Versioning

Versioning of interfaces is an old problem. Microsoft Windows spawned the condition known as "DLL hell" in which incompatible versions of the same interface implementations (e.g., in dynamic link libraries) were needed by different programs on the same computer. In the .NET Framework Microsoft has solved this problem by creating versioned assemblies.

The same versioning issue occurs with JAR files. The Java Community Process is developing the Java Module System (JSR-277).* JMS "defines a distribution format and a repository for collections of Java code and related resources." One of its components is a "versioning scheme that defines how a module declares its own version as well its versioned dependencies upon other modules."

*See `http://www.jcp.org/en/jsr/detail?id=277`. David Bock, a member of the committee, suggested this example.

particular interface is implemented. Later, you bypass the directory service when getting the service.

We show in Chapter 9 a general directory mechanism for services. Directory services of frameworks such as CORBA and web services work similar to the one shown in that chapter. Similar to you using the Yellow Pages for finding a pizza place, a service consumer (e.g., a client) can use the directory service to discover the identity of service providers (e.g., a server) for a particular service. In addition, the directories can provide the communication protocol and other information needed to communicate with a service.

The consumer may be able to choose between providers (e.g., from which host to download an open source software package). Alternatively, the directory service can select one, based on some type of algorithm (round robin, who is the cheapest, etc.). This automatic selection of a server is often used when your browser requests a web server (one that implements the Hypertext Transfer Protocol [HTTP] interface) for a particular domain. The directory server returns the address of one of a number of servers, making the multiplicity of servers opaque to the client.

Published versus Unpublished

You can easily redesign interfaces used only within a program. You only have that program to worry about. Restructuring or refactoring the code to make the internal interfaces more consistent, more cohesive, and less coupled is always beneficial.

Interfaces that are used by multiple systems or development groups are more challenging. An interface used by multiple systems can be considered "published." Martin Fowler suggests delaying publishing interfaces since they cannot easily be changed: Once they're published, you have to worry about all the users of the interfaces.*

For intracorporate interfaces, an individual or committee could be responsible for the coordination of publishing interfaces, especially for interfaces that are planned as utility interfaces for the enterprise. The coordinators look at the interface from the user's side. For example, they may check to see that its style matches those of existing interfaces and the protocol is logical. Examples of enterprise interfaces are a user authentication interface for single logon and a universal logging capability for applications.

*Cf. `http://www.martinfowler.com/ieeeSoftware/published.pdf`.

6.5 More on Document Style

Since using documents as interfaces may be less familiar to many developers, we continue our exploration of some of the facets of this style.

Document Representation

You can represent documents in either external format or internal format. Common external formats are XML, EDI, and ANSI X12. In most languages, a pure data object, such as a DTO, can represent the internal format of a document. Most documents have tree-like structures, which DTOs can parallel. To keep the external representation opaque, modules (other than those that actually convert the data to and from external format) should deal with the DTO, rather than with the external form.

Suppose we decided to use XML as the external format for the PizzaOrder shown earlier in this chapter:

```
<Pizza>
    <Size>Large</Size>
    <Toppings>
        <Topping>Pepperoni</Topping>
        <Topping>Mushroom</Topping>
    </Toppings>
    <Address>
        <Street>1 Oak Street</Street>
        <City>Mytown</City>
    </Address>
</Pizza>
```

We can create an internal representation of this document:

```
class PizzaOrder
    set_size(Size)
    set_topping(Topping)
    Size get_size()
    Topping [] get_toppings()
    set_address(Address)
    Address get_address()
```

Only when we send a PizzaOrder to an external interface do we need to convert the data to and from an external from, such as XML:[10]

Validating a Document

Did what you wrote on the order make sense to the pizza parlor staff? Is the information readable? Based on an order that passes a validity check, can the pizza parlor process the order? Does it have all the necessary ingredients?

Like procedural interfaces, documents have interface contracts. There are multiple levels of validity checking for a document to ensure that it meets the interface contract: the syntax of the text itself, constraints on the values, and validity of the values.[11] Before sending a document, you should validate the data and check the constraints as thoroughly as possible to avoid needless retransmission.

[10]If the document is large and converting the entire document is resource intensive, you can deal with it in an iterative form, such as SAX (shown in Chapter 3).

[11]A procedural-style interface has fewer levels of validity checking, since the syntax is checked by the compiler and some constraints can be specified by using specific data types for method parameters.

Suppose we represent the data in an XML format. The XML data must be well-formed (e.g., the tags are correctly formed with nested begin/end tags, etc.). An XML schema describes the required and optional fields in an XML document. The organization of XML data should be valid; that is, it should conform to a particular schema.[12]

Values of each of the data items have constraints that need to be honored, such as the type of data and low and high limits. An XML schema not only can outline the structure of XML data; it can also give constraints on the data.

Some data constraints are expressible in a schema, and some are not. For example, the schema can require that a data item must be one of a set of choices or that a date must be in a particular range (the order must be placed today or in the future). However, cross-data constraints are difficult to express in a schema. For example, if you order beer in a pizza parlor, then you have to order a pizza.[13]

You might perform validity checks by using an XML tool or by transforming the document into a DTO and performing the checks in another language. If you use a DTO to represent a document, you can perform many of the validity checks in the set methods. You can also easily perform cross-value checks, as well as any other rules that are not expressible in a schema.

You can perform many data checks on the client computer. For example, the client can check a CustomerID for the number and types of characters. There are other checks that only the server can perform. For example, is a CustomerID one that is registered in the system? Some other checks might be difficult for both systems (the client or the server) to perform, such as whether the sender of a document legally represents a particular customer.

Document Encoding

Although XML is a very adaptable and general form of encoding documents, it is not the only one. Depending on the environment in which the document is used, you might find other encodings more efficient.

[12]Creating a valid, well-formed XML document is a relatively simple task with modern tools. (See `http://www.w3.org/TR/xmlschema-0/` for more information about schemas.)

[13]This is a law in at least one town in Massachusetts.

Here are several alternative text encodings, along with examples of where they are employed:

- Straight text (text with line breaks is a lingua franca). Examples:
  - Email messages
  - Pipes and filters of Unix
- Text with field delimiters. Examples:
  - Email headers
  - Comma or tab delimited files
- Text with named tokens. Examples:
  - Property files[14]
  - HTTP form parameters
- Structured text, e.g., XML and YAML.[15] Examples:
  - Web services
  - Configuration files
- Electronic Document Interchange (EDI) standards. Examples:
  - Airline reservation systems
  - Electronic orders[16]

Your application may require you to use a particular format for communication. If not, select the simplest representation. XML is a good choice for complex documents because of the number of tools available for manipulating that format. However, if all you require is a few values, using named tokens can be simpler.

Document-Based Business Flow

Similar to protocols for calling methods for a procedure-style interface, there are protocols for document-style interfaces. You can't send a series of documents in just any order. The order must correspond to a business flow.

[14]The old-style Microsoft Windows `.ini` files are an example of a property file.
[15]See `www.yaml.org`.
[16]See "Using Standard Documents" in the appendix.

Let's look at something that you may often do on the Web and see how this flow is represented as an exchange of documents.[17] These documents correspond to data sent from forms in your browser and data returned on the displayed web pages. The encoding in one direction is named tokens, and the encoding in the other direction is structured text (Hypertext Markup Language [HTML]).

You go to Orbitz, Travelocity, or another travel site. You enter a request for a one-way airfare from your home to New York City and click Search. The transmitted information can be considered a *request for quotation* document. The web browser transmits this information in the form of tokens in the HTTP request:

```
Document: RequestForQuotationForOneWayAirfare
    Origin
    Destination
    Date
    Number Travelers
```

The response to this document is another document:[18]

```
Document: QuotationForOneWayAirfare
    Selections []
        FareAllInclusive
        Legs []
            Flight Number
            Origin
            Destination
            DateTime Departure
            DateTime Arrival
```

Once you have made your choice of the selections, you send back another document:

```
Document: ReserveOneWayAirfare
    Passenger
    FareAllInclusive
    Legs []
        Flight
        Number
        Origin
        Destination
        DateTime Departure
        DateTime Arrival
```

[17]To see another example, check out the Oasis Universal Business Language (UBL) 1.0, published September 15, 2004. (ID: cd-UBL-1.0). You can find a copy at `http://docs.oasis-open.org/ubl/cd-UBL-1.0/`.

[18]And perhaps a call from the Department of Homeland Security; one-way flights seem to get them a little skittish.

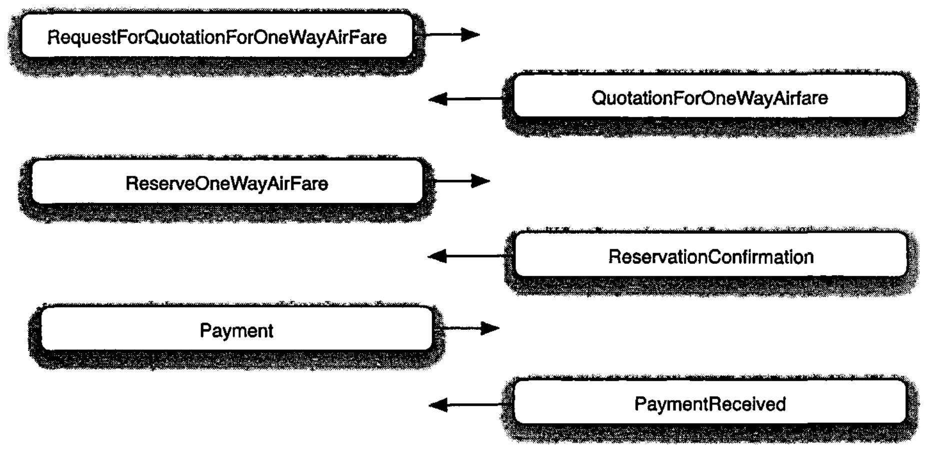

Figure 6.1: DOCUMENT FLOW

```
     // Put up GUI with selection
ReserveOneWayAirfare reserve;
     // When user has selected an item, fill in reserve
     // and send it
ReservationConfirmation confirmation = send_message(reserve)
Payment payment;
    // Fill in payment with amount and other information
PaymentReceived payment_received = send_message(payment)
```

6.6 Security

Whenever you have a remote interface that is available to the outside world, you need to worry about security. A detailed examination of security topics is beyond the scope of this book, so we'll just discuss some general matters.[19] Frameworks such as web services and CORBA provide some solutions to general security issues such as authentication and confidentiality. However, you should examine each remote interface to ensure that it is secure.

You need to authenticate the user of your interface. Authentication determines, to a reasonable degree of certainty, that the user is really

[19]See *Software Security: Building Security In* by Gary McGraw (Addison-Wesley Professional, 2006) for a full discussion of security.

the user. Typical authentication schemes range from passwords to encrypted certificates to biometric scans.

Once you know who the user is, you should check to see that the user is authorized to access the interface or perform an operation. Typically, authorization is performed using an access control list (ACL) or similar mechanism. The ACL lists the users who are authorized to perform a particular operation. To simplify administration, users are often assigned roles, such as Customer or PizzaOrderer. The ACL associates roles with particular operations or set of operations.

When data is transmitted over a network, you particularly need to be concerned with confidentiality and data integrity. Encrypting data can provide a degree of confidentiality. It can help prevent the data from being altered as it traverses the network.

Even if you have mechanisms for providing all the previously mentioned security aspects, you need to worry about information escaping. Suppose you work for a hospital and you use a remote provider to check the ZIP codes for addresses. If those addresses are your patients' addresses, you just provided private patient data to an unauthorized party.

You should be particularly concerned with contract checking. Any document-style interface should completely check the validity of the received documents. Implementations of procedural-style interfaces should employ strong contract checking, since the implementations cannot be assured that the client has followed the contract.

6.7 Testing

In general, every external interface should have a testing implementation (e.g., a stub or a mock object). The testing implementation can respond quicker than a remote implementation. The quick response improves the speed of testing other units that depend on the remote interface. The testing implementation can have additional methods (a testing interface) that allow the user to set up a desired request/response scenario (e.g., a request document and a response document). It can also provide a way to create a failure condition (e.g., server unavailable) to see how the unit under test handles those conditions. Simulating these failures in software for rapid testing is a lot easier than having to disconnect the network.

6.8 Things to Remember

We covered both procedural-style and document-style remote interfaces. When using or designing a remote interface, consider how it may react to network delay or failure and make provisions for handling those situations. For a document-style interface, follow these tips:

- Precisely specify the document flow protocol.
- Perform validity checking before transmitting a document.
- Use an appropriate document encoding schema.

We discussed a few matters that you should consider when you employ a remote interface:

- Use a DTO or a data interface to make the external representation of a document more opaque.
- Examine the security implications of any remote interface.

Chapter 7

A Little Process

You probably already have a software development process. You may be using the Rational Unified Process (RUP) or Extreme Programming (XP) or something in between. This chapter shows how interface-oriented design can fit into your process; we'll outline the development phases for a system with a concentration on interfaces. In the next three chapters, we will create three types of systems to show interfaces at work in other contexts.

Creating interfaces may seem contrary to a common measure of simplicity, which is having the fewest lines of code. Using interfaces can add code. However, designing with interfaces focuses on creating a contract for an interface and testing that contract. It can prevent you from getting mired in implementation details too early. Even if you decide not to have the interface layer in code, thinking in interfaces helps keep you focused on the real problems. Since refactoring consists of changing code without changing its external interface, a well-designed interface can make it easier to refactor.

7.1 The Agile Model

Since I do agile development in my work, I present interface-oriented design in that context. We are not going to cover the details of agile development methodologies. You can read these details in various books on agile processes.[1]

[1]Books include *Extreme Programming Explained: Embrace Change* by Kent Beck and Cynthia Andres (Addison-Wesley, 2004), or *Agile Software Development with SCRUM* by Ken Schwaber and Mike Beedle (Prentice Hall, 2001).

In agile development, you create your software in iterations. During each iteration, you create the highest-priority remaining features desired by the customer. This chapter shows discrete tasks of development—vision, conceptualization, testing, analysis, design, and implementation. During an iteration, you may go through all these tasks to develop a feature.

I outline specific development tasks to be able to separate the discussion of the various portions of the process; no absolute milestones signal the end of one phase and the beginning of another. And these phases aren't followed linearly—feedback exists among them. You may discover during implementation that an ambiguity existed in conceptualization. You would go briefly back to conceptualizing, disambiguate the concept, and then continue implementing.

7.2 Vision

Every software project needs a vision. The vision is a sentence or two that outlines the business purpose for the software. In this chapter as an example, we are going to create a pizza shop automator. The vision is that this system decreases the amount of work necessary to process pizza orders.

7.3 Conceptualization

Once you've established a vision, you need to capture requirements for the software system. Functional requirements can be expressed in many ways including formal requirement documents, user stories, and use cases. I have found that use cases are easy for both the end user and the developer to understand. They quickly demonstrate what the system does for the user, as well as show what is inside the system and outside the system. They form the basis for acceptance tests as well as for internal interface tests. If a system cannot perform what the use cases state, then the system has failed.[2]

We create the use cases together with the person requesting the system and the end users. For the pizza shop automator, the actual making

[2]Passing tests for the use cases is necessary for system success but not sufficient. The system must also pass nonfunctional tests (e.g., performance, robustness).

Joe Asks...

Use Cases

A use case* "defines the interactions between external actors and the system under consideration to accomplish a business goal. Actors are parties outside the system that interact with the system; an actor can be a class of users, a role that users can play, or another system."

Use cases treat the system as a "black box." Interactions with the system, including system responses, are documented from outside the system. This is deliberate policy, because it simplifies the description of requirements and avoids the trap of making assumptions about how this functionality will be accomplished.

Note that this definition parallels how interfaces should be treated.

Each use case has a name, shown in the use case diagram. For each use case, you write down the individual steps that list what the user does and how the system responds. A simple format is as follows:†

U Use Case Title

1. User asks.
2. System responds

*This definition comes from `http://en.wikipedia.org/wiki/Use_cases`.

†You can add many more features to use cases. See *Writing Effective Use Cases* by Alistair Cockburn (Addison-Wesley Professional, 2000) for details.

the pizza and delivering the pizza are outside of the software system.[3] For this system, the main concern is the information flow between the various actors.

We first identify the actors by the roles that they play, not their specific

[3]If we were creating an automated pizza shop, then the making of the pizza would be inside the system.

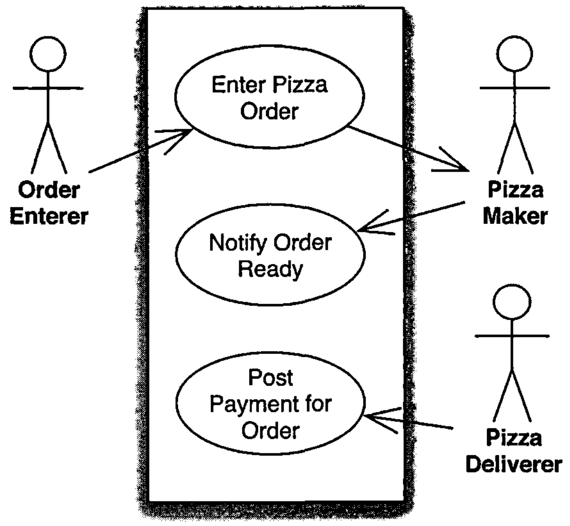

Figure 7.1: USE CASES

positions. An OrderEnterer could be the person on the other end of the phone, or it could be a customer on a web site. A PizzaDeliverer could be a guy in a car, or it could be someone behind the counter who delivers the pizza to a walk-in customer. We next determine what the actors will use the system for and list these in a use case diagram (Figure 7.1).

For each use case, we list the interactions between the user and the system. We try to write the interactions in a more general way (e.g., select the size of pizza), rather than in a specific way (e.g., choose the size of pizza from a drop-down list).[4] We want to explore the functionality of the user interface, rather than a specific implementation at this point. Here are the details for each use case:

Use Case: Enter Pizza Order

1. Order Enterer enters order details.
2. System responds with time period till order is to be delivered.

[4]You can create GUI prototypes that use specific widgets, if the user needs to envision how the system will appear to the end user in order to understand it.

Use Case: Notify Order Ready

1. Pizza Maker notifies system that order is complete.
2. System notifies Pizza Deliverer that order is ready for delivery.

Use Case: Post Payment for Order

1. Pizza Deliverer enters payment received from customer.
2. System records order as paid.
3. Pizza Deliverer puts remainder of money into his pocket as tip.

Each use case represents a portion of the user's interface to the system. Like the code-based interfaces we have been discussing, the user's interface has a contract and a protocol. We can also document the preconditions and postconditions for each use case. For example, "Notify Order Ready" has a precondition that "Enter Pizza Order" has occurred.

Before moving onward, it's a good idea to write down some of the assumptions we are making for this system. We simplified the flow to concentrate on the process. We know that one pizza is just not enough for a hungry customer. Our assumptions include the following:

- An order is for one pizza and no other menu items. Otherwise, we need to handle partial orders (e.g., one pizza is ready, and another is not).
- The system will handle only cash payments for pizza.
- Pizzas vary only by size and number of toppings.

Testing

It may seem odd to mention testing before even starting to design the system, but outlining tests can provide additional insights in understanding a system. Often, you can find ways to structure your interfaces to make them easier to test. On the user level, we examine the use cases to create an outline of the acceptance tests. Here are some tests that we generated from the use cases:

Test Case: Normal Pizza Order

1. Order Enterer enters Pizza order.
2. Order Enterer should see time to deliver.

3. Pizza Maker should see order.
4. Pizza Maker notifies order is complete.
5. Pizza Deliverer should see delivery notice.
6. Pizza Deliverer enters payment from customer.
7. System should show order is paid.

Test Case: Normal Pizza Orders

1. Repeat Normal Pizza Order several times.
2. System should show all orders are paid for.

Test Case: Random Pizza Payments

1. Like Normal Pizza Orders, but Pizza Deliverer pays for orders in random sequence.
2. System should show all orders are paid for.

These tests suggest that we should have a reporting mechanism that lists orders and whether they have been paid. We can use that reporting mechanism to determine the success of these tests. This reporting mechanism suggests that we didn't capture a use case involving reports.

You can also generate "misuse" cases, which describe how the system might be accidentally or deliberately misused. Misuse cases can include the "fringe" cases in which you try entering values that are on the edge of an acceptable range. Here are a few misuse cases:

Test Case: Pay for Paid Order

1. Pizza Deliverer pays for order that has already been paid.
2. System should not allow this.

Test Case: Overburden Pizza Maker

1. Order Enterer enters numerous orders for same address.[5]

[5]The same address is used in order to differentiate the response from that due to delivery to different addresses.

2. System should respond with increasing time to deliver.

Test Case: Send Pizza Deliverer on Goose Chase

1. Order Enterer enters order for faraway place.
2. System should respond with what?

The third test suggests that we have not yet captured some requirements, since we are not sure how the system should respond. Our users inform us that a pizza shop has a limited delivery area to ensure that the delivered pizza is hot. You should not accept an order if the delivery address is outside that delivery area. The idea of addresses inspires a few more misuse cases:

Test Case: Place Order to Nonexistent Address

1. Order Enterer places order to nonexistent address.
2. System should respond that it cannot deliver to that address.

Test Case: Make Them Run All Over the Place

1. Order Enterer places orders to address that are widely separated and timed so that they cannot all be delivered together.
2. System should respond that delivery time will be excessive.

These cases revolve around a common theme: the system should be able to determine the validity of an address and to determine the delivery time for a particular address. We already know of some implementations that perform these operations (e.g., `maps.google.com` and `mapquest.com`). When we get around to creating this feature, we'll create an interface that makes transparent which implementation we are using.

We can come up with other misuse cases that test how the system responds to large demands:

Test Case: Make Them Run Out of Ingredients

1. Order Enterer orders many, many pizzas.
2. System at some point should respond that it cannot handle any more.

This "Denial of Pizza" attack is the equivalent of a "Denial of Service" network attack. Keep pounding at something to see whether it gives up. The first release may not be able to pass this test, but we'll keep it on the list for future releases.

We could make up many more misuse cases, but you should get the point by now. Each of these misuse cases helps us define the contract for the user interface—how the system should respond for each operation that the user requests.

7.4 Analysis and Design

The real boundary between analysis and design is not precise, but you can separate them with formal definitions. *Analysis* is the process of identifying the essential concepts (abstractions) in a system and determining the work that needs to be performed. *Design* develops those abstractions into realizable components. In implementation, you write the code for those components. In reality, you may develop some concepts and implement them and then realize that you did not completely understand the concept.

The term *design* is often applied to both analysis and design. The title of this book is *Interface-Oriented Design* (IOD). The ideas of IOD appear both in developing the abstractions and in turning those abstractions into components.

7.5 Interface-Oriented Design

Interface-oriented design encompasses ideas found in other design philosophies as responsibility-driven design, role-based design, and test-first development.

Interface-oriented design revolves around these concepts:

- An interface represents a set of responsibilities.
- A responsibility is a service that the interface provides.
- Modules implement interfaces. A module can be a class, a component, a remote service, or even a human being performing operations manually.
- A role represents one or more related interfaces.

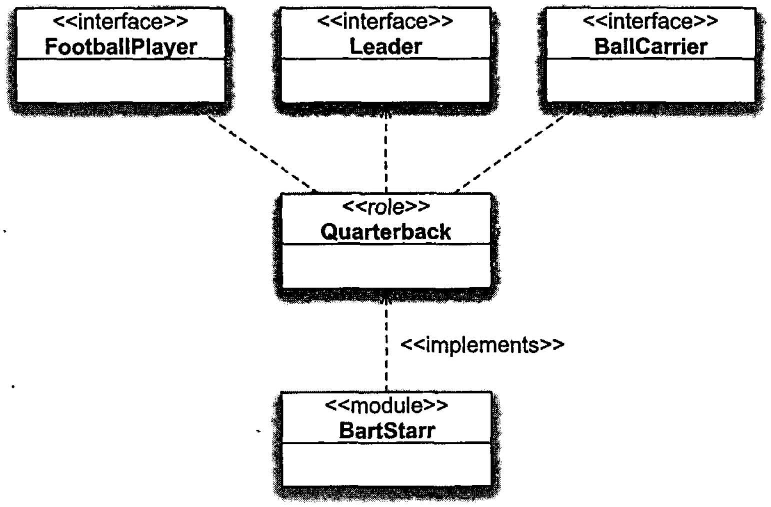

Figure 7.2: INTERFACES, ROLES, AND MODULES

- A module can implement one or more interfaces so that it can play a role or multiple roles.

The diagram in Figure 7.2, shows the relationship between interfaces, roles, and implementations. The interfaces and roles are the ones described in Chapter 5. Bart Starr is an implementation of the Quarterback role.[6]

In analysis, interface responsibilities are expressed in general terms. In design, these responsibilities are usually expressed as methods. Some responsibilities assigned to an interface may not be exposed as a method, but simply end up as internal operations. In analysis, you can come up with a rough draft of the interfaces for a system. You then assign responsibilities to those interfaces. Alternatively, you can start by grouping responsibilities into sets (roles) and then assigning names to those roles.

Once you've come up with a draft of the interfaces and their responsibilities, you work your way through the use cases to see whether they can be performed with those interfaces.

[6]Bart Starr was the quarterback for the Green Bay Packers and MVP of Super Bowls I and II.

How do you determine which interface is assigned which responsibility? Here are a few general guidelines:

- Put together cohesive interfaces. The responsibilities should be ones that seem like they go together.
- Decouple interfaces—separate responsibilities that may be implemented differently.
- Divide into more interfaces to simplify the testing of each interface.[7]

Rebecca Wirfs-Brock in *Responsibility Driven Design*[8] suggests a number of stereotypes for objects. A stereotype describes the general category of use for an object. Those stereotypes have a correspondence in interface-oriented design. We already listed two (data interface and service interface) in Chapter 2. Other stereotypes include the following:

- Storage interfaces (to hold persistent data)
- Entity interfaces that reflect models and business rules
- Interfaces to outside world
  - Document interfaces
  - View/controller GUI interface

No guaranteed way exists to determine what responsibilities should go with which interfaces; designing interfaces takes the same effort as designing classes. Bertrand Meyer says, "Finding classes is the central decision in building an object-oriented software system; as in any creative discipline, making such decisions right takes talent and experience, not to mention luck." He goes on to say, "No book advice can replace your know-how and ingenuity." The same can be said for finding the right set of cohesive interfaces.

IRI Cards

For analysis, I like to use what I call *IRI cards*, which are a variation of the CRC cards of Ward Cunningham, Kent Beck, and Rebecca Wirfs-Brock. CRC stands for class-responsibility-collaboration. IRI stands

[7]If you find yourself winding up with most interfaces having a single method, then you are probably breaking up cohesive responsibilities.

[8]See *Object Design: Roles, Responsibilities, and Collaborations* by Rebecca Wirfs-Brock and Alan McKean (Addison-Wesley Professional, 2002).

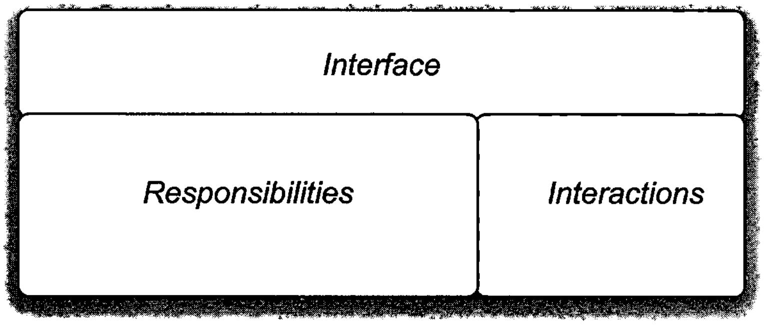

Figure 7.3: IRI CARD TEMPLATE

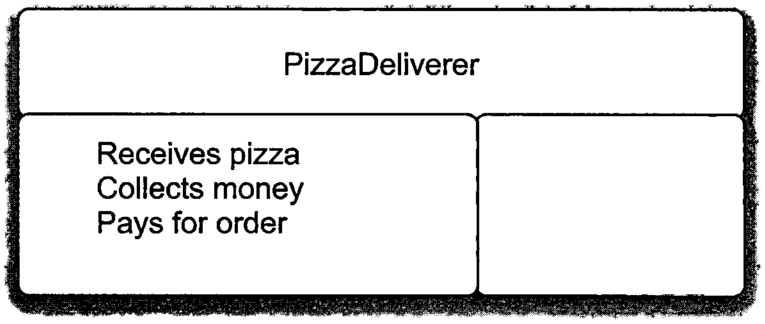

Figure 7.4: IRI CARD EXAMPLE

for interface-responsibility-interaction. A IRI template is shown in Figure 7.3. The main difference is that emphasizing interfaces makes the cards more useful in designing both class- and service-oriented designs, and it removes some emphasis on implementation issues, such as classes.

On an index card, write down an interface name and a set of responsibilities. Continue until all responsibilities are assigned to an interface.

If the responsibilities are complex, break them down into simpler ones.