7

Distributed Application Architecture

Each thing is, as it were, in a space of possible states of affair. This space I can imagine as empty, but I cannot imagine the thing without the space.

—Ludwig Wittgenstein
Tractatus Logico Philosophicus

In isolation, your Java objects have no meaning, i.e., they do nothing. Java objects represent things outside the application: a customer, a savings account, and so on. Before getting into the details of individual objects, you truly need to understand the space in which you expect them to operate. Architecture is the space in which software objects operate.

In this chapter, I will lay the foundations for database development in the object-oriented world of Java by examining the architecture of an application you will be building over the rest of this book. You may find that these foundations span a broad spectrum of issues. I will not touch JDBC, EJB, or any of the other details required for the creation of individual objects. Instead, my goal is to help you cut down on the work you will need do over and over again each time you build a database application and to maximize the relevance of what you build to future needs. Many of the classes I show you in this chapter are common and generic, perhaps something that you could use to create a standard package for use in all kinds of applications.

One thing you may have noticed about Java or similar object-oriented languages such as Smalltalk or Python is that there are so many classes. You want to try to understand what class X does, but you find that it in turn extends class Z, which contains classes A and B. If you are totally comfortable with the object-oriented paradigm, this interweaving of classes may not faze you. On the other hand, it may easily seem confusing to people used to dealing with languages such as C. Unlike C,

for which you may have a library function and perhaps an associated data structure, Java bundles up data and functions inside classes for manipulating that data. Java data never gets directly manipulated except by the class that owns the data.

I will, of course, continue operating in Java's object-oriented framework. Among other things, this means that whenever you need to represent a new concept, you will use new classes. You should approach each new class trying to understand what class it extends, which interfaces it implements, and what others it relates to in other ways. I will help you along graphically wherever possible by providing UML-standard* diagrams that illustrate the class relationships.

Architecture

The value of system architecture is only recently being recognized in the software industry. As I stated before, paraphrasing Wittgenstein, architecture is the space in which objects operate. It defines the contracts through which they interact with external system components and each other. The primary duty of a system architect is to ask the question, "What if . . . ?".

Strategic Versus Tactical

During most software development processes, each role in the process is responsible for some level of tactical thought and some level of strategic thought. By tactical thought, I mean thinking about the problem at hand and ignoring hypothetical issues. Strategic thought, on the other hand, means thinking about all possible worlds and weighing their probability. An example of a strategic decision might be to have your application design abstract to a generic concept of "product" when the only product your company sells is fuzzy dice. That strategic decision will let your company move into selling seat covers in the future without rewriting the system. Tactically speaking, however, the system only requires that you support fuzzy dice.

A good system architect is a heavily strategic thinker. The architect needs to understand these tactical issues and determine the best high-level solution that minimally addresses the tactical issues, while doing no harm to the ability to address strategic issues without good cause. In the previous example, it is certainly easier for everyone to think about the system in terms of fuzzy dice. The path of least resistance therefore would be to code a system made up of fuzzy dice objects. Everyone could agree that the business is about selling fuzzy dice, and everyone clearly understands what fuzzy dice are. Taking that path of least resistance, however, harms one important strategic question: What if the company decides to sell

* UML stands for Unified Modeling Language. It is a new standard for documenting object-oriented analysis and design.

other products? Furthermore, building the system as a system of fuzzy dice instead of a system of products provides absolutely no advantage to mitigate the harm done to the strategic question.

Architectural Questions

As odd as it may sound, development teams often unwittingly plan for failure instead of success. How often have you heard someone say, "I know that is the right way to do this, but you just need to be able to support . . . " or some variation thereof? That statement represents planning for failure. Any successful software project will see its software used in realms well beyond its original intentions. A poor piece of software, however, may minimally serve some short-term goals before it is eventually replaced. The job of an architect is to make sure no tactical decisions fall into the "planning for failure category." The architect assumes success and structures the system accordingly. Typical architectural questions are:

- How do I partition my system?
- Should I support diverse user bases?
- How do I enable the system to integrate with third-party applications?
- What standards should the design and development teams adhere to?
- What tools best meet these needs?

The first question is listed first for a reason: it is the first question any architect should address for a system. I will now present a high-level view of two distinct architectures and then introduce a very specific architecture that you will follow for the banking application you build in this book.

Common Architectures

There are basically two major kinds of modern architectures: two-tier client/server and three-tier—also commonly called n-tier. Each one has many variations. At a high level, these architectures focus on the partitioning system processing. They decide on what machine and in what process space a given bit of code executes.

Client/server is often a generic umbrella term for any application architecture that divides processing among two or more processes, often on two or more machines. Any database application is a client/server application if it handles data storage and retrieval in the database process and data manipulation and presentation somewhere else. The server is the database engine that stores the data, and the client is the process that gets or creates the data. The idea behind the client/server architecture in a database application is to provide multiple users with access to the same data.

Two-tier client/server

The simplest shape a client/server architecture takes is called a *two-tier* architecture. In fact, most client/server architectures are two-tier. The term “two-tier” describes the way in which application processing can be divided in a client/server application. A two-tier application ideally provides multiple workstations with a uniform presentation layer that communicates with a centralized data storage layer. The presentation layer is generally the client, and the data storage layer is the server. Some exceptional environments, such as the X Window System, shuffle the roles of client and server.

Most Internet applications—email, Telnet, FTP, gopher, and even the Web—are simple two-tier applications. Without providing a lot of data interpretation or processing, these applications provide a simple interface to access information across the Internet. When most application programmers write their own client/server applications, they tend to follow this two-tier structure.

Figure 7-1 shows how two-tier systems give clients access to centralized data. If you use the Web as an example, the web browser on the client side retrieves data stored at a web server.

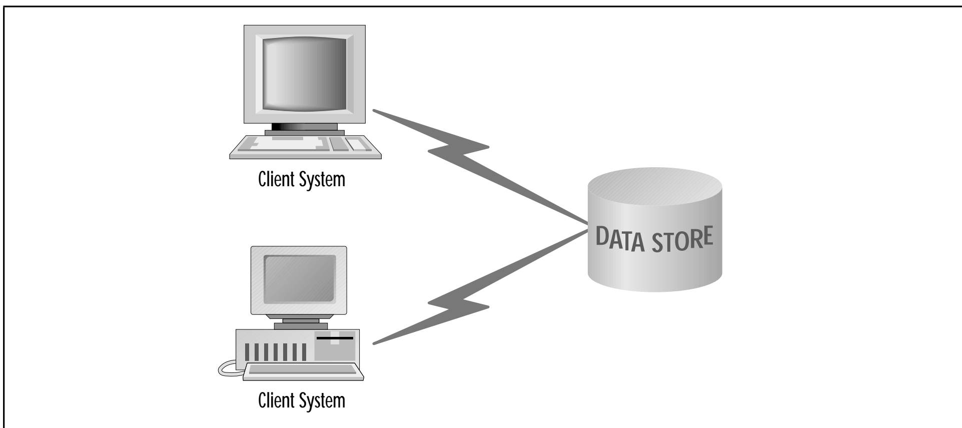

Figure 7-1. The two-tier client/server architecture

The architecture of a system depends on the application. For situations such as the display of simple web pages, you do not need anything more than a two-tier design has to offer. This is because the display of static HTML pages requires very little data manipulation, and thus there is very little to fight over. The server sends the page as a stream of text, and the client formats it based on the HTML tags. There are none of the complicating factors you will see in upcoming applications: choices between different tasks, redirecting of tasks to subordinate methods, searches through distributed databases, and so on.

Even when your application gets slightly more complex-—such as a simple data-entry application like the many web-based contests that appeared in the early days of the Web—a two-tier architecture still makes sense. Let's look at how you might build such an application to see why additional complexity is unnecessary. Figure 7-2 shows this application in the context of a two-tier architecture that you saw in Figure 7-1.

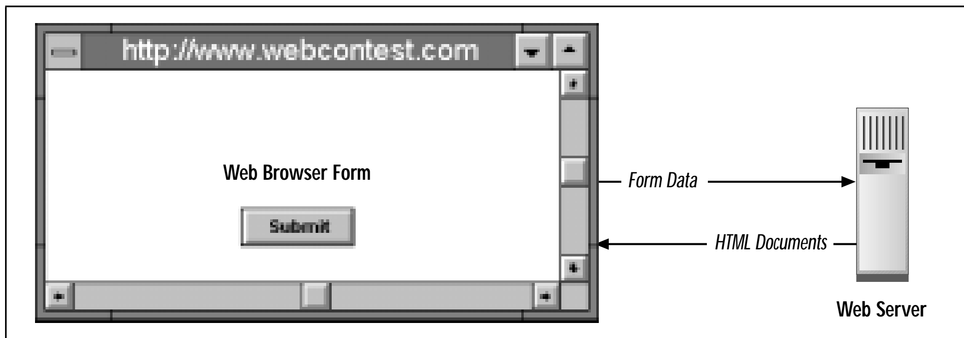

Figure 7-2. A contest registration application using a two-tier architecture

In addition to displaying the HTML forms you receive from the server, your client also accepts user input and sends it back to the server for storage. You might even build in some batch processing on the server to validate a contest entry and reject any ineligible ones. The application requires no additional processing in order to do its job.

The application can easily handle all its processing between the client and server; nothing calls for an additional layer. But what if the application were even more complex? Let's say that instead of waiting for ineligible entries to be submitted before rejecting them, you filter out ineligible entries on the client so people are not forced to fill out a form, only to be rejected. In place of handling entry processing in a separate application, you now need to write logic somewhere that rejects these rogue entries. Where are you going to do it?

Without Java, a client-side scripting language like JavaScript, or a peculiar browser plug-in, you can forget handling this processing on the client side. Processing in a web browser can happen using only one of those three solutions. The browser plug-in solution is unlikely since it requires the user to download and install a foreign application to simply fill out a one-time entry into a contest. Browser scripting languages, on the other hand, lack Java's portability and its ability to cleanly interface with server systems. The enterprise solution is Java.

Two-tier limitations

Using Java on the client, you can preserve the simple two-tier architecture. This solution is great if your processing is limited to simple data validation (Is your birthday a real date? Did you enter all the required fields?). But what if you add even more complexity? Now you require the application to be a generic contest entry application that you can sell to multiple companies running Internet contests. One of the implications of this generic design is that your application must be able to adapt to contests having different entry eligibility rules. If your eligibility rules are located in a Java applet on the client that is talking directly to the database, then changing eligibility rules essentially means rewriting the application! In addition, your direct database access ties your application to the same data model without regard for the individuality of the different contests the application is supposed to serve. Your needs have now outstripped the abilities of your two-tier architecture.

Fat clients. Perhaps you have seen the sort of scope-creep in a single application in the way I introduced new functions into the contest application. Ideally, the client/server architecture is supposed to let each machine do only the processing relevant to its specialization. Workstations are designed to provide users with a graphical interface, so they do data presentation. Servers are designed to handle complex data and networking management, so they serve as the data stores. But as systems get more complex, more needs appear that have no clear place in the two-tier model.

This evolution of application complexity has been paralleled by opposing trends in hardware. Client machines have grown larger and more powerful and server machines have scaled down and become less expensive. While client machines have been able to keep up with more complex user interface needs, servers have become less capable of handling more complex data storage needs. Whereas a single mainframe once handled a company’s most complex databases, you might find today the same databases distributed across dozens of smaller servers. As odd as this sounds, companies do this because it is immensely cheaper to buy a dozen small workstations than one mainframe. Financial pressures have thus pushed new processing needs onto the client, leading to what is commonly called the problem of the “fat client.”

Two-tier, fat-client systems are notorious for their inability to adapt to changing environments and scale with growing user and data volume. Even though a client’s primary task is the presentation of data, a fat client is loaded with knowledge completely unrelated to that task. What happens if you need to distribute your data across multiple databases or multiple servers? What happens if some small bit

of business logic changes? You have to modify, test, and redistribute a client program whose core functionality has nothing to do with the changes you made.

Object reuse. Object reuse is a very vague but central concept in object-oriented development.* You know it is a good thing, but you may have very different things in mind when you speak of it. In one common sense, object reuse is simply code reuse. You used the same code to build application X and application Y. But in another sense, object reuse means using exactly the same object instances in one application that you are using in another application. For instance, the customer account objects you likely built for an ATM system could be used by a new web interface you are building. While a two-tier system can contort to achieve the former sense of object reuse, it can almost never achieve the latter.

In the simplest form of object reuse, you would like to take code developed for one application, rewrite small bits of it, and have it run with minimal work. Two-tier solutions have had a nasty time doing this because they are so closely tied to the database. In PowerBuilder, for example, your GUI objects map directly to tables in a database! You need to throw away a large chunk of your GUI when moving from one environment to the next. Some very clever people-—including a few I have worked with-—have built complex class libraries to work around this problem. But my experience has been that such systems lack the flexibility you want in a toolkit of reusable objects.

Source code reuse is not the real object reuse you are looking for. You want to reuse actual object instances. If you look at building a system for viewing bank accounts both from the Web and an ATM, you really want the user's web browser and the ATM to look at the exact same data, especially if they are looking at that data at the same instant. Doing this with a two-tier system is nearly impossible since each client ends up with its own copy of the data. When one client changes some data, other clients end up with old data, resulting in a problem called *dirty writes*. A dirty write is a situation in which someone tries to modify data based on an old, out-of-date copy of the database. If my spouse makes a withdrawal at an ATM while I am paying bills at home, I want to see that change happen. If we each look at copies of the original data, however, I will end up paying a bill with money that my spouse just withdrew!

If a client, on the other hand, simply observes objects located in some centralized location, it always deals with the most recent information. When my spouse withdraws that last $100, my web browser is immediately notified so that I do not act on stale information.

* I am talking specifically about reuse in the development workflows of a project. The most effective reuse occurs in the analysis and design workflows of a project.

When to use a two-tier design

Two-tier solutions do have a place in application development. Simple applications with immediate deadlines that do not require a lot of maintenance are perfect for a two-tier architecture. The following checklist provides important questions to ask before committing yourself to a two-tier design. If you can answer "yes" to each of the questions in the checklist, then a two-tier architecture is likely your best solution. Otherwise you might consider a three-tier design.

- Does your application emphasize time-to-market over architecture?
- Does your application use a single database?
- Is your database engine located on a single host?
- Is your database likely to stay approximately the same size over time?
- Is your user base likely to stay approximately the same size over time?
- Are your requirements fixed with little or no possibility of change?
- Do you expect minimal maintenance after you deliver the application?

Three-tier

You can avoid the problems of two-tier client/server by extending the two tiers to three. A three-tier architecture adds to the two-tier model a new layer that isolates data processing in a central location and maximizes object reuse. Figure 7-3 shows how this new third layer might fit into an application architecture.

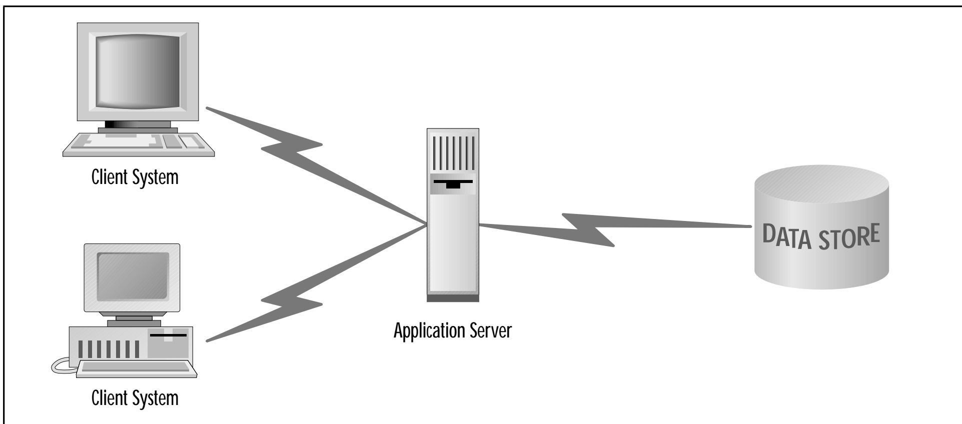

Figure 7-3. A three-tier architecture

Isolated database connectivity. I've already mentioned that JDBC frees you from concerns related to portability among proprietary database APIs. Unfortunately, it does not—it could not—provide us with a means to liberate your applications from your data model. If your application uses JDBC in a two-tier environment, it is still

talking directly to the database. Any change in the database ends up costing you a change in the presentation layer.

To avoid this extra cost, you should isolate your database connection so that your presentation does not care anything about the way you store your data. You can take advantage of Java's object-oriented nature and create a system in which your clients talk only to objects on the server. Database connectivity becomes an issue hidden within each server object. Suddenly, the presentation layer stops caring about where the database is, or if you are even using a database at all. The client sees only middle-tier objects.

Centralized business processing. The middle tier of a three-tier architecture—the application server—handles the data-processing issues you have found out of place in either of the other tiers. The application server is populated by problem-specific objects commonly called *business objects*. Returning to the banking application, your business objects are such things as accounts, customers, and transactions that exist independently of how a user might see them. An account, for example, is concerned with processing new banking transactions. It does not care whether an ATM is viewing it, a teller is viewing it at his or her console, or the bank is planning to allow users to access it from the Web. An application server layer will generally consist of a data store interface and a public API.

The data store interface is hidden from external objects. Internally, a business object has methods to save it to and restore it from a database. Information about how this happens is not available outside the object, and thus does not affect other objects. On the other hand, the business object does provide a limited, public interface to allow external objects access to it. In a two-tier application, your GUI would have displayed account information directly from the database. In a three-tier system, however, the GUI learns everything about an account from an account business object instead of from the database. If the GUI wants to know something it should not be allowed to know, the business object can prevent it. Similarly, if the GUI wants to make a change to that object, it submits that change to the object instead of the database.

These centralized rules for processing data inside business objects are called *business rules*. No matter what your problem domain is, the rules for how data should be processed rarely change. For example, no matter what application talks to your account objects, that application should not (unfortunately) allow you to write checks for money you do not have. It is a rule that governs the way the bank does business. A three-tier architecture allows you to use the same business rules across all applications that operate on your business objects.

Business object presentation. User interfaces are very ephemeral things. They change constantly in the development process as users experiment with them; their final appearance depends heavily on the hardware being used, the application's user base, and the purpose of the application in question. The presentation layer of a three-tier application should therefore contain only user interface code that can be modified easily on a whim.

Your banking application could use any of these different presentation layers:

- The teller window's console at the bank
- The ATM machine
- The web applet

Figure 7-4 shows how you intend to present an account from a teller PC.

Figure 7-4. An account as viewed from the teller console

Database vendors know that data presentation is a central requirement, and they have developed some fancy solutions for creating GUIs. The good ones are easy to use and produce nice-looking results, but since they are based on a two-tiered vision, they allow rules and decision making to leak into the presentation layer. For instance, take PowerBuilder, which has been on the leading edge of designing a rapid application development environment for building database frontends. It uses GUI objects called DataWindows to map relational data to a graphical user interface display. With a DataWindow, you can use simple drag-and-drop to associate a database column with a particular GUI widget.

Because a DataWindow maps the user interface directly to the database, it does not work well in a three-tier system where you map the user interface to intermediary business objects. You will, however, create a user interface library in Chapter 10, *The User Interface*, that captures the DataWindow level of abstraction in a way that better suits a three-tier distributed architecture. Instead of mapping rows from a database to a display area on a user interface, you will create a one-to-one mapping of business objects to specific GUI views of them.

Drawbacks to the three-tier architecture

While the three-tier architecture provides some important advantages, it is not without its downside. Chief among its drawbacks is the level of complexity it adds to a system. You have more distinct components in your system, and therefore more to manage. It is also harder to find software engineers who are competent in three-tier programming skills such as transaction management and security. While tools like the EJB component model aim to minimize this complexity, they cannot elimintate it.

The Network Application Architecture

The term Network Application Architecture (NAA) is one I coined to describe a specific kind of three-tier architecture I found works best for enterprise Java systems. Variations on it are certain to work, but its essential characteristic is that it is a distributed three-tier application architecture. Distributed three-tier architectures enable the three logical tiers of a three-tier architecture to be distributed across many different process spaces. Under a distributed three-tier architecture, for example, my database could be split into an Oracle database on the server named **athens** for my business data and an Informix database on the server named **carthage** for my digital assets. Furthermore, my product and inventory business objects could be located on a third server, and the marketing and finance business objects on a fourth.

Figure 7-5 illustrates the NAA. It treats the mid-tier business objects as a logical mid-tier that can be split into many different partitions. This split is entirely transparent to the other tiers in the architecture. The result is that you can move objects around at runtime to enhance system performance without worrying that a client is looking for a particular object on a particular machine.

This principal also applies to the database layer. The business objects do not know or care whether the underlying database is a single or a distributed database.

The NAA makes a joke of the concept "web-enabled." Because the NAA is not specific to any kind of presentation layer, it can support ultra-thin web clients or just moderately thin Java clients. If access to your business systems via interactive television is the next great wave, your system is already enabled to support it.

The NAA addresses issues other than simple application partitioning. It is also based on the principles of enterprise systems I enumerated in Chapter 1, *Java in the Enterprise*. The NAA is not only a distributed three-tier architecture, it is a distributed three-tier architecture based on the EJB component model and RMI/IIOP for distributed computing support. It mandates full support for internationalization and localization and assumes a hostile security environment by requiring encrypted communications.

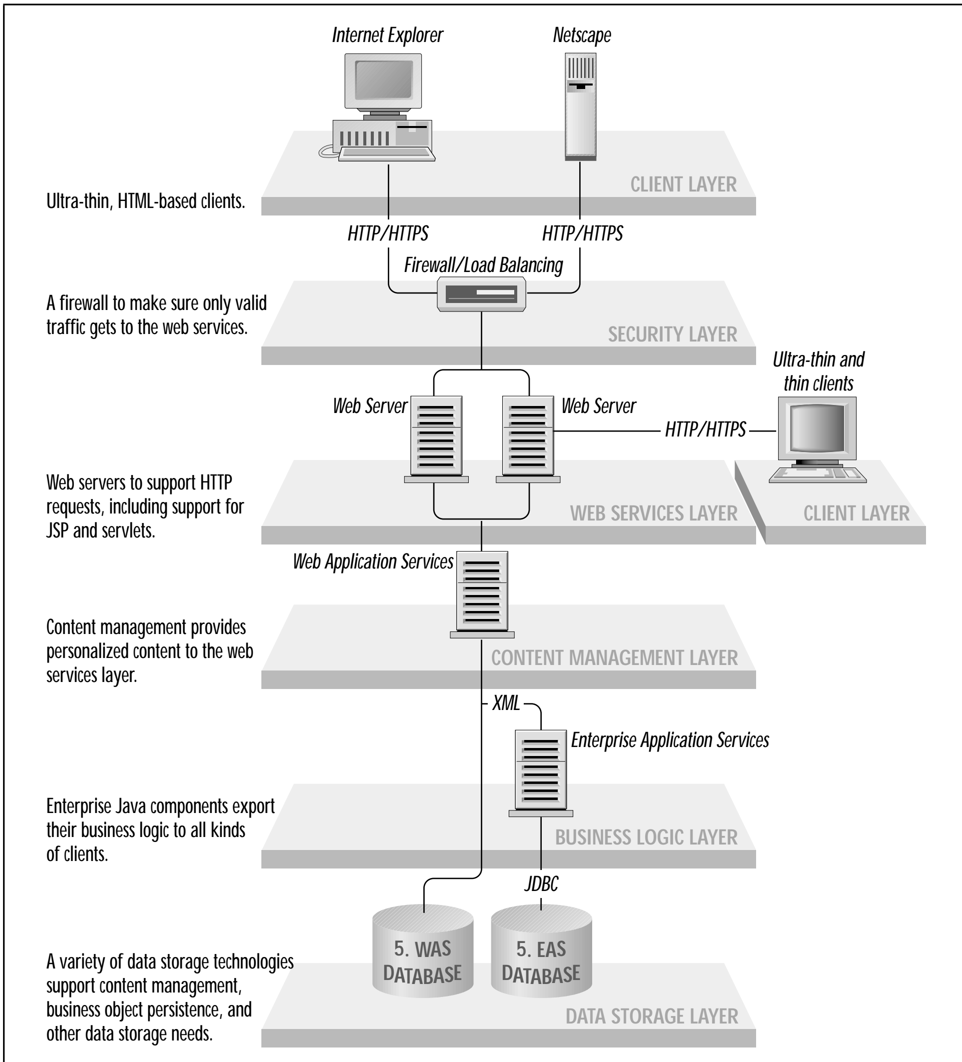

Figure 7-5. The Network Application Architecture

Design Patterns

Design patterns have been popularized by the book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch (Addison-Wesley). They are recurring forms in software development that you can capture at a low level and reuse across dissimilar applications. Within any application scope are problems you have encountered; patterns are the result of your recognizing those common problems and creating a common solution.

The first step to identifying the design patterns is identifying problems in generic terms. You already know that you need to relate GUI widgets on the client to business objects in the application server; you should create GUI widgets that observe changes in centralized business objects. You also know that you need to make those business objects persistent by saving them to a database. You should therefore look at a few common patterns that will help accomplish these goals.

Client Patterns

The user interface provides a view of the business specific to the role of the user in question. In the Network Application Architecture, it provides this view by displaying the business components housed on a shared server. Good client design patterns will help keep the user interface decoupled from the server.

The model-view-controller pattern

Java Swing is based entirely on a very important user-interface pattern, the *model-view-controller pattern* (MVC). In fact, this key design pattern is part of what makes Java so perfect for distributed enterprise development. The MVC pattern separates a GUI component's visual display (the view) from the thing it is a view of (the model) and the way the user interacts with the component (the controller).

Imagine, for example, a simple two-tier application that displays the rows from a database table in a GUI table display. The columns of the GUI table match the columns of the database table, and the rows in the GUI table match those in the database table. The model is the database. The view is the GUI table. The controller is a less obvious object that handles the user mouse clicks and keypresses and determines what the model or view should do in response to those user actions.

Swing actually uses a variant of this pattern called the *model-delegate pattern*. The model-delegate pattern combines the view and the controller into a single object that delegates to its model. Applied to the Network Application Architecture, a single model can model the business objects on the server to provide a common look for multiple GUI components. As a result, you could have a user interface with a tree and table showing two different views of the same objects for which deleting an object in the table is immediately reflected in the tree on the left without any programming effort.

The listener pattern

Perhaps saying that the tree knows about the change "without any programming effort" is a bit of an overstatement. The tree and the table may be sharing the same model, but the tree will not know to repaint itself unless it somehow knows that the model has been changed. Swing uses another key design pattern to make sure the tree knows about the change: the *listener pattern*.

The listener pattern enables one object to listen to specific events that occur to another object. A common listener in the JavaBeans component architecture is something called a `PropertyChangeListener`. One object can declare itself a `PropertyChangeListener` by implementing the `PropertyChangeListener` interface. It then tells other objects that it is interested in property changes by calling the `addPropertyChangeListener()` method in any objects it cares about. The import part of this pattern is that the object being listened to needs to know nothing about its listeners except that those objects want to know when a property has changed. As a result, you can design objects that live well beyond the uses originally intended for them.

The distributed listener pattern

A variation of the listener pattern is the *distributed listener pattern*. It captures situations in which one object is interested in changes that occur in an object on another machine. Returning to the banking application, my spouse at the ATM can withdraw money from a checking account that I am looking at on the Web. If the GUI widgets in my applet are not actively observing those accounts, they will not know that my account has less money than it did a minute ago. If the application uses the distributed listener pattern, however, anytime a change occurs in the checking account, both the ATM and the web browser get notified.

Distributed computing adds unique challenges to the distributed listener pattern. First of all, Java RMI's JRMP protocol cannot support the server calling a method in a user interface object for which that user interface is hidden behind a firewall. You can get around this problem by creating a special smart proxy object that acts like an RMI stub but is in fact a local user interface object. This stub polls the actual RMI stub for changes in the remote object. If it detects a change, it fires an event that local user interface objects can listen for. Figure 7-6 shows a class diagram detailing the object relationships. Figure 7-7 provides a sequence diagram to show what happens when a change occurs.

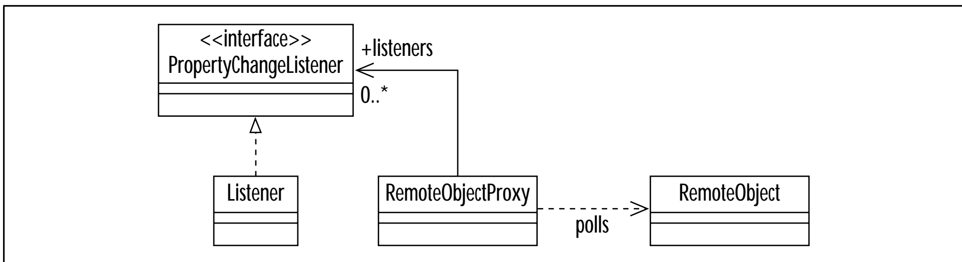

Figure 7-6. A UML class diagram of the distributed listener pattern

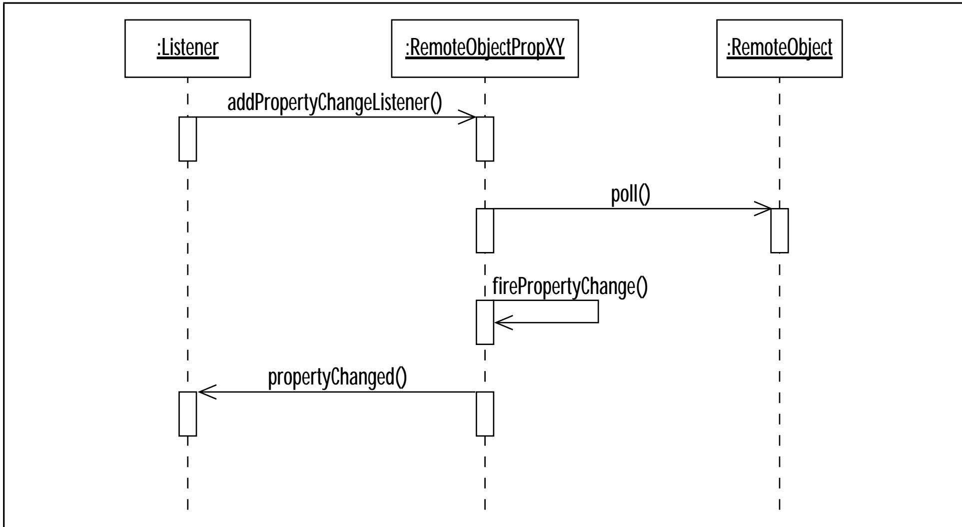

Figure 7-7. A sequence diagram describing sequences of events in the distributed listener pattern

Business Patterns

As a general rule, the mid-tier business logic is likely to use just about every design pattern in the *Design Pattern* book. The two most common general patterns I have encountered are the *composite* and *factory* patterns. I will provide a brief description of those two patterns here, but I strongly recommend the purchase of *Design Patterns* for greater insight into these and a wealth of other useful general patterns.

The composite pattern

The composite pattern appears everywhere in the real world. It represents a hierarchy in which some type of object may be both contained by similar objects and contain similar objects. Figure 7-8 shows a UML diagram describing the composite pattern.

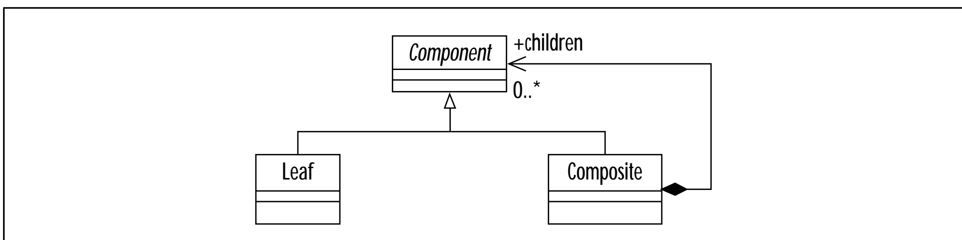

Figure 7-8. A class diagram of the composite pattern

To put a more concrete face on the composite pattern, think of a virtual reality game that attempts to model your movements through a maze. In your game, you might have a Room class that can contain Tangible objects, some of which can contain other Tangible objects (e.g., a bag). Your room is a container, and bags are containers. Things like stones and money, however, are unable to contain anything. The room, on the other hand, cannot be contained by anything greater than it. The result is the class diagram contained in Figure 7-9.

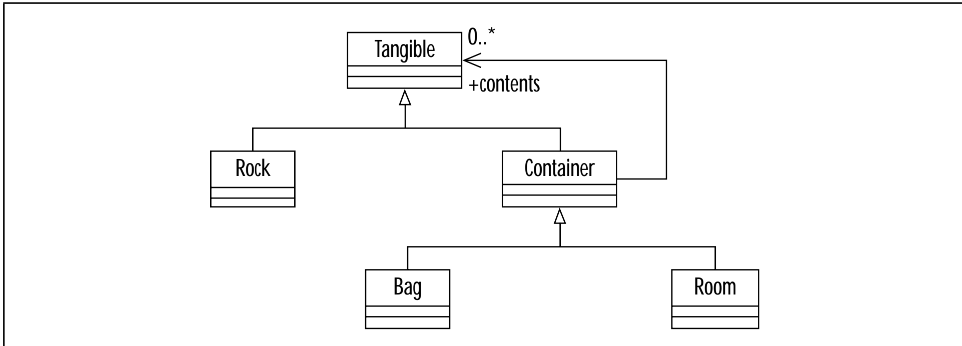

Figure 7-9. The composite pattern in practice

The factory pattern

Another common pattern found in the core Java libraries is the factory pattern. The `AppServerImpl` class from the previous chapter is an example of this pattern in use. The pattern encapsulates the creation of objects inside a single interface. The Java 1.1 internationalization support is peppered with factory classes. The `java.util.ResourceBundle` class, for example, contains logic that allows you to find a bundle of resources for a specific locale without having to know which subclass of `ResourceBundle` is the right one to instantiate. A `ResourceBundle` is an object that might contain translations of error messages, menu item labels, and text labels for your application. By using a `ResourceBundle`, you can create an application that will appear in French to French users, German to German users, and English to English users.

Because of the factory pattern, using a `ResourceBundle` is quite easy. To create a save button, for example, you might have the following code:

```
ResourceBundle bundle =
    ResourceBundle.getBundle("labels",Locale.getDefault());
Button button = new Button(bundle.getString("SAVE"));
```

This code actually uses two factory methods: `Locale.getDefault()` and `ResourceBundle.getBundle()`. `Locale.getDefault()` constructs a `Locale` instance representing the locale in which your application is being run. For a

French user, this `Locale` instance would represent France and the French language. For a German, it would represent Germany and the German language. The `ResourceBundle.getBundle()` method in turn finds a `ResourceBundle` instance that supports the language and customs for that locale. The French `ResourceBundle` will thus return "Enregistrer" for the `getString()` call and the English one would return "Save."

The goal of the factory pattern is to capture the creation logic of certain objects in one method. The benefit of providing a single location for the creation of certain objects is that you can handle any rules regarding the creation of those objects once. If you use `new` everywhere, however, a change in business rules will require a change and retest of every single `new` in your code.

Data Access Patterns

One key to smooth three-tier development is providing a clear division between data storage code and business logic code. At some point, a business object needs to save itself to a data store. You will use some key data access patterns to make sure that the business object knows nothing about how it stores itself to a database. It only knows that it is saving itself.

The persistence delegate pattern

To some degree, most applications are concerned about some sort of *persistence*. Persistence is simply the ability to have information from an application instance exist for later instances of the application (or even other applications) to use. An object-oriented database application uses a database to enable its business objects to exist beyond the traditional object lifecycle. You want your customer object to exist at least as long as the customer. You want the accounts to exist at least as long as they are open. The persistence pattern solves this problem by creating a single interface for any potential method of extending an object's life.

The simplest implementation of object persistence is the creation of flat files. Each time an object changes, it saves itself to a file. When your application recreates the object later on, it goes back to that file and restores itself. While this sort of persistence is useful for small systems, it is extremely inefficient and does not scale at all for big systems.

Another backend tool for object persistence is the relational database. Here you have arrived at your goal. Instead of saving to a file with each change in an object, your objects update the database. Although saving to a database is a lot different from saving to a file, the same basic concepts of saving, restoring, committing, and aborting apply to both. What differs is the system-dependent specifics; for instance, you execute a simple file write for a file save, but a SQL UPDATE for a

database save. You can therefore write a generic persistence interface that provides a single set of methods for persistence, regardless of whether you use a database or flat files. Your business objects never care how they are being saved, so the business logic just calls a `store()` method and allows a persistence-specific class to handle how that saving is implemented.

The persistence pattern defines an interface that prescribes these four behaviors:

`create()`
: Creates a primary key for the object and inserts it into the data store

`load()`
: Tells the object to load its data from the data store

`remove()`
: Removes the object from the database

`store()`
: Attempts to save any changes made to the object to the data store

The memento pattern

In the persistence delegate pattern, how does the persistence delegate know about the state of the object it is persisting? You could pass the object to the persistence methods, but that action requires your delegate to know a lot more about the objects it supports than you would likely want. Another design pattern from the *Design Patterns* book comes to the rescue here, the *memento pattern*.

A memento is a tool for capturing an object's state and safely passing it around. The advantage of the memento pattern is that it enables you to modify the beans and the persistence handlers independent of one another. A change to the bean does not affect any code in the persistence handler. The memento knows how to capture that change. The persistence handler knows how to get data from the memento. Similarly, any change in the way data is retrieved or saved to the data store is irrelevant to the bean. It always just passes its state to the delegate and lets the delegate worry about all persistence issues. I provide a concrete implementation of the memento pattern later in the book when I discuss address persistence.

The Banking Application

It is time to put everything together into a concrete banking application. This application is naturally not something your local bank would want to implement to support its business, but it does illustrate all of the key concepts I have covered in this book. The application is specifically a simple user interface that enables you to view your accounts and transfer money between any of them. It is a three-tier application that takes advantage of the Network Application Architecture, but it is

not EJB-based. You will therefore have to construct tools along the way that you get for free with EJB. As a result, you will get a very practical feel for all the issues involved in distributed software development that you can apply to both EJB and nonEJB development.

The Business Objects

With a distributed enterprise application, your starting point is the middle tier. The business classes that make up the middle tier are the key to a successful design. It is therefore the starting point for your development efforts.* Figure 7-10 is a UML class diagram describing the mid-tier business objects for the banking application. You will build these objects in Chapter 8, *Distributed Component Models*.

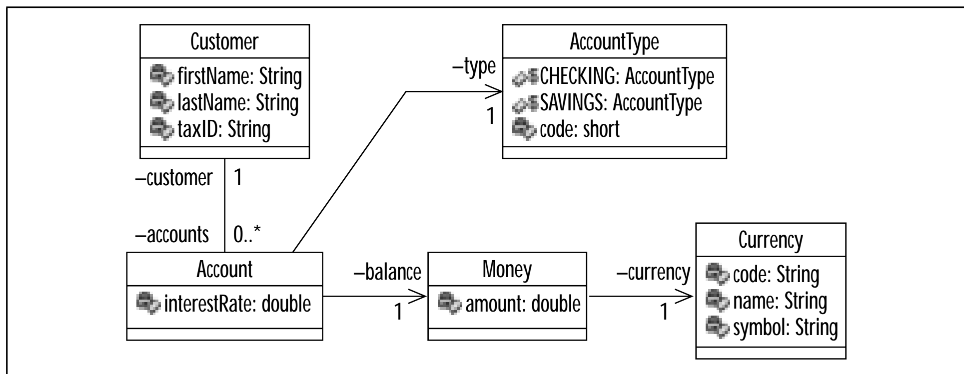

Figure 7-10. The middle tier of the banking application

Persistence

Persistence is a huge topic. A distributed application that persists its objects against a relational database has so many complex issues, such as security, locking, transaction management, object/database consistency, and performance to deal with. Enterprise JavaBeans worries about a lot of these issues for you. A book on database programming with JDBC and Java, however, would certainly be negligent if it did not address these problems. For the banking application, you will put together a library of objects that take care of these issues for you and apply it to the business objects in the banking application. Chapter 9, *Persistence*, covers the details of persistence in a distributed application.

---

* This fact is often in opposition to political considerations, namely that people want to see something. This urge is similar to asking a home builder to put up the siding on a house before building the frame, but it is a reality in software development today. My suggestion is to build time into your budget for a separate prototyping effort that involves a completely separate development team. That team can use rapid prototyping techniques both to provide some "feel good" enthusiasm and gather useful user interface (UI) design feedback for later stages of the system's design.

The User Interface

A good book on Swing, such as *Java Swing* by Robert Eckstein, Marc Loy, and Dave Wood, will tell you 99 percent of what you need to know about user interface programming in Java in a distributed environment. In Chapter 10 I focus on that extra one percent and the issues you need to deal with in Swing programming using the Network Application Architecture. The result will be the window in Figure 7-4 that enables users to view their accounts and make transfers.