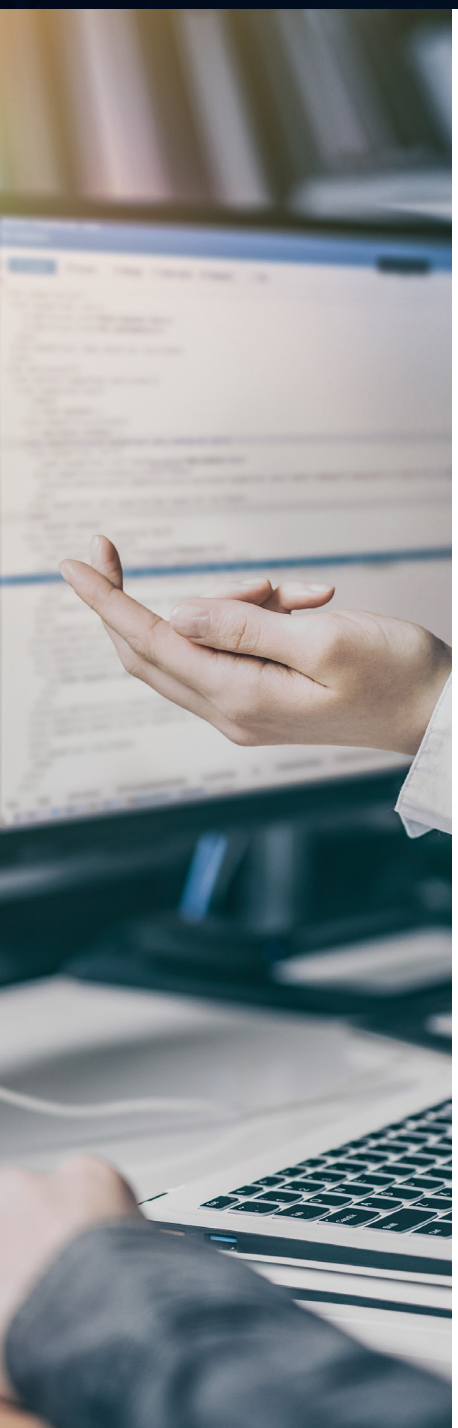
Get Unit Testing Done Right

Top Tips for Java Developers

INTRODUCTION

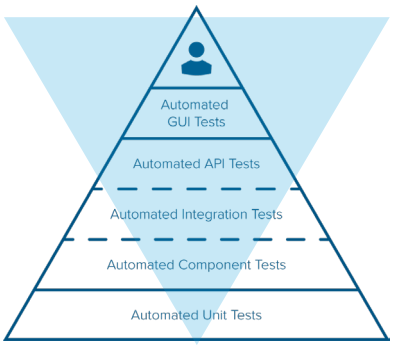
Today's systems are larger and more complex than ever, while release cycles are becoming shorter and more frequent. Because of these trends, unit testing is an invaluable practice you cannot risk skipping. But many organizations either fail to adequately allocate resources for testing or forego unit testing altogether. In this paper, we will discuss some of the challenges associated with unit testing and offer some best practices to help you overcome them.



THEORY AND PRACTICE

Unit testing is generally considered a good practice with plenty of benefits:

- Testing validates your code and functionality, providing a solid foundation for future development and extending test scenarios.
- Defects can be identified at early stages of the production process, which limits the costs of fixing them in later stages of the development cycle.
- Unit tested code is safer to refactor and more predictable.
- Developers are forced to consider how well the production code is designed in order to make it suitable for unit testing.
- Writing unit tests makes developers look at their code from a different perspective and helps them understand the production code better.
- Unit tests are an additional medium for collecting feedback about the production code and how it should behave.



It's clear that unit testing can help you make your system more reliable and robust, but what is the best way to test? The "Testing Pyramid" is a commonly-used model to describe the ideal distribution of testing resources.

Automated unit tests, which provide a solid foundation for any other test types, should make up the bulk of your tests. **Component, integration,** and **functional tests** should be used a little more sparingly, while **manual** tests should constitute a minimal percentage of the overall pyramid structure.

That's the theory. In practice, however, development teams often resort to an inverted pyramid in which manual testing is the primary testing practice followed before each release. Manual tests,

PARASOFT®
Automated Software Testing

though, are expensive in terms of time and human resources. If implemented at all, unit testing is perceived as a necessary evil that should be kept to a minimum.

WEIGHING THE COSTS AND BENEFITS OF UNIT TESTING

A significant reason for the discrepancy between theory and practice, despite the numerous advantages of unit testing, are the costs:

- Creating unit tests requires extra time and resources. From the developer perspective, creating unit tests involves writing additional code that does not deliver functionality.
- Changes in production code also require changes and verification of existing unit test code, adding overhead during development.
- To be effective, unit tests should be continuously executed and monitored, and all failures must be reviewed.
- Tests that fail need to be fixed, which requires an understanding of why they failed.

If unit testing adds these additional expenses, how can it be worth the cost? Can relying on static analysis or manual testing to catch all the errors be sufficient? These are some of the fundamental questions the organizations with which we have worked over the years have had about implementing unit testing. For organizations that are serious about improving software quality and team efficiency over the long haul of the development lifecycle, unit testing is absolutely worth the investment. It is not effortless or magical, though. Realizing the benefits of unit testing requires planning, a commitment to enforcing development policies, and dedication to long-term goals.

As with any other kind of investment, a cost-benefit analysis is key. The costs associated with unit testing can be discouraging, but these costs must be weighed against the cost of software failure. If you only consider the extra time and effort that it takes to write and maintain tests, you might conclude that the costs of unit testing are greater than the benefits. But factor in the cost and risk of a security breach or safety-critical software failure, and unit testing becomes a much wiser investment.

Once you determine that the safety, security, and reliability of your applications is worth the cost of testing, you'll need to start off on the right foot. In the next section, we'll discuss best practices and methods for making your unit testing practice sustainable and efficient.

WRITING CLEAN AND MAINTAINABLE TESTS

There are plenty of well-established unit testing frameworks that can assist you with unit testing from the moment you get started. They range from basic **JUnit** libraries to mocking and stubbing frameworks, such as **Mockito, JMockit** or **PowerMock.** There are also a number of additional utilities, such as **JUnitParams** or **Hamcrest,** for parameterizing unit tests or creating assertions. These useful toolkits alone, however, do not drive success. Setting up useful and maintainable tests is the key. Here are some tips on how to do it.

Create Focused Tests and Keep Them Organized

Creating your unit tests at the right point in the process, focusing on the goals for the tests, and organizing them appropriately is one of the keys to making the unit testing process sustainable.

Write your tests as you write your application code. It's easier to write meaningful tests that correctly validate the code functionality when the code is fresh in your mind. In addition, writing tests at the same time as code helps you to validate that the code is correct. Understanding production code becomes harder when tests are created later or by a different team, which increases the overall cost of creating the tests. Many teams have experienced this with legacy code; they have found that adding tests later can be very costly.

Write testable code. You may need to refactor the production code to make it more testable if you find that writing a test for your code is difficult or very complex. Your production code should be free of artificial "access points" created only for the sake of testing. The tests should fit into the production design.

Focus tests on functional use cases that are realized by production code. As a rule, primarily use the public API of tested units to build test scenarios (rather than testing protected/private methods).

Organize your code for clarity and readability. This is a best practice for production code, but it also applies to unit test code. One way to achieve this is to apply "Given-When-Then" organization, which helps you state the purpose, conditions, and expectations around a unit of code.

```
public void testAmount() {
    // Given
    Money underTest = new Money();
    // When
    int result = underTest.amount();
    // Then
    assertEquals(0, result);
}
```

Reduce the amount of boilerplate code. For example, the use of static imports for Assert, Mockito and other similar classes can make code more compact and easier to read.

Parameterize your tests when possible. Parameterization can help you test many different use cases, while keeping the test code size maintainable. Here is an example of a parameterized test that verifies different inputs using a single test method:

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

    @Test
    @Parameters(method = "testMultiply_Parameters")
    public void testMultiply(int factor, int expected) {
        Account underTest = new Account(5);
        Account result = underTest.multiply(factor);
        assertEquals(expected, result.getBalance());
    }

    @SuppressWarnings("unused")
    private static Object[] testMultiply_Parameters() {
        // Parameters: factor={0}, expected={1}
        return new Object[] {
            new Object[] { 0, 0 },
            new Object[] { 1, 5 },
            new Object[] { 2, 10 }
        };
    }
}
```

Write tests so failures are easy to understand - this will help you (and others) to fix them faster.

- Give test classes and test methods descriptive names

- Include comments that reference requirements or issues in your bug tracking system. This will help people who review the tests in the future to understand the purpose of a given test case by being able to review the related requirement or bug.
- All failure messages from assertion checks should have enough detail to understand the failure when viewed in a report.

Write readable, clear, and self-descriptive assertions that explain the expected outcome. Only write assertions for elements of tested objects that should be affected or should stay unchanged for the given use case. You don't need to verify every field. This will keep the purpose of the test case clear and ensure that it is more stable over time.

Reuse well-written existing tests. When adding or modifying functionality for existing code, copy and/or refactor existing tests and modify them to cover additional use cases.

Write tests for negative scenarios, such as unexpected situations or exceptions. Mock objects and stubs can help you trigger negative conditions that would otherwise be hard to test.

Review code coverage information. Code coverage is not a measure of test quality, or a good goal in itself, but it can provide useful feedback about which code has not been tested. Use code coverage metrics to identify where additional tests need to be added to handle expected use cases.

Use change-based testing. Introduce change-based testing to allow you to rerun only the test cases that are directly related to your most recent source code modifications. This allows developers to easily identify which tests to run before committing code, as well as limit the number of tests that need to be run. While the best practice is to run all unit tests for every build, teams with thousands of unit tests may benefit from using change-based testing as part of CI processes to decrease the time it takes to run the test suite, in order to get faster feedback on whether the changes are safe.

Keep Tests Stable, Maintainable and Optimized

Run your tests locally before checking them into source control and adding them to a continuous integration system. This may seem obvious but is important to ensure that new tests won't immediately fail and break the build due to not being fully validated.

Monitor the results of the entire test suite daily. Make a habit to review failures, execution times, and coverage information.

- **Monitor your tests for regressions.** Issues should be fixed as soon as they are found. Teams should maintain a culture of intolerance of test failures.
- **Monitor execution times.** If you notice an increase in execution time, first review your code to see if performance regressions were introduced. However, the simple act of adding new test cases causes test suites to take longer to run. Over time you may need to review your test suite and identify tests to exclude to keep your test execution time reasonable.
- **Use code coverage results** to identify gaps in testing.

Identify and fix unstable tests. A classic problem in test automation is unstable tests that fail periodically due to environmental factors but not product issues. It can be incredibly difficult to track down and fix the causes of the instability. Unstable tests of this nature should either be fixed or removed from the test suite. Teams often want to hold on to unstable tests since they cover a use case, even though they can't find time to fix them. This leads to decreasing the value of the overall test suite since the teams start ignoring the test results due to the instability – which can result in test failures that expose release regressions being ignored. Often the most effective solution is to simply remove the tests.

Keep your tests in sync with the evolving application. Refactor the tests when you refactor the code and ensure that existing tests continue to pass.

Perform static analysis on unit test code to eradicate common issues and identify "test code smells".

Evolve existing test suites to improve maintainability. Apply the various best practices discussed in this paper to existing tests that did not originally follow those practices.

Keep tests atomic. The scope for an individual unit test should be focused on an individual code unit (method, class, etc.). Don't try to test too much at one time. Tests that have complex setup and verify a lot of code will be hard to maintain and keep stable. It is easier to create and understand 10 simple tests for 10 cases than 1 huge test for all cases.

Avoid writing tests for the sake of writing tests. Since additional tests increase the cost of maintenance, unit tests should only be added when appropriate. There are some cases where it does not make sense to write a unit test because another kind of test would be more cost effective.

Test in Isolation

Execution of a unit test should not affect other tests. For example, a unit test that changes static variables, system properties, or the runtime environment can affect tests that run after it that depend on those same global properties. These kinds of changes can appear either in the code being tested or in the unit test itself. If possible, modify your test to leave the global properties unchanged. In many cases, however, the change is unavoidable; in those cases you need to ensure that your environment is reverted to the initial state after the test runs. This is usually done by adding cleanup code in a finally block within the unit test or in a teardown method in the unit test class. Other specific cases to consider cleaning up are files that were created, database changes that were made, or separate processes that got started.

Create tests that succeed independently of the order that they run. Otherwise, when the order of executed tests changes (which is common in JUnit for example), it will be very difficult to determine why tests are failing if they affect each other.

Use mocks and subs to isolate a test from its dependencies. Isolation makes it quicker to identify the source of failure since a unit test depends on fewer classes. In addition, it can make it possible to write unit tests for code that would otherwise be difficult or impossible to unit test. Here are a few tips how to do it well:

- Use mock objects instead of instantiating nested complex objects for values that are passed to the method under test. Define the expected behavior of those mock objects for only the methods that will be accessed as the test is run. For example, instead of initializing a temporary zip file in order to be able to pass it to a method:

```
File tempFile = File.createTempFile("data", ".zip");
tempFile.deleteOnExit();
ZipOutputStream out = null;
try {
    out = new ZipOutputStream(new FileOutputStream(tempFile.getAbsolute
        Path()));
    out.putNextEntry(new ZipEntry("data.txt"));
    String data = "example data file content";
    out.write(data.getBytes(StandardCharsets.UTF_8), 0, data.length());
} finally {
    if (out != null)
    out.close();
}
ZipFile zipFile = new ZipFile(tempFile);
```

Create a mock object and defined the behavior only for the expected interactions:

```
ZipFile zipFile = Mockito.mock(ZipFile.class);
InputStream is = new ByteArrayInputStream("example data file
    content".getBytes(StandardCharsets.UTF_8));
when(zipFile.getEntry("data.txt")).thenReturn(new ZipEntry("data.txt"));
when(zipFile.getInputStream(any(ZipEntry.class))).thenReturn(is);
```

- Add assertions that verify that the expected methods get called on your mock objects and that no unexpected methods get called. For example, one way to do this is to use the Mockito verify() method to ensure that particular methods did or did not get called as the unit test executed:

```
// Given
Bank underTest = new Bank();
Account account = Mockito.mock(Account.class);
when(account.getCustomerId()).thenReturn("customer_1");
underTest.addAccount(account);

// When
underTest.closeAccount(account);

// Then
assertNull(underTest.getAccount("customer_1"));
verify(account, never()).setBalance(anyInt());
verify(account, times(2)).getCustomerId();
```

- Some code is hard to unit test because it makes static method calls or instantiates objects that depend on the environment being initialized in a certain way. Stubbing these static methods or constructors using a framework like PowerMockito can make it possible to easily control the behavior of those static calls or created objects and can enable you to write tests for code that would otherwise be impossible or very difficult to unit test. For example, consider a class that sets a system property to some unknown value:

```
public class SystemInjection {
    .....
    private void environmentInjection() throws Exception {
         String sSomeValue = … // value which will modify system environment
         System.setProperty("someProperty", sSomeValue);
    }
}
```

When writing a unit test for that class, you can control the behavior of setProperty() to ensure that the property is set to something that your unit test controls:

```
@PrepareForTest({ SystemInjection.class})
@RunWith(PowerMockRunner.class)
public class SystemInjectionTest {
    @Test
    public void testSystemInjection() throws Throwable {
        // Given
        spy(System.class);
        String setPropertyResult = "";
        doReturn(setPropertyResult).when(System.class, "setProperty",
        anyString(), anyString());

        SystemInjection underTest = new SystemInjection();
        ....
    }
}
```

- Frameworks like PowerMockito can enable you to stub final or private method calls for objects under test if they make unwanted calls to objects outside the tested class.
- Some stubbing frameworks, such as those that use bytecode instrumentation and specialized runners (like PowerMock), are heavy and can affect test stability. Use them only when necessary. However, in some cases using them could be the only option to set up stable tests for some code.

Develop Experience in Writing Unit Tests

It may seem obvious, but the ability to write good tests increases as you gain more knowledge and experience. Staying current with trends in software development technologies and approaches can help you create better tests and improve your application quality.

- Experiment with several frameworks and libraries. This will enable you to choose the best set for each use case.
- Read books and other publications on unit testing to extend your knowledge. Encourage your team to do the same.
- Continuously improve your coding skills. There is always more to learn about general coding best practices. General coding best practices around reducing code complexity and improving quality, maintainability, performance, and documentation are also applicable to unit testing.

JTEST UNIT TEST ASSISTANT

The best practices we've discussed thus far will lead to more sustainable unit testing practices and ultimately improve code quality, and in some cases they will reduce the cost of writing and maintaining unit tests. However, you will still likely spend a significant amount of time evolving your unit test suite. While this can be time-consuming even for an experienced developer, it can be especially so for those who are new to unit testing.

Parasoft's Unit Test Assistant is a powerful unit testing solution specifically designed to reduce the time and effort of creating meaningful, sustainable unit tests. The Unit Test Assistant guides you through the process of creating, assessing, and improving unit tests. As a result, it helps you:

- Accelerate unit test adoption and extend your knowledge about different aspects of unit testing
- Easily create functional templates of regular and parameterized test cases
- Generate functional test cases with required dependencies already set up
- Review the test execution flow and coverage information for individual tests
- Review total code coverage and identify coverage gaps
- Identify changes that happen to object state as the unit test executes, and set up assertions that are focused on those changes
- Instantiate or mock required dependences
- Monitor the interaction with mock objects to identify additional behavior that needs to be defined
- Isolate test case dependences by injecting stubs for static methods and constructors
- Review object states on test execution paths without needing to run with debugging
- Identify tests that modify the environment in ways that can cause test instability
- Create tests in bulk for legacy code without sufficient tests
- And much more

SUMMARY

If you are running a complex long-term software project, building your testing pyramid on a solid foundation of unit tests is the only way to ensure that it is stable, maintainable, and free of regressions. Unit testing comes with costs you need to consider before deciding how to implement unit testing into your project, especially:

- The cost of learning unit testing best practices and testing frameworks
- The cost of establishing a consistent practice of writing tests
- The cost of executing, refactoring, redesigning, and maintaining test suites

Accepting the costs is a long-term decision that requires full commitment. Unit testing can dramatically improve your code quality, but only if you take it seriously and make it an integral part of your software project. Following the guidelines discussed in this paper can help you efficiently and effectively maximize the ROI of unit testing.

www.parasoft.com

Parasoft Headquarters:
+1-626-256-3680

Parasoft EMEA:
+31-70-3922000

Parasoft APAC:
+65-6338-3628

PARASOFT®
Automated Software Testing