The Technical Support Guide to Development Best Practices

By Dr. Adam Kolawa

TECHNICAL SUPPORT PROFESSIONALS—ESPECIALLY THOSE RESPONSIBLE FOR software testing—can benefit from understanding both development and testing best practices. When you understand development best practices, you can identify when developers are not following the required practices and notify the person in charge of enforcing development best practices (technical support professionals should not be responsible for this enforcement). When developers follow the appropriate best practices, you will waste less time chasing bugs during testing, or, when bugs make it into the release, supporting customers who are impacted by the bugs. Moreover, technical support professionals that are tasked with software testing should have an understanding of testing best practices in order to test the software as thoroughly and efficiently as possible.

This article introduces the development and testing best practices that technical support professionals should be aware of.

DEVELOPMENT BEST PRACTICES

Practice #1: Defensive Programming

Defensive programming is the practice of anticipating where failures can occur and then creating an infrastructure that tests for errors, reports when anticipated failures occur, and performs specified damage-control actions—such as stopping program execution, redirecting users to a backup server, enabling debugging information that can be used to diagnose the problem, and so on. These defensive programming infrastructures are typically built by adding assertions to the code, implementing Design by Contract, developing software defensive firewalls, or simply adding code that validates user inputs. By applying defensive programming techniques, developers can detect problems that might otherwise go unnoticed, prevent minor problems from growing into disasters, and save themselves a lot of debugging and maintenance time in the long run.

Hint: If the developers are performing defensive programming, they will be able to compile the code in two ways—with or without the defensive programming infrastructure.

Defensive Programming Resources

- Hunt, Andrew and David Thomas, *The Pragmatic Programmer*, 1999.
- Eldridge, Geoff. Java and Design by Contract. www.elj.com/eiffel/feature/dbc/java/ge/.
- Kolawa, Adam, Wendell Hicken, and Cynthia Dunlop, *Bulletproofing Web Applications*. 2001.
- Maguire, Steve, *Writing Solid Code*, 1993.
- McConnell, Steve. *Code Complete*, 1993.
- Meyer, Bertrand. *Object-Oriented Software Construction*, 2000.
- Payne, Jeffrey E., Michael A. Schatz, and Matthew N. Schmid. Implementing Assertions for Java. *Dr. Dobb's Journal* (January 1998). www.ddj.com/articles/1998/9801/9801d/9801d.htm.
- Plessel, Todd. Design by Contract: A Missing Link in the Quest for Quality Software. www.elj.com/eiffel/dbc/.

Practice #2: Code Review

A code review is the process where the developers and architects meet and discuss code. Its purpose is to exchange ideas about how code is written, and to establish a consistent interpretation of code throughout the group. During these reviews, developers should be given the opportunity to explain their code to one another. Often, simply explaining the code helps developers identify problems and envision new solutions for previously troubling dilemmas. When the group members discuss the code, their discussion should focus on important issues such as algorithms, object-oriented programming, and class design.

Hint: If developers are not complaining about code reviews, they probably are not performing them.

Code Review Resources

- Hunt, Andrew and David Thomas, ***The Pragmatic Programmer***, 1999.
- Kaner, Cem, *Testing Computer Software*, 1988.
- McConnell, Steve. *Code Complete*, 1993.
- Meyers, Glenford J., *The Art of Software Testing,* 1979.

Practice #3: Coding Standard Compliance

Coding standards are language-specific programming rules that greatly reduce the probability of introducing errors into applications. Coding standards originated from the intensive study of industry experts who analyzed how bugs were generated when code was written and correlated these bugs to specific coding practices; they took these correlations between bugs and coding practices and came up with a set of rules that prevent coding errors from occurring. In a team environment or group collaboration, coding standards ensure uniform coding practices, reducing oversight errors and the time spent in code reviews. When work is outsourced to a third-party contractor, having a set of coding standards in place ensures that the code produced by the contractor meets all quality guidelines mandated by the client company.

Hint: If there is no system set up to scan code on a regular basis, the developers probably are not following coding standards.

Coding Standard Resources

- Kernighan, Brian and P.J. Plauger. The Elements of Programming Style, 1988.
- Kolawa, Adam, Wendell Hicken, and Cynthia Dunlop, *Bulletproofing Web Applications*. 2001.
- McConnell, Steve. *Code Complete*, 1993.

Practice #4: Unit Testing

Unit testing involves testing software code starting with its smallest functional point, which is typically a single class. Each individual class should be tested in isolation before it is tested with other units or as part of a module or application. While it begins with testing software at its smallest functional point—typically a single class—it also spans through units and sub-modules, on to modules and applications, testing functionality of the appropriate pieces at each stage. By testing every unit individually, and then together, most of the errors that might be introduced into the code over the course of a project can be detected or prevented entirely. The objective of unit testing is to test not only the functionality of the code, but also to ensure that the code is structurally sound and robust, and to be able to respond appropriately in all conditions. If code in these systems is not tested appropriately, its vulnerabilities can be used to break into the code and lead to a security risk (a memory leak or stolen pointer, for example) as well as performance issues.

Unit testing involves testing software code starting with its smallest functional point, which is typically a single class. Each individual class should be tested in isolation before it is tested with other units or as part of a module or application.

Hint: If the developers do not regularly run a substantial number of test cases that contain harnesses, stubs, and calls to main(), they are not performing unit testing.

Unit Testing Resources

- Beck, Kent. *Extreme Programming Explained: Embrace Change*, 1999.
- Hunt, Andrew and David Thomas, *The Pragmatic Programmer*, 1999.
- Kaner, Cem, *Testing Computer Software*, 1988.
- Kolawa, Adam, Wendell Hicken, and Cynthia Dunlop, *Bulletproofing Web Applications*. 2001.
- McConnell, Steve. *Code Complete*, 1993.
- Meyers, Glenford J., *The Art of Software Testing*, 1979.

Practice #5: Construction Testing (White-Box Testing)

Construction testing, also known as white-box testing, is used to confirm that software code is structurally sound, and can process a wide variety of inputs without failure. It is a type of unit testing that begins with testing software at its smallest functional point—typically a single class—and spans through units and sub-modules, on to modules and applications, testing the structure of the appropriate pieces at each stage. It is much like posing a series of "what if?" questions that determine whether the application continues to behave appropriately under unusual or exceptional conditions, and verifies that any inputs thrown at the code will be received and addressed with the proper behavioral response. Construction testing ensures that the application is built correctly and can help prevent problems such as application failure and security breaches.

Hint: If you see test cases that don't correlate to specification entries, the developers are probably performing white-box testing.

Construction Testing Resources

- Beizer, Boris, *Software Testing Techniques*, 1990.
- Cole, Oliver. White-Box Testing. *Dr. Dobb's Journal* (March 2000).
- Kaner, Cem, *Testing Computer Software*, 1988.
- Kolawa, Adam, Wendell Hicken, and Cynthia Dunlop, *Bulletproofing Web Applications*. 2001.
- McConnell, Steve. *Code Complete*, 1993.
- Meyers, Glenford J., *The Art of Software Testing*, 1979.

Practice #6: Functional Testing (Black-Box Testing)

Functional testing, also known as black-box testing, is used to verify that software conforms to its specification and that all of the intended functionality is included and working correctly. It begins with testing software at its smallest functional point—typically a single class—and spans through units and sub-modules, on to modules and applications, testing functionality of the appropriate pieces at each stage. It is used to validate that software works correctly when users perform a series of actions within the software's specification. The intent is to confirm behavior that is expected from the smallest possible unit of code to the entire application and ecause it tests each component in isolation and as part of the system, it allows developers to frame or isolate the functionality of each piece and isolate any potential errors that could affect system functionality.

Hint: If you see test cases that correlate to specification entries, the developers are performing black-box testing.

Functional Testing Resources

- Beck, Kent. *Extreme Programming Explained: Embrace Change*, 1999.
- Beizer, Boris, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, 1995.
- Beizer, Boris, *Software Testing Techniques*, 1990.
- Kaner, Cem, *Testing Computer Software*, 1988.
- Kolawa, Adam, Wendell Hicken, and Cynthia Dunlop, *Bulletproofing Web Applications*. 2001.
- McConnell, Steve. *Code Complete*, 1993.
- Meyers, Glenford J., The Art of Software Testing, 1979.

Practice #7: Coverage Analysis

Coverage refers to how much of the system the test cases cover. Coverage is typically measured either as line coverage, branch coverage, or path coverage. With line coverage, a line of code is deemed

covered if the test case has touched any part of it. For example, if you have an if condition and one path through it is executed, the affected lines are deemed covered even when all conditions were not exercised. Branch coverage yields more precise coverage details than line coverage, but less precise details than path coverage. With branch coverage, a piece of code is considered 100% covered when each branch of an if statement has been executed at least once. Path coverage accounts for whether the different paths of a statement are exercised. With path coverage, an if statement with multiple branches is not deemed fully covered unless all of its different possible paths are executed. Coverage data can be used to monitor how well the current test suite is covering the code, and to pinpoint which additional code lines, branches, and paths need to be exercised in order to achieve the more coverage.

Hint: If the developers' test reports do not show coverage information, the developers probably are not measuring the coverage of their tests.

Resources

- Kaner, Cem, *Testing Computer Software*, 1988.
- Meyers, Glenford J., The Art of Software Testing, 1979.

Practice #8: Regression Testing

Regression testing involves testing modified code under the same set of inputs and test parameters used in previous test runs to ensure that modifications have successfully eliminated errors and have not introduced new errors. It is important to use regression testing as a regular part of any testing process to find new errors during development (when they are easiest to fix) and before new lines of code are added that depend on the current code base. Ideally, regression testing is performed nightly (during automated nightly builds) to ensure that errors are detected and fixed as soon as possible.

Hint: If the developers do not have a system that automatically executes all available tests every day, they are not performing regression testing.

Resources

- Beck, Kent. *Extreme Programming Explained: Embrace Change*, 1999.
- Hunt, Andrew and David Thomas, ***The Pragmatic Programmer***, 1999.
- Kaner, Cem, *Testing Computer Software*, 1988.
- Kolawa, Adam, Wendell Hicken, and Cynthia Dunlop, *Bulletproofing Web Applications*. 2001.
- McConnell, Steve. *Code Complete*, 1993.

Practice #9: Integration Testing

Integration testing is used to verify that the multiple units of code work correctly once combined. It is also called interface testing because the actual test process verifies that the interfaces between the units are compatible. It can be used to test at a high level—for example testing complete transactions in a Web service also tests the integration between individual components (larger groups of units); or at a more granular level—for example, one could systematically test a larger number of units by combining two and testing them, adding a third and testing, and so on.

Hint: If the developers do not have complex test cases that verify different parts of the system, they probably are not performing integration testing.

Resources

- Hunt, Andrew and David Thomas, *The Pragmatic Programmer*, 1999.
- Meyers, Glenford J., The Art of Software Testing, 1979.

TESTING BEST PRACTICES

In addition to checking whether developers follow best practices, testers can benefit by adopting testing best practices. These practices are designed to help you verify that the product really works and is solid.

Practice #1: Understand the Product Architecture Before you Start Testing the Product

If you do not understand the architecture and inner workings of the product you are testing, you will not be able to anticipate where it is most error prone. As a result, you could overlook easy opportunities to uncover a large amount of errors in a small amount of time.

In addition to checking whether developers follow best practices, testers can benefit by adopting testing best practices. These practices are designed to help you verify that the product really works and is solid.

As you gain experience testing over the years, you will learn that there are some parts of programs that are more error prone than others. Generally, the most error prone parts of a program are the interfaces between different modules. Why? Because different groups of developers work on different modules. These groups often misunderstand one another's intentions and assumptions, and errors typically result when their code interacts. Consequently, a lot of bugs are usually hidden in program interfaces.

Another trick is to learn which development groups worked on which program segments, and use those development groups' track records to anticipate which parts of the program are most error prone. For example, if you know that Group C worked on a certain part of the program and that Group C usually produces code with a lot of errors, you might want to focus a large percentage of your testing efforts on the part of the program created by Group C. Likewise, if you know that Group A almost always delivers clean, functional code, it is probably safe to spend a smaller percentage of your time testing the parts of the program that Group A worked on.

Just like a police detective needs to understand the entire situation before he can solve a murder mystery, you need to understand the entire situation to solve the mystery of "does this software really work?"

Practice #2: Anticipate Potential Misuses and Verify How the Software Responds in Those Cases

Don't think that your job is done once you have verified that the software does what it is supposed to do. Users inevitably try to use software in unexpected ways—sometimes because they see additional usages for the product, sometimes because they misunderstand how to use the product, and sometimes because they want to launch security attacks through the product.

The specification is the best starting point for testing unexpected usages. If you don't have a specification, find one or write one yourself

if needed. For each feature in the specification, try to imagine what unexpected paths could be taken by a new user exploring the program, an experienced user trying to maximize the program, and a hacker trying to manipulate the program. For example, what happens if the user tries to apply a tool to an unexpected type of source? If the user does not provide critical information? If the user designs and sends unexpected inputs in an attempt to gain access to privileged data or to gain control of a program?

The appropriate response to these unexpected situations depends on the program and the situation. In all cases, the response should be intelligent. For example, if the user does not provide critical information, it is better to have the program display a helpful dialog explaining the problem than to simply fail to perform the requested action.

Practice #3: Clearly Record the Procedure to Reproduce Each Error Found

For each problem that you detect, be sure to record detailed, unambiguous instructions for reproducing that problem, as well as a detailed description of the environment and context in which the problem occurred. If your bug report documentation is incomplete or confusing, two problems could occur.

One problem is that developers might not be able to reproduce the error and thus probably will not be able to fix the error. If the developer does manage to reproduce the error without proper instructions, he or she will have probably wasted a lot of time in the process.

Another problem is that you will not be able to effectively verify whether the developer's modification corrected the problem. If you don't test the repair using the exact same environment and procedures that produced the error in the first place, a passed test will not necessarily mean that the error was corrected.

Practice #4: Help the Team Prevent Errors

Typically, when testers find errors, they add a report to the bug tracking system, the responsible developer tries to reproduce and repair the problem, then the tester must verify that the modification corrected the reported problem and did not introduce any new problems. This approach is not only time-consuming and costly, but also inefficient because it doesn't help prevent the same types of errors from recurring. Moreover, it causes the team to waste a significant amount of time, effort, and resources on the same types of errors thousands of times over.

When your entire team adopts a concept called Automated Error Prevention, you can ensure that any time an error is discovered, your process is improved, and that error—along with entire classes of similar errors—are prevented from recurring.

Parasoft AEP methodology defines the practices that apply the concept of AEP to the software lifecycle, and describes how those practices can be used in a group environment. The concept of AEP advocates the automation of five specific procedures, which are combined to improve the development process and prevent software errors:

1. Detect an error
2. Isolate the cause of the error
3. Locate the point in the process that created the error
4. Implement practices to prevent the error from reoccurring
5. Monitor for improvements

The key to AEP is that developers or testers should find each type of error only once. The knowledge the team gains from finding and analyzing errors should be used to improve the process so that you never encounter repeat occurrences of errors similar to those you have already found.

This process benefits the entire team by improving quality and reducing costs, but it is especially beneficial to testers because if developers are performing the required error prevention practices, you won't need to repeatedly chase after errors that developers could have easily found or prevented, and you will have more time to dedicate to higher level verification tasks, such as anticipating how the software can be misused and checking how the software performs when it is used unexpectedly (see practice #2).

You can further error prevention not only by finding bugs, but also by helping the team architect, manager, and developers pinpoint the source of each error you uncover and by suggesting ways to prevent recurrences of that error.

You can further error prevention not only by finding bugs, but also by helping the team architect, manager, and developers pinpoint the source of each error you uncover and by suggesting ways to prevent recurrences of that error.

For example, say that some of your load tests reveal that a heavy load stops the system. One course of action is to report the problem in the bug tracking system and hope that someone else figures out why it was caused and how to prevent it. However, if other team members don't have the time or understanding to determine how to prevent the error, you are likely to encounter similar performance problems in the future. Another course of action is to work with the team to pinpoint the source of the error and reach a group consensus on how to prevent the error from recurring. Typically, when you combine the expertise of a tester, the developer of the code, and other resourceful team members, you can identify and resolve the problem source faster and more accurately than any team member working independently could. After the problem and resolution are identified, the architect and manager should determine how to implement and enforce the error prevention measure; this should not be responsibility of the tester. However, if your testing indicates that a required error prevention measure is not being followed correctly, it's important to notify the architect or manager so that he or she can address the problem.

NaSPA member Dr. Adam Kolawa is the co-founder and CEO of Parasoft, a leading provider of Automated Error Prevention (AEP) software solutions. Kolawa, co-author of Bulletproofing Web Applications (Hungry Minds 2001), has contributed to and written over 100 commentary pieces and technical articles for publications such as The Wall Street Journal, CIO, Computerworld, Dr. Dobb's Journal, and IEEE Computer; he has also authored numerous scientific papers on physics and parallel processing. His recent media engagements include CNN, CNBC, BBC, and NPR. Kolawa holds a Ph.D. in theoretical physics from the California Institute of Technology, and has been granted ten patents for his recent inventions. In 2001, Kolawa was awarded the Los Angeles Ernst & Young's Entrepreneur of the Year Award in the software category.