Preface

Interface-Oriented Design explores how you can develop software with interfaces that interact with each other. We'll look at techniques for breaking down solutions into these interacting interfaces and then for determining appropriate implementations for these interfaces to create well-structured programs. We have plenty of examples that will show you ways to create effective designs composed of interfaces to objects, components, and services. And we'll even have some fun along the way.

You've probably learned about (and experienced) software development using object-oriented design. Interface-oriented design concentrates on the interfaces of modules, which may or may not be implemented with object-oriented languages. Designs that emphasize interfaces are loosely coupled—and that's a good thing. If you have only an interface to which to code, you cannot write code dependent on an implementation, which helps keep us honest.

Distributed computing, such as service-oriented architectures, places a particular emphasis on interfaces. The interfaces may be procedure oriented (such as Remote Procedure Calls) or document oriented (such as web services). We'll explore the transparency and loose coupling traits that are key to distributed interfaces to help you build better distributed systems.

Inheritance is often a tricky technique to get correct—it is often one of the most abused features in object-oriented languages. We'll look at designs that employ inheritance versus ones that emphasize interfaces to demonstrate the trade-offs between the two.

This ongoing emphasis on interfaces may seem a bit extreme. But by looking at one extreme, you'll start to see a different viewpoint that can give you fresh insights into your current approach to software development.

About the Cover

Concentrating on interfaces is key to decoupling your modules.[1] You probably learned to type on a QWERTY keyboard, as shown on the cover. That interface is the same regardless of whether the implementation is an old-fashioned typewriter, a modern electric typewriter, or a computer keyboard. There have been additions to the keyboard, such as function keys, but the existing layout continues to persist.

But other layouts, such as Dvorak,[2] are more efficient for typing. You can switch your computer keyboard to use an alternate layout; the switching module works as an adapter. Inside the keyboard driver, the keystrokes are converted to the same characters and modifiers (e.g., Shift, Alt, etc.) that are produced by the regular keyboard.

The QWERTY keyboard layout was derived from concern about implementation. According to one web site,[3] "It is sometimes said that it was designed to slow down the typist, but this is wrong; it was designed to allow *faster* typing—under a constraint now long obsolete. In early typewriters, fast typing using nearby type-bars jammed the mechanism. So Sholes fiddled the layout to separate the letters of many common digraphs (he did a far from perfect job, though; *th*, *tr*, *ed*, and *er*, for example, each use two nearby keys). Also, putting the letters of *typewriter* on one line allowed it to be typed with particular speed and accuracy for demos. The jamming problem was essentially solved soon afterward by a suitable use of springs, but the keyboard layout lives on."

Creating interfaces that are easy to use and decoupling their use from their implementation are two facets that we'll explore a lot in this book. (And you may have thought the cover was just a pretty picture.)

So, What Else Is in Here?

Simple Unified Modeling Language (UML) diagrams show the class and interface organization throughout the book. We use Interface-Responsibility-Interaction (IRI) cards, a variation of Class-Responsibility-Colla-

[1] As Clemens Szyperski puts it, "The more abstract the class, the stronger the decoupling achieved." See `http://www.sdmagazine.com/documents/sdm0010k/`.

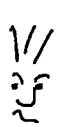
[2] See `http://www.microsoft.com/enable/products/dvlayout.aspx`.

[3] See `http://www.ctrl-c.liu.se/~ingvar/jargon/q.html`.

boration (CRC) cards, as the primary method for creating interface-oriented designs. You'll also find code examples in multiple languages to show how interfaces are implemented in those languages.

On a terminology note, the OMG Modeling Language Specification (revision 1.3) uses the phrase *realize interface*, which means a component implements the services defined in the interface. Allen Holub in *Holub on Patterns* uses the term *reify*, which means "consider an abstract concept to be real." I thought about alternating one of those verbs with the word *implements*, but they are less familiar. If you get tired of seeing *implementing*, just imagine it's *reifying*.

You will see a few sections that look like this:



"Joe Asks..."
These sections provide answers for some common questions.

Acknowledgments

I would like to thank my reviewers for reading the draft copies of this book and contributing numerous comments that helped improve the book. Thanks to David Bock, Tom Ball, Ron Thompson, Gary K. Evans, Keith Ray, Rob Walsh, David Rasch, Carl Manaster, Eldon Alameda, Elias Rangel, Frédérick Ros, J. Hopkins, Mike Stok, Pat Eyler, Scott Splavec, Shaun Szot, and one anonymous reviewer. Thanks to Michael Hunter, an extraordinary tester, who found a lot of "bugs" in this book. Thanks to Christian Gross, a reviewer who gave me many suggestions that just couldn't fit into this book and to Kim Wimpsett for proofreading the manuscript.

I appreciate Andy Hunt, my editor and publisher, for encouraging me to write this book, and all his help with the manuscript.

Thanks also to Leslie Killeen, my wife, for putting up with me writing another book just as soon as I finished my previous book, *Prefactoring*, winner of the 2006 Software Development Jolt Product Excellence Award.[4]

And now, here we go!

[4]See `http://www.ddj.com/dept/architect/187900423?pgno=3/`.

Chapter 1

Introduction to Interfaces

We'll start our introduction to interfaces with ordering a pizza. The pizza order is not just to ensure that reading begins on a full stomach; by using non-computer-related matter, we can explore some general topics relating to interfaces, such as polymorphism and implementation hiding, without getting bogged down in technology. Then we'll switch to real systems to show code and textual interfaces as background for topics in later chapters.

1.1 Pizza-Ordering Interface

If you're a real programmer, or a serious pizza eater, you've probably performed the following scenario hundreds of time.

The Pizza Order

You're hungry so you call your favorite pizza joint.

"Yo," the voice on the other end answers, ever so politely.

"I'd like a large pizza," you reply.

"Toppings?" the voice queries.

"Pepperoni and mushrooms," you answer.

"Address?" is the final question.

"1 Oak Street," you reply.

"Thirty minutes," you hear as the phone clicks off.

The steps you have just performed conform to the PizzaOrdering interface, which is implemented by thousands of pizza shops all over the

world. You provide information on the size and toppings and where to deliver the desired pizza. The response is the amount of time until it will be delivered.

Using the same interface but with potentially different implementations is the central concept of *polymorphism*. Multiple pizza shops provide the same functionality. If I picked up the phone, dialed the number, and handed the phone to you, you might not know from which particular shop you were ordering. But you would use the same interaction with any pizza shop. In addition, you would not have any knowledge of how they really make the pizza. This interface does not constrain how the pizza shop makes its pizza, how many people they employ, the brand of flour they use, or anything else about the implementation.

How did you find an implementation of the PizzaOrdering interface? You probably used an implementation of the PizzaOrderingFinder interface. You looked in a directory under *Pizza*, went down the list of names, and picked one you used before or one that had been recommended to you by a friend. If you're in a new place, you may just start with the first name on the list. We'll explore other ways to find pizza shops later in this book.

PizzaOrderingFinder returns a pizza shop. Each pizza shop is different; otherwise, there would be no need for more than one brand of pizza shop. Shops vary on the quality of implementation: how fast they make the pizza and the tastefulness of the result. You may be aware of the variations between different pizza shops and ask PizzaOrderingFinder to find one whose characteristics fit your needs. If you're really hungry, you might substitute speed for quality. Pizza shops also vary on price (the requirements on the resources in your wallet). Whether resource requirements bear any relationship to quality is an interesting question that we'll discuss later in regard to software.

The Pizza Interfaces

Now for those readers who are having a hard time relating pizza to software development, let's create a more formal definition of the PizzaOrdering interface. We'll use this example later in the book as we describe the various facets of interfaces.

```
interface PizzaOrdering
    enumeration Size {SMALL, MEDIUM, LARGE}
```

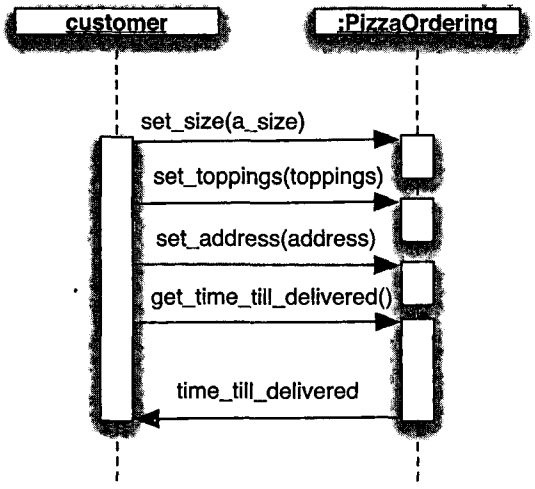


Figure 1.1: PIZZAORDERING SEQUENCE DIAGRAM

```
enumeration Toppings {PEPPERONI, MUSHROOMS, PEPPERS, SAUSAGE}
set_size(Size)
set_toppings(Toppings [])
set_address(String street_address)
TimePeriod get_time_till_delivered()
```

Note that setting the address in our simulated conversation actually returned the time_till_delivered. Having a function that sets a value return a value of a completely different type gives me a bad feeling in my stomach, and it ain't from the pepperoni. So I added a method to retrieve the time_till_delivered.

Figure 1.1 shows the PizzaOrdering interface in a UML sequence diagram. The diagram captures the sequence of interaction between a customer and an implementation of the PizzaOrdering interface (i.e., a pizza shop). For those less familiar with sequence diagrams, I'll explain the details in Chapter 2.[1]

Here's a more formal description for how you might find a pizza shop:

[1]You may look at this interface and say, "I know another way to do this." Good! Write it down. If you don't see an equivalent interface later in this book, send it to me. I'll present several variations of interfaces in this book, but there is never one "best" answer, and alternatives are always worth considering.

```
interface PizzaOrderingFinder
    PizzaOrdering find_implementation_by_name
        (String name_of_pizza_shop);
    PizzaOrdering find_first_implementation()
    PizzaOrdering find_default_implementation()
```

This interface finds vendors that implement the PizzaOrdering interface. An implementation might not necessarily be a pizza shop; it might be a regular restaurant that offers pizzas as a menu item for delivery. It could be Sammy's Computer and Snack Shop that started offering pizzas when Sammy discovered that software development made programmers hungry.

You could realize this interface in a number of ways. You could perform the operations yourself by grabbing a phone book and getting a number and dialing it. You could ask a friend to do that for you. You could call the operator and ask for a particular pizza shop. This variation of possible implementations is another example of polymorphism in action. The PizzaOrderingFinder interface illustrates another pattern that will be discussed later in this book.[2]

1.2 Real-Life Interfaces

Software is not developed by pizza alone, even though it fuels much development. Let's first see what a software interface is all about, and then we'll explore examples of interfaces that exist in current systems.

What Is an Interface?

Interfaces are declared in code. In some languages, such as Java and C#, **interface** is a keyword. It applies to a set of method signatures (names and parameter lists). You use **implements** in Java to show that a class implements an interface. For example:

```
interface SampleInterface
    {
    double findAverage(double [] numbers);
    }
```

[2] In particular, the Factory Method pattern. See *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995).

Other Interfaces

Even a non-object-oriented language, such as C, can support interfaces. In C, an interface is a set of functions that apply to a common concept, such as the set of functions that operate on a file, as we'll explore in the next section.[4]

Text is another form of interface. The text specifies the functions to perform (typically in a human-readable format but not necessarily). For example, the command prompt of Windows is a textual interface—you type a command to perform along with parameters. We will use the term *textual interface* to differentiate this type of interface from one in a programming language.

Unix Devices

The Unix operating system has a standard interface to all devices (hard drives, displays, printers, and keyboards) and files, and it is expressed in the C language. This interface is an example of polymorphism in a non-object-oriented language: you always use the same functions, regardless of the type of device or file.

To the user, each device appears as a file. Every device has an entry in the file system in the /dev directory. For example, a printer entry might be /dev/lp0 (for line printer 0). You open and write to a printer in the same way you write to a file. The basic set of functions include the following:[5]

```
open(filename, flags);
    // Returns a file descriptor.
    // Flags include O_RDONLY, O_WRONLY or O_RDWR
close(file_descriptor);
read(file_descriptor, buffer, count);
write(file_descriptor, buffer, count);
```

For example, you open a file with this:

```
file_descriptor1 = open("/home/ken/myfile.txt", O_WRONLY);
```

To open a printer, you use:

```
file_descriptor2 = open("/dev/lp0", O_WRONLY);
```

[4] In C, a structure that contains members that are function pointers acts as a polymorphic interface. For example, the file_operations structure for device drivers in Linux describes the functions that a driver must support.

[5] This is a simplified version.

After this point, the remainder of your program reads and writes using the file descriptors. The calls to these functions are the same, regardless of whether you are communicating to a file or the printer. For example, both of these function calls write nine characters to the corresponding device:

```
write(file_descriptor1, "Something", 9);
write(file_descriptor2, "Something", 9);
```

The polymorphism that this interface provides is powerful. Suppose you write a program that is intended to read from the keyboard and write to the display. You can freely substitute other devices for the keyboard and the display. From the command line, this is known as I/O redirection. Because there is no difference between reading from a file and reading from the keyboard, you can put commands into a file and have a program read the commands from that file.[6]

For example, the cat program (short for *concatenate*) is nominally set so that input comes from the keyboard and output goes to a display. Using I/O redirection, if you write

```
cat < input_file > output_file
```

cat reads from input_file and writes to output_file. Suppose you want to display your entire hard disk, assuming you have the necessary privilege to read the device. You can use the following:

```
cat < /dev/hda1
```

You can copy an entire disk to another disk as simply as doing this:

```
cat < /dev/hda1 > /dev/hda2
```

The Interface

Object diehards might not consider the preceding set of C functions to be an interface. The set of functions follows a common pattern that starts with initiating a service request (open()) that returns an opaque data identifier (a file descriptor). The identifier is passed to other functions (write(), read(), and close()) for subsequent processing. Service patterns like this can be transformed into a more familiar-looking interface. A realization of this interface will have a file descriptor as a private data member, but that is an implementation detail that is not part of an interface.

[6]You can also use the polymorphic behavior to set the output of one program to be the input of another program. Unix systems refer to this as a *pipe*.

```
interface File
    open(filename, flags) signals UnableToOpen
    read(buffer, count) signals EndOfFile, UnableToRead
    write(buffer, count) signals UnableToWrite
    close()
```

We'll look at this interface again in a few chapters, so if this thin veneer doesn't particularly appeal to you at the moment, wait a while, and we'll fix it up.

Textual Interfaces

Since every device in Unix operates through a common interface, you need some method to communicate device-specific commands. One way is to use a textual interface for these directives. The commands are sent as a string of characters. A common textual interface is the original modem interface created by the manufacturer Hayes. For example, some of the common commands are as follows:

AT (attention, commands follow)

D (dial a number)

T (dial a number using tones, rather than pulses)

To dial a number, you send the modem the "ATDT9195551212" sequence. The replies from the modem are also textual. If the connection was successful, the modem returns the "CONNECT" string. If the connection was not successful, it returns a string denoting the error, as the "NO CARRIER" or "BUSY" string. To hang up the phone, you send the "ATH" string.

An advantage of a textual interface is that you can store the commands in a file. Later you can read the file and send them to the device.

Other textual interfaces include the common Internet protocols such as Simple Mail Transfer Protocol (SMTP) and File Transfer Protocol (FTP). For example, FTP commands include the following:

```
open hostname     #Open a connection
get filename      #Get a file
close             #Close a connection
```

You may run across other textual interfaces, although you might not necessarily think of them as such. Both Unix and Windows can create text files containing commands and data for printers; a standard language for these commands is PostScript. The document is text, so

it can be stored in a file or sent to a printer device. An example of PostScript commands to print "Hello world" on a page is as follows:

```
/Times-Roman findfont
12 scalefont
setfont
newpath
200 300 moveto
(Hello world) show
showpage
```

The printer interprets the commands and prints the page. The set of printers that understand PostScript can be considered polymorphic implementations of the PostScript interface.[7] Just like pizza shops, their output may vary in quality and speed. But they all implement the same functionality.

We'll examine in Chapter 3 how to translate a textual interface, such as the FTP commands, into a programmatic interface. The PostScript file acts like a document-style interface. We'll explore document-style interfaces in more detail in Chapter 6.

The GUI Interface

Packages that support graphical user interfaces make extensive use of polymorphism. In both Java and Windows, you draw in a graphics context. In Java, the context is the Graphics class. For Windows, the graphic context for the Microsoft Foundation Classes (MFC) is the CDC (for *device context*) class. The graphics context could refer to a display, a printer, an in-memory screen buffer, or a metafile. The user drawing on the graphics context may not be aware to what they are actually outputting.

In Java, you call drawString() to output a string to the display at a particular position:

```
void drawString(String string, int x, int y);
```

Given a reference to a Graphics object (say g), to output the string you would code this:

```
g.drawString("Hello world", 200, 300);
```

For example, in MFC, you write text to the device context using the following method:

[7] You can display PostScript files on Windows and Unix with GSView (`http://www.cs.wisc.edu/~ghost/gsview/get47.htm`).

```
BOOL TextOut(int x, int y, const CString & string);
```

With a pointer to a CDC object (say, pDC), the code to output a string is as follows:[8]

```
pDC->TextOut(200, 300, "Hello world");
```

- Both graphics contexts are state-based interfaces; they contain the current font with which the text is drawn as well as a plethora of other items. In Chapter 3, we'll see how we can translate this state-based interface to a non-state-based interface.
- The PostScript text in the previous section and these two code examples perform the same operation. All three represent a realization of an interface that you could declare as follows:

```
interface DisplayOutput
    write_text(x_position, y_position, text)
```

I'll describe many of the interfaces in this book at this level of detail. This is to emphasize the functionality that an interface provides, rather than the detailed code for any particular language.

1.3 Things to Remember

We've begun our exploration of interfaces with an emphasis on polymorphism. You've seen interfaces with a variety of functionality—from ordering pizza to writing to devices to displaying text. You've seen the same functionality as expressed in a programmatic interface and a textual interface. In the next chapter we'll get down to business and discuss contracts that modules make when they implement an interface.

[8]The values of 200 and 300 in these examples do not refer to the same coordinate system. For PostScript, the values are in points (1/72"). For drawstring(), the values are in pixels.

Chapter 2

Interface Contracts

In this chapter, we're going to examine contracts. These contracts are not the ones you make when you order pizzas but are the contracts between the users of interfaces and their implementation. If you or the implementation violates the contract, you will not get what you want, so understanding contracts is essential.

We'll start by considering three laws that all implementations should obey, regardless of what services they offer. Then we'll look at Bertrand Meyer's *Design by Contract* that outlines conditions for methods. You cannot be sure that an implementation fulfills its contract until you test it; contracts for pizzas and for files offer an opportunity to show types of tests you can apply to interfaces. Also, you don't measure the quality of a pizza by just its speed of delivery. The nonfunctional qualities of pizza are also important, so we conclude with a look at implementation quality.

2.1 The Three Laws of Interfaces

One way to express one of the facets of the contract for an interface is with three principles inspired by the Three Laws of Robotics. Isaac Asimov first presented these laws in 1950 in his short-story collection, *I, Robot.*[1] Since computer programs often act like robots, this analogy of the laws seems appropriate.

[1] You can also find the original laws, as well as more details, on the web page at `http://www.asimovonline.com/`.

1. An Interface's Implementation Shall Do What Its Methods Says It Does

This law may seem fairly obvious. The name of a method should correspond to the operations that the implementation actually performs.[2]

Conversely, an implementation should perform the operations intended by the creator of the interface. The method should return a value or signal an error in accordance with the explained purpose of the method.

If the purpose and meaning of a method are not unambiguously obvious from the method's name and its place within an interface, then those aspects should be clearly documented.[3] The documentation may refer to interface tests, such as those described later in this chapter, to demonstrate method meaning in a practical, usage context.

An implementation needs to honor the meaning of a return value. The sample PizzaOrdering interface in the previous chapter included the method TimePeriod get_time_till_delivered(). The return value represents the amount of time until the pizza shows up on your doorstep. A delivery should take no more than this amount of time. If TimePeriod is reported in whole minutes, an implementation that rounds down an internal calculated time (e.g., 5.5 minutes to 5 minutes) will return a value that does not correspond to the described meaning.

2. An Interface Implementation Shall Do No Harm

Harm refers to an implementation interfering with other modules in a program or with other programs. The user of an interface implementation should expect that the implementation performs its services in an efficient manner.[4]

In particular, an implementation should not hog resources. Resources in this case might include time, memory, file handles, database connections, and threads. For example, if the implementation requires connecting to a database that has limited connections, it should disconnect as soon as the required database operation is complete. Alter-

---

[2]This is also known as the *Principle of Least Surprises*.

[3]Michael Hunter suggests, "They should be documented regardless. Conversely, if they need documentation, the name should be improved."

[4]Andy Hunt suggests that implementation should use only those resources suggested by its interface. For example, an interface whose purpose is to write to the screen should not require a database connection.

Liskov Substitution Principle

The first law corresponds to the Liskov Substitution Principle (LSP), which states that a subtype should be indistinguishable in behavior from the type from which it is derived. For object design, methods in a base class should be applicable to derived classes. In another words, a derived class should obey the contract of the base class. Thus, any object of a derived class is "substitutable" as an object of the base class. Barbara Liskov and Jeannette Wing introduced this principle in their paper "Family Values: A Behavioral Notion of Subtyping."*

*The full discussion is at http://www.los.mit.edu/publications/pubs/pdf/MIT-LCS-TR-562b.pdf.

natively, the implementation could use a shared connection and release that connection as soon as the operation finishes.[5]

If an implementation uses excessive memory, then it may cause page faults that can slow down not only the program itself but also other programs.

3. If An Implementation Is Unable to Perform Its Responsibilities, It Shall Notify Its Caller

An implementation should always report problems that are encountered and that it cannot fix itself. The manner of report (e.g., the error signal) can either be a return code or be an exception. For example, if the implementation requires a connection to a web service (as described in Chapter 5) and it cannot establish that connection, then it should report the problem. If there are two or more providers of a web service, then the implementation should try to establish a connection with each of the providers.

[5]For example, implementations of J2EE Enterprise JavaBeans (EJBs) interfaces share connections.

Joe Asks...

What's a Page Fault?

If you've ever seen your computer pause and the disk light come on when you switch between two programs, you've seen the effects of page faults. Here's what happens.

A computer has a limited amount of memory. Memory is divided into pages, typically 16 KB each. If a number of programs are running simultaneously, the total number of pages they require may exceed the amount of physical memory. The operating system uses the disk drive to store the contents of pages that cannot fit in physical memory and that programs are not currently accessing. The disk drive acts as "virtual" memory.

When a program accesses a page not in physical memory (a page fault), the operating system writes the current contents of a memory page to disk and retrieves the accessed page from the drive. The more memory required by programs, the greater the chance that virtual memory is required and thus the greater possibility of page faults and the slower the program will run.

- Only if it is unable to connect to any of them should it report the problem to the caller.[6,7]

- The errors that are denoted on the interface (either return codes or exceptions) are part of the interface contract; an interface should produce *only* those errors. An implementation should handle nonspecified situations gracefully. It should report an error if it cannot determine a reasonable course of action.

[6]Michael Hunter notes that there can be hidden dangers if each interface implementation implements a retry process. An implementation may call another interface implementation. If both of them perform retries, then the report of failure to the user will take longer. In one application, this failure report took more than five minutes because of the number of interfaces in the process.

[7]For debugging or other purposes, the implementation may log the unsuccessful attempts to connect with each of the services.

2.2 Design by Contract

To successfully use an interface, both the caller and implementer need to understand the contract—what the implementation agrees to do for the caller. You can start with informal documentation of that agreement. Then, if necessary, you can create a standard contract.

Bertrand Meyer popularized Design by Contract in his book *Object-Oriented Software Construction* (Prentice Hall, 1997). In the book, he discusses standards for contracts between a method and a caller.[8] He introduces three facets to a contract—preconditions, postconditions, and class invariants.

The user of an interface needs to ensure that certain conditions are met when calling a method; these stipulations are the *preconditions*. Each method in an interface specifies certain conditions that will be true after its invocation is complete; those guarantees are the *postconditions*. The third aspect is the *class invariant*, which describes the conditions that every object instance must satisfy. When dealing with interfaces, these class invariants are typically properties of a particular implementation, not of the interface methods.

If a precondition is not met, the method may operate improperly. If the preconditions are met and a postcondition is not met, the method has not worked properly.[9] Any implementation of an interface can have weaker preconditions and stronger postconditions. This follows the concept that a derived class can have weaker preconditions and stronger postconditions than the base class.

Contract Checking

An interface implementation is not required to check the preconditions. You may assume that the user has met those preconditions. If the user has not, the implementation is free to fail. Any failures should be reported as in the Third Law of Interfaces.

If you decide to check the preconditions, you can do so in a number of ways:

---

[8] See `http://archive.eiffel.com/doc/manuals/technology/contract/` for a discussion of contracts for components.

[9] You can use the Object Constraint Language (OCL) in UML to document the preconditions and postconditions.

Pizza Conditions

Suppose a pizza-ordering interface specified that the allowed toppings are pepperoni, mushrooms, and pineapple. An implementation that provides only pepperoni and mushrooms would work only for a limited range of pizzas. It has stronger preconditions. A pizza shop that also offered broccoli and ham has weaker preconditions. An implementation with weaker preconditions can meet the contract for the interface. One that has stronger preconditions cannot.

Likewise, suppose that your requirement for delivery time is a half hour. A pizza shop that may take up to one hour has a weaker postcondition. One that may deliver in ten minutes has a stronger postcondition. An implementation with stronger postconditions meets the contract; one with weaker postconditions does not.

- You could use code embedded within each method to check the conditions.
- In a less imposing way, you could use aspects,[10] if a particular language supports them.
- A third way is to use a contract-checking proxy. Chapter 11 describes the Proxy pattern.[11]
- nContracts is a C# language specific method. nContracts uses C# attributes to specify the preconditions and postconditions. It does not require change to the implementation source (like aspects), but works like a contract checking proxy.[12]

A contract-checking proxy is an implementation of the interface that checks the preconditions for each method. If all preconditions are met, the proxy calls the corresponding method in the implementation that does the actual work. Otherwise, it signals failure. If the corresponding method returns and the postconditions are not met, it could also signal failure.

[10]See aspect-oriented programming at `http://aosd.net/`

[11]The pattern can also be considered the Decorator pattern. See also *Design Patterns*.

[12]See `http://puzzleware.net/nContract/nContract.html`.

Pizza Contract

Let's take a look at the PizzaOrdering interface. What are the contractual obligations of this interface? OK, the pizza shop agrees to make and deliver a pizza, and you also have to pay for the pizza. But you have other facets. The interface requires a certain flow to be followed. If you started by saying "1 Oak Street," the order taker may get really flustered and try to make you an Oak-sized pizza. So, the conditions for each of the methods are as follows:

Method	Preconditions	Postconditions
`set_size()`	None	Size set
`set_toppings()`	Size has been set	Toppings set
`set_address`	Size and toppings set	Address set
`get_time_till_delivered`	Size, toppings, address set	None

Now you may want an interface that is a little less restrictive. You might think you ought to be able to set the size, toppings, and address in any order. You would eliminate the preconditions for the three set methods, but the one for get_time_till_delivered() would still remain. For a product as simple as a pizza, the strictness of the order is probably unwarranted. For a more complex product, the method order may be essential. For example, if you're ordering a car, you can't choose the options until you've chosen the model.

File Contract

For a more computer-related example, let's examine the contract for the File interface we introduced in Chapter 1. Here's the interface again:

```
interface File
    open(filename, flags) signals UnableToOpen
    read(buffer,count) signals EndOfFile, UnableToRead
    write(buffer, count) signals UnableToWrite
    close()
```

Before we investigate the contract for this interface, let's examine the abstraction that this interface represents. A realization of this interface has these responsibilities:

Method	Preconditions	Postconditions

`open(filename, flags)` signals `UnableToOpen`	None	If (for writing)
		  if user has permission
		  File is opened for writing
		
		if (for reading)
		  if file exists and
		  user has permission
		  File is opened for reading

`read(buffer, count)` signals `EndOfFile`, `UnableToRead`	File opened for reading	If not at end of file
		  If count < bytes left in file
		  Set file position to bytes after current
		  else
		  Set file position to end of file

`write(buffer, count)` signals `UnableToWrite`	File opened for writing	File position incremented by count
`close()`	File is open	File closed

Figure 2.1: PRE- AND POSTCONDITIONS FOR FILE INTERFACE

For writing out
: Output a sequence of bytes to the device or file in the order in which they are sent.

For reading from
: Input a sequence of bytes from the device or file in the order in which they are received or read.

For files (not devices)
: Save the output sequence of bytes for later retrieval by another process. The saved sequence should persist after a system shutdown and reboot.

The contract for this interface includes the preconditions and postconditions shown in Figure 2.1.

If the caller does not call open(), the other methods will fail. They should inform the caller of the failure. They should not cause harm in the event of this failure (see the Second Law of Interfaces). For example,

suppose an implementation initialized a reference in open(). Without calling open(), that reference is uninitialized (e.g., null). If the write method attempted to use that uninitialized reference and an exception or memory fault resulted, that would violate the second law.

Protocol

You can list the operations involved in an interface, including preconditions, postconditions, parameters and their types, return values, and errors signaled. But you need more than just how to use operations and when to use each operation. You also need to know the protocol to the interface—the set of allowable sequences of method calls. The preconditions often imply a sequence, but they may not. The protocol can also show the callbacks that an interface may create, events that are generated, or observers that are called.

For the File interface, you must follow a distinct sequence of methods. You must open a file before you can read, write, or close it. The protocol can be documented in words, in a sequence diagram, in a state diagram, or in test code.

To express the protocol in words, I use a form of a use case.[13] A use case describes an interaction between a user and a system that fulfills a goal. An internal use case describes an interaction between a caller and an interface. To differentiate between the two cases, I refer to an internal use case as a *work case*. Use cases are usually expressed in technology-independent terms; work cases might include the names of the methods in the interface. The work cases demonstrate the protocol for an interface.

For the File interface, we have the following work cases:

Work Case: Read a File

1. Open a file for reading (open()).
2. Read bytes from file (read()).
3. Close file (close()).

[13] For more details, see *Writing Effective Use Cases* by Alistair Cockburn (Addison-Wesley, 2000) and http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm.

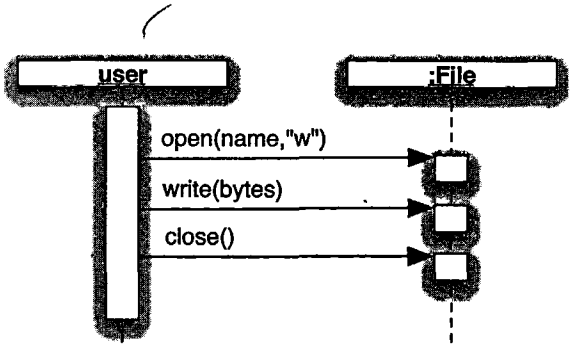


Figure 2.2: SEQUENCE DIAGRAM FOR THE PROTOCOL

Work Case: Write a File

1. Open a file for writing (open()).
2. Write bytes to file (write()).
3. Close file (close()).

A UML sequence diagram can also demonstrate the protocol. Figure 2.2 shows the sequence diagram that corresponds to the second work case.

A UML state diagram provides a different way of indicating a protocol. For each state, it shows what methods you can call. Calls to methods may alter the state of an implementation and therefore alter what other methods you may call.

For a file, the states include CLOSED, and OPEN_FOR_READING, OPEN_FOR_WRITING. If an error, such as reading a file opened for writing, causes the file to become unusable, you could have an ERROR state.[14]

Figure 2.3, on the next page, shows the state diagram for File. Note that read() and write() transition into the ERROR state, if the file has been opened in the opposite mode. The diagram does not show transitions

[14]Note that in many languages, there are input streams and output streams. You cannot invoke read on an output stream or write on an input stream, since the methods do not exist in the corresponding interfaces. That separation of interfaces decreases the number of possible state transitions and possible ways that errors can be generated.

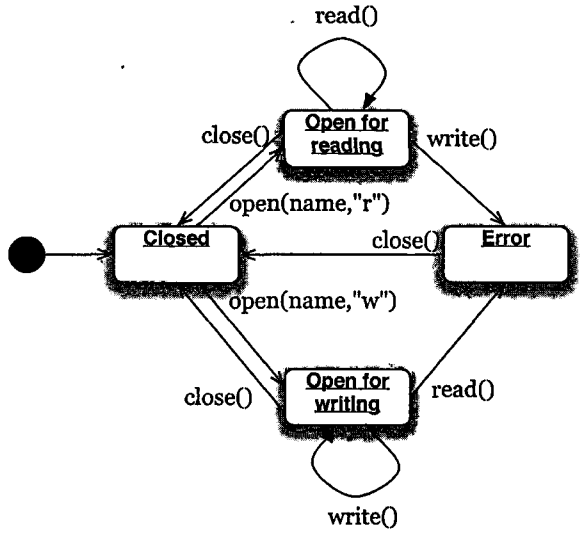


Figure 2.3: STATE DIAGRAM FOR FILE

from ERROR for read() and write(). So, these method calls are ignored in that state, according to the diagram.[15]

2.3 Testing Interfaces against Contracts

There is no question that automated unit and acceptance testing dramatically help in developing software. Extreme Programming (XP) and other agile processes have reemphasized and reinvigorated the concepts of testing. But what are you testing for? Essentially, you are testing to ensure that an implementation of an interface meets its contract. You can specify a contract in documentation; however, a coded test can make the contractual obligation clearer. Plus, it can verify that the contract is met. As a general rule, no interface definition is complete until you have all the contractual tests successfully running for at least one implementation.

[15]We leave it up to you to determine for your particular language or operating system whether there is an implied ERROR state and what happens with an incorrect use of read() and write(). See whether the documentation clearly describes the situation.

Joe Asks...

What's a UML Sequence Diagram?

A UML sequence diagram shows a sequence of interactions between modules. The interactions take the form of messages or method calls. The modules may be instances of classes or implementations of interfaces. A box and dotted line represent a module. The name is within the box. The name may represent an actual module or an unspecified implementation of an interface. To show the latter, you use a colon before the name of the interface. The dotted line represents the "life" of the module. For example, for an object, the line represents how long the object exists.

You show calls to a method by drawing a line between the caller and the callee and giving the name of the method. Lines in the reverse direction show return values from methods.

When an implementation is "active" (that is, doing something rather than just existing), the lifeline shows up as a rectangle, rather than a dotted line.*

*This explanation gives only facets of sequence diagrams that we use in this book. More details are at `http://www.sparxsystems.com/resources/uml2_tutorial/`.

The *Design Patterns* book states, "Design to an interface, not an implementation." A parallel guideline exists in testing. "Test to an interface, not an implementation". This is termed *black box* testing.[16] You test an interface without looking inside to see how it is coded. With more services being provided remotely, you probably will not have access to the code. Therefore, you can test only to the interface.

Writing tests for an interface can also help you work out difficulties in the interface. You can find ambiguities or unclearness in the contractual obligations, the method definitions, or the protocol. If you find that your interface is hard to test, then it probably will be hard to use. If this happens, you can redesign your interface without even having coded an implementation.

[16] *White box* testing uses knowledge of the code to devise tests, typically tests that check performance and robustness.

Tests for the File Contract

For the File interface, we devise tests for each of the work cases, as well as for the individual methods. The tests for the two work cases would include the following:[18]

Test Case: Write Bytes to File

1. Open a file for writing.
2. Write a number of bytes that have known values.
3. Close file.
4. Verify that the bytes have been written.

Test Case: Read Bytes from File

1. Open a file for reading.
2. Read a number of bytes.
3. Verify that the bytes are equal to known values.
4. Close file.

We can create "*misuse*" cases (or "*miswork*" cases). Misuse cases state ways that a caller or user might accidentally (or deliberately) violate the protocol. Here are some misuse cases for File:

Test Case: Read a File Opened for Writing

1. Open a file for writing.
2. Read bytes from file. This operation should signal an error.
3. Close file.

Test Case: Write to an Unopened File

1. Write bytes to file. This operation should signal an error.

You should ensure that you have tests for all possible sequences of method calls. A state diagram such as Figure 2.3, on page 24, can clarify the sequences that need testing. For example, in addition to the

[18]These tests represent only a portion of the tests. We would also create variations that include writing zero bytes, a single byte, and a large number of bytes.

previous tests, we should try writing to a file opened for writing and then reading from it.

2.4 Levels of Contracts

Christine Mingins and Jean-Marc Jézéquel suggest that there are several levels of contracts.[19] The levels are:

- Basic type contracts as in typed programming languages
- Semantic contracts that include the preconditions and postconditions
- Performance contracts for real-time systems
- Quality of service contracts that are hard to quantify

You need to test for conformance with all these contracts. For typed programming languages, the compiler enforces the type contract (we'll take a look at untyped languages in the next section). Testing for quality of service contracts is usually more difficult than testing for performance contracts: quality may include resource usage, reliability, scalability, and the other "ilities."[20] We'll examine quality more shortly.

Other nonfunctional facets of interfaces include transactional behavior (does it participate in a transaction?), security (e.g., is it called by or on behalf of authorized users?), and logging. These facets can be applied by aspect-based code, such as Java aspects, or by frameworks, such as J2EE, in which the implementations reside. In either case, the interface implementation has code only for the essential behavior. The aspects or framework take care of the rest. We don't cover security or transactions in this book, because they require entire books by themselves.

Explicit versus Implicit Type Contracts

Languages differ in how they enforce the parameter data type contracts. In some languages, such as Java and C++, you have to be explicit about the data types of the parameters that are passed to a method. In other languages, such as Python, Perl, and Ruby, the data type is implicit.

---

[19] See `http://archive.eiffel.com/doc/manuals/technology/bmarticles/sd/contracts.html`.

[20] See *Software Requirements* by Karl Wiegers (Microsoft Press, 2003) for a full discussion of the "illities."

You do not specify a parameter type. How a method uses a parameter implies the type.

Let's look at an example of implicit typing in the Observer pattern.[21] In this common pattern, one object is interested in changes in the state of another object. The interested party is the observer, and the watched party is the observed.

Suppose we have a Customer class. An observer may be interested in the event that the name or the address changed in the class. If we program this class in a dynamically typed language such as Ruby, we might code this:[22]

```
class Customer
    def add_observer(observer)
        @observer = observer
    end

    def address=(new_address)
        @address=new_address
        @observer.notify_address_change(new_address)
    end

    def name=(new_name)
        @name=new_name
        @observer.notify_name_change(new_name)
    end
```

Note that observer must have two methods. If set_address() or set_name() is called and the observer does not have a matching notify() method, a runtime error occurs. The error occurs because the implicit contract (having these two methods) has been violated.

In Java, the methods required by an observer are described by an explicit interface:[23]

```
interface CustomerObserver
    notify_address_change(address)
    notify_name_change(name)
```

You state explicitly that the observer must have these methods by its type declaration in the parameter list for add_observer(). For example:

[21] See *Design Patterns* for more details on the Observer pattern.

[22] In Ruby, we could also use the Observer mixin.

[23] This has been reduced from the usual multiple observers to a single observer to keep the code simple.