```
class Customer
    {
    CustomerObserver the_observer;
    void add_observer(CustomerObserver observer)
        {
        the_observer = observer;
        }
    void set_address(Address an_address)
        {
        // set address, then
        the_observer.notify_address_change(an_address)
        }
    void set_name(name)
        {
        // set name, then
        the_observer.notify_name_change(name)
        }
    }
```

The Java compiler complains if you pass an object that does not provide the CustomerObserver interface to the add_observer() method.

In both Ruby and Java, you should write sufficient tests to ensure that the observer/observed code works as you intended. So, one might argue that an explicit interface is not required. However, typing the parameter as CustomerObserver enforces in code the contract that the object must support those methods. With implicit typing, the need for the methods must be shown in the documentation.

2.5 Contractual Quality

We introduced a question in the original pizza example of whether price has any relationship to quality; we might also ask whether speed of delivery is more important than tastiness. We're going to let these questions of pizza be an area of discussion for you and your fellow readers of this book.

But these questions do have important analogies in software—the quality of implementation trade-offs. For example, one implementation of an interface might be faster but require more memory.

The speed of an implementation can vary based on the demands placed upon it. This makes it tricky if you cannot determine in advance the requirements for the caller. For example, suppose you had a method sort(Object()) that sorted the passed array using a comparison function

of Object class.[24] There are many different algorithms for sorting an array. They differ in speed and memory resources required.[25] The algorithms' variance in speed is often due to the degree of sorting already in the array. For example, if an array is already sorted, a bubble sort is the fastest, but the quicksort algorithm is faster than a bubble sort most of the time. If the array is already sorted in the reverse direction, quicksort is slower by far. Because of its recursive nature, quicksort can demand somewhat more resources than the bubble sort. By using interfaces to a sorting method, rather than specifying a particular one, you can always substitute an implementation that is more aligned with the requirements that spontaneously arise.

The sort example is a specific case of the general issue that implementation trade-offs are important. By using interfaces, rather than concrete implementations, you can make choices appropriate to a given situation without having to alter code that uses the interface.

2.6 Things to Remember

An implementation of an interface should obey the three laws:

- Do what its methods say it does.
- Do no harm.
- Notify its caller with meaningful errors if unable to perform its responsibilities.

Establish a contract for each interface (either formally or informally):

- Indicate the preconditions and postconditions for each method.
- Specify the protocol—the sequence of allowable method calls.
- Optionally, spell out nonfunctional aspects such as performance or quality of implementation.
- Create tests to check that an implementation performs according to its contract and that it obeys the three laws.

[24]You might supply a comparison function to the method to make the sort more flexible.

[25]Donald E. Knuth's classic book on sorting and searching gives a whole slew of algorithms: *Art of Computer Programming, Volume 3: Sorting and Searching* (Addison-Wesley,1998).

Chapter 3

Interface Ingredients

Now that we've covered the basics of interfaces, it's time to examine the ingredients of interfaces. Almost every interface you employ or develop has a combination of these ingredients, so understanding them helps you appreciate the whole pie. In this chapter, we'll look at the spectrum of interfaces from data-oriented to service-oriented and cover the trade-offs in three distinct approaches to data-access interfaces.

You can always adapt an interface paradigm from one type to another to make it more amenable to your project, so we'll explore how to adapt a stateful interface to a stateless one. Then we'll look at transforming a textual interface into a programmatic one and creating an interface from a set of existing related methods.

3.1 Data Interfaces and Service Interfaces

There is a spectrum between data interfaces and service interfaces. We use the term *data interface* when the methods correspond to those in a class that contains mostly attributes. The methods in the interface typically set or retrieve the values of the attributes.[1] We use the term *service interface* for a module whose methods operate mostly on the parameters that are passed to it.

One example of a data interface is the classic Customer class. Customer usually has methods like

- set_name(String name)

[1]Data interfaces also correspond to JavaBeans or pretty much any class that is a wrapper around attributes with a bunch of getter/setters.

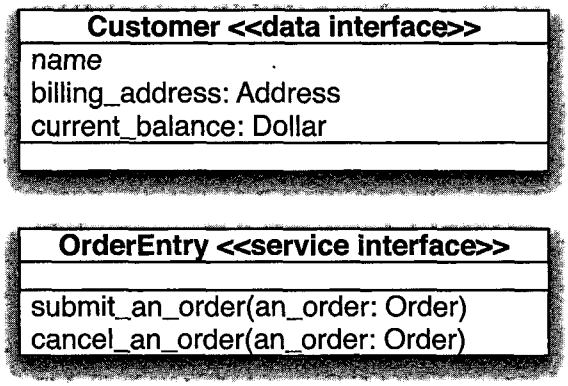

Figure 3.1: DATA VS. SERVICE INTERFACE

- set_billing_address(Address billing_address)
- get_current_balance().

Each of these methods affects or uses an attribute in the class. Implementations of data interfaces have state, which consists of the set of values of all attributes in the class.

Service interfaces have methods that act on the parameters passed to them, rather than the attributes of the implementation, for example the methods submit_an_order(Order an_order) and cancel_an_order(Order an_order). Figure 3.1 shows how data interfaces have just attributes and service interfaces have just methods.

Service interface implementations usually have no attributes or only ones that are associated with providing the service, such as connection information that identifies where to submit an order or where to find the current price for a stock. Implementations of service interfaces may have no state, other than that of internal configuration values such as this connection information.

This data versus service interface comparison is not pure black and white, but rather a spectrum. An interface can range from a data transfer object (DTO), whose methods refer only to attributes of the object, to a command interface, which usually contains only service methods. We could move away from a pure data interface by adding methods to the Customer interface. We might add charge_an_amount(), which

Entities, Control, Boundary

In *Object-Oriented Software Engineering*, Ivar Jacobsen introduced three stereotypes for objects: entity, boundary, and control. An *entity* depicts long-lived objects. *Boundary* objects communicate between the system and the actors (users and external systems). A *control* object represents behavior related to a specific use case. It communicates with boundary objects and entity objects to perform an operation.

These stereotypes relate to the data and service interfaces. Data interfaces correspond to the entity objects. The underlying data mechanism (e.g., database table or XML file) is opaque to the user of the entity object. An interface such as Pizza, which contains just the size and toppings, is an entity.

A boundary corresponds to a service interface. You push a button on a GUI or make a call to a method, and the underlying service is performed. The PizzaOrdering interface presented in Chapter 1 is a boundary interface.

A controller also corresponds to a service interface. Its methods are typically called by a boundary interface. It can embody business rules or services. A PizzaMaker that controls the making of the Pizza() could exist between the PizzaOrdering() and a Pizza(). The PizzaMaker() would be a controller.

alters current_balance; mail_statement(), which mails the current_balance to the address; or is_credit_worthy(), which applies some business rules to determine whether to extend credit to the customer.

Let's take the PizzaOrdering interface in the first chapter and transform it into two interfaces on each end of the spectrum. First we make a pure DTO—a Pizza class containing just data on the pizza. For example:

```
class Pizza
    set_size(Size)
    set_topping(Topping)
    Size get_size()
    Topping [] get_toppings()
```

We now create a service interface that accepts a Pizza and places the order:

```
interface PizzaOrderer
    TimePeriod order_pizza(Pizza)
```

The method calling this interface first creates a Pizza and then passes the Pizza to order_pizza().

We can turn the Pizza class into a class endowed with more behavior, which is a guideline for object-oriented design. Let's add a method so that a Pizza orders itself:

```
class Pizza
    // as above, plus:
    order()
```

Pizza's order() method could call an implementation of the PizzaOrderer interface. One implementation could communicate the order over the telephone; another implementation could fax it or email it. The user of Pizza does not need to know about PizzaOrderer, unless they intend to change the implementation that order() uses.[2]

In Chapter 1, you ordered a pizza over the phone. If PizzaOrderer represented a phone-based system, before accessing order_pizza(), you need to call the shop. If we include that operation in this interface, it would look like this:

```
interface PizzaOrderer
    call_up()
    TimePeriod order_pizza(Pizza)
    hang_up()
```

Now PizzaOrderer represents a *service provider interface*, a variation of the service interface. A service provider adds methods to the interface that control the life cycle of the service provider. These methods are often called *initialize*, *start*, or *stop*. Java applets, servlets, and session beans are examples of service provider interfaces.

3.2 Data Access Interface Structures

You may run across different paradigms for interfaces that access data, so it's a good idea to appreciate the differences among them. An interface can provide sequential or random retrieval of data. Users can either pull data or have it pushed upon them.

[2] We explore ways to configure different implementations in Chapter 7.

Sequential versus Random Retrieval

Data can be available in a sequential or random manner. For example, the Java FileInputStream class allows only sequential access, while RandomAccessFile allows access to the data in a file in any order.

The same dichotomy exists within collections and iterators. An iterator interface allows access to a single element in a collection at a particular time. Some styles of iterators, such as Java's Iterator or C++'s forward iterators, permit only one-way access. You have to start at the beginning of the collection and continue in one direction to the end. On the other hand, a vector or array index, or a C++ random-access iterator, allows random access to any element in the set. If you have data available with only sequential access and you want it to have random access, you can build an adapter. For example, you can create a vector and fill it with the elements from an iterator.

Other examples of sequential vs. random access are two Java classes for accessing the data in an XML file. The Simple API for XML (SAX) parser provides for sequential access to the XML elements; SAX does not keep the data in memory. On the other hand, the Document Object Model (DOM) allows random access. It creates an in-memory representation of the XML data. Note that a DOM parser can use a SAX parser to help create the memory representation. These two interfaces have corresponding advantages and disadvantages.

SAX: SEQUENTIAL ACCESS

Advantage—requires less resources to parse the file

Disadvantage—application cannot change the XML data

DOM: RANDOM ACCESS

Advantage—application can change the XML data

Disadvantage—requires memory to store the entire document

We'll revisit SAX and DOM in a little more detail in a later section.

Pull and Push Interfaces

Interfaces move data in one of two ways: push or pull. You ask a pull-style interface—for example, a web browser—for data. Whenever you desire information, you type in a URL, and the information is returned. On the other hand, a push-style interface transfers data to you. An email subscription is a push-style interface. Your mail program receives

information whenever the mail subscription program sends new mail. You don't ask for it; you just get it.

You can use either a pull style or a push style when going through a collection.[3] An example of a pull style is the typical iteration through a list or array:

```
Item an_item
for_each an_item in a_list
    {
    an_item.print()
    }
```

For each element in a_list, the print() method is explicitly called. The push style for this operation is as follows:

```
print_item(Item passed_item)
    {
    passed_item.print()
    }
a_list.for_each(print_item)
```

The for_each() method iterates through a_list. For each item, for_each() calls the print_item() method, which is passed the current item on the list.

For each language, the actual code for the push style is different. For example, in C++, you can use the for_each() function in the Standard Template Library (STL). With this function, each item in the vector is pushed to the print_item() function.

```
void print_item(Item item)
  {
  cout << item <<' ';
  }
vector <Item> a_list;
for_each(a_list.begin(), a_list.end(), print_item);
```

In Ruby, the code could be as follows:

```
a_list = [1,2,3]
a_list.each { |passed_item| passed_item.print_item()}.
```

PUSH STYLE

Advantage—can be simpler code, once paradigm is understood

[3] *Design Patterns* refers to pull and push styles for a collection as *internal* and *external* iterators.

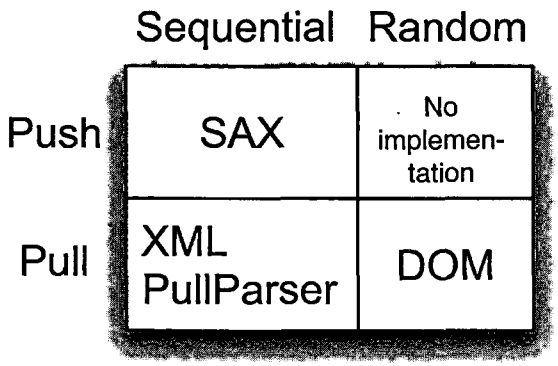

Figure 3.2: EXAMPLES OF DATA INTERFACES

PULL STYLE

Advantage—appears as a common control style (e.g., loop) in multiple languages

One from Each Column

Pull/push style and sequential/random styles can be intermixed in combinations. As an example of a set of combinations in a specific area, let's revisit XML parsing. SAX is push/sequential; DOM is pull/random. There is also a pull-style sequential interface called XMLPullParser.[4]

Figure 3.2 shows how these three styles relate. The "No implementation" box shows a variation for which no current implementation exists.[5] Depending on what elements you want to retrieve from an XML file, what you want to do with the elements, and memory constraints, you choose one of these interface styles to create simpler code. To compare how you might employ each of these versions, let's take a look at some logic in pseudocode. In each of these examples, we print the count for one element in an XML file. The XML file looks like this:

[4]See `http://www.xmlpull.org/v1/doc/api/org/xmlpull/v1/XmlPullParser.html` for full details.

[5]We can't think of a need for this variation, so that may be why no one has created one.

```
set_bold()
set_italic()
set_underline()
```

Second, you could have a single method with a parameter:

```
enumeration FontModifier { BOLD, ITALIC, UNDERLINE, NORMAL }
set_font_modifier(FontModifier)
```

The behavior in both these cases is the same—each method alters the appearance of the formatted text.[6] With only three font modifiers, the trade-off between the two versions does not weigh heavily one way or the other. In the first version, the method call is shorter, but you have more methods. However, the second interface can be more resilient to change. If you add a STRIKETHOUGH modifier, the method signatures of the implementers do not have to change. Previous versions of the set_font_modifier() method will not output STRIKETHROUGH, but they may fail gracefully.

A similar trade-off occurs with the Observer pattern. In our example from Chapter 2, we had two methods for the CustomerObserver. They were the following:

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

If you add another event to the interface (say notify_balance_change()), you have to change all the places where this interface is implemented. Instead, you could cut down the interface to a single method. The method could provide an indicator of what has changed, like this:

```
interface CustomerObserver
    enumeration ChangeType {ADDRESS, NAME}
    notify_change(ChangeType value)
```

If you add BALANCE to ChangeType, you do not have to change any observers. Alternatively, you might have the method pass the new and old values:

```
interface CustomerObserver
    notify_change(Customer old_customer, Customer new_customer)
```

[6]The two sets of methods differ on a more subtle level. A reviewer noted that the definition of each alternative method did not specify whether the other font modifiers were reset. For example, does calling set_bold() turn off all the other modifiers, or do you need unset() methods? If set_font_modifier() allowed for multiple modifiers (e.g., BOLD and ITALIC), then its contract could be to turn off all other modifiers.

The observer would need to determine what the differences between the old_customer and the new_customer were. But now you would not have to change CustomerObserver at all.

- MANY METHODS

  Advantage—implementer does not have to determine type of parameter

  Disadvantage—implementer has to implement all the methods

- SINGLE METHODS

  Advantage—can be more resilient to change, because new methods do not have to be implemented

  Disadvantage—must check parameter type to determine flow

This trade-off appears in listener interfaces in Java that are observer interfaces for many of the GUI widgets. For example, to listen to events occurring in a window, you must implement WindowListener. This interface contains many methods:

```
interface WindowListener
    {
    windowActivated(WindowEvent)
    windowClosed(WindowEvent)
    windowClosing(WindowEvent)
    windowDeactivated(WindowEvent)
    windowDeiconified(WindowEvent)
    windowIconified(WindowEvent)
    windowOpened(WindowEvent)
    }
```

A listener knows what event has occurred by which method is called. The parameter that is passed is needed only for the details of the event, not the type of event.

You may be interested in only one of the events, but you still need to code a body for all the methods in the interface. This can be annoying, so Java supplies a WindowListenerAdapter, which is really a WindowListener with default method bodies. Each method body does nothing. If your class inherits from WindowListenerAdapter, it needs to override the methods only for the events for which you are interested.

In this configuration, you really have not inherited an implementation. You have just overcome a constraint of the language that requires a class to implement all methods in an interface. The Java listener interface could have been designed differently. There could have been multiple interfaces, like this:

```
interface WindowClosedListener
    {
    windowClosed(WindowEvent)
    }
interface windowIconifiedListener
    {
    windowIconified (WindowEvent)
    }
```

If you wanted to listen to only a single event, you would need to implement only a single method. You would not need a WindowListenerAdapter.

In C# and Ruby, the callback method equivalent to windowClosed() does not need to be part of an interface.[7] The method can simply be any one that matches the signature for a callback method. For example, in Ruby, you hook up a listener for the window-closed event with the following:

```
root = TkRoot.new()
frame = TkFrame.new(root)
frame.bind('Destroy') { puts 'Window Closed' }
```

The code following `frame.bind` is executed when the window is destroyed.

3.4 Stateless versus Stateful Interfaces

An interface implementation can either contain state (stateful) or not contain state (stateless). In a stateful interface, the methods operate differently based on the current state, which is changed by the sequence of method invocations. In a stateless interface, the behavior is not dependent on the history of method invocations. Its behavior is always the same. A stateful versus stateless PizzaFinder service illustrates the difference. Let's examine the trade-offs between these two types. Along the way, we'll see how a stateful interface can be adapted to a stateless one.[8]

Imagine if I were to call my mother and ask her to find the number for Tony's Pizza. She looks it up and says, "Well, there are five." I say, "Thanks, Mom, please give me the first one." I hang up and call it only to find out they don't deliver to my neighborhood. I can call up my mom

[7] In C#, the keyword **delegate** is used to denote the signature of a callback method. In Ruby, there are no interface declarations.

[8] Thanks to David Bock for providing this example.

and say, "Hi, Mom, please give me the next one." She can then respond with the next one. The conversation with my mother has state. She remembers that I had asked about Tony's Pizza and where I was in the list.

Contrast this to calling the operator. I ask for the first Tony's Pizza on the list; they respond; I call, and it isn't the right one. If I call back information and say, "Give me the next," the operator will say, "Next one what, sir? I don't know what you are talking about." The conversation with the operator is stateless. I will have to tell the operator on the second phone call that I am calling about Tony's Pizza, and I want the second one in the list.

- STATELESS

  Advantage—a small number of operators can service many requests. My mom would not be able to juggle more than a few requestors at a time.

- STATEFUL

  Advantage—there is less chatter to get the same amount of work done.

Let's look at some program examples of stateful and stateless interfaces. We introduced programming a GUI in Chapter 1. In most languages, you have the equivalent of a GraphicsContext interface that contains the state of the current font modifiers (e.g., BOLD). Suppose you want to write a series of text with different modifiers. With a stateful interface, you might code the equivalent of this:

```
GraphicsContext graphics_context;
graphics_context.set_font_modifier(BOLD)
graphics_context.print_text('In Bold')
graphics_context.print_text('Also in Bold')
graphics_context.set_font_modifier(ITALIC)
graphics_context.print_text('In Italics');
```

You first set the `graphics_context` to print in BOLD, which alters the state of the GraphicsContext. All text from that point is printed in BOLD, until the state of the graphic context is set to ITALIC.

The opposite form—a stateless interface—requires that you specify all the font information in each call. The interface does not keep track of the font modifiers. Calls to a stateless interface would look like this:

```
graphics_context.print_text(BOLD, 'In Bold')
graphics_context.print_text(BOLD, 'Also in Bold')
graphics_context.print_text(ITALIC, 'In Italics')
```

Consistency

If you're creating an interface, you should be consistent in your parameter placement. You may have noticed that drawString() and TextOut() have parameter lists that are reversed. Similarly, the parameter lists to create a font are reversed. Which one is correct? Take a poll of your fellow programmers, and see whether there is a preference.

I don't have a definitive guideline other than being as consistent as possible with the other interfaces you use in your organization.

STATELESS

Advantage—order of the method calls does not matter

Disadvantage—parameter lists are longer

STATEFUL

Advantage—parameter lists shorter

Disadvantage—order of method calls important

If you move the call to `set_font_modifier(ITALIC)` down one line in the stateful example, then "In Italics" prints in bold. You can alter the sequence of calls in the stateless interface, and the text is always printed with the same font modifier.

The graphics contexts for Java and MFC (Graphics and CDC, respectively) are stateful. The state includes values such as the current font (e.g., Times Roman) for text output and the foreground and background color of text. In both frameworks, you set the font, and then subsequent drawing is in that font. In this Java example, graphics is the graphics context:

```
Font font = new Font('Times-Roman',12)
graphics.setFont(font);
graphics.drawString('Hello world', 200, 300);
```

MFC works in a similar fashion. The graphics context is represented with a pointer, pDC.

```
CFont font;
font.CreateFont(12,'Times-Roman');
pDC->SelectObject(&font);
pDC->TextOut(200,300,'Hello world');
```

These graphics context methods eventually wind up calling methods in Windows' Graphics Display Interface (GDI), such as DrawText(). The GDI methods then call methods in the Display Driver Interface (DDI), such as DrvTextOut(). This function sends text out to a device. Here are some of the relevant parameters:[9]

```
BOOL DrvTextOut( STROBJ * pointer_string_object,
   FONTOBJ * pointer_font_object,
   RECTL * pointer_opaque_rectangle );
```

This interface is stateless. The GDI passes everything that is needed—the font in which to render the text (pointer_font_object), the position at which to write the text (pointer_opaque_rectangle), and the text itself (pointer_string_object). The current state of the graphics context is used to fill in those parameters. So, the stateful interface of Java and MFC invokes the stateful interface of GDI, which then executes the stateless DDI. A stateful interface is transformed into a stateless one.[10] This statelessness simplifies coding of the device driver. It does not have to remember "and what font was I printing in?"

Each device driver provides an implementation of the methods such as DrvTextOut(). For a printer, the driver implementation converts the information into a type specific to that particular printer. The driver could output the text in PostScript or Hewlett Packard's Printer Control Language (PCL). Or the driver could create a bitmap of the text in the processor and output that bitmap to the printer. The details of each printer are opaque to the user of the Java or MFC interface.

If the output to the printer is a PostScript file, then the driver has created a textual interface from a procedural one. So, the flow goes from a stateful procedural interface at the user level to a stateless procedural interface at the driver level to a textual interface. This sort of conversion from one type of interface to another is common in many systems.

3.5 Transformation Considerations

"If you can't be with the one you love, love the one you're with," wrote Stephen Stills. You don't have to love an interface you're given. If you don't like the way an interface works, transform it into one you like.

---

[9] You can find the full interface at `http://msdn.microsoft.com`.

[10] A device driver may keep internal state to improve performance, but callers of the interface do not rely upon that fact or even know it.

Nonobject Interface into Object Interface

Let's examine a few ways that the Unix input/output interface can be transformed into an object-oriented interface. In Chapters 1 and 2, we presented the File interface:

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

An implementation of this interface—such as MyFile—has a private instance variable that represents the common element (the file_descriptor) of the original C functions:

```
class MyFile implements File
    private file_descriptor
    open(filename, flags) signals UnableToOpen
    read(buffer, count) signals EndOfFile, UnableToRead
    write(buffer, count) signals UnableToWrite
    close()
```

This straightforward translation of the original functionality raises an interesting question as to the protocol for the interface. Should you have a class that allows for possibly invalid instances? For example, you could create an instance of the MyFile type without it being connected to an open file. Then other methods (e.g., read(), write()) should check the state and not attempt to perform an operation on an unopened file.

On the other hand, if a File object should exist only in the open state, then a constructor should assume the functionality of the open method. The destructor can perform the closing of the file. Objects that cannot exist in an invalid state can make it easier to check the preconditions and postconditions that were covered in Chapter 2.

A class with a constructor looks like this:

```
class FileWithConstructor
    private file_descriptor
    FileWithConstructor(filename, flags) signals UnableToOpen
    read(buffer, count) signals EndOfFile, UnableToRead
    write(buffer, count) signals UnableToWrite
    destructor()
```

The FileWithConstructor class can implement a different interface, such as FileAlwaysOpen. For example:

```
interface FileAlwaysOpen
    read(buffer, count) signals EndOfFile, UnableToRead
    write(buffer, count) signals UnableToWrite
```

Use a factory method to make use of this interface more opaque:

```
interface FileAlwaysOpenFactory
    FileAlwaysOpen get_file(filename, flags) signals UnableToOpen
```

Inside the get_file() method, you create an instance of FileWithConstructor, which implements the FileAlwaysOpen interface. The pseudocode for the interface looks like this:

```
FileAlwaysOpen get_file(filename, flags) signals UnableToOpen
return new FileWithConstructor(filename, flags)
```

Note that the parameters to the get_file() method are the same as the parameters for the constructor. You could always create a FileWithConstructor directly:

```
FileAlwaysOpen file = new FileWithConstructor('myfile.txt', READ_ONLY);
```

More Specific Interfaces

We discussed in Chapter 1 that you might want to send specialized commands to particular types of devices. The Unix/Linux ioctl() method and textual interfaces are two ways of sending device-specific commands. To hide the details of communication, you could create device-specific interfaces. The methods in the interface are applicable to only a single device. For example, for a modem, you might have the following:

```
interface ModemDevice
  hangup()
  dial_phone_number(PhoneNumber) signals NoAnswer, NoDialTone, NoCarrier
    // sends 'ATDT' + PhoneNumber to modem
```

The dial_phone_number() method sends the "ATDT" string, as shown in Chapter 1. The user of this interface does not need to remember the specific command string.

With the general FileAlwaysOpen factory, you could perhaps create a FileAlwaysOpen and then ensure that it is a ModemDevice before you try to dial a number with it. For example:

```
FileAlwaysOpen a_stream= FileAlwaysOpenFactory.get_file('/dev/mdm',
  READ_WRITE);
if (a_stream is_a ModemDevice)
    (ModemDevice) a_stream.dial_phone_number('5551212')
```

Alternatively, you could create a more specific factory method that returns only a device that implements the ModemDevice interface. For example:

```
class Modem implements FileAlwaysOpen, ModemDevice

Modem m = ModemFactory.get_modem();
m.dial_phone_number('5551212');
```

You could make an interface that parallels the "never invalid" concept of FileAlwaysOpen. In this case, ModemDevice is hidden. The factory always returns an instance of a dialed number.

```
class PhoneConnectionAlwaysOpen implements FileAlwaysOpen

interface PhoneConnectionFactory
    PhoneConnectionAlwaysOpen  get_phone_connection(PhoneNumber)
        signals NoAnswer, NoDialTone, NoCarrier

PhoneConnectionAlwaysOpen m =
    PhoneConnectionFactory.get_phone_connection('5551212');
```

Transforming Textual Interfaces to Programmatic Interfaces

Like the modem commands, you can transform most textual interfaces into an interface with methods. For example, the File Transfer Protocol (FTP) mentioned in Chapter 1 has commands such as the following:

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

You can transform these commands into a method interface by creating methods for each command with the appropriate parameters and the appropriate error signals. For example:

```
interface FTPService
    open(hostname, username, password) signals
        UnableToConnect, IncorrectUsernameOrPassword
    get(filename) signals NoSuchFile, UnableToCommunicate
    close()
```

On the server, the textual commands received from the client can be transformed into method calls against a similar interface. The service directory example in Chapter 9 gives another example of converting from a method interface to a textual interface and back again.

DEVICE SPECIFIC

Advantage—hides device/protocol implementation issues

3.6 Multiple Interfaces

You are not stuck with a one-to-one relationship between an interface and a module. A module can implement more than one interface. When you implement multiple interfaces, you may have interface collision problems. As an example, we'll look at a pizza shop that also sells ice cream.

Ice Cream Interface

You're hungry for an ice cream cone. The PizzaOrdering interface doesn't allow you to order ice cream. So, you find an implementation of an IceCreamOrdering interface that appears as follows:

```
enumeration Flavor { Chocolate, Vanilla, Coffee, Strawberry, ..}
enumeration ConeType { Sugar, Waffle}
interface IceCreamOrdering
    set_number_scoops(Count)
    set_cone_type(ConeType)
    set_flavors(Flavor [])
```

Now an ice cream–only shop may implement the IceCreamOrdering interface. On the other hand, a shop that offers the PizzaOrdering interface may also offer the IceCreamOrdering interface. To do so, the shop must provide an implementation of all methods in both interfaces as:

```
enumeration Toppings {PEPPERONI, MUSHROOMS, PEPPERS, SAUSAGE}
class CombinedPizzaIceCreamShop implements PizzaOrdering
  and IceCreamOrdering
    set_number_scoops(Count)
    set_cone_type(ConeType)
    set_flavors(Flavor [])
    set_size(Size)
    set_toppings(Toppings [])
    set_address(String street_address)
    TimePeriod get_time_till_delivered()
```

Trying to implement multiple interfaces can lead to some interesting issues. Suppose that IceCreamOrdering also had a set_toppings() method:

```
enumeration Toppings {WHIP_CREAM, CHERRY, CHOCOLATE_JIMMIES, PINEAPPLE}
set_toppings(Toppings [])
```

If the Toppings enumeration contained all the choices for both pizzas and ice cream, then set_toppings() would need to differentiate between which item was being ordered. Otherwise, you might get sausage on your ice cream. With most languages, the enumeration of Toppings for IceCream and Pizza can be placed in its own namespace (e.g., in a mod-

ule or package). So, CombinedPizzaIceCreamShop would implement two methods that took different Topping enumerations:

```
set_toppings(PizzaOrdering.Toppings [])
set_toppings(IceCreamOrdering.Toppings [])
```

If the methods took more primitive parameters (e.g., String or double), then differentiating them would be harder. For example, suppose you had the following interface methods:

```
interface PizzaOrdering
    set_toppings(String toppings)
interface IceCreamOrdering
    set_toppings(String toppings)
```

then the combined shop could have only a single method:[11]

```
class CombinedPizzaIceCreamShop implements PizzaOrdering
  and IceCreamOrdering
    set_toppings(String toppings)
```

The set_toppings() method has to be a bit more complicated to handle both ice cream and pizza toppings, or else you might be eating chocolate syrup on your pizza or pepperoni on your vanilla cone.[12]

3.7 Things to Remember

We've looked at several key ingredients in creating interfaces, and we explored the spectrum between data and service interfaces. When designing code, keep in mind the advantages and disadvantages of several approaches to interfaces:

- Sequential versus random data access
- Push versus pull interfaces
- Stateful versus stateless interfaces
- Multiple methods versus single methods

A module may implement multiple interfaces by providing method implementations for all of the methods. You can transform interfaces from non-object-oriented code to object-oriented interfaces and textual interfaces into procedural-style interfaces.

[11] C# permits overriding methods that have the same signature if they appear in different interfaces.

[12] I'm sure there is someone somewhere who reads that and goes, "Wow, what a great idea!" Let me know if you are this someone, so I can avoid having to watch you eat.

Chapter 4

What Should Be in an Interface?

One of the most difficult decisions in developing a program is determining what should be in an interface. The oft-quoted guideline for object-oriented programming is that classes should be cohesive and loosely coupled. In this chapter, we'll see how these two concepts apply to interfaces.

You often face another question: how many methods should you have in an interface? Many methods can make an interface more difficult to understand but can also make it more powerful. We will explore the trade-offs in minimal to complete interfaces.

4.1 Cohesiveness

Methods in an interface should be cohesive. They should provide services that revolve around a common concept.[1] The problem is that the definition of commonness is relative. For example, a share of stock is a liability to a corporation, an asset to the owner, and something to sell for a broker. According to Flood and Carson, the United Kingdom "could be seen as an economy by economists, a society by sociologists, a threaded chunk of nature by conservationists, a tourist attraction by some Americans, a military threat by rulers of the Soviet Union, and the green, green grass of home to the more romantic of us Britons."[2]

[1]The cohesion quality predates object-oriented programming. An original reference is W.P. Stevens, G.J. Myers, and L.L. Constantine's, "Structured Design", *IBM Systems Journal*, Vol. 13, No. 2, 1974.

[2]See *Dealing with Complexity* by Robert Flood and Ewart Carons (Plenum Press, 1988).

What Sticks Together?

I had a psychology professor who gave exams that were designed to make you fail. He would give four terms and ask the question, how many of these go together? Well, it all depended on how you interpreted "go together." He collected all the exams, so I don't have an example. But let me give you one from programming. These are programming languages: Fortran, C, C++, and Java. How many of these languages relate to one another? a.) Two, b.) Three, c.) Four, d.) None

If you and your team agree on an answer, then you probably share a common approach to cohesiveness.

You can find commonness in almost anything. For example, "Why is a lightbulb like Doonesbury?" Neither one can whistle.

We're going to look at the interface to a printer to demonstrate a range of cohesiveness. Depending on your point of view, you might consider that all printer operations belong in one interface, since they are all related to printing. Or you might consider a narrower view of cohesiveness that divides the operations into multiple interfaces.[3]

4.2 A Printer Interface

You have a number of different printers in your office; each of them has different capabilities. Let's create a spreadsheet that shows the different features for each printer. You probably can think of many more capabilities, but we need to have the table fit onto a single printed page. In Figure 4.1, on the next page, a check mark indicates that a printer performs a particular operation.

Suppose you were to create your own printing subsystem. The question is, how do you place these capabilities into interfaces? Do you have a single interface or multiple ones? You need to determine which capabilities are available when printing a page. For example, if the printer has the capability to turn over the page, you want to ask the user whether

[3]For a look at eight kinds of cohesion (from Functional Cohesion to Coincidental), see `http://elearning.tvm.tcs.co.in/SDO/SDO/3_3.htm`.

Similar to the set_font_modifier() method in Chapter 3, these multiple methods could be turned into a single one, like this:[4]

```
enumeration Operation {TURN_OVER_PAGE, PRINT_PCL,...}
Boolean can_perform(Operation)
```

Your printing subsystem asks the printer whether it can perform a particular operation before calling the corresponding method:

```
printing_subsystem (Printer a_printer)
    if (a_printer.can_perform(TURN_OVER_PAGE)
        // ask user if they want duplex printing
```

A second way to organize the model/feature table is to break up the methods into multiple interfaces. Each interface consists of a related set of methods. A particular printer model implements only the interfaces for which it has capabilities. For example, the printer interfaces could be as follows:

```
interface BasicPrinter
    print_text(Position, Font, String)
    eject_page()
interface ColorPrinter
    print_image_in_color(Position, Image)
    print_image_in_black_and_white(Position, Image)
interface MonochromePrinter
    print_image_black_and_white(Position, Image)
interface DoubleSidedPrinter
    turn_over_page()
    Side which_side_are_you_on()
interface MultiTrayPrinter
    select_tray(TrayID)
    TrayID [] get_tray_ids()
interface PostscriptPrinter
    print_postscript(PostscriptString)
interface PCLPrinter
    print_pcl(PCLString)
```

How do we decide what operations to put into what interface? It's a matter of cohesiveness. If the operations (e.g., turn_over_page() and which_side_are_you_on()) will be used together, they should go into the same interface. If printers always supply operations as a set, then they should go together.

The single Printer interface collects all operations relating to printers. So, you may consider it a cohesive interface. On the other hand, each

[4]Note that in Windows you can call the capability method, GetDeviceCaps(), to ask whether a particular operation is supported. For example, GetDeviceCaps(TEXTCAPS) returns a value indicating text capabilities, such as TC_UA_ABLE (can underline).

of these specialized interfaces has methods relating only to a particular capability. So, you might think of them as more cohesive. Note that a printer does not need to have any knowledge of interfaces it cannot provide.[5] We do not have to ask a printer "can you do this for me?" for each operation. We see that a printer can do something by the fact that it implements an interface.

Before we move on, let's quickly look at how you might find a particular type of printer. A printer provides an implementation of one or more of the interfaces. For example:

```
class MySpecialPrinter implements BasicPrinter, ColorPrinter,
    MultiTrayPrinter
```

You can provide a method that lets the user find a printer that implements a particular interface. For example, they may want to find one that can print in color. So, the user codes the following:

```
my_printing_method()
    {
    ColorPrinter a_printer = (ColorPrinter)
        PrinterCollection.find(ColorPrinter)
    a_printer.print_image_in_color(position, image)
    }
```

Now what if you want to pass around just a reference to a BasicPrinter, and inside a method you wanted to use it as a ColorPrinter? You could simply cast the reference to a ColorPrinter. If it did not implement the interface, then a cast exception would be thrown:

```
a_function (BasicPrinter a_printer) throws CastException
    {
    ColorPrinter color = (ColorPrinter) a_printer
    color.print_image_in_color(position, image)
    }
```

If you really needed to find out whether the printer had more capabilities, you could ask it whether it implements a desired interface. This is the equivalent of testing for capabilities (e.g., calling can_perform() for Printer) but for a related set of capabilities.[6]

---

[5]An alternative is to have a base class, ComprehensivePrinter, that implements all interfaces but has null operations for most of the methods. Then each printer inherits from ComprehensivePrinter. We look at inheritance in Chapter 5.

[6]The code looks like *downcasting* (casting a base class to a derived class). You should usually avoid downcasting. In this example, the cast is to an interface, rather than a derived class.

```
a_function (BasicPrinter a_printer)
    {
    if (a_printer is_a ColorPrinter)
        {
        ColorPrinter color = (ColorPrinter) a_printer
        color.print_image_in_color(position, image)
        }
    }
```

However, if a_function() really required a ColorPrinter, it should be passed a reference to one, rather than having to test for it. That makes its contract explicit. The cast exception will occur when the reference is passed.

```
a_function (ColorPrinter a_printer)
    {
    a_printer.print_image_in_color(position, image)
    }
```

SINGLE PRINTER INTERFACE

Advantage—can have single capability query method

Disadvantage—related capabilities may not be logically grouped together

MULTIPLE PRINTER INTERFACES

Advantage—printer need only implement interfaces it supplies

Disadvantage—lots of interfaces

4.3 Coupling

Coupling measures how one module depends on the implementation of another module. A method that depends upon the internal implementation of a class is tightly coupled to that class. If that implementation changes, then you have to alter the method. Too much coupling—especially when it's not necessary—leads to brittle systems that are hard to extend and maintain.

But if you rely on interfaces instead, then it's difficult to tightly couple a method to another implementation. A method that just calls the methods in another interface is loosely coupled to that interface. If you simply use a reference to an interface implementation without calling any methods, then the two are considered really loosely coupled.

In the printer example, my_printing_method() is loosely coupled to ColorPrinter and PrinterCollection:

Who's Job Is It Anyway?

You probably have printed digital photographs. Printing a digital photograph brings up an interesting question: if you want to print an image in a resolution different from the printer's resolution, where should you assign the job of converting the image to a different resolution?

You have two options:

- Pass the printer driver the image, and let it perform its own resolution conversion (perhaps by calling a graphics library).
- Ask the printer for the resolution it can handle, convert the image to that resolution, and pass the converted image to the printer.

You might say, what's the difference? The result should be the same. Maybe, maybe not. This is where the quality of implementation comes into play. The program that you are using to print the image may have a much higher quality of resolution conversion than the graphics library. The developers may have done a better job in reducing conversion artifacts.

Although you may often trust the implementation of an interface to do the right thing, you may want to perform your own set of processing to ensure that you get exactly what you want. This quality of implementation issue sometimes makes it hard to determine what is the appropriate job for an interface.

```
my_printing_method()
    ColorPrinter a_printer = (ColorPrinter)
        PrinterCollection.find(ColorPrinter)
    a_printer.print_image_in_color(position, image)
```

If this method did not call a method in ColorPrinter, then it would be really loosely coupled. For example, it could simply pass the reference to another method, as:

```
my_printing_method()
    ColorPrinter a_printer = (ColorPrinter)
        PrinterCollection.find(ColorPrinter)
    print_some_color_image(a_printer, position, image);
```

Loose coupling allows you to vary the implementation of the called interface without having to change the code that calls it. On the other

hand, tight coupling forces the code to change. Here's a silly example to show tight coupling:

```
class Pizza
    wake_up_johnny
    order
```

In this example, Johnny is the implementation of the order taker. He needs to be woken up before he can take an order. If Johnny leaves and Sam takes over, Sam may be able to stay awake. The wake_up_johnny() method would go away, forcing any code that calls the method to be altered. The solution is to decouple the implementation by using an interface and hiding the implementation. For example:

```
interface Pizza
    order

class PizzaWithJohnny implements Pizza
    order
        calls wake_up_johnny
        and then performs regular order
```

TIGHT COUPLING
: Disadvantage—callers have to be changed if implementation changes

LOOSE COUPLING
: Advantage—callers do not need to be changed if implementation changes

4.4 Interface Measures

Interfaces can be subjectively measured on a number of scales. Let's look at two of these measures: minimal versus complete and simple versus complex. An interface you design or use can fall anywhere in these ranges.

Minimal versus Complete

A minimal or sufficient interface has just the methods that a caller needs to perform their work cases. A complete interface has more methods. The File interface in Chapter 3 had the following:

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

You might note that the interface does not have a skip() method. This method would allow a caller to skip over a number of bytes so that they do not have to read the intermediate bytes. A caller who needs to skip some number of bytes can simply read that many bytes and ignore them. If a caller wants to go backward, they can close the file, open it again, and start reading from the desired position.[7] The interface is sufficient for a user to perform the needed functionality.

On the other extreme, a caller might want the File interface to have additional methods, like these:

```
read_a_line()
find_a_regular_expression(expression)
```

Adding these methods makes the interface more complete, in the sense that it will have all the potential methods that a user might need. However, a more complete interface becomes more difficult to implement, because of the number of methods. An alternative is to create another interface with these methods, like this:

```
interface FileCharacterReader
    read_a_line()
    find_a_regular_expression(expression)
```

This interface would use an implementation of the minimal File interface for the read() method and add the logic to return the appropriate set of characters. Creating this interface can also help with cohesiveness. You can place methods that treat a File as a set of characters in FileCharacterReader.

MINIMAL

Advantage—easier to implement and test with fewer methods

Disadvantage—user must code their particular functionality and may wind up with duplicate code for same functionality

COMPLETE

Advantage—user has all needed methods

Disadvantage—may be harder to understand an interface with numerous methods

[7] A skip() method would probably be more efficient if it were implemented as part of the interface.

Simplicity versus Complexity

If you were making a pizza yourself, rather than ordering one, you might have a class like this:

```
class SimplePizza
    set_size(Size)
    set_topping(Topping)
    Size get_size()
    Topping [] get_toppings()
    make()
```

At the completion of the make() method, a SimplePizza is ready for your eating pleasure.

You could have a more complex interface, such as this:

```
class PizzaYourWay
    set_size(Size)
    set_topping(Topping)
    Size get_size()
    Topping [] get_toppings()
    mix_dough()
    spin()
    place_toppings(Topping [])
    bake()
    slice()
```

PizzaYourWay allows you to control the pizza-making process with more precision. You could slice() the pizza before you place_toppings() and then bake() the pizza. If you were splitting the pizza with a vegetarian, you would not get the artichoke juice mixed in with your pepperoni (or vice versa).

The implementation of each method in PizzaYourWay is simpler. However, you have made the user's job more difficult. This "slice before placing toppings" flow is now the caller's responsibility to code. They have to call five methods in the appropriate sequence in order to make a pizza.

The make() method in the SimplePizza may internally call private versions of mix_dough(), spin(), place_toppings(), bake(), and slice(). The make() method would handle any errors that these functions generated, thus simplifying the caller's code.

If you want to offer alternative flows, such as "slice before placing toppings," you could create another SimplePizza method such as make_by_slicing_before_placing_toppings(). The user simply calls the appropriate method, without having to deal with complexity. Now you are on the way to having a complete interface (see the previous section).

Simplicity versus Complexity

You always have trade-offs in life. The trade-off of "Simplicity, but Where?"* suggests you should strive for simplicity. You can make the API simpler, which will put more complexity (such as error handling) into the responsibility of the implementation, or you can make the implementation simpler, by adding complexity to the interface. This trade-off is also referred to as the "Law of Conservation of Complexity"†

*David Rock suggested this name
†Ron Thompson suggested this name

You could offer both interfaces to the outside world. The SimplePizza interface would call the appropriate methods in PizzaYourWay. In a sense, this trade-off acts as in reverse of the minimal versus complete. You create a simpler interface for a complex one.

SIMPLE

Advantage—easy for the user to perform common functions

Disadvantage—variations must be coded as new methods

COMPLEX

Advantage—users have flexibility to "do it their way"

Disadvantage—may be harder for users to understand

4.5 Things to Remember

Design cohesive interfaces. Determining what makes a cohesive interface is the hard part.

Aim for loose coupling. Using interfaces drives you there.

Measures of interfaces include the following:

- Minimal to complete
- Simple to complex

If in doubt, make an interface at one end of a measure, and use it from one made at the other end.