

Better, Faster, Lighter Java™



O'REILLY®

Bruce A. Tate & Justin Gebtland

CHAPTER 3

Do One Thing, and Do It Well

There's only one way to eat an elephant: a bite at a time. That's also the best way to code. Each bite of clear, simple Java code must have a single purpose. The best Java programmers keep a maniacal focus on a single problem at a time and go to extraordinary lengths to maintain that focus. If you want to improve, emulate them.

I'm a whitewater kayaker. For a long time, I walked around every serious rapid that I faced. I could see how to run a one-shot waterfall or drop, but I couldn't get my head around the linked moves that would take me safely through Humpty-Dumpty on the Little River, or .25-mile Pine Creek on the Arkansas. A light clicked on for me on the Watauga River in Tennessee. I learned that I just couldn't bomb down continuous Class IV rapids with a preconceived set of moves in my head. Instead, I needed to find the natural break points *within* the rapids, and run many little ones. I learned to read the river and find the natural resting places within the whole. Then I could conquer one section, set up, and attack the next section. When I approached the problems this way, I received unexpected benefits. Coding, to me, is similar:

- It's usually easier to clearly define a piece of a big problem than the whole. We all tend to get overwhelmed by large problems, but not as much by many smaller ones. Our brains just work that way, whether we're on a river or behind a keyboard.
- When things go wrong, it's easier to adjust or adapt if your plan is segmented. Plans change; it's harder to change a grand, sweeping plan than several smaller ones.
- You can protect yourself from disaster. On the river, I now plan for safety one small section at a time. While coding, we must build test cases that identify problems quickly in each logical section.
- You can better reuse techniques and code. On the river, I learn new moves that I can use elsewhere. Instead of frantically paddling through a section, I use a draw stroke to avoid one rock, an aggressive brace and sweep to punch a hydraulic, and so on. In code, I build collateral and learn techniques to solve smaller, more

general problems. Each design pattern that you learn in context is worth any 20 that you read about.

Whether you're a kayaker running a Class V rapid, a physicist working on a space station, or a programmer dealing on a massive problem, your approach should be the same.

Understand the problem

You must understand a problem to solve it well. Effective programming goes beyond your code editor and debugger. You need to be able to accurately gather requirements, and control the scope and expectations of your customers. There is nothing worse than a solution without a problem; it just generates more problems.

Distill the problem to its essence

It's often said that programming is more of an art than a science. Nowhere is this axiom more obvious than in the ability to cut through clutter and find the core problem. You've got to recognize what belongs in the center of your problem space and what you can push out to other classes around the perimeter, or out of the project altogether. Most of the time, less is more.

Layer the architecture

If you're solving a difficult problem and you can only focus on one thing at a time, you must layer your architecture, with each layer handling one task. The broadest, most successful design patterns help you build effective, decoupled layers. Model-view-controller lets you design user interfaces with three layered components. Façades allow you to build clean interfaces between major layers in your application. If you like to study design patterns, pay attention to how often this theme arises.

Periodically refine your approach

Left to its own devices, software loses focus over time. You've got to make a concentrated effort to refactor, decoupling your software into autonomous layers. Automated tests will help you refactor with confidence and design decoupled code in the first place. Getting feedback from teammates and clients often helps make sure the approach still solves the right problems.

In this chapter, we explore each of these concepts in detail. You don't need to go through them all to see a difference in your programming—you can benefit from each technique individually. But combining them multiplies their impact.

Understanding the Problem

Communication is a huge part of programming. Writing better Java won't do you any good if you're building the wrong thing: you must understand before you can code. As you advance in your programming career, you'll find that more and more of your job demands effective communication. It doesn't really matter where your requirements come from—your team lead, an analyst, or the customer. In each case,

your first job is to accurately gather each requirement and focus your customer on what you can reasonably achieve with your resources at hand. Don't plow your customer under with enormous requirements and arcane design documents. If the customers can't understand your documents, they can't focus on the real issues. Like your code, keep your requirements simple and clear.

If you don't like communication, I've got news for you: things are going to get worse before they get better. In a global economy, you've got to be efficient. Increasingly, that means developers must handle more and more of the software development process. Out of college, I worked for the largest software company in the world. We had more testers than coders on a project (often, by a factor of 3 to 1), and teams of 10 planners supporting 40 developers. The overall effort was 200 developers strong. Now, that project might use 10 developers and half the original timeframe to develop the same piece of software. Since the development of better automation and unit testing, each modern developer must shoulder more of the testing load. Coders also do more design work and planning than ever before. However, experience shows that many of those developers are not equipped to handle many of the increased planning and analysis roles that they face.

Gathering Requirements

This book would not do its readers justice if we didn't talk about dealing with change from a process perspective. If you're one of those shops that tries to do more with less, you need to do two things in the planning process: first, weed out software requirements that don't contribute much to the final project, and second, weed out unnecessary work that supplements traditional development but does not contribute much to the overall content or quality of your code.

Many programmers believe that they need to build object-oriented design documents like class diagrams or object interaction diagrams in order to support development. The danger with this approach is that you can spend too much time maintaining your design documentation, and not enough time building working code. The best design resources are a lightweight requirements list, working code, and an on-site customer. A requirements document often consists of little more than single line items, each taking less than half a day to several days to complete. Often, you don't need formal project management software. I've managed requirements with expensive project management tools, word processor documents, and spreadsheets. Low tech usually works better than high tech.

Controlling Scope Creep

Once you've got requirements together, you'll want to keep a manageable scope. I often spend a good deal of my time improving communication and understanding at client sites. By far, the biggest problems I encounter come from controlling the scope

and expectations of the project. Often, the power comes from a pocketbook, so it's tempting for those in charge to try to add whatever features they want, whenever they want them. There's a necessary tension between productivity and change. Many teams succeed because they adapt to rapidly changing requirements, but just as many projects falter because teams fail to adequately control scope. This is the central problem in software development. The only way to effectively develop software is to succeed at solving this problem.

Managing good change

Iterative development is effective partially because it allows you to adapt. But regardless of the process that you use, your customers must understand the overall cost of change. Whenever someone requests significant changes, you can't threaten to beat them with a plastic whiffleball bat. You've got to be receptive but firm. Your first response should be something like, "We'll try to work it in. Let me tell you what it will cost." This type of answer immediately gives your customer positive feedback, and also lets them know that change affects the overall cost of the project. It doesn't matter who the customer is. Whether your project is internal or external, some facts never vary: development takes money, time, and manpower. You can only do so much with a given pool of resources.

Figure 3-1 illustrates the three ways you can react to expanded requirements:

1. Reduce the remaining scope by removing an unimplemented item of the same cost from this release.
2. Increase the time allotted: request more time, and ask for more money.
3. Increase your manpower: ask for more money, and increase your team size.

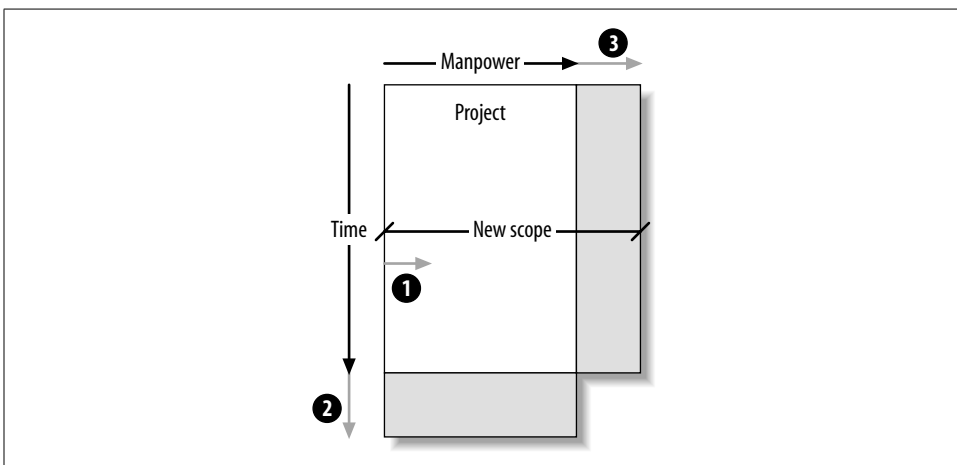


Figure 3-1. Reacting to expanded requirements

You can also combine two or more of these options. The first method is by far the best.

Notice that increasing hours is not a viable option. Heavy overtime means that the project managers are not doing their job. (Increasingly, the developer is the project manager!) Heavy overtime usually leads to defections, mistakes, and sloppiness that cost more in the long run. Nor should you increase your manpower in other ways. Unless you're grossly understaffed, increasing the size of an optimal team is going to sacrifice efficiency, and may even delay the project further. Scheduling conflicts also cause problems. If you frequently slip your schedules, you're delaying the value to your customers, and that's rarely a good idea.

By far, the preferred way to pay for a change in a release is to push other functions out of the release. If you get into the habit of negotiating a healthy give-and-take with your customers, they'll get better at understanding of what's most important to them, and you'll get accustomed to delivering on your commitments.

Curtailing disruptive change

The ability to react to your customers is important, but not all change is good. Disruptive change keeps your code out of the hands of your customers, and code that's in the lab doesn't do your customers any good. Disruptive change takes several forms:

- Unchecked scope creep can quickly derail any project. You must consciously mitigate each change in scope, and do so with minimal disruption to your team and schedule. That usually means pulling features out for each one that you add.
- Changes that are outside the central purpose of a project reduce focus. This kind of disruptive change often deals more with infrastructure and middleware than with end applications. For example, I have seen customers build messaging into their persistence frameworks, or business logic into their user interfaces. These types of short-term compromises usually backfire.
- Changes outside of the normal release schedules disrupt a team, and changes that happen very late in an iteration (for example, after a code freeze) are especially disruptive. In general, you improve your ability to change by shortening iterations rather than forcing unnatural change into an ongoing iteration. Figure 3-2 shows the cost of change at each step in the development cycle. The sawtooth pattern is characteristic of the cost of change in an iterative development process. It's better to have the customer wait for the next iteration.

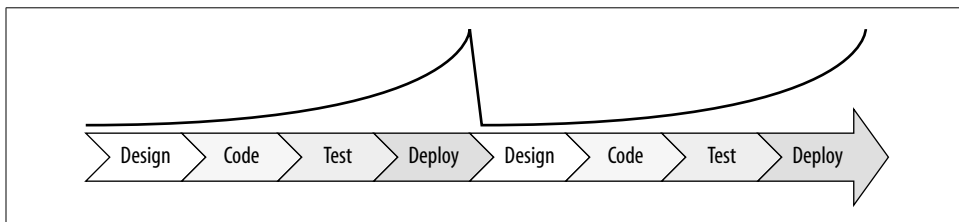


Figure 3-2. At the end of each cycle, especially after a code freeze, change gets more expensive

In general, certain types of changes cause more chaos than others: late changes, large changes, and shifts in focus. The reality, though, is that you're going to have change, and unless you're both disciplined and lucky, you're going to have to deal with some late change. The rest of this chapter, and this book, deals with adapting to change.

Distilling the Problem

Virtuosos in any profession have a common gift: they can distill a problem to its basic components. In physics, Einstein identified and captured many complex relationships in a simple equation, $e=mc^2$. Beethoven captured and repeated a motif consisting of four notes in his fifth symphony that's endured for centuries. Programming demands the same focus. You've got to take a set of requirements, identify the essential elements, strip away everything that doesn't belong, and finally break down and solve the problem.

To improve your programming, you don't have to live in a cave, reading about a design pattern that covers every possibility. You don't need to know the latest, hottest framework. You've just got to focus on the right problem, distill it to the basics, and hammer out the simplest solution that will work. In this section, I'm going to take a set of requirements, distill them, and turn them into code.

Collecting Requirements

Let's take a simple example. Say that you're building an ATM. Your job is to build the support for an account. In keeping with Agile programming practices, you've decided to keep a simple set of requirements in a table. You'll record the requirement number, a brief description, a size (timeline), and choose a programmer. Your team is small and your cycles are short, so that's all that you think you'll need. Table 3-1 shows the basic requirements.

Table 3-1. Requirements for account project

Number	Description	Size (hours)	Assigned
1	Keep a balance and account number		
2	Report a negative balance		
3	Have a six-digit account number		
4	Don't let users change the account number		
5	Remember the balance when the user comes back		
6	Let users make deposits		
7	Let users make withdrawals		
8	Display the balance		
9	Display debit/credit buttons		
10	Print a new balance when the user is done		

Table 3-1. Requirements for account project (continued)

Number	Description	Size (hours)	Assigned
11	Make the account secure		
12	Display the bank's logo		
13	Use lighter colors		
14	Let the users print out their balance		
15	Make the user type a four digit password		
16	Make the user insert a card		
17	It should be transactional (it all works, or it all fails)		

These requirements are typical of the types of things you'll get from your customers. They are far from complete, but that, too, is normal. The job of the requirements document is to accumulate requirements as you understand more about your application. At the moment, you're the one assigned to this task. You'll size tasks later. At this point, you should focus on the problem at hand. Your job is to build support for an account.

Whittling Away the Noise

Your first job is to whittle away some of the noise. Take any of the issues that may fit neatly elsewhere and push them out to the perimeter. Immediately, you recognize that you should separate the user interface from the base account. You also see that keeping a balance means that the account will need to be persistent. You'll use your relational database. Security probably doesn't belong in the account itself; it would probably be better left to another layer of the architecture, like perhaps a façade, but security is also a special case. Too many developers treat security as an afterthought, something they can sprinkle on top of an application to make it "safe." No such pixie dust exists, though; security layers should be treated with the same respect you show to everything else in your code.

Essentially, you need to build a persistent account. Rather than trying to build a design document, you'll start to code. It's a small enough problem to get your head around, and you can always refactor. In fact, you'll probably refactor several times as you think of ways to simplify and improve your design. The new set of requirements is shown in Table 3-2.

Table 3-2. Updated, whittled-down requirements

Number	Description	Size (hours)	Assigned
1	Keep a balance and account number	2	BT
2	Report a negative balance	1	BT
3	Have a six-digit account number	1	BT
4	Don't let users change the account number	1	BT

Table 3-2. Updated, whittled-down requirements (continued)

Number	Description	Size (hours)	Assigned
5	Remember the balance when the user comes back	4	BT
6	Let users make deposits	1	BT
7	Let users make withdrawals	1	BT
17	It should be transactional (it all works, or it all fails)	?	BT

Keep in mind what you're trying to accomplish. You're not discarding the rest of the tasks. You'll still need security and a user interface. Instead, you first want to carve out a manageable size of development. When that's completed and tested, you can go ahead and layer on the other aspects of the application, like the user interface and the persistence. Also, keep in mind that as an example, these requirements may have a finer grain than they would in a production project. When you've got a rough unit, you can start to code.

I'm going to omit the JUnit test cases to keep this example (and this book) brief, but I recommend that you code test cases first, as we did in Chapter 2. The next task is to rough out an interface. You don't yet need a full implementation. I recommend that you oversimplify, attaching everything to a single class until the ultimate design becomes clear. For this example, start with the implementation for requirements 1, 6, and 7, in a simple class called Account. Scoping and a simple constructor can take care of requirement 4. Stub out the rest with empty methods, so that you've got a simple interface.

Start by organizing in a package, and add a simple constructor:

```
package bank;

public class Account {

    float balance = 0;
    private String accountNumber = null;

    Account (String acct, float bal) {
        accountNumber = acct;
        balance = bal;
    }
}
```

Next, add the accessors for the members. Remembering your requirements, you want to keep the account number private, so you scope it accordingly, and omit the setter.

```
public float getBalance () {
    return balance;
}

public void setBalance(float bal) {
    balance = bal;
}
```

```

private String getAccountNumber () {
    return accountNumber;
}

public float debit(float amount) {
    balance = balance - amount;
    return balance;
}

public float credit(float amount) {
    balance = balance + amount;
    return balance;
}

```

Finally, add some stubs for methods that you'll need later. You may not decide to do things in this way, but it helps the ultimate design to emerge if you can capture some placeholders.

```

    public void save() {}
    public void load() {}
    public void beginTransaction() {}
    public void endTransaction() {}
    public void isValid(String accountNumber) {}
}

```

This is a reasonable start. You've covered requirements 1, 3, 4, 5 and 6, and you've got a head start on the rest. You're probably not completely happy with the design. It's already time to refactor. Since you've been writing unit tests all along, you can do so with confidence, knowing that your tests will let you know if you break anything along the way.

Refining Your Design

Sometimes, a metaphor can help you analyze your design. In this case, think of the job that we want the account to do. At least four things surface:

1. The getters and setters should tell you that this class holds data. That's a classic data access object, or value object, depending on the terminology you're used to.
2. The validation, debit, and credit methods should tell you that you're doing business logic.
3. The class saves and retrieves data from the database.
4. The `beginTransaction` and `endTransaction` suggest that you're also doing transactional processing.

Some of my past clients would have stopped designing at this point. If you're an EJB programmer, you're thinking that you've got a match: the class is transactional, persistent, and possibly distributed. Step away from that sledge-o-matic, and pick up a plain old ordinary hammer. It's time to break this puppy down.

Not many would complain if you suggested that it's a good idea to separate the business logic from the value object. Today, many modelers like to always separate value objects from the business domain. Persistence frameworks and other middleware made it easier to build systems that way. But designs are simpler and much easier to understand when you can leave them together.

Now, think about the save and load methods, as well as the transactional methods. Another metaphor is useful in this situation: think of a folder that holds paper. The paper represents your data and the folder represents a value object. Think of the save and load methods as filing the folder for later access. You would not expect the folder to be able to file itself. In principle, it makes sense to break the persistence methods away from the accessor methods and the business logic. For now, let's move the transactional methods with the persistence.

The result is clean, well-defined business logic, and a data access object (DAO) built explicitly to access the database. The DAO should be able to save and retrieve accounts. Here's the code to load an account using JDBC:

```
public static Account load(String acct) throws NotFoundException, SQLException {

    Account valueObject;
    ResultSet result = null;
    String sql = "SELECT * FROM ACCOUNT WHERE (accountNumber = ? ) ";
    PreparedStatement stmt = null;
    stmt = conn.prepareStatement(sql);

    try {

        stmt.setString(1, acct);
        result = stmt.executeQuery();

        if (result.next()) {

            account.setAccountNumber(result.getString("accountNumber"));
            account.setBalance((float)result.getDouble("balance"));
            return account;

        } else {
            throw new NotFoundException("Account Object Not Found!");
        }
    } finally {
        if (stmt != null) {
            stmt.close();
        }
    }
}
```

The save code is similar. It's a little ugly, but that's okay. You'll only be reading this code when you're interested in the database details. You'll be able to test the business logic of the account without wiring it to the data access object. You'll also be

able to add sophistication to the business logic without thinking about persistence, and you can change the persistence layer without impacting the business logic.

Consider transactions for a moment. Rather than bringing in the heavyweight artillery like JTA or EJB, start with the simplest solution. You can lean on the transaction support of your database engine and access it through your JDBC connection. That means the JDBC connection should probably be attached elsewhere, because you'll want all of your different data access objects to potentially participate in the same transaction. For example, if a user opened an account, you'd probably want to update the user and the first account deposit at the same time.

You know you need to refactor. Where's the correct place for the JDBC connection, and the associated transaction support? It's not in the account itself or the Account data access object. You'll need to create something, and you'll need some type of connection manager. If that strategy doesn't work out, you can always refactor again. Lean on these types of iterative refinements to improve your design as you progress.

Although this is a trivial example, it demonstrates how the process works. You write tests, code a little, refactor a little, and repeat the cycle until your eventual design emerges. After wading through these details, it's time to look at issues at a higher level.

Layering Your Architecture

This book is about building complex code in a simple way. The principle in this chapter, “Do one thing, and do it well,” may seem like it argues against building complex software. But it simply means each major piece focuses on a single aspect of the overall solution.

You can organize an application in layers, so that you'll only have to deal with small, focused chunks of the application at any given point of time. In this section, I'll talk about the anatomy of a layer, the interfaces between layers, and the typical layers that you're likely to find in better Java applications. Before I start piling on generic information about how you should build layers, here's what you should not do:

- Don't bite off too much in any given layer. Your layers should have a simple purpose, and they should be easy to digest at one sitting. If your layers are too fat, they'll be too hard to test, maintain, and understand.
- Don't add a layer because you read about it in a book. In most cases, you should add new layers and design patterns based on real, experienced need, not assumed need.
- If you're testing as you go, your tests will dictate some level of layering. Don't resist. Your tests mirror usage patterns in many ways, and they'll make your code easier to reuse and decouple as the need arises.

- Pay attention to names. Think of each independent layer as a library of services, even if there's just one client at the moment. If you misname your services, your layer will be misused. If a name becomes a problem, change it. If it's too hard to change names, get a tool that lets you do so. IntelliJ's IDEA was used to build some of the software in this book, and its refactoring tools are extremely useful.

All developers layer; some just do so more effectively and intentionally. In the last section, you probably noticed that I threw some requirements out to handle later. I did so because the postponed requirements were natural layers for the emerging architecture. I explicitly defined layers for the business domain model and the data access object. I implicitly defined a layer for the user interface, a potential façade, and security. Let's look at the anatomy of a layer.

The Basics of Layering

OOP makes effective layering much easier. Each layer should do one fundamental thing. There are many different layers, but they only do a few different types of things:

Abstraction

Often, a software layer does what you need it to, but it may have an interface that's complex or awkward for your application. An abstraction layer presents a simpler interface for its clients. Façade layers are abstraction layers.

Application

These layers do the specific work of an application. The Account object is a functional application layer.

Service

These layers are similar to application layers, but they provide common services potentially needed by many applications. The line between a service layer and an application layer can be blurry.

Interfaces Between Layers

Figure 3-3 shows how the layers fit together. Ideally, you want a strict hierarchy. Typical layers are clients of layers below, and provide services for layers above. The lower-level layer usually exposes and maintains the interface, but that's not always the case. When you expose an interface (like the darker areas of Figure 3-3), you need to harden it more. Peer layers are at the same level of abstraction and do roughly the same type of work. Peer interfaces might have a tighter level of coupling than other layers, because peers often share bi-directional interfaces. Often, you might not want that tighter coupling.

Think about the business model that we defined in our example above. The lower-level services that you use, those defined by the Java JDK, know nothing about our

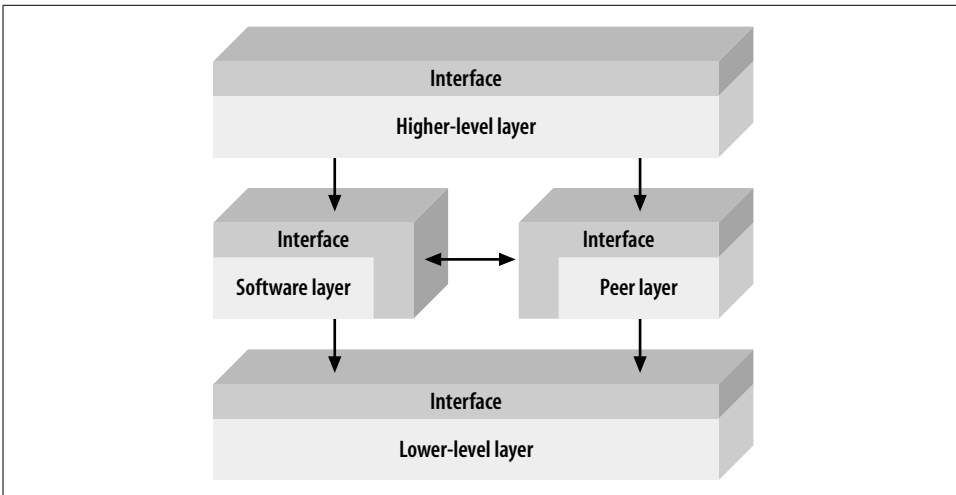


Figure 3-3. A typical intermediate software layer has layers above and below

account. The account uses JDK library classes like `String`. If the interface of the JDK changes, then you've got to move `Account` to change with it. The `Account`, on the other hand, knows nothing about the DAO layer. The `Account` DAO layer that saves and loads an account is a client of the `Account`.

Most people think about interfaces as two equal operators, like two people shaking hands with presumably the same equipment. Usually, that's not the case. Most interfaces impose a direction. One layer presents an interface, and the others use it. Your interface should simultaneously provide complete services to the consuming layer while protecting the inner workings of the presenting layer.

In general, you'd like your layers to observe a strict hierarchy. Lower-level layers should not have direct knowledge of higher-level layers. Peer relationships can be especially troublesome, and deserve strict attention.

A word about good interfaces

You can make your layers much more effective if you concentrate on building a good interface around them. An interface bears some resemblance to food: everyone knows when it's good and when it's not, although not everyone will agree; it also takes whole lot more skill to make it right than to consume it. You could write a whole book on interfaces and still not say everything that needs to be said, but here are some guidelines for good interfaces. They should be:

Complete

You should be able to use your layer to do everything that you need to do through your interface. Everything doesn't need to be easy, but everything does need to be available.

Compact

Don't allow unnecessary duplication in your interface. When you duplicate, you're more prone to error, and you create additional maintenance burdens. Don't build in future requirements until you need them; don't add too many convenience methods; don't expose private methods.

Convenient

It should make it easy to do the things that you do the most. Convenience and compactness are often at odds. You'll want to assure compactness first, then convenience. That doesn't mean that you can completely blow off convenience.

Hardened

An interface should protect you from incorrect usage. If a null value breaks your code, throw the appropriate exception.

Consistent and predictable

An interface should observe standards, and should do similar tasks in similar ways.

Well-factored

Often, a few smaller, more focused interfaces are better than one all-encompassing interface.

Interfaces can take many forms, but all should observe these basic principles. They apply to messaging models, distributed remote procedure calls, and basic method invocations, too. The need for good programming hygiene is magnified between major levels of your architecture. Two excellent books for good interface design are *The Pragmatic Programmer* by Andrew Hunt and David Thomas and *Effective Java* by Joshua Bloch, both published by Addison-Wesley.

Common Layers

In my classes, many of my students ask me what layers a Java application should have. I tell them that it depends on what you want to do. Java application architectures are converging around a common architecture in several critical places. In truth, many of those layers are unnecessary, or mandated by ineffective architectures like EJB CMP. In other places, certain effective layers, in the right circumstances, are beginning to emerge.

Business domain models

Some development shops are moving away from a classic business domain model. Frankly, for some applications (like those built to baby-sit big relational databases), I'd have to agree with their direction. When you're doing nothing but viewing and entering relational data, a full object-oriented model is overkill.

When you do have a legitimate need for an effective business model, it should usually be the center of your application. Build services and interfaces around that layer

to persist, display, notify, and manipulate that model. The model is too important, and often too complex, to clutter with other details—so transparency becomes extremely important.

Data access

Many strategies and frameworks exist to persist business objects. At the low end is a simple data access object implemented with JDBC. At higher levels, full persistence frameworks build in value-add services like lazy loading and caching across clusters. Since EJB CMP left such a bad taste in people's mouths for so long, Java developers are moving back toward simpler JDBC-based architectures, and also toward transparent persistence solutions such as JDO and Hibernate.

Communication

The era of CORBA, where a single business domain model was distributed across many different nodes, is dying rapidly. Instead, you're more likely to see strategic communication between hierarchical layers of an application (like session beans between an application server and a presentation server), and between major applications. As such, packaging a service with a technology like web services or JMS with XML is much more prevalent. Older systems come into play here; also, disparate technologies like Microsoft's .NET platforms are increasing in popularity.

Façades

A façade is often the primary client of your business model. Your goal is to provide a higher-level interface for the rest of the world. Before you add a façade layer, you must understand the value that it's providing. I've seen clients mirror the interface of their DAO layer, verbatim, within their façade. Façades are much more interesting when you're doing some consolidation, such as returning all members of a department across a distributed interface instead of making a separate round trip for each one.

For distributed architectures, you often want to present a simple, coarse-grained interface to clients, instead of a complex, complicated business domain model. The façade layer rolls many fine-grained methods up to a simpler interface.

If you're one of those developers who tend to believe everything written by a major software vendor, now would be a good time to pick up some rotten tomatoes and eggs. I'm going to get a little heretical. Your façade layer need not be distributed at all. You can simply deploy your presentation layer and your business layers on the same box. If your façade is not distributed, you probably don't need those session beans. And if you're not getting transactional integrity or security from your façade, you may not need a façade layer at all. You'll see more about the role of an effective façade in Chapter 4.

User interfaces

One of the most famous patterns for layering software is the model-view-controller architecture made popular by the Smalltalk community. Java model-view-controller architectures break user interfaces into three separate sections: a browser-based user interface, a controller, and a wrapper around the domain model.

Java developers fully embrace this concept and generally support an open source implementation called Struts to separate the user interface layers for a web-based application. Figure 3-4 shows the typical arrangement. A browser-based HTML interface calls a controller servlet, via HTTP. This servlet invokes an action, which wraps the domain model. Then, the controller calls a JSP, which compiles to a servlet, and returns a page to the client, possibly with results from the domain model.

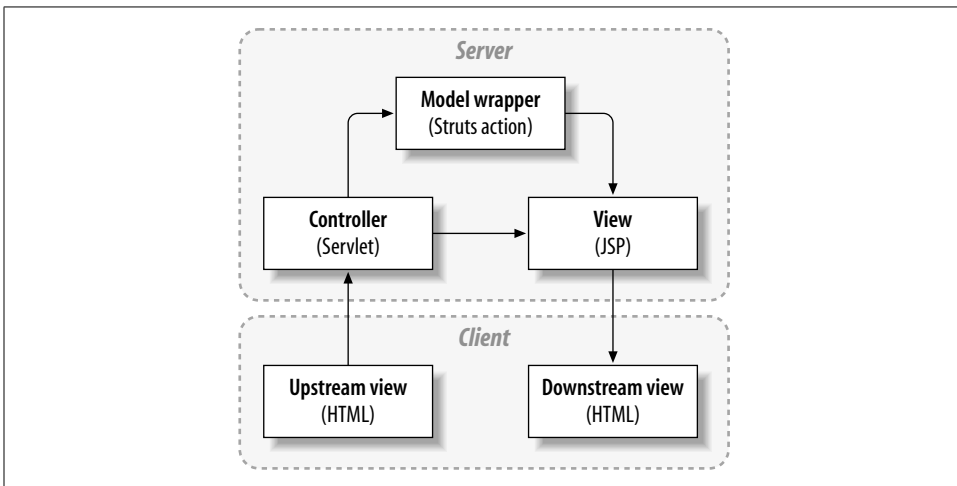


Figure 3-4. The model-view-controller architecture provides an approach to layering user interface code (this figure is a slight variation of the original, and supports web user interfaces via Struts)

In the near future, JavaServer Faces may present a more attractive option than Struts, because it supports more sophisticated user interfaces. Still, I'm skeptical about JSF for a few reasons:

- As with all event-based client-server models, JSF potentially increases communication costs between the client and server.
- JSF has been designed in a closed committee. That's often been a recipe for disaster.
- JSF is complex and will require specialized tools to be effective. The tools from some vendors tend to be proprietary, so it's important to watch the requirements evolve.

Starting with clean layers is only the first step. In the best of cases, you're going to want to refine your designs, and adjust your approaches. That's the subject of the next section.

Refactoring to Reduce Coupling

You may start with a cleanly defined design and you may layer your design, as we've discussed, so that each layer does one autonomous job. You may have coupling only at the appropriate places. But if you don't try to maintain that design, it won't last. Your code will naturally move toward tighter coupling unless you fight that tendency. In the last part of this chapter, we review some of the types of coupling, and how to avoid them. The benefits of looser coupling include:

- Decoupling protects major subsystems of your architecture from each other. If the coupling between your model and view are low, then changes in one will not severely affect the other.
- Loosely coupled code is easier to read. If you couple your business domain model to your persistence framework, then you need to understand both to read your domain model. Loosen the coupling and you can read your domain model unencumbered.
- Decoupling can improve reuse. It's harder to reuse big blobs of software, just like it's harder to reuse a full engine than a spark plug.

Keep in mind that some coupling is natural. You've got to have some degree of coupling to do anything at all. Coupling gets out of hand when things that don't belong together are bound together. Your goal should be to avoid accidental coupling—you want any coupling in your application to be intentional and useful.

Also, keep in mind that decoupling often comes at a price. You can add JMS queues and XML messages between every class, but you'll work harder and your application will be dog slow. Decoupling becomes much more important between major subsystems and layers of your architecture.

Microcoupling

Most of the code written today has many relationships, and most of those relationships are tightly coupled. Anything from a method call to the use of a common variable increases your coupling. Like I said earlier, that's not inherently bad. You just want to keep it intentional, and keep your coupling confined to an area of the architecture.

Your software design strongly suggests where your coupling should be. Excessive coupling across packages, and across layers—in general, excessive coupling across a focused idea—breaks down your ability to do one thing and do it well. When you find it, refactor it.

Direct access

The easiest type of coupling to find is direct access. When you directly call a method on another class or access its member functions, you're coupled to it. You can break coupling in a number of ways. When you're trying to loosen the coupling from two

classes, often the easiest way to is to insert some kind of intermediary. The Java programming language includes interfaces for this purpose. Bear in mind that interfaces are useless for decoupling unless they are paired with a factory. This line of code:

```
MyInterface myinterface = new MyObject();
```

is no less tightly coupled than this one:

```
MyObject myobject = new MyObject();
```

Whereas this one accomplishes the task and is completely decoupled:

```
MyInterface myinterface = MyFactory.getObject();
```

Think about cars. They work because they have combustion engines that drive axles, which spin wheels. In order to drive a car, you don't manipulate the engine and axles directly; you turn a steering wheel, press gas and brake pedals, and maneuver a shift. The steering wheel, pedals, and shift make up the interface to a car. There is a big difference between a '72 Beetle and a '04 Ferrari under the hood, but anybody can drive either because they share an interface.

An interface lets you couple to a capability rather than an implementation. Let's say that you're building classes that you'd like to fire when an event occurs. You could have your code explicitly call the fire method on all of the classes that you want to notify. This approach is limited to behavior that you can anticipate at compile time.

A slightly better approach is to build a class that supports the method fire. Then, everything that needs to be triggered can inherit from that class. That's the solution many novice Java developers use. It's limiting, because you may want to trigger other types of classes too. Instead, use an interface called `Firable`:

```
interface Firable {  
  
    public void fire();  
  
}
```

Notice that you don't see an implementation. Now, whenever you want a `Firable` class, you simply implement the interface:

```
public class AlarmClock implements Firable {  
  
    public void fire() {  
        System.out.println("Ring!");  
    }  
  
}
```

Now, other classes can use your "fire" method without coupling directly to yours:

```
public void wakeUp(Firable clock) {  
    clock.fire();  
}
```

The idea is to couple to an idea rather than an implementation. You don't want to build an interface that repeats every method of a class. Instead, break out the

concepts that you want to expose in the interface. If you find yourself addicted to JUnit as I have, you'll use this trick with some frequency. The nice thing about this approach is that you don't have to have any special behavior to test the alarm clock. You can also quickly mock a `Firable` class to help test code that fires your interface.

Interfaces serve as intermediaries, and you can decouple with other kinds of intermediaries as well. A façade is an intermediary that is nothing more than a thin wrapper. At first glance, you might think that you're trading coupling from one area of the application to the other, so you gain nothing at all. That premise is not entirely true. You'll see a few direct benefits:

- Your thin façade hides the details of the existing outbound interface. If the code it is wrapping ever changes, you can react to that change in the façade, leaving the other code intact.
- The façade is thin, and cheap. The code it is wrapping probably isn't. If you need to throw away the façade, you have not lost much.

You've probably seen other kinds of intermediaries as well. Rather than initialize a class with a new one, followed immediately by many sets, you can insert a constructor as an intermediary to enforce a policy for construction and consolidate several method calls. If you need to consistently call five methods to do a job, such as to establish a JDBC connection, you can wrap that code into a single method, or a class, like a connection manager.

Inheritance

Inheritance is one of the least understood mechanisms in modern programming. It is tempting to use the casual "inheritance is for is-a relationships," but this is just semantic handwaving. Everything "is-a" something else. Conceptually, there are two kinds of inheritance: *implementation* and *interface*. When a class inherits from another class and by doing so inherits actual implementation details (field values or code blocks), that is implementation inheritance. When a class implements an interface, thus promising to provide the services described there but without inheriting any specific values or code, that is interface inheritance.

In languages like C++, where multiple implementation inheritance is allowed, the problem can be quite severe. Classes that inherit from multiple direct parents can become logical Frankenstein's monsters, half-living beasts that don't quite look normal and never behave. Newer languages like Java solve part of the problem by eliminating multiple implementation inheritance. A Java class can have only one direct parent class (which in turn can have one direct parent, and so on). The chain is easier to follow and the results more predictable. However, classes can implement as many interfaces as they desire. Since interfaces do not impart specific implementation details to the implementer, just a public contract for services provided, the results are again easier to predict. Since any implementation code sitting behind an

interface is living in the class itself, there is never the question of hidden conflicts and accidental overrides creating random runtime behavior.

In order to decide which kinds of ideas require which kind of inheritance, it requires a little common sense and a little Zen meditation. When two or more classes represent specific categories of a single idea (Employees and Customers are both a kind of Person), then implementation inheritance makes sense. Person is a good candidate for a superclass. All logical children of that idea share the data and methods abstracted back to the Person object.

Interfaces, on the other hand, are useful for identifying services that cross-cut the logical model of the application. Imagine you are writing an application for a veterinary clinic. You might have two classes, Employee and Customer, which both inherit from Person. You might also have three other classes, Cat, Dog, and Bird, all inheriting from Animal. If they should be persistent, you can implement the PersistentObject interface as needed. The key is that each kind of person must be a Person; they need not necessarily be persistent. Each kind of animal must be an Animal, but they only *may be* persistent.

Transitive coupling

Keep in mind that coupling is transitive. If A is coupled to B, and B is coupled to C, then A is coupled to C. This type of coupling often seems innocuous, but it can get out of control in a hurry. It's especially painful when you're dealing with nested properties. Whether you have something like this:

```
store.getAddress().getCountry().getState().getCity()
```

or something like this:

```
address.country.state.city
```

you're building a whole lot of assumptions into a very small place. Dave Thomas, founder of the Pragmatic Programmer practice, calls this programming style the "Java train wreck." The worst form of the train wreck reaches into many different packages. The problem is that you're coupling all four classes together. Think of the things that might some day change. You might need to support multiple addresses, or international provinces instead of states.

Decouple this kind of code. You might decide to add some convenience methods for your customers or you might need to build a flatter structure, or even determine why you need access to the city in the first place. If it's to compute a tax, you might have a `getTax` method that isolates this coupling to one place. If it's because stores in certain cities have special attributes, you may add the attributes or methods to `Store` to loosen the overall coupling.

The role of transparency

Sometimes, you want to apply a little extra energy and completely focus certain pieces of code. For example, recall that we wanted to add security to our Account class without adding special security methods. We would do so with another layer. You would say that the Account class is transparent with respect to security.

Business rules often need special treatment, because they tend to be complex and tend to change with great frequency. Increasingly, leading edge Java developers look to find ways to isolate the business domain model from other concerns. Right now, the Java community is struggling to find the right way to package service layers, in order to keep business domain models fully transparent. Component architectures like EJB say that you should build your application as components and snap them into containers that provide the services. This architecture has tended to be too invasive and cumbersome. Instead, others say that services should be packaged as aspects, using a new development model called Aspect-Oriented Programming (see Chapter 11). As a compromise, many people are working to develop lighter containers that allow plain Java objects rather than special components. Pico and Spring (covered in Chapter 8) are two lightweight containers that are growing in popularity.

Testing and coupling

As you've already seen, your first defense against tight coupling is good, solid unit testing of bite-sized building blocks. As you code, you'll likely build an implementation, use that implementation in your code, and then reuse it again within your unit tests, as in Figure 3-5. Since you've built at least two clients into your development model and intend to test bite-sized pieces, you're much more likely to keep your coupling down to a level that's easy to manage. Further, your test cases will use each new class outside of its original context. With test-first development, you'll quickly understand where your coupling and reuse problems lie.

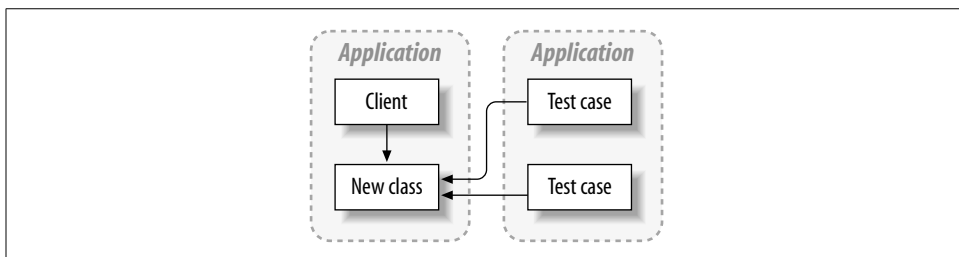


Figure 3-5. Testing offers the chance to have multiple clients for your new classes

Macrocoupling

Coupling at a higher level, or *macrocoupling*, is usually a much more serious problem than microcoupling because you want to keep each software layer as

autonomous as possible. For years, distributed technologies forced significant coupling, making many kinds of refactoring nearly impossible. Communication protocols forced clients and servers to manage intricate handshakes even to make a connection. Later, remote procedure call technologies forced clients to bind directly to a named procedure with a fixed set of parameters and fixed orders and types. CORBA took things a step further, and forced clients to bind to a whole specific object.

Today, you don't usually have to couple as tightly. A variety of technologies help build and connect independent systems. You can fight macrocoupling on several different levels.

Communication model

Your communication model can have a dramatic impact on the degree of coupling between your systems. Early in your design process, make some painless decisions that reduce the coupling between your systems:

- Prefer asynchronous messages where coupling is a factor. A one-time message using something like JMS generally requires less coupling than synchronous technologies like session beans or RMI.
- Prefer established standards to reduce dependencies. If you build a standards-based communication interface, you're placing fewer restrictions on either end of the wire.
- To reduce coupling, use a flexible payload. If you fix the number, order, or type of parameters, you're going to increase your coupling. Instead, try to use a more flexible payload, like a system using name-value pairs (such as a JMS mapped message) or an XML document.

Each of these techniques can reduce coupling, but remember that sometimes coupling is good. If you've got strict control of both ends of an interface, and if you don't expect the interface to change, then a tighter coupling can possibly buy you better performance. For the most part, however, it's usually worth it to pay a small performance penalty to reduce coupling from the beginning.

Façades

Façade layers don't really reduce your coupling. Instead, they let you couple to something that's a little less permanent. In addition, façades have some other important benefits:

- You can change a lightweight façade much more easily than you can change your business domain model.
- A façade lets you adapt your interface. You may want to translate value objects to XML documents to have a looser coupling over a distributed interface. A façade is a logical place for that conversion.

- A façade can let you build a coarse-grained interface to a fine-grained model. For example, instead of forcing your user interface to read each element of your invoice, you could return an entire invoice, such as the one in Figure 3-6.
- A façade provides a convenient attachment point for services. Transactions and security work well within a façade layer.

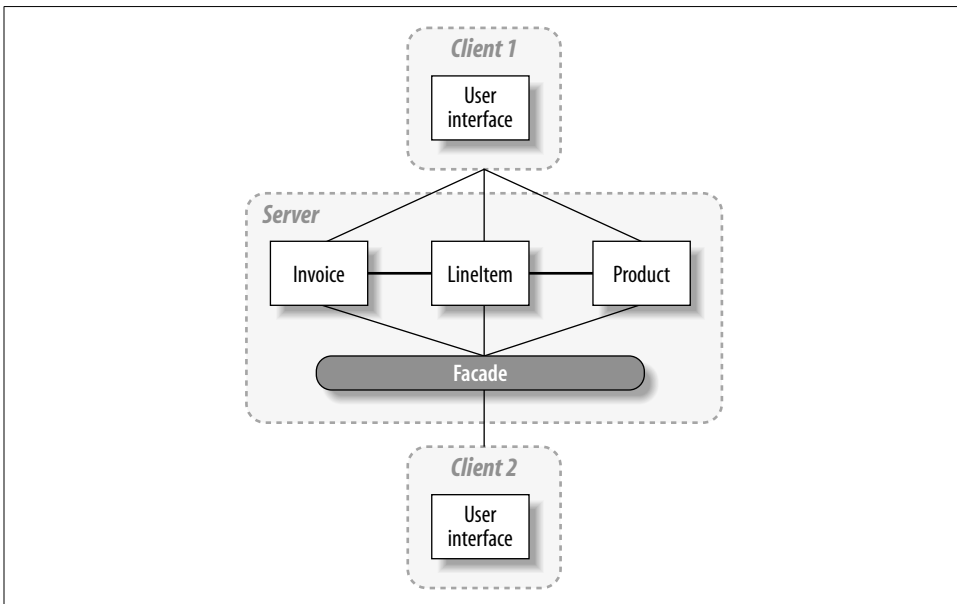


Figure 3-6. Client 1 must make four round-trip communications to the server; Client 2 reduces the total number of communications to one

Shared data

Applications that share interfaces usually need to share data as well. Whether you're using a buffer or a parameter list, the problem is the same. If you fix the number, type, order, or size of parameters, you're asking for trouble, because changes in the payload invariably force both sides of the interface to change. These strategies can help you to reduce coupling between subsystems that share data:

Allow optional parameters

When you don't control both ends of an interface, you need to have features that allow both sides to advance at their own pace. Optional parameters let you support a new feature without mandating immediate support on both ends of the interface.

Use name-value pairs

For interfaces that have a list of parameters, it's often better to name each parameter and pass them as a series of name-value pairs. Hash tables, maps, and

XML documents all let you build name-value pairs as input parameters. If you can support arbitrary text, you can handle XML. If XML seems to be overkill, JMS supports mapped messages, and you can use collections to handle this message type if you share memory addresses. The advantage of this approach is that applications need not depend on order, and optional parameters can be safely ignored.

Flexible data interchange formats are both a blessing and a curse. Your endpoints are more flexible in the face of changes to the payload, but it is more difficult to know exactly what is being shared. The more loosely typed your data interchange format, the more self-describing it must be. This is vital. If you pass name-value pairs, make sure that the consumer of the data can enumerate over both the values and the names. XML is a perfect format, since it is inherently self-describing.

Databases

The data access layer is one of the most problematic for Java developers to isolate. It doesn't need to be. Many good frameworks and solutions let you build an independent, transparent business model that knows nothing about persistence. Many persistence frameworks (such as Hibernate, JDO, and OJB) handle this well.

You must also ask whether you need a full relational database management system (RDBMS). Relational databases are large, expensive (in both resources and dollars) and complex. Sometimes, flat files are all that is needed. Make sure that you need what it is you are trying to wrap.

Regardless, you need not bite off a full persistence framework to solve a good chunk of this problem. You can build a lightweight DAO layer (like the one that we started for this chapter's example) to manage all data access for your application. There are a variety of IDEs and standalone tools that generate DAO layers automatically.

Configuration

Many times, you might want to avoid a particular standardized service to use a lighter, faster proprietary service. If you did so, you would have better performance and an easier interface, but you could be boxing your users into a corner. The makers of Kodo JDO faced that problem, and decided to make the service configurable. Increasingly, frameworks use configuration to decouple systems. Better configuration options invariably reduce coupling.

This list is far from exhaustive. If you want to excel at finding coupling problems, you've got to sharpen your observation skills. There's simply no substitute for reading code and watching the usage patterns, especially around the perimeter of a layer.

Summary

This chapter makes only one point: great software maintains focus on one task. To focus software, sharpen your ability to collect requirements and control your customers. If you're not careful, scope creep can confuse the basic theme of your software. When you've got a more complex problem, break each fundamental theme into a layer, or subsystem. In general, common layers are always evolving for Java technologies. Many of the accepted practices are sound, but others are suspect. Better layers share a common purpose and an effective interface.

Once you've designed effectively layered software and built clean software with a distilled purpose, maintain your clarity of purpose. To keep software focused on a central theme, you'll need to frequently refactor to loosen the coupling around tightly coupled components. Loose coupling is desirable at a lower level, and you can control it by testing and refactoring with techniques like interfaces. Also, pay attention to coupling at a higher level, so that each major subsystem is as isolated as possible. You'll improve reuse and isolate one subsystem from changes in others. In the next chapter, we'll discuss how to take the some extreme steps to reduce coupling between business domain models and services through increased transparency.