# Roadmap for Implementing Unit Testing

The purpose of this paper is to illustrate the importance of unit testing to preventing errors in software, and to explain how Parasoft can assist your organization in implementing unit testing practices as part of the larger methodology of Automated Error Prevention. This paper is the third part in a series of Parasoft white papers on Automated Error Prevention, following "*Understanding Workflow within a Coding Standards Implementation*" and "*Roadmap for Coding Standards Implementation*." We recommend reading the documents sequentially to understand all of the practices discussed. We will focus on Object-Oriented programming languages such as Java, C/C++, C#, Visual Basic, but many of the practices described can be applied regardless of the language (or platform/development process) you choose.

## INTRODUCTION

Automated Error Prevention methodology recommends best practices for improving software development processes, with the ultimate goal of improving software quality and reliability. Practices endorsed by the methodology include the use of coding standards, unit testing, regression testing, load testing, and monitoring, among others. Because it would be difficult to implement all of these practices simultaneously, most organizations begin to apply Automated Error Prevention by first adopting and enforcing the use of coding standards (industry guidelines for developing error-free software code).

Once you have coding standards and the ability to automatically enforce them in place, the next important step is to employ unit testing as part of your development group's standard processes. Unit testing is very much like applying coding standards to your code, except that the validation is performed dynamically rather than statically. When we enforce static coding standards, we simply read a piece of code and verify that it meets a set of coding standards. With automated unit testing, your developers code to the standards you define, and our solutions dynamically verify that the code was built correctly by actually running each piece of code.

## UNIT TESTING DEFINED

Many people use the term "unit testing" to mean functionality testing at the component level, but unit testing is much more than this. We define unit testing as testing software code at its smallest functional point, which is typically a single class. Each individual class should be tested in isolation before it is tested with other units or as part of a module or application. By testing every unit individually, most of the errors that might be introduced into the code over the course of a project can be detected or prevented entirely.

The objective of unit testing is to test not only the functionality of the code, but also to ensure that the code is structurally sound and robust, and able to respond appropriately in all conditions. We will discuss both structural and functional unit testing practices in this paper, along with the process we recommend for implementing them.

## OBSTACLES TO ADOPTION

While unit testing has been proven to reduce software errors, many of the development organizations we speak to resist the practice, usually because they misunderstand its value, and because unit testing can be very labor-intensive if performed manually.

Those who do perform unit testing often focus only on "black-box" (functionality) testing as a debugging mechanism. While this is important, it can leave as much as half of your code untested. However, using Parasoft "white-box" (structural) testing solution, the remaining could be checked automatically, and with very little cost. Parasoft can assist you in implementing a complete unit testing solution that will automates the most time-consuming and important practices, and improve the quality and reliability of your software.

## THE UNIT TESTING PROCESS: STRUCTURE BEFORE FUNCTION

We recommend that you begin unit testing by performing white-box testing on your code, then move on to black-box testing. Black-box testing validates that software works correctly when users perform a series of actions within the specification—its intent is to confirm behavior that is expected. White-box testing, on the other hand, is much like posing a series of "what if?" questions to determine whether your code behaves appropriately in unexpected conditions. This means verifying that any inputs you throw at the code will be received and addressed with the proper behavioral response. While both types of testing are critical, white-box testing can be automatically performed as soon as your first class is written and compiled, with little or no human intervention.

In the software industry, most developers without automated tools skip right to black-box testing and omit white-box testing completely, leaving a large percentage of their code untested. The challenge is that when developers write a piece of code, they are normally focused on making it work, not thinking how people might break it or use it in an unexpected way. Consider Microsoft as an example – the company designs its systems to provide a great deal of functionality that all of its users want – and doesn't specifically focus on security or uncommon uses for the functionality it creates. Unfortunately for users, there are people who don't like Microsoft, and who exploit any vulnerability they find in its software to distribute viruses, and cause a general level of mayhem for everyone.

In order to ensure that software will not only keep running under normal conditions, but also respond appropriately in unexpected conditions, software code should be tested for both construction and functionality at the earliest possible stage of development. This becomes increasingly important as we enter into new technologies such as Web services, where internal system interfaces are exposed to the outside world. If code in these systems is not tested

appropriately, its vulnerabilities can be used to break into the code and lead to a security risk (a memory leak or stolen pointer, for example) as well as performance issues.

## WHITE-BOX TESTING WITH DYNAMIC CODING STANDARDS

The first step in implementing white-box testing is to determine what criteria will be used to allow code to be checked into your source control system. This means defining dynamic coding standards that will be used during testing to identify bad code. Dynamic coding standards are similar to static coding standards, but they are enforced by actually executing the code during unit testing, rather than simply reading it during static analysis. We recommend four dynamic coding standards that relate to your code's ability to identify, classify, and respond to unexpected conditions:

### *Uncaught Exceptions*

The first dynamic coding standard to implement is *no uncaught exceptions allowed*. An uncaught exception occurs when your code receives inputs that cause it to enter a state it doesn't understand. "Exception handling" code should accompany methods to ensure that the code can continue to execute even if it receives unexpected inputs. If no exception handling code accompanies a given method, it should be commented to express how exceptions are escalated for handling elsewhere. Code that does not include either exception handling code or comments should be found during white-box testing, and should prompt a code review with your developer to discuss how and where exceptions should be handled.

Uncaught exceptions in any code can affect the functionality, stability, and security of the software, so they must always be detected and treated somehow. In C/C++ programming, uncaught exceptions will typically cause a software application to crash, and in Java, they could lead to a null pointer exception where the consequences are unknown. By implementing this dynamic coding standard, you can ensure that the code checked into source control has been tested for its ability to receive unexpected conditions without crashing.

When you first enforce a dynamic coding standard to prevent uncaught exceptions, it's likely that you will receive a number of objections to the requirement. For example, many developers have told us that a class they've written is never going to be used with a condition found through testing, because such arguments will never be passed to the code. They insist that because they wrote the code, it will continue to be used in the manner they intended. You should be prepared to address this objection, and you can do so by discussing the method's encapsulation.

## Why Uncaught Exceptions Shouldn't Occur

Consider how changing assumptions affect your development projects. While you may have an individual developer in charge of an application right now, someone else may be working on it a few months from now, and the original developer's assumptions will no longer be valid. Even before exposing your code to external factors, it may be accessed in ways that it was not originally intended.

To help developers create robust code and avoid such problems, object-oriented programming languages provide different levels of "encapsulation," or protection to control how pieces of code are used. Each class can be classified to determine how it can be accessed. Some of these classifications include:

a. ***Private -*** Only methods within the class can call a private class. Think of methods within a class as a group of people in the same room. Anyone in the room can talk to anyone else, but everything said remains private.
b. ***Protected -*** Protected methods are accessible to classes in the same package and classes in different packages that extend the class with the protected method. Consider people in different rooms speaking to one another, as well as employees in another building but the same company.
c. ***Public*** – Methods exposed as public can be accessed by any other method.
d. ***Package-private*** (the default encapsulation level for Java) – Package-private methods are only accessible to classes in the same package.

Many of the errors we see encountered by development organizations occur because a method is exposed as public but not tested to respond appropriately to unexpected conditions. If a method should not be faced with any conditions other than those it was designed to address, the class should be made protected or private to remove potential problems. Parasoft reports violations on only public methods by default, so those we report are critical and must be addressed to prevent anyone from accessing your code in unexpected ways.

If you don't prevent unexpected exceptions from being allowed in public methods, you are essentially trusting that no one will maliciously call your code once it's deployed, and you may be faced with security issues as well as reliability issues. The only way to test for these conditions is to perform white-box unit testing as part of your development process.

### *Testing Exception Handling Code*

The second dynamic coding standard we recommend requires *100% coverage of exception handling code*. Once you've ensured that your code can accept unexpected conditions, you'll need to ensure that it is able to react to the condition appropriately, which is called "exception handling." By requiring 100% coverage, you're saying that all exception-handling code must be executed at least once to ensure that it doesn't crash.

For example, you may have software in the refrigerator in your home or office that runs well under normal conditions. However, if the outside temperature were to suddenly increase, this should not crash the software (this would be an example of an uncaught exception). Instead, the software should decide what to do; it might increase power to compensate for the temperature increase outside, or it might shut off or sleep until something else changes to prevent the motor from burning out. Either way, the software must handle itself with an appropriate response to the unexpected condition.

Exception handling code often goes untested by development organizations because it's difficult to create such extreme conditions in a normal application environment. However, Parasoft can help you automatically perform this testing at the individual class level before moving any further in development. Parasoft solutions automatically create test cases to check such conditions. Your developers might have to extend the test suite by hand, but we can create a majority of the test

cases and inputs automatically to force your code to take exception conditions, and identify what conditions create the exception.

***Verification of Boundaries***

The third important step in white-box testing is to ensure that your code does not treat valid inputs as exceptions. Boundary conditions help your code determine whether an input it receives is valid or should produce an exception. Inputs at the boundaries of any valid range of inputs must be handled correctly without the code throwing an exception. We often see problems like security breaks occur because code was not subjected to this type of testing. To prevent such situations, you need to require verification of boundaries for all code before it can be checked into source control.

For example, a developer might create a method that expects a 16-bit number from -32,000 to 32,000 (for a range of 64,000). If the code is not created to expect a negative number but receives –32,000, it might treat this as 64,000 or something entirely different, with unknown consequences. Parasoft solutions automatically perform verification of boundaries to assist with this process. If an argument passed to a method is an integer, we will use a very large negative integer, a very large positive integer, zero, negative one, and a variety of other numbers that are on the boundaries of what is expected for verification. This ensures anything that is passed to a method and is outside its boundaries will be handled as an exception, and the code will react properly to anything within its boundaries.

***Code Coverage***

While the first three steps in white-box testing will assist your development organization in detecting and preventing a great deal of errors in their code, the final important step is to analyze and maximize code coverage. We recommend that you implement a dynamic coding standard requiring that *all code must have 100% line coverage* upon check in to source control.

Many coverage analysis tools can help you measure the amount of code coverage you've achieved, but expect you to create inputs manually. Parasoft technology is unique because our solutions actually induce coverage by automatically creating most test cases and inputs for you. Any remaining coverage necessary to achieve 100% can be extended by hand. For example, developers could use our tools to execute JUnit, CppUnit, or NUnit test cases and measure the amount of coverage these provide. Then, they could use Parasoft's automatic white-box testing to create additional test cases and inputs, and finally run both the existing and new test cases together. Our solutions automatically create inputs for all of them and you will see that we increase the amount of coverage. This is the final step in initial implementation of white-box testing.

## LINKING WHITE-BOX TESTING AND BLACK-BOX TESTING

After you've implemented initial requirements for white-box testing, the next logical step is to require contracts for public methods. This practice works much like a bridge between white-box testing and black-box testing, making white-box testing easier and allowing some black-box testing to be automated.

***Require Contracts***
We recommend that you require developers to publish contracts or assumptions for all of the methods they expose as public. This means that your developers will need to place a contract in front of each method they expose as public, to specify how it can be used. It explains that once we enter a method, it should be in a certain state (pre-condition), and once we exit the method, it should be in another state (post-condition) as the developer has defined. The contract becomes a public assumption to prevent code from being sent unexpected inputs, and it serves to minimize noise during testing.

In the Java world, Design by Contract is a standard that has emerged for just this purpose. There is currently no standard for C++ contracts, but they can be created in very much the same way as they are in Java. When you implement a coding standard that requires each public method to have a contract, we can dynamically analyze the code and report violations, and we can also use the contracts to automatically generate test cases and inputs for the methods to test their functionality.

When you begin using our solution to perform automatic white-box testing, it's likely that you will receive a great deal of violations that need to be fixed. However, as soon as your developers have added contracts to the appropriate methods or changed the encapsulation to protected or private, our tools will not create unexpected inputs for these methods, and the amount of violations will decrease.

The greatest benefit of going through this process is that it encourages your team to perform an automatic review of the code. Through analyzing the assumptions they've made about how the code was designed, your development organization can determine whether methods need to be exposed as public, or should be changed. You can then rely on the dynamic coding standards you've implemented to ensure that no exceptions are allowed.

***Require Preconditions***
Preconditions define when and how a specific method can be invoked. By adding preconditions to methods your developers can prevent them from being called in unexpected ways. To ensure that contracts include preconditions where necessary, we recommend that you require that contracts for all public methods *must have preconditions.*

***Require Postconditions***
The final step in implementing contracts within your code is to verify the output of the code by requiring a postcondition. We recommend that you require that contracts for certain methods *must have postconditions* (it wouldn't make sense to add post-conditions on all methods, so be selective). You can define which methods need them using different qualifications. For example, you could create rules that say if a class has a complexity above a certain amount, its contract must contain a postcondition.

The benefit of using postconditions is that when we test the code, we can begin to verify its functionality by ensuring that the method produces the same result that it is supposed to produce, as specified by the postcondition (the post-condition doesn't have to require a single result; it might define a range into which the result must fit). This is moving toward black-box testing, or specification verification, except that it's done at a granular level and very early in the process.

The use of post-conditions can be illustrated with a function that adds two integers. We could write a post-condition that states that after we've executed the code, we should receive an output that is equal to the sum of the two integers used as inputs. Functional testing is all about comparing – testing to make sure that results from the code match the results from the contract. You compare the results to verify that they're the same. If they're not, you know that you already have a problem with this code. The contract ensures that the output generated by your code adheres to whatever condition you've specified, and allows us to use the contract to automatically perform functionality testing at the class level.

## BLACK-BOX TESTING REQUIREMENTS

Up to now, most of what we've described (with the exception of writing or modifying code) is fully automated and can be implemented with very little human intervention. Next, we're going to talk about black box testing, which will require more effort on the part of your developers. If your development team built the application from scratch, this is the point when they've created enough functions or classes to have something to test, or have fully implemented the first specification requirement.

In many cases, your development team will be using a unit testing a framework such as JUnit, CppUnit, or NUnit, might begin to write and execute tests at this point. Because black-box testing requires human intervention, you want them to perform it as effectively as possible. As we discussed earlier, you can perform some functionality testing automatically using contracts, but the rest will need to be extended manually. Your goal is to achieve complete coverage of the specification, meaning that for every piece of code, your developers should have a corresponding specification and test case.

### *Require 100% Specification Coverage*

In order to ensure that each specification requirement is met, we recommend that you require 100% coverage of the specification.

Many of your functionality test cases will fail initially if they're created at the same time as the code they're intended to test. But over time, as your developers continue to add code to address the functionality specifications, these failures will diminish. If your tests were built to cover all of the functionality required in the specification, you can be confident that the code meets the specification when it passes all of the tests successfully.

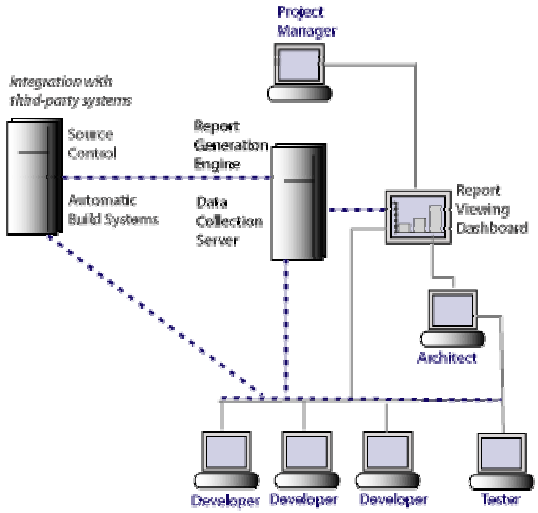
## PERFORM ONGOING REGRESSION TESTING

As you implement each of the unit testing practices we've described, your development organization will continue adding to the code base, so that you start with single classes, then groups of classes, sub-modules, modules, and finally a fully functional application. It is important to perform regression testing on a regular basis to ensure that none of the changes introduce errors into your code base.

In addition to running static and dynamic analysis at the developer desktop and when code is checked into source control, we configure your system to perform all of the same tests during the nightly build process. All of the results will be tracked and reported to ensure that no new errors have been introduced.

## PARASOFT SOLUTION

All of the unit testing practices we've discussed are implemented using the same organizational workflow we've discussed for static coding standards. Parasoft solutions provide a team infrastructure that addresses each of the key roles within your development organization and automates the most time-consuming and important unit testing practices.




We connect the solution to your source control and automatic build systems, allowing architects to define dynamic coding standards that require unit testing to be performed on all of the code entered into your source control system. Developers use the automatic unit testing functionality on their desktops to create and execute test cases.

All of the information captured from test runs by each developer and automatic build process is stored and analyzed by the Parasoft Enterprise Solutions Global Reporting Server (GRS). Developers, architects, and project managers can log into the system through a standard web browser and view customized reports based on their role. Team members can view the quality and coverage of code, as well as the performance on a group, project, or developer level.

Using this information about your organization's unit testing activities, you can continually improve your development process. For example, you might implement new coding standards to address the most common errors found during unit testing, to actually prevent errors that would otherwise be introduced into the code over the course of the project.

To learn more about Parasoft Enterprise solutions infrastructure, read the Parasoft white paper "*Configuring Your Automated Error Prevention Infrastructure.*"

## IMPLEMENTATION PROCESS

In the best possible cases, all code is written with structural and functional testing in mind, however, most organizations find themselves in the midst of a variety of development projects, with an immediate need to test and improve code quality. Parasoft Professional Services group will guide your organization through the complete implementation process to help you achieve immediate benefits from unit testing.

Our services team will guide you through a phased implementation of unit testing requirements, training your team members not only to modify code and requirements, but also how to use results to improve the process. The implementation plan allows us to introduce each requirement based on priority and degree of automation, minimizing the amount of violations or "noise" that your team members could receive, and providing you with meaningful information.

The first step to begin the process is to choose a cut-off date for code that will be subjected to the unit testing requirements you establish. This ensures that your developers won't be overwhelmed by the need to fix all of the code at once, which could prevent the practice from being implemented effectively. Once the cut-off date is decided, we guide you through the unit testing practices and automate as many of them as possible.

The diagram below illustrates the phased implementation process.

**Unit Testing Implementation Process**

| Practice | Priority | Implementation Phase | | |
|---|---|---|---|---|
| **A. White-box Testing** | | **1** | **2** | **3** |
| *1. Uncaught exception testing* | *Must have* | X | | |
| *2. Testing exception-handling code* | *Should have* | | X | |
| *3. Boundary checking* | *Nice to have* | | | X |
| *4. 100% line coverage* | *Nice to have* | | | X |
| **B. Contracts** | | | | |
| *1. Contracts for public methods* | *Should have* | | X | |
| *2. Preconditions* | *Should have* | | X | |
| *3. Postconditions* | *Should have* | | X | |
| **C. Black-box Testing** | | | | |
| *1. 100% spec coverage* | *Must have* | X | | |

## Use Unit Testing Results To Improve Your Process

Unit testing is only a piece of Automated Error Prevention methodology that provides information to feedback to the rest of the process. Our services team will help you determine how to effectively use the results generated by the unit testing practices. When errors are detected, you'll be able to analyze the process and determine how it can be modified to prevent that error from reoccurring. It's critical that you gather information from outside the process, use the data from each of the test runs and from all of the members of your development team, and improve the process to make sure that errors don't happen again.

One of the most important factors in implementing unit testing or any other Automated Error Prevention practice is the ability to track your organization's progress and use the information you gain to improve the entire processes. You'll receive comprehensive reports that allow you to view and analyze data from each phase in your process. You'll learn to measure the success of all tests and how to implement process changes to improve the quality of your code.

## Conclusion

Unit testing is an important Automated Error Prevention practice that allows you to detect errors, and to modify your organization's development processes to prevent future errors from being introduced into code. Performing thorough unit testing reduces allows you to ensure the quality, security, and reliability of your code from the earliest stages of development, and drastically reduces the potential for errors. Parasoft solutions provide you with the tools, best practices, and knowledge that enable you to automate unit testing and ensure quality throughout your software lifecycle.

## Additional Resources

To learn more about Automated Error Prevention, see: "*What is Automated Error Prevention: Using Error Prevention to Reduce Cost, Improve Quality, and Facilitate Change*."

For information about implementing coding standards, see: "*Understanding Workflow within a Coding Standards Implementation*" and "*Roadmap for Coding Standards Implementation*."

For more information about how Parasoft Enterprise Solutions architecture supports the key practices in Automated Error Prevention, see: "*Configuring Your Automated Error Prevention Infrastructure*."

For additional information about unit testing and coding standards for Java applications, see: "*Automatic Java™ Software and Component Testing: Using Jtest to Automate Unit Testing and Coding Standard Enforcement.*"

## References


1. "Glossary of Terms" http://java.sun.com/docs/books/tutorial/information/glossary.html
2. "Controlling Access to Members of a Class" http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html