PARASOFT®

Unit Testing Best Practices

This paper is part of a series of interviews in which Adam Kolawa—Parasoft CEO and Automated Defect Prevention: Best Practices in Software Management (Wiley-IEEE, 2007) co-author—discusses why, when, and how to apply essential software verification methods. The series also addresses code analysis, code review, memory error detection, message/rotocol testing, functional testing, and load testing. In this interview, Kolawa counters claims that unit testing has peaked, explains how unit testing delivers value above and beyond "early testing," and warns that code coverage is not the best measure of test suite completeness. Read on to learn how unit testing can help your team rapidly modify code with confidence—and how to get started as painlessly as possible.

Unit testing seems much more popular in theory than in practice. Why do you think that's the case?

Well, there's a huge barrier to entry because setting up realistic and useful tests can be quite complex. People tend to have a good understanding of how the application is supposed to work from the outside—how it's supposed to respond to different user actions or operations—but they have a limited grasp of how it's supposed to work "under the hood." This is quite a handicap when it comes to writing unit tests.

With unit tests, you need to create relevant initial conditions so the part of the application you're trying to test behaves like part of the complete system. If these initial conditions aren't set correctly, the test won't be exercising the code in a realistic context, and it won't be terribly useful to you. You can't really tell if something is working properly if you're not testing it under realistic conditions. The context is critical. For instance, a beam might provide proper support under many conditions... but if you're building with it, you really need to know if it will hold up under your specific construction. If it wasn't verified to work in a similar context, you can't rest assured that it will work for you here.

Creating the initial conditions can be difficult, but it's absolutely essential. I think the people who do try to accomplish this are frustrated by the amount of work required, and those who don't are disappointed that they're not getting valuable results.

There's been a lot of debate lately about whether unit testing is going the way of the dinosaur. What do you think?

I think that unit testing is more valuable now than ever. But I think we need a different flavor of unit testing than the one everyone always focuses on, which is early testing. Let me explain. The systems we're building today, such as SOA and RIA, are growing in size and complexity—plus developers are being asked to evolve them faster than ever before. We typically have a good idea of what functionality we need to test because the use cases are usually well-defined in requirements. However, it's difficult to test them in their entirety because they are so complicated—they are connected to the outside world, staging systems are difficult to establish and maintain, each test requires considerable setup, and so on. It might be feasible to test each use case once, but you really need to test all your use cases daily so that you're immediately alerted in case your constant modifications end up breaking them.

And that's where unit testing comes in?

Yes. Unit testing allows you to continuously test parts of the application without the hassles of dealing with the complicated system. Using a technology like Parasoft Tracer, you can create unit test cases that capture the functionality covered in your use cases. From the application GUI or using a SOA or Web test client, you just exercise your use cases while Tracer is enabled. As the use case is executed, Tracer monitors all the objects that are created, all the data that comes in 2 and goes out, and then it creates you a unit test that represents this operation—and even sets up the proper initial conditions.

You can then take this unit test case and execute it on a separate machine, away from the original system. This means that you can use a single machine—even a standard developer desktop—to reproduce the behavior of a complicated system during your verification procedure.

Add all of these tests to your regression test suite, run it continuously, and you'll immediately be alerted if this functionality that you captured is impacted by your code modifications.

Essentially, this helps you address the complexity of today's systems in two ways:

1. It relieves you from having to perform all the time-consuming and tedious work required to set up realistic test conditions.
2. It gives you a test case that you can run apart from the system—a test case you can run continuously and fully automatically.

So unit tests are created after the application can be exercised realistically. How does that mesh with the idea that unit testing is early testing?

This type of unit testing is not early testing at all. It's a completely different flavor of unit testing, and I think it's even more valuable than early testing. But, there's still a time and place for early testing.

Say you have a brand new application that you're writing from scratch. As you implement each chunk of functionality, why not write a unit test that captures how this functionality is supposed to work? At this point, you're familiar with the requirement being implemented and you know all about how the code works ... so this is the best possible time to set up the conditions properly and write a test that really verifies the requirement. While you're at it, specify what requirement it implements in the test case itself. This way, it can always be correlated to the appropriate requirement. In other words, if the test fails, you'll know what requirement was affected.

When you're done writing each test, add it to the regression test suite and execute it continuously—just like you would do with the other type of unit tests.

You could write these tests right after the code is implemented. Or you could write them before you even start working on code, essentially practicing TDD.

Is this requirement-based test development process always a manual one?

This process of writing tests that verify requirements is always a creative one. The developer really needs to think about the code and how to test it effectively. But, there is some help.

Automated unit testing tools like Parasoft Jtest can reduce the work required to implement realistic and useful test cases.

For instance, the object repository stores initialized objects, which are very helpful to use when you're trying to set up realistic initial conditions. The stub library can be used to "stub out" external references so the unit can be tested in isolation. It's not a trivial task to set up stubs and objects manually, so having this assistance can save you a lot of time. Also, Test Case Parameterization can be used to feed additional inputs into the

test cases—be it inputs from a data source, or a stream of automatically generated inputs using corner case conditions.

These things don't relieve you from having to think creatively about what the test should do... but they do help you implement your plan as efficiently as possible.

Another widely-held belief about unit testing is that the more code coverage unit testing achieves, the more valuable it is. Do you agree?

Actually, I think that code coverage is irrelevant in most cases. People want high code coverage... but to achieve something like 80% code coverage or higher, you inevitably end up executing code mindlessly. It's like asking a pianist to cover 100% of the piano keys rather than hit just the keys that make sense in the context of a given piece of music. When he plays the piece, he gets whatever amount of key coverage makes sense.

Another problem is that when code coverage exceeds a certain level (above about 70%) the test suite becomes increasingly difficult to maintain. When the test suite has higher coverage than this, it's usually a sign that methods are executed mindlessly. You typically have very fine-granularity assertions, and these assertions are very sensitive to code changes. As a result, many assertion failures are reported and need to be dealt with each day—but these assertion failures typically don't alert you to any significant changes.

Many people ask me if they can get 100% code coverage. The answer is yes... but it won't do you much good, other than satisfying some internal mandate for 100% code coverage.

How do you measure test suite completeness then?

While a piano concerto is complete when all the notes have been covered, the test suite is complete when every use case has been covered.

Each use case should have at least one test case. This might be a unit test, another test, or a combination of test types. When I have all of my use cases captured as test cases, then I can run my regression test suite nightly and it will tell me if the day's changes broke or negatively impacted the functionality related to my use cases. This gives you a safety net that's essential when you're constantly evolving the application.

Is there any point in even measuring code coverage then?

Yes, it actually tells you some interesting things. Say you have one test case for every use case, and this represents only 40% code coverage. This might mean that 60% of your code is unrelated to your use cases... which is a bit scary. Do you have a lot of useless code? Is the application being written without a clear understanding of what it's supposed to do? Or do you have major omissions in your use case definitions? Maybe you're missing some use cases for your requirements. Or maybe your requirements themselves are incomplete. This is definitely worth exploring.

On the other hand, what if your coverage is high even though you don't have many test cases? This might be a sign that you have well-written, tight code, with a lot of reuse.

Once you've built up a set of test cases that give you good coverage of your use cases, what do you do with them?

You need to ensure that they're maintained. Unit testing is not about creating unit test cases. It's about maintaining unit test cases. If you let them grow out of synch with the application, they quickly become useless.

To keep the process going, you need a supporting infrastructure and workflow. Ideally, each night the infrastructure automatically:

1. Gets the latest code from source control.
2. Runs the entire regression test suite.
3. Determines what assertions failed as a result of the day's modifications.
4. Figures out which developer caused each assertion failure.
5. Distributes this information to the responsible developers.

This is all automated, and doesn't require any development resources at all.

Then, when the developers arrive at work each morning, they import the results into their desktops or IDEs, review any assertion failures reported for the code they authored, and respond to them. When they respond, they are either addressing functional defects in the code or updating the test to reflect the correct behavior of the code.

With this daily process, a little effort each morning goes a long way in terms of extending the test cases' value nd life span—and also, of course, in terms of exposing unexpected impacts as they are introduced so you can nip them in the bud.

Without a supporting process and an infrastructure to drive this process and keep it on track, it's only a matter of time before your unit testing efforts decay.

Any tips on keeping this daily process as painless as possible?

Well, the automated assertion failure assignment and distribution I just mentioned is key. We learned the value of this ourselves, in the Parasoft development team. Years ago, when we started examining our unit testing process, we found that developers would write functional unit tests to verify requirements as code was implemented. However, when test assertions later failed as the application evolved, the failures weren't being addressed promptly.

In each case, someone needed to review the failure and decide if it was an intended change in behavior or an unwanted regression error. Our open source unit test execution tools couldn't tell us what tests were failing from whom, for what, since when. Instead, our nightly often produced a report that said that something like 150 of our 3,000+ tests failed. Each developer could not determine if his own tests failed unless he reviewed the full list of failures—one at a time—to see if it was something that he should fix or if it was a task for someone else. Our system was lacking accountability.

Initially, we tried asking all developers to review all test failures, but that took a lot of time and never became a regular habit. Then, we tried designating one person to review the test failures and distribute the work. However, that person was not fond of the job because it was tedious and the distribution of tasks was not well-received by the others.

Eventually, we built automated error assignment and distribution into our unit testing products, and started using it internally. Now, assertion failures are automatically assigned to the responsible developers based on source control data. If a developer causes one or more assertion failures, he is notified via email, imports only the relevant results into his IDE, and resolves them. As a result, failures are now resolved in days instead of months.

I think I understand what you're saying about keeping the process going. But where do you start… especially when you have an existing application without many test cases?

Working on a code base that has minimal tests or no tests is like walking on eggshells: every move you make has the potential to break something… but with software, the damage is not always immediately obvious. Before you touch another line of code, start building a unit test suite that serves as a change-detection safety net.

This way, you can rest assured that you'll be alerted if modifications impact application behavior. If you know what the use cases for the existing functionality are, you can start by "tracing" unit test cases as you execute the use cases. This way, you'll get a set of test cases that you can run daily to ensure that you're not breaking or changing this core functionality.

Even if you don't have use cases, you can automatically generate what I call a behavioral regression test suite: a baseline unit test suite that captures the code's current functionality. As I said before, test suite maintainability really diminishes above a certain code coverage level, so if your tool gives you a choice between generating a maintainable test suite and a high coverage one, choose the maintainable one. To detect changes from this baseline, ensure that the evolving code base is automatically run against this test suite on a regular basis—ideally, daily. You'll be alerted when the baseline functionality is impacted, and as you explore these impacts, you'll eventually learn more and more about the existing functionality.

Any closing comments?

Because today's applications are so complex, your quality efforts and regression test suites also need to be complex. Unit testing is a great starting point... but it's not a religion. Unit testing alone cannot deliver quality, will not expose all of your defects and change impacts, and is not a silver bullet.

If you really want to build quality into the code and gain a 360 degree view of how your daily modifications affect an existing application's functionality, you need a continuous quality process that includes everything from static analysis, to peer code reviews, to unit testing, to message/ protocol layer testing, to load testing. It takes some effort to get all the components established and working together, but the payoff in terms of team productivity as well as improved application quality is tremendous.

About Parasoft Unit Testing Capabilities

Parasoft has been providing automated unit testing since 1997—when we were the first to bring it to market. Parasoft Quality solutions deliver a complete framework to create, manage, and extract greater value from unit tests. Our patented technologies automatically generate unit tests that not only verify code correctness and reliability at multiple levels, but also capture the code's behavior to establish a maintainable baseline for regression testing. Once this baseline test suite is in place, you can incrementally improve its intelligence and value by:

- Executing your use case scenarios while Parasoft Tracer is enabled. Parasoft Tracer monitors the application as you're interacting with it, then captures those actions in realistic functional unit test cases.
- Extending the automatically-generated xUnit (JUnit, Cactus, CppUnit, NUnit, etc.) test cases by hand, or with capabilities such as a robust object repository, a stub library, Test Case Parameterization, etc. to verify specific requirements.
- Integrating your legacy xUnit test cases (or writing new ones)—allowing you to centralize your test execution, reporting, and coverage analysis and take advantage of our automated error assignment and distribution.

Collectively, these test cases establish a safety net that alerts the team when modifications impact application behavior.

To ensure long-term success with unit testing, it's essential to implement unit testing in such a way that not only leverages the automation provided by enterprise-grade technologies, but also addresses the process and workflow details essential to a sustainable, scalable quality process. Parasoft Services can assist you with this; we have over 10 years of experience helping organizations establish unit testing as part of a continuous quality process that delivers greater productivity as well as significantly fewer software defects.

About Parasoft

Parasoft researches and develops software solutions that help organizations deliver defect-free software efficiently. By integrating development testing, API testing, and service virtualization, we reduce the time, effort, and cost of delivering secure, reliable, and compliant software. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing, requirements traceability, coverage analysis, functional and load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives.

For more information, visit http://www.parasoft.com

Contacting Parasoft

GLOBAL HEADQUARTERS
101 E. Huntington Drive,
2nd Floor Monrovia, CA 91016
USA
Toll Free: (888) 305-0041
Tel: (626) 256-3680
Email: info@parasoft-embedded.com
URL: http://www.parasoft.com

ASIA HEADQUARTERS
11F, 508 Chung Hsiao E. Rd.
Section 5
Taipei, Taiwan
Phone +886 2 6636-8090
Support +886 2 6636 8090
Email: info-psa@parasoft.com

Parasoft Poland
Parasoft SA
Kielkowskiego 9, Krakow 30-704
Poland
Phone +48 12 290 91 01
Support +48 12 296 6088
Email: office-pl@parasoft.com

Parasoft Netherlands B.V.
Lange Voorhout 70
2514 EH Den Haag
The Netherlands
Phone +31 70 3922000
Email: netherlands@parasoft.com

Parasoft Deutchland GmbH
m Zollhafen 24
Kranhaus Süd
50678 Köln
Phone +49 221 650 784 18
Email: info-de@parasoft.com

Parasoft UK Ltd
The Lansdowne Building
2 Lansdowne Road
Croydon.
CR9 2ER
United Kingdom
Phone +44 (0)208 263 6005
Email: sales@parasoft-uk.com

Other Locations
See http://www.parasoft.com/contacts

© 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.