Bulletproof .NET Code: A Practical Strategy for Developing Functional, Reliable, and Secure .NET Code

.NET languages are becoming increasingly popular for driving the application logic for business critical SOA and Web applications. In these contexts, functional errors are simply not acceptable, and reliability, security, and performance problems can have serious repercussions. Yet, few development teams have the resources to ensure that their code is free of implementation errors, let alone also worry about reliability, security, and performance. Whether or not your team has a satisfactory strategy for functional testing, you're taking several significant risks if you have not yet implemented a comprehensive team-wide quality management strategy:

- New code might cause the application to become unstable, produce unexpected results, or even crash when the application is used in a way that you did not anticipate (and did not test).
- New code might open the only door that an attacker needs to manipulate the system and/or access privileged information.
- The functionality that you worked so hard to design, implement, and verify might be broken when other team members add and modify code.

This paper explains a simple four-step strategy that has been proven to make .NET code more reliable, more secure, and easier to maintain—as well as less likely to experience functionality problems:

1. As you write code, comply with development rules for improving code functionality, security, performance, and maintainability.
2. Immediately after each piece of code is completed or modified, use unit-level reliability testing to verify that it's reliable and secure.
3. Immediately after each piece of code is completed or modified, use unit-level functional testing to verify that it's implemented correctly and functions properly.
4. Use regression testing to ensure that each piece of code continues to operate correctly as the code base evolves.

All four steps can be automated to promote a consistent implementation and allow your team to reap the potential benefits without disrupting your development efforts or adding overhead to your already hectic schedule. This paper focuses on explaining and elaborating on the general four-step strategy outlined above. When appropriate, this paper gives implementation examples to illustrate how Parasoft .TEST facilitates the recommended steps. It focuses on explaining the general four-step strategy, not discussing .TEST features or usage in depth. If you would like a general overview of .TEST or specific details on how to use it, visit http://www.parasoft.com.

It is important to note that the strategy and practices discussed here are designed for team-wide application. Unless they are applied consistently across an entire development team, they will not significantly improve the software that the team is building. Having a development team inconsistently apply software development standards and best practices as it implements code is like having a team of electricians wire a new building's electrical system with multiple voltages and incompatible outlets. In both cases, the team members' work will interact to form a single system. Consequently, any hazards, problems, or even quirks introduced by one "free spirit" team member who ignores the applicable guidelines and best practices can make the entire system unsafe, unreliable, or difficult to maintain and upgrade.

1. Comply with development rules for improving code functionality, security, performance, and maintainability

The first step in improving the quality, reliability, and security of your code is to comply with applicable development rules as you write it. Many developers think that complying with development rules involves just beautifying code. However, there is actually a wealth of available .NET development rules that have been proven to improve code robustness, security, performance, and maintainability. In addition, each team's experienced developers have typically developed their own (often informal) rules that codify the application-specific lessons they've learned over the course of the project.

Why is it important?

The key benefits of complying with applicable development rules are:

- *It cuts development time and cost by reducing the number of problems that need to be identified, diagnosed, and corrected later in the process.*

  Complying with meaningful development rules prevents serious functionality, security, and performance problems. Each defect that is prevented by complying with development rules means one less defect that the team needs to identify, diagnose, correct, and recheck later in the development process (when it's exponentially more time-consuming, difficult, and costly to do so). Or, if testing does not expose every defect, each prevented defect could mean one less defect that will impact the released/deployed application. On average, one defect is introduced for each ten lines of code (A. Ricadela, "The State of Software", *InformationWeek*, May 2001) and over half of a project's defects can be prevented by complying with development rules (R.G. Dromey, "Software Quality – Prevention Versus Cure", *Software Quality Institute*, April 2003). Do the math for a typical program with millions of lines of code, and it's clear that preventing errors with development rules can save a significant amount of resources. And considering that it takes only 3 to 4 defects per 1,000 lines of code to affect the application's reliability (A. Ricadela, "The State of Software", *InformationWeek*, May 2001), it's clear that ignoring the defects is not an option.

- *It makes code easier to understand, maintain, and reuse.*

  Different developers naturally write code in different styles. Code with stylistic quirks and undocumented assumptions probably makes perfect sense to the developer as he's writing it, but may confuse other developers who later modify or reuse that code—or even the same developer, when his original intentions are no longer fresh in his mind. When all team members write code in a standard manner, it's easier for each developer to read and understand code. This not only prevents the introduction of errors during modifications and reuse, but also improves developer productivity and reduces the learning curve for new team members.

What's required to do it?

a. Decide which development rules to comply with.

First, review industry-standard .NET development rules and decide which ones would prevent your project's most serious or common defects. The rules implemented by automated .NET static analysis tools offer a convenient place to start. If needed, you can supplement these rules with the ones listed in books and articles by .NET experts. As you are deciding which rules to comply with, aim for quality over quantity to ensure that the team members get the greatest benefit for

the least work. The point of having the team comply with development rules is to help the team write better code faster. Checking compliance with too many insignificant rules could defeat that purpose.

Next, consider practices and conventions that are unique to your organization, team, and project (for instance, an informal list of lessons learned from past experiences). Do your most experienced team developers have an informal list of lessons learned from past experiences? Have you encountered a specific bug that can be abstracted into a rule so that the bug never occurs in your code stream again? Are there explicit rules for formatting or naming conventions that your team is expected to comply with?

Simplifying this task

If you're using Parasoft .TEST to check whether your code complies with development rules, you can take a shortcut on this step: adopt .TEST's core set of critical rules, then add to it any critical organizational and application-specific rules that your team is currently following. The core set of critical rules includes rules that have been proven to make an immediate and significant improvement to code. By working with .NET development teams worldwide, we have learned that .NET software that complies with this core set of rules will be faster, more secure, easier to maintain, and less likely to experience functional problems.

Extending or customizing the core set of rules to suit your team's projects and priorities is simple. .TEST includes nearly 300 development rules based on .NET best practices developed by Microsoft and other .NET experts. Most rules prevent serious functionality, security, or performance problems; others make code easier to maintain and reuse. To help you determine which rules to comply with, rules are categorized by topic (for instance, security, error handling and raising, COM, CLS compliance, class member usage, and so on) as well as ranked by severity. New rules are actively being added to the product with each release. In addition to rules that examine the IL code, .TEST also provides rules that examine the C# source code; this enables .TEST to check for many code issues that cannot be identified by IL-level analysis (for example, formatting issues, empty blocks, etc.).

Teams can also enforce specific project and organizational requirements, as well as prevent application-specific errors from recurring, by creating and checking custom rules. Custom IL-level and C# rules can be created with the RuleWizard module, then checked automatically within .TEST. With RuleWizard, rules are created graphically (by creating a flow-chart-like representation of the rule) or automatically (by copying/pasting C# code that demonstrates a sample rule violation). No knowledge of the parser is required to write or modify a rule. In fact, RuleWizard is so easy to use that a beginning RuleWizard user can usually create and implement a custom rule in 15 minutes or less.

b. Configure all team tools to check the designated rules consistently.

To fully reap the potential benefits of complying with development rules, the entire development team must check the designated set of rules consistently. Consistency is required because even a slight variation in tool settings among team members could allow non-compliant code to enter the team's shared code base. If the team has carefully selected a set of meaningful development rules to comply with, just one overlooked rule violation could cause serious problems. For instance, assume a developer checks in code that does not comply with rules for closing external resources. If your application keeps temporary files open until it exits, normal testing—which can last a few minutes or run overnight—won't detect any problems. However, when the deployed application runs for a month, you can end up with enough temporary files to overflow your file system, and then your application will crash.

Simplifying this task

.TEST automatically manages the sharing and updating of standard team test settings and files to provide all team members a hassle-free way to perform tests uniformly. With .TEST, the team-wide configuration process begins with the team architect or lead developer configuring a test scenario to check the exact set of development rules that the team has decided to comply with. First, the architect enables/disables .TEST's built-in rules according to the team's designated rule list, then customizes those rules as needed. Most teams prefer to select and customize one of the preconfigured sets of rules included with .TEST rather than define a custom set of rules from scratch. The architect can also configure .TEST to limit the number of rule violations reported per test. This prevents rule compliance from becoming an overwhelming task.

The next step is ensuring that all team developers are equipped to check the same set of rules in the same way. With .TEST, the necessary sharing and updating is managed automatically through Parasoft Team Configuration Manager (TCM). The architect adds the team-standard development rule settings and files to TCM, then they are automatically distributed to all developer .TEST installations that are connected to TCM. If the architect later decides to modify, add, or remove settings and related files, TCM makes the appropriate updates on all of the team's .TEST installations. This way, developers do not need to waste time configuring settings or copying/pasting files. Further, there is no risk of developers checking compliance with an outdated or incomplete set of development rules.

c. Check new/modified code before adding it to source control.

Study after study has shown that the earlier a problem is found, the faster, easier, and cheaper it is to fix. That's why the best time to check whether code complies with development rules is as soon as it's written or updated. If you check whether each piece of code complies with the designated development rules immediately, while the code is still fresh in your mind, you can then quickly resolve any problems found and add it to source control with increased confidence.

Simplifying this task

You can use your desktop version of .TEST to test your "in progress" code and ensure that it complies with the designated team development rules before you add it to source control. In just seconds, .TEST provides the type of "constructive criticism" you would hope to receive from code reviews or pair programming—but without requiring your team members to take time away from their own work.

d. Check all new/modified code in the team's shared code base on a nightly basis.

Even if all team members intend to check and correct code before adding it to source control, code with problems might occasionally slip into the team's shared code base. To maintain the integrity of the team's shared code base, you schedule your testing tool to automatically check the team's code base at a scheduled time each night.

Simplifying this task

With .TEST, this nightly checking can be set up easily. First, configure a team ".TEST Server" installation to check the code from the command line interface. Then, use cron or the Windows Scheduled Tasks functionality to ensure that the test runs automatically each night. If any problems are detected, .TEST will email the responsible developer a report that explains the discovered problems; the developer can then import the test results into the .TEST GUI to facilitate error examination and correction. In addition, a comprehensive report will be sent to managers and be available for team review.

2. Use reliability testing to verify that each piece of code is reliable and secure

The next step toward reliable and secure code is to perform unit-level reliability testing (also known as white-box testing or construction testing). In .NET, this involves exercising each function/method as thoroughly as possible and checking for unexpected exceptions.

Why is it important?

If your unit testing only checks whether the unit functions as expected, you can't predict what could happen when untested paths are taken by well-meaning users exercising the application in unanticipated ways—or taken by attackers trying to gain control of your application or access to privileged data. It's hardly practical to try to identify and verify every possible user path and input. However, it's critical to identify the possible paths and inputs that could cause unexpected exceptions because:

- *Unexpected exceptions can cause application crashes and other serious runtime problems.*

  If unexpected exceptions surface in the field, they could cause instability, unexpected results, or crashes. In fact, Parasoft has worked with many development teams who had trouble with applications crashing for unknown reasons. Once these teams started identifying and correcting the unexpected exceptions that they previously overlooked, their applications stopped crashing.

- *Unexpected exceptions can open the door to security attacks.*

  Many developers don't realize that unexpected exceptions can also create significant security vulnerabilities. For instance, an exception in login code could allow an attacker to completely bypass the login procedure.

What's required to do it?

a. Design, implement, and execute reliability test cases.

To identify unexpected exceptions, you test each method with a large number and range of potential inputs, then check whether exceptions are thrown.

Simplifying this task

.TEST generates reliability test cases automatically. To prompt .TEST to automatically generate test cases, tell it which class or set of classes to test, then click the Start button. .TEST will then use its unique test case generation technology to generate and execute test cases to check how the code behaves in corner case conditions. This is essentially stress testing. .TEST generates inputs and captures execution results.

You can customize the tool's test case generation settings (for example, to prevent it from using NULL input values in tests when NULL values are not possible method inputs). Expected exceptions can be marked as such to result in positive tests, rather than negative tests.

b. Review and address all reported exceptions.

All exceptions exposed by the tests should be reviewed and addressed before proceeding. Each method should be able to handle any valid input without throwing an exception. If code should not throw an exception for a given input, the code should be corrected now, before you (or a team member) unwittingly introduce additional errors by adding code that builds upon or interacts with the problematic code. If the exception is expected or if the test inputs are not expected/permissible, document those requirements in the code. When other developers working with the code know exactly how the code is supposed to behave, they will be less likely to introduce errors.

c. Check all new/modified code in the team's shared code base on a nightly basis.

Same reason and procedure as step 1.d.

3. Use functional testing to verify that each piece of code is implemented correctly and operates properly

Next, extend your reliability test cases to verify each unit's functionality. The goal of unit-level functional testing is to verify that each unit is implemented according to specification before that unit is added to the team's shared code base.

Why is it important?

The key benefit of verifying functionality at the unit level is that it allows you to identify and correct functionality problems as soon as they are introduced, reducing the number of problems that need to be identified, diagnosed, and corrected later in the process. Finding and fixing a unit-level functional error immediately after coding is easier, faster, and from 10 to 100 times less costly than finding and fixing that same error later in the development process. When you perform functional testing at the unit level, you can quickly identify simple functionality problems, such as a "++" in a prefix notation substituted for a "++" in postfix, because you are verifying the unit directly. If the same problem entered the shared code base and became part of a multi-million line application, it might surface only as strange behavior during application testing. Here, finding the problem's cause would be like searching for a needle in a haystack. Even worse, the problem might not be exposed during application testing and remain in the released/deployed application.

What's required to do it?

a. Add and execute more functional test cases as needed to fully verify the specification.

After you've worked through the exceptions reported by automatically generated test, check if the code you've written actually functions as expected. Functional unit tests are meant to do just that. Without regard to internal function behavior, this means specifying function inputs and checking if the output is as expected. Such tests should be created based on the class API specification, or class use cases.

> *Simplifying this task*
>
> With .TEST, you can jumpstart your functional testing by reviewing the unverified results of .TEST's automatically-generated reliability test cases. If you choose to review these test cases, you tell .TEST whether they produced the correct outcomes and—if not—specify the correct outcomes. When the classes are retested after test case validation, .TEST will report these test cases as "failed" if and only if the given inputs do not produce the correct outcomes. The verified test cases will be treated specially by .TEST, and they will not be overwritten. They will become a permanent part of your functional regression test suite, where they are used to verify that modifications do not introduce problems.
>
> To add the realistic functional test cases needed to ensure that new or modified code implements the functionality described in the specification, you can extend the automatically-generated NUnit test classes, or develop new ones. If you know NUnit or C#, you already know how to understand and extend the automatically-generated test cases. User-defined stubs can be added to intercept calls to complex objects (such as databases) and redirect them to local resources; this allows you to test any given class in isolation. Stubs can also be used to redirect the paths covered to increase code coverage.
>
> To help you gauge the effectiveness of your test suite and determine what additional tests to add, .TEST executes and monitors coverage for all valid NUnit test cases—including the team's legacy NUnit test cases, which are fully supported by .TEST.

4. Use regression testing to ensure that each piece of code continues to operate correctly as the code base evolves

Finally, collect all of the project's existing test cases to create an automated regression test suite that verifies whether each unit continues to function as expected when the code base grows and evolves.

Why is it important?

The key benefit of performing unit-level regression testing is to ensure that code additions and changes do not break the verified functionality. In the rush to accommodate last minute requests, developers often unknowingly change or break previously-verified functionality. Moreover, previously-verified functionality is often impacted by the code modifications made during routine maintenance. In fact, studies have shown that, on average, 25 percent of software defects are introduced while developers are changing and fixing existing code during maintenance. (R.B. Grady, *Software Process Improvement*, Prentice Hall, 1997).

Why are code additions and modifications so problematic? With complex software, even a seemingly innocuous change in one part of the application can impact other functionality. However, these changes are difficult to detect without a thorough unit-level regression test suite.

Application testing might catch obvious problems that affect the application's interface, but subtle internal problems could easily go unnoticed.

What's required to do it?

a. Configure the regression test to run nightly in the background.

The regression test should check the team's shared code base by executing all applicable functional and reliability test cases. To ensure that regression testing is performed regularly and unobtrusively, schedule your unit testing tool to automatically run the complete test suite at a scheduled time each night.

> *Simplifying this task*
>
> With .TEST, automated nightly regression testing can be set up by configuring a designated team ".TEST Server" installation to run a regression test from the .TEST command line interface (dottestcli), then using cron or the Windows Scheduled Tasks functionality to run the test automatically each night. To ensure that the complete regression test suite is run, .TEST will automatically access all applicable test cases, user-defined stubs, and other test assets as long as they are added to source control along with the project.

b. Respond to test findings daily.

The purpose of regression testing is to expose all changes as soon as they are introduced, so that an appropriate response can be taken immediately. Each test case failure (a test case that does not produce the baseline outcome expected for a set of baseline input[s]) indicates a change in the code's behavior. This change may be intentional or unintentional. When code functionality changes intentionally—as a result of a feature request, specification change, etc.—test cases related to that behavior are expected to fail because the new expected outcomes will be different than the ones recorded in the baseline. However, very often, other test cases will also fail unexpectedly. If so, this reveals a complex functional problem caused by the code modifications. If no unexpected failures are identified, you know that the modifications did not break the existing functionality.

The appropriate response to a test case failure depends on whether the change was expected. If the new outcome is now the correct outcome, the expected test case outcome is updated, and it becomes a part of the baseline. If not, the code is corrected.

Simplifying this task

When the team arrives at work each morning, all testing will be completed and results will be ready for review. If any test outcomes changed, .TEST will email the responsible developer a report that explains the problems found; the developer can then import the test results into the .TEST GUI to facilitate error examination and correction. Test results can also be accessed by checking the generated HTML report with team-wide results or checking the dashboards produced by the Parasoft Group Reporting System, which collects and analyzes data from .TEST and other testing products, then organizes data into role-based dashboards tailored for the needs of managers, architects, developers or testers.

When a problem is reported, the responsible developer should review the findings and either update the test case outcome or correct the code. If the new outcome is now the correct outcome, the developer updates the expected test case outcome, and it becomes a part of the baseline. This can be done automatically, using .TEST's Quick Fix feature. If the new outcome is not the correct outcome, the developer corrects the code.

c. Update the regression test suite as needed.

If the regression test suite is not kept current, it might report annoying false positives or overlook true errors. When code functionality changes intentionally, some of the existing test cases may need to be updated to reflect new expected outcomes, reflect removed or renamed classes/methods, and so on. When new code is added, new test cases will need to be written for that code.

Simplifying this task

With .TEST, you do not need to manually update the regression test suite. .TEST automatically monitors code modifications, then updates and refactors the test suite as needed when classes/methods are renamed, moved, and so on. For instance, if new methods are added to the code, new test cases are added automatically; you do not have to tell .TEST to recreate test cases or alert it to the code modifications. If methods are removed, obsolete unverified test cases are deleted. If code is modified in a way that should not affect the test suite, .TEST understands this and does not try to recreate new test cases. Instead, it reruns the existing functional regression test suite so that it can alert you if modifications cause the class behavior to change.

Conclusion

This paper explained a four-step strategy that has been proven to make .NET code more reliable, more secure, easier to maintain, and less likely to experience functional problems. Other desirable side effects of performing this strategy consistently across a .NET development team include:

- Developers spend less time finding and fixing errors, which means less "crunch time" at the end of the project and more time for more challenging and interesting tasks, such as developing and implementing new technologies.
- The QA team no longer has to chase implementation-level errors, and has more time to dedicate to higher-level verification.
- Releases are more likely to occur on time, on budget, and with the expected functionality.

By applying this strategy with an automated technology such as Parasoft .TEST, development teams can significantly improve the quality of their software without disrupting their development efforts or adding overhead to their already hectic schedule. With .TEST, implementing the recommended strategy is fast and simple, regardless of your team's size/organization, projects, or development methodology. Essentially, there is nothing to lose, and much to gain.

To learn more about how you can use Parasoft .TEST to implement the strategies discussed in this paper, or to try .TEST for free, contact Parasoft.

.TEST can also be used to help development and QA organizations test large and complex code bases. For details on how .TEST supports this strategy, see the Parasoft white paper "Best Practices for Improving the Functionality, Reliability, Security, and Performance of a Large and Complex .NET Code Base," which is available at http://www.parasoft.com/dottest.

About Parasoft

Parasoft is the leading provider of innovative solutions for automated software test and analysis and the establishment of software error prevention practices as an integrated part of the software development lifecycle. Parasoft's product suite enables software development and IT organizations to significantly reduce costs and delivery delays, ensure application reliability and security and improve the quality of the software they develop and deploy through the practice of Automated Error Prevention (AEP). Parasoft has more than 10,000 clients worldwide including: Bank of America, Boeing, Cisco, Disney, Ericsson, IBM, Lehman Brothers, Lockheed, Lexis-Nexis, Sabre Holdings, SBC and Yahoo. Founded in 1987, Parasoft is headquartered in Monrovia, CA. For more information visit: http://www.parasoft.com.

Contacting Parasoft

USA

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: +44 (0)1923 858005
Germany: Tel: +49 7805 956 960
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com

© 2006 Parasoft Corporation

All rights reserved. Parasoft and all Parasoft products and services listed within are trademarks or registered trademarks of Parasoft Corporation. All other products, services, and companies are trademarks, registered trademarks, or servicemarks of their respective holders in the US and/or other countries.