1

Bitter tales

This chapter covers

- A programming horror story
- Techniques for finding and fixing antipatterns
- Examples of antipatterns in other industries

On a cold day in Eastern Tennessee, my kayak is perched precariously atop a waterfall known as State Line Falls. The fall has a nasty reputation among kayakers. One of our team is walking this one. He was injured and shaken up last year at the same spot. This time around he wants no part of it.

From the top, there is no clue of the danger that lurks around the bend, but we know. We have been thinking ahead to this rapid for several days. We have read about what I cannot yet see. Five truck-sized boulders guard four slots. The water rushes through the slots and plunges to the bottom of the fall. I will see the entire waterfall only seconds before I go over it. Most of the water lands on boulders barely covered by two feet of water. Three of the four slots are reputed to be too violent and dangerous for your mother-in-law. Through the fourth, the river rips into the narrows and picks up speed. It drops sharply over the lip and crashes onto the jagged rocks 16 feet below. I am a programmer by trade, a father of two, and a kayaker of intermediate skill. I have no business going over a Class V waterfall described in guidebooks as "marginal." But here I am, looking for the landmarks. I pick my slot, sweep left, and brace for the soft landing—or the crash. I am in free fall.

1.1 A Java development free fall

The sales team was strong. They got all the right sponsors, lined them up, and marched them into the executive's office. They all said the same thing. The development cycle times were outrageous. Each project was longer than the last, and the *best* project overshot deadlines by 85 percent. It did not take the CIO long to add up the numbers. The cost overruns ran well into seven figures.

The answer was Java. The lead rep presented a fat notebook showing references from everywhere: the press, the news, and three major competitors. The proposed tools won awards and added to the outrageous productivity claims promised by even the most conservative vendors. They never cited the downside or training requirements. In early 1996, hardly anyone did. The sales team brought in the big gun: a proof-of-concept team that quickly hammered out an amazingly robust prototype in a short time. The lead rep had practiced the close many times, but in this case, the deal was already sealed. She was able to get even more of the budget than she expected. After all, a product and language this easy and this similar to C++ should not require much training, so she got most of that allocated budget too.

But a year and a half later, the lead programmer was sitting behind a desk in the middle of the night while the sales rep celebrated her third National Circle sales award in Hawaii. In truth, the programmer seemed genuinely happy to be there. He knew that he was in over his head, and he needed help badly. He could see that clearly now. When the project started, the programming team had just enough time to learn the syntax of the new language. They had been using an object-oriented language for four years without ever producing an object-oriented design. Their methodology called for one large development cycle, which provided very little time to find all of the mistakes—and even less time to recover. The insane argument at the time was that there was no time for more than one iteration.

As a member of the audit team dispatched to help the customer pick up the pieces, I was there to interview the programmer. My team had composed a checklist of likely culprits: poor performance, obscure designs, and process problems. We had written the same report many times, saving our customers hundreds of thousands of dollars, but the interviews always provided additional weight and credibility to back up our assertions.

"Is your user interface pure HTML, then?" I asked.

"Yeah," the programmer replied. "We tried applets, but that train crashed and burned. We couldn't deal with the multiple firewalls, and our IT department didn't think they would be able to keep up with the different browser and JVM configurations."

"So, where is the code that prints the returning HTML?"

He winced and said, "Do you really want to go near that thing?" In truth, I didn't want any part of it. I had done this long enough to know that this baby would be too ugly for a mother to love, but this painful process would yield one of the keys to the kingdom. As we reviewed the code, we confirmed that this was an instance of what I now call the Magic Servlet antipattern, featured in chapter 3. The printout consisted of 30 pages of code, and 26 were all in a single service method. The problem wasn't so much a bad design as a lack of any design at all. We took a few notes and read a few more pages. While my partner searched for the code fragment that processed the return trip, I looked for the database code. After all, I had written a database performance book, and many of the semiretired database problems were surfacing anew in Java code.

"Is this the only place that you connect to the database?" I asked.

"No," he answered. "We actually connect six different times: to validate the form, to get the claim, to get the customer, to handle error recovery, to submit the claim, and to submit the customer." I suppressed a triumphant smile and again reviewed the code. Connection pooling is often neglected but incredibly powerful. In chapter 7, the Connection Thrashing antipattern shows how a method can spend up to half of its time managing connections, repeating work that can usually be done once.

I also jotted down a note that the units of work should be managed in the database and not the application. I noticed that the database code was sprinkled throughout, making it difficult to change this beast without the impact rippling throughout the system. I was starting to understand the depth of the problem. Even though most of these audits were the same, at some point they all hit me in the face like a cold glass of water.

Over the next four hours, we read code and drew diagrams. We found that the same policy would be fetched from 4 to 11 times, depending on the usage scenario. (The caching antipatterns at this customer and others prompted discussions in chapter 5, where you'll learn about the caching and serialization techniques that can make a huge difference.) We drew interaction diagrams of the sticky stuff and identified major interfaces. We then used these diagrams to find iteration over major interface boundaries and to identify general chatty communications that could be simplified or shifted.

We left the customer a detailed report and provided services to rework the problem areas. We supplied a list of courses for the programmers and suggested getting a consulting mentor to solidify the development process. When all was said and done, the application was completed *ahead* of the revised schedule and the performance improved *tenfold*. This story actually combines three different customer engagements, each uglier than this one. I changed some details to protect the names of the guilty, but the basic scenario has been repeated many times over the course of my career. I find problems and provide templates for the solutions. While most of my peers have focused on design patterns, I find myself engaged with *antipatterns.*

1.1.1 Antipatterns in life

On the Watauga River, with all of the expectations and buildup, the run through State Line is ultimately anticlimactic. I land with a soft "poof" well right of the major turbulence. The entire run takes less than 20 seconds. Even so, I recognize this moment as a major accomplishment.

How could a journeyman kayaker successfully navigate such a dangerous rapid? How could I convince myself that I would succeed in light of so many other failures? I'd learned from the success and failure of those who went before me. The real extremists were those that hit rock after rock, breaking limbs and equipment, while learning the safest route through the rapid. I see a striking similarity between navigating rivers and writing code. To make it through State Line Falls, I simply did three things:

- *I learned to use the tools and techniques of the experts.* As a programmer, I attend many conferences to learn about best practices, and to find the new frameworks and tools that are likely to make a difference on my projects.
- *I did what the experts did.* I learned the easiest line and practiced it in my mind. We can do the same thing as programmers, by using design patterns detailing successful blueprints to difficult architectural problems.
- *I learned from the mistakes before me.* The first time down a rapid, it's usually not enough to take a good plan and plunge on through, torpedoes be damned. Good plans can go bad, and it's important to know how to react when they do. As a programmer, I do the same thing. I am a huge fan of "merc talk," or the stories told around the table in the cafeteria about the latest beast of a program. This is the realm of the antipattern.

When I was told how to run State Line Falls, I asked what-if questions. What should my precise angle be? How can I recover if I drift off that angle? How far left is too far? What's likely to happen if I miss my line and flip? I got answers from locals who had watched hundreds of people go down this rapid with varying degrees of success. The answers to these questions gave me a mental picture of what usually happened, what could go wrong, and what places or behaviors to avoid at all cost. With this knowledge, I got the confidence that it took to run the rapid. I was using design patterns and antipatterns.

1.2 Using design patterns accentuates the positive

Design patterns are solutions to recurring problems in a given context. A good example is the Model-View-Controller design pattern introduced in chapter 3. It presents a generic solution to the separation of the user interface from the business logic in an application. A good design pattern should represent a solution that has been successfully deployed several times. At State Line Falls, when I read about the successful line in guidebooks and watched experienced kayakers run the rapid, I was essentially using design patterns. As a programmer, I use them for many reasons:

- *Proven design patterns mitigate risk.* By using a proven blueprint to a solution, I increase my own odds of success.
- *Design patterns save time and energy.* I can effectively use the time and effort of others to solve difficult problems.

- *Design patterns improve my skill and understanding*. Through the use of design patterns, I can improve my knowledge about a domain and find new ways to represent complex models.

Embracing design patterns means changing the way we code. It means joining communities where design patterns are shared. It means doing research instead of plowing blindly into a solution. Many good sources are available.

Books

This is a sampling of books from the Java design pattern community and the definitive source for design patterns (*Design Patterns: Elements of Reusable Object-Oriented Software*). As of this writing, five or more are under development, so this list will doubtlessly be incomplete. Amazon (http://www.amazon.com) is a good source for finding what's out there.

- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (The Gang of Four)
- *Refactoring: Improving the Design of Existing Code*, by Martin Fowler, Kent Beck (contributor), John Brant (contributor), William Opdyke, and Don Roberts
- *Core J2EE Patterns*, by John Crupi, Dan Malks, and Deepak Alur
- *Concurrent Programming in Java: Design Principles and Patterns*, by Doug Lea
- *Patterns in Java, Volume 3: A Catalog of Enterprise Design Patterns Illustrated with* UML, by Mark Grand
- *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*, by Bruno R. Preiss
- *Java Design Patterns: A Tutorial*, by James William Cooper

1.2.1 Design patterns online

Manning Publications has a series of author forums for discussion. These authors discuss server-side architectures, Java programming techniques, Java Server Pages (JSPs), Extensible Markup Language (XML), and servlets. The author of this book also has an online community to discuss Java antipatterns.

Manning authors

- Manning author forums: http://www.manning.com/authoronline.html
- Java antipatterns: http://www.bitterjava.com

Java vendors

- IBM: http://www-106.ibm.com/developerworks/patterns/
- Sun: http://java.sun.com/j2ee/blueprints/

1.2.2 UML provides a language for patterns

The design pattern community has exploded in recent years partially because there is now a near universal language that can be used to express patterns. Unified Modeling Language (UML) brings together under one umbrella several of the tools supporting object-oriented development. Concepts such as scenarios (use cases), class interactions (class diagrams), object interface interaction (sequence diagrams), and object state (state diagrams) can all be captured in UML. Though this subject is beyond the scope of this book, there are many good UML books, tools, and resources as well.

Books

- *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, by Martin Fowler and Kendall Scott
- *Enterprise Java with UML*, by C. T. Arrington
- *The Unified Modeling Language User Guide*, by Grady Booch, et al.

Tools

- Rational: http://www.rational.com
- Resource center at Rational: http://www.rational.com/uml/index.jsp
- TogetherJ from Together Software: http://www.togethersoft.com

1.3 Antipatterns teach from the negative

AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis by William J. Brown, et al., is an outstanding book dedicated to the study of antipatterns. The antipattern templates that follow each chapter in this book come from Brown's text. In it, the authors describe an antipattern as "a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences." The words that caught my attention are *commonly occurring solution* and *decidedly negative consequences.* Many others have presented some of the negative examples in this book as the right way to do things. Some, like the Magic Servlet, are forms of programs published in tutorials, created by wizards, or captured in frameworks. As for negative consequences, anyone who has followed software engineering closely knows that a high percentage of software projects fail. The *AntiPatterns* text cites that five of six software projects are considered unsuccessful. Java projects are not immune. Earlier this weekend, I heard about a canceled Java project using servlets and JSPs at a Fortune 100 company that will be replaced with a new project using CICS and C++!

Some of the madness in our industry is caused by outright malice. Some vendors sell software that they know isn't ready or doesn't work. Some managers resist change and sabotage projects. Some coworkers take shortcuts that they know they will not have to repair. Most of the time, though, it is simple ignorance, apathy, or laziness that gets in the way. We simply do not take the time to learn about common antipatterns. Ignorant of software engineering history or the exponentially increasing cost of fixing a bug as the development cycle progresses, we might kid ourselves into thinking we'll take a shortcut now and fix it later.

1.3.1 Some well-known antipatterns

As programmers, we will run across many antipatterns completely unrelated to Java. For the most part, we will not go into too many of them, but here are a few examples to whet your appetite:

- *Cute shortcuts*. We've all seen code that optimizes white space. Some programmers think that the winner is the one who can fit the most on a line. My question is, "Who is the loser?"
- *Optimization at the expense of readability*. This one is for the crack programmers who want you to know it. In most cases, readability in general is far more important than optimization. For the other cases, aggressive comments keep things clear.
- *Cut-and-paste programming*. This practice is probably responsible for spreading more bugs than any other. While it is easy to move working code with cut and paste, it is difficult to copy the entire context. In addition, copies of code are rarely tested as strenuously as the originals. In practice, cut-and-paste programs must be tested *more strenuously* than the originals.
- *Using the wrong algorithm for the job*. Just about every programmer has written a bubble sort and even applied it inappropriately. We can all find a shell sort if pressed, and if we understand algorithm analysis theory, we know that a bubble sort is processed in $O(n^2)$ time, and a simple shell sort is processed in $O(n\log(n))$ time, which is much shorter for longer lists.

- *Using the wrong class for the job.* In object-oriented languages, we've got to choose between classes like tables and arrays that have similar function but different characteristics. If our algorithm calls for random access of a collection, using a b-tree or hash table will be much faster than an array. If we're going to frequently index or enumerate the collection, an array is faster.

1.3.2 Antipatterns in practice

The study and application of antipatterns is one of the next frontiers of programming. Antipatterns attempt to determine what mistakes are frequently made, why they are made, and what fixes to the process can prevent them. The practice is straightforward, if tedious. The benefits are tremendous. The trick to the study of antipatterns is to:

1. *Find a problem.* This might be a bug, a poor-performing algorithm, or unreadable method.
2. *Establish a pattern of failure.* Quality control is a highly specialized and valued profession in manufacturing circles. A good quality engineer can take a process and find systemic failures that can cost millions. Software process can create systemic failure, too. The Y2K bug was a systemic failure of a very simple bug that was created and copied across enterprises hundreds of millions of times. Sometimes, the pattern will be related to a technology. Most often, process problems involve people, including communications and personalities.
3. *Refactor the errant code.* We must of course refactor the code that is broken. Where possible, we should use established design patterns.
4. *Publish the solution.* The refactoring step is obvious but should be taken a bit further than most are willing to go. We should also teach others how to recognize and refactor the antipattern. *Publishing the antipattern is as important as publishing the related solution.* Together, they form a refactoring guide that identifies the problem and solves it.
5. *Identify process weaknesses.* Sometimes, frameworks or tools encourage misuse. Other times, external pressures such as deadlines may encourage shortcuts. We must remember that a process must ultimately be workable by imperfect humans. In many cases, education may be the solution.
6. *Fix the process.* This is the most difficult, and most rewarding, step. We effectively build a barrier between our healthy enterprise and the

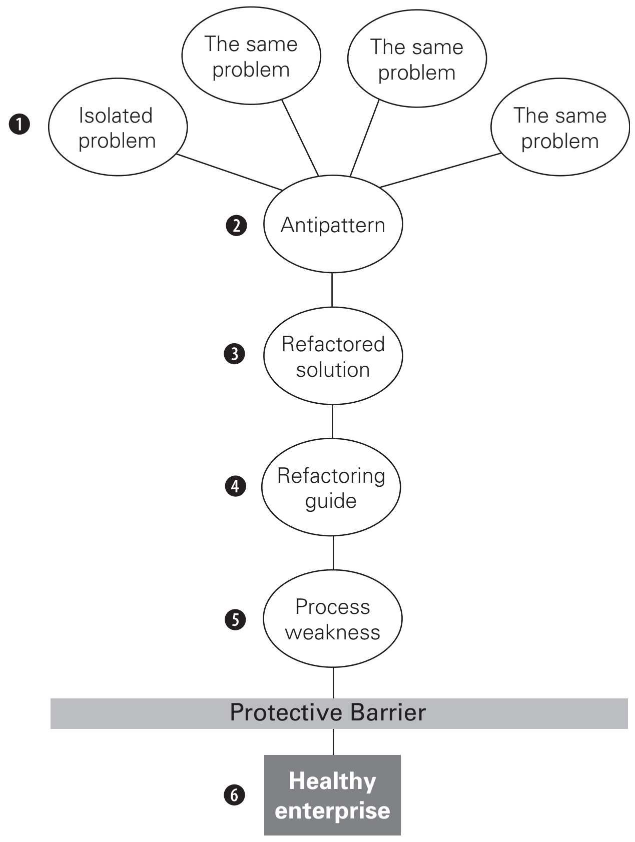

Figure 1.1 The antipattern process involves finding a problem ❶, establishing a pattern and publishing an antipattern ❷, refactoring the solution ❸, building a guide so that the problem can be resolved and fixed en masse ❹, identifying process weaknesses ❺, and building a barrier between the healthy enterprise and the antipattern ❻.

disease. Here, we take a hard look at what's broken. In simple cases, we fix the problem. In more extreme cases, we might need to establish a risk/reward analysis and win sponsorship to fix the problem.

Figure 1.1 illustrates the antipattern process.

1.3.3 Antipattern resources

The antipattern community is gathering momentum, looking for things that break in a methodical way and capturing those experiences. Some engines use pattern recognition to find bugs from software *source code*. Many programmers are starting to publish *bug patterns* for common programming mistakes. The http://www.bitterjava.com site has some links to Eric Allen's series "Bug Patterns."

The design pattern community also has a counterpart: the *antipattern* community. This group is interested in learning from common experience and capturing that knowledge in a uniform, methodical way.

AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis brings these concepts together better than any other source I have seen. With Brady Flowers, who contributed the Enterprise JavaBeans (EJB) examples for this book, I had started to do bitter Java sessions at conferences before we found *AntiPatterns*. When we found it, we immediately fell in love with the ideas expressed in this book. Most of the book's antipatterns went beyond theory and explained the cultural conditions prompting a problem. The book is extraordinarily useful to programmers who strive for excellence. We hope to take these concepts into the Java community to continue the momentum that *AntiPatterns* has created. We will go beyond generic antipatterns and dive into those that are most prevalent to the Java community. These are some online resources for antipatterns:

- The authors have an online source for Java antipatterns. You can find it at http://www.bitterjava.com. On the site, we will attempt to provide you with articles, discussion boards, and useful links.
- The http://www.antipatterns.com site has articles, events, and message boards.

1.4 Antipattern ideas are not new

Should developers spend more time on the study of antipatterns or design patterns? I will answer this with another true adventure story. Throughout the better part of this past century, mountain climbers across the world had an ultimate goal: to scale Mt. Everest, the highest summit in the world. Over time, mountaineers tried many different approaches that would allow political passage to the mountain, solid expedition logistics, and the best chances for success. Two routes go through Tibet. George Mallory was an early British mountain climber, famous for saying he climbed Everest "Because it is there." He made his attempts on the north face, over terrain called the North Col. The other northern route was considered much too dangerous for early mountaineers. Edmund Hillary, who became the first to climb Everest, eventually succeeded on the southern face, through Nepal. That route is called the South Col route. After the first ascent, expeditions climbed this dangerous mountain with greater regularity and greater margins of safety. They began to unlock the secrets of operating at high altitude and to find where the inevitable danger spots were likely to be. They began to understand when the summer monsoons directed the jet stream away from Everest to provide a window of acceptable weather. They learned to leave their tents at midnight so that they would not be trapped on the summit in the afternoon, when the weather frequently deteriorated. They were using design patterns.

Inevitably, people started to guide trips up the mountain with increasing success. Many debated that some of the paid clients did not have the appropriate skills to be on the mountain and would not be able to handle themselves in the event of an emergency. These criticisms turned out to be prophetic. Two expeditions led by the strongest guides in the world got trapped at the top of Everest through a series of poor decisions and bad luck. An afternoon storm killed many of them, including three of the six guides and several of the clients. Jon Krakauer made this incident famous in the book *Into Thin Air*. The design patterns were able to get them to the top but were unable to get them safely back down. Good application of climbing antipatterns, like avoiding the top of the mountain at dangerous times and holding fast to a prescribed turnaround time, could have made the difference.

1.4.1 Learning from the industry

In many real-world situations, the principles of design patterns and antipatterns are combined. In heath care, aggressive preventive care (design patterns) is combined with systematic diagnostics of health-related issues (antipatterns). In manufacturing, quality certification programs like ISO 9000 (design patterns) are combined with aggressive process analysis, problem identification, and continuous improvement (antipatterns). Road signs are combined to point out good driving behaviors like "Pass on left" and hazards like "Watch for falling rock." In many other fields, the two practices go hand in hand. Software engineers should try to combine these two approaches.

A powerful movement in the quality industry, from the late '70s through the '80s, sought to involve front-line assembly workers in the quality process. These teams were tightly integrated with quality professionals. The teams, sometimes with light-handed management direction, would identify problems and contribute a block of time weekly toward solutions to those problems. My father, Robert G. Tate, Jr., became so attached to this process that he left a high-level position at Dover Elevators to pursue a consulting career installing "quality circles" around the world. He found that something magical happened with the involvement of the actual blue-collar plant floor. The relationships changed. Management, quality control, and the product builders began to work together. The process was remarkably simple:

- Quality circles would form for the purpose of solving quality problems.
- Participants would become involved in the identification and solution of quality problems.
- Management would empower them to deal with quality problems directly.
- Participants were educated to perform these tasks.

Many of the quality groups showed staggering returns. Other programs, such as Zero Defects, also thrived. Awards and accreditations, like Malcolm Baldrige and ISO 9000, gathered steam. The United States again discovered the value of quality.

In a very real sense, this book represents the same ideas that we see in other areas and places them in the context of Java application development. We are taking responsibility for bringing quality code to the desk of the common programmer. We want to identify places where our assembly line is broken. We want to spot holes in process and procedure that can cripple our customers or even ourselves down the road. We want to *know* when major systematic problems, like the routinely late turnaround times on Everest, occur. We then want to systematically solve them and save others from repeating our mistakes. Most of this book deals with antipatterns that are already well entrenched in Java programs, processes, and programmers. We should now talk briefly about the discovery process.

1.4.2 Detective work

Experienced, conscientious programmers find most antipatterns. While teaching the instincts of a detective may be difficult, I can provide some rules of thumb from my consulting experience. These tips represent the places and methods that I use to find antipatterns hiding in a customer's process, or my own.

Bug databases contain a bounty of wealth

Most organizations already track quality metrics in the form of bug databases. We can look to establish patterns based on keyword searches and spot checks. Are we seeing a pattern of memory leaks? If so, misconceptions or frameworks could be a source of bad behavior. Are the change lists for view-related maintenance particularly long? If so, this could point to tight coupling. Are certain objects or methods particularly vulnerable to bugs? If so, they might be refactoring targets.

Early performance checks can point out design flaws

Sanity checks for performance early in a process can point to design flaws. Some of these might be isolated incidents. Some, even at an early stage, are likely to be common enough to warrant special attention. Internet applications are particularly vulnerable to communication overhead. Several of the antipatterns in this book deal with round-tripping, or making many communications do a relatively isolated task. Sloppy programming, including many of the issues in chapter 9, can also cause performance problems, especially in tight loops.

Frequent code inspections and mentors help

Beginners and early intermediates can be a common source of antipatterns. Pairing them with more experienced programmers and architects for code reviews and mentoring can head off many bad practices before they start. At allmystuff, the engineering department did a nice job of mentoring the solutions development staff, which typically consisted of weaker developers with better customer skills. Even a five-minute code inspection can reveal a surprising amount of information. Are the methods too long? Is the style readable and coherent? Are the variable names appropriately used? Does the programmer value her intelligence above readability?

End users are unusually perceptive

Later in my career, I began to appreciate the impact of end-user involvement at all stages of development. I found that end users can be brutally honest, when we allow them to be. When I began to truly listen to feedback, I could tell very early if my team would need to bear down or change direction. Too often, we ask for the questions and listen only if we hear what we want or expect.

Outsiders can use interviews

The most powerful tool for someone outside a development organization is the interview. People are put off when we try to propose answers without asking questions. Getting them to open up in an interview is usually not difficult but may occasionally be troublesome. When we are digging for trouble, people are much more perceptive if they perceive that we are helping to solve problems and not looking for someone to blame. Interviews are most useful if we can script at least a set of high-level questions, as well as anticipate some low-level questions.

Establishing a pattern

By itself, a problem is only a bug. We should already have processes and procedures for identifying and fixing bugs. Indeed, many of my father's customers had adequate measures for detecting and removing bad products from the line. The problems with these reactive approaches are twofold. First, we will never find all of the bugs. Second, if we do not fix the machinery or the process, we will create more bugs! After we have established a pattern, we need to elevate it from bug to antipattern.

1.4.3 Refactoring antipatterns

After we find a problem and establish a pattern, our strategy calls for refactoring it to form a better solution and process. Here, we are overlapping the realms of design patterns and antipattern. My intuition is that this combination is part of what is missing in the software quality industry. The combination of design patterns and antipatterns is practical and powerful. Poor solutions can be identified through antipatterns and redesigned into more proven and practical alternatives using design patterns. The process of continually improving code through restructuring for clarity or flexibility and the elimination of redundant or unused code is called *refactoring*.

Many experts advocate the rule "If it isn't broke, don't fix it." In the realm of software development, following this rule can be very expensive, especially at the beginning of a program's life cycle. The average line of code will be changed, modified, converted, and read many times over its lifetime. It is folly to view a refactoring exercise as time wasted without considering the tremendous savings over time. Instead, refactoring should be viewed as an investment that will pay whenever code is maintained, converted, read, enhanced, or otherwise modified. Therefore, refactoring is a cornerstone of this book.

1.5 Why Bitter Java?

In the Java community, the study and promotion of design patterns, or blueprints for proven solutions, has become well established and robust. The same is not true of the antipattern. As an architect and consultant, I have seen an amazing sameness to the mistakes that our customers tend to make. While the problem of the month may change slightly in a different domain or setting, the patterns of poor design, culture, and even technology stay remarkably consistent from one engagement to the next. I strongly believe that the study of antipatterns inherently changes the way we look at the software process. It keeps us observant. It makes us communicate. It helps us to step beyond our daily grind to make the fundamental process changes that are required to be successful.

Most of the antipatterns in *Bitter Java* have a relatively limited focus compared to the more general antipatterns in the *AntiPatterns* text. Each is applied to the server-side programming domain, which is popular right now and young enough to have a whole new set of common mistakes. Our hope is that this book will continue the evolution of the study of antipatterns and bring it into the Java community.

1.5.1 *The* Bitter Java *approach*

Bitter Java will take a set of examples, all related to a simple Internet message board, and redesign them over many chapters. Each iteration will point out a common antipattern and present a refactored, or redesigned, solution that solves the problem. In many cases, there may still be problems in the refactored solution. In most cases, these problems are addressed in later chapters. The others are left as an exercise for the reader. Regardless, the focus of the antipattern is to refactor a single problematic element.

The focus of *Bitter Java* is on server-side programming. The base architecture uses common server-side standards of servlets, JSPs, Java connectors, and EJBs. Where possible, the solutions are not limited to any vendor, though EJB implementations are currently platform specific.

1.5.2 Bitter Java *tools*

Based on my experience, I have chosen VisualAge for Java, WebSphere, and DB2 because the software and support are readily available to the authors. All of the implementations stress open Java designs and software architectures. Free, open alternatives to our software include:

- The home page for Java, with pages for base toolkits and specifications for J2EE extensions, can all be found at http://java.sun.com.
- A free servlet container, for the execution of servlets and JSPs either in a stand-alone mode or with a web server, can be found at http://jakarta.apache.org/tomcat/.
- A free web server can be found at http://apache.org.

BEA Systems' WebLogic also supports all of the classes and constructs used in this book, though they have been tested only on the original toolset. We do use the IBM database drivers (and I feel that the native database driver is almost always the best option), but we do not use the IBM-specific framework for databeans or servlet extensions, opting for the open counterparts instead.

1.5.3 The Bitter Java *organization*

Bitter Java presents some background information in chapters 1 and 2, and subsequent chapters present a series of antipatterns. The patterns are collected into themes. Chapters 3 and 4 focus on a design pattern called Model-View-Controller, and an associated Java design pattern called the Triangle. Chapters 5 and 6 concentrate on optimizing memory and caching. Chapters 7 and 8 concentrate on EJBs and connections. Chapters 9 and 10 address programming hygiene and good performance through scalability. The chapters are organized in the following manner:

- Background material for the chapter.
- A basic description of the antipattern, including some of the root causes and problems.
- Sample code illustrating the use of an antipattern.
- One or more refactored solutions.
- Sample code illustrating the refactored solution.
- A summary containing the highlights of the chapter.
- A list of all antipatterns covered in the chapter.

Antipatterns and templates

Each antipattern is presented twice: once in the main text, and once in template form at the end of each chapter. The templates that we have chosen, both within the chapters and at the end of most chapters, are based on the templates suggested in the *AntiPatterns* book. Those in the chapter text choose a minimalist organization with the keyword *antipattern* followed by its name in a heading, followed by some background material. Finally, we present a refactored solution following the *solution* keyword. At the end of each chapter is a more formal template, following the conventions in *AntiPatterns*. In this way, we make this initial contribution to the initial collection of Java antipatterns.

If you are looking for particular technologies or techniques, this is where to find them:

Table 1.1 The technologies and techniques in *Bitter Java* are presented in an order that suits the ongoing refactoring of a set of examples. This table can help you navigate to particular concepts that might interest you.

Technologies	Chapter
JSP design and composition	3, 4
Servlet design and composition	3, 4
JDBC, database programming	3, 4, 5, 6, 7
Connections framework	7
XML antipatterns	7
Web services	7
EJBs	8
Caching	5
Model-view-controller	3, 4
Performance antipatterns, tuning, and analysis	10
Antipatterns and the development process	1, 2, 11
Connection pooling	6
Coding standards and programming hygiene	9

For the programming examples, http://www.manning.com/tate/ has the complete code for all of the examples, as well as forums for discussing the topics of the book. The code in the book will be in the Courier style:

```
code = new Style("courier");
```

Where possible, long programs will have embedded text to describe the code. In other places, there may be in-line code `that looks like this`. Most of the programming samples are based on VisualAge for Java, version 4, and WebSphere Studio version 4. Most Java examples are based on JSP 1.1 and on Java 1.2. We'll tell you if the version is different. Some of the code examples for the antipatterns are for instructional purposes only and are not running programs. We have compiled and tried all of the good programming examples. They work.

1.5.4 The Bitter Java *audience*

Bitter Java is not written like a traditional technical manual or textbook. To keep things lively, we will mix in real-life adventure stories at the beginning of each chapter, with a programming moral later in the chapter. We hope that the style will engage many, and it might put off a few. If you are a reader who likes to cut to the chase, you will probably want to skip to chapter 3, and you may even want to skip the story at the front of each chapter. If you are looking for a dry reference with little extraneous content, this book is probably not for you.

The skill level for bitter Java is intermediate. If you have all of the latest Java design pattern books and have bookmarks for all of the key design pattern communities, this book is probably not for you. If you do not yet know Java, then you will want to try some introductory books and then come back to this one. If, like most Java programmers, you are an intermediate who could use some advice about some common Java pitfalls, then this book *is* for you. Those who have converted to Java from a simpler language, a scripting language, or a procedural language like C may find this book especially compelling.

Finally, *Bitter Java* is intended to be at a slightly lower level of abstraction than project management books, or the first *AntiPatterns* text. We intend to introduce code and designs that do not work and to refactor them. From these, we will show the low-level impact of process flaws, a failure to educate, and shortcuts. From this standpoint, architects and programmers will find appropriate messages, but they may find following the examples challenging. Project managers may also find some compelling thoughts, though the programming content may be slightly advanced. For both of these communities, antipattern summaries are listed at the end of each chapter in a concise template based on those in the original *AntiPatterns* text.

1.6 *Looking ahead*

Bitter Java is about programming war stories. Books like *The Mythical Man Month*, by Fredrick P. Brooks, have left an indelible impression in the minds of a whole generation of programmers. We aim to do for Java programmers what Brooks did for project managers. *Bitter Java* is about the quest for *imperfection*. We are not looking for isolated examples. We are looking for problems in process and culture, demonstrated by technically flawed designs. We are setting out to find the repeated mistakes that have bite. We are recording some of the useful mythology of the Java programmer.

In the next chapter, we will focus on the current landscape of the industry and why it is so ripe for antipatterns. Next, we will look at basic server-side designs and some antipatterns that plague them. Then, we will focus on common problems with resources and communication. Finally, we will look at advanced antipatterns related to enterprise Java deployments. So, settle down with this cup of bitter Java. We hope that when you're done, your next cup will be a smoother, more satisfying brew.