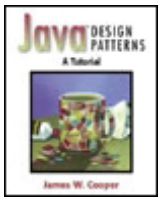


Java™ Design Patterns: A Tutorial

By James W. Cooper

Publisher : Addison Wesley
Pub Date : January 28, 2000
ISBN : 0-201-48539-7
Pages : 352

Design patterns have become a staple of object-oriented design and programming by providing elegant, easy-to-reuse, and maintainable solutions to commonly encountered programming challenges. However, many busy Java programmers have yet to learn about design patterns and incorporate this powerful technology into their work.

Java(TM)Design Patterns is exactly the tutorial resource you need. Gentle and clearly written, it helps you understand the nature and purpose of design patterns. It also serves as a practical guide to using design patterns to create sophisticated, robust Java programs.

This book presents the 23 patterns cataloged in the flagship book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides. In *Java(TM)Design Patterns,* each of these patterns is illustrated by at least one complete visual Java program. This practical approach makes design pattern concepts more concrete and easier to grasp, brings Java programmers up to speed quickly, and enables you to take practical advantage of the power of design patterns.

Key features include:

- Introductory overviews of design patterns, the Java Foundation Classes (JFC), and the Unified Modeling Language (UML)
- Screen shots of each of the programs
- UML diagrams illustrating interactions between the classes, along with the original JVISION diagram files
- An explanation of the Java Foundation Classes that illustrates numerous design patterns
- Case studies demonstrating the usefulness of design patterns in solving Java programming problems
- A CD containing all of the examples in the book, so you can run, edit, and modify the complete working programs

After reading this tutorial, you will be comfortable with the basics of design patterns and will be able to start using them effectively in your day-to-day Java programming work.

Table of Content

Copyright

Library of Congress Cataloging-in-Publication Data

Cooper, James William, 1943–

Java design patterns : a tutorial / James W. Cooper.

p. cm.

Includes bibliographical references and index.

1. Java (Computer program language) I. Title

QA76.73.J38 C658 2000

005.l3'3—dc21

99-056547

4 5 6 7 8 9 MA 03 02 01

4th Printing January 2001

Credits

Acquisitions Editor: Paul Becker

Editorial Assistant: Ross Venables

Production Coordinator: Jacquelyn Doucette

Compositor: Stratford Publishing Services

Preface

This is a practical book that tells you how to write Java programs using some of the most common *design patterns.* It is structured as a series of short chapters, each describing a design pattern and giving one or more complete, working, visual example programs that use that pattern. Each chapter also includes Unified Modeling Language (UML) diagrams illustrating how the classes interact.

This book is not a "companion" book to the well-known *Design Patterns* text [Gamma, 1995] by the "Gang of Four." Rather, it is a tutorial for people who want to learn what design patterns are about and how to use them in their work. You need not have read *Design Patterns* to gain from reading this book, but when you are done here you might want to read or reread that book to gain additional insights.

In this book, you will learn that design patterns are a common way to organize objects in your programs to make those programs easier to write and modify. You'll also see that by familiarizing yourself with these design patterns, you will gain a valuable vocabulary for discussing how your programs are constructed.

People come to appreciate design patterns in different ways—from the highly theoretical to the intensely practical—and when they finally see the greatpower of these patterns, they experience an "Aha!" moment. Usually this moment means that you suddenly had an internal picture of how that pattern can help you in your work.

In this book, we try to help you form that conceptual idea, or *gestalt,* by describing the pattern in as many ways as possible. The book is organized into six main sections:

- An introductory description
- A description of patterns grouped into three sections: Creational, Structural, and Behavioral
- A description of the Java Foundation Classes (JFC) showing the patterns they illustrate
- A set of case studies where patterns have been helpful

For each pattern, we start with a brief verbal description and then build simple example programs. Each example is a visual program that you can run and examine so as to make the pattern as concrete as possible. All of the example programs and their variations are on the CD-ROM that

accompanies this book. In that way, you can run them, change them, and see how the variations that you create work.

All of the programs are based on Java 1.2, and most use the JFC. If you haven't taken the time to learn how to use these classes, there is a tutorial covering the basics in Section 5 where we also discuss some of the patterns that they illustrate.

Since each of the examples consists of a number of Java files for each of the classes we use in that example, we also provide a Visual SlickEdit project file for each example and place each example in a separate subdirectory to prevent any confusion.

As you leaf through the book, you'll see screen shots of the programs we developed to illustrate the design patterns; these provide yet another way to reinforce your learning of these patterns. You'll also see UML diagrams of these programs that illustrate the interactions between classes in yet another way. UML diagrams are just simple box and arrow illustrations of classes and their inheritance structure, with the arrows pointing to parent classes and dotted arrows pointing to interfaces. If you are unfamiliar with UML, we provide a simple introduction in the first chapter.

Finally, since we used JVISION to create the UML diagrams in each chapter, we provide the original JVISION diagram files for each pattern as well, so you can use the demo version of JVISION included on the CD to play with them. We also include the free Linux version of JVISION.

When you finish this book, you'll be comfortable with the basics of design patterns and will be able to start using them in your day to day Java programming work.

James W. Cooper
Wilton, CT
Nantucket, MA
November, 1999

About the Author

James W. Cooper is a research staff member in the Advanced Information Retrieval and Analysis Department at the IBM Thomas J. Watson Research Center. He is also a columnist for *Java Pro* magazine and a reviewer for *Visual Basic Programmer's Journal*. His previous books include *Principles of Object-Oriented Programming Using Java 1.1* (Ventana) and *The Visual Basic Programmer's Guide to Java* (Ventana).

Acknowledgments

Writing a book on Java design patterns has been a fascinating challenge. Design patterns are intellectually recursive; that is, every time that you think that you've wrapped up a good explanation of one, another turn of the crank occurs to you or is suggested by someone else. So, while writing is essentially a solitary task, I had a lot of help and support.

Foremost, I thank Roy Byrd and Alan Marwick of IBM Research for encouraging me to tackle this book and providing lots of support during the manuscript's genesis and revision. I also

especially thank Nicole Cooper for editing my first draft; she definitely improved its clarity and accuracy.

The design pattern community (informally called the "Pattern-nostra") were also a great help. In particular, I thank both John Vlissides and Ken Arnold for their careful and thoughtful reading of the manuscript. Among the many others, I thank Ralph Johnson, Sherman Alpert, Zunaid Kazi, Colin Harrison, and Hank Stuck. I'm also grateful to John Dorsey and Tyler Sperry at *JavaPro* magazine for their encouragement and editorial suggestions on some of the columns that I wrote that later became parts of this book. Thanks also to Herb Chong and Mary Neff for lending their names and part of their project descriptions to the case studies chapter. Finally, thanks to my wife Vicki, who provided endless support during the ups and downs of endless writing and seemingly endless revision.

Section 1: What Are Design Patterns?

In this first section of the book, we outline what design patterns actually are and give a few simple examples. You'll see that much of the discussion about patterns is really a discussion about various classes and how they communicate.

Then we show you how to use UML diagrams to represent the relationships between classes in your programs and how they can quickly reveal the patterns in these programs.

Chapter 1. Introduction

Sitting at your desk in front of your workstation, you stare into space, trying to figure out how to write a new program feature. You know intuitively what must be done and what data and which objects come into play, but you have this underlying feeling that there is a more elegant and general way to write this program.

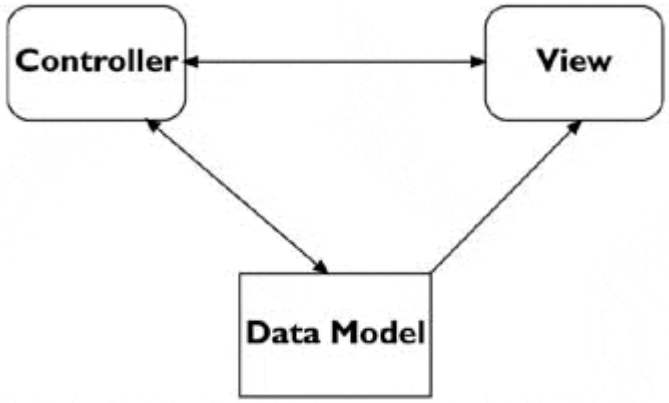
In fact, you probably don't write any code until you can build a picture in your mind of what code does and how the pieces of the code interact. The more that you can picture this "organic whole" or *gestalt*, the more likely you are to feel comfortable that you have developed the best solution to the problem. If you don't grasp this whole right away, you might stare out of the window for a long time, even though the basic solution to the problem is quite obvious.

In one sense, you feel that the more elegant solution will be more reusable and more maintainable. However, even if you are likely to be the only programmer, you feel reassured once you have designed a solution that is relatively clean and doesn't expose too many internal inelegancies.

One of the main reasons that computer science researchers began to recognize design patterns was to satisfy this need for good, simple, and reusable solutions. The term *design pattern* sounds a bit formal to the uninitiated and can be somewhat off-putting when you first encounter it. But, in fact, a design pattern is just a convenient way of reusing object-oriented (OO) code between projects and between programmers. The idea behind design patterns is simple: to catalog common interactions between objects that programmers have often found useful.

A common pattern cited in early literature on programming frameworks is Model-View-Controller (MVC) for Smalltalk [Krasner and Pope, 1988], which divides the user interface problem into three parts (see Figure 1.1):

Figure 1.1. The Model-View-Controller framework.



- **Data Model,** which contains the computational parts of the program
- **View,** which presents the user interface
- **Controller,** which interacts between the user and the view

Each aspect of the problem is a separate object, and each has its own rules for managing its data. Communication between the user, the graphical user interface (GUI), and the data should be carefully controlled; this separation of functions accomplishes that very nicely. Three objects talking to each other using this restrained set of connections is an example of a powerful design pattern.

In other words, a design pattern describes how objects communicate without becoming entangled in each other's data models and methods. Keeping this separation has always been an objective of good OO programming. If you have been trying to keep objects minding their own business, you are probably already using some of the common design patterns. It is interesting that the MVC pattern is used throughout Java 1.2 as part of the JFC, also known as Swing components.

More formal recognition of design patterns began in the early 1990s when Erich Gamma [1993] described patterns incorporated in the GUI application framework, ET++. Programmers began to meet and discuss these ideas. The culmination of these discussions and a number of technical meetings was the publication of the seminal book, *Design Patterns—Elements of Reusable Software*, by Gamma, Helm, Johnson, and Vlissides [1995]. This book, commonly called the Gang of Four, or GoF, book, became an all-time bestseller and has had a powerful impact on those seeking to understand how to use design patterns. It describes 23 common, generally useful patterns and comments on how and when you might apply them. We will refer to this groundbreaking book as *Design Patterns*, throughout this book.

Since the publication of *Design Patterns*, several other useful books have been published. One closely related book is *The Design Patterns Small talk Companion* [Alpert, Brown, and Woolf, 1998], which covers the same 23 patterns but from the Smalltalk point of view. In this book, we refer to it as the *Smalltalk Companion.*

Defining Design Patterns

We all talk about the way that we do things in our everyday work, hobbies, and home life and recognize repeating patterns all the time.

- Sticky buns are like dinner rolls, but I add brown sugar and nut filling to them.
- Her front garden is like mine, except that in mine I use astilbe.
- This end table is constructed like that one, but in this one, the doors replace drawers.

We see the same thing in programming when we tell a colleague how we accomplish a tricky bit of programming so that the colleague doesn't have to recreate it from scratch. We simply recognize effective ways for objects to communicate while maintaining their own separate existences.

Some useful definitions of design patterns have emerged as the literature in this field has expanded.

- "Design patterns are recurring solutions to design problems you see over and over." [*Smalltalk Companion*]
- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." [Pree, 1994]
- "Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design... and implementation." [Coplien and Schmidt, 1995]
- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it." [Buschmann and Meunier, et al., 1996]
- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." [Gamma, Helm, Johnson, and Vlissides, 1993]

But while it is helpful to draw analogies to architecture, cabinetmaking, and logic, design patterns are not just about the design of objects; they are also about *interaction* between objects. One possible view of some of these patterns is to consider them as *communication patterns.*

Some other patterns deal not just with object communication, but also with strategies for object inheritance and containment. It is the design of simple, but elegant methods of interaction that makes many design patterns so important.

Design patterns can exist at many levels, from very low-level specific solutions to broadly generalized system issues. There are now hundreds of patterns in the literature. They have been discussed in articles and at conferences at all levels of granularity. Some are examples that apply widely. A few writers have ascribed pattern behavior to class groupings that apply to just a single problem [Kurata, 1998].

Thus you don't just write a design pattern off the top of your head. Rather, most such patterns are *discovered.* The process of looking for these patterns is called *pattern mining* and is worthy of a book of its own.

The 23 design patterns included in *Design Patterns* all had several known applications and were on a middle level of generality, where they could easily cross application areas and encompass several objects. The authors divided these patterns into three types:

- **Creational patterns:** create objects for you, rather than your having to instantiate objects directly. Your program gains more flexibility in deciding which objects need to be created for a given case.
- **Structural patterns:** help you compose groups of objects into larger structures, such as complex user interfaces and accounting data.
- **Behavioral patterns:** help you to define the communication between objects in your system and how the flow is controlled in a complex program.

We'll be looking at Java versions of these patterns in the chapters that follow. This book takes a somewhat different approach than *Design Patterns* and the *Smalltalk Companion;* we provide at least one complete, visual Java program for each of the 23 patterns. This way you can not only examine the code snippets we provide, but run, edit, and modify the complete working programs on the accompanying CD-ROM. You'll find a list of all the programs on the CD-ROM at the end of each pattern description.

The Learning Process

We have found that learning design patterns is a multiple step process:

1. Acceptance
2. Recognition
3. Internalization

First, you accept the premise that design patterns are important in your work. Then, you recognize that you need to read about design patterns in order to know when you might use them. Finally, you internalize the patterns in sufficient detail that you know which ones might help you solve a given design problem.

For some lucky people, design patterns are obvious tools, and they grasp their essential utility just by reading summaries of the patterns. For many of the rest of us, there is a slow induction period after we've read about a pattern followed by the proverbial "Aha!" when we see how we can apply them in our work. This book helps to take you to that final stage of internalization by providing complete, working programs that you can try out for yourself.

The examples in *Design Patterns* are brief and are in C++ or, in some cases, Smalltalk. If you are working in another language, it is helpful to have the pattern examples in your language of choice. This book attempts to fill that need for Java programmers.

A set of Java examples takes on a form that is a little different than in C++, because Java is more strict in its application of OO precepts—you can't have global variables, data structures or pointers. In addition, we'll see that Java interfaces and abstract classes are a major contributor to how we implement design patterns in Java.

Studying Design Patterns

There are several alternate ways to become familiar with these patterns. In each approach, you should read this book and the parent *Design Patterns* book in one order or the other. We also strongly urge you to read the *Smalltalk Companion* for completeness, since it provides an alternate description of each pattern. Finally, there are a number of Web sites on learning and discussing design patterns.

Notes on Object-Oriented Approaches

The fundamental reason for using design patterns is to keep classes separated and prevent them from having to know too much about one another. Equally important, these patterns help you to avoid reinventing the wheel and allow you to describe your programming approach succinctly in terms that other programmers can easily understand.

Recently, we obtained from a colleague some code that had been written by a very capable summer student. However, in order to use it in our project, we had to reorganize the objects to use the Java Foundation Classes (JFCs). We offered the revised code back to our colleague and were able to summarize our changes simply by noting that we had converted the toolbuttons to Command patterns and had implemented a Mediator pattern to process their actions. This is exactly the sort of concise communication that using design patterns promotes.

OO programmers use a number of strategies to achieve the separation of classes, among them *encapsulation* and *inheritance.* Nearly all languages that have OO capabilities support inheritance. A class that inherits from a parent class has access to all of the methods of that parent class. It also has access to all of its nonprivate variables. However, by starting your inheritance hierarchy with a complete, working class, you might be unduly restricting yourself, as well as carrying along specific method implementation baggage. Instead, *Design Patterns* suggest that you always

Program to an interface and not to an implementation.

Putting this more succinctly, you should define the top of any class hierarchy with an *abstract* class or an *interface,* which implements no methods but simply defines the methods that the class will support. Then, in all of your derived classes you will have more freedom to implement these methods to best suit your purposes.

The other major concept that you should recognize is *object composition.* This is simply the construction of objects that contain other objects, that is, encapsulation of several objects inside another one. While many beginning OO programmers use inheritance to solve every problem, as you begin to write more elaborate programs, the merits of object composition become apparent. Your new object can have the interface that is best for what you want to accomplish without

having all of the methods of the parent classes. Thus the second major precept suggested by *Design Patterns* is

Favor object composition over inheritance.

At first this seems contrary to the customs of OO programming, but you will see any number of cases among the design patterns where we find that inclusion of one or more objects inside another is the preferred method.

The Java Foundation Classes

The Java Foundation Classes, which were introduced after Java 1.1 and incorporated into Java 1.2, are a critical part of writing good graphical Java programs. They were also known during their development as the Swing classes and still are informally referred to that way. They provide easy ways to write very professional-looking user interfaces and allow you to vary the look and feel of your interface to match the platform on which your program is running. Further, they, too, utilize a number of the basic design patterns and thus make extremely good examples for study.

Nearly all the example programs in this book use the JFC to produce the interfaces you see in the example code. Since not everyone may be familiar with these classes and since we are going to build some basic classes from the JFC to use throughout our examples, we include an appendix introducing the JFC and showing the patterns that it implements. While not a complete tutorial in every aspect of the JFC, it does present the most useful interface controls and shows how to use them.

Java Design Patterns

Each of the 23 patterns in *Design Patterns* is discussed in the following chapters, and each discussion includes at least one working program example that has some sort of visual interface to make it more immediate to you. Each also uses the JFC to improve the elegance of its appearance and make it a little more concrete and believable. This occurs despite the fact that the programs are necessarily simple so that the coding doesn't obscure the fundamental elegance of the pattern we are describing. However, even though Java is our target language, this isn't a book on the Java language. We make no attempt to cover all of the powerful ideas in Java; only those which we can use to exemplify patterns. For that reason, we say little or nothing about either exceptions or threads, two of Java's most important features.

Chapter 2. UML Diagrams

We have illustrated the patterns in this book with diagrams drawn using Unified Modeling Language (UML). This simple diagramming style was developed out of work done by Grady Booch, James Rumbaugh, and Ivar Jacobson and resulted in a merging of ideas into a single specification and eventually a standard. You can read details of how to use UML in any number of books, such as those by Booch, Rumbaugh, and Jacobson [1999], Fowler and Scott [1997], and Grand [1998]. We'll outline the basics you'll need in this introduction.

Basic UML diagrams consist of boxes that represent classes. Let's consider the following class (which has very little actual function):

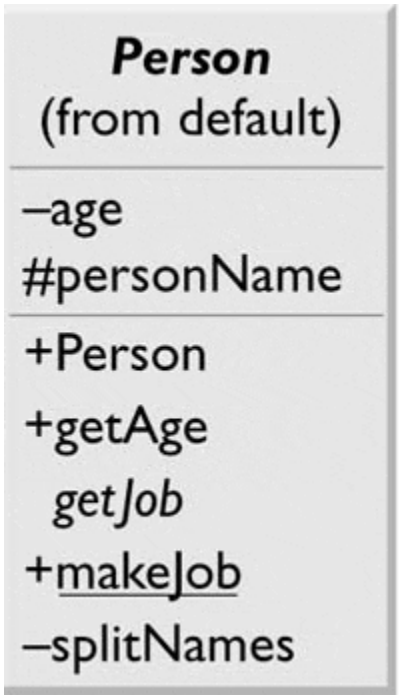
```
public abstract class Person {
    protected String personName;
    private   int    age;

    public Person (String name) {
        personName = name;
    }

    static public String makeJob() {return "hired";}
    public int getAge() {return age;}
    private void splitNames() { }
    abstract String getJob();
}
```

We can represent this class in UML as shown in Figure 2.1.

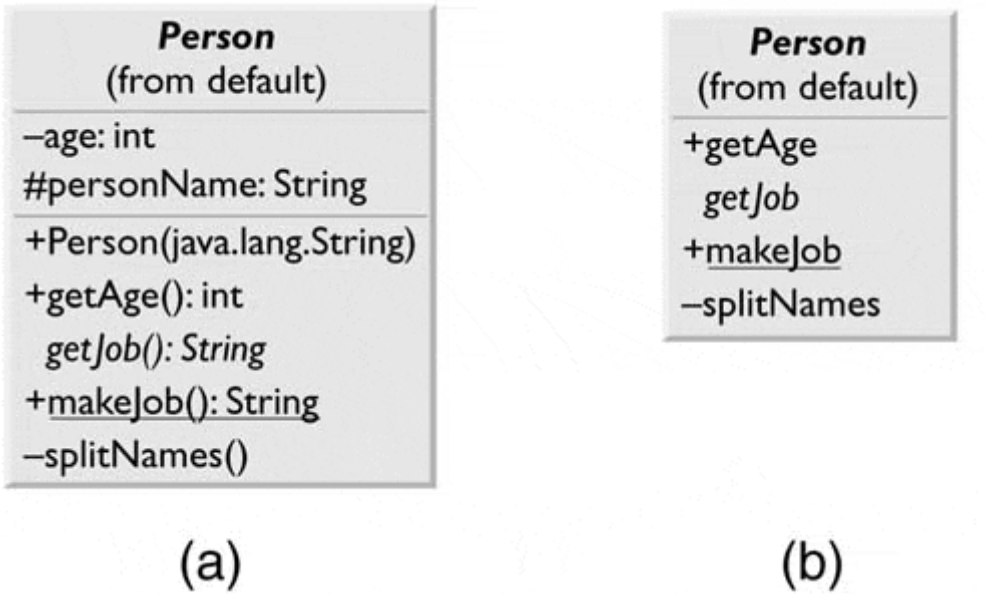
Figure 2.1. The *Person* class, showing private, protected, and public variables and static and abstract methods.



The top part of the box contains the class name and the package name (if any). The second part lists the class's variables, and the bottom lists its methods. The symbols in front of the names indicate that member's visibility, where + means public, - means private, and # means protected. In UML, methods whose names are written in italics are abstract. The class name is also abstract, since it contains an abstract class, so it, too, is in italics. Static methods are shown underlined.

You can also show all of the type information in a UML diagram, where that is helpful, as illustrated in Figure 2.2(a).

Figure 2.2. The *Person* class UML diagram shown both with and without the method types.



UML does not require that you show all of the attributes of a class; usually only the ones of interest to the discussion at hand are shown. For example, in Figure 2.2(b), we have omitted all the variables and the constructor declaration from the methods compartment.

Inheritance

Inheritance is represented using a solid line and a hollow triangular arrow. For the simple Employee class, which is subclass of Person, we write the following code:

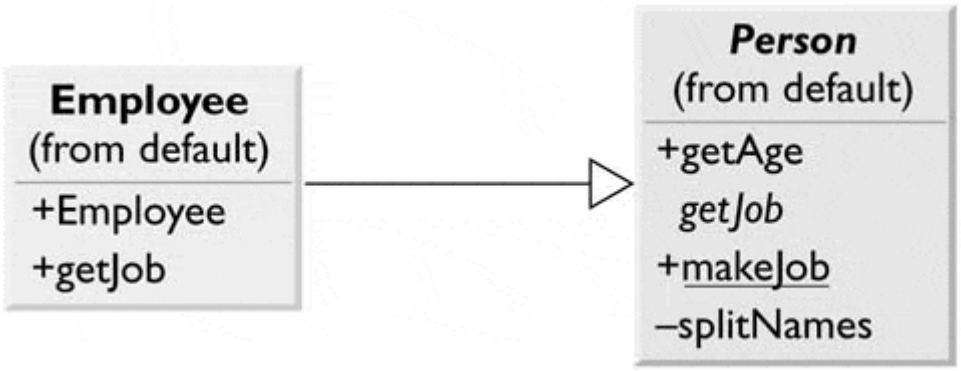
```
public class Employee extends Person {

    public Employee (String name) {
        super (name);
    }

    public String getJob() {
        return "Research Staff";
        }
}
```

This is represented in UML as shown in Figure 2.3.

Figure 2.3. UML diagram showing Employee derived from Person.

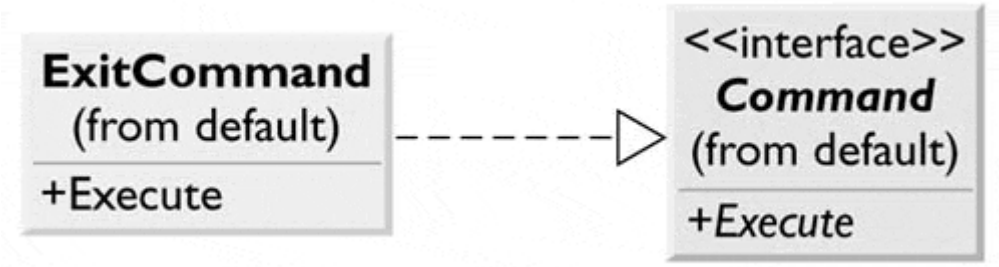


Note that the Employee class name is not in italics. This is because the class is now a concrete class because it includes a concrete method for the formerly abstract *getJob* method. While it has been conventional to show inheritance with the arrow pointing *up* to the superclass, UML does not require this, and sometimes a different layout is clearer or uses space more efficiently.

Interfaces

An interface looks much like inheritance, except that the arrow has a dotted line tail, as shown in Figure 2.4

Figure 2.4. ExitCommand implements the Command interface.



Note that the name <<*interface*>> is shown enclosed within double angle brackets (or *guillemets*).

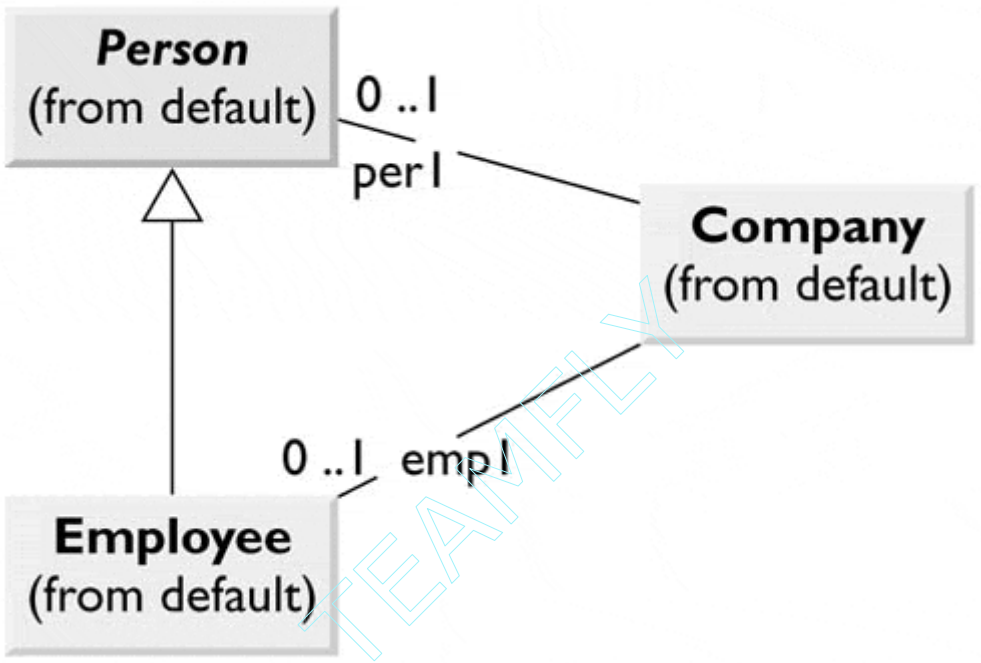
Composition

Much of the time, a useful representation of a class hierarchy must include how objects are contained in other objects. For example, a Company might include one Employee and one Person (perhaps a contractor).

```
public class Company {
  Employee emp1;
  Person   per1;
  public Company() {
    }
}
```

We represent this in UML as shown in Figure 2.5

Figure 2.5. Company contains instances of Person and Employee.



The lines between classes show that there can be 0 to 1 instances of Person in Company and 0 to 1 instances of Employee in Company. If there can be many instances of a class inside another, such as the array of Employees shown here,

```
public class Company1 {
  Employee[] emp1;
  public Company1() {
  }
}
```

we represent that object composition as a single line with either an * or 0, * on it, as shown in Figure 2.6.

Figure 2.6. Company 1 contains any number of instances of Employee.

Some writers use a hollow and a solid diamond arrowhead to indicate containment of aggregates and a circle arrowhead for single object composition, but this is not required.

Annotation

You will also find it convenient to annotate your UML or insert comments to explain which class is calling a method in which other class. You can place a comment anywhere you want in a UML diagram, either enclosed in a box with a turned-down corner or just entered as text. Text comments are usually shown along an arrow line, which indicates the nature of the method that is called, as shown in Figure 2.7.

Figure 2.7. A comment is often shown in a box with a turned-down corner.

UML is a powerful way of representing object relationships in programs, and the full specification contains more diagram features. The previous discussion covers the markup methods used in this text.

JVISION UML Diagrams

All of the UML diagrams in this book were drawn using the JVISION program from Object Insight. This program reads in the actual compiled classes and then generates the UML class diagrams. Many of these class diagrams have been edited to show only the most important methods and relationships. However, the complete JVISION diagram files for each design pattern are stored in that pattern's directory on the accompanying CD-ROM. Thus you can run your copy of JVISION and read in and investigate the detailed UML diagram starting with the same drawings you see here in the book.

Visual SlickEdit Project Files

All of the programs in this book were written using Visual SlickEdit 4.0 using the project file feature. Each subdirectory on the CD-ROM contains the project file for that project so that you can load the project and compile it as we did.

Section 2: Creational Patterns

All of the Creational patterns deal with ways to create instances of objects. This is important because your program should not depend on how objects are created and arranged. In Java, of course, the simplest way to create an instance of an object is by using the new operator:

```
fred = new Fred();    //instance of Fred class
```

However, doing this really amounts to hard coding, depending on how you create the object within your program. In many cases, the exact nature of the object that is created could vary with the needs of the program. Abstracting the creation process into a special "creator" class can make your program more flexible and general. The six Creational patterns follow:

- **Factory Method pattern** provides a simple decision-making class that returns one of several possible subclasses of an abstract base class, depending on the data that are provided. We'll start with the Simple Factory pattern as an introduction to factories and then introduce the Factory Method pattern as well.
- **Abstract Factory pattern** provides an interface to create and return one of several families of related objects.
- **Builder pattern** separates the construction of a complex object from its representation so that several different representations can be created, depending on the needs of the program.
- **Prototype pattern** starts with an instantiated class, which it copies orclones to make new instances. These instances can then be further tailored using their public methods.
- **Singleton pattern** is a class of which there may be no more than one instance. It provides a single global point of access to that instance.