Typically these simpler responsibilities will be offered by other interfaces. Create another card with names for those additional interfaces and the responsibilities. Those additional interfaces are listed as interactions on the interface card that needs them.

It's not always necessary to go from the top down (starting from the exterior responsibilities that are large and going inward to the most detailed interface). You can start at either place. Starting at the bottom may point out more reusable interfaces.

Examining our use cases, we come up with a preliminary set of interfaces. We could show these on index cards, such as the example shown in Figure 7.4, on the page before, but that would just take up more room. I tend to create more interfaces in the beginning. I have found it's easier to combine responsibilities from two or more cohesive interfaces and then separate an interface's responsibilities into multiple interfaces.

- Interface Pizza
  - Keep track of size and kinds of toppings
- Interface Address
  - Keep track of street, city, ZIP, phone number
  - Determine distance to another address
- Interface Order
  - Contains Pizza and Address
- Interface OrderEnterer
  - Enter pizza order
  - Display time to deliver
- Interface PizzaMaker
  - Display pizza order to create
  - Notify when order is ready
- Interface PizzaDeliverer
  - Receives pizza
  - Collects money
  - Pays for order

Now that we have a preliminary set of interfaces, we start with the use cases and see whether we have captured all the responsibilities. Let's take the first use case, which we restate with the interfaces we have created:

Use Case: Normal Pizza Order

1. OrderEnterer enters Pizza order.
2. OrderEnterer should see time to deliver.

Between the OrderEnterer, Order, and Pizza, we think that we have captured all the responsibilities for this use case, but we need to check. So we trace how the system determines the time till an order is ready. The OrderEnterer asks the PizzaMaker how long it will take to complete the order. Then the OrderEnterer asks the PizzaDeliverer how long it will take to deliver the pizza, assuming it is ready when the PizzaMaker says it will be. Then the system can respond with this time period. With this flow, we come up with additional responsibilities for the PizzaMaker and the PizzaDeliverer:

- Interface PizzaMaker
  - Determine how long it will take to make a pizza.
- Interface PizzaDeliverer
  - Determine how long it will take to deliver a pizza, assuming it will be ready at a particular time.

OrderEnterer now has interactions with these two interfaces, which we show on the card in Figure 7.5, on the following page.

We could have the PizzaMaker report the time to delivery by asking the PizzaDeliverer for the delivery time. However, that would tie PizzaMaker to PizzaDeliverer. With that coupling, testing will become more complicated.

We can also run through the misuse test cases to see whether we need to assign additional responsibilities. The Place Order to Nonexistent Address misuse case comes to mind. The PizzaDeliverer needs to notify the OrderEnterer if it cannot deliver the pizza because the address does not exist (in compliance with the Third Law of Interfaces). At this point we don't specify the means of notification. When we start designing and coding, we can determine how PizzaDeliverer should report this condition.

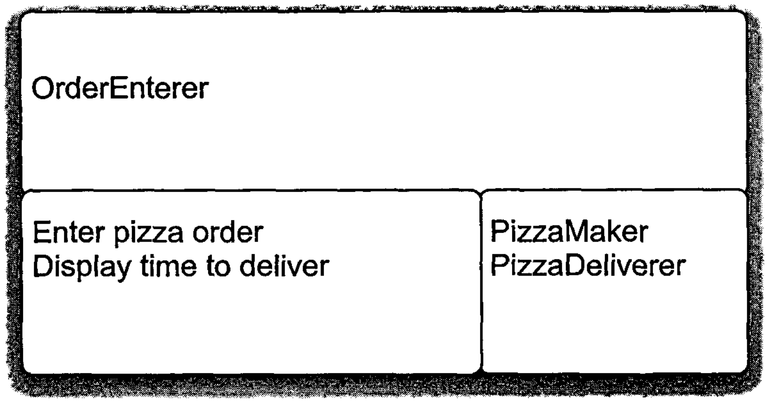


Figure 7.5: IRI CARD WITH INTERACTIONS

7.6 Design

Once we are somewhat satisfied that we have captured the essential concepts and that we can perform the use cases with these abstractions, we can turn these IRI cards into more detailed interfaces. You can write the interface declarations directly in your language of choice, or you can put the interface definitions into some design tool and let it generate the initial code. Many of the responsibilities show up as methods in the corresponding interfaces:

```
enumeration Topping {MUSHROOMS, PEPPERONI, ...};
enumeration Size {SMALL, MEDIUM, LARGE};
interface Pizza
    Size size
    Topping [] toppings
interface Address
    String street
    String city
    String zip
    String phone_number
    DistanceInMiles distance_to_address(Address another)
interface Order
    Pizza
    Address
interface OrderEnterer
    TimePeriod enter_order(Order) signals AddressOutOfDeliveryArea
    place_order(Order)
interface PizzaMaker
    TimePeriod time_to_be_ready(Order)
```

Configuration Is Yet Another Interface

Most programs are not completely static. The end user should be able to change some facets of a system without involving the developer. Although for simplicity we showed Size and Toppings as enumerations in the previous interface definitions, most likely these will need to change. These changes can be handled by a configuration interface. For example:

```
interface Configuration
    Size [] get_pizza_sizes()
    Topping [] get_pizza_toppings()
```

Configuration information may be kept in a file, a database, or an application registry. The Configuration interface hides its actual representation. Alternative configurations can be used for testing. These alternatives may be handled by either replacing the entire representation for a configuration or by adding set methods to the Configuration interface.*

The appendix discusses configuration in more detail.

*This configuration works as a Service Locator pattern. You might also use the Dependency Injection pattern for configuration. See `http://www.martinfowler.com/articles/injection.html` for a discussion on the differences. Fowler appears to favor the Service Locator pattern for application components and Dependency Injection for components that are distributed in general libraries.

```
    Order get_next_order()
interface PizzaDeliverer
    notify_order_ready(Order)
    pay_for_order(Order)
    TimePeriod time_to_deliver(Order, TimePeriod when_ready)
```

The previous definitions are language-independent. You could write the definitions directly in your implementation language. You could create a separate interface definition and then create classes or modules that implement that interface, or you could simply use the interface as the basis for a class. For data interfaces, the latter is the simplest approach. For service interfaces, the former is more flexible, especially for testing, as we shall see.

Getting Something Working

You usually won't be able to take every consideration into account when defining interfaces. You should validate your initial interface design by creating an implementation with the basic responsibilities. This implementation corresponds to the Pragmatic Programmer's *tracer bullets** Following that guideline, you "get something working": an end-to-end implementation of a system. That implementation can help verify that you made a good cut at creating an initial set of interfaces. You can then tweak your design by creating new interfaces, adding methods, altering method parameters, and so forth.

*See the original in *The Pragmatic Programmer: From Journeyman to Master* by Andy Hunt and Dave Thomas (Addison-Wesley Professional, 1999)

Interfaces Outside the Software System

Interfaces and services also apply outside the software realm. We know we need a nonsoftware implementation of a PhysicalPizzaMaker whose job is to actually create the physical pizza. An implementation of this interface may delegate responsibilities to other interfaces, such as the DoughMaker, the DoughThrower, the PizzaCreator, and the PizzaBaker. A real person might implement just the PhysicalPizzaMaker or might implement the other interfaces as well. With all these interfaces that have a smaller set of responsibilities, the implementation for each can be assigned to a different individual. Each individual interface implementation can be independently tested.

An implementation of the PhysicalPizzaMaker might be an automated system. Such a system could make the dough, throw the dough, place toppings on the pizza, put the pizza in the oven, remove the pizza, and put it in a box. When we create tests against the PhysicalPizzaMaker interface, we simply want to ensure that an implementation meets its obligations. The tests should not be dependent on whether humans or machines are making the pizza. You just keep running the tests until the results meet your expectations. This might be fattening.

Testing

Given the explicit methods of the interfaces, we can turn the outline of the tests we have developed directly into code; we'll develop the tests for the interface prior to creating the implementation. If we find that the tests are difficult to create, that is usually a sign that the interface may not be optimal. For example, here's a portion of the test for a Normal Pizza Order:[9]

```
Pizza pizza = new Pizza()
pizza.set_size(SMALL);
Topping [] toppings = {MUSHROOMS};
pizza.set_toppings(toppings);

Address address = new Address();
address.set_street("1 Oak Lane");
address.set_city("Durham");
address.set_zip("27701");
address.set_phone_number("919-555-1212");

Order order = new Order();
order.set_pizza(pizza)
order.set_address(address);

OrderEnterer order_enterer = OrderEntererFactory.get_instance();
TimePeriod time_period = order_enterer.enter_order(order);
AssertNotNull("Time period for order", time_period);
```

When we test this program, it'll be hard to have cooks keep pushing a button saying that a pizza is done. So we'll need to write simulators: implementations that simulate the operations contracted by an interface. For example, here's the interface for PizzaMaker again:

```
interface PizzaMaker
    TimePeriod time_to_be_ready(Order)
    place_order(Order)
    Order get_next_order()
```

A simulator for PizzaMaker would accept an order (place_order()), wait some period of time, and then invoke notify_order_ready() on the PizzaDeliverer interface. A more elaborate simulator might wait an amount of time based on the number of pizzas currently on order. For fast testing purposes, the simulator can be set to respond as quickly as possible.

[9]This test code uses the interface methods to set values since we're dealing with interfaces. All these set methods suggest that you might add constructors to the Pizza, Address, and Order classes. The constructors will decrease the code for the test.

Developer testing would be stifled if it took 20 minutes for a response to occur.

7.7 Implementation

As we get into implementing a particular interface, more interfaces may be created. For example, to determine how long till an order is complete, the PizzaMaker will need to keep some queue of Orders in process. The PizzaMaker uses the number of Orders in the queue to determine the amount of time before an order can be started. So in a lower level, we may have an OrderQueue interface. We will create tests for that interface that check that it performs according to its contract.

7.8 Things to Remember

In interface-oriented design, the emphasis is on designing a solution with interfaces. When using IOD, here are some tips:

- Use IRI cards to assign responsibilities to interfaces.
- Keep service interface definitions in code separate from the code for classes that implement it.
- Write tests first to determine the usability of an interface.
- Write tests against the contract for the interface.

Chapter 8

Link Checker

This chapter and the next two present three interface-oriented designs. Each design emphasizes different aspects of interfaces:

- The Link Checker demonstrates using an interface to hide multiple implementations that parse a web page.
- The Web Conglomerator shows how to delegate work among multiple implementations.
- The Service Registry presents a document-style interface and demonstrates some issues with documents.

Having broken links on your web site can annoy your visitors. I'm sure I have had several broken links on mine; the Net is an ever-changing place. A link checker that ensures all links are working is a valuable tool to keep your visitors happy. In this chapter, we'll create a link checker, and along the way, we'll see how designing with interfaces allows for a variety of easily testable implementations.

8.1 Vision

The vision for this system is short and sweet. The link checker examines links in the pages of a web site to see whether they refer to active pages. It identifies links that are broken.

8.2 Conceptualization

It's always a good idea to try to get definitions straight at the beginning. We consider the two types of links and one variation. The user is going to specify a domain, as "www.pughkilleen.com," or a URL that includes

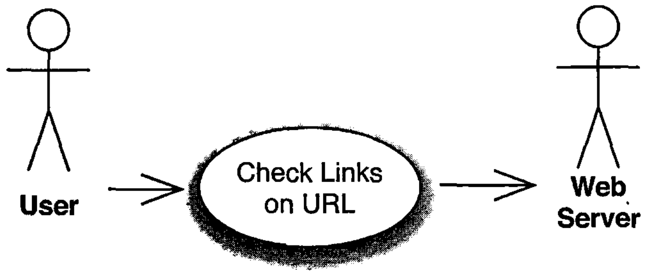


Figure 8.1: USE CASE FOR LINK CHECKER

a domain, such as "www.pughkilleen.com/classes.html." An *internal link* is a link to a page with the same domain as the specified one. An *external link* has a different domain. A variation on a link is one with an anchor. An *anchor* is a specific location within a web page, denoted by a label following a #, such as "www.pughkilleen.com/classes.html#Java." We should examine the referenced web page to see whether the anchor exists in that page. To keep the first iteration short, we will save that aspect to the next iteration.

The single use case is as follows:

Use Case: Check Links on URL

1. User enters a URL.
2. The system reports all broken internal and external links on all pages in the domain that can be reached from the URL.

Even with one use case, a use case diagram such as Figure 8.1 is often a nice way to depict what interactions a system has with outside actors.

Let's describe in more detail the work that the system will perform in response to the entered URL.

Use Case: Check Links on URL

1. User enters a URL.
2. The system determines the domain from the URL.

3. The system retrieves the page for the URL from the appropriate web server.

4. The system examines the retrieved page for internal and external links:

    a) For internal links, the system recursively retrieves the page for each link and examines that page for links.

    b) If a link is broken, the system reports it.

    c) For external links, the system just retrieves the page to see whether it is accessible.

5. The system stops when all internal links and external links have been examined.

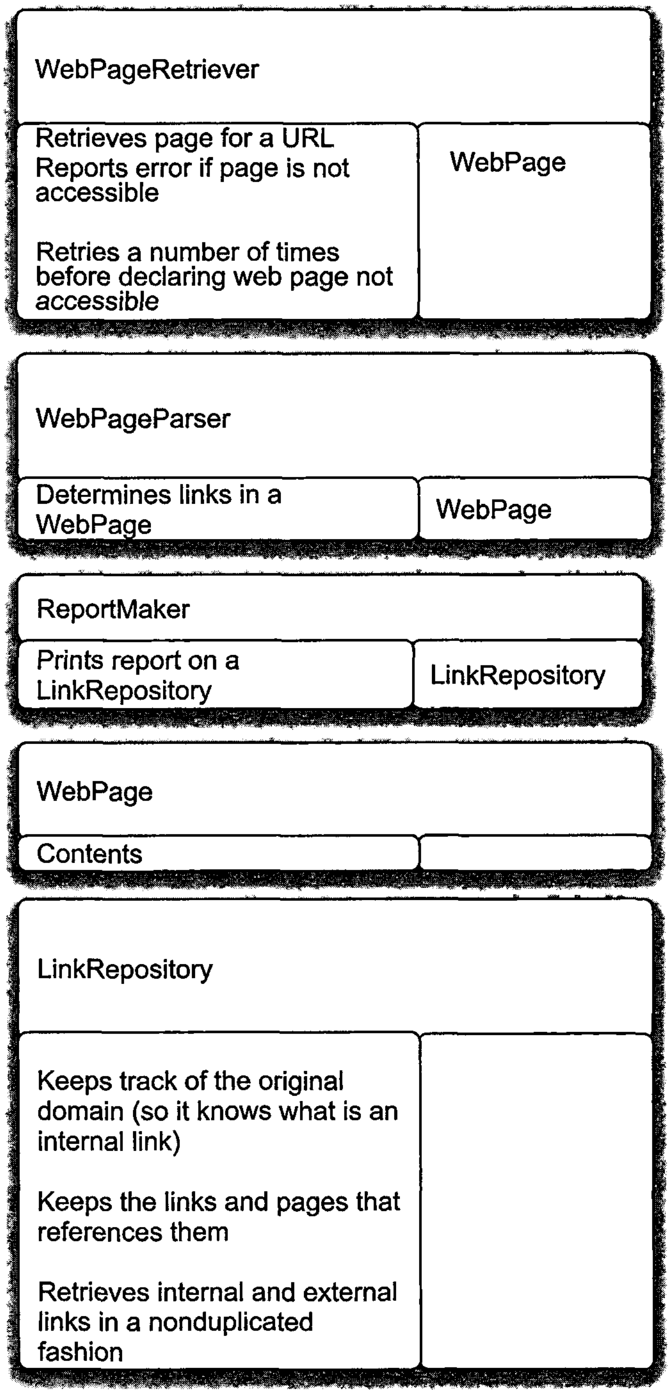
Since this GUI is really basic, a prototype report can help developers and users visualize the results of the use case. We present an outline of a report here:

Prototype Report

```
Domain:  a_domain.com
    Page: a_domain.com/index.html
        Internal Broken Link(s)
            Page: whatsup.html
            Page: notmuch.html
        External Broken Link(s):
            Page: www.somewhere-else.com/nothingdoing.html
            Page: www.somewhere-else.com/not_here.html
```

8.3 Analysis

Based on the conceptualization, we come up with a number of responsibilities and assign them to interfaces using IRI cards. We follow the guideline from Chapter 7 to decouple interfaces that may be implemented differently. We know we need to retrieve pages, so we create a WebPageRetriever interface that returns WebPages. We need to parse a WebPage into links, so we add a WebPageParser. We include a LinkRepository to keep track of the links. The IRI cards we come up with appear in Figure 8.2, on the following page. Each of the interfaces has clearly defined responsibilities.



WebPageRetriever	
Retrieves page for a URL Reports error if page is not accessible Retries a number of times before declaring web page not *accessible*	WebPage

WebPageParser	
Determines links in a WebPage	WebPage

ReportMaker	
Prints report on a LinkRepository	LinkRepository

WebPage	
Contents	

LinkRepository	
Keeps track of the original domain (so it knows what is an internal link) Keeps the links and pages that references them Retrieves internal and external links in a nonduplicated fashion	

Figure 8.2: IRI CARDS

8.4 Design

We take the interfaces on the IRI cards and develop them into more specific methods.

The Web Page

WebPage is just a data interface:

```
interface WebPage
    set_url(URL)
    set_contents(String)
    String get_contents().
```

Parsing the Web Page

WebPageParser has a single method:

```
interface WebPageParser
    URL [] parse_for_URLs(WebPage)
```

At this point, we're not sure how we are going to parse a web page into links. We could use a regular expression parser. We could use SAX or DOM (Chapter 3), if the web pages are well-formed. Or we could use javax.swing.text.html.parser.Parser, which parses most web pages. Having this interface allows us to easily test whatever implementation we decide to use. There is not much of a contract to enforce (Chapter 2). The contractual tests consist of passing web pages with a variety of content and checking that all the links are returned.

Using this interface decouples the implementation from the tests. If we create a second implementation, we can use the same functional tests. If we want to compare the two implementations for speed or ability to handle poorly formed input, we write the tests against this interface.

Having the interface makes selecting an implementation less critical. We pick one. If it's too slow, we pick another. The code that requires the parsing does not need to be changed.

The WebPageParser returns an array of URLs.[1] This URL interface contains mostly data:

```
data interface URL
    protocol (e.g., http://)
```

[1]If you're familiar with Java, you may recall that the Java library has a URL class.

Multiple Implementations

Creating multiple implementations of the same interface is often employed in high-reliability software. For example, three teams each code an airplane guidance module. Each team uses a different algorithm; another module compares the results of the three. If they all agree, the comparison module uses that value. If fewer than three agree, the module signals a problem to the pilot. If only two agree on a value, the comparison module uses that value. If none of them agree, the comparison module has to make a decision. It might default to using the one module that agreed most in the past with the other two modules.

```
    domain (e.g., www.pughkilleen.com)
    port (optional, e.g., :8080)
    file (e.g., /index.html)
    anchor (optional, comes after '#')
    additional (optional, comes after '?')
    to_string() // returns string of URL
    from_string(String) // parses string into URL
```

Retrieving the Web Page

The WebPageRetriever retrieves the WebPage corresponding to a particular URL. We don't want to report that a link is bad if there is just a temporary failure in the Internet. So, WebPageRetriever could signal an error if it cannot locate the URL in a reasonable number of tries, rather than in a single try. It has a single method:

```
interface WebPageRetriever
    WebPage retrieve_page(URL) signals UnableToContactDomain,
        UnableToFindPage
```

Storing the Links

The LinkRepository stores the URLs that have been found in retrieved pages. It needs to know the base domain so that it can distinguish internal links from external links. LinkRepository also records which URLs are broken and which are OK. LinkRepository is probably going to create a type of object (say a Link), which contains the URL and this information. But we really don't care how it performs its responsibilities. We just want it to do its job, which is defined like so:

Combining Interfaces

We could add retrieve() and parse() methods to Webpage to make it have more behavior. Those methods would delegate to WebPageParser and WebPageRetriever the jobs of retrieving and parsing the page. The interface would look like this:

```
interface WebPage
    set_url(URL)
    set_contents(String)
    String get_contents()
    retrieve()
    URL [] parse_for_URLs()
```

The methods are cohesive in the sense that they all deal with a WebPage. Initially, we'll keep the interfaces separate to simplify testing. Later we can add the methods to WebPage. At that point, we'll need to decide how flexible we want to be. If the implementations for WebPageParser and WebPageRetriever should be changeable, we can set up a configuration interface, which is called when a WebPage is constructed:

```
interface WebPageConfiguration
    WebPageParser get_web_page_parser()
    WebPageRetriever get_web_page_retriever()
```

Alternatively, we can use the Dependency Injection (Inversion of Control) pattern* to set up the implementations. With this pattern, we supply the WebPage with the desired implementations:

```
interface WebPage
    set_parser(WebPageParser)
    set_retriever(WebPageRetriever)
    set_url(URL)
    set_contents(String)
    String get_contents()
    retrieve()
    URL [] parse_for_URLs()
```

USING CONFIGURATION

Advantage—hides implementation requirements

Disadvantage—services have dependency on a configuration interface

USING INVERSION OF CONTROL

Advantage—common feature (used in frameworks)

Disadvantage—can be harder to understand

*See http://martinfowler.com/articles/injection.html for more details.

```
interface LinkRepository
    set_base_domain(Domain base_domain)
    add_URL(URL link, URL reference)
        // adds reference (web page that it comes from)
    URL get_next_internal_link()
        // null if no more links
    URL get_next_external_link()
        // null if no more links
    set_URL_as_broken(URL)
    set_URL_as_unbroken(URL)
```

LinkRepository has a more complicated contract than WebPageRetriever. For example, if you already know the status of the link, you don't want a URL to be returned by get_next_internal_link(). So, you need to check that LinkRepository properly returns the URLs that have not already been retrieved, regardless of how many times they may be referenced.

You should review your interfaces before you implement them. Otherwise, you may implement methods that turn out to be useless.[2] We could add to LinkRepository the job of cycling through the links, retrieving the pages, and parsing the pages. Its current responsibilities center on differentiating between internal and external links and retrieving them in a nonduplicated manner.

We could add a push-style interface to LinkRepository (see Chapter 3) to perform the operation of cycling through the links. The push style in this instance is somewhat more complicated. The method that is called may add additional entries into the LinkRepository that invoked it. So, we'll start with pull style. Shortly, we'll create another interface that actually does the pulling.

We probably want an add_URLs(URL [ ] links, URL reference)[3] as a convenience method. After all, we are retrieving sets of URLs from pages, not just a single URL. So, making a more complete interface simplifies its use.

The two get_next() methods return links that haven't yet been retrieved. If a link is internal, we're going to retrieve the page, parse it, and add the new links to the LinkRepository. If a link is external, we are just going to retrieve the page to see whether it exists, but not parse it. Now that sounds like we might want to have an additional interface (say Link) with two implementations: ExternalLink and InternalLink. They would contain a

[2]Thanks to Rob Walsh for this thought. He adds "or completely wrong."

[3]Do the parameters in the method seem reversed? Should the links go after the reference? Making the order consistent with every code reviewer's idea of correct order is impossible, as you can probably imagine.

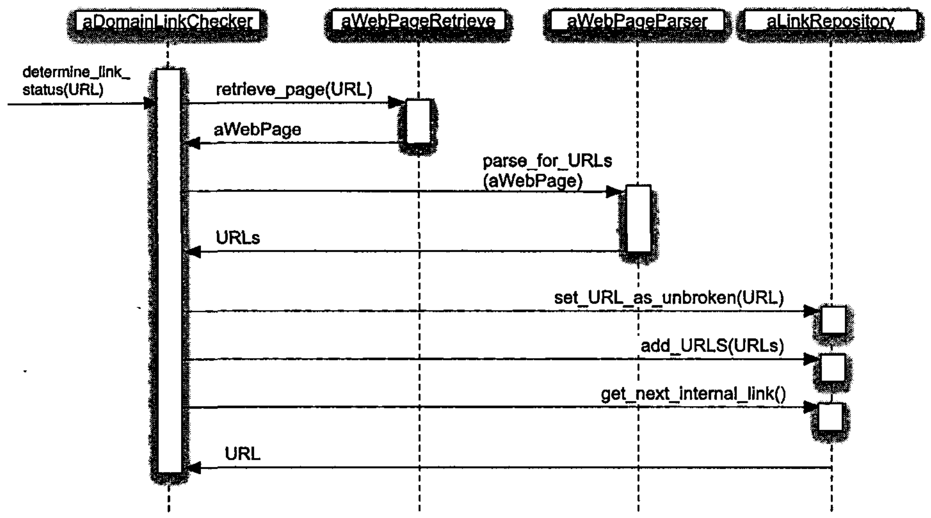


Figure 8.3: SEQUENCE DIAGRAM (FOR INTERNAL LINKS THAT ARE NOT BROKEN)

process method that implements the different steps we just noted. We leave that alteration as an exercise to the reader.

Controlling the Cycling

We create a separate control interface (see Chapter 3), called DomainLinkChecker, for the logic that goes through each link, retrieves it, and checks it. It's going to need a LinkRepository into which to put all the links. Alternatively, DomainLinkChecker could return to us a LinkRepository. The former is simpler, the latter more complex (see Chapter 4). One reason for passing the LinkRepository is that we could record the link status for multiple URLs in the same repository.

```
interface DomainLinkChecker

    set_link_repository(LinkRepository)
    determine_link_status(URL beginning_url)
        // Recursively cycles through links
```

Figure 8.3 illustrates a sequence diagram for determine_link_status(). It depicts how the interfaces we have introduced so far interact.

Simple or Complex?

You may have noted that LinkReferenceCollection contains two methods for sorting the collection:

```
interface LinkReferenceCollection
    sort_by_referring_page()
    sort_by_referred_to_page()
```

This is an example of a simple interface (see Chapter 4). The sort methods are supplied to the user; they do not have to do any additional coding. However, they are limited to those sorts. As an alternative, we could provide a more complex interface:

```
interface LinkReferenceComparator
    boolean greater_than(LinkReference one, LinkReference two)

interface LinkReferenceCollection
    sort_your_way(LinkReferenceComparator)
```

With sort_your_way(), the user provides a greater_than() that compares to LinkReferences and returns an indication of which should come later in the sort order. This is a little more complex, but a lot more flexible.

that matches their needs. You'll also create tests on LinkRepository to check that it can produce the proper data in a LinkReferenceCollection.

With the LinkReferenceCollection, LinkRepository needs a method:

```
LinkReferenceCollection get_link_reference_collection()
```

Now the ReportMaker really needs only a LinkReferenceCollection, rather than the entire LinkRepository, so let's change its interface to the following:

```
interface ReportMaker
    set_link_reference_collection(LinkReferenceCollection)
    String get_output()  // returns text stream
```

8.5 Tests

Before starting implementation, we create an outline of the tests to be run against these interfaces. We derive these tests from the workflow introduced in the "Analysis" section. The tests may yield insights into the degree of coupling between the interfaces.

- WebPage
  - This class does not have much to test. It just has get and set methods.
- WebPageParser
  - Create a WebPageParser, and parse several different WebPages.
  - Check to see that the links agree with manually identified links on these pages.
- URL
  - Create some URLs, and see whether the parts match manually identified parts.
- LinkReferenceCollection
  - Add some LinkReferences, sort them, and see whether results are correct.
- LinkRepository
  - Add URLs to the repository.
    - Check that the next_link() methods return the appropriate values.
  - Add URLs to the repository.
    - Set some as broken.
    - Check to see list of broken links in LinkReferenceCollection matches expectations.
- WebPageRetriever
  - Retrieve some web pages, both internal and external, existing and nonexisting, and see whether results are as expected.
- ReportMaker
  - Print report on a LinkReferenceCollection (e.g., as created in the LinkReferenceCollection test), and see whether the user agrees with its format.

Since there are few dependencies in these tests, the interfaces are loosely coupled. LinkRepository does not perform the URL retrieval, so it is not coupled to WebPageRetriever or WebPageParser. If LinkRepository were coupled to these interfaces, we would either have to create stub implementations for these two interfaces or wait until the real implementation was complete before testing LinkRepository. This easier testing procedure supports our design choice.

The tests suggest an order for creation. URL and WebPage should be completed first; WebPageRetriever, WebPageParser, and LinkRepository can be done in parallel. LinkReferenceCollection should be created prior to ReportMaker. Then the developer for ReportMaker can work with the end user to determine the actual layout.

8.6 Implementation

We discussed in Chapter 2 that an interface definition is not complete until there is at least one implementation of the interface. But my publisher does not want me to fill up this book with code; he says the purpose of web sites is to provide code, so the full implementation of the Link Checker appears at the URL listed in the preface.

Using interface-oriented design, just as with other techniques, does not guarantee that you'll never have to restructure your interfaces or refactor your code. As you develop a system, you usually discover additional information that suggests a restructuring of the design.

For example, the tests for the system revealed that links in a web page are not just links to other web pages.[4] The links may be links to email addresses ("mailto"), file transfer links ("ftp"), or other types of links. Now comes the question: what to do with these links? How do you check a "mailto"? Do you send the address a message? Do you see whether the address is invalid? Do you wait for a response from the recipient? File transfer links are a little easier. You could attempt to retrieve the file. Even if the file were an internal link, you probably would not want to parse the file for links.

Separating policy decisions from implementation can permit greater reuse of interfaces and implementations. LinkRepository can store the links, regardless of the type. Indeed, that is how these other links were revealed. DomainLinkChecker can decide what to do with these other links when it retrieves one from LinkRepository. We can add to the URL interface a method that tells us the type of link, we can retrieve that

[4]You probably knew this already, like I did, but it was absent from my mind until running the tests.

information from the URL and perform the test ourselves, or we can create yet another interface.[5]

The Code

The system is coded in Java, since that's a common language and readily understood by most object-oriented programmers. The method names have been changed to camel case to conform to typical Java coding standards. Since exceptions are a widespread way of signaling in Java, the error signals are thrown exceptions. Separate exceptions were thrown for each type of error that the caller may want to deal with differently.

To keep the code simple, interface implementations are created in the code itself, rather than by calling a factory method. The following is the code for the main routine. It reads the initial URL from the command line. ReportMakerSimple produces tab-delimited output, suitable for import in a spreadsheet or other data manipulation program.[6]

```
public class LinkChecker
    {
    public static void main(String[] args)
        {
        if (args.length >= 1)
            {
            try
                {
                MyURL initialURL = new MyURL(args[0]);

                DomainLinkChecker checker =
                    new DomainLinkCheckerImplementation();
                LinkRepository repository =
                    new LinkRepositoryImplementation();

                checker.setLinkRepository(repository);
                checker.determineLinkStatus(initialURL);
                LinkReferenceCollection linkRefColl =
                    repository.getLinkReferenceCollection();

                ReportMaker reportMaker = new ReportMakerSimple();
                reportMaker.SetLinkReferenceCollection(linkRefColl);
                String output = reportMaker.getOutput();

                // Could print it to a file
```

[5]We leave that decision for the reader to pursue.

[6]I renamed the URL class to MyURL to keep it distinct from the Java URL class. The underlying code delegates most of its work to the Java library class.

GUI Ideas

You can readily adapt this code to use in a GUI. The code in the main() method can become the code for the method of an interface such as LinkChecker:

```
interface LinkChecker
        String check_links(MyURL url)
```

You can set up a dialog box that contains a text field for a URL and a submit button. Clicking the button invokes the check_links() method. The string returned by the method can be displayed in an edit box or a separate window.

A GUI often has some sort of feedback mechanism to indicate to the user that progress is being made. We can use a callback interface (push-style interface from Chapter 3) to perform the feedback. We need to make minor changes in the interfaces to pass a callback method. Every time a new link is accessed, the current_link() method is called. The method could display the name in a read-only text box.

```
interface LinkCheckerCallback
        current_link(MyURL link)

interface DomainLinkChecker
    determineLinkStatus(MyURL url, LinkCheckerCallback)

interface LinkChecker
        String check_links(MyURL url, LinkCheckerCallback )
```

8.7 Retrospective

Separating responsibilities into several interfaces allows these interfaces to be employed in programs other than the one presented in this chapter. For example, since web page retrieval has been separated from the parsing, you can employ these classes in a web page (HTML) editing program. You can use WebPageParser to obtain links referenced in a page for display in a window.

8.8 Things to Remember

The Link Checker demonstrated the following:

- Use IRI cards as an interface design tool.

Chapter 9

Web Conglomerator

We're going to create an interface-oriented design for a web-based application. A common development question is, when should you make a design more general? In this application, we'll show one case of transforming a specific interface into a more general one and one case where that transformation is deferred.

9.1 Vision

The Web Conglomerator is your own custom browsing portal. Many web sites offer the ability to customize a web page with the information in which you have an interest. The Web Conglomerator performs this service on your machine. It presents a custom web page to you with content derived from many sites. The example in this chapter specifically shows travel-related information, but the information could be about anything from the stock market to butterflies.

9.2 Conceptualization

We have only two use cases for the Web Conglomerator (See Figure 9.1, on the following page): to configure the system and to request a current display.[1]

Here are descriptions for both of the use cases:

[1]As a side note, the system should be arranged so the user must request an update of the page. Creating a system that automatically updates places a burden on the information providers.