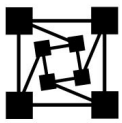


ParaSoft®

Debugging the Software Development Process:

A How-to Approach for Developers and Managers

Introduction

Do you want to know what development process will ensure the release of the highest quality product on time and on budget? It's whatever development process best suits your project and controls bugs the most effectively. Fortunately for developers and their managers, the same bug-control practices can be applied to any development process; this means that once you master these techniques, you can easily apply them to whatever types of development processes your current and future projects demand. This paper will help you determine which types of processes best suit which types of projects, then it will introduce bug-control practices that improve quality while reducing development time, cost, and effort.

Considering the variations in training, expertise, projects, tools, etc. across development teams, it is not surprising that different development teams have different development processes. This variety of processes is beneficial: certain processes will be better suited for certain projects than others, and the more processes that are available, the easier it will be to find one that perfectly suits the needs of the project at hand. However, despite this variety, all development processes share the same fundamental components. If you look at any development process that has been used to successfully develop a product, you will see that it contains four phases:

- Design
- Implementation
- Integration
- Testing

The position (relative to one another) and the length of these phases are what distinguish one development process from another. When viewed in terms of the position and length of these four phases, every software development process is similar to either the waterfall model or the Extreme Programming model.

The waterfall model contains long design, implementation, integration, and testing phases. It is likened to a physical waterfall because its progression is a steady, irreversible flow towards the destination: just as a waterfall always moves downward towards a river, stream, or lake, a waterfall software development process always moves downward towards the next phase and towards the release.

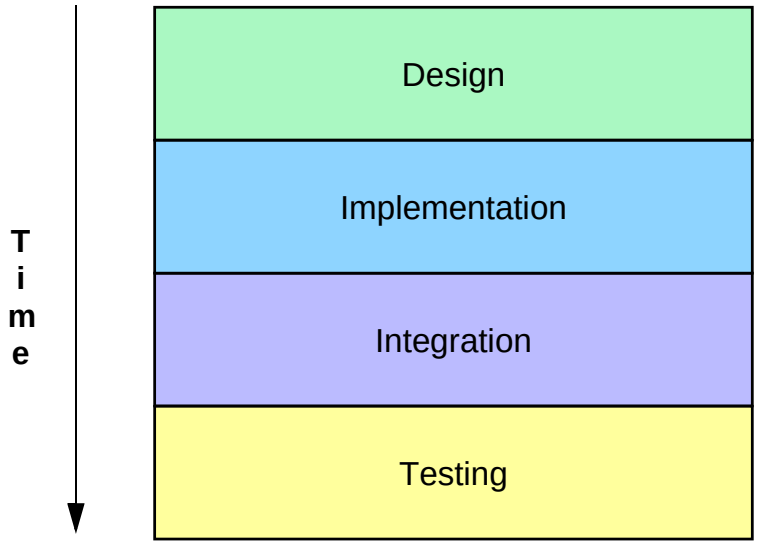


Figure 1: A Waterfall Development Process

The waterfall model is only efficient when changes are not made late in the process: a significant change in integration or testing can require much costly re-design, re-implementation, and re-coding. With that comes increased expenses, a much longer development process, and an increased likelihood of introducing errors. Because of this model's inability to accommodate late changes, it is best-suited for well-known, well-defined projects. For example, a waterfall model would be a wise choice if you were developing an accounting application and had definite plans to include all of the traditional accounting features. Projects with vague or rapidly-changing requirements generally should not be developed using a waterfall model.

If you have a project with vague or rapidly changing requirements, the Extreme Programming model is a better solution than the waterfall model. In Extreme Programming, an application is developed in a series of many brief design-implementation-integration-testing iterations, each of which implements the features that an on-site customer decides are critical for that release. Releases do not occur after the entire application is "finished," but rather after a single iteration has successfully implemented the features that the on-site customer requested for that iteration. Releases will thus be more frequent in this model than they are in the waterfall model, and the difference from version to version will probably be less noticeable.

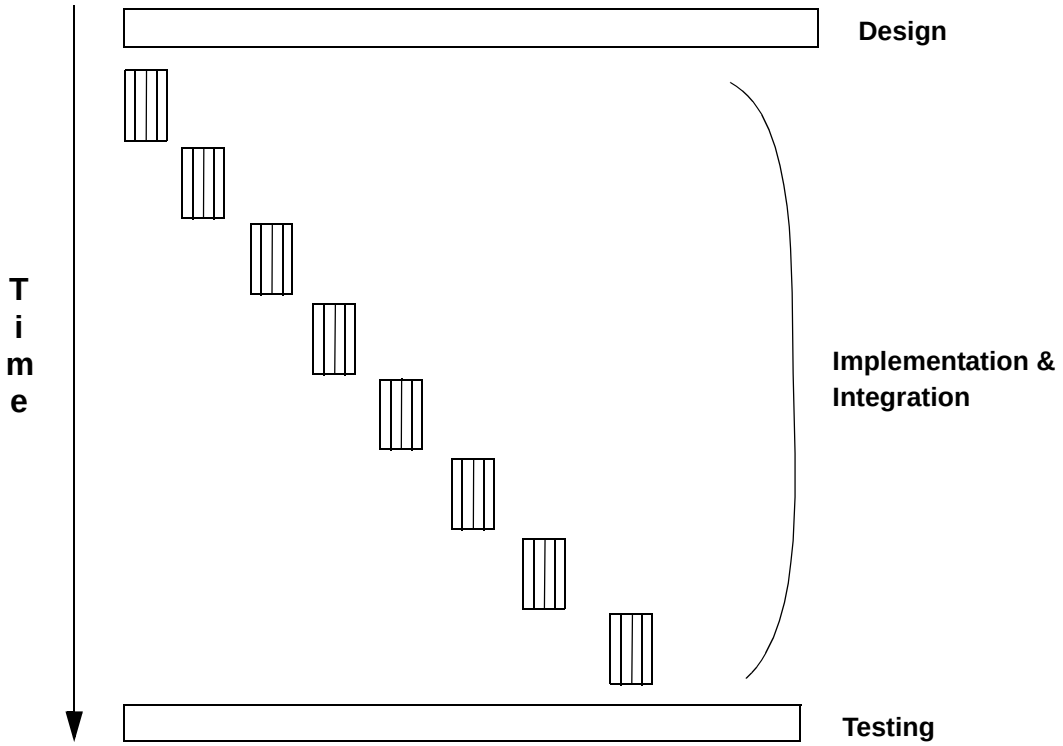


Figure 2: An Extreme Programming Development Process

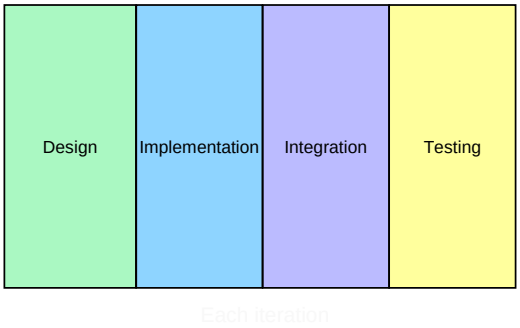


Figure 3: One Iteration of an Extreme Programming Development Process

It is interesting to note that the series of iterations resembles one waterfall-like process if you look at the process from a distance (as shown in Figure 2): much design is performed before a single iteration starts, the series of small iterations themselves resemble an implementation and integra-

tion phase, and a large amount of testing is performed after the completion of all iterations, when the application is truly complete.

Extreme programming's many small iterations allow it to accommodate the sort of frequent and late changes that would cause a project in the waterfall model to overshoot its release date and budget. This accommodation of change makes this model particularly well-suited to situations where it is impossible to have a clear idea of the project's scope (for example, if you are developing a unique, ground-breaking application, or an application that targets a rapidly-changing market such as the Internet market). This model's series of frequent iterations makes it easy for you to release a working version of the product as soon as possible-- so customers can use the product and decide what additional features are needed.

These two models are the two extremes available; there are also many other models that fall somewhere in between the one long iteration of the waterfall model and the countless iterations of the Extreme Programming model. No matter what type of development process you use, you can ensure its success by integrating the following practices into it:

- Focusing your work on necessary, important features.
- Keeping bugs under control and preventing them from increasing exponentially.
- Automating as much of the development process as possible.

We will discuss the most effective way to integrate these practices into the four fundamental phases that all development processes share. You can then integrate them into different development processes by adapting the associated tasks' position and length according to the position and length of the phases that they are integrated into.

Focusing Work on Important Features

The key to focusing your work on important features is eliciting feedback so that you know what features are indeed most important. Unless you are developing a product for software developers or managers exactly like yourself or your own team, you must elicit some degree of outside feedback to discover what features customers really want and to ensure that these features are implemented in a way that the customer deems both usable and valuable.

The type and degree of feedback that you need depends on the completeness and stability of your project's specification. If you are developing a stable product, you should aim for specific customer feedback. If you are developing a new, breakthrough product, you should first try to elicit market feedback, then later aim for customer feedback.

The best way to elicit feedback depends on your company and targeted market, and lies beyond the scope of this paper. The only bit of advice that we offer on this matter is that if you are serious about receiving a substantial amount of feedback on feature after feature, you might want to consider using an Extreme Programming development process: this process facilitates feedback because it allows you to deliver early betas and deliver releases incrementally.

Controlling Bugs

In a bad development process, bugs are not controlled; rather, they are introduced, then ignored until the final stages of the development process. This is more dangerous than most people realize: when you allow bugs to enter and remain in your code, the bugs build upon and interact with one another. This interaction usually has the critical effect of causing bugs to increase exponentially instead of increasing linearly with time and number of lines of code. When this happens, the amount of bugs will dramatically increase as time and lines of code increase-- often to the point where they cause the project to be cancelled.

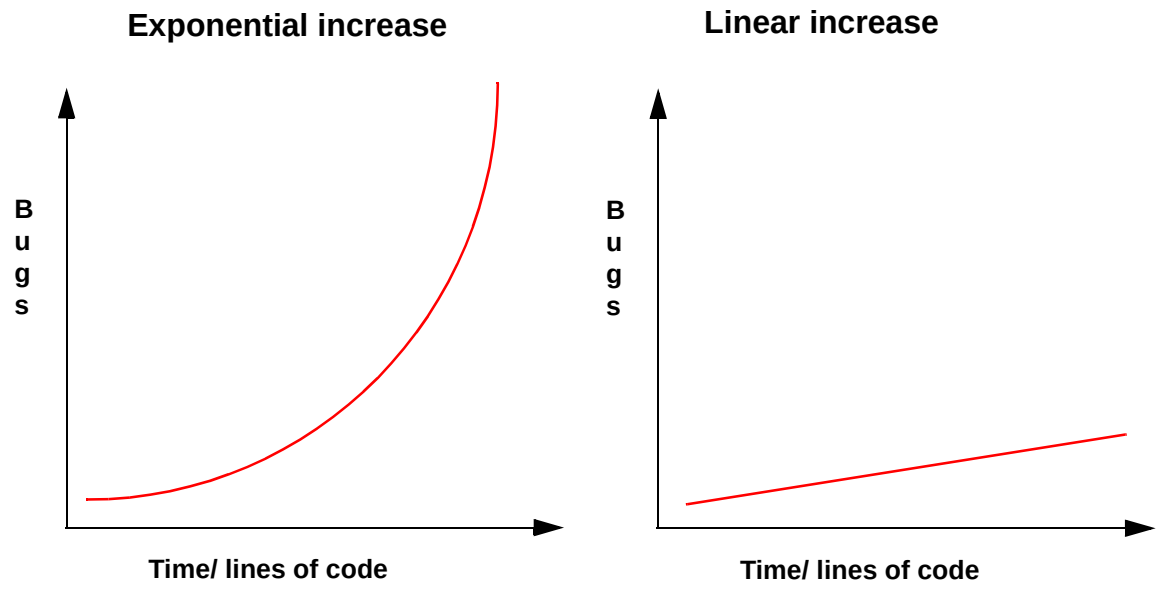


Figure 4: Exponential Increase vs. Linear Increase

The key to controlling bugs is preventing them from increasing exponentially. You can do this by preventing as many errors as possible and finding and fixing existing bugs as early as possible. As Figure 5 demonstrates, you can prevent bugs from increasing exponentially by integrating error-prevention and error-detection measures into your development process. The specific error-prevention and error-detection techniques shown in the diagram will be discussed later in this paper.

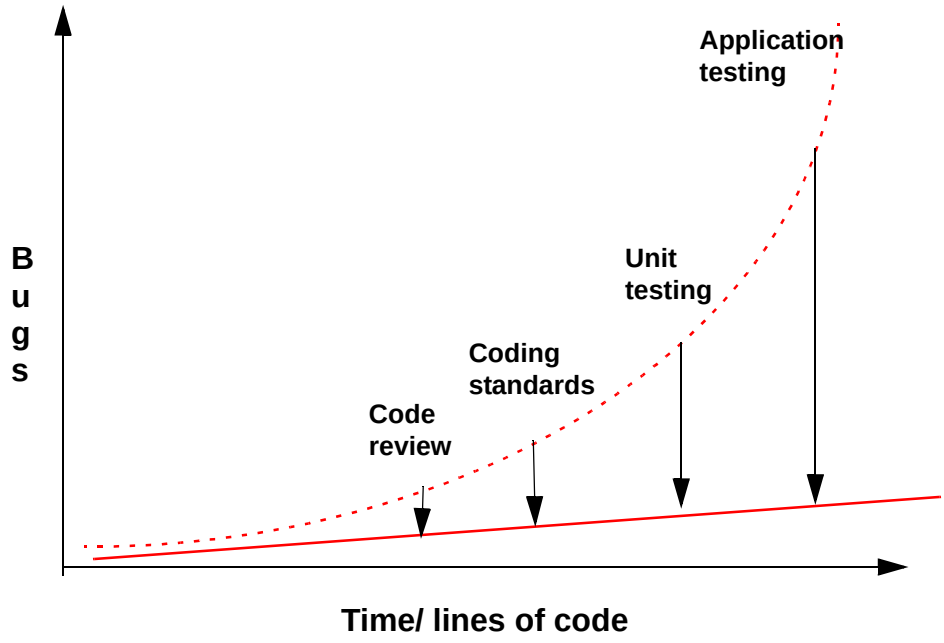


Figure 5: The Effect of Bug-Control Efforts on Number of Bugs

The first step in preventing errors is realizing that bugs can indeed be prevented. One of the greatest hurdles in keeping bugs under control is the widely-held belief that bugs are inevitable. This is completely false. Errors don't just happen; every error appears in code because a developer introduced a defect into the code. Humans are not flawless, so even the best developer will occasionally introduce defects when given the opportunity to do so. The key to preventing errors is thus reducing the opportunity for making errors. As we will explain later in this paper, one of the best ways to do this is to implement and enforce coding standards for all developers, in all languages, every day.

Once you have realized that bugs can be prevented, you can start taking steps towards controlling them. The best way to control bugs is to ask yourself two questions every time that you find an error:

1. How could I have automatically detected this bug?
2. How could I have prevented this bug?

Routinely asking and answering these questions every time that you find an error will help you prevent similar errors from occurring in the future, as well as improve your ability to detect as many bugs as possible. Because bugs build upon one another, each bug prevented or detected early in the process usually means not just one, but *many,* fewer bugs to find and fix later.

A final step in controlling bugs is preventing the errors that you have found and fixed from reentering code as you modify it. This process ("regression testing") is discussed in more detail later in this paper.

Many developers and managers claim that they do not have the time or money to spend on such bug-control efforts. The fact is that those who are most concerned about releasing products on time and on budget are the ones who are most in need of such practices. Study after study has confirmed that:

- Focusing on error prevention results in shorter development schedules and higher productivity.
- The longer a defect remains in the system, the more expensive and difficult it becomes to remove.

Capers Jones' findings in a study of IBM practices nicely sums up the benefits of constantly preventing errors, detecting errors, and improving error detection:

> "Projects that aim from the beginning at achieving the shortest possible schedules regardless of quality considerations tend to have fairly high frequencies of both schedule and cost overruns. Software projects that aim initially at achieving the highest possible levels of quality and reliability tend to have the best schedule adherence records, the highest productivity, and even the best marketplace success."

Automating the Development Process

Although bug control can and should reduce development time and cost, it does not always do so. While bug control contains great potential for slashing development cost and time, this potential is often not realized because the inefficiencies of the development process do not make bug control a feasible and efficient strategy. Specifically, if the development process and bug-control measures are not as automatic as possible, they may end up consuming almost as much time, money, and effort as they have the potential to save, and your project quality will not improve as much as it could if you had automated your development and bug-control efforts. If a development process is going to successfully control bugs and-- at the same time-- reduce development time, effort, and cost-- it needs to have the following elements built into it:

- A source code repository
- Nightly builds
- A bug-tracking system
- Automatic development tools

Source Control Repository

A source code repository establishes a central place where the entire source base can be stored and accessed. When you use a source control repository, you can not only track the history of the

code, but also improve efficiency by ensuring that revisions are not carelessly overwritten. The ability to revert back to archived versions also allows you to take risks with your revisions, and gives you the option of starting over again when so many bugs have been introduced that re-coding is easier than debugging.

Tools that can be used to establish a source code repository include GNU RCS, GNU CVS, and Rational ClearCase. You can use these tools as is, or customize them to your team's specific needs by wrapping them with your own scripts.

Nightly Builds

Nightly builds automatically build the application every night. The minimal nightly build should pull all necessary code from the source code repository, clean and compile that code, then build the application. The ideal nightly build should also run all available test cases (both unit test suites and application test suites) and report any failures that occur. At least, this process minimizes the overhead involved in assembling the application pieces. At best, it ensures that the application continues to run as expected and detects any errors introduced by newly integrated code.

Bug-Tracking System

A bug-tracking system such as GNU GNATS or Mozilla Bugzilla has two main uses. The first and most important use is to record and track all errors that were not detected by your test suite. Loyally entering every bug found into the system facilitates problem tracking and provides valuable data about the types of errors that teams or developers tend to make-- data that can be used to hone error-prevention and error-detection efforts. Ideally, the system used should ensure that the appropriate people are automatically notified about the problem, and it should correlate bugs to source versions.

The second use of a bug-tracking system is to record feature requests that are not yet being implemented. Having a reliable method for storing features facilitates the design phase of the next iteration: all features can easily and quickly be recalled when it is time to debate the next feature set.

Automatic Development Tools

Automatic development tools come in many flavors; for bug-control purposes, you want automatic development tools that will:

- Prevent and detect errors at the unit level.
- Detect application-level errors.
- Perform regression testing.

The time that you spend evaluating multiple tools and finding the best solution will pay off in the long run. The time spent evaluating a tool can easily be regained if you find a tool that automates more processes than another, or one that lets you find and prevent more bugs than another. The development tools that will help you control errors most effectively are those that:

- **Contain the most effective technology:** The tools with the best technology will find and prevent the most errors. If a tool does not effectively find or prevent errors, all of its other features are irrelevant.
- **Require minimal user-intervention**: Compare how much user-intervention each tool requires. Look for features like automatic creation of test cases, harnesses, and stubs, easy ways to enter user-defined test cases, etc..
- **Are customizable:** The better you can tailor the tool to your specific team and project needs, the more efficiently you will be able to control bugs.
- **Have interactive and batch modes:** Look for a tool that you can run interactively and in batch mode. Interactive mode should be used as you are testing newly-developed code and fixing errors found; batch mode should be used during the nightly build to perform regression testing.
- **Integrate with other infrastructure components:** Many development tools can be integrated into the compilation and building process. Such integration is helpful in ensuring that no new errors are introduced and catching new errors as soon as possible-- when the code is fresh in the responsible developer's mind, and before that error spurs additional errors.

Controlling Bugs With Tools and Automation

Once you have all of the necessary tools to automate your development process, you can start integrating them in a way that controls bugs. You should position your testing tools as gates that prevent developers from progressing until they have prevented and/or detected as many errors as is possible at their current development phase. If you use gates in this way, you can:

- Prevent problems from spurring errors.
- Prevent errors from spawning more errors.
- Ensure that errors can be found and fixed as easily, quickly, and cheaply as possible.
- Ensure that you do not make continue introducing the same problems and errors into the code.

The other tools should be used to build an infrastructure that supports the use of the automatic development tools.

Controlling Bugs During Design

Bug control should be an issue as early as the design phase. The first step in the design phase is to determine what features should be implemented in the current iteration (either one brief Extreme Programming iteration, or one long waterfall iteration). To do this, the developers and managers should make a master list of all possible feature candidates (including those entered in the bug-tracking system), then-- with the help of customer and/or market feedback-- decide which features should be implemented for the current release. Chosen features should be assigned to specific

team members; the remaining features should be recorded in the bug-tracking system, so that they can easily be accessed when it is time to design subsequent iterations.

After the feature set is selected and specific tasks are assigned, developers need to determine how to lay out the new code. In doing so, they should strive for flexible classes and objects. The design that is the most simple, readable, and flexible is the design that will foster the least amount of errors as the code is modified. This process can be automated with the use of CAD tools such as Rational Rose.

The design phase should conclude with a design review where developers explain their designs to one another. Simply explaining the design sometimes exposes complexity and ambiguity that can later lead to errors during the initial implementation or during modification.

Controlling Bugs During Implementation

Because implementation is the phase where most bugs are introduced, it is the prime phase to perform bug control. The main steps in controlling bugs during implementation are:

- Performing code reviews
- Enforcing coding standards
- Performing unit testing
- Using test suites as gates

Performing Code Reviews

After code is written, developers should get together and perform a code review. This review is similar to the design review, but instead of verbally explaining their design, the developers verbally explain their code. As in the design review, problems that could later lead to errors are often exposed during the developer's verbal explanation.

In some companies, code reviews also involve coding standard enforcement. We do not believe that coding standard enforcement should be included in code reviews. Coding standards can be enforced more quickly, precisely, and objectively when they are enforced automatically. Thus, it makes little sense to waste precious developer time on an inferior method of coding standard enforcement.

Enforcing Coding Standards

Coding standards are language-specific "rules" that, if followed, will significantly reduce the opportunity for developers to introduce errors into an application. Coding standards do not uncover existing problems; rather, they actually prevent errors from occurring. The best way to explain what coding standards are and how they work is to show you some examples.

Examples

Coding standards could prevent problems in the following C++ code:

```
1:   class FOO {
2:   private:
3:      char *cptr;
4:   public:
5:      FOO(char *x) {
6:         cptr=strdup(x);   // ignore null case for this example
7:      }
8:      virtual ~FOO( ) {
9:         if(cptr) free(cptr);
10:      }
11:   };
```

The developer has written a class that contains a pointer member, but has not defined a copy constructor. The compiler will copy the class using a default copy constructor, and the pointer will be copied to a new class. The outcome will only be correct if the developer did not intend to create a new memory location. Failing to define a copy constructor is a very complicated logical error, and the best way to expose this kind of error is to enforce coding standards. Coding standards will remind you to write a copy constructor in cases like these.

Here's an example of Java code that contains an error that can be prevented with coding standards. In this case, the developer is using external resources and intends to make a clean exit from the code using the file.close command:

```
1:   public class ReadFile {
2:      public String read (String path)
3:         throws IOException
4:      {
5:         FileInputStream file =  newFileInputStream (path);
6:         String contents = readFile (file);
7:         file.close ( );
8:         return contents;
9:      } //…
10:   }
```

Although this code appears to run well, the developer may run into trouble the next time he or she tries to deal with external resources. Potentially, the code will throw an exception. In this example, if the method ReadFile throws an exception, control will be transferred out of the method read and the file will not be closed. Incidentally, Java will generally attempt to close the file for you. However, you cannot rely on implementation to make this work. The solution to this predicament is to write a finally clause when you are using external resources. When you enforce coding standards, you can automatically be reminded to use the finally clause in cases like these.

In the example below, the developer has made a common HTML mistake involving a menu:

```
1: <HTML>
2: <HEAD><TITLE>Test Page</TITLE></HEAD>
3: <BODY>
4: <FORM ACTION="www.parasoft.com" METHOD=GET>
5: <SELECT NAME=MENU SIZE=3 MULTIPLE>
6: <OPTION>This is option 1
7: <OPTION>This is option 2
8: </SELECT>
```

```
9: <OPTION>This is option 3
10: </FORM>
11: </BODY></HTML>
```

In this case, the <OPTION> tag in line 9 should have been nested between the <SELECT> and </SELECT> tags. Because it was not, the text that follows that <OPTION> tag will not be listed as an option: the text “This is option 3" will be printed outside of the menu (as shown in Figure 6), and users will not be able to select this option.

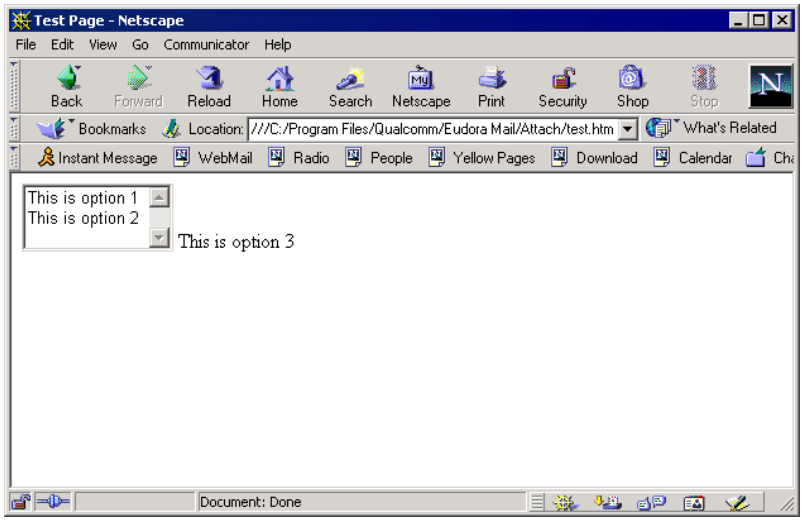


Figure 6: An HTML Problem that Coding Standards Could Prevent

Once again, this problem could have been prevented by enforcing coding standards.

Enforcing Coding Standards

There are generally two types of coding standards that help you prevent errors:

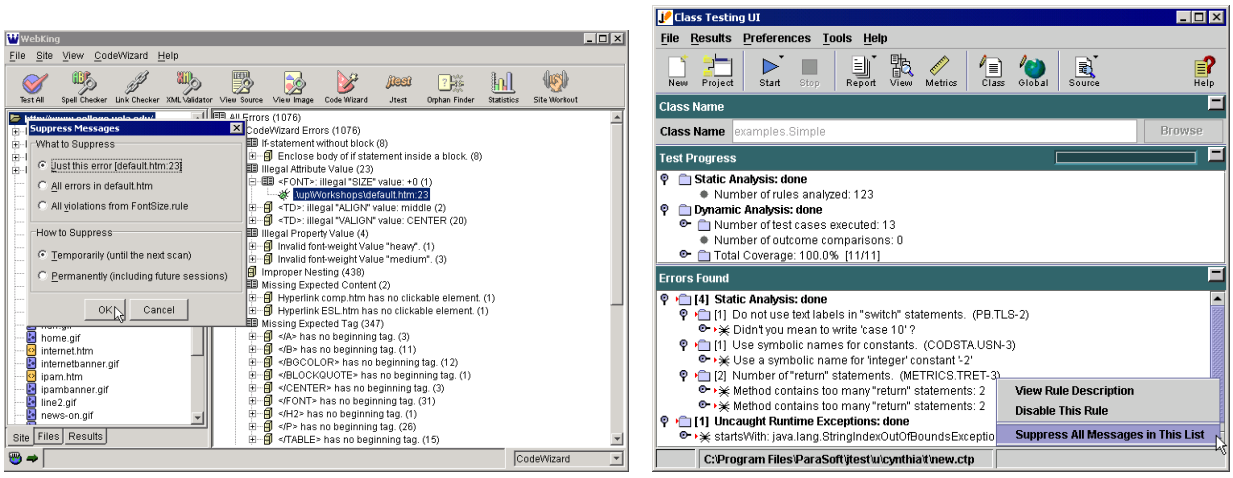
- **Industry-wide coding standards:** Rules that are accepted as “best practices” by experts in the given language. (For example, the C++ coding standard “Write `delete` if you write `new`”, or the Java coding standard “Use `StringBuffer` instead of `String` for nonconstant strings”).
- **Custom coding standards:** Rules that are specific to a certain development team, or even a certain developer. There are two types of custom coding standards: company coding standards and personal coding standards. Company coding standards are rules that are specific to your company or development team. For example, a rule that enforces a naming convention unique to your company would be a company coding standard. Personal coding standards are rules that help you prevent your most common errors. Every time that

you make an error, you should determine why it occurred, then design a personal coding standard that prevents it from reoccurring.

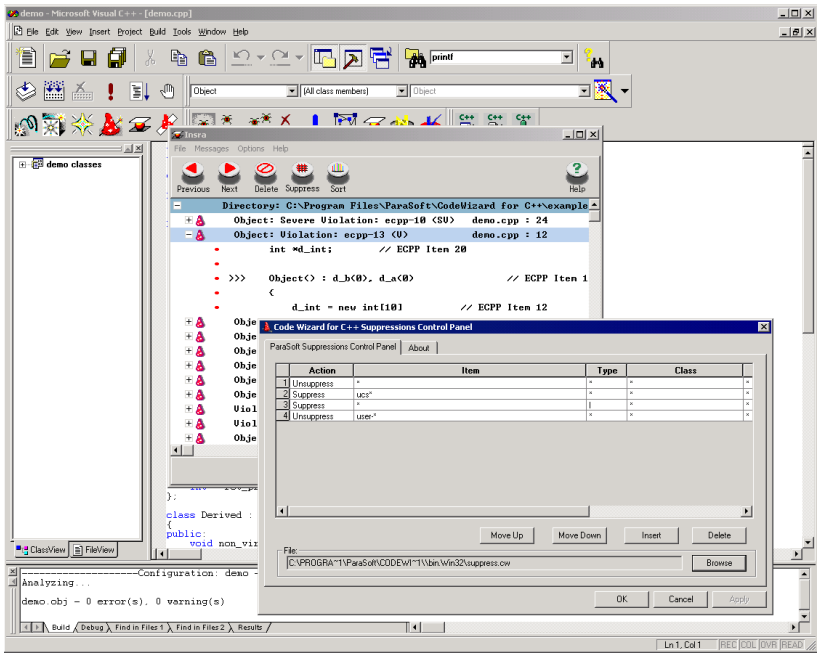
Because coding standards are designed to prevent bugs rather than detect them, you should use coding standards all the time, in all languages, to reduce the possibility of errors. However, every coding standard is not important for every development team, and moving from no coding standard enforcement to enforcing every possible coding standard can be overwhelming. That's why you should implement coding standards incrementally. First, your development team should meet and decide which coding standards you eventually want to enforce. Then, divide these standards into several groups based on the severity of their violations (does a violation have a 100% chance of introducing an error? a 75% chance? a 10% chance?). When you are ready to start enforcing standards, enforce only the standards that belong in the group with the highest severity. As the team becomes comfortable with that set of standards and generally avoids violating these standards, you can also enforce the second most severe set. When developers are comfortable with that set, they add the next severe set. And so on. No matter what stage of enforcement developers are in, all appropriate coding standard violations should be cleared before code is checked into the source code repository; this prevents the error-prone code from spawning errors or becoming increasingly error-prone, and forces the developer to face and fix the problem before he or she continues to introduce the same problem into new code.

From what you have read so far, coding standards enforcement may seem like a lot of work. It is. That's why it is best to enforce coding standards automatically with a tool that offers a set of meaningful industry-wide coding standards, and that lets you easily create and enforce meaningful custom coding standards. The best examples of tools that incorporate all of these vital features are CodeWizard (for C/C++), Jtest (for Java) and WebKing (for Web development languages). These tools automatically enforce a comprehensive set of meaningful coding standards for their respective languages. To enforce coding standards with these tools, you simply need to specify which file or files you want to check, then click a button. Or, check coding standards in the background by setting up batch processing or integrating these tools into your makefile. Each violation message contains the exact line of the violation and a brief explanation of the problem; the associated help file provides a more detailed description of the problem, an example of the problem, and a suggestion on how to repair the problem.

It is easy to customize these tools to your development team's projects and priorities. Each tool divides its coding standards into thematic and/or severity categories, so it is easy to determine which rules you do and do not want to enforce. Once you decide which coding standards you want to enforce, you can customize the tool by creating flexible suppression options or disabling/enabling specific rules or rule categories.



WebKing **Jtest**



CodeWizard

Figure 7: Suppressing Coding Standard Violations

All of these tools also contain the RuleWizard feature, an easy-to-use GUI that lets you create your own company and personal coding standards. With RuleWizard, you create custom rules by graphically expressing the pattern that you want to look for during automatic coding standard enforcement. Rules are created by selecting a main “node,” then adding additional elements until the rule expresses the pattern that you want to check for. Rule elements are added by pointing,

clicking, and entering values into dialog boxes. You can also use this tool to customize many of the built-in coding standards included with each tool.

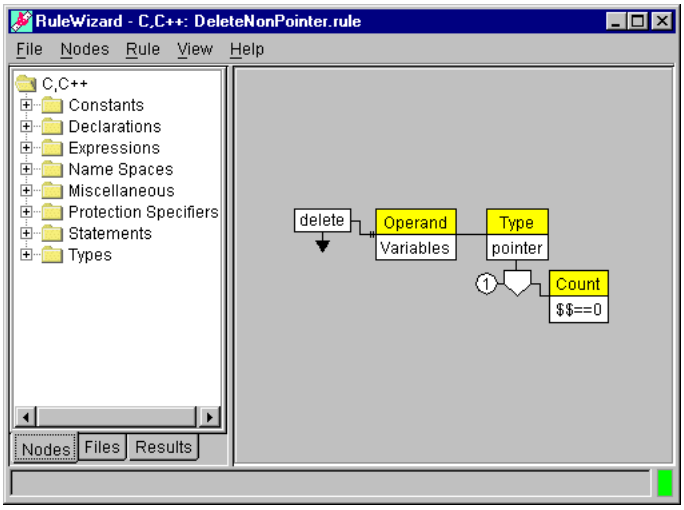


Figure 8: The RuleWizard GUI Lets You Create Custom Coding Standards

Performing Unit Testing

Unit testing involves testing the smallest possible unit of an application. In terms of C++ and Java, unit testing involves testing a class as soon as it is compiled. Unit testing is universally recognized as an essential component of the software development process. Unit testing practitioners enjoy such benefits as easier error detection, which has the very desirable end result of increasing software quality at the same time that it reduces development time and cost. How does unit testing facilitate error detection? First of all, when you test at the object level, you are much closer to the methods, and have a much greater chance of designing inputs that actually reach errors and of achieving 100% coverage (see Figure 9). Second, if you test code this soon after it is written, you will not have to wade through layer after layer of errors in order to find and fix a simple error: just find the single, simple error, and that problem is solved. This easier error reduction leads to reduced development time, effort, and cost because less time and resources are consumed finding and fixing errors.

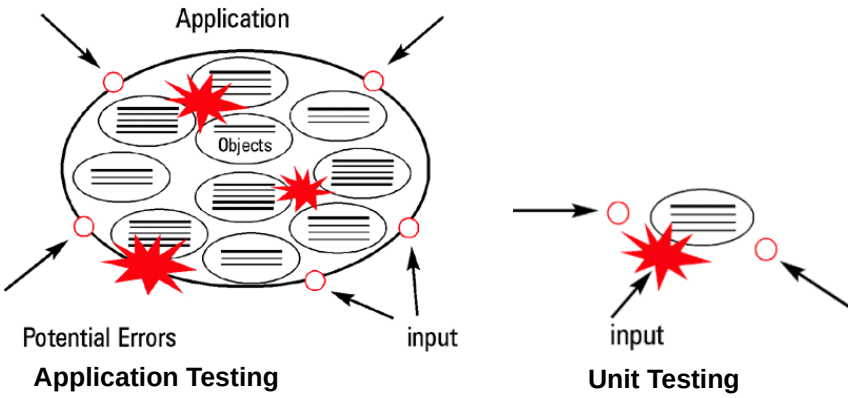


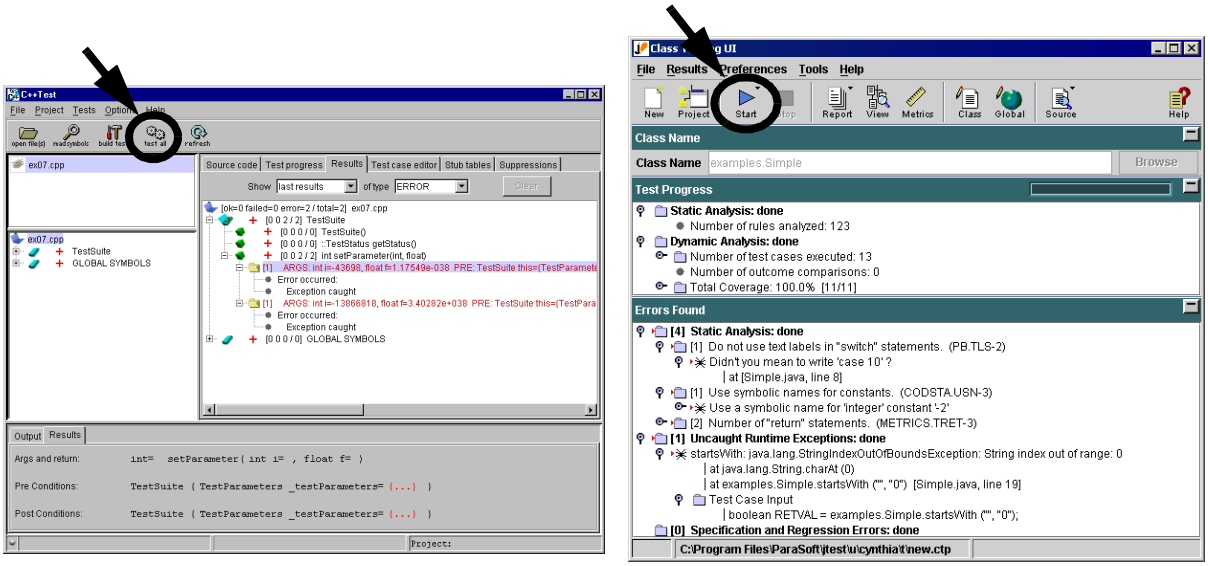
Figure 9: Ease of Reaching Errors With Unit Testing

However, unit testing can be difficult to perform. Just making a unit testable is often difficult: making a C++ or Java class testable usually requires the creation of scaffolding and stubs, and making a dynamic Web application's output page testable requires the deployment of the program as well as the creation of a specific instance of the related output page. In addition, unit testing involves several complex types of testing:

- White-box testing: Ensures that code is constructed properly and does not contain any hidden weaknesses.
- Black-box testing: Ensures that code functions in the way that it is intended to function.
- Regression testing: Ensures that modifications do not introduce errors into previously correct code.

Fortunately, there are ways of integrating unit testing into your development process that will not only improve quality, but also save you significantly more time and resources than it consumes.

If you have the right tool, unit testing can improve the quality and efficiency of any type of development process and any type of development. If you are developing in C, C++, or Java, you can use C++Test (C/C++) or Jtest (Java) to automate your unit testing. As soon as you have written and compiled a class or function, you can simply run it through C++Test or Jtest, and with the click of a button, the tool instantly makes the class/function testable by automatically designing any necessary harnesses or test stubs. These tools can then automatically perform white-box testing and regression testing, and automate as much of the black-box testing process as is possible (these testing techniques are described in the following sections). This almost completely diminishes the overhead involved in performing unit testing.



C++Test **Jtest**

Figure 10: ParaSoft's Unit Testing Tools Can Make a Class/ Function Testable With a Click of a Button

Web developers can also automate unit testing-- with a twist. When it is applied to Web development, unit testing takes a slightly different form: Web-box testing. We will explain this Web version of unit testing before we proceed to explain the unit testing components that apply to all flavors of unit testing (including Web-box testing).

Web-Box Testing

Unit testing can and should also be applied to Web site development. Because critical dynamic Web site problems (such as load-related problems) are often the result of design or implementation flaws, fixing these problems frequently requires redesigning and/or rewriting the entire application. However, if you test each servlet, bean, or other type of program immediately after it is written (i.e., perform unit testing), you can spot and resolve critical flaws before they become widespread. Essentially, you can prevent many problems that would be difficult and costly to fix.

WebKing's Web-box testing feature automates the unit testing process for dynamic Web site developers. After you perform a few quick configuration steps, WebKing will automatically deploy a script or program, then test the program and the page(s) that it produces. Essentially, it acts as a harness for testing that page, just as scaffolding and stubs act as a harness for testing an object apart from the entire application. Because you do not have to manually deploy the program and create each page, you can quickly and easily focus on the task at hand: ensuring the quality of the program that you just developed or modified.

WebKing's Web-box testing feature facilitates the unit testing process by automating the tasks you perform as you deploy and test programs and related output pages (site units). After you create or modify a program, you typically compile it, perform any necessary initializations, publish it on the Web server, then manually invoke at least one instance of each output page related to the program. WebKing gives you the infrastructure to automatically perform these functions with just a click of a button. Essentially, it acts as a harness for testing that page, just as scaffolding and stubs act as a harness for testing an object apart from the entire application. Because you do not have to manually deploy the program and create each page, you can quickly and easily focus on the task at hand: ensuring the quality of the program that you just developed or modified.

The goal of Web-box testing is to help you test your programs or scripts at two levels-- the program level and the output level-- as early as possible. The precise meaning of “testing at the program level” depends upon the language in which the program was written. For example, it could mean exposing all exceptions (Java), exposing core dumps (C/C++), exposing other programming failures (Visual Basic and other languages), or making sure that scripts do not exit. “Testing at the output level” means testing that the HTML page that the program returns is correct and error-free. By testing at both of these levels, you can expose problems associated with all aspects of the program, and fix these problems at the stage when it is easiest, fastest, and cheapest to do so.

Before you perform Web-box testing, you need to establish an infrastructure for automatically building, deploying, and testing the program or script. In WebKing, you can set up this infrastructure by entering commands into dialog boxes.

You can then automate the deployment and testing process by:

1. Clicking a button to compile, deploy, and test the program.
2. Indicating what paths related to this program you want to test, or configuring WebKing to automatically create paths through and around this site unit.
3. Clicking a button to perform load testing, white-box testing, and black-box testing.

These tests can be saved so they can replayed for unit-level regression testing, extended into application-level tests, and passed over to testers when the site is ready for the QA stage.

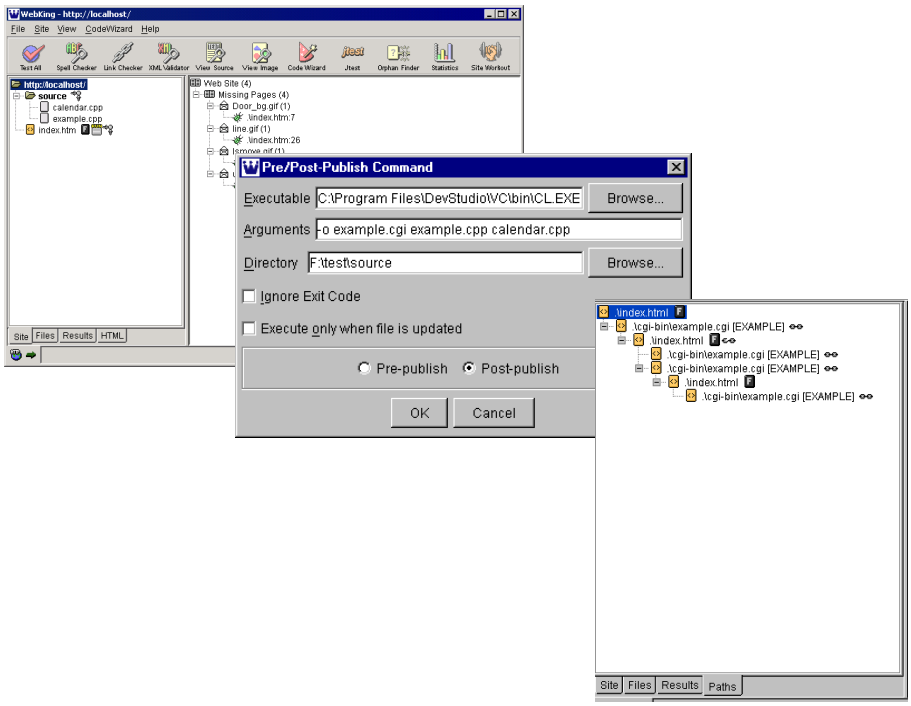


Figure 11: WebKing Can Automatically Build, Deploy, and Test Your Programs After a Few Simple, One-time Configuration Steps

White-Box Testing

White-box testing checks that code is robust by determining if it performs correctly when it encounters unexpected input. This type of testing must be performed with full knowledge of the unit's implementation details. The goal of white-box testing is to execute every branch of code under different input conditions to uncover any abnormal behavior.

White-box test cases should uncover defects by fully exercising the code with a wide variety of inputs. However, this is incredibly difficult to do manually. To create effective white-box test cases, you must examine the given unit's internal structure, then write test cases that will cover all of the code as fully as possible, and uncover inputs that will cause the unit to crash. Achieving the scope of coverage required for effective white-box testing mandates that a significant number of paths are executed. For example, in a typical 10,000 line program, there are approximately 100 million possible paths; manually generating input that would exercise all of those paths is infeasible.

Though incredibly beneficial to code quality, white-box testing is one of the most difficult and time-consuming testing techniques available. Practically speaking, the only way to perform white-box testing with today's development schedules and budgets is to automate it. Not only is

automatic white-box testing faster, easier, and cheaper than manual white-box testing, but it is also more precise and thorough.

ParaSoft's Jtest (for Java), C++Test (for C/C++), and WebKing (for Web applications) are the premiere white-box testing solutions available. All of these tools completely automate the white-box testing process. C++Test and Jtest start by creating any necessary harnesses or stubs. Next, they scrutinize the code under test, then design and execute inputs designed to thoroughly exercise that code. This process uncovers crashes (C/C++) and exceptions (Java). When WebKing performs white-box testing, it automatically exercises the site unit by designing and traversing realistic user paths through the unit. As it traverses each path, it tests for problems such as bottlenecks, programs that cannot handle certain types of user traffic, servlets that throw exceptions, CGIs that core dump, databases that crash, multiple types of broken links, and a wide variety of HTML, CSS, and JavaScript problems that could affect navigation, security, presentation, and performance. Tests are developed instantly, then run automatically.

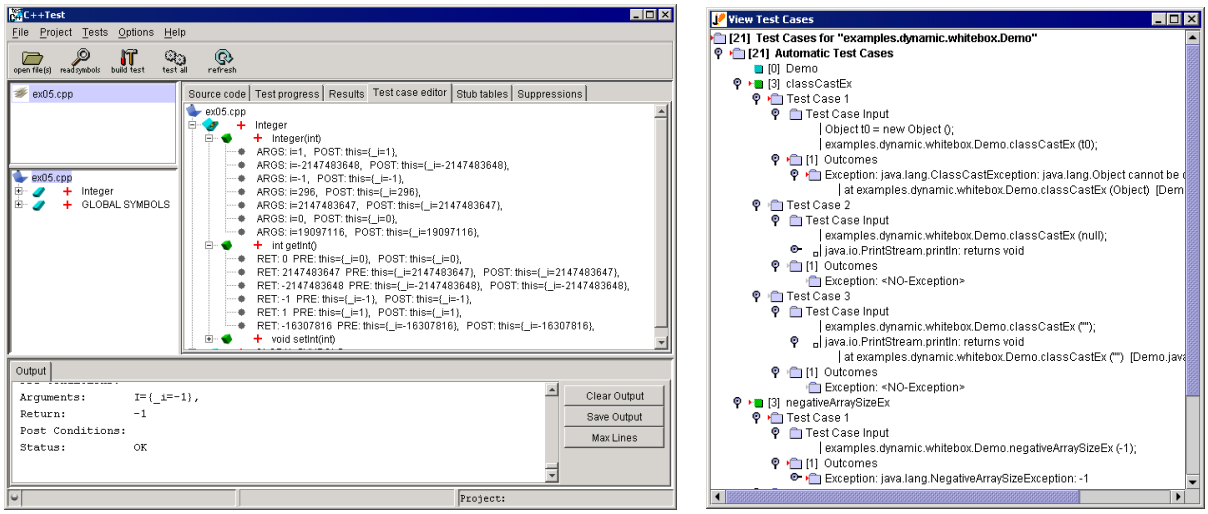


Figure 12: Automatically Generated White-Box Test Cases

Errors found are reported in an easy-to-read tree structure, and test parameters and results can be saved to facilitate regression testing.

Black-Box Testing

Black-box testing checks code's functionality by determining whether or not the unit's public interface performs according to specification. This type of testing is performed without paying attention to implementation details.

Black-box testing generally involves the following steps:

1. Creating a test plan based on the unit's specification.
2. Designing inputs that will test the functionality (i.e., developing a test suite).
3. Running the test suite.
4. Verifying that the output is correct.

Black-box test cases should be based on the specification document. Specifically, at least one test case should be created for each entry in the specification document; preferably, these test cases should test the various boundary conditions for each entry. Additional black-box test cases should be added for each error that is uncovered, and for any other tests that you deem necessary.

In cases where the specification is incorporated into the code (for example, code using the Design by Contract principle integrates specifications into code by means of specialized comments), it is possible to automatically generate and execute all necessary black-box test cases. However, under most circumstances, the specification is not incorporated into the code itself. In these cases, the best thing to do is use a tool that generates a comprehensive set of test cases during white-box testing. You can use the automatically-generated white-box test cases as a base black-box testing suite, then add any additional test cases that are needed to fully test the functionality detailed in your specification document.

ParaSoft's Jtest (for Java), C++Test (for C/C++), and WebKing (for Web applications) are the most efficient unit-level black-box testing solutions available. All of these tools automatically build a base black-box test suite during white-box testing. These tests are designed to achieve maximum coverage, but because the tools cannot access your specification, they cannot design tests that check every aspect of the unit's intended functionality. That's why these tools all offer you easy ways to expand the test suite by adding your own test cases. In Jtest, you can add test cases by adding method inputs directly to a method's tree node, selecting constants and methods that you added to global or local repositories, or adding test classes for test cases that are too complex or difficult to be added as method inputs. C++Test lets you add test cases by entering simple inputs in a test case skeleton, and entering object-type inputs in an Object Editor. WebKing makes it easy to test the specific site functionality that you are interested in by offering a simple way to create and test critical paths though the site (for example, the multiple paths available from selecting an item to actually completing a transaction). You can extend an automatically-generated set of paths and inputs by clicking links and entering form inputs in WebKing's special browser.

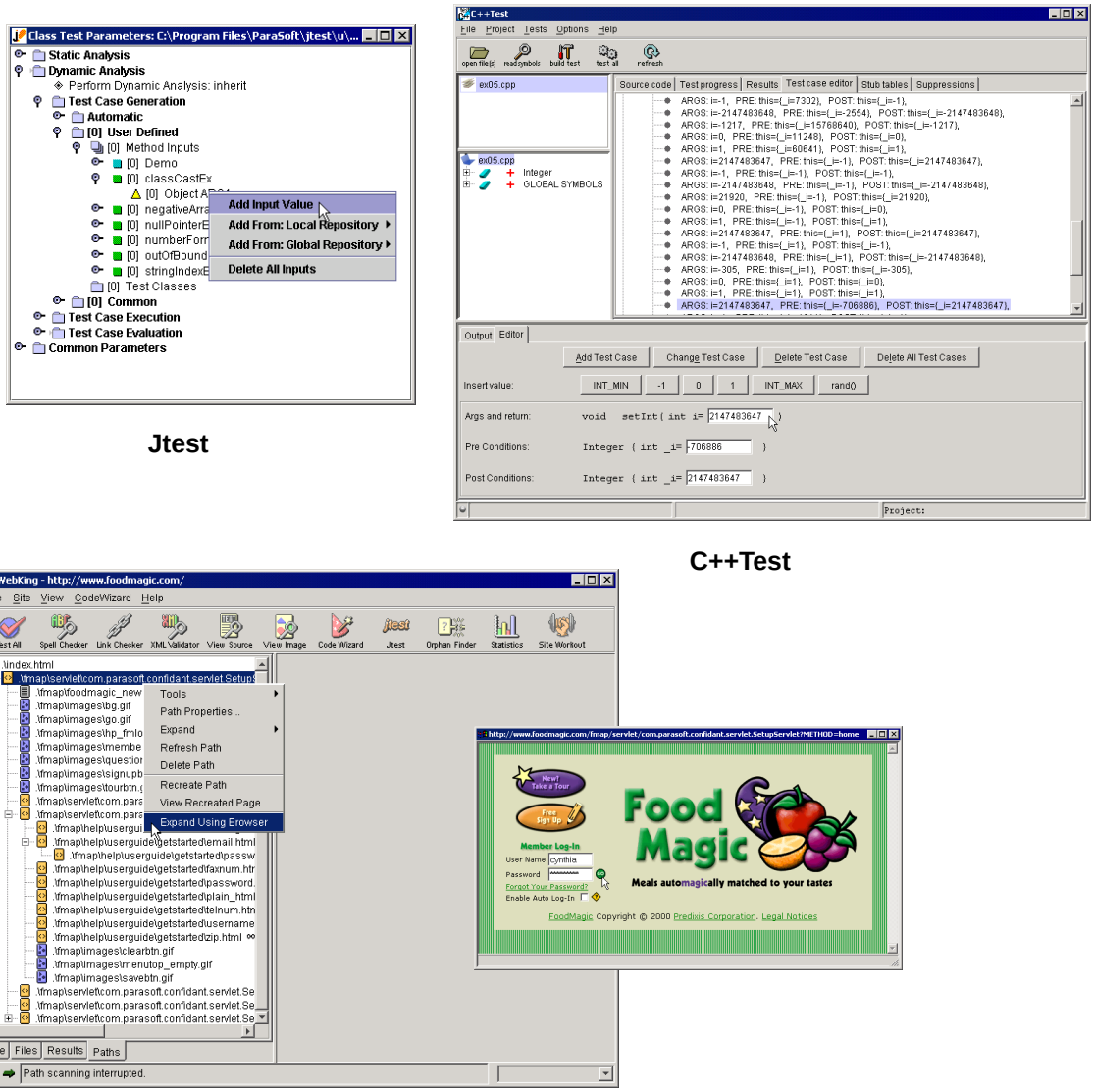


Figure 13: Adding User-Defined Black-Box Test Cases

These three tools also let you save black-box inputs and outcomes with a click of a button; this makes unit-level regression testing virtually effortless.

No tools that automatically create black-box test cases based on actual specifications are currently available, but ParaSoft will be releasing such a tool soon.

Completing Unit Testing

When performing white-box or black-box testing immediately after completing or updating a unit, it is best to run the unit testing tool/Web-box testing tool in interactive mode (i.e., by opening

the GUI, customizing test parameters, and seeing feedback as it is produced). As you find errors, remember to determine the cause of each error, then design a coding standard that prevents that type of error from reoccurring. You should then fix the errors found, and re-run the same tests until all construction and functionality problems are fixed. Code with unit-level errors should not be integrated into the application.

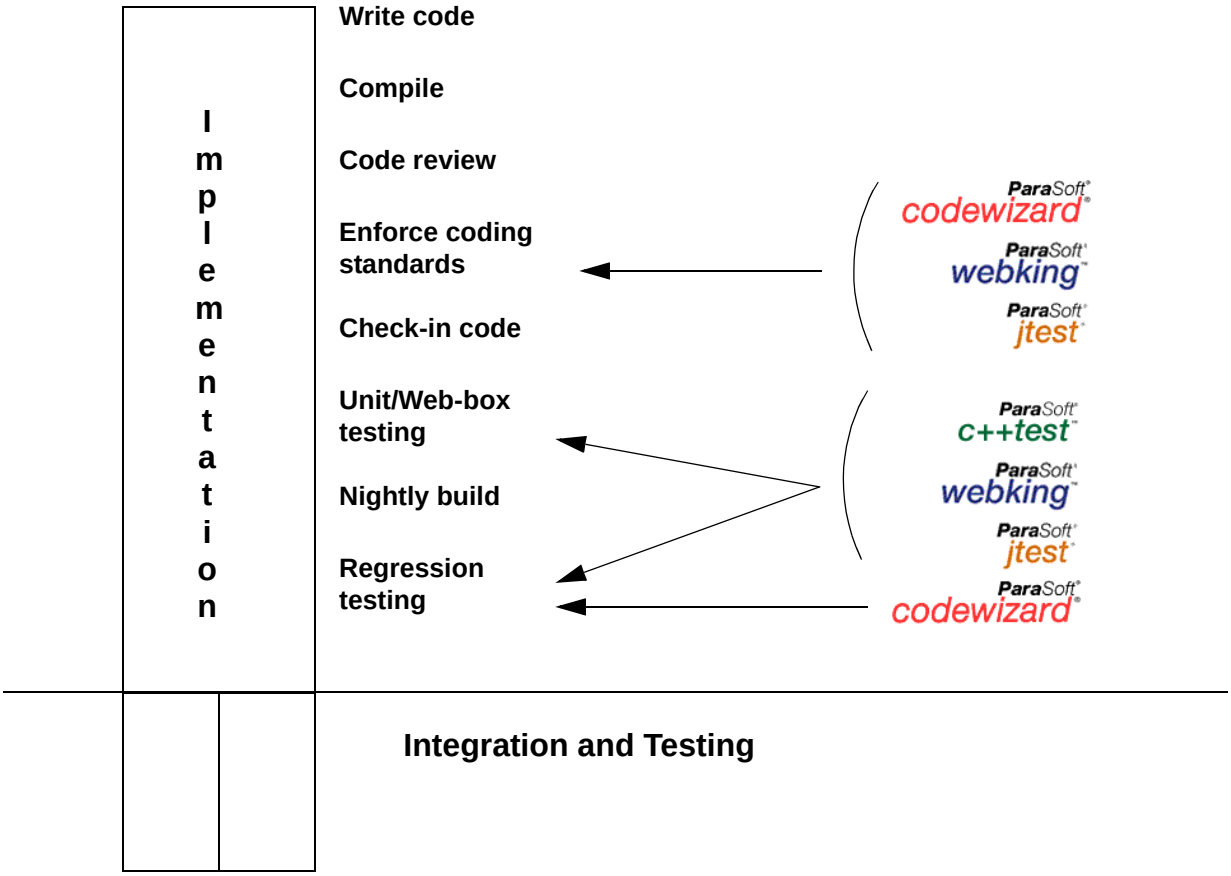


Figure 14: Integrating Tools into Implementation

In sum, before you proceed out of implementation and into integration and testing, you should perform a code review, enforce coding standards, perform unit or Web-box testing, and correct all problems and errors found. In addition, you should start performing nightly builds. Nightly builds should begin as soon as you write your first chunk of code; at this phase in the development process, your nightly build should compile and build the code, then have your unit testing tool(s) run your entire test suite to ensure that changes have not introduced errors (i.e., perform regression testing). The test cases that are run at this point should be the same test cases that you used when you performed unit testing. Because you and the other developers have presumably corrected all unit-level errors prior to the build, you should perform regression testing by running the tools in

batch mode (rather than interactive, or GUI mode), and simply check the report each morning to determine if any errors were found. C++Test, Jtest, and WebKing can all be run in this manner.

There are two ways you can perform regression testing. The first way is to have a developer or tester examine every test case and determine which test cases are affected by the modified code. This approach uses human effort and time to save the computer work.

A more efficient approach is to have a computer automatically run all test cases every time the code is modified. When you take this approach, you can preserve precious and costly developer time because you do not need to have a developer examine the entire test suite to determine which test cases need to be run and which do not. Even if you need to add additional systems to run all of your test cases in a reasonable period of time, you will save money: it is always cheaper to add systems than to pay developers good money to perform menial tasks.

Controlling Bugs in Integration and Testing

When you are ready to start building the whole product, you should add integration into your nightly builds. After performing integration, your nightly builds should perform the following tests in the background:

- Application-level black-box testing (including checking for invariable elements, where applicable)
- Application-level white-box testing
- Unit- and application-level regression testing

These measures will ensure that any errors related to the interaction between units are detected as soon as possible, find problems (such as memory corruption) that cannot be detected until integration, and ensure that modifications do not introduce new errors into previously clean, functional code.

Application-Level Black-Box Testing

Application-level black-box testing checks whether the entire application performs according to specification. If you performed unit testing, you can be confident that each unit works as expected, but you must wait until the integration phase to determine if the units interact according to specification. As soon as you start to build the application, you should start building an application-level black-box testing suite. This suite should include a test case for every aspect of the program's application-level functionality and for every error detected.

Tools that can automate black-box testing at the GUI level include Mercury's WinRunner and Segue's SilkTest; ParaSoft's WebKing can automatically test GUI functionality at the HTML level.

A type of application-level functional testing that applies only to Web applications is checking for invariable elements. The greatest difficulty involved in testing a dynamic Web application's func-

tionality is dealing with its ever-changing content and appearance. Contents change based on the date, the user, and the users' actions. This flexibility makes it difficult to test if a certain GUI element (for example, a button, image, advertising banner, navigation bar, etc.) occurs where it should occur. Record/playback tools could test for such elements' presence in static pages, but attempting to use such a tool on dynamic sites would result in too many false positives to make it worth the effort.

We have found that this problem is best solved by using the Graphical Scripting Language featured in ParaSoft's WebKing to describe what content, design, or presentation features you do (or do not) want to appear in specified pages. By using a language that purposely allows ambiguous tests, you can avoid false positives. For example, if you wanted to check that every page with a certain title contained a calendar, you could use the Graphical Scripting Language to write a test (such as the one displayed in Figure 15) that said to check for any page with the given title, and in each of those pages, check if there is a calendar that contains a valid month name, valid day names, and a valid number of dates. This test would flag only pages with missing or incorrect calendars. Because record/playback tools flag *any* difference-- even intended differences-- as errors, these tools would report false positives on every day except for the day when you recorded the test.

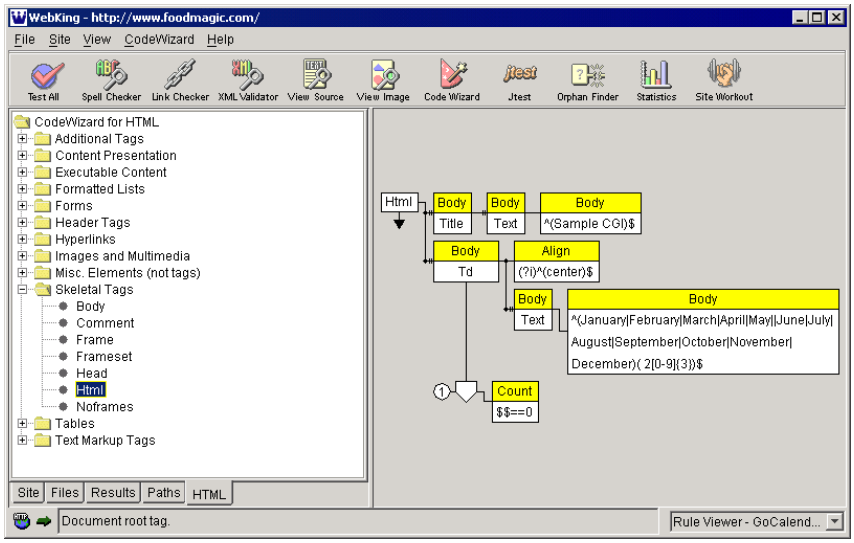


Figure 15: Example Rule that Tests Functionality

Another type of application-level black-box testing for Web site development is testing critical paths at the path level. This involves taking the unit level black-box tests that test specific paths through each unit and extending them so that they test paths through the entire application.

Application-Level White-Box Testing

Application-level white-box testing examines the construction and performance of the entire application.

In terms of C/C++, this type of testing involves checking for memory problems such as memory corruption/uninitialized memory, memory leaks, memory allocation errors, variable definition conflicts, stack memory problems, I/O errors, and the like. It should also look for logic and algorithmic errors.

Because complete coverage is often difficult to achieve at this level, you must be vigilant about monitoring these tests' coverage: there is an incredible difference between uncovering 10 errors when your test suite has covered the majority of your application and discovering 10 errors when your test suite has only covered 2% of the application. Without coverage data, error-found information is a useless metric. More importantly, without coverage data, you will never know how many serious problems may be lurking in the untested parts of your code. You should not consider this phase of testing complete until you have covered 70-80% of the full-featured application's code.

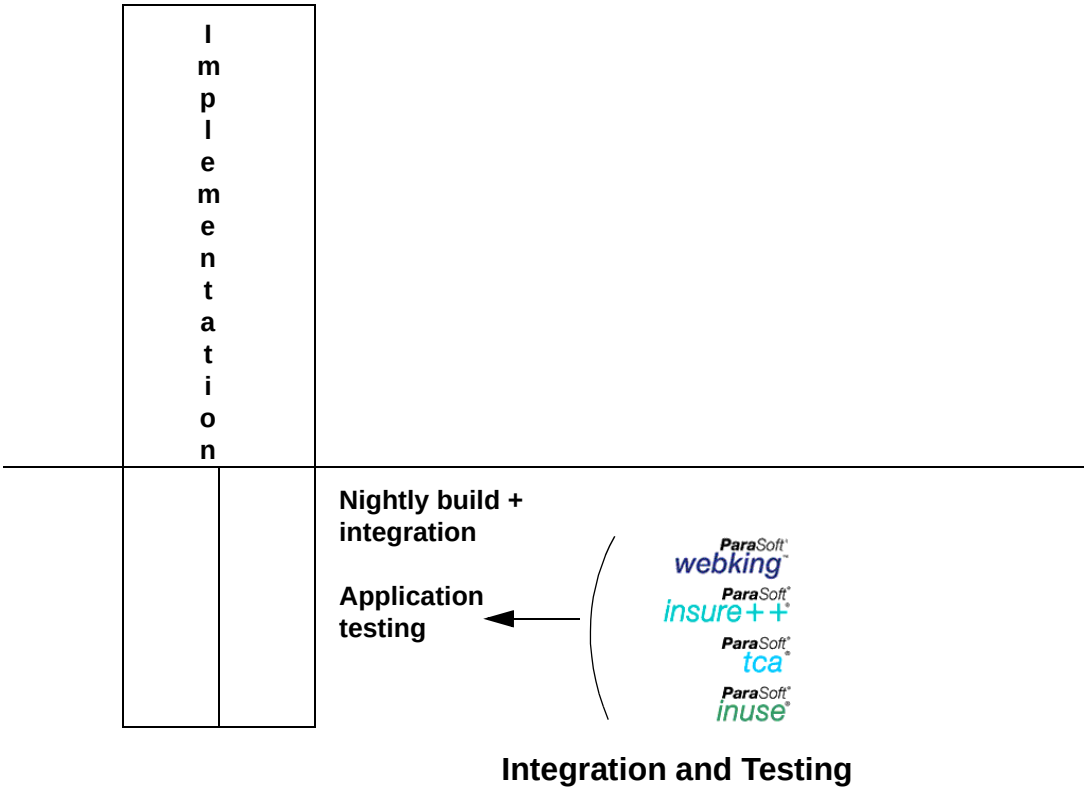


Figure 16: Integrating Tools into Integration/Testing

ParaSoft's Insure++ (an automatic runtime error-detection tool for C/C++) is the most outstanding example of tools that can automatically perform this breed of application-level testing. Insure++ automatically detects large classes of programming and runtime errors, including algorithmic anomalies, bugs, and deficiencies. Insure++ uses patented Source Code Instrumentation (patent #5,581,696), Runtime Pointer Tracking technology (patent #5,842,019), and Mutation Testing technologies to automatically locate bugs in source code. Operating in place of a compiler, Insure++ builds a database of all program elements. At runtime, Insure++ then verifies memory references and program implementation by checking data values against the database.

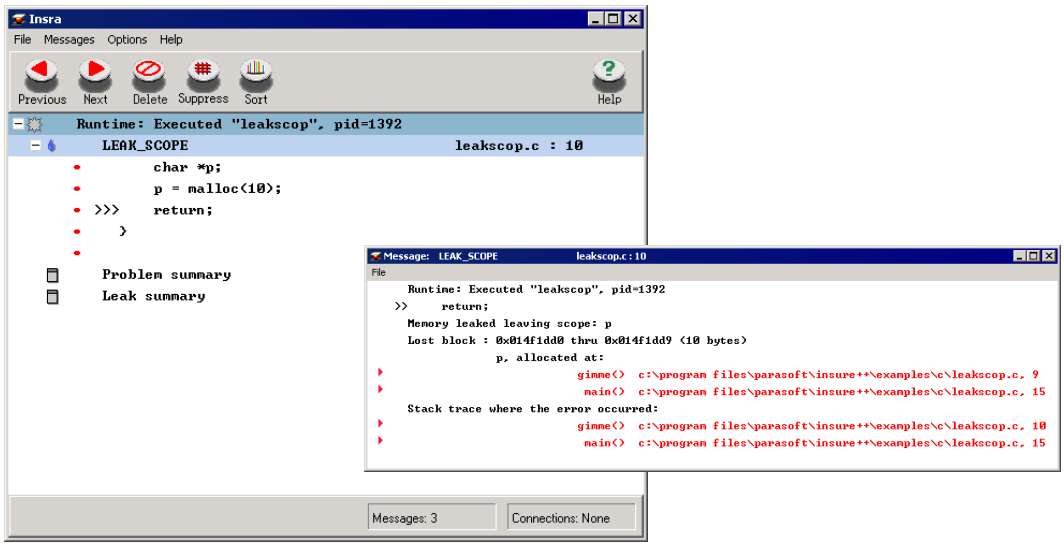


Figure 17: Detecting Application-Level Runtime Errors With Insure++

Insure++ detects the following categories of errors:

- Memory corruption/uninitialized memory
- Memory leaks
- Memory allocation errors
- Variable definition conflicts
- I/O errors
- Pointer errors
- Library errors
- Logic errors
- Algorithmic errors

Insure++ provides a complete diagnosis of each problem found, including a description of the

error, the line of source code containing the error, and stack trace information

While testing a program, Insure++ keeps track of which parts of the program are being tested. Once the program finishes, Insure++ saves coverage information for ParaSoft's TCA (an Insure++ add-on) to use. TCA shows which parts of the code were tested, how much code was tested, and how many times different code blocks were executed. This makes it easy to determine when you have reached the 70-80% coverage that you should achieve before you progress from this testing phase.

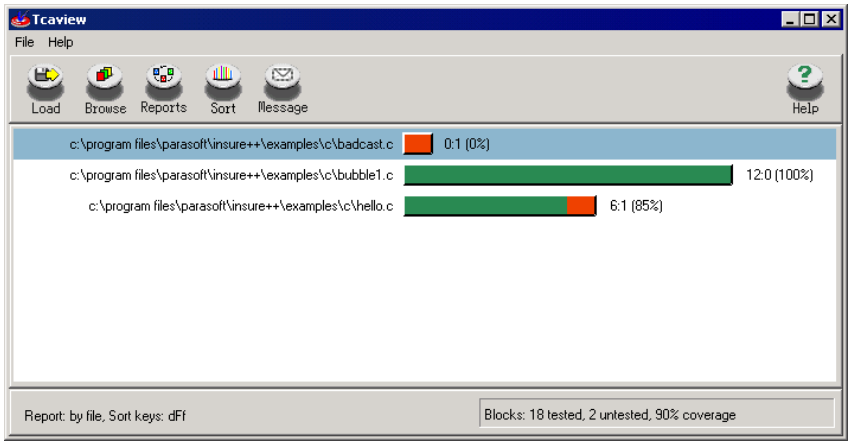


Figure 18: Gauging Coverage with TCA

For Web development, application level white-box testing is basically an extension of unit level white-box testing. It involves flushing and testing as many paths through the site as possible, then checking if each path contains construction problems bottlenecks, programs that cannot handle certain types of user traffic, servlets that throw exceptions, CGIs that core dump, databases that crash, multiple types of broken links, and a wide variety of HTML, CSS, and JavaScript problems that could affect navigation, security, presentation, and performance. You can automate this testing with WebKing. WebKing will automatically exercises the site by designing and traversing realistic user paths through the site. As it traverses the paths, it tests for problems such as bottlenecks, programs that cannot handle certain types of user traffic, servlets that throw exceptions, CGIs that core dump, databases that crash, multiple types of broken links, and a wide variety of HTML, CSS, and JavaScript problems that could affect navigation, security, presentation, and performance.

The key to performing thorough application-level white-box testing on Web sites is creating a wide variety and number of realistic paths. Because every path through a dynamic site can create different pages, each possible path may contain different problems. The only way to expose all of these problems is to exhaustively create and test paths through each site unit and across the entire application. Because each dynamic site contains so many potential different paths, it would be virtually impossible and incredibly tedious to manually map a sufficient number of realistic paths

through an average sized site to thoroughly expose path-related problems. The only practical way to create enough paths is to have a tool like WebKing create them automatically.

Application-Level Regression Testing

You should constantly create test cases as you build units and integrate units into an application. Creating a comprehensive, well-thought out test suite helps you control errors in two ways. First, simply running the test cases will detect errors. Second, this test suite can be used a gate: you should not work on other features or move to other iterations until this test suite is passed cleanly.

You can automate your application-level regression test suite with a tool that automatically runs all test cases and reports only failed test cases. Such tools are simple to build. If you have your test cases report output in text format, you can simply use `diff` to find failed test cases: if the difference between the expected value and the actual value is zero, no output is reported; if the difference is not zero, an error is reported.

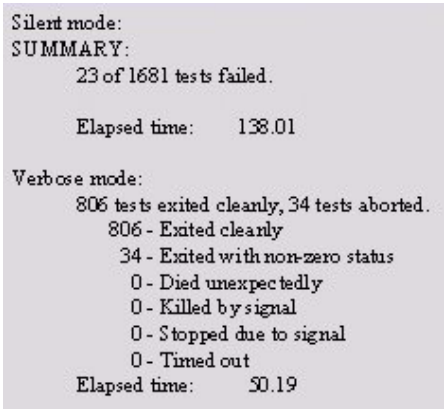


Figure 19: Output of a Regression Testing Tool

You should also continue to run your unit-level regression tests by running your unit testing tool(s) in batch mode during the nightly builds.

Completing Application Testing

In sum, every time that you integrate new code, you should run your application-level white-box and black-box tests as well as all available regression tests. As you find errors throughout this testing phase, remember to determine their cause and write coding standards that prevent them. Also, each time you discover an error that your tools or test cases did not detect:

1. Enter it in your bug-tracking system.
2. Create a test case for it.
3. Determine why this error was not detected and adjust your test suite, tool settings, and/or practices accordingly.

After you have performed all of these steps, achieved 70-80% coverage of your full-featured build, are passing most-- if not all-- of your test cases, you are ready to move on to the next iteration or project.

Conclusion

No matter what development process you use, you can improve its quality and efficiency by building the aforementioned tools and practices into it. By performing all necessary testing at the earliest stages possible, you will be able to prevent, find, and fix bugs as efficiently and effectively as possible. If you set up the prescribed gates, you will always keep your bugs at a manageable level. And if you automate as many of these development and testing processes as possible, you will find that your bug-control efforts not only improve product quality, but they also reduce development time and cost.

ParaSoft prides itself in its ability to help clients improve their development processes. For more information on how ParaSoft tools can help you and your department, talk to a Software Quality Specialist today at 1-888-305-0041, or visit http://www.parasoft.com.