Bug Patterns in Java

ERIC ALLEN

Apress™

`Bug Patterns in Java`
Copyright ©2002 by Eric Allen

ISBN (pbk): 1-59059-061-9

Printed and bound in the United States of America 12345678910

Technical Reviewer: Corky Cartwright
Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager: Sofia Marchant
Development Editor: Kane Scarlett
Copy Editor: Josh Fruhlinger
Compositor: Impressions Book and Journal Services
Indexer: Ron Strauss
Cover Designer: Kurt Krames
Production Manager: Kari Brooks
Manufacturing Manager: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany. In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 2

Bugs, Specifications, and Implementations

We rigorously define the concept of a bug, explain why a specification is crucial for controlling software bugs, highlight the differences between a specification and an implementation, and discuss a cost-effective means for developing specifications.

What Is a Bug?

This book is about debugging software. In order to discuss the act of debugging, it is important to define precisely what does and does not constitute a bug.

For the purposes of this text, I will define a bug as "program behavior that deviates from its specification." This definition does not include:

- Poor performance, unless a threshold level of performance is included as part of the specification.

- An awkward or inefficient user interface. Although user interface design is an important topic, it's not the subject of this book.

- Lack of features, lack of a particular useful feature, or lack of any feature not included in the program specification (even if it was *intended* to be in the specification).

The lack-of-features category illustrates an important aspect of our definition of bugs: they are inextricably linked to a program specification. If there is no program specification, then there literally are no bugs. To be sure, there are some generally accepted behavioral qualities expected from any software, e.g., it won't crash, it won't run forever without producing output, etc. Properties like these are implicitly part of the specification of any software. But these properties are the exception; most behavior must be explicitly specified. Because specifications define behavior which defines bugs, we had better discuss what constitutes a specification.

TIP *The simplest definition of a bug is "program behavior that deviates from its specification."*

Intuitively, a program specification is a description of the behavior of a program. Therefore, having *some* kind of specification is essential to determining when the system is misbehaving. What form would we like this specification to take? First, let's consider how traditional software engineering answers this question.

TIP *Bugs and program specifications are inextricably linked. Since specifications define behavior, without a specification, bugs are not possible.*

Specification as Monolithic Treatise

The traditional method of software engineering is to develop a thorough specification of the system's functionality before entering a single line of code. This specification is made as formal as possible, so as to minimize ambiguities. The programmers then slog through the various details of this specification (often a large book) as they implement the system.

This method of specification was adapted from other engineering disciplines, where it can be extremely costly to make any changes to a specification after deployment begins. Microprocessor design is one of these disciplines. Currently, the specifications of microprocessors are interpreted and analyzed automatically. In fact, many aspects of a microprocessor design can be proven sound by unaided machines. But such techniques would be impossible if the specification weren't formalized.

In the software arena, where changes to a specification after deployment aren't nearly as costly, it's natural to question whether this style of up-front, formal specification is so useful. To consider this question, let's first examine how well that specification style works for a particular type of software artifact: a programming language.

Among software systems, programming languages are most similar to microprocessors in terms of the cost of modifying a specification. The cost of making even minor modifications to a language design after people have begun using it can be especially high. All the existing programs written in that language will have to be modified and recompiled.

As we might expect, the specifications of programming languages, compared with other software systems, are often quite formal, especially in the case of syntax. Virtually all modern programming languages have a formally specified syntax. Most parsers are constructed through the use of automatic *parser-generators* that read in such grammars and produce full-fledged parsers as output.

TIP *The specifications of virtually all modern programming languages contain a formally specified syntax.*

What about language semantics? Let's take a look at the following four languages, all either currently or formerly popular, and examine the relative degree of precision in the semantic specification for each:

- C++
- Python
- ML
- Pascal

For each language, let's look at how the degree of precision in the specification has helped determine its effectiveness.

C++

The C++ language specification leaves many parts of the specification implementation dependent, and even declines to define the behavior of many valid C++ programs. Although the designers of C++ would claim that the programs for which C++ semantics is undefined are not valid C++ programs, it is impossible in principle for a machine to determine automatically whether a program is valid under this criteria, implying that many (most?) real world software applications written in C++ are not valid C++ programs.

The result is that many C++ programs don't behave as intended when ported from one platform to another.

Python

The Python Language Reference is an informal language specification that leaves many details implementation dependent. In this case, the decision not to use a formal specification was made deliberately, with full awareness of the formalisms available for language semantics. In the words of Guido van Rossum, Python's inventor:

> *While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here.*

But the ambiguities of the English language aren't just a problem for Martians. Various implementations of Python, such as JPython and CPython, have faced a formidable challenge in providing compatible behavior across platforms. This problem would be much worse if it weren't for the relative simplicity and elegance of the Python language.

ML

The ML specification formally defines the full operational semantics of the language. Consequently, ML programmers enjoy an unprecedented level of precision and cross-platform standardization. The formal specification of ML has even allowed computer scientists to discover subtle inconsistencies in the ML type system, and correct them. Such inconsistencies no doubt exist in many other languages, but they are difficult to find without a formal specification.

Pascal

Pascal is one language that suffered for quite a while with an inconsistency in its specification: the rules for determining type equivalence were left unspecified. For example, consider the following two Pascal types:

```
type complex = record
                  left : integer;
                  right : integer;
                end;
type coordinate = record
                      left : integer;
                      right : integer;
                    end;
```

Are these two types identical? Clearly, they contain the same types of subcomponents. Depending on how we define the language, a value of type coordinate may be passed to a procedure that takes an argument of type complex, and vice versa. If so, our language would be said to use *structural equivalence* when identifying types. Alternatively, we might define two types as equivalent if and only if they have the same name. Then our language would use *name equivalence.*

What choice is made in the Pascal language specification? Originally, no choice was made at all; nobody had yet realized that there was more than one way to define type equivalence! As a result, each implementation team for Pascal had to make this choice on its own (and, often, those teams didn't realize they were making a choice either). But the result of this ambiguity is that Pascal code written for one implementation can behave in a drastically different fashion on others.

Benefits of Specifications

Although no formal specification (akin to that of ML) exists for Java, a good deal of care was put into development of a precise informal specification. Many smaller, toy versions of Java have been formalized from this specification, and correctness properties have been proven about them. Furthermore, Java is typically compiled to bytecode for the Java Virtual Machine, which itself is well specified (although, at the time of this writing, the process of bytecode verification is not). The result is an unprecedented level of portability for programs written in this language.

TIP *Java programs have a higher degree of portability because of the language's precise, albeit informal, specification.*

The conclusion we can draw from this is that there really are advantages to having as precise a specification as possible. The costs of an ambiguity or inconsistency can be quite high, leading to decreased portability, reliability, or even to a security hole.

But even in the world of programming languages, where problems in a specification are most costly, formal specifications are rare. Some of the reasons for this are:

- **Few language properties are checked automatically.** The process of proving properties about a programming language specification hasn't been automated, as of yet, to the same degree that proving properties about hardware design has. As a result, there's not quite as much advantage to formalizing them.

- **Many language users prefer the informal.** Informal specifications are preferred by most of the people who will actually read them, like compiler writers. (In fact, compiler writers often revel in less formal specifications because it gives them more room to optimize a program.) The other, occasional, users of a language specification are the programmers, and most of them greatly appreciate an informal specification that they can easily understand.

- **It costs money to produce a formal specification.** Producing a formal specifi-cation up front is expensive. Companies have found it more cost effective to ship early and flesh out the details of a specification later (or, more often, never). Admittedly, if a development team commits to producing a specification, it may not finish formally specifying its system before its competitors have already shipped! If Sun had waited to produce formal language semantics for Java before releasing it, the language may not have come out in time to ride to fame as the preferred language for Internet programming.

But if up-front and formal specification is too costly, what approach should a development team take in specifying software? Many development teams have been so turned off by the cost of up-front specification that they've renounced specification entirely. But that's never a good idea.

Implementations Are Not Specifications

Like it or not, a great deal of industrial software is implemented without a discernible specification. If and when the software is completed, the implementation is then presented as the specification. Whatever behavior the software exhibits is said to be the specified behavior. Some poor souls might argue that this is a good approach, since it doesn't bog the developers down with working out some sort of formal plan that is bound to change anyway. But, while it is true that project specifications often change, an implementation makes for a lousy specification in several respects:

- **Many of the choices made in an implementation are arbitrary.** Thus, a team that wishes to implement the system on another platform has nothing to go on but the existing implementation. The developers will have to wade through numerous implementation details to determine the behavior that the implementation entails. It is much easier to determine such behavior when it is specified at a higher level of abstraction.

- **You cannot define a bug.** If an implementation is literally taken as its own specification, then, as in the case where there is no specification at all, it is impossible to identify any behavior as a bug!

- **Initial developers have no model of behavior.** Obviously, an implementation cannot serve as the specification for the initial developers, since no such implementation yet exists. These developers must rely on *some* model of behavior for the system they're creating. But the source of this model should then serve as the software's specification.

This last point sheds some light on what sort of a specification a developer might use with reasonable cost. While it's true that developers must have some mental model of the feature they are implementing, they needn't have a mental model of the entire application.

In other words, specifications can be developed a piece at a time. Not only does this make them more tractable, but it also allows them to be modified more efficiently as the customers' needs change.

Building Cost-Effective Specifications with Stories

One good way to develop software requirements incrementally is in the manner advocated by extreme programming (XP), an agile software-development method that has become quite popular over the past few years. The name comes from the concept that many commonly accepted and uncontroversial development practices that are usually executed alone (such as testing, incremental development, pair programming, etc.) can create a synergistic effect when all practiced together in a radical, or *extreme*, form. Let's focus on the way in which XP teams specify the functionality of their software. (More on XP in **Chapter 3: Debugging and the Development Process.**)

TIP *With extreme programming, functionality is specified incrementally via stories—brief descriptions of an aspect of system behavior.*

In an XP project, the required functionality of a system is specified incrementally through the use of *stories*. Each story briefly describes one aspect of the system's behavior. Let's consider a simple story from a real-world XP project: a free, open-source Java IDE called DrJava.

DrJava was developed at the JavaPLT research laboratory of Rice University and was designed to provide an extremely simple but powerful user interface that enables programmers at all levels to manipulate, test, and debug their code. It not only integrates testing and debugging support, but also provides an integrated "read-eval-print loop" that allows users to evaluate arbitrary Java expressions interactively. DrJava will be the basis of many examples in this book for the following reasons:

- It provides a great example of code developed in an XP project.
- I'm familiar with its code base.
- All of the code is open source, so you can download the DrJava jar file, along with all of the source code, at http://drjava.sourceforge.net.

Let's consider the following story from DrJava's early stages of development:

> *As the user types words into the editor, occurrences of Java keywords are automatically colored **blue**. String and character literals are colored **green**, and comments are colored **red**.*

Believe it or not, that's actually a hard story to get right in Java, partly because of some peculiar properties of block comments. Depending on the velocity of the development team, it may be advisable to break that story up into two or more smaller ones.

TIP *Small, new stories can be easily added to modify functionality. This method works well in situations where requirements change frequently and bits of new functionality need to be rolled out quickly.*

Still, notice that the functionality specified by this story is of a tiny scope when compared to a traditional, full, up-front specification for an IDE. Also, this story is written in simple, clear language. That makes it easy to split up into smaller stories when necessary, and it prevents *coupling* (unwanted entanglements) between the parts of a specification.

Let's look at another story, this time for an "interactions window" in which the user can enter Java statements and expressions dynamically and then see the results. We wanted to add to this window the ability to scroll through earlier commands with the up and down arrow keys:

> *The user enters a new command at the prompt in the interactions window. Once the user hits Enter, the command is executed, the result is displayed, and the cursor is moved down to a new prompt. Previous commands can be recalled to the current prompt using the up and down arrows, allowing the user to scroll through a history of commands.*

In this case, the story is slightly longer, but the specified functionality is still very limited. Some time after we implemented this story, a user complained that he couldn't easily get back to a blank prompt after he started scrolling through the old commands by going to the bottom of the list. (Actually, the user could have pressed the Escape key to clear the line, but his suggestion made for a more natural interface.) No problem. We extended the existing specification by writing a new story:

> *When scrolling through the history in the interactions window using the arrow keys, the user moves down to the most recently entered command. The user hits the down arrow once more, and a blank line appears. He then types in a new command. This feature is convenient for entering new commands after scrolling through previously entered ones.*

Very little new functionality is specified in this story. If support for scrolling through previous commands was already implemented, it's not hard to imagine that a pair of programmers could implement this new story in less than an hour. Because the stories are small, new ones can be added to modify program functionality without having to completely overhaul the specification. For this reason, stories work particularly well when the requirements for the software product change frequently.

Additionally, by specifying and implementing stories incrementally, the programmers are able to release new functionality quite rapidly, allowing the customers to get more value from the software more quickly.

TIP *Stories may be prone to the same ambiguities and inconsistencies as any informal specification. Accompanying tests can eliminate these errors.*

Although stories allow us to specify software incrementally, they have the disadvantage of not being as formal as those traditional up-front software specs. Therefore, they are prone to the same ambiguities and inconsistencies as any other informal specification. But if the traditional formal specifications are too costly, is there any way that these errors can be eliminated?

Include tests to eliminate specification errors

One way to eliminate ambiguities and inconsistencies in a story is to include tests with it. If there's a section or clause in a story that has multiple interpretations, just write a test that helps to define that aspect of the interpretation. Provided the programming language you write the test in isn't ambiguous, that test will nail down the behavior of the program. In addition, if a set of unit tests specifies inconsistent functionality, it will be impossible for a program to pass them all.

Extreme programming uses two forms of tests: *acceptance tests* and *unit tests*. Acceptance tests check user-observable functionality. Unit tests are small tests that check specific "units" of program functionality.

A key feature of both kinds of tests is that they automatically check the desired functionality; it's not necessary for the programmer to examine the output of each test to ensure it's correct. If a test fails when run, the programmer is notified; otherwise, he knows that it passed.

In extreme programming, testing is a way of life. The programmers start writing tests before they write any of the implementation at all, and they continue writing more unit tests for each new aspect of program functionality. A rigorous suite of tests laid over a software project provides several advantages:

- The tests are an important form of documentation.
- The tests expedite the process of refactoring.
- The tests complement stories as part of the specification.

The tests are an important form of documentation

Since the tests (ideally) cover every aspect of the implementation, and since they invoke the functionality in simple ways to make sure it is working, it is easy for a programmer who is joining a project (or taking over maintenance of code) to read through the tests and determine what the various functional components do.

When first hearing of the concept that a test can be considered documentation, some people are skeptical: "How can you write documentation for a program in the same language that the program is written in?" This question misses the point of code documentation. Code should never be documented to explain what the code is doing; the code itself already does that.

Instead, documentation should explain *why* a block of code is doing what it does. Anyone reading the code should be already familiar with the language used; if not, then documentation in any language is unlikely to help. Granted, it is not always clear how a block of code interacts with the rest of a program, and documentation is good for that purpose. But since the reader of the code is (or should be) familiar with the language, it is perfectly valid to explain the intention behind the code in the same language as the code.

TIP *Documentation should explain why a block of code is doing what it is doing.*

Tests expedite the process of refactoring

When a suite of tests can be run over the code at any time to determine if any of the functionality has been broken, programmers can refactor the code with much more confidence that they aren't stomping over the invariants of each other's code. The vast majority of bugs introduced can be detected as soon as they're introduced.

Tests complement stories as part of the specification

But what is mentioned less often is that tests complement stories as part of the specification. And just as stories allow for the incremental and informal specification of a system, unit tests allow for the incremental and formal specification of the same system. Although no set of unit tests can nail down all aspects of a system, a test suite can define the most ambiguous aspects. Furthermore, tests have huge advantages over most forms of formal specification:

- Each test can be written independently of the rest.

- The tests can be automatically verified. Few other forms of formal specification have this property (the most notable exception being static types). Compare the process of running tests to a manual proof of correctness for a program. If there is a flaw in the proof, all bets are off. But if only one test fails, at least we know that the rest of them passed. Plus, we don't need to rewrite the unit tests whenever we modify the implementation as we would for a manual proof.

- The tests can be written in the same language as the program. Thus, programmers needn't learn another formalism to formally specify functionality.

TIP *Tests are important as documentation, to expedite refactoring, and to complement stories as part of the specification.*

For an example of how unit tests can help to better define a specification, let's return to our story concerning the history of commands in the DrJava interactions window. As we mentioned, the user can scroll back through this history with the up and down arrows, and extract text for forming new commands. One of the classes used to implement this story in DrJava is a `History` class, which stores the list of commands that have occurred so far.

What happens when the user issues the same command twice in a row? This question isn't answered by the story shown previously. But we'd like the History to store only one of the two commands; it's tedious to have to scroll through a series of identical commands. We could write the following unit test to enforce this property:

```
public void testMultipleInsert() {
  _history.add("new Object()");
  _history.add("new Object()");
  assertEquals("Duplicate elements inserted", 1, _history.size());
}
```

Notice that the method takes no arguments and returns void. That's because it is run automatically. We don't need to feed it input or check its output; if it doesn't pass, it'll throw an exception. (This test is written in the form used by JUnit, a free, open-source testing harness for Java. JUnit is part of the xUnit suite of test harnesses, providing open-source testing tools for most popular programming languages.)

The test starts with a fresh `History` object (set in the `_history` field) and adds two identical commands. It then checks that the length of the `History` is exactly one. The `assertEquals` method takes three arguments: a message to signal if the test fails and two values. If the values are equal, the test succeeds; otherwise it fails.

What are some other tests we could put in our History class? Why don't we formalize the property stated in the final story example: that we can move back to a blank line at the end of the `History`. Following is a test for that:

```
public void testCanMoveToEmptyAtEnd() {
  _history.add("some text");

  _history.movePrevious();
  assertEquals("Prev did not move to correct item",
               "some text",
               _history.getCurrent());

  _history.moveNext();
  assertEquals("Can't move to blank line at end",
               "",
               _history.getCurrent());
}
```

Notice that these tests are gradually winnowing down the definitions for the set of methods that the `History` class will have to implement.

Writing tests is a great way to determine the interface that a class should implement. Because you have to program to that interface yourself, you'll see just how difficult, or easy, you're making it to work with your interface. Your own preference for using simple interfaces will help you keep your own interfaces simple. It'll also help you maintain your tests in an easy-to-read form.

Of course, there are many other tests we can include over class `History`. But we shouldn't write them all before implementing some of the functionality. The better procedure is to:

1. Write just a few tests;

2. Write code to pass the tests from Step 1;

3. Repeat Steps 1 and 2 as many times as needed.

This way, we can integrate the code at each step (and make sure we didn't break anything).

The History class was implemented in DrJava in the following way:

```
 /**
  * Keeps track of what was typed in the interactions pane.
  * @version $Id: History.java,v 1.9 2002/03/06 18:59:02 eallen Exp $
  */
public class History {
  private Vector<String> _vector = new Vector<String>();
  private int _cursor = -1;

  /**
   * Adds an item to the history and moves the cursor to point
   * to the place after it.
   *
   * To access the newly inserted item, you must movePrevious first.
   */
  public void add(String item) {

    if (item.trim().length() > 0) {
      if (_vector.isEmpty() || ! _vector.lastElement().equals(item)) {
        _vector.addElement(item);
      }
      moveEnd();
    }
  }

  /**
   * Move the cursor to just past the end. To access the last element,
   * you must movePrevious.
   */
  public void moveEnd() {
    _cursor = _vector.size();
  }

  /** Moves cursor back 1, or throws exception if there is none. */
  public void movePrevious() {
    if (!hasPrevious()) {
      throw  new ArrayIndexOutOfBoundsException();
    }
    _cursor–;
  }
  /** Moves cursor forward 1, or throws exception if there is none. */
  public void moveNext() {
    if (!hasNext()) {
      throw  new ArrayIndexOutOfBoundsException();
    }
    _cursor++;
  }

  /** Returns whether moveNext() would succeed right now. */
  public boolean hasNext() {
    return  _cursor < (_vector.size());
  }

  /** Returns whether movePrevious() would succeed right now. */
  public boolean hasPrevious() {
    return  _cursor > 0;
  }

  /**
   * Returns item in history at current position, or throws exception if none.
   */
  public String getCurrent() {
    if (hasNext()) {
      return  _vector.elementAt(_cursor);
    }
    else {
      return "";
    }
  }

  /**
   * Returns the number of items in this History.
   */
  public int size() {
    return _vector.size();
  }
}
```

Now, here's a great example of just how easy it is to incrementally add to the formal specification of a program with unit tests. Let's say that, after writing the code above, we decided we wanted to limit the length of the History to 500 items in order to prevent runaway memory consumption in long-standing processes. So, we add the following unit test to our suite:

```
/**
 * Ensures that Histories are bound to 500 entries.
 */
public void testHistoryIsBounded() {

  int maxLength = 500;

  for (int i = 0; i < maxLength + 100; i++) {
    _history.add("testing " + i);
  }
  while(_history.hasPrevious()) {
    _history.movePrevious();
  }
  assertEquals("history length is not bound to " + maxLength,
               "testing 100",
               _history.getCurrent());
}
```

This new test adds 600 elements to the `History` and checks that a few assertions hold. Notice that it doesn't just check that only 500 entries are included in the History; it checks that items are removed in a FIFO (first-in-first-out) order. It accomplishes that check by ensuring that the oldest element in the History is the 100th element added, which is exactly what it should be if the oldest elements are removed with every command after the 500th entry.

Modifying class `History` to pass this test was easy: first, we added the following constant to class `History`:

```
private static final int MAX_SIZE = 500;
```

Then we modified the add() method as follows:

```
/**
 * Adds an item to the history and moves the cursor to point
 * to the place after it.
 * Note: Items are not inserted if they would duplicate the last item,
 * or if they are empty. (This is in accordance with bug #522123 and
 * feature #522213.)
 *
 * Thus, to access the newly inserted item, you must movePrevious first.
 */
public void add(String item) {
    if (item.trim().length() > 0) {
      if (_vector.isEmpty() || ! _vector.lastElement().equals(item)) {
        _vector.addElement(item);

        // If adding the new element has filled _vector to beyond max
        // capacity, spill the oldest element out of the History.
        if (_vector.size() > MAX_SIZE) {
          _vector.removeElementAt(0);
        }
      }
      moveEnd();
    }
  }
```

With this fix, the code behaves as specified.

Unit tests can't do everything

As the preceding example demonstrates, unit tests are an essential complement to stories for the incremental specification of a software system. In fact, some might be tempted to use a suite of unit tests as the *sole* specification of a system. But using unit tests to form the only specification has one big disadvantage: the set of tests over a system are inevitably incomplete.

No matter how many tests we specify over a system, there will always be more inputs and states of the system than we could ever hope to represent. We could interpret the tests as specifying the “most reasonable” extension, but such an extension will often be ambiguous. That’s where the strength of stories comes in. Just as unit tests can clarify the intended *specific* aspects of a story, a story can clarify the intended *general* aspects of a unit test. Both are needed for an effective and agile software specification.

TIP *Unit tests can clarify the intended specific aspects of a story. A story can clarify the intended general aspects of a unit test.*

The use of stories and unit tests can aid software development in many ways, but here we’ve described their use solely for efficiently specifying software systems. And we’ve also emphasized the need to use specifications by pointing out that they’re necessary for precisely identifying bugs in a program. Thus, a serious concern for debugging can influence the way we program, even at the level of *specifying* software.

A Quick Recap

In this chapter, we've learned to:

- Define the concept of a bug.
- Explain why a specification is crucial for controlling software bugs.
- Understand the differences between a specification and an implementation.
- Use stories and unit tests when developing specifications.
- Introduce cost-effective means for developing specifications.

Now, let's discuss the impact that the debugging process can have on designing and coding software systems.

In **Chapter 3,** we'll look at how six tenets of extreme programming make the debugging process easier and more effective. We'll also discuss the crucial interdependence between effective debugging and effective development, highlight extreme programming development methods, and peek at a future of test-oriented languages.