



IMPROVE UNIT TESTING FOR JAVA WITH AUTOMATION

BEST PRACTICES FOR JAVA DEVELOPERS

Table of Contents

03 Introduction: Improve Unit Testing

- 03 What Is Unit Testing?
- 03 Improve Unit Testing With Automated Testing Tools

04 Best Practices for Developers

- 04 Unit Testing Best Practices: How to Get the Most out of Your Test Automation
- 10 JUnit Tutorial: Setting Up, Writing, and Running Java Unit Tests
- 17 Mocking in Java: How to Automate a Java Unit Test, Including Mocking and Assertions
- 24 How to Create JUnit Parameterized Tests
- 34 Get More Out of Unit Testing and Reduce Maintenance Efforts With Runtime Analysis

Introduction: Improve Unit Testing

WHAT IS UNIT TESTING?

Unit testing is the practice of testing individual units or components of an application in order to validate that each of those units is working properly. Generally, a unit should be a small part of the application – in Java it's often a single class. Note that there is no strict definition of "unit" here, and it is up to the developer to decide the scope of tested code for each test.

People sometimes contrast the term "unit testing" with "integration testing" or "end-to-end testing". The difference is that, generally, unit testing is done to validate the behavior of an individual testable unit, whereas integration tests are validating the behavior of multiple components together, or the application as a whole. As mentioned above, the definition for what constitutes a "unit" is not strictly defined, and it's up to you to decide the scope for each test. If the scope is too broad, it may not be possible to determine why a test failure occurred.

With these challenges, unit testing just isn't easy. It requires a lot of development skill and effort, and it takes commitment and time to maintain test suites. This ebook provides helpful tips and techniques as well as best practices to help you improve unit testing with JUnit and Parasoft Jtest.

IMPROVE UNIT TESTING WITH AUTOMATED TESTING TOOLS

With automated testing tools, developers are able to reduce late-cycle defects with better unit tests and automated static code analysis. They can focus more time on new feature development for the business. Developers also benefit from immediate feedback. They're able to rapidly identify whether their code changes are breaking functionality in the application and addressing it quickly.

Parasoft has focused on improving automated testing for over 30 years. [Parasoft Jtest](#) is a key enabler of delivering quality at speed for unit testing. This integrated Java solution enables development teams to be agile and deliver faster without sacrificing quality, making the business successful.

Best Practices for Developers

UNIT TESTING BEST PRACTICES: HOW TO GET THE MOST OUT OF YOUR TEST AUTOMATION

Unit testing is a well-known practice, but there's lots of room for improvement! This section will cover the most effective unit testing best practices, including approaches for maximizing your automation tools along the way. It will also discuss code coverage, mocking dependencies, and overall testing strategies.

WHY UNIT TEST?

Unit testing is a proven technique for ensuring software quality, with plenty of benefits. Here are (more than) a few great reasons to unit test:

- » Unit testing **validates** that each piece of your software not only works properly today, but continues to work in the future, providing a solid foundation for future development.
- » Unit testing **identifies defects at early stages** of the production process, which reduces the costs of fixing them in later stages of the development cycle.
- » Unit-tested code is generally **safer to refactor**, since tests can be re-run quickly to validate that behavior has not changed.
- » Writing unit tests forces developers to consider how well the production code is designed in order to make it **suitable for unit testing**, and makes developers look at their code from a **different perspective**, encouraging them to consider corner cases and error conditions in their implementation.
- » Including unit tests in the **code review process** can reveal how the modified or new code is supposed to work. Plus, reviewers can confirm whether the tests are good ones or not.

It's unfortunate that all too often, developers either don't write unit tests at all, don't write enough tests, or they don't maintain them. Unit tests can sometimes be tricky to write, or time-consuming to maintain. Sometimes there's a deadline to meet, and it feels like writing tests will make us miss that deadline. But not writing enough unit tests or not writing good unit tests is a risky trap to fall into.

So please consider the following best-practice recommendations on how to write clean, maintainable, automated tests that give you all the benefits of unit testing, with a minimum amount of time and effort.

UNIT TESTING BEST PRACTICES

Let's look at some best practices for building, running, and maintaining unit tests, to achieve the best results.

Unit Tests Should Be Trustworthy

The test must fail if the code is broken and only if the code is broken. If it doesn't, we cannot trust what the test results are telling us.

Unit Tests Should Be Maintainable and Readable

When production code changes, tests often need to be updated, and possibly debugged as well. So, it must be easy to read and understand the test, not only for whoever wrote it, but for other developers as well. Always organize and name your tests for clarity and readability.

Unit Tests Should Verify a Single Use Case

Good tests validate one thing and one thing only, which means that typically, they validate a single use-case. Tests that follow this best practice are simpler and more understandable, and that is good for maintainability and debugging. Tests that validate more than one thing can easily become complex and time-consuming to maintain. Don't let this happen.

Another best practice is to use a minimal number of assertions. Some people recommend just one assertion per test (this may be a little too restrictive); the idea is to focus on validating only what is needed for the use-case you are testing.

Unit Tests Should Be Isolated

Tests should be runnable on any machine, in any order, without affecting each other. If possible, tests should have no dependencies on environmental factors or global/external state. Tests that have these dependencies are harder to run and usually unstable, making them harder to debug and fix, and end up costing more time than they save (see [trustworthy](#), above).

Martin Fowler, a few years ago, wrote about "solitary" vs "sociable" code, to describe dependency usage in application code, and how tests need to be designed accordingly. In his article, "solitary" code doesn't depend on other units (it's more self-contained), whereas "sociable" code does interact with other components. If the application code is solitary, then the test is simple, but for sociable code under test, you can either build a "solitary" or "sociable" test. A "sociable test" would rely on real dependencies in order to validate behavior, whereas a "Solitary test" isolates the code under test from dependencies. This is visually shown in [Figure 1](#). You can use mocks to isolate the code under test and build a "solitary" test for "sociable" code. We'll look at how to do that below.

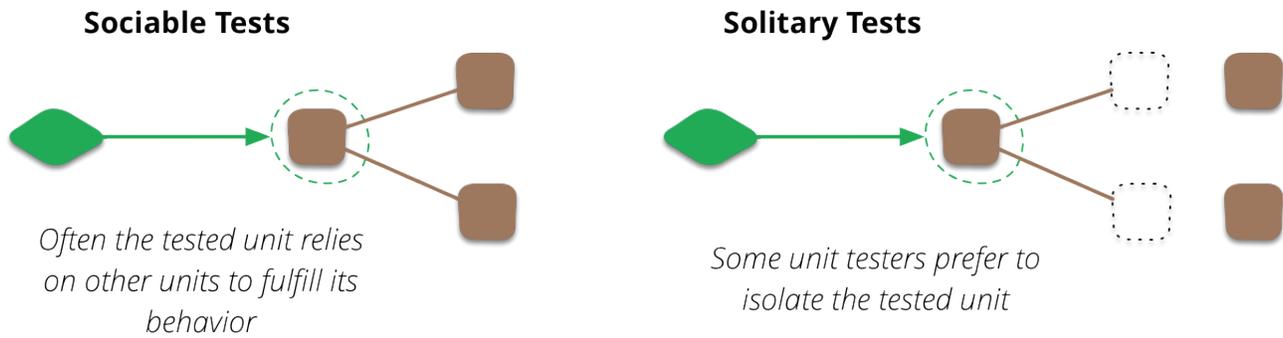


Figure 1:
Comparison of Sociable
and Solitary Tests.
Source: Martin Fowler,
2014, "UnitTest"

In general, using mocks for dependencies makes our life easier as testers, because we can generate "solitary tests" for sociable code. A sociable test for complex code may require a lot of set up and may violate the principles of being isolated and repeatable. But since the mock is created and configured in the test, it is self-contained, and we have more control over the behavior of dependencies. Plus, we can test more code paths. For instance, you can return custom values or throw exceptions from the mock, in order to cover boundary or error conditions.

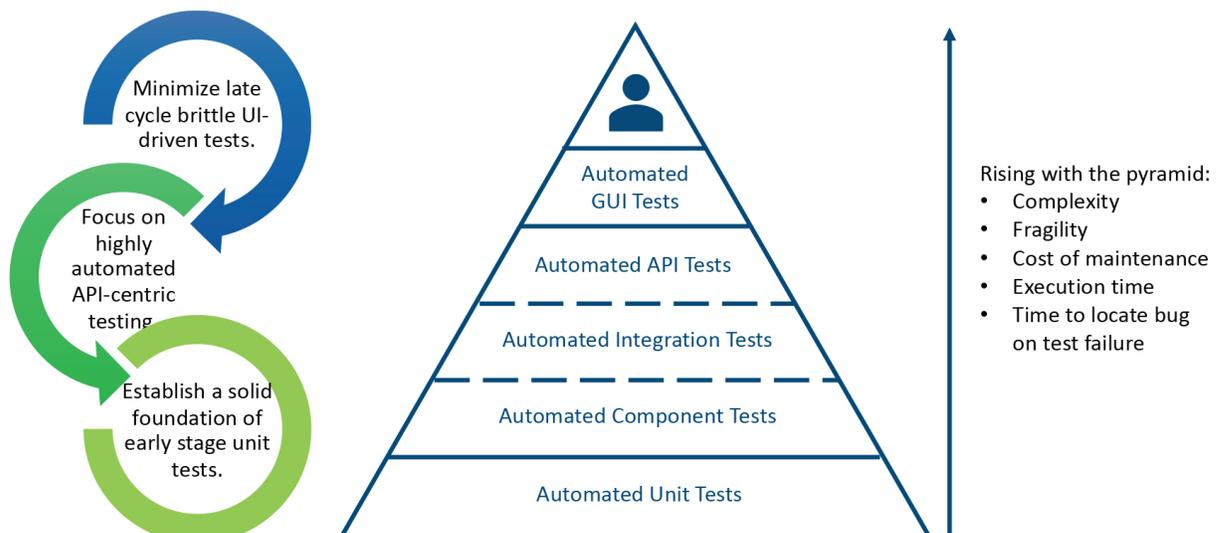
Unit Tests Should Be Automated

Make sure tests are being run in an automated process. This can be daily, or every hour, or in a Continuous Integration or Delivery process. The reports need to be accessible to and reviewed by everyone on the team. As a team, talk about which metrics you care about: code coverage, modified code coverage, number of tests being run, performance, etc. A lot can be learned by looking at these numbers, and a big shift in those numbers often indicates regressions that can be addressed immediately.

Use a Good Mixture of Unit and Integration Tests

Michael Cohn's book, [Succeeding with Agile: Software Development Using Scrum](#), addresses this using a testing pyramid model (see illustration in Figure 2). This is a commonly used model to describe the ideal distribution of testing resources. The idea is that as you go up the pyramid, tests are usually more complex to build, more fragile, slower to run, and slower to debug. Lower levels are more isolated and more integrated, faster, and simpler to build and debug. Therefore, automated unit tests should make up the bulk of your tests.

Figure 2:
Testing Pyramid Model



Unit tests should validate all of the details, the corner cases and boundary conditions, etc. Component, integration, UI, and functional tests should be used more sparingly, to validate the behavior of the APIs or application as a whole. Manual tests should be a minimal percentage of the overall pyramid structure but are still useful for release acceptance and exploratory testing. This model provides organizations with a high level of automation and test coverage, so that they can scale up their testing efforts and keep the costs associated with building, running, and maintaining tests at a minimum.

Unit Tests Should Be Executed Within an Organized Test Practice

In order to drive the success of your testing at all levels, and make the unit testing process scalable and sustainable, you will need some additional practices in place. First of all, this means **writing unit tests as you write your application code**. Some organizations write the tests before the application code ([test-driven](#) or [behavior-driven](#) programming). The important thing is that tests go hand-in-hand with the application code. The tests and application code should even be reviewed together in the code review process. Reviews help you understand the code being written (because they can see the expected behavior) and improve tests too!

Writing tests along with code isn't just for new behavior or planned changes, it's critical for bug fixes too. **Every bug you fix should have a test that verifies the bug is fixed**. This ensures that the bug stays fixed in the future.

Adopt a zero-tolerance policy for failing tests. If your team is ignoring test results, then why have tests at all? Test failures should indicate real issues so address those issues right away – before they waste QA's time, or worse, they get into the released product.

The longer it takes to address failures, the more time and money those failures will ultimately cost your organization. So, you should run tests during refactoring, run tests right before you commit code, and don't let a task be considered "done" until the tests are passing too.

Finally, **maintain those tests**. As mentioned previously, if you're not keeping those tests up to date when the application changes, they lose their value. Especially if they are failing, failing tests are costing time and money to investigate each time they fail. Refactor the tests as needed, when the code changes.

As you can see, maximizing your returns on money and time invested in your unit tests requires some investment in applying best practices. But in the end, the rewards are worth the initial investment.

WHAT ABOUT CODE COVERAGE?

In general, code coverage is a measurement of how much of the production code is executed while your automated tests are running. By running a suite of tests and looking at code coverage data, you can get a general sense of how much of your application is being tested.

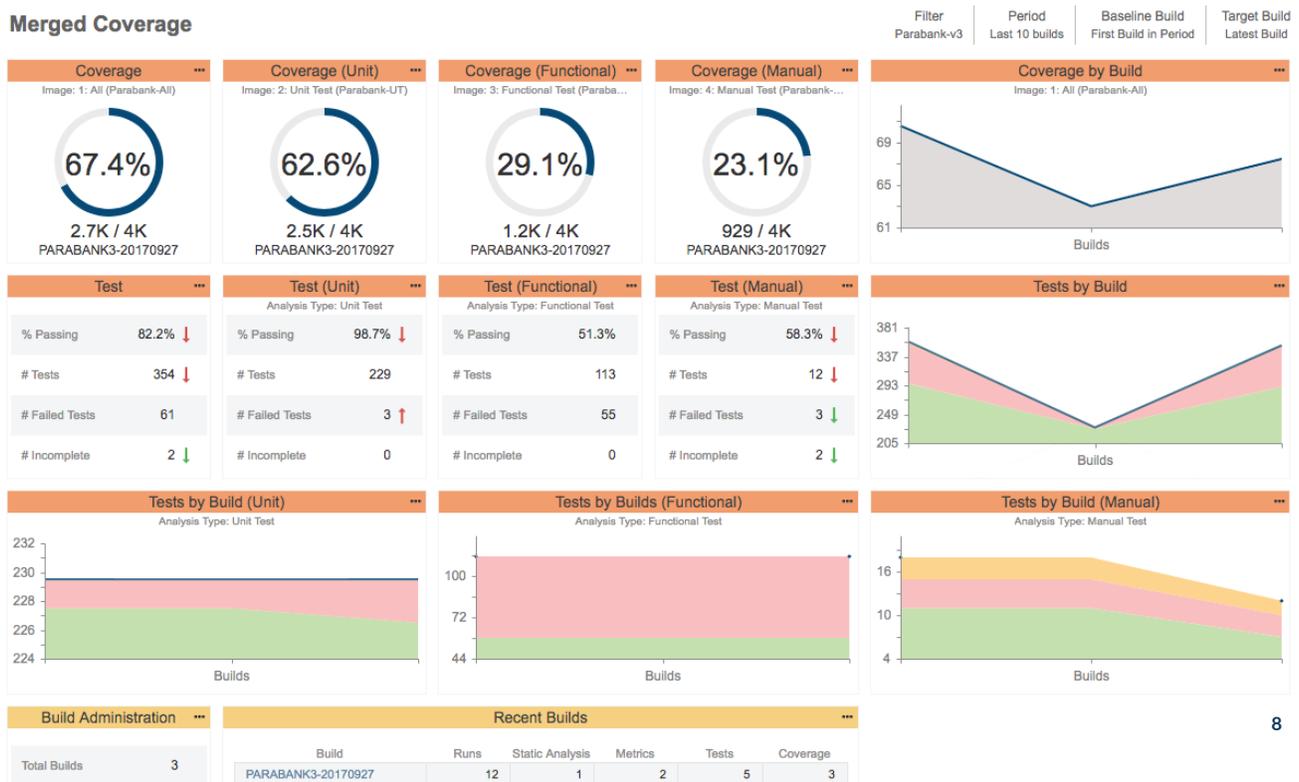
There are many kinds of code coverage – the most common ones are line coverage and branch coverage. Most tools focus on line coverage, which just tells you if a specific line was covered. Branch is more granular, as it tells you if each path through the code is covered.

Code coverage is an important metric but remember that increasing it is a means to an end. It's great for finding gaps in testing, but it's not the only thing to focus on. Be careful not to spend too much effort trying to achieve 100% coverage – it may not even be possible or feasible, and really the quality of your tests is the important thing. That being said, achieving at least 60% coverage for your projects is a good starting point, and 80% or more is a good goal to set. Obviously, it's up to you to decide what that goal should be.

It's also valuable if you have automated tools that not only measure code coverage but also keep track how much modified code is being covered by tests, because this can provide visibility into whether enough tests are being written along with changes in production code.

Figure 3 shows an example code coverage report from [Parasoft DTP](#), the reporting and analytics hub, that you can navigate through if you are using [Parasoft Jtest](#) for your [unit testing](#):

Figure 3:
Example Code
Coverage Report



Another thing to keep in mind is that, when writing new tests, be careful of focusing on line coverage alone, as single lines of code can result in multiple code paths, so make sure your tests validate these code paths. Line coverage is a useful quick indicator, but it isn't the only thing to look for.

The most obvious way to increase coverage is simply to add more tests for more code paths, and more usecases of the method under test. A powerful way to increase coverage is to use parameterized tests. For JUnit 4, there was the built in JUnit 4 Parameterized functionality and third-party libraries like JunitParams. JUnit 5 has built-in parameterization.

Finally, if you aren't already tracking test coverage, we highly recommend you start. There are plenty of tools that can help, like Parasoft Jtest. Start by measuring your current coverage numbers, then set goals for where it should be, address important gaps first, and then work from there.

SUMMARY OF UNIT TESTING BEST PRACTICES

Although unit testing is a proven technique for ensuring software quality, it's still considered a burden to developers and many teams are still struggling with it. In order to get the most out of testing and automated testing tools, tests must be trustworthy, maintainable, readable, self-contained, and be used to verify a single use case. Automation is key to making unit testing workable and scalable.

In addition, software teams need to practice good testing techniques, such as writing and reviewing tests alongside application code, maintaining tests, and ensuring that failed tests are tracked and remediated immediately. Adopting these unit testing best practices can quickly improve your unit testing outcomes.

JUNIT TUTORIAL: SETTING UP, WRITING, AND RUNNING JAVA UNIT TESTS

This tutorial will help you understand the basics and scale your unit testing practice like a pro.

Before we go into JUnits, let's talk a little bit about unit testing and regression testing and why they matter in general. We'll get into some good examples as we go on.

UNIT TESTING

Unit testing is a form of white box testing, in which test cases are based on internal structure. The tester chooses inputs to explore particular paths and determines the appropriate output. The purpose of unit testing is to examine the individual components or piece of methods/classes to verify functionality, ensuring the behavior is as expected.

The exact scope of a "unit" is often left to interpretation, but a nice rule of thumb is for a unit to contain the least amount of code that performs a standalone task (e.g. a single method or class). There is a good reason that we limit scope when unit testing -- if we construct a test that incorporates multiple aspects of a project, we have shifted focus, from functionality of a single method, to interaction between different portions of the code. If the test fails, we don't know why it failed, and we are left wondering whether the point of failure was within the method we were interested in, or in the dependencies associated with that method.

REGRESSION TESTING

Complementing unit testing, regression testing makes certain that the latest fix, enhancement, or patch did not break existing functionality, by testing the changes you've made to your code. Changes to code are inevitable, whether they are modifications of existing code or adding packages for new functionality -- your code will certainly change. It is in this change that the most danger lies, so with that in mind, regression testing is a must.

WHAT IS JUNIT?

JUnit is a unit testing framework for the Java programming language that plays a big role in regression testing. An open-source framework, it is used to write and run repeatable automated tests.

As with anything else, the JUnit framework has evolved over time. The major change to make note of is the introduction of annotations that came along with the release of JUnit 4, which provided an increase in organization and readability of JUnits. The rest of this blog post will be written from usages of JUnit 4 and 5.

HOW TO SET UP JUNIT

The more common IDEs, such as Eclipse and IntelliJ, will already have JUnit functionality installed by default. If one is not using an IDE and perhaps relying solely on a build system such as Maven or Gradle, the installation of JUnit 4/5 is handled via the pom.xml or build.gradle, respectively. It is important to note that JUnit 5 was split into three modules, one of those being a vintage module that supports annotation/syntax of JUnit 4.

JUNIT 4

To add JUnit 4 to your Maven, build the following to the pom.xml.
Be mindful of version:

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.12</version>
<scope>test</scope>
</dependency>
```

For Gradle, add the following to the build.gradle:

```
apply plugin: 'java'

dependencies {
    testCompile 'junit:junit:4.12'
}
```

JUNIT 5

Adding JUnit 5 is a bit different. Because of the modular fashion of JUnit 5, a BOM is used to import all aspects. If only particular classes are needed, individual groups or artifacts can be specified.

To add JUnit 5 to Maven, add the following to pom.xml:

```
<dependency>
<groupId>org.junit</groupId>
<artifactId>junit-bom</artifactId>
<version>5.2.0</version>
<scope>test</scope>
</dependency>
```

For Gradle, add the following to the build.gradle:

```
apply plugin: 'java'

dependencies {
    implementation 'org.junit:junit-bom:5.2.0'
}
```

Although not typically needed, the raw .jar file, which allows one to use the JUnit framework, is also accessible to manually put on the class path. Git [houses the code](#) for JUnit. JUnit 4 has the .jar available to download directly. It is most common to include the JUnit 5 .jar files using a build management system, like Maven or Gradle. See the JUnit 5 docs online for specifics.

WRITING UNIT TESTS: THE ANATOMY OF A JUNIT

Now that we talked a little about unit testing and setup, let's move on to actual construction and execution of these tests. To best illustrate the creation of JUnits, we want to start with something basic. In the example image below, we have a simple method (**left**) that converts Fahrenheit to Celsius, and the JUnit (**right**) associated with our method. The JUnits are numbered in sections below and we'll discuss each in detail.

<pre>package examples.nbank; public class Conversion { public double tempConversion (double temperature, String unit) { if (unit.equals("F")) return (temperature - 32) * (5.0/9.0); else return (temperature * (9.0/5.0)) + 32; } }</pre>	<pre>package examples.nbank; 1 import static org.junit.Assert.assertEquals; 2 import org.junit.*; 3 public class ConversionTest { 4 @Test 5 public void testTempConversion() throws Throwable { // Given 6 Conversion underTest = new Conversion(); // When 7 double temperature = 80.0d; String unit = "F"; 8 double result = underTest.tempConversion(temperature, unit); // Then - assertions for result of method tempConversion(double, String) 9 assertEquals(176.0d, result, 0.0); } }</pre>
--	--

Figure 4:
Example Code With
Example Unit Test

Sections 1 and 2

These are imports for the JUnit libraries needed to leverage the testing framework. The imported libraries can be specified down to a particular functionality of JUnit but are commonly imported with asterisks to have access to all functionality.

Section 3

This has the start of our test class, and the important thing to take note of here is the naming convention for the class, which follows `ClassNameTest`.

Section 4

Here, we see our first JUnit-specific syntax, an annotation. Annotations are extremely important when creating JUnits. This is how JUnit knows what to do with the processing section of code. In our example case, we have an `@Test` annotation, which tells JUnit that the public void method to which it is attached can be run as a test case.

There are many other annotations, but the more common are `@Before` (which runs some statement/precondition before `@Test`, public void), `@After` (which runs some statement after `@Test`, public void e.g. resetting variables, deleting temporary files, variables, etc.), and `@Ignore` (which ignores some statement during test execution -- note that `@BeforeClass` and `@AfterClass` are used for running statements before and after all test cases, public static void, respectively).

Section 5

The takeaway here is again naming convention. Note the structure, `testMethodName`.

Section 6

Here we construct a new instance of our class object. This is necessary so we can call the method we are testing on something. Without this object instance, we cannot test the method.

Section 7

Variables associated with the method need to be established, so here we declare variables corresponding to our method. These should be given meaningful values (note: if a parameter is an object, one can instantiate it, or mock it), so that our test has meaning.

Section 8

This variable declaration could be argued as optional, but it's worthwhile for the sake of organization and readability. We assign the results of our method being tested to this variable, using it as needed for assertions and such.

Section 9

The assert methods (which are part of the `org.junit.Assert` class) are used in determining pass/fail status of test cases. Only failed assertions are recorded. Like with annotations, there are many assert options. In our example JUnit above, we use `assertEquals(expected, actual, delta)`. This takes in the **expected outcome**, which the user defines, the **actual**, which is the result of the method being called, and the **delta**, which allows for implementing an allowed deviation between expected and actual values. The purpose of an assertion is validation. Although not required to run your JUnit, failing to add assertions arguably defeats the purpose of your test. Without assertions, you have no verification and at most a smoke test, which gives feedback only when a test errors out.

HOW TO RUN A JUNIT

Choose your own adventure! Here, we will look at three ways to run JUnits: straight from the command line, from the IDE (Eclipse and IntelliJ), and using build systems (Maven and Gradle).

How to Run a JUnit From the Command Line

To run a JUnit directly from the command line, you need a few things: JDK on your path, raw Junit jar file, and the test cases. The command is as follows (this example is for JUnit 4):

```
java -cp /path/to/junit.jar org.junit.runner.JUnitCore <test class name>
```

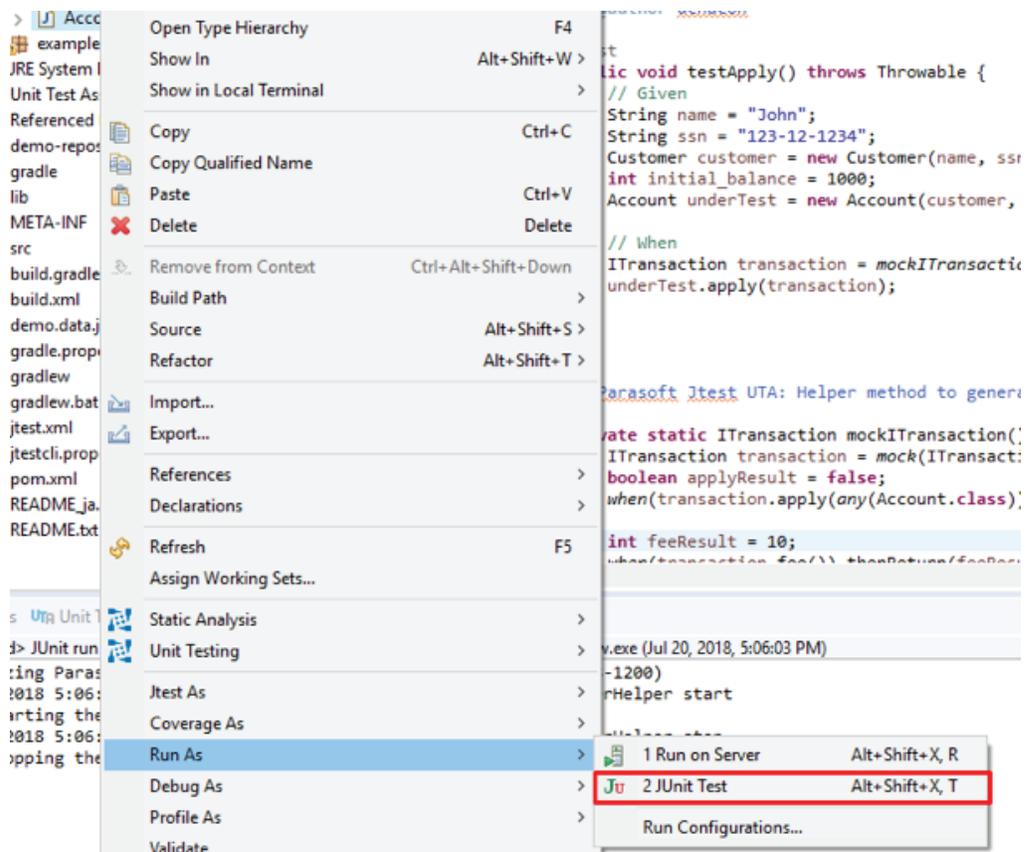
NOTE: it is unlikely in a professional setting that one would be running a test manually from the command line, without some build system, but the ability is there.

How to Run a JUnit From the IDE

Eclipse

To run from Eclipse, from your Package Explorer locate your JUnit test, in whichever folder you have designated it to. Right-click, and move down to Run As JUnit Test. This will execute your test and open a new JUnit window if not already open.

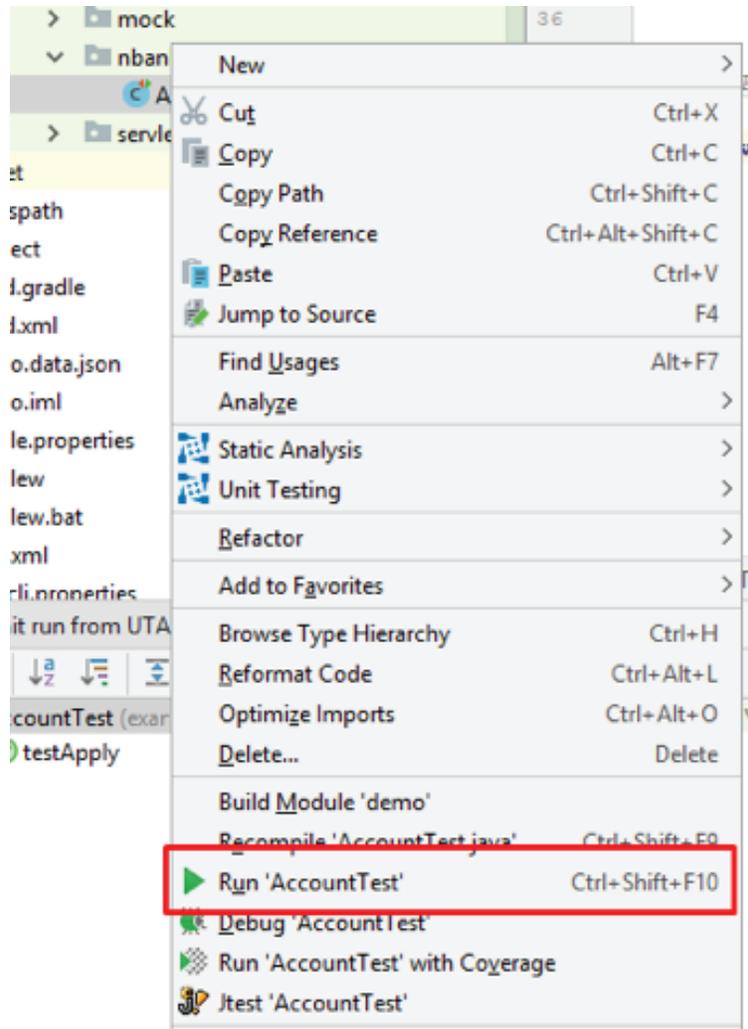
Figure 5:
Eclipse Menu to Run
as JUnit Test



IntelliJ

Running a test in IntelliJ is very similar to Eclipse. From the Project window, locate test, right-click, and select Run 'testName'. Like Eclipse, a JUnit window will open with the results of the test.

Figure 6:
IntelliJ Menu to Run
a Unit Test



How to Run a JUnit Using Build Systems

Maven

Maven made running tests simple. Ensure you are in the proper location from your command line, and the project pom.xml is properly configured. Then you can run the following to execute your JUnits:

To run entire test suite:

```
mvn test
```

To run single/specific test(s):

```
mvn -Dtest=TestName test
```

Gradle

Gradle, like Maven, made running tests simple.

To run entire test suite:

```
gradlew test
```

To run single/specific test(s):

```
gradlew -Dtest.single=testName test
```

NOTE: Maven and Gradle are their own monster -- what is shown here is minimal to cover the basics.

CONTINUING WITH UNIT TESTING

Our example ran through a very simple snippet of code, and of course, this is just the start of unit testing. More complex methods call databases or other methods, but to reassure functionality we need isolation, which we achieve through mocking. Mocking helps us isolate units of code to focus our validation. Frameworks commonly used for mocking are Mockito and PowerMock.

The benefits of unit testing are clear:

- » Identify defects with isolation and focused testing.
- » Assure behavior of individual methods or pieces of methods.
- » Helps to ensure addition or modification of code does not break the application.
- » Boundary analysis makes it easier to check for invalid/bad input.
- » Test every aspect of the method to increase code coverage.

It's helpful to deploy powerful unit testing tools like [Parasoft Jtest](#) that can remedy much of the pain associated with JUnits and save developers valuable time.

MOCKING IN JAVA: HOW TO AUTOMATE A JAVA UNIT TEST, INCLUDING MOCKING AND ASSERTIONS

What is mocking in Java? You can auto-generate a unit test with a single button click, including all of the mocking and validations.

Good unit tests are a great way to make sure that your code works today and continues to work in the future. A comprehensive suite of tests, with good code-based and behavior-based coverage, can save an organization a lot of time and headaches. And yet, it is not uncommon to see projects where not enough tests are written. In fact, some developers have even been arguing against their use completely.

WHAT MAKES A GOOD UNIT TEST?

There are many reasons why developers don't write enough unit tests. One of the biggest reasons is the amount of time they take to build and maintain, especially in large, complex projects. In complex projects, often a unit test needs to instantiate and configure a lot of objects. This takes a lot of time to set up and can make the test as complex (or more complex) than the code it is testing, itself.

Let's look at an example in Java:

```
public LoanResponse requestLoan(LoanRequest loanRequest,
LoanStrategy strategy)
{
    LoanResponse response = new LoanResponse();
    response.setApproved(true);
    if (loanRequest.getDownPayment().compareTo(loanRequest.
getAvailableFunds()) > 0)
    {
        response.setApproved(false);
        response.setMessage("error.insufficient.funds.for.down.
payment");
        return response;
    }
    if (strategy.getQualifier(loanRequest)
< strategy.getThreshold(adminManager))
    {
        response.setApproved(false);
        response.setMessage(getErrorMessage());
    }
    return response;
}
```

Here we have a method that processes a `LoanRequest`, generating a `LoanResponse`. Note the `LoanStrategy` argument, which is used to process the `LoanRequest`. The strategy object may be complex – it may access a database, an external system, or throw a `RuntimeException`. To write a test for `requestLoan()`, we need to be aware of which type of `LoanStrategy` we're testing with and we probably need to test the method with a variety of `LoanStrategy` implementations and `LoanRequest` configurations.

A unit test for `requestLoan()` may look like this:

```
@Test public void testRequestLoan() throws Throwable
{
    // Set up objects
    DownPaymentLoanProcessor processor = new
    DownPaymentLoanProcessor();

    LoanRequest loanRequest = LoanRequestFactory.create(1000, 100, 10000);
    LoanStrategy strategy = new AvailableFundsLoanStrategy();

    AdminManager adminManager = new AdminManagerImpl();
    underTest.setAdminManager(adminManager);
    Map<String, String> parameters = new HashMap<>();
    parameters.put("loanProcessorThreshold", "20");

    AdminDao adminDao = new InMemoryAdminDao(parameters);
    adminManager.setAdminDao(adminDao);

    // Call the method under test
    LoanResponse response = processor.requestLoan(loanRequest, strategy);

    // Assertions and other validations
}
```

As you can see, there's a whole section of our test which just creates objects and configures parameters. It wasn't obvious looking at the `requestLoan()` method what objects and parameters need to be set up. To create this example, we had to run the test, add some configuration, then re-run again and repeat the process over and over. We spent too much time figuring out how to configure the `AdminManager` and the `LoanStrategy` instead of focusing on our method and what needed to be tested there. And I still need to expand our test to cover more `LoanRequest` cases, more strategies, and more parameters for `AdminDao`.

Additionally, by using real objects to test with, this test is actually validating more than just the behavior of `requestLoan()` – we're depending on the behavior of `AvailableFundsLoanStrategy`, `AdminManagerImpl`, and `AdminDao` in order for our test to run. Effectively, we're testing those classes too. In some cases, this is desirable, but in other cases it is not. Plus, if one of those other classes changes, the test may start failing even though the behavior of `requestLoan()` didn't change. For this test, we would rather isolate the class under test from its dependencies.

MOCKING IN JAVA

One solution for the complexity problem is to mock those complex objects. For this example, let's start by using a mock for the `LoanStrategy` parameter:

```
@Test
public void testRequestLoan() throws Throwable
{
    // Set up objects
    DownPaymentLoanProcessor processor = new
DownPaymentLoanProcessor();
    LoanRequest loanRequest = LoanRequestFactory.create(1000,
100, 10000);
    LoanStrategy strategy = Mockito.mock(LoanStrategy.class);

    Mockito.when(strategy.getQualifier(any(LoanRequest.class))).
thenReturn(20.0d);

    Mockito.when(strategy.getThreshold(any(AdminManager.
class))).thenReturn(20.0d);

    // Call the method under test
    LoanResponse response = processor.
requestLoan(loanRequest, strategy);

    // Assertions and other validations
}
```

Let's look at what's happening here. We create a mocked instance of `LoanStrategy` using `Mockito.mock()`. Since we know that `getQualifier()` and `getThreshold()` will be called on the strategy, we define the return values for those calls using `Mockito.when(...).thenReturn()`. For this the test, we don't care what the `LoanRequest` instance's values are, nor do we need a real `AdminManager` anymore because `AdminManager` was only used by the real `LoanStrategy`.

Additionally, since we aren't using a real `LoanStrategy`, we don't care what the concrete implementations of `LoanStrategy` might do. We don't need to set up test environments, dependencies, or complex objects. We are focused on testing `requestLoan()` – not `LoanStrategy` or `AdminManager`. The code-flow of the method under test is directly controlled by the mock.

This test is a lot easier to write with Mockito than having to create a complex `LoanStrategy` instance. But there are still some challenges.

- » For complex applications, tests may require lots of mocks.
- » If you are new to Mockito, you need to learn its syntax and patterns.
- » You may not know which methods need to be mocked.
- » When the application changes, the tests (and mocks) need to be updated too.

SOLVING MOCKING CHALLENGES WITH A JAVA UNIT TEST GENERATOR

We designed [Parasoft Jtest](#) to help address the challenges above and manage the risks of Java software development.

On the unit testing side of things, Parasoft Jtest helps you automate some of the most difficult parts of creating and maintaining unit tests with mocks. For the above example, it can auto-generate a test for `requestLoan()` with a single button-click, including all of the mocking and validations you see in the example test.

Figure 7:
Test Generation
With Parasoft Jtest



Below, the “Regular” action in the Parasoft Jtest *Unit Test Assistant* Toolbar generates the following test:

```
@Test public void testRequestLoan() throws Throwable
{
    // Given    DownPaymentLoanProcessor underTest = new
DownPaymentLoanProcessor();
    // When
double availableFunds = 0.0d; // UTA: default value
double downPayment = 0.0d; // UTA: default value
double loanAmount = 0.0d; // UTA: default value

LoanRequest loanRequest =
LoanRequestFactory.create(availableFunds, downPayment,
loanAmount);
LoanStrategy strategy = mockLoanStrategy();
LoanResponse result = underTest.requestLoan(loanRequest,
strategy);
    // Then
    // assertNotNull(result);
}
```

All the mocking for this test happens in a helper method:

```
private static LoanStrategy mockLoanStrategy() throws
Throwable
{
    LoanStrategy strategy = mock(LoanStrategy.class);
    double getQualifierResult = 0.0d; // UTA: default value
    when(strategy.getQualifier(any(LoanRequest.class))).
thenReturn(getQualifierResult);

    double getThresholdResult = 0.0d; // UTA: default value
```

```

        when(strategy.getThreshold(any(AdminManager.class))).
            thenReturn(getThresholdResult);

        return strategy;
    }

```

All the necessary mocking is set up for us – Parasoft Jtest detected the method calls to `getQualifier()` and `getThreshold()` and mocked the methods. Once we configure values in the unit test for `availableFunds`, `downPayment`, etc, the test is ready to run (we could also generate a parameterized test for better coverage!). Note also that the assistant provides some guidance as to which values to change by its comments, “UTA: default value”, making testing easier.

This saves a lot of time in generating tests, especially if we don’t know what needs to be mocked or how to use the Mockito API.

HANDLING CODE CHANGES

When the application logic changes, the tests often need to change also. If the test is well-written, it should fail if you update the code without updating the test. Often, the biggest challenge in updating the test is understanding what needs to be updated, and how exactly to perform that update. If there are lots of mocks and values, it can be difficult to track down what the necessary changes are.

To illustrate this, let’s make some changes to the code under test:

```

public LoanResponse requestLoan(LoanRequest loanRequest,
    LoanStrategy strategy)
{
    ...
    String result = strategy.validate(loanRequest);
    if (result != null && !result.isEmpty()) {
        response.setApproved(false);
        response.setMessage(result);
        return response;
    }
    ...
    return response;
}

```

We have added a new method to `LoanStrategy` – `validate()`, and are now calling it from `requestLoan()`. The test may need to be updated to specify what `validate()` should return.

Without changing the generated test, let's run it within the Parasoft Jtest Unit Test Assistant:

Figure 8:
Example of Mocking
in Parasoft Jtest

testRequestLoan

 **Mocking**

Possibly mockable method call - `LoanStrategy.validate(LoanRequest) : String`

[Highlight](#) [Go to](#) [Mock it](#)

Parasoft Jtest detected that `validate()` was called on the mocked `LoanStrategy` argument during our test run. Since the method has not been set up for the mock, the assistant recommends that we mock the `validate()` method. The “Mock it” quick-fix action updates the test automatically. This is a simple example – but for complex code where it isn't easy to find the missing mock, the recommendation and quick-fix can save us a lot of debugging time.

After updating the test using the quick fix, we can see the new mock and set the desired value for `validateResult`:

```
private static LoanStrategy mockLoanStrategy() throws
Throwable {

    LoanStrategy strategy = mock(LoanStrategy.class);
    String validateResult = ""; // UTA: default value

    when(strategy.validate(any(LoanRequest.class))).
thenReturn(validateResult);
    double getQualifierResult = 20.0d;

    when(strategy.getQualifier(any(LoanRequest.class))).
thenReturn(getQualifierResult);
    double getThresholdResult = 20.0d;

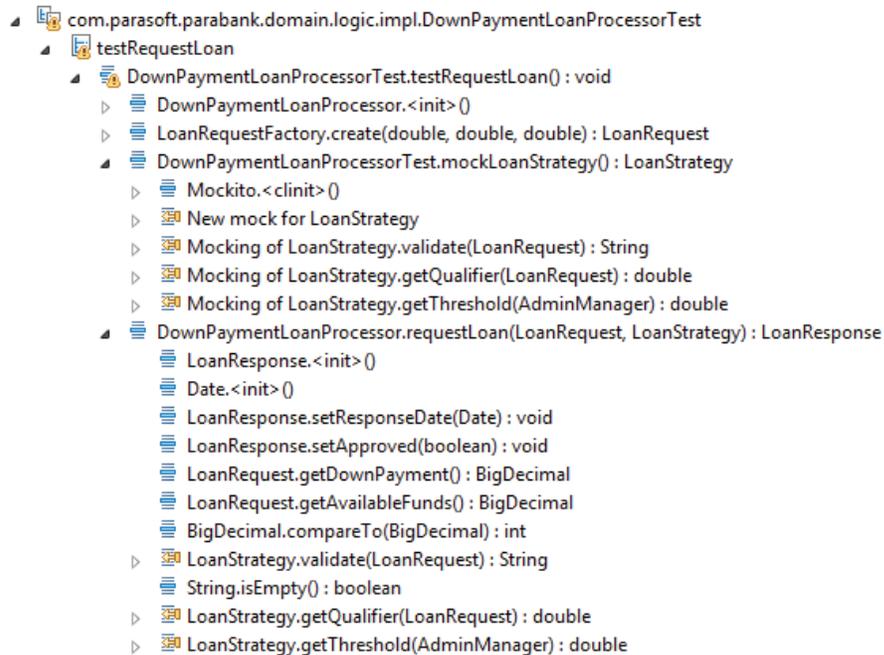
    when(strategy.getThreshold(any(AdminManager.class))).
thenReturn(getThresholdResult);
    return strategy;
}
```

We can configure `validateResult` with a non-empty value to test the use case where the method enters the new block of code, or we can use an empty value (or null) to validate behavior when the new block is not entered.

ANALYZING THE TEST FLOW

The assistant also provides some useful tools for analyzing the test flow. For instance, here is the flow tree for our test run:

Figure 9:
The Parasoft Jtest Unit
Test Assistant's Flow Tree,
showing calls made during
test execution.



SUMMARY OF MOCKING IN JAVA

You can automate many aspects of unit testing. [Parasoft Jtest](#) helps you generate unit tests with less time and effort, reducing the complexity associated with mocking. It also makes many other recommendations to improve existing tests based on runtime data, and has support for parameterized tests, Spring Application tests, and PowerMock (for mocking static methods and constructors).

HOW TO CREATE JUNIT PARAMETERIZED TESTS

Parameterized tests are a good way to define and run multiple test cases, where the only difference between them is the data. Here, we look at three different frameworks commonly used with JUnit tests.

When writing unit tests, it is common to initialize method input parameters and expected results in the test method itself. In some cases, using a small set of inputs is enough; however, there are cases in which we need to use a large set of values to verify all of the functionality in our code. Parameterized tests are a good way to define and run multiple test cases, where the only difference between them is the data. They can validate code behavior for a variety of values, including border cases. Parameterizing tests can increase code coverage and provide confidence that the code is working as expected.

There are a number of good parameterization frameworks for Java. In this article, we will look at three different frameworks commonly used with JUnit tests, with a comparison between them and examples of how the tests are structured for each. Finally, we will explore how to simplify and expedite the creation of parameterized tests.

JUNIT PARAMETERIZED TEST FRAMEWORKS

Let's compare the 3 most common frameworks: JUnit 4, JUnitParams, and JUnit 5. Each JUnit parameterization framework has its own strengths and weaknesses.

JUnit 4

Pros

- » This is the parameterization framework built into JUnit 4, so it requires no additional external dependencies.
- » It supports older versions of Java (JDK 7 and older).

Cons

- » Test classes use fields and constructors to define parameters, which make tests more verbose.
- » It requires a separate test class for each method being tested.

JUnitParams

Pros

- » Simplifies parameter syntax by allowing parameters to be passed directly to a test method.
- » Allows multiple test methods (each with their own data) per test class.
- » Supports CSV data sources, as well as annotation-based values (no method required).

Cons

- » Requires the project to be configured with the JUnitParams dependency.
- » When running and debugging tests, all tests within the class must be run – it is not possible to run a single test method within a test class.

JUnit 5

Pros

- » This parameterization framework is built into JUnit 5 and improves what was included with JUnit 4.
- » Has a simplified parameter syntax like JunitParams.
- » Supports multiple data-set source types, including CSV and annotation (no method required).
- » Even though no extra dependencies are required, more than one .jar is needed.

Consideration

- » Requires newer versions of Java and your preferred build system (e.g. Gradle or Maven Surefire). Check the JUnit 5 specifications for details.

EXAMPLE OF A JUNIT PARAMETERIZED TEST

As an example, suppose that we have a method that processes loan requests for a bank. We might write a unit test that specifies the loan request amount, down payment amount, and other values. We would then create assertions that validate the response – the loan may be approved or rejected, and the response may specify the terms of the loan.

```
public LoanResponse requestLoan(float loanAmount, float
downPayment, float availableFunds)

{
    LoanResponse response = new LoanResponse();
    response.setApproved(true);

    if (availableFunds < downPayment) {
        response.setApproved(false);
        response.setMessage("error.insufficient.funds.for.down.
payment");
    }

    return response;
}

if (downPayment / loanAmount < 0.1) {
    response.setApproved(false);
    response.setMessage("error.insufficient.down.payment");
}

return response;
}
```

[View raw.](#)

[View parameterized test example.](#)

First, let's look at a regular test for the above method:

```
@Test
public void testRequestLoan() throws Throwable
{

    // Given|
    LoanProcessor underTest = new LoanProcessor();

    // When
    LoanResponse result = underTest.requestLoan(1000f, 200f,
    250f);

    // Then
    assertNotNull(result);
    assertTrue(result.isApproved());
    assertNull(result.getMessage());
}
```

[View raw.](#)

[View parameterized test example 2.](#)

In this example, we are testing our method by requesting a \$1000 loan, with a \$200 down payment and indicating that the requestor has \$250 in available funds. The test then validates that the loan was approved and didn't provide a message in the response.

In order to make sure that our `requestLoan()` method is tested thoroughly, we need to test with a variety of down payments, requested loan amounts, and available funds. For instance, let's test a \$1 million loan request with zero down payment, which should be rejected. We could simply duplicate the existing test with different values, but since the test logic would be the same, it is more efficient to parameterize the test instead.

We will parameterize the requested loan amount, down payment, and available funds, as well as the expected results: whether the loan was approved, and the message returned after validation. Each set of request data, along with its expected results, will become its own test case.

AN EXAMPLE OF A PARAMETERIZED TEST USING JUNIT 4 PARAMETERIZED

Let's start with a JUnit 4 Parameterized example. To create a parameterized test, we first need to define the variables for the test. We also need to include a constructor to initialize them:

```
@RunWith(Parameterized.class)
public class LoanProcessorParameterizedTest {

    float loanAmount;
    float downPayment;
    float availableFunds;
    boolean expectApproved;
    String expectedMessage;

    public LoanProcessorParameterizedTest(float loanAmount, float
downPayment,

    float availableFunds, boolean expectApproved, String
expectedMessage)

    {
        this.loanAmount = loanAmount;
        this.downPayment = downPayment;
        this.availableFunds = availableFunds;
        this.expectApproved = expectApproved;
        this.expectedMessage = expectedMessage;
    }

    // ...

}
```

[View raw.](#)

[View parameterized test example 3.](#)

Here, we see that the test uses the `@RunWith` annotation to specify that the test will run with the JUnit 4 Parameterized runner. This runner knows to look for a method which will provide the value-set for the test (annotated with `@Parameters`), initialize the test properly, and run the tests with multiple rows.

Note that each parameter is defined as a field in the test class, and the constructor initializes these values (you can also inject values into fields using the `@Parameter` annotation if you don't want to create a constructor). For each row in the value-set, the Parameterized runner will instantiate the test class and run each test in the class.

Let's add a method which provides the parameters to the Parameterized runner:

```
@Parameters(name = "Run {index}: loanAmount={0},  
downPayment={1}, availableFunds={2}, expectApproved={3},  
expectedMessage={4}")  
  
public static Iterable<Object[]> data() throws Throwable  
  
{  
    return Arrays.asList(new Object[][] {  
  
        { 1000.0f, 200.0f, 250.0f, true, null }  
    });  
}
```

[View raw.](#)

[View parameterized test example 4.](#)

The value-sets are built as a *List of Object* arrays by the *data()* method, which is annotated with `@Parameters`. Note that `@Parameters` sets the name of the test using placeholders, which will be replaced when the test runs. This makes it easier to see values in test results, as we will see later. Currently, there is only one row of data, testing a case where the loan should be approved. We can add more rows to increase coverage of the method under test.

```
@Parameters(name = "Run {index}: loanAmount={0},  
downPayment={1}, availableFunds={2}, expectApproved={3},  
expectedMessage={4}")  
  
public static Iterable<Object[]> data() throws Throwable  
  
{  
    return Arrays.asList(new Object[][] {  
  
        { 1000.0f, 200.0f, 250.0f, true, null },  
  
        { 1000.0f, 50.0f, 250.0f, false, "error.insufficient.down.  
payment" },  
  
        { 1000.0f, 200.0f, 150.0f, false, "error.insufficient.funds.for.  
down.payment" }  
    });  
}
```

[View raw.](#)

[View parameterized test example 5.](#)

Here, we have one test case where the loan would be approved, and two cases in which it should not be approved for different reasons. We may want to add rows in which zero or negative values are used, as well as test boundary conditions.

We are now ready to create the test method:

```
@Test

public void testRequestLoan() throws Throwable

{
    // Given
    LoanProcessor underTest = new LoanProcessor();

    // When
    LoanResponse result = underTest.requestLoan(loanAmount,
    downPayment, availableFunds);

    // Then
    assertNotNull(result);
    assertEquals(expectApproved, result.isApproved());
    assertEquals(expectedMessage, result.getMessage());
}
```

[View raw.](#)

[Parameterized test example 6.](#)

Here, we reference the fields when invoking the `requestLoan()` method and validating the results.

JUNITPARAMS EXAMPLE

The JUnitParams library simplifies parameterized test syntax by allowing parameters to be passed directly to the test method. The parameter values are provided by a separate method whose name is referenced in the `@Parameters` annotation.

```
@RunWith(JUnitParamsRunner.class)
public class LoanProcessorParameterizedTest2 {

    @Test

    @Parameters(method = "testRequestLoan_Parameters")
    public void testRequestLoan(float loanAmount, float
    downPayment, float availableFunds,
    boolean expectApproved, String expectedMessage) throws
    Throwable
    {
        ...
    }
}
```

```
@SuppressWarnings("unused")
private static Object[][] testRequestLoan_Parameters() throws
Throwable {

// Parameters: loanAmount={0}, downPayment={1},
availableFunds={2}, expectApproved={3}, expectedMessage={4}

return new Object[][] {

{ 1000.0f, 200.0f, 250.0f, true, null },
{ 1000.0f, 50.0f, 250.0f, false, "error.insufficient.down.
payment"},
{ 1000.0f, 200.0f, 150.0f, false, "error.insufficient.funds.for.
down.payment" }
};
}
}
```

[View raw.](#)

[View parameterized test example 7.](#)

JUnitParams has the additional benefit that it supports using CSV files to provide values in addition to providing the values in code. This allows the test to be decoupled from the data and data values to be updated without updating the code.

JUNIT 5 EXAMPLE

JUnit 5 addresses some of the limitations and shortcomings of JUnit 4. Like JUnitParams, JUnit 5 also simplifies the syntax of parameterized tests. The most important changes in syntax are:

- » The test method is annotated with `@ParameterizedTest` instead of `@Test`.
- » The test method accepts parameters directly, instead of using fields and a constructor.
- » The `@RunWith` annotation is no longer needed.

Defining the same example in JUnit 5 would look like this:

```
public class LoanProcessorParameterizedTest {

@ParameterizedTest(name="Run {index}: loanAmount={0},
downPayment={1}, availableFunds={2}, expectApproved={3},
expectedMessage={4}")

@MethodSource("testRequestLoan_Parameters")
public void testRequestLoan(float loanAmount, float
downPayment, float availableFunds,

boolean expectApproved, String expectedMessage) throws
Throwable
```

```
{
...
}

static Stream<Arguments> testRequestLoan _ Parameters() throws
Throwable {

return Stream.of(

Arguments.of(1000.0f, 200.0f, 250.0f, true, null),

Arguments.of(1000.0f, 50.0f, 250.0f, false, "error.insufficient.
down.payment"),

Arguments.of(1000.0f, 200.0f, 150.0f, false, "error.insufficient.
funds.for.down.payment")
);
}
}
```

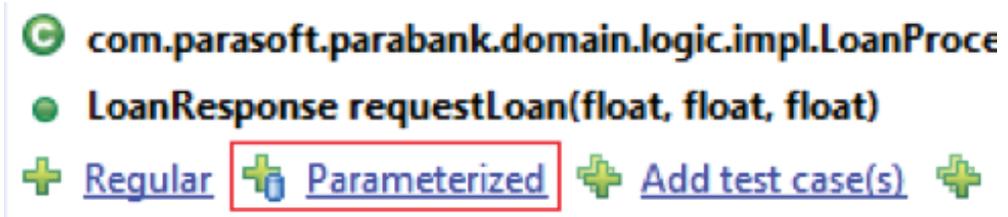
[View raw.](#)

[View parameterized test example 8.](#)

EFFICIENTLY CREATE PARAMETERIZED TESTS

As one might imagine, writing the above parameterized test can be a bit of work. For each parameterized test framework there is some boilerplate code that needs to be written correctly. It can be hard to remember the correct structure, and it takes time to write out. To make this much easier, you can use [Parasoft Jtest](#) to generate parameterized tests, automatically, like the ones described above. To do this, simply select the method you want to generate a test for (in Eclipse or IntelliJ):

Figure 10:
Select Parameterized
Test Generation in
Parasoft Jtest

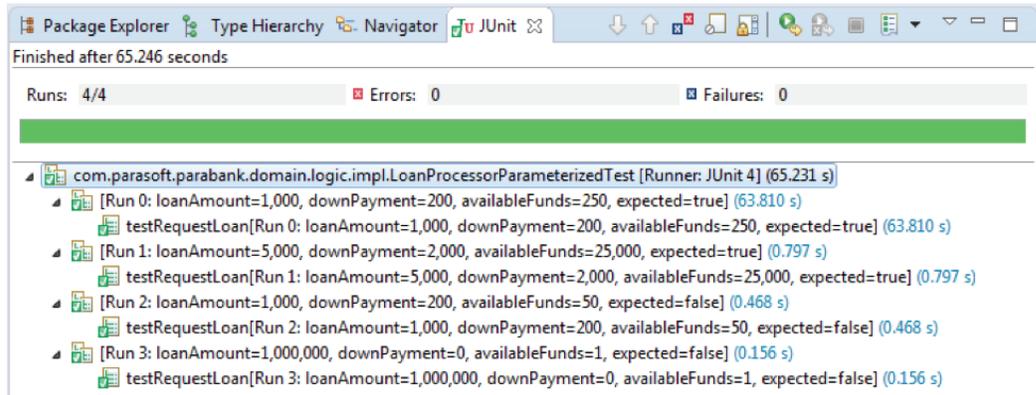


The test is generated, using default values and assertions. You can then configure the test with real input values and assertions and add more data rows to the `data()` method.

RUNNING THE PARAMETERIZED TEST

Parasoft Jtest can run parameterized tests directly in both Eclipse and IntelliJ.

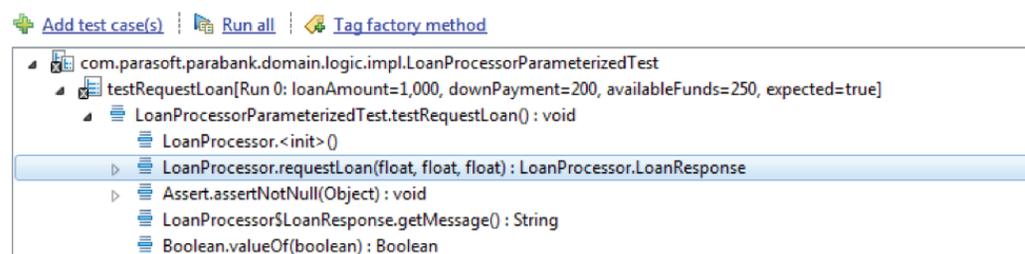
Figure 11:
The JUnit View in Eclipse



Note that the name of each test, as shown, includes input values from the dataset and expected result values. This can make debugging the test much easier when it fails, since the input parameters and expected outputs are shown for each case.

You can also use the Run All action from Parasoft Jtest:

Figure 12:
The Flow Tree View
in Parasoft Jtest



It analyzes the test flow and provides detailed information about the previous test run. This allows you to see what happened in the test without needing to rerun the test with breakpoints or debugging statements. For instance, you can see parameterized values in the Variables view:

Figure 13:
The Variables View
in Parasoft Jtest

Field	Value	Old Value
▲ This		
LoanProcessor	id = 2	
▲ Outcome		
LoanProcessor.LoanResponse	id = 3	
▲ Arguments		
float	1000.0	
float	200.0	
float	250.0	

SUMMARY OF CREATING JUNIT PARAMETERIZED TESTS

Each of the three frameworks that we reviewed are fine choices and work well. If using JUnit 4, JunitParams is preferred over the built-in JUnit 4 Parameterized framework, due to the cleaner design of the test classes and the ability to define multiple test methods in the same class. However, if using JUnit 5, we recommend the built-in JUnit 5 framework since it addresses the shortcomings in JUnit 4 and requires no extra libraries. We also like using Parasoft Jtest's unit testing capabilities to make creation, execution, and debugging of parameterized tests more efficient.

GET MORE OUT OF UNIT TESTING AND REDUCE MAINTENANCE EFFORTS WITH RUNTIME ANALYSIS

To realize the benefits of unit testing, you can observe a unit test during its execution via runtime analysis. Runtime analysis during unit test execution is critical to improving test efficiency and effectiveness.

Unit testing is a best practice to test individual units/components of a software, but it can be tedious and costly for Java developers. It's painstaking to test each unit for correct behavior with manual assertions, and isolate each method with mocking, and unit testing itself is open to bugs and misunderstood behavior. To improve this situation, you can use a runtime analysis tool to detect data and control flow, external dependencies, and to calculate test code coverage.

With this collected data from the runtime analysis, an enterprise-grade solution like [Parasoft Jtest](#) can prompt the developer about how to improve the tests, by automatically recommending assertions for correct behavior, and methods for mocking to improve test isolation. This integration between automatic unit test generation and runtime analysis reduces the manual intervention required for unit testing for Java.

BENEFITS OF UNIT TESTING

Unit testing is a well-known practice, but its implementation requires improvement in many projects. Unit testing, done well, improves the agility of agile process, increases quality and security, and brings long-term cost savings.

Unfortunately, regardless of these benefits, developers can still struggle with unit testing, despite the desire to achieve better results. The amount of time and effort needed for test creation and maintenance can be too much to justify increasing testing efforts. Often, test suites are fragile because of poor unit/object isolation from dependencies. Proper mocking of dependencies becomes the bane of software testers, as does creating the assertions needed to determine correct program logic. Even parameterizing tests for scenarios can be tedious and time consuming.

Software development teams must address these problems with test creation, isolation, and maintenance if they want to achieve the benefits of thorough unit testing. The answer starts with test automation tools, but simply automating the execution of tests and collecting results isn't enough. Runtime analysis, the process of observing a running executable and recording key metrics, is an innovative way to help improve unit testing creation, mocking, and test stability.

RUNTIME ANALYSIS CAN IMPROVE UNIT TESTING

In most cases, developers don't consider runtime analysis important in early stages of unit testing. Most tools are used for catching errors that unit testing missed, or simply in calculating code coverage. But while these benefits are important, runtime analysis can also observe the execution of the first iteration of a unit test to make recommendations to improve the test and detect changes to the test runtime environment that interfere with test stability.

Test frameworks such as JUnit create sparse code that requires further developer input. This work is tedious, so it can be automated to fill in more of the details based on the observed program logic. For example, the following unit test can be automatically generated by Parasoft Jtest:

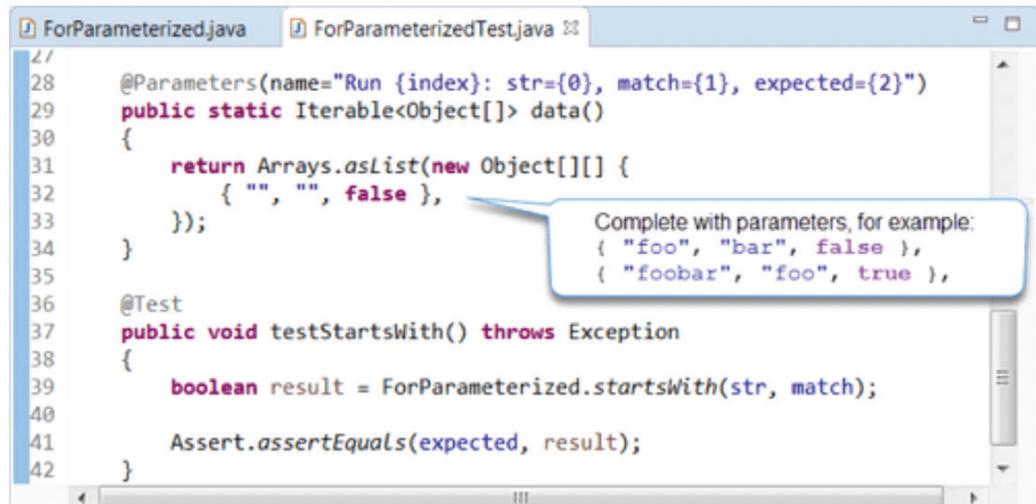
Figure 14:
Unit Test Generation
in Parasoft Jtest



```
13  @Test
14  public void testGetLabel() throws Exception
15  {
16      // Given
17      int value = 0; // UTA: default value
18      InputStream is = null; // UTA: default value
19      ForObject underTest = new ForObject(value, is);
20
21      // When
22      String sTitle = ""; // UTA: default value
23      String result = underTest.getLabel(sTitle);
24
25      // Then
26      // Assert.assertNotNull(result);
27      // Assert.assertEquals("", result);
28  }
```

Similarly, for unit tests with parameterized inputs, shown below:

Figure 15: Parameterized Test Generation in Parasoft Jtest

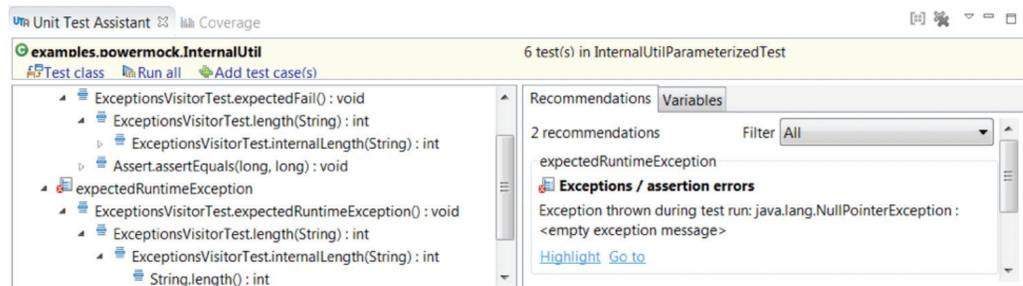


```
27
28 @Parameters(name="Run {index}: str={0}, match={1}, expected={2}")
29 public static Iterable<Object[]> data()
30 {
31     return Arrays.asList(new Object[][] {
32         { "", "", false },
33     });
34 }
35
36 @Test
37 public void testStartsWith() throws Exception
38 {
39     boolean result = ForParameterized.startsWith(str, match);
40
41     Assert.assertEquals(expected, result);
42 }
```

Complete with parameters, for example:
{ "foo", "bar", false },
{ "foobar", "foo", true },

Since the created tests are executable from the start, they can be observed by runtime analysis for both results and execution flow. For example, a test may fail due to a raised exception, shown below.

Figure 16: Exception Error Shown in Parasoft Jtest Unit Test Assistant

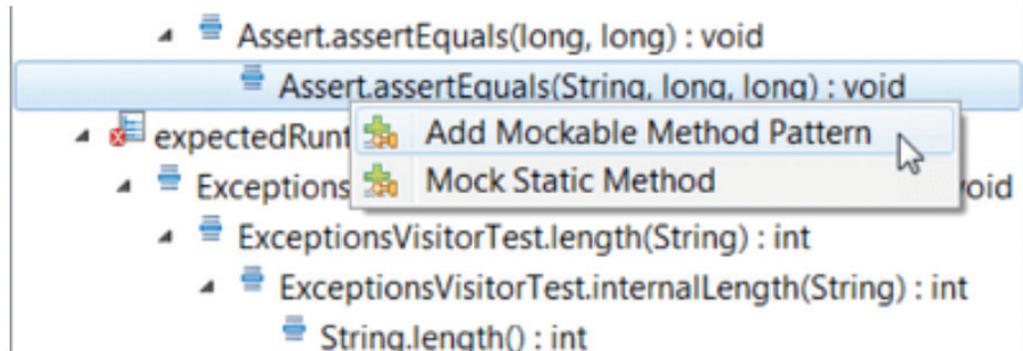


DETECTING DEPENDENCIES AND MOCKING WITH RUNTIME ANALYSIS

In addition, runtime tools observe the execution path into dependencies and recommend potential mocks to increase the isolation of the test. Although visual inspection of an object under test will reveal its dependencies, automating the detection and mocking of these dependencies saves lots of tedious and error-prone work.

In the example below, Parasoft Jtest offers the developer a choice of what to mock based on the execution trace of the unit test:

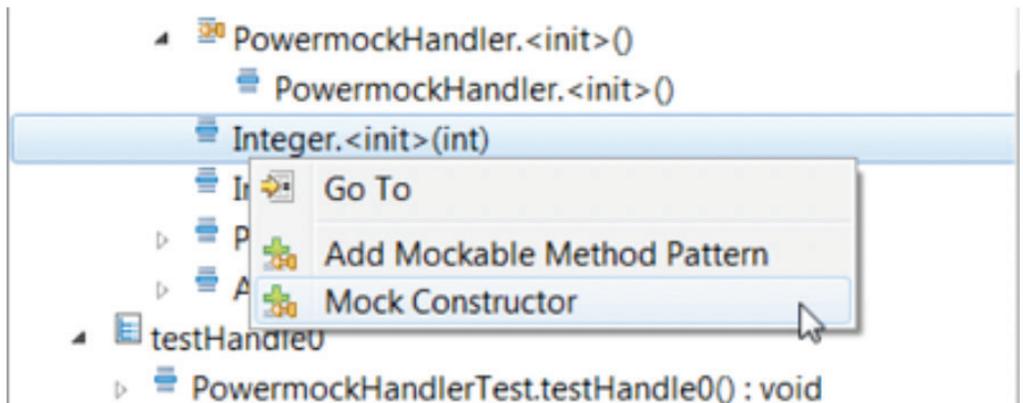
Figure 17:
Execution Trace
of Unit Test



In this case, adding a mockable method pattern adds the method to a list of mocks to be handled by a mocking framework such as PowerMock.

Mocking static constructors are also possible, as shown below.

Figure 18:
Example for Mocking
Constructors

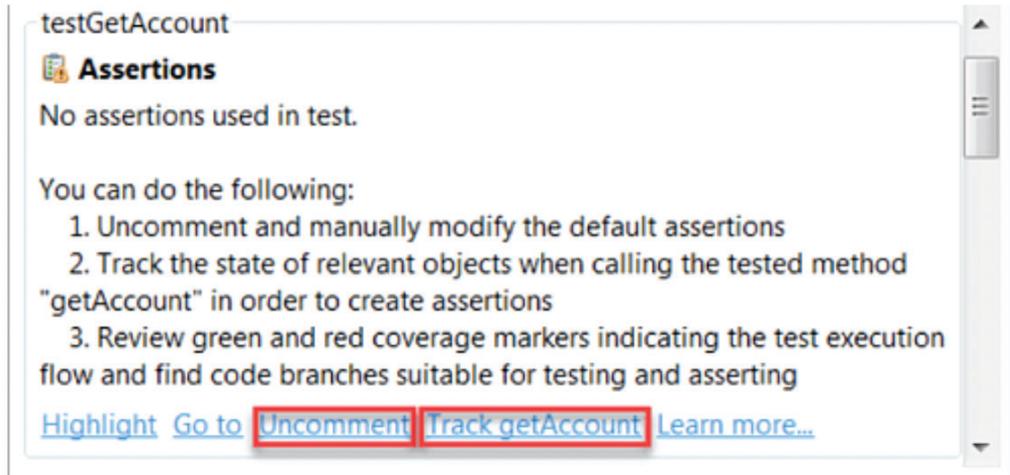


IMPROVING TESTING FIDELITY WITH RUNTIME ANALYSIS

With full knowledge of the execution flow, plus parameters used in method calls, runtime analysis can be used to provide useful recommendations to the developer to improve the test code. Although assertions are provided, statically, when a test is created, they may not be enabled or correct. At test execution, failed and missing assertions trigger warnings which then lead to recommendations to remedy the problem.

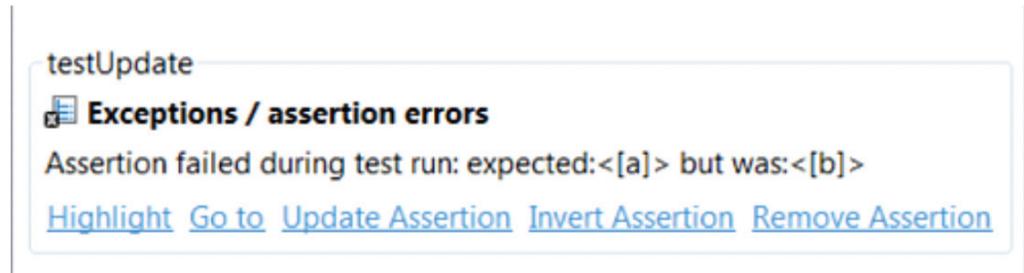
For example, after creating a new test, and no recommended assertions have been uncommented, you would see the following:

Figure 19:
Example of Assertion
Recommendations



Or if an assertion fails, for example, the following is displayed:

Figure 20:
Example of Assertion
Failure



Whatever happens, it is the constant feedback about corrective action for assertions that closes the loop on test creation to complete unit testing. Additionally, as the unit under test is changed, these changes can be dealt with in the same manner, continually reducing the manual test maintenance required.

IMPROVING TEST STABILITY WITH RUNTIME ANALYSIS

Runtime analysis can also detect changes in the test environment during execution that impact the ability to recreate an identical test environment for subsequent tests. Tests that pass at one time and fail later can be a cause of great frustration and lost time and effort. Some examples of instabilities that you can detect with runtime analysis include the following:

- » **A changed system property** during a test that hasn't changed back to its original state. A subsequent test may rely on this property.
- » **Additional execution threads** in the background that may interfere with a test run.
- » **A new file creation** during test execution that might impact subsequent runs if they rely on the file and its contents.
- » **Modified static fields** that might impact future tests that use these same fields.

It's critical that each test execution has an identical starting point, to ensure reliable results. Preventing test instability with runtime detection removes guesswork from the test debug phase.

CONCLUSION

You can see that runtime analysis isn't just for computing code coverage. Runtime analysis during test execution is critical to improving test efficiency and effectiveness. Monitoring execution paths provides information about dependencies to improve the handling of dependencies and mocking. Assertions can be monitored, and automatic recommendations improve test fidelity. Detecting changes in the runtime test environment that affect test stability removes frustration and reduces debugging cycles for test code.

TAKE THE NEXT STEP

Learn how [Parasoft Jtest](#) can help you improve your Java code quality and team productivity. [Contact us today.](#)

ABOUT PARASOFT

Parasoft helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award winning reporting and analytics dashboard delivers a centralized view of quality enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives—cybersecure, safety-critical, agile, DevOps, and continuous testing.