diamonds to show a `Window` composite. Obviously, the "parts" that form the composite, such as sliders, scrollbars, and so on, will cease to exist when a `Window` is closed.

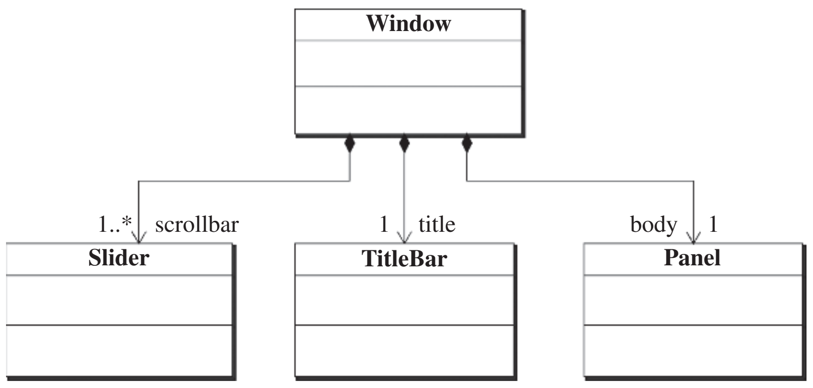

Fig. 1.7

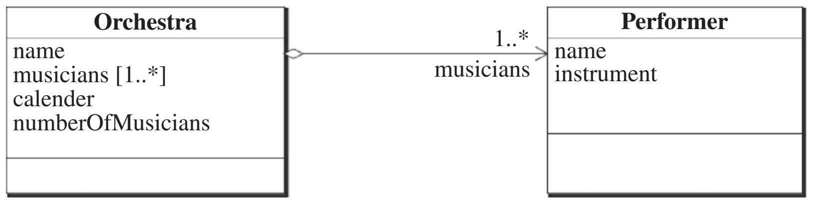

Fig. 1.8

When you have a whole–parts relationship in which the parts can exist independently of the whole, you have an *aggregation*. In the aggregation depicted in Figure 1.8, the performers would continue to exist even after the `Orchestra` object has ceased to do so. An aggregation is depicted with a hollow diamond at one end of the association link, the end that is an aggregate of the objects at the other. Although this type of a relationship could also be displayed as a straightforward association with appropriate navigability arrows and multiplicities, the concept of an aggregation is supposed to capture the fact that even though an orchestra is the sum total of its performers, the performers would continue to exist even if the orchestra ceased to do so.

1.6.3 Representing Attributes

As mentioned earlier, the attributes of a class — which could either be instance variables or static variables — are shown in the middle partition of the box that represents a class. The UML convention for displaying an attribute is as follows:

```
visibility  name [N] : type =  initialValue {property-string}
            ---------------
```

where the visibility is one of

```
+ for  public visibility

#  for protected visibility

-  for private visibility
```

although the keywords *public, protected*, and *private* can also be used directly. The absence of a visibility marker indicates only that the visibility is not shown (not that it is undefined or public) because, say, its depiction is not important to use intended for the class diagram.

In the UML notation for displaying the attributes, the name of the attribute goes where you see the string `name`. The symbol `N` inside square brackets denotes the multiplicity allowed for the attribute. The convention for expressing multiplicity is the same as for an association. For example, if an attribute is allowed to take one or more values, the multiplicity symbol `N` would be replaced by '`1..*`'. The absence of multiplicity designation means that exactly one value is allowed for the attribute.

A language-dependent specification of the implementation type of the attribute goes where you see the string `type`. The string `initialValue` is a language-dependent expression for the default value of the attribute for a newly created instance of the class, and `property-string` is a string for expressing those traits of the attribute that are not captured by the rest of the syntax. For example, for an attribute that is read-only (such as an attribute that is declared to be `final` in Java), the `property-string` would be set to `frozen`.

The underscore, shown under `name` and `type`, if used, signifies that the attribute has class scope, which means the same thing that it is static or one per class, as opposed to one per instance. Except for the `name`, all other elements of the syntax specification are optional.

1.6.4 Representing Operations

The third partition from the top, when it exists, of a class box shows its *operations*, meaning the methods defined for the class. When a class is drawn at the specification level, only the public operations of the class are displayed. However, at the implementation level, you'd also want to show the private and the protected operations. The full UML syntax for an operation is

```
visibility name (parameter-list) : return-type {property-string}
           -----------------------------------
```

where *visibility* and *name* mean the same as for the case of attributes. The *parameter-list* is a comma-separated list of the formal parameters for the operation, each specified using the syntax

```
kind name : type = defaultValue
```

where `kind` can be *in, out*, or *inout*, where *in* is for a parameter that passes a value to the operation, *out* is for a parameter that fetches a value from the operation, and *inout* for a parameter that can play both roles. The symbols `name`, `type`, and `defaultValue` serve their usual roles.

Regarding the syntax for an operation, the symbol `return-type` is an implementation-dependent language type of the value returned by the operation. The `property-string` can be used to express such traits as whether an operation is abstract, which is the case when only the header of the method is a part of the class definition and no implementation code is provided. Finally, operations that have class scope — meaning that they are static — are underlined as shown.

It is useful to make a distinction between two types of operations: *query* and *modifier*. A query operation simply tries to get the value of some class attribute without changing the state of the object. On the other hand, a modifier operation will change the state of the object. Informally, these two types of operations are also referred to as the *getter* and the *setter* methods of a class, respectively.