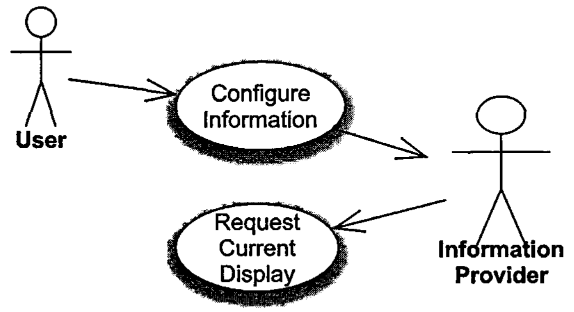


Figure 9.1: USE CASES

Use Case: Configure Information

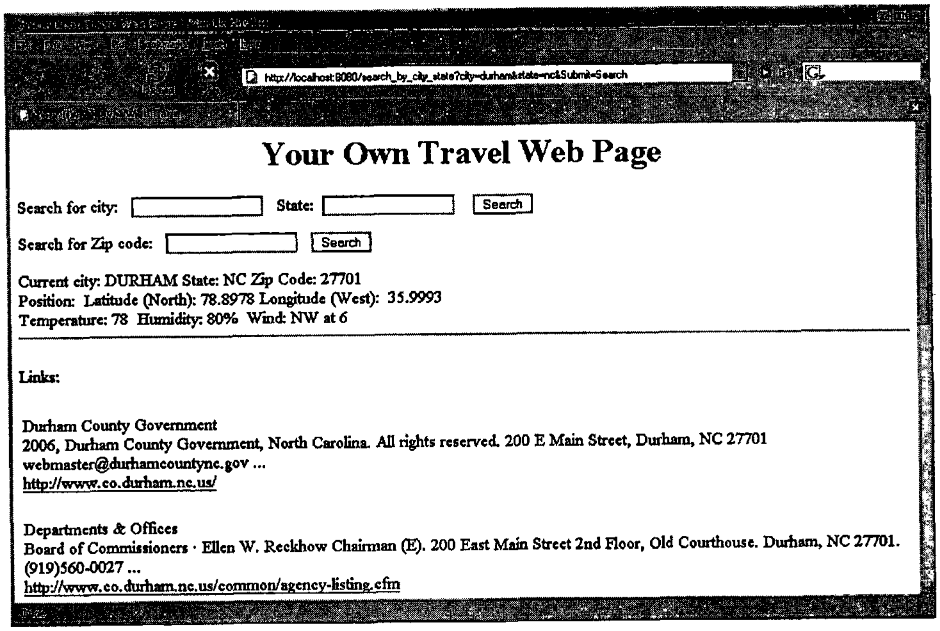
1. User determines the blocks of information to be presented.
2. System stores the configuration information.

Use Case: Request Current Display

1. User requests that the Web Conglomerator display the current page.
2. System responds with a page containing the current information.

GUI prototypes help both the developer and the user understand the intended operation of a system. So, we prototype what our travel-related conglomerator may look like (other forms of information could easily be conglomerated as well).

From the user's viewpoint, he initially loads a page into his browser, say `localhost:8080//travel.html`. The browser displays that page, as shown in Figure 9.2, on the next page. When the user submits a search for a city or ZIP code, the browser contacts the web conglomerator, which returns a page with the information corresponding to the display. The web conglomerator creates the web page from content provided by either web services or other web pages, or both.


http://localhost:8080/search_by_city_state?city=durham&state=nc&Submit=Search

Your Own Travel Web Page

Search for city: State: Search

Search for Zip code: Search

Current city: DURHAM State: NC Zip Code: 27701
Position: Latitude (North): 78.8978 Longitude (West): 35.9993
Temperature: 78 Humidity: 80% Wind: NW at 6

Links:

Durham County Government
2006, Durham County Government, North Carolina. All rights reserved. 200 E Main Street, Durham, NC 27701
webmaster@durhamcountync.gov ...
http://www.co.durham.nc.us/

Departments & Offices
Board of Commissioners · Ellen W. Reckhow Chairman (E). 200 East Main Street 2nd Floor, Old Courthouse. Durham, NC 27701.
(919)560-0027 ...
http://www.co.durham.nc.us/common/agency-listing.cfm

Figure 9.2: WEB BROWSER RESULTS

9.3 Analysis

Let's create some IRI cards for potential interfaces. We stated that we are going to display the information in a web browser. However, the display of information should not be coupled to how we obtain the information. Otherwise, changing a provider for a piece of information can have ramifications throughout the system. So, we create data gatherer interfaces, as well as data formatter interfaces. Instead of showing the actual cards, we save a little paper and just list the proposed interfaces:

- IndividualDataGatherer
  - Retrieves one type of information (e.g., weather) from an information provider
- IndividualDataFormatter
  - Formats information of a single information type
- WebConglomerator
  - Formats information from IndividualDataFormatters

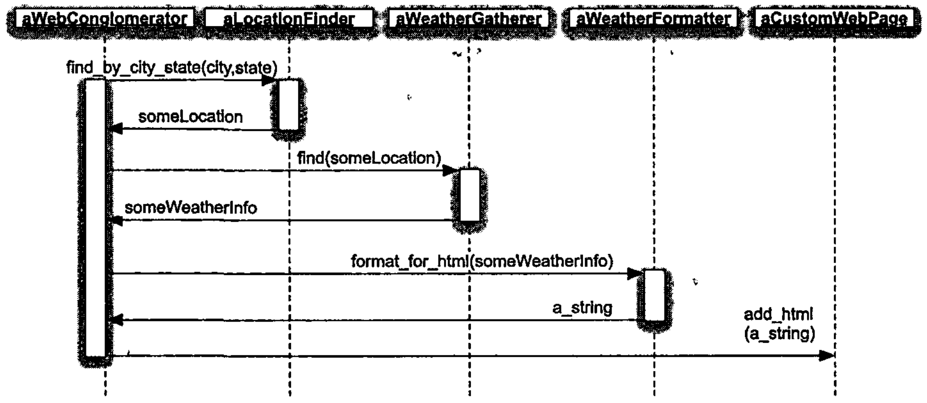


Figure 9.3: SEQUENCE DIAGRAM FOR THE WEB CONGLOMERATOR

are then added to the CustomWebPage. A sequence diagram for the interactions between these interfaces appears in Figure 9.3.

9.4 Testing

Before continuing to design, we outline some tests to be run against these interfaces. Creating tests can point out coupling issues.[2]

- DataGatherer
  - Check that the information returned by find_by_location() matches information from other sources. Because of the dynamic nature of the data, we may need to create two implementations of DataGatherer and compare the data returned.[3]
- DataFormatter
  - Put the display strings into a simple HTML file, and see whether the output is displayed in a readable form.
- WebPageLinkDataGatherer
  - Check that the number of links and the data in each one correspond to the information from the search results.

[2]A tester noted that not all these tests might be automated.
[3]See the "Multiple Implementations" sidebar in Chapter 8.

- Construct tests for interfaces as you create the interfaces.
- Create functional tests that are implementation independent.
- Separate information retrieval from information display.

The system also showed when developing interfaces that may be useful in a variety of contexts, you can:

- Create a concrete implementation of an interface before abstracting it.
- Develop application specific interfaces before generalizing them.
- Generalize interfaces using frameworks or templates

Chapter 10

Service Registry

In Chapter 5 on remote interfaces, we discussed looking up service providers for an interface in a directory. In this chapter, we're going to create a general service registry as a means for exploring some issues in providing networked services. This registry also will give us an opportunity to experience a document-style interface.

10.1 Vision

The Service Registry allows users to advertise the availability of service providers, even if their computers do not have static Internet Protocol (IP) addresses.

The services do not have to be on reserved ports, such as the HTTP port (80), or use any particular protocol, such as SOAP or RMI. A couple of examples demonstrate how the Service Registry will work.

Suppose you have a video camera connected to your computer in your house. Your computer connects to the Internet via a dynamically assigned IP address.[1] Since the IP address is dynamic, you need some way to discover it. The Service Registry provides that ability.

The program connected to your camera registers a service identifier (a ServiceID), an IP address, and a port with the Service Registry. You are sitting at work and want to see the picture on your web cam. On

[1]Dynamic addresses tend to be the same for long periods of time. Suppose you want to look at the video from the camera on other computers, such as the one in your office. You might note the IP address before you left the house. The address may be the same when you attempt to connect. However, in general, you should not rely on a "static" dynamic IP address.

IP Addresses and Ports

The Domain Name System (DNS) supplies IP addresses (e.g., 66.15.240.233) for host names (e.g., www.pughkilleen.com).[*] The IP address is usually the same for long periods of time (static). A dynamic DNS address provides a way to connect to computers that do not have static IP addresses, such as those connected via cable, DSL, or phone lines. Dynamic DNS's are updated frequently with new addresses.

Servers, such as a web server, communicate over a port. You can think of a port as the equivalent to the number of a phone extension. To communicate with a server, you need to know on what port it is communicating. Standard services, such as web servers, have fixed port numbers. For example, web servers communicate on port 80 and mail servers on port 25.

Nonstandard services may communicate on any port. DNS (normal or dynamic) provides only IP addresses. So, it's harder to provide nonstandard services that do not have fixed port numbers.[†]

[*] See `http://en.wikipedia.org/wiki/Dns` for more information.

[†] Remote Procedure Calls (RPCs) have a separate mechanism (e.g., Unix's RPC Mapper) that runs on each host for providing port numbers for services. Java's JINI provides a Java language version of a combined IP address/port number lookup.

your office computer, the program that displays the video looks up the ServiceID in the Service Registry, retrieves the IP address and port, and then connects to the program on your home computer.

As another example, suppose you are developing a new interactive game that involves communication between two or more players. The players install your software on their computers and want to link up. Unless they are using fixed IP addresses with fixed port assignments, they will have to manually communicate to each other their current IP addresses and ports.

When your game program starts up on each player's computer, it sends to the Service Registry your game's ServiceID and the IP address and port that the game program has been assigned by the operating system. Your game also looks up on the Service Registry any current registered providers for your game's ServiceID. It can use the retrieved IP

Chapter 11

Patterns

11.1 Introduction

We introduced some patterns in the previous chapters. In this chapter, we'll review those patterns and cover a few patterns that revolve primarily around the substitutability of different implementations. Using common patterns can make your design more understandable to other developers.

Each pattern has trade-offs. The "Gang of Four" book[1] details the trade-offs. In this chapter, we list some of the prominent trade-offs.

11.2 Factory Method

The Factory Method pattern is the quintessential way to obtain an implementation of an interface. You call a method that returns a reference to an implementation. With a factory method, you can easily substitute implementations without having to change a single line of code in the calling method. Let's take the example of the WebPageParser. In the code that uses the method, we could create an instance of a particular parser. For example:

```
class RegularExpressionWebPageParser implements WebPageParser
    {// some code} .
WebPageParser web_page_parser = new RegularExpressionWebPageParser();
```

From that point onward in the code, we really don't care what the implementation is for WebPageParser. However, if we want to change

[1] *Design Patterns* by Gamma, et al.

the implementation, we have to alter the type of object we are creating. Instead, we can use a factory method to create a WebPageParser:[2]

```
class WebPageParserFactory
    static WebPageParser get_web_page_parser()

// Use of method
WebPageParser web_page_parser =
    WebPageParserFactory.get_web_page_parser();
```

Now we can change the implementing class in a single place, and all users will get the new implementation.

We could make the factory method a little more intelligent and specify some criteria for choosing a particular parser. We might want a fast but possibly inaccurate parser or a meticulous one that can parse anything. We do not need to specify a particular name, just our needs. For example:

```
WebPageParser web_page_parser =
    WebPageParserFactory.get_web_page_parser(SLOW_BUT_METICULOUS);
```

- A registry lookup works like the Factory Method pattern. You request an implementation of an interface using a service identifier. The method call is a little more generic. For example:

```
WebPageParser web_page_parser =
    (WebPageParser) registry.lookup(WebPageParserID)
```

- The Abstract Factory pattern works one more level up from a Factory Method pattern. You may have multiple interfaces, and for each one you want to create related implementations. In the Service Registry example in Chapter 10, we had several different documents. The representation of these documents could be in XML, YAML, tab-delimited text, or another format. We could provide implementations for each document that encoded the data in a particular representation. For example:

```
interface RegistrationDocument
    set_version()
    set_service_provider_information()
interface LookupResponseDocument
    set_version()
    set_service_id()
    set_connection_information()
```

[2]We show this as static, which makes get_web_page_parser() a class method. Otherwise, we would have to create or find an implementation of WebPageParserFactory before we can get a WebPageParser.

```
interface DocumentFactory
    RegistrationDocument get_registration_document()
    LookupResponseDocument get_lookup_response_document()
```

Now you create multiple implementations of DocumentFactory. Each version of DocumentFactory creates RegistrationDocuments and LookupResponseDocuments that encode information in a particular format.

FACTORY METHOD

Advantage—makes implementation of interface transparent

ABSTRACT FACTORY

Advantage—makes implementation of set of interfaces transparent

11.3 Proxy

In the Proxy pattern, you have multiple implementations of a single interface. One implementation (the proxy) acts as an intermediary to another implementation. Each method in the proxy may perform some internal operations and then call the corresponding method in another implementation. The implementations may be in the same program, in the same computer, in different processes, or in different computers.

The caller of a proxy usually need not be aware that the proxy being called is making calls to other implementations. The functions that proxies can perform include security, caching of results, and synchronization[3] A common use of a proxy is to provide a local interface to a remote interface. This is termed a *remote proxy*. For example, here's an interface to get the current price for a stock:

```
interface StockTickerQuoter
    Dollar get_current_price(Ticker)
```

You can get a local copy by using either a specific class or a factory method, as in the previous section. For example:

```
StockTickerQuoter stock_ticker_quoter =
    new StockTickerQuoterLocalImplementation()
```

The local implementation connects to a remote implementation via a network connection, typically a Remote Procedure Call (see Chapter 6).

[3]See http://www.research.umbc.edu/~tarr/dp/lectures/Proxy-2pp.pdf for more details.

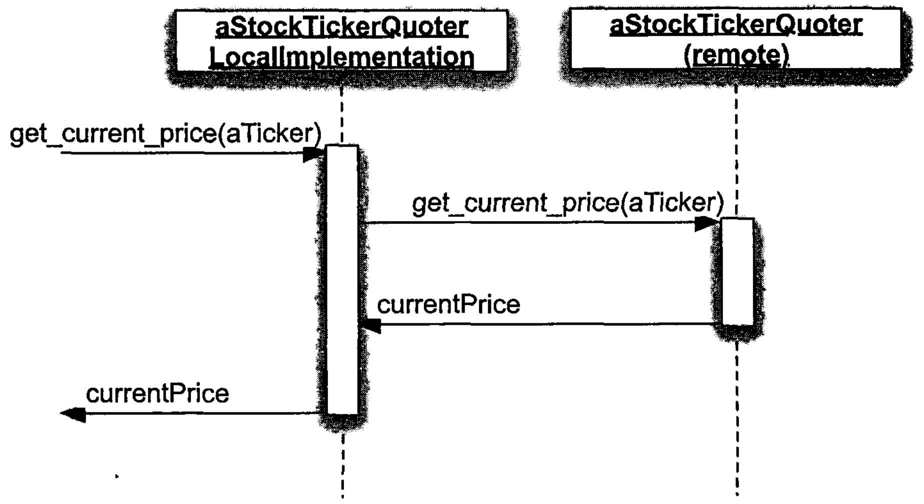

aStockTickerQuoter LocalImplementation
aStockTickerQuoter (remote)
get_current_price(aTicker)
get_current_price(aTicker)
currentPrice
currentPrice

Figure 11.1: CONNECTION SEQUENCE

The remote implementation has exactly the same interface. Figure 11.1 shows the sequence diagram.

The same interface might be used with multiple proxies that perform various services. The services could include security checking, encryption/decryption, and logging.[4] If we want these services to be dynamically added, typically the interface includes a way to denote another proxy in the chain. For example, we could log each time a price was requested by adding another implementation in the chain:

```
interface StockTickerQuoter
    Dollar get_current_price(Ticker)
    set_next_stock_ticker_quoter(StockTickerQuoter);
class StockTickerQuoterLogger implements StockTickerQuoter
    {
    StockTickerQuoter next_stock_ticker_quoter;
    set_next_stock_ticker_quoter(StockTickerQuoter stq)
        {
        next_stock_ticker_quoter = stq;
        }
```

[4]Some authors suggest that these additional services would make this a Decorator pattern (see the next section).

```
    Dollar get_current_price(Ticker a_ticker)
        {
        Logger.send_message("Another quote");
        return next_stock_ticker_quoter.
            get_current_price(Ticker a_ticker)
        }
    }
```

Now we can give the original implementation of the interface another implementation to invoke:

```
StockTickerQuoter stock_ticker_quoter =
    new StockTickerQuoterLogger();
stock_ticker_quoter.set_next_stock_ticker_quoter
    (new StockTickerQuoterLocalImplementation());
```

(Some authors consider this a variation of the Decorator pattern, which is discussed next. They suggest that the StockTickerQuoterCounter decorates another StockTickerQuoter by adding a logging functionality.)

When we get a current price, the price is not only retrieved, but a message is also logged. After the initial setup, the use of StockTickerQuoter in the remainder of the program does not change.

PROXY

Advantage—additional features (security, logging, etc.) can be added transparently.

11.4 Decorator

The Decorator pattern works like the Proxy pattern. The decorator has the same interface as the decorated interface. Typically, it also has an additional interface that represents the decoration. In the previous Proxy pattern example, the user might not be aware that StockTickerQuoterLogger was part of the proxy chain.

On the other hand, the user typically is the one decorating a class. For example, we might add a feature that counts the number of times we asked for a stock quote. This feature is provided by an additional interface, Counter. The decorator implements this interface as well as the original one. For each method in the original interface, it may just call the one in the decorated implementation, or it may add some functionality. For example:

```
interface Counter
    {
    int get_count();
    reset_count()
    }
class StockTickerQuoterCounter implements StockTickerQuoter, Counter
    {
    int counter;
    int get_count()
        {
        return counter;
    }
    void reset_count()
        {
        counter = 0;
    }
    StockTickerQuoter next_stock_ticker_quoter;
    set_next_stock_ticker_quoter(StockTickerQuoter stq)
        {
        next_stock_ticker_quoter = stq;
    }
     Dollar get_current_price(Ticker a_ticker)
         {
        counter++;
        return next_stock_ticker_quoter.
            get_current_price(Ticker a_ticker)
      }
    }
```

The get_current_price() method delegates to another implementation. In addition, it increments a count that is accessible by the Counter interface. When we're using this class, we give it an implementation to decorate:

```
StockTickerQuoterCounter stock_ticker_quoter_decorator =
    new StockTickerQuoterCounter();
stock_ticker_quoter_decorator.set_next_stock_ticker_quoter
    (new StockTickerQuoterLocalImplementation());
```

This code looks a lot like the proxy version. The difference is that you can call additional methods on stock_ticker_quoter_decorator, such as get_count(). Another difference is that proxies are often designed to work together. The Decorator pattern can be used on classes that existed before the decorator is created.

- DECORATOR

  Advantage—adds behavior on classes not explicitly designed for expansion

11.5 Adapter

Adaptation is a major feature of interface-oriented design. You create the interface you need. If a module exists whose interface is close to your need, you can alter your interface to match that existing interface. Otherwise, you can adapt the interface of that module to match your desired interface. For example, in the Link Checker, we created a WebPageParser interface:

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

Java has an HTMLParser that performs the services needed by a realization of this interface. It has different method names. So, we create an adapter that implements the WebPageParser interface by using the HTMLParser class.

An implementation of WebPageParser can adapt different interface styles to the style represented by WebPageParser. For example, WebPageParser uses a pull style. You ask it for a set of links after it parses the document. You could create an implementation that employs the push style, such as the SAX parser. Internal methods would gather the links into a collection, and parse_for_URLs() would return that collection.

The interface an existing class provides may not meet the paradigm that you want for your classes. So, you create an adapter for that interface. For example, suppose you find that opening a buffered file in Java is a little complicated. You create a class MyFileInputStream that adapts that class. The interface for this class could look like the following:[5]

```
class MyFileInputStream
    {
    MyFileInputStream(Filename name) throws FileNotFoundDeviation,
        FileNotAccessibleDeviation, FileError
        { ...  }
    read() throws EndofFileDeviation, FileError
        {...}
    close()
        {...}
    //... Other methods,
    }
```

[5]This is a simple example of an adapter. You always have trade-offs in using an adapter class for library routines. A developer new to a project needs to learn a new class.

A Textual Adapter

Adapting interfaces is not limited to such programmatic interfaces. You can adapt textual interfaces. An example of adaptation is the sendmail configuration process. sendmail is a common email server. It uses the configuration file `sendmail.cf`. The syntax for that file is, well, to put it nicely, "interesting." To simplify configuration, there is a `sendmail.mc` file. The syntax of that file is in a macro language (m4). You transform `sendmail.mc` into `sendmail.cf` by using the `m4` program.

The `sendmail.mc` file represents a textual adapter. Now, people might suggest that the syntax of that file is still somewhat "interesting." But at least it's simpler than the `sendmail.cf` file.

The buffering is internal to this class. The methods signal a set of conditions that you find more meaningful than just IOException.[6] The close method does not throw an exception. The exception is caught within the adapter, which simplifies using the interface. If an exception occurs, it is logged.

ADAPTER

Advantage—you use an interface designed on your needs.

Disadvantage—adapting standard interfaces creates more interfaces to learn.

11.6 Façade

The Adapter pattern turns a single interface into a different interface. The Façade pattern turns multiple interfaces into a single interface. For example, the WeatherInformationTransformer interface in the Web Conglomerator solution (Chapter 9) hides the multiple interfaces of WeatherInformation, WeatherInformationDataGatherer, and WeatherInformationDisplayFormatter.

[6] I like to split errors into deviations (failures that may occur during normal operation of a program) and errors (failures that should not normally occur). For example, not having permission to read a file is something that could happen. A user could alter the permissions of the file and rerun the operation. An error reading from a file because of a hardware problem should never be normally encountered.

The user of WeatherInformationTransformer is not aware of the underlying interfaces.

FAÇADE

Advantage—a single interface can be easier to understand than multiples

11.7 Composite

In the Composite pattern, an interface to a group of objects is the same as that to a single object. The Composite pattern is commonly used with trees. A leaf on a tree has an interface. A branch (which has multiple leaves) has the same interface, plus an additional interface for adding or removing leaves. For example, the organization of a GUI usually works as a tree. The leaves are Components, and the branches are Containers:

```
interface Component
    draw()
interface ComponentContainer
    add_component(Component)
    remove_component(Component)
Widget implements Component
    draw()
Window implements ComponentContainer, Component
    draw()
    add_component(Component)
    remove_component(Component)
```

The draw() method for a Container such as Window invokes the draw() method for each of the Components it contains. Suppose a method receives a reference to a Component and wants to draw it. It does not matter whether the reference actually refers to a Widget or a Window. In either case, the method simply calls draw().

We can also use the Composite pattern to make a group of interfaces act like a single interface. For example, given the following interface:

```
interface StockTickerQuoter
    Dollar get_current_price(Ticker)
```

we could create an implementation that simply connects to a single source to provide the current price:

```
class StockTickerQuoterImplementation implements
    StockTickerQuoter
```

Alternatively, we could have an implementation that connects to multiple sources. It could average the prices from all sources, or it could call each source in turn until it found one that responded. The user's code would look the same, except for setting up the StockTickerQuoterContainer:

```
interface StockTickerQuoterContainer
    add(StockTickerQuoter)
    remove(StockTickerQuoter)
class StockTickerQuoterMultiple implements StockTickerQuoter,
    StockTickerQuoterContainer
```

A factory method can make using a composite like this transparent. Inside the method, StockTickerQuoter implementations can be added to StockTickerQuoterContainer. We might obtain an implementation with this:

```
enumeration StockTickerQuoterType = {
    FREE_BUT_DELAYED_SINGLE_SOURCE,
    PRICEY_BUT_CURRENT_SINGLE_SOURCE,
    AVERAGE_ALL_SOURCES};
StockTickerQuoter stock_ticker_quoter =
    StockTickerQuoterFactory.get_instance(AVERAGE_ALL_SOURCES)
```

COMPOSITE

Advantage—calls to multiple implementations appear the same as calls to a single implementation

11.8 Things to Remember

We've looked at a number of patterns that deal with interfaces. These patterns are ways that multiple interfaces or multiple implementations can be opaque to the user. Employing these patterns can make your code more flexible and potentially easier to test.

- Factory Method
- Proxy
- Adapter
- Decorator
- Façade
- Composite

Appendix

You may be reading this because you didn't see the title "Appendix." Why is this material in an appendix? Well, it's "an accessory structure of the body" of the book. Some people might find particular topics in here that they think should have been part of the regular text. Just because the topics are in here doesn't mean they are not important; they just didn't fit into the flow. So, they got relegated to here.[1]

A.1 More about Document Style

We discussed document-style interfaces in Chapter 6. Here are a few more suggestions on how to employ them.

Document Suggestions

The XML Design ASC X12C Communications and Controls Committee compiled a number of suggestions for creating XML documents.[2] These suggestions can be useful, regardless of the actual form of the document. The suggestions include the following:

- Keep the use of features and choices to a minimum.
- Make the schema prescriptive: have schemas that are specific to a particular use, not generalized. More schemas make for fewer options, and thus you can have tighter validation.
- Limit randomness: have a limited number of variations.[3]

[1]And they really might not be that important to some people—that's OK; I won't be offended if you skip them.

[2]http://www.x12.org/x12org/xmldesign/X12Reference_Model_For_XML_Design.pdf

[3]The committee notes that limiting randomness provides a good philosophical basis for disallowing features such as substitution groups and the "ANY" content model when designing document schemas.

- Create reusable parts and context-sensitive parts. Examples of reusable parts are Addresses and Contact Information.
- *Form a tree of compositions with optional and repetition specifiers*
- Have a document version number. If changes have been made in the document, update the version number. (Or have a chain of unique IDs that reference previous versions.)

Using Standard Documents

If you are developing documents for internal use, you are free to develop your own forms. However, you may want to investigate the standards to avoid re-creating the wheel. Standard documents may have more data than you really need or desire, but they offer a good starting point for creating your own documents. For example, if you are developing a commerce system, investigate all the standard transaction sets of the federal government. These standards are being converted to XML (OASIS ebXML). If those sets do not make sense in your application, then search the Internet for ones that might be more appropriate.

To make use of the standards, set up a data interface (a DTO) that parallels the document structure. Use this interface to create the document with appropriate validation. For an example, let's use the Federal Information Processing Standards.[4] One of the standard documents for a purchase order is the FIPS 850 Purchase Order Transaction Set.[5] A data interface, such as the DTO given next, can represent this transaction set. We don't show the entire interface, since the specification for this transaction set is 262 pages long.[6] The specification is broken down into elements that have initials. The element initials are at the beginning of each name in the DTO. Normally the names in a DTO should not be tied to specific numbers. In this case, however, they are used to explicitly tie the fields to the specification.

```
data interface PurchaseOrderTransactionSet
    ST_TransactionSetHeader
        ST01_TransactionSetCode = 850
        String ST02_TransactionSetControlNumber
        enumeration PurposeCode {ORIGINAL, DUPLICATE, ...}
```

[4]See `http://www.itl.nist.gov/fipspubs/` to get complete documents.

[5]See `http://www.sba.gov/test/wbc/docs/procure/csystsba.html` for an introductory description of the ASC X12 transaction sets.

[6]See `http://fedebiz.disa.mil/FILE/IC/DOD/3010/R3010/R3010/Rev1/31r850_a.pdf`.

```
enumeration POTypeCode {CONTRACT, GRANT...}
BEG_BeginningSegmentForPurchaseOrder
    PurposeCode BEG01_TransactionSetPurposeCode
    POTypeCode BEG02_PurchaseOrderTypeCode
    CommonString BEG03_PurchaseOrderNumber
    enumeration UnitCodes {EACH, FEET,...}
    PO1_baseline_item_data []
        String PO101_assigned_identification
        Count PO102_quantity_ordered
        UnitCodes PO103_unit_or_basis_for_measurement
        Money PO104_unit_price
```

Just creating the DTO may help you understand the organization of the document. You can build into this DTO checks for the rules for the transaction set. For simple validation, the set methods for each field can enforce the rules. An overall validation method can handle the cross-validation of fields.

Your internal document may not require all the fields listed in the specification. So just include the ones you need. If you need to convert your internal document to an external one that matches this specification, you'll find it easy to copy the corresponding fields.

A.2 Service-Oriented Architecture

Service-oriented architecture (SOA) is an emerging standard for providing remote services both to clients internal to an organization and to external organizations. Chances are you soon will communicate using SOA services. An SOA service is reusable, well-defined, published, and standards-compliant. An SOA service groups a set of operations, similar to the way that service interfaces, described in Chapter 2, group a set of methods. Like service interfaces and implementations, how an SOA service performs an operation is entirely opaque to a consumer of that service.

SOA operations tend to be business-oriented operations, such as the individual steps in the airline reservation example shown in Chapter 6, rather than technical operations, such as retrieving a database record. A business process uses a sequence of these operations to perform a particular action, such as the airline reservation process. Examining an enterprise for reusable services and creating an entire service-oriented architecture are beyond the scope of this book. Designing the services' contract and their protocol (sequence of operations or documents) is the essence of creating an SOA service. The same contractual and protocol design trade-offs that we have explored in this book for interfaces also apply to services. Services are effectively interfaces, albeit at a higher level.

SOA frameworks provide additional services beyond standards for documents and communication protocols that ensure interoperability. For example, they can offer security: encryption of communication, authentication (identification of a user), and authorization (determining that a user is entitled to perform specific operations). Using an SOA framework with its attendant services decreases the amount of time involved in creating an external interface.[7]

As this book is going to print, the Organization for the Advancement of Structured Information Standards (OASIS) is working on SOA standard models. Depending on your viewpoint, SOA uses either only message-based interfaces (document-style) or both message-based and Remote Procedure Call–based (procedural-style) interfaces. A general consensus says that an SOA must have the following characteristics:

- Interface is platform-independent (OS, language)—loosely coupled.
- Services are discoverable—dynamically locatable.
- Services are self-contained.

SOA frameworks such as web services and CORBA meet these characteristics. Some authors suggest another constraint:

- Service provider and consumer communicate via messages (i.e., a document-style interface)

In that case, CORBA would not be considered an SOA.

An SOA framework, mapped to web services and CORBA, is shown in Figure A.1, on the next page.

The PizzaOrdering document interchange shown in Chapter 6 can form the basis for a simple SOA service. Figure A.2, on the facing page shows how a service consumer connects to an implementation of a PizzaOrdering service. The consumer contacts the directory service and requests information about a host that provides an implementation of

[7]See `http://www-128.ibm.com/developerworks/webservices/library/ws-soad1/` for more details about SOAs.