9 Testing in the Agile Environment

Increased competition and interconnectedness in all markets have forced businesses to shorten their time-to-market while continuing to provide high-quality products to their customers. This is particularly true in the software development industry where the Internet makes possible near-instant delivery of software applications and services. Whether creating a product for the masses or for the human resources department, one fact remains immutable: The twenty-first century customer demands a quality application delivered almost immediately. Unfortunately, traditional software development processes cannot keep up in this competitive environment.

In the early 2000s, a group of developers met to discuss the state of lightweight and rapid development methodologies. At the gathering they compared notes to identify what successful software projects look like; what made some projects succeed while others limped along. In the end, they created the "Manifesto for Agile Software Development," a document that became the cornerstone of the Agile movement. Less a discrete methodology, the Agile Manifesto (Figure 9.1) is a unique philosophy that focuses on customers and employees, in lieu of rigid approaches and hierarchies.

Features of Agile Development

Agile development promotes iterative and incremental development, with significant testing, that is customer-centric and welcomes change during the process. All attributes of traditional software development approaches

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck	Mike Beedle	Arie van Bennekum
Alistair Cockburn	Ward Cunningham	Martin Fowler
James Grenning	Jim Highsmith	Andrew Hunt
Ron Jeffries	Jon Kern	Brian Marick
Robert C. Martin	Steve Mellor	Ken Schwaber
	Jeff Sutherland	Dave Thomas

© 2001, the above authors
this declaration may be freely copied in any form,
but only in its entirety through this notice.

FIGURE 9.1 Manifesto of Agile Software Development.

neglect or minimize the importance of the customer. Although Agile methodologies incorporate flexibility into their processes, the main emphasis is on customer satisfaction. The customer is a key component of the process; simply put, without customer involvement, the Agile method fails. And knowing their interaction is welcomed helps customers build satisfaction and confidence in the end product and development team. If the customer is not committed, then more traditional processes may be a better development choice.

Ironically, Agile development has no single development methodology or process; many rapid development approaches may be considered Agile. These approaches do, however, share three common threads: They rely on customer involvement, mandate significant testing, and have short, iterative development cycles. It is beyond the scope of this book to cover each methodology in detail, but in Table 9.1 we identify the methodologies considered Agile and give a brief description of each. (We urge you to learn

TABLE 9.1 Agile Development Methodologies

Methodology	Description
Agile Modeling	Not so much a single modeling methodology, but a collection of principles and practices for modeling and documenting software systems. Used to support other methods such as Extreme Programming and Scrum.
Agile Unified Process	Simplified version of the Rational Unified Process (RUP) tailored for Agile development.
Dynamic Systems Development Method	Based on rapid application development approaches, this methodology relies on continuous customer involvement and uses an iterative and incremental approach, with the goal of delivering software on time and within budget.
Essential Unified Process (EssUP)	An adaptation of RUP in which you choose the practices, (e.g. use cases or team programming) that fit your project. RUP generally uses all practices, whether needed or not.
Extreme Programming	Another iterative and incremental approach that relies heavily on unit and acceptance testing. Probably the best known of the Agile methodologies.
Feature Driven Development	A methodology that uses industry best practices, such as regular builds, domain object modeling, and feature teams, that are driven by the customer's feature set.
Open Unified Process	An Agile approach to implementing standard Unified practices that allows a software team to rapidly develop their product.
Scrum	An iterative and incremental project management approach that supports many Agile methodologies.
Velocity Tracking	Applies to all Agile development methodologies. It attempts to measure the rate, or "velocity," at which the development process is moving.

more about them because they represent the essence of the Agile philosophy.) In addition, we cover Extreme Programming, one of the more popular Agile methodologies, in greater detail later in this chapter, and offer a practical example.

It's worth noting that some Agile methodologies are collections, or adaptations, of traditional software development processes. The Essential Unified Process (EssUP) is an example. EssUP takes processes from the Rational Unified Process (RUP) and other well-known software development process models that support the Agile development philosophy.

Make no mistake, adopting an Agile development methodology is challenging. It takes the right combination of developers, managers, and customers to make it work. But in the end, the product will benefit from constant testing and heavy customer involvement.

Agile Testing

In essence, Agile testing is a form of collaborative testing, in that everyone is involved in the process through design, implementation, and execution of the test plan. Customers are involved in defining acceptance tests by defining use cases and program attributes. Developers collaborate with testers to build test harnesses that can test functionality automatically. Agile testing requires that everyone be engaged in the test process, which requires a lot of communication and collaboration.

As with most aspects of Agile development, Agile testing necessitates engaging the customer as early as possible and throughout the development cycle. For example, once developers produce a stable code base, customers should begin acceptance testing and provide feedback to the development team. It also means that testing is not a phase; rather, it is integrated with development efforts to compel continuous progress.

To ensure that the customer receives a stable product with which to perform acceptance testing, developers generally begin by writing unit tests first, then move to coding software units. The unit tests are failure tests, in that developers design them to cause their software to fail some requirement. Paradoxically, developers must write failing software to, in effect, test the test. Once test harnesses are in place, developers proceed to write software that passes the unit tests.

To facilitate the timely feedback needed for rapid development, Agile testing relies on automated testing. Development cycles are short, so time is valuable, and automated testing is more reliable than manual testing approaches. Not only is manual testing time-consuming, it may itself introduce bugs. Numerous open-source and commercial testing suites exist. It really does not matter which of these available testing suites is used, only

that developers and testers use one. Although some problems may require exploratory manual testing, automated testing is preferred.

Agile development environments often comprise only small teams of developers, who also act as testers. Larger projects with more resources may include an individual tester or a testing group. In either case, testers should not be considered finger-pointers. Their job is to move the project forward by providing feedback about the quality of the software so that developers can implement bug fixes and make requirement changes and general improvements.

Agile testing fits well into the Extreme Programming methodology whereby developers create unit tests first, then the software. In the remainder of this chapter we cover Extreme Programming and Extreme Testing in more detail.

Extreme Programming and Testing

In the 1990s an innovative software development methodology termed Extreme Programming (XP) was born. A project manager named Kent Beck is credited with conceiving this lightweight, Agile development process, first testing it while working on a project at Daimler-Chrysler in 1996. Although several other Agile software development processes have since been created, XP is still the most popular. In fact, numerous open-source tools exist to support it, which is testimony to XP's popularity among developers and project managers.

XP likely was developed to support the adoption of programming languages such as Java, Visual Basic, and C#.

These object-based languages allow developers to create large, complex applications much more quickly than with traditional languages such as C, Fortran, or COBOL. Developing with these languages often requires building general-purpose libraries to support the application's coding efforts. Methods for common tasks such as printing, sorting, networking, and statistical analysis are not standard components. Languages such as C# and Java ship with full-featured application programming interfaces (APIs) that eliminate or reduce the need for creating custom libraries.

However, along with the benefits of rapid application development languages came liabilities. Although developers were creating applications much more quickly, their quality was not guaranteed. If an application compiled, it often failed to meet the customer's specifications or

expectations. The XP development methodology facilitates the creation of quality programs in short time frames. Although classical software processes still work, they often take too much time, which equates to lost income in the highly competitive arena of software development.

Besides customer involvement, the XP model relies heavily on unit and acceptance testing. In general, developers run unit tests for every incremental code change, no matter how small, to ensure that the code base still meets its specification. In fact, testing is of such importance in XP that the process requires you to create the unit (module) and acceptance tests first, then your code base. This form of testing is called, appropriately, Extreme Testing (XT).

Extreme Programming Basics

As mentioned, XP is a software process that helps developers create high-quality code, rapidly. Here, we define "quality" as a code base that meets the design specification and customer expectation.

XP focuses on:

- Implementing simple designs.
- Communicating between developers and customers.
- Continually testing the code base.
- Refactoring, to accommodate specification changes.
- Seeking customer feedback.

XP tends to work well for small to medium-size development efforts in environments that have frequent specification changes, and where near-instant communication is possible.

XP differs from traditional development processes in several ways. First, it avoids the large-scale project syndrome in which the customer and the programming team meet to design every detail of the application before coding begins. Project managers know this approach has its drawbacks, not the least of which is that customer specifications and requirements constantly change to reflect new business rules or marketplace conditions. For example, the finance department may want payroll reports sorted by processed date instead of check numbers; or the marketing department may determine that consumers will not buy product XYZ if it doesn't send an e-mail after website registration. In contrast, XP planning sessions

focus on collecting *general application requirements*, not narrowing in on every detail.

Another difference with the XP methodology is that it avoids coding unneeded functionality. If your customer thinks the feature is needed but not required, it generally is left out of the release. Thus, you can focus on the task at hand, adding value to a software product. Concentrating only on the required functionality helps you produce quality software in short time frames.

But the primary difference of XP compared to traditional methodologies is its approach to testing. After an all-inclusive design phase, traditional software development models suggest you code first and create testing interfaces later. In XP, you *must* create the unit tests first, and then write the code to pass the tests. You design unit tests in an XP environment by following the concepts discussed in Chapter 5.

The XP development model has 12 core practices that drive the process, summarized in Table 9.2. In a nutshell, you can group the 12 core XP practices into four concepts:

1. Listening to the customer and other programmers.
2. Collaborating with the customer to develop the application's specification and test cases.
3. Coding with a programming partner.
4. Testing, and retesting, the code base.

Most of the comments for each practice listed in Table 9.2 are self-explanatory. However, a couple of the more important principles, namely planning and testing, warrant further discussion.

XP Planning

A successful planning phase lays the foundation of the XP process. The planning phase in XP differs from that in traditional development models, which often combine requirements gathering and application design. Planning in XP focuses on identifying your customer's application requirements and designing user stories (or case stories) that meet them. You gain significant insight into the application's purpose and requirements by creating user stories. In addition, the customer employs the user stories when performing acceptance tests at the end of a release cycle. Finally, an intangible benefit of the planning phase is that the customer gains ownership and confidence in the application by participating intimately in it.

TABLE 9.2 The 12 Practices of Extreme Programming

Practice	Comment
1. Planning and requirements	Marketing and business development personnel work together to identify the maximum business value of each software feature. Each major software feature is written as a user story. Programmers provide time estimates to complete each user story. The customer chooses the software features based on time estimates and business value.
2. Small, incremental releases	Strive to add small, tangible, value-added features and release a new code base often.
3. System metaphors	Your programming team identifies an organizing metaphor to help with naming conventions and program flow.
4. Simple designs	Implement the simplest design that allows your code to pass its unit tests. Assume change will come, so don't spend a lot of time designing; just implement.
5. Continuous testing	Write unit tests before writing the code module. Each unit is not complete until it passes its unit test. Further, the program is not complete until it passes all unit tests, and acceptance tests are complete.
6. Refactoring	Clean up and streamline your code base. Unit tests help ensure that you do not destroy the functionality in the process. You must rerun all unit tests after any refactoring.
7. Pair programming	You and another programmer work together, at the same machine, to create the code base. This allows for real-time code review, which dramatically facilitates bug detection and resolution.
8. Collective ownership of the code	All code is owned by all programmers. No single programmer is dedicated to a specific code base.
9. Continuous integration	Every day, integrate all changes; after the code passes the unit tests, add it back into the code base.
10. Forty-hour workweek	No overtime is allowed. If you work with dedication for 40 hours per week, overtime will not be needed. The exception is the week before a major release.

Table 9.2 (*continued*)

Practice	Comment
11. On-site customer presence	You and your programming team have unlimited access to the customer, to enable you to resolve questions quickly and decisively, which keeps the development process from stalling.
12. Coding standards	All code should look the same. Developing a system metaphor helps meet this principle.

XP Testing Continuous testing is central to the success of a XP-based effort. Although acceptance testing falls under this principle, unit testing occupies the bulk of the effort. Unit tests are devised to make the software fail. Only by ensuring that your tests detect errors can you begin correcting the code so it passes the tests. Assuring that your unit tests catch failures is key to the testing process—and to a developer's confidence. At this point, the developer can experiment with different implementations, knowing that the unit tests will catch any mistakes.

You want to ensure that any code changes improve the application and do not introduce bugs. The continuous testing principle also supports refactoring efforts used to optimize and streamline the code base. Constant testing also leads to that intangible benefit already mentioned: confidence. The programming team gains confidence in the code base because you constantly validate it with unit tests. In addition, your customers' confidence in their investment soars because they know the code base passes unit tests every day.

Example XP Project Flow Now that we've presented the 12 practices of the XP process, you may be wondering, how does a typical XP project flow? Here is a quick example of what you might experience if you worked on an XP-based project:

1. Programmers meet *with* the customer to determine the product requirements and build user stories.
2. Programmers meet *without* the customer to divide the requirements into independent tasks and estimate the time to complete each task.

3. Programmers present the customer with the task list and with time estimates, and ask them to generate a priority list of features.
4. The programming team assigns tasks to pairs of programmers, based on their skill sets.
5. Each pair creates unit tests for their programming task using the application's specification.
6. Each pair works on their task with the goal of creating a code base that passes the unit tests.
7. Each pair fixes, then retests their code until all unit tests have passed.
8. All pairs gather every day to integrate their code bases.
9. The team releases a preproduction version of the application.
10. Customers run acceptance tests and either approve the application or produce a report identifying the bugs/deficiencies.
11. Upon successful acceptance tests, programmers release a version into production.
12. Programmers update time estimates based on latest experience.

Although compelling, XP is not for every project or every organization. Proponents of XP conclude that if a programming team fully implements the 12 practices, then the chances of successful application development increase dramatically. Detractors say that because XP is a process, you must do all or nothing; if you skip a practice, then you are not properly implementing XP, and your program quality may suffer. Detractors also claim that the cost of changing a program in the future to add more features is higher than the cost of initially anticipating and coding the requirement. Finally, some programmers find working in pairs very cumbersome and invasive; therefore, they do not embrace the XP philosophy.

Whatever your views, we recommend that you consider XP as a software methodology for your project. Carefully weigh its pros and cons against the attributes of your project and make the best decision based on that assessment.

Extreme Testing: The Concepts

To meet the pace and philosophy of XP, developers use Extreme Testing, which focuses on constant testing. As mentioned earlier, two forms of testing make up the bulk of XT: unit testing and acceptance testing. The theory used when writing the tests does not vary significantly from the theory presented in Chapter 5; however, the stage in the development process in

which you create the tests does differ. XT mandates creating tests *before* coding begins, not after. Nonetheless, XT and traditional testing share the same goal: to identify errors in a program.

In the rest of this section we provide more information on unit and acceptance testing, from an Extreme Programming perspective.

Extreme Unit Testing

Unit testing, the primary testing approach used in Extreme Testing, and has two simple rules: All code modules must have unit tests before coding begins, and all code modules must pass unit tests before being released into acceptance testing. At first glance this may not seem so extreme. Closer inspection reveals the big difference between unit testing, as previously described, and XT unit testing: The unit tests must be defined and created before coding the module.

Initially, you may wonder why you should, or how you can, create test drivers for code you haven't yet written. You may also think that you do not have time to create the tests and still meet the project deadline. These are valid concerns, but concerns we can address easily by listing a number of important benefits associated with writing unit tests before you start coding the application:

- You gain confidence that your code will meet its specification and requirements.
- You express the end result before you start coding.
- You better understand the application's specification and requirements.
- You may implement simple designs initially and confidently refactor the code later to improve performance, without worrying about breaking the specification.

Of these benefits, the insight and understanding you gain of the application's specification and requirements cannot be underestimated. For example, if you start coding first, you may not fully understand the acceptable data types and boundaries for the input values of an application. How can you write a unit test to perform boundary analysis without understanding the acceptable inputs? Can the application accept only numbers, only characters, or both? If you create the unit tests first, you *must* understand the specification. The practice of creating unit tests first is the shining star in the XP methodology, as it forces you to understand the specification to resolve ambiguities *before* you begin coding.

As mentioned in Chapter 5, you determine the unit's scope. Given that today's popular programming languages such as Java, C#, and Visual Basic are mostly object-oriented, modules are often classes, or even individual class methods. You may sometimes define a module as a group of classes or methods that represent some functionality. Only you, as the programmer, know the architecture of the application and how best to build the unit tests for it.

Manually running unit tests, even for the smallest application, can be a daunting task. As the application grows, you may generate hundreds or thousands of unit tests. Therefore, you typically use an automated software testing suite to ease the burden of running these unit tests. With these suites you script the tests and then run all or part of them. In addition, testing suites typically allow you to generate reports and classify the bugs that frequently occur in your application. This information may help you proactively eliminate bugs in the future.

Interestingly enough, once you create and validate your unit tests, the "testing" code base becomes as valuable as the software application you are trying to create. As a result, you should keep the tests in a code repository, for protection. Likewise, you should institute adequate backups of the test code, and ensure that needed security is in place.

Extreme Acceptance Testing Acceptance testing represents the second, and equally important, type of XT that occurs in the XP methodology. Acceptance testing determines whether the application meets other requirements, such as functionality and usability. You and the customer create the acceptance tests during the design/planning phases.

Unlike the other forms of testing discussed thus far, customers, not you or your programming partners, conduct the acceptance tests. In this manner, customers provide the unbiased verification that the application meets their needs. Customers create the acceptance tests from user stories. The ratio of user stories to acceptance tests is usually one too many; that is, more than one acceptance test may be needed for each user story.

Acceptance tests in XT may or may not be automated. For example, an unautomated test is required when the customer must validate that a user input screen meets its specification with respect to color and screen layout. An example of an automated test is when the application must calculate payroll values using data input via some data source such as a flat file to simulate production values.

Through acceptance tests, the customer validates an expected result from the application. A deviation from the expected result is considered a bug and is reported to the development team. If the customer discovers several bugs, then he or she must prioritize them before passing the list to your development group. After you correct the bugs, or after any change, the customer reruns the acceptance tests. In this manner, the acceptance tests also become a form of regression testing.

An important note is that a program may pass all unit tests but fail the acceptance tests. How is this possible? Because a unit test validates whether a program unit meets some specification, such as calculating payroll deductions, correctly, not some defined functionality or aesthetics. For a commercial application, the look and feel is a very important component. Understanding the specification, but not the functionality, generally results in this scenario.

Extreme Testing Applied

In this section we create a small Java application and employ JUnit, a Java-based open-source unit testing suite, to illustrate the concepts of Extreme Testing (see Figure 9.2). The example itself is trivial; the concepts, however, apply to most programming situations.

Our example is a command-line application that simply determines whether an input value is a prime number. For brevity, the source code,

JUnit is a freely available open-source tool used to automate unit tests of Java applications in Extreme Programming environments. The creators, Kent Beck and Erich Gamma, developed JUnit to support the significant unit testing that occurs in the Extreme Programming environment. JUnit is very small, but very flexible and feature rich. You can create individual tests or a suite of tests. You can automatically generate reports detailing the errors.

Before using JUnit, or any testing suite, you must fully under- stand how to use it. JUnit is powerful but only after you master its API. However, whether or not you adopt an XP methodology, JUnit is a useful tool to provide sanity checks for your own code.

Visit www.junit.org for more information and to download the test suite. In addition, there is a wealth of information on XP and XT at this website.

FIGURE 9.2 JUnit Description and Background.

`check4Prime.java`, and its test harness, `check4PrimeTest.java`, are listed in Appendix. In this section we provide snippets from the application to illustrate the main points.

The specification of this program is as follows:

> Develop a command-line application that accepts any positive integer, *n,* where 0<=*n*<=1,000, and determine whether it is a prime number. If *n* is a prime number, then the application should return a message stating it is a prime number. If *n* is not a prime number, then the application should return a message stating it is not a prime number. If *n* is not a valid input, then the application should display a help message.

Following the XP methodology and the principles listed in Chapter 5, we begin the application by designing unit tests. With this application, we can identify two discrete tasks: validating inputs and determining prime numbers. We could use black-box and white-box testing approaches, boundary value analysis, and the decision coverage criterion, respectively. However, the XT practice mandates a hands-off black-box approach, to eliminate any bias.

Test-Case Design We begin designing test cases by identifying a testing approach. In this instance, we will use boundary analysis to validate the inputs because this application can only accept positive integers within a certain range. All other input values, including character datatypes and negative numbers, should raise an error and not be used. Of course, you could certainly make the case that input validation could fall under the decision coverage criterion, as the application must decide whether the input is valid. The important concept is to identify, and commit to, a testing approach when designing your tests.

With the testing approach identified, the next step is to develop a list of test cases based on possible inputs and expected outcome. Table 9.3 shows the eight test cases we identified for this example. (Note: As stated, we are using a very simple example here to illustrate the basics of Extreme Testing. In practice, you would have a much more detailed program specification, which might include items such as user interface requirements and output verbiage. As a result, the list of test cases would increase substantially.)

TABLE 9.3 Test Case Descriptions for check4Prime.java

Case Number	Input	Expected Output	Comments
1	*n* = 3	Affirm *n* is a prime number.	Tests for a valid prime number. Tests input within boundaries.
2	*n* = 1,000	Affirm *n* is not a prime number.	Tests input equal to upper bounds. Tests whether *n* is an invalid prime.
3	*n* = 0	Affirm *n* is not a prime number.	Tests input equal to lower bounds.
4	*n* = - 1	Print help message.	Tests input below lower bounds.
5	*n* = 1,001	Print help message.	Tests input greater than the upper bounds.
6	*n* = "a"	Print help message.	Tests input is an integer and not a character datatype.
7	Two or more inputs	Print help message.	Tests for correct number of input values.
8	*n* is empty (blank)	Print help message.	Tests whether an input value is supplied.

Test case 1 from Table 9.3 combines two test scenarios. It checks whether the input is a valid prime and how the application behaves with a valid input value. You may use any valid prime in this test.

We also test two scenarios with test case 2: What happens when the input value is equal to the upper bounds and when the input is not a prime number? This case could have been broken out into two unit tests, but one goal of software testing in general is to minimize the number of test cases while still adequately checking for error conditions.

Test case 3 checks the lower boundary of valid inputs, as well as testing for invalid primes. The second part of the check is not needed because test case 2 handles this scenario. However, it is included by default because 0 is not a prime number. Test cases 4 and 5 ensure that the inputs are within the defined range, which is greater than or equal to 0 and less than or equal to 1,000.

Case 6 tests whether the application properly handles character input values. Because we are doing a calculation, it is obvious that the

TABLE 9.4 Test Driver Methods

Methods	Test Case(s) Examined
`testCheckPrime_true()`	1
`testCheckPrime_false()`	2, 3
`testCheck4Prime_checkArgs_char_input()`	6
`testCheck4Prime_checkArgs_above_upper_bound()`	5
`testCheck4Prime_checkArgs_neg_input()`	4
`testCheck4Prime_checkArgs_2_inputs()`	7
`testCheck4Prime_checkArgs_0_inputs()`	8

application should reject character datatypes. The assumption with this test case is that Java will handle the datatype check. This application must handle the exception raised when an invalid datatype is supplied. This test will ensure that the exception is thrown. Last, tests 7 and 8 check for the correct number of input values; any number of inputs other than 1 should fail.

Test Driver and Application Now that we have designed both test cases, we can create the test driver class, `check4PrimeTest`. Table 9.4 maps the JUnit methods in `check4PrimeTest` to the test cases covered.

Note that the `testCheckPrime_false()` method tests two conditions, because the boundary values are not prime numbers. Therefore, we can check for boundary value errors and for invalid primes with one test method. Examining this method in detail reveals that the two tests actually do occur within it. Here is the complete JUnit method from the `check4JavaTest` class listed in the Appendix.

```
public void testCheckPrime_false(){
  assertFalse(check4prime.primeCheck(0));
  assertFalse(check4prime.primeCheck(10000));
}
```

Notice that the JUnit method, `assertFalse()`, checks to see whether the parameter supplied causes the method to return a *false* Boolean value. If *false* is returned, the test is considered a success.

The snippet also demonstrates one of the benefits of creating test cases and test harnesses first. You may notice that the parameter in the

`assertFalse()` method is another method, `check4prime.primeCheck(n)`. This method will reside in a class of the application. Creating the test harness first forced us to think about the structure of the application. In some respects, the application is designed to support the test harness. Here we need a method to check whether the input is a prime number, so we included it in the application.

With the test harness complete, application coding can begin. Based on the program specification, test cases, and the test harness, the resultant Java application will consist of a single class, `check4Prime`, with the following definition:

```
public class check4Prime {
  public static void main (String [] args)
  public void checkArgs(String [] args) throws
    Exception
  public boolean primeCheck (int num)
}
```

Briefly, per Java requirements, the `main()` procedure provides the entry point into the application. The `checkArgs()` method asserts that the input value is a positive integer, *n,* where $0<=n<=1{,}000$. The `primeCheck()` procedure checks the input value against a calculated list of prime numbers. We implemented the sieve of Eratosthenes to quickly calculate the prime numbers. This approach is acceptable because of the small number of prime numbers involved.

Summary

With the heightened competitiveness of software development today, there is a growing need to introduce products very quickly into the marketplace. The Agile development process, when strictly adopted, provides a way for developers to create quality software for their customers at a faster rate than using traditional software development models. The end result is a satisfied customer, whether an internal or commercial consumer.

The Extreme Programming model is one of more popular Agile methodologies. This lightweight development process focuses on communication, planning, and testing. The testing aspect of Extreme Programming, termed Extreme Testing, focuses on unit and acceptance tests. You run unit tests during development and whenever a change to the code base occurs. The customer runs the acceptance tests at major release points.

Extreme Testing also requires you to create the test harness, based on the program specification, before you start coding your application. In this manner, you design your application to pass the unit tests, thus increasing the probability that it will meet the specification.