Automating .NET Development Best Practices:
The Fast Track to Reliable and Agile .NET Software
Over the years, software industry experts have identified best practices that reliably improve developer productivity and product quality. These best practices include techniques such as unit testing, regression testing, automated code inspection, and peer code review.

However, such practices are not always feasible — especially for teams working on complex and constantly-evolving .NET applications. Such practices promise to save the team time in the long run. However, many development teams’ workload and deadlines leave no resources to spare for tasks such as:

- Reviewing hundreds of reported coding standard violations in hopes of finding a few “gems.”
- Writing test code that verifies each method’s functionality and checks its response to corner case conditions.
- Manually retesting the entire application after each change to determine if modifications broke existing functionality.
- Identifying which code needs to be reviewed, actually performing the review, then following up on suggestions.

As a result, best practices are typically shortchanged or overlooked altogether. The code gets written and minimal tests are run to confirm that it works. But it typically comes back to haunt the developers under the guise of:

- Late night debugging before the release.
- Bug reports and countless QA cycles interrupting current projects.
- The hassle of engineering emergency post-release patches.
- The inability to touch one part of the software without breaking existing functionality or introducing new defects.

Best practices that prevent such problems can indeed be practical for today’s .NET development teams. The key is to 1) unobtrusively integrate them into the team’s existing processes, and 2) support them with extensive automation that not only relieves developers from mundane tasks, but also captures their intelligence and converts it to repeatable processes.

This paper introduces the best practices shown to have the greatest impact on .NET code’s reliability, agility, and maintainability, then explains how automation via Visual Studio plugins can help your team take advantage of these best practices without disrupting or slowing your normal processes.

By following the recommended strategies, your development team can:

- Prevent important classes of bugs with minimal effort.
- Instantly expose runtime bugs that elude common testing techniques.
- Develop a more effective unit test suite in less time.
- Immediately determine if code modifications break existing functionality.
- Automate the peer review process to ensure that all new code gets reviewed and all identified issues are resolved.
- Measure and track progress during the entire software development cycle.
1. Use Static Analysis to Find Defects, Ambiguities, and Other “Gotchas” in .NET Code Without Executing It

Developers have always been fascinated with figuring out bugs, or gaining some understanding of the code’s properties without executing it. This, in fact, is a cornerstone of code reviews—using a human brain to read and understand the code, looking for defects with a fresh eye.

Peer code reviews remain the best approach for finding code defects. On average, 60% of defects can be removed via code reviews (Boehm and Basili, “Top Ten Software Defect Reduction List,” Computer, January 2001). This is hardly surprising, since code reviews use the finest analysis instrument available: the human brain.

The most efficient use of the brain, however, is to have the reviewer concentrate on tasks of higher intelligence rather than validate conformance to coding conventions. The quest to automate the latter task is what drove the development of automated static analysis tools.

When properly-implemented, automated static analysis effectively identifies the code’s soft spots and possible bugs in a relatively short time—something that can’t be achieved manually. The techniques used for static analysis include pattern matching, data and control flow analysis, statistical analysis, and many analysis techniques based on heuristics.

Pattern Matching Static Analysis

The pattern matching approach helps teams check whether code adheres to a set of coding practices or a coding policy. For .NET, many such “best practices” are defined in Microsoft’s .NET Framework Design Guidelines,” as well as books such as Effective C# and .NET Gotchas. In addition to supporting common .NET guidelines, static analysis tools can enforce custom rules that suit your specific application and implementation context, as well as help you identify code patterns indicative of application-specific defects that you previously uncovered.

Even though static analysis is simple, efficient, and easy to use, it is nevertheless the most effective way to prevent large classes of common coding errors. It also results in code that is more readable and easier to maintain.

Data and Control Flow Analysis

Static analysis based on data flow can be very sophisticated in implementation and algorithms. Data flow analysis is based on the idea that program code can be compiled into a graph-like representation, which can be simulated to determine whether certain execution paths can, in principle, lead to structural bugs such as NullReferenceExceptions, resource leaks, SQL injections, and other security vulnerabilities. This bug detection technique is very powerful because it does not depend on user input to identify bugs that, in reality, are data-dependent.

Integrating Static Analysis into the Existing Process

Effective application of static analysis goes well beyond acquiring the appropriate tools; it also requires the judicial application and monitoring of an appropriate process. To start, the team architect or other designated leader defines a coding policy that the team members understand and respect, and then configures the tool to automatically check that coding policy. The next step is typically to scan your existing .NET code base and resolve all reported violations. This
removes a significant number of hidden defects in the code you will be building upon, improving the stability of your foundation.

After cleaning the violations in the existing .NET code base, establish a process for building upon that code base. Ideally, this involves implementing two levels of automated static analysis. The first level occurs at the developer desktop: as developers complete new or modified code, they perform static analysis from their IDE, and then resolve all reported violations before committing the related code to source control. The second level is performed from a team server machine, where batch-mode analysis automatically checks the entire code base each night, and violation counts are tracked and correlated with other application quality metrics.

As additional bugs are reported (by testers, customers, etc.), analyze their root causes, try to identify a set of rules that will prevent the same bugs from reoccurring, then configure the static analysis tool to check these rules.

When planning the static analysis implementation, it's important to consider that the rules being enforced often have exceptions. Although a given rule might apply in most contexts, there could be situations where it is reasonable (and necessary) to break the rule. To be successful in enforcing the rules, you need an effective way to handle the exceptions.

**Automating Static Analysis with Parasoft .TEST**

Using Parasoft® .TEST™, development teams can automate the static analysis techniques mentioned above to prevent many bugs as well as detect bugs that are conventionally found by running the application. Parasoft .TEST is a complete .NET developer’s toolkit for code analysis, code review, automated unit testing, coverage analysis, and regression testing — on the desktop as a Microsoft Visual Studio plugin and in batch processes.

**Enforcing Coding Patterns with Parasoft .TEST**

To automate pattern matching static analysis, teams can have .TEST check compliance with the team’s designated coding policy. To ensure that this analysis suits the team’s policies and priorities, .TEST allows users to define their own rule sets with built-in and custom rules. Different sets can be used for new code and existing code. .TEST comes preconfigured with hundreds of built-in rules—including guidelines from Microsoft’s .NET Framework Design Guidelines, from Effective C# and .NET Gotchas books, and the experiences of software developers in many companies. These rules have been proven to help identify potential bugs from improper language usage, enforce coding best practices, and improve code maintainability and reusability.

In addition to checking rules that examine the IL code, .TEST also allows teams to check rules that examine the C# source code. This enables automated checking of design patterns and identification of bugs that are not apparent at the IL level (due to compiler transformations).

For example, the following graphic shows how .TEST’s pattern matching static analysis identified a situation where the developer intended to use logical `and`, but instead used `bit and`. 
Since bit and is not short circuited, this results in the evaluation of ssn, even when ssn is null and causes an exception to be thrown. Although the exception is obvious in a simple case, the exception could easily escape testing in more complex cases.

To ensure that the process is as effective and streamlined as possible, custom IL-level and C# rules, which are created with a graphical RuleWizard editor, can also enforce specific project and organizational requirements and prevent the recurrence of application-specific defects after a single instance has been found. Rule names and severity categories can easily be mapped to match your team’s internal coding policies and priorities. Moreover, case-specific suppressions provide a systematic way to follow rules in general, but still make some exceptions that are deemed acceptable by you or your team.

This type of static analysis virtually eliminates the need for line-by-line inspections during peer code reviews. Reviews can then focus on examining algorithms, reviewing design, and searching for subtle errors that automated tools cannot detect. Although unit testing can identify issues with your code, automated code analysis is the only way to prevent entire classes of errors from occurring in your code. The ability to customize the rules, along with the ability to handle exceptions to rules, makes .TEST a compelling tool for .NET development.

**Flow Analysis and Other Techniques with Parasoft .TEST**

For dataflow static analysis, Parasoft’s BugDetective uses several analysis techniques, including simulation of application execution paths, to identify paths that could trigger runtime defects. Defects detected include using null reference exceptions, division by zero, and resource leaks.

For example, the following graphic shows three issues that BugDetective found in a sample banking application:
Taking a closer look at the Avoid NullReferenceException error, note that BugDetective shows the complete path that leads to the issue it is pointing out. In this case, cust was set to null on line 64. On line 68, there is a call to LookupCustomerName. An exception was thrown from within that method on line 48. This is indicated by the red ball. The control then goes to the catch block where a null reference occurs on line 74. This problem was likely to have escaped normal testing.

Note how the analysis has crossed method boundaries. With the ability to analyze execution paths that are taken even when exceptions are thrown and paths that cross method boundaries, BugDetective can point out many issues that are easily missed with code review or testing.
2. Develop Test Cases then Run Them Regularly for Regression Testing

Many developers recognize the value of writing test cases to verify whether new or modified code works correctly before checking it in. A less common, but equally critical, task is to develop regression test cases that capture the current behavior of the existing code base before modifying or extending it. These test cases are not designed to validate functionality or expose bugs, but rather to help developers determine if their modifications have unexpected impacts on the rest of the code base.

Regression issues are inevitable: on average, 25% of software defects are introduced while programmers are changing and fixing existing code during maintenance. (R.B. Grady, *Software Process Improvement*, Prentice Hall, 1997). However, you can minimize their impact by identifying them as soon as they are introduced, and resolving them immediately.

We begin this section with an explanation of how to develop tests for existing code base and then move on to developing tests for new or evolving code. This is followed by a demonstration of how automation helps in this process. We end the section with a brief description of how to get the maximum value out of the tests by running them as regression tests on a nightly basis. This promotes early identification of issues that result from code modifications.

Developing Test Cases for Existing Code

Many teams find themselves tasked with extending or modifying a code base that lacks sufficient tests. It is risky to proceed without any safety net to verify whether changes break something in other areas of the code. In such cases, try to resist the temptation to “hack at it anyway,” and first create some regression tests before making any code changes. Otherwise, attempting to maintain the integrity of the previous functionality while introducing new features becomes a high-stakes crap shoot.

There are two complementary approaches to creating such a test suite: one manual, the other largely automated. A manual approach involves writing some tests at the highest abstraction level of the code. Since writing tests by hand is generally expensive, it’s best to minimize that effort upfront while still capturing the reference behavior. This can be accomplished by identifying the module-under-test’s high-level APIs, and exercising them first. The tests should follow the module’s “positive” behavior, and produce code coverage for expected conditions. This is beneficial because a few tests aimed at the high-level API will exercise many of the module’s lower-level functions. The resulting test coverage can be amended with lower-level tests as required to increase the test coverage to a comfortable level before you change the code.

Another way to approach this problem is by creating tests from the bottom up, starting from the leaf-level functions. Such tests typically exercise more paths through the code. However, the bottom up approach is much more labor-intensive because many more tests need to be created. To get results in a reasonable time, you need an tool capable of automatically generating API tests (either with semi-random inputs or using specified value ranges) and capturing their results. Some commercial testing tools offer this capability; for public domain software, check tools such as NUnit (available at http://sourceforge.net/projects/nunit).

Developing Test Cases for New/Modified Code

When code is being added or modified, it is the prime time to apply Test Driven Development (TDD) or any other technique that keeps test cases at the front of— or at least tightly linked to—
the development process. Depending on the nature of the code, functional-level (application-level) regression tests and/or unit-level regression tests may be appropriate. Each has its own benefits.

The functional-level tests are preferred in cases when tests can feasibly be set up to represent a high-level requirement to implement—or a specific bug that should be addressed—and the tested program can produce sufficient output to effectively validate the code's behavior (via the direct outputs it generates, log messages, debug messages, etc.).

Unit tests are appropriate when test setup is complicated because the error condition itself is not a common scenario, when the changed code is buried deep in the module hierarchy, or when it’s difficult to validate the test outcome at the functional level (e.g., the program lacks debugging facilities). This is frequently the case with corner-case conditions or error-handling functions. These are supposed to trigger in exceptional circumstances, which would be difficult to set up in a test harness.

No matter what type of tests you create for a new code fragment, they must be saved so they can establish the regression test suite discussed in the next practice.

**Developing Test Cases with Parasoft .TEST**

Using .TEST, your development team can create test suites very quickly. The generated test cases are in NUnit format, making it easy for you to review the tests and modify them if necessary. In addition to exposing potential reliability problems, these test cases capture the code’s current behavior, establishing a regression baseline that becomes an invaluable resource when trying to understand how code modifications impact the rest of the code base.

.TEST's automated test case generation allows you to create a more effective test suite in less time. With .TEST, you do not need to write any code to generate a foundational set of test cases that exercise each function, and you can easily add more test cases using C# and the industry-standard NUnit framework. You typically do not need to worry about writing test cases for simple methods, and can focus your resources on extending/adding tests for the more complex methods.

**Generating Tests Automatically**

The test generation process is best explained with an example. To provide control to the user, test generation is performed in two steps.

In the first step, we simply select the project in the Visual Studio Solution Explorer and run a built-in (preconfigured) Test Configuration that generates tests with calls to API that are used in the second step to generate assertions for the test. This creates a new project that includes a complete test class for testing each file under test.

Each of the generated tests uses the same format. First, there are some custom attributes that provide context information about the test (describing that it was automatically generated, which method it tests, and so on). These attributes are followed by code that creates an object to be used for testing and then calls methods in this object. Finally, there is a “Record State” region, which contains calls to methods that can record the state of the variables and objects when the
tests are executed. You can influence what values will be recorded in the next phase by modifying the code that records the current state.

In the second step, .TEST uses the recording API mentioned above to actually record the current state and generate assertions based on the result. To do this, we run the “Run Tests and Update Assertions” Test Configuration.
At this point, each test case captures the current state of the code base, and is perfectly suited for regression testing, which is discussed in the subsequent practice. Note how the single statement for recording the state of `var0` got expanded to many assertions.

The tests are now complete and can be added to the regression suite for regularly-scheduled runs. If the code later changes so that the assertions are no longer valid, you have the choice of modifying them in place or regenerating them (as we did the first time) and reviewing them.

**Using Stubs**

Stubs, which allow all calls to one method to be directed to another method, are an important part of unit testing. Stubs are most commonly used for testing classes in isolation. This addresses one of the greatest challenges in writing unit tests: getting complex objects in different states. For instance, calls to a database API can be redirected to return values that you designate. Stubs are also helpful for supporting unattended testing. For example, by stubbing out methods that open dialog boxes, you can disable dialog boxes from popping up during your nightly runs.

.TEST ships with a set of commonly-used stubs (for example, stubs for functions that perform file operations). It can also automatically generate stubs for the methods in an assembly. Like the automatically-generated test cases, the automatically-generated stubs can be customized and extended as needed. For an example of how stubs are implemented in .TEST, consider the following example, which stubs out the `show` method to always return `OK`.

```csharp
#region Record State
vr.Int32Value(inputArg0, "inputArg0");
vr.Int32Value(inputArg1, "inputArg1");
vr.StringValue(inputArg2, "inputArg2");
vr.ObjectValue(var0, "Bank.CreditCard", "var0");
vr.FinishRecording();
#endregion

#region Assertions
Assert.AreEqual(0, inputArg0);
Assert.AreEqual(0, inputArg1);
Assert.AreEqual("a", inputArg2);
#endregion

#region Assertions for (var0)
Assert.AreEqual(0, var0.CurrentBalance);
Assert.AreEqual("0000-0000-0000-0000", var0.CreditCardNumberString);
Assert.AreEqual(0, var0.CurrentBalance);
Assert.AreEqual("0000", var0.ZipCodeString);
Assert.AreEqual(0, var0.CustomerId);
Assert.AreEqual("a", var0.CustomerName);
#endregion
```
In addition to providing support for using method-specific stubs while executing unit tests, .TEST also allows you to statically instrument the assemblies to use method-specific stubs. This feature provides tremendous flexibility when running tests. Some example applications of this feature are:

- You can use method-specific stubs to log information about the parameters of some interesting methods while running unit or integration tests.
- By stubbing some calls (such as those involving DateTime properties etc.) you can make the tests behave in a more predictable manner.
- By using different stubs, the same tests can be used in different situations. For example, database calls can be stubbed out in one situation and routed to an actual database in another situation.
- Method-specific stubs can be written in such a way that they act as mock objects. It is important to note that this can be achieved without introducing mock objects into the tests. The same tests can be run either using the mock objects or real objects. It is also possible to have different kinds of mock objects.

Regression Testing with Parasoft .TEST

The test cases that your team automatically generates using .TEST, along with any other NUnit test cases you have developed, establish a robust regression test suite that detects if incremental code changes break existing functionality. These tests are configured to run automatically on a regularly scheduled basis— preferably nightly. .TEST, running in batch mode, scans the modified project code base (accessed from the source control system), automatically executes the existing
regression test suite, and also automatically generates regression tests for new code. It then compares the current results with those from the originally captured "golden set."

For instant feedback on whether their code changes broke the existing functionality, each developer can import (into Visual Studio) the regression failures caused by their modifications.

The results from this automated test execution not only tell the team which test case assertions failed as a result of the previous day's code modifications, but also indicate exactly which developer modified the code that broke each assertion. Since the regression failures are directed to the developers responsible for them, the overall process of fixing them is much more streamlined than it would be if all developers were looking at the same list of regression failures.

3. Peer Review all New/Modified Code

Once new or modified code is verified and checked in, it should be finalized by a team code review. It does not matter how the code review is conducted. It can be integrated into the code authoring process (as in pair programming), a distributed code review with automatic notification based on source control commits, or the old-fashioned gathering of people into the room with printouts of code. The key is to have the code reviewed by a human brain.

Code Review with Parasoft .TEST

To streamline the part of the code review process that cannot be replaced by automated code analysis, your team can use .TEST’s Code Review module to automate review preparation, notification, and tracking. .TEST automatically identifies updated code by scanning the source
control system, matches the code with designated reviewers, and then tracks the progress of each review item until closure. With the Code Review module, teams can establish a bulletproof review process—where all new code gets reviewed and all identified issues are resolved.

When Code Review is implemented via .TEST, a source scanner scans the code base on a nightly basis and creates review tasks based on the check ins that were committed on that day. The developers who need to perform the review receive email notifications, and then review the specific task details in Visual Studio. Check ins are grouped based on their timestamps and comments. A sample display of tasks is shown in the following graphic:

Double-clicking on a file reported in this list opens the referenced file in a Diff viewer, which highlights the changes that need to be reviewed.
After reviewing the changes, you can create a thread for writing comments. Once you have finished reviewing the code and adding comments, you can either accept or reject the change. If the change is rejected, the original author will get a task for following up this thread. If it is accepted, its status will be reported as “approved.”

The following graphic shows a change that is being accepted.

Automating the code review process in this manner makes code reviews more productive and practical for software development organizations, and is especially valuable for distributed teams.
4. Ensure Consistency and Monitor for Continuous Improvement

An easy-to-use workflow and a good deployment strategy are both essential for making the recommended practices a sustainable part of real world development processes. Consistency is also critical. 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.

Although each of the techniques introduced above is a good practice on its own, combining them in an automated process framework helps teams to effectively address their productivity and code quality goals, and to confidently manage development risks. Further gains can be realized by monitoring and continually improving the process.

.TEST Workflow for Team Development

.TEST’s support for team deployment standardizes testing team-wide and provides a sustainable workflow for integrating best practices into the team's existing processes—with minimal disruption. The architect defines the team's designated test configurations and static analysis rules, and then Parasoft Team Configuration Manager (TCM) automatically shares them across all team .TEST installations. Developers can test code directly from Visual Studio to find and fix problems before adding it to source control. Additionally, .TEST Server can test the entire project code base each evening, then email reports to the architect and responsible developers if any problems are detected. Developers can then import results into Visual Studio to review and repair the errors reported for code they authored.

Monitoring with Parasoft GRS

Your team can use Parasoft GRS to find problem areas in your code and processes, obtain the details that you need to pinpoint their root causes, and then monitor improvements. GRS gathers data from:

- Your development infrastructure (source control, bug tracking, build system, requirements management system, etc.)
- Parasoft .TEST
- Complementary products (such as Parasoft SOAtest, an automated test and analysis tool for ensuring secure, reliable, and compliant Web services).

It correlates data from these sources to provide comprehensive and objective insight into application quality, team productivity, and project risk factors.
For example, analyzing GRS graphs can help development teams answer questions such as:

- Am I on track to release on schedule?
- Why do I have so many bugs?
- Why do I miss deadlines?
- How can I quantify increases in quality and productivity?
- How can I reduce the cost of maintaining and updating the software?
- How can I reduce the number and length of QA cycles?
- How can I maximize available resources?
To learn more about how Parasoft .TEST and other Parasoft solutions can put your team on the fast track to more reliable and agile code—or to request an evaluation—contact Parasoft as described below.

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 89 4613323-0
Email: info-europe@parasoft.com

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

Other Locations
See http://www.parasoft.com/jsp/pr/contacts.jsp?itemId=268

© 2007 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.