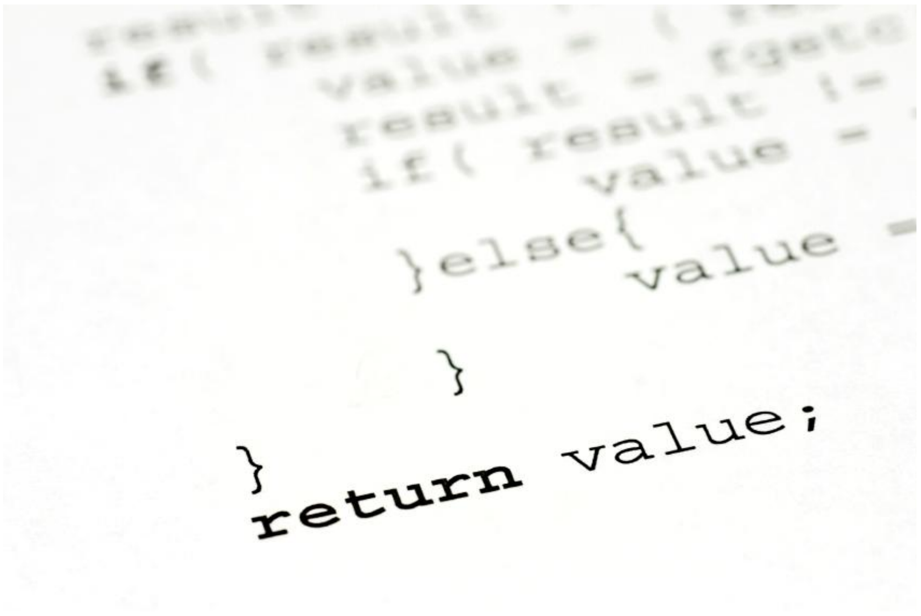
Java CoP

Pragmatic Testing

A Guide to Practical Java Unit Testing

Presented by the Java Community of Practice



Abstract

Pragmatic Java unit testing is a method to systematically test code as it is developed and maintained throughout the lifecycle of a Java software development project to reduce costs, improve quality, and decrease delivery time.

Waiting until the end of the development cycle is the least effective way to uncover defects. The later a defect is identified in the Software Development Life Cycle (SDLC), the more costly it is to fix. The higher cost is due to the time and resources to revisit the code, identify and fix the defect, update the associated requirements and documentation, and re-test the fix (e.g., integration test, system test). There is also the risk of introducing a new defect while fixing another, known as defect injection. Introducing smaller quality steps throughout the lifecycle of the project will reduce costs, improve quality, and decrease delivery time. Pragmatic testing is a way to build that quality into Java custom development projects from the beginning.

Unit Testing Background

There are multiple interpretations of unit testing. Each project team is responsible for establishing a common definition and approach to unit testing.

Definition of Unit Testing

The Common Body of Knowledge (CBOK) for the Certified Software Tester program defines unit testing as:

Testing individual programs, modules, or components to demonstrate that the work package executes per specification, and validate the design and technical quality of the application. The focus is on ensuring that the detailed logic within the component is accurate and reliable according to pre-determined specifications. Testing stubs or drivers may be used to simulate behavior of interfacing modules.[1]

In simpler terms, the purpose of unit testing is to test each individual Java function enough to produce mostly defect free code at the lowest possible level. Testing at the individual class and method level is much more effective in exercising the multiple logic paths of a software system. As you move closer to the user or system interface and further from the core code, other logic limits the logic paths. At a point in time, the overall system may meet the testing objectives, a simple change to the user interface or system interface can uncover incomplete testing of logic paths.

Common Unit Testing Approaches

On many projects, the developer who creates or modifies the source code is responsible for writing the unit tests. The developer should know the paths of logic in that source code, and should be in the best position to write effective unit tests.

Most developers follow one of two approaches to unit testing:

- Create the unit test prior to code implementation; write and debug the code until all tests are passed. This is often referred to as Test Driven Development or TDD.

- Write the unit tests after code implementation to validate the implementation.

Both approaches have strengths and weaknesses, but due to familiarity with the second choice, most projects fall in this category. In either case, the developer will execute some form of testing on a local machine prior to checking in and promoting the code into a shared environment for building and testing.

[1] *Common Body of Knowledge (CBOK) for Certified Software Tester (CTSE)*, Quality Assurance Institute, 2012

Pragmatic Unit Testing Approach

There are three elements to pragmatic unit testing: Understand the requirements and the solution, Use white box testing as a form of validation, and Use available Java tools. By following these best practices, Java custom development projects can produce higher quality software solutions.

Understand the Requirements and the Solution

"The importance of the developers understanding the requirements and the functional analyst understanding the solution cannot be overstated."

Requirements are gathered by functional analysts who work with stakeholders to understand the business activity to be automated. It is very challenging to ask business stakeholders to state their needs in terms that a software developer can implement. As a result, there are multiple steps required to move from a business need or problem, to a software implementation.

A common complaint about requirements is that they are not clear, concise, and testable. In order to write code that the business users will find useful and valuable, the requirements need to be understood by the development team and the business stakeholders. Developers and functional analysts need to agree and understand two key areas:

- The business problem being addressed by the requirements
- The proposed technical solution meeting the business requirements

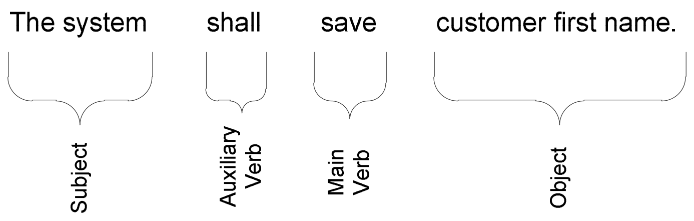
The importance of the developers understanding the requirements and the functional analyst understanding the solution cannot be overstated. The proverbial 'throwing the requirements over the wall' is a consistent theme when functional and technical resources do not spend the time needed to understand both perspectives.

Characteristics of Testable Requirements

Requirements are best translated into source code when they hold the following characteristics:

- **Testable:** A requirement is testable when the requirement statement can be confirmed true and the statement can be proven false. Can a test be written with a pre-condition, steps, and an expected outcome that can be validated? Do not use generic terminology and catch all phrases like: user friendly, big, lots, etc. as those cannot be tested.
- **Clear:** A requirement is clear when it defines what is necessary to make the software product complete, and does not leave room for interpretation or misinterpretation. Requirement reviews between the functional analysts, developers, and other stakeholders provide insight into requirement interpretation.

- **Concise:** A requirement is concise when the requirement is not wordy; it is short and to the point. Requirements are broken down into a granular level, which prevents compounding several requirements into one requirement. Look for words to identify compound requirements: and, or, and/or, but.
- **Consistent Form:** A requirement is in consistent form when all of the requirements are written in the same format using a simple word form. The following example uses a subject, auxiliary verb, main verb, and object.



Misunderstandings occur when the developers and functional analysts do not understand the requirements and the solution. Poorly written requirements are a common cause of the misunderstanding which propagates throughout the SDLC. The table below provides an example of poorly written requirements and the modifications necessary to turn the requirement into a testable requirement using the characteristics defined in this section.

Example of Bad Requirement	Modifications	Example of Testable Requirement
Users want their personal information in the system, but not their social security number.	Testable: A 'want' or 'desire' is not something that can logically be confirmed true or proven false. Clear: The requirement does not indicate if the information is required or optional for the user. The main verb is updated to a more descriptive term which is testable. Clear: The 'Personal information' term does not clearly define the field level information. The objects are updated to include specific personal data elements. Concise: The 'but not their social security number' portion of the requirement is a compound requirement. The compound is removed and two requirements replace the one compound. Consistent Form: The subject is not related to the system, which is what the focus of the software requirements that will be coded by a developer. The subject is updated to reflect the system that is under development.	The system shall require the following data fields: First Name, Last Name. The system shall not provide a data entry field for social security number.

Example of Bad Requirement	Modifications	Example of Testable Requirement
Use the data entered by the user to create the full name.	Testable: The requirement does not specify which user entered data the system should take into account and it does not state what should be done with the data. Without the key pieces of information, the requirement is not something that can logically be confirmed true or proven false. Clear: The requirement does not specify which user entered data the system should take into account. The objects are updated to include specific personal data elements. Clear: The verb 'create' can leave room for interpretation. The main verb is updated to be clearer on the specific action that is to be completed. Consistent Form: The subject is not related to the system, which is the focus of the software requirements that are coded by a developer. The subject is updated to reflect the system that is under development.	The system shall concatenate the First Name field and Last Name field to create a Full Name field.
Show user friendly error messages.	Testable: The requirement does not specify which situation generates an error. Without knowing the condition of when to show an error message, the requirement is not something that can logically be confirmed true or proven false. Clear: The requirement does not specify what causes an error message. The objects are updated to include specific personal data elements. Clear: The verb 'show' can leave room for interpretation. The main verb is updated to be clearer on the specific action of displaying a message to the user. Consistent Form: The subject is not related to the system, which is the focus of the software requirements that are coded by a developer. The subject is updated to reflect the system that is under development.	The system shall display an error message inline with the field when required fields are not provided.

Unit testing can be better explained using examples, such as a Customer Contact system. The following testable requirements are used throughout this paper:

1. The system shall require the following data fields: First Name, Last Name.

2. The system shall concatenate the First Name and Last Name fields to create a Full Name field

3. The system shall display an error message inline with the field when required fields are not provided.

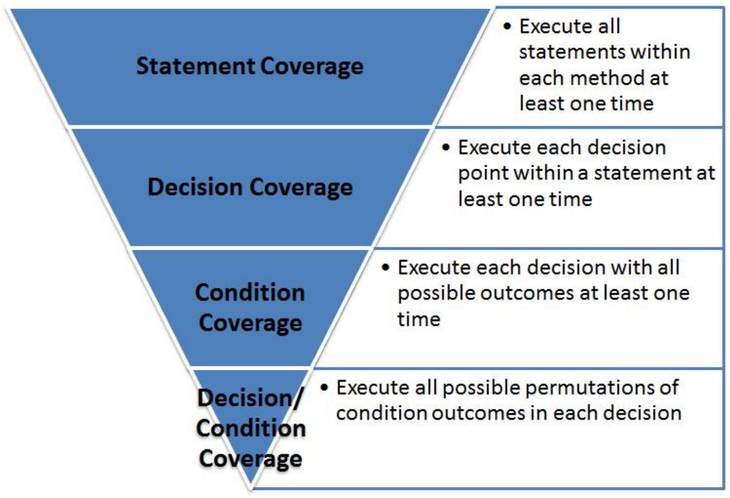
The requirement examples provided above are testable, but they may not account for every possible requirement necessary to meet the goals of the system. The term implicit requirement is used to describe a requirement that is not explained or documented, but a stakeholder assumes that it is part of the requirements. Depending on the experience level of a developer, the implicit requirement may or may not be included in the solution. Both developers and functional analyst need to understand the business problems addressed by the requirements so that they can identify implicit requirements.

Requirement 1 states the First Name and Last Name fields are required. If the user enters blank spaces into either field, it is technically acceptable from the developer's perspective but, raises the question as to what is acceptable from the business point of view. This is when it is important for developers to work with the functional analyst to clarify the requirements and eliminate ambiguity.

After the functional analysts and developers establish a common understanding of the testable requirements and the solution, the next step is to write the code and create the unit tests. The second point of the pragmatic unit testing approach is to apply white box testing techniques as a form of validation.

Use White Box Testing as a Form of Validation

Black box testing tests the system without any knowledge of the implementation. An alternative approach is white box testing where the test author uses the implementation details to create tests tailored to the solution. Program elements such as class structure, method calls and internal design drive the types of unit tests that are written. The graphic below describes the common white-box testing techniques.


Statement Coverage
• Execute all statements within each method at least one time
Decision Coverage
• Execute each decision point within a statement at least one time
Condition Coverage
• Execute each decision with all possible outcomes at least one time
Decision/ Condition Coverage
• Execute all possible permutations of condition outcomes in each decision

White Box Testing Example in Java

The following method is an implementation of the example requirements:

```
public class Customer
{
     public String getFullName(String firstName, String lastName)
     {
     if((null == firstName) || firstName.trim().isEmpty())
     {
          throw new IllegalArgumentException("Invalid First Name");
     }
     if((null == lastName) || lastName.trim().isEmpty())
     {
          throw new IllegalArgumentException("Invalid Last Name");
     }
     return firstName + " " + lastName;
     }
}
```

Statement Coverage

Using the Java library JUnit, the following test case executes all statements of the implementing method at least once.

```
@Test Case 1
public void testGetFullName_1()
{
     String expectedResult = "John Doe";
     String actualResult = new Customer().getFullName("John", "Doe");
     assertTrue(expectedResult.equals(actualReuslt));
}
```

Decision Coverage

A decision in Java code is implemented as an *if/then/else* statement or a *case*/*switch* construct. Decision Coverage is having test cases that cover each decision branch (path) at least once. There are two decision branches:

- Decision 1 - `if((null == firstName) || firstName.isEmpty())`
- Decision 2 - `if((null == lastName)  || lastName.isEmpty())`

Test Case 1 provides decision coverage since both *if* statements are called.

Conditional Coverage

Conditional coverage means you need to test both the *true* and *false* outcome of each condition. To have proper decision coverage each branch should evaluate to both *true* and *false* at least once. Test Case 1 evaluates each decision branch as false so, no additional test cases are necessary for that scenario. Additional test cases are needed to evaluate each branch as true. In these cases, we expect the code to throw exceptions, so in test cases 2 and 3 below, we catch it.

```
@Test Case 2
public void testGetFullName_2()
{
	try
	{
		new Customer().getFullName(null, "Doe");
		fail("Expected IIllegalArgumentException for First Name");
	}
	catch(IllegalArgumentException ignore)
	{
		//No need to check here
	}
}


@Test Case 3
public void testGetFullName_3()
{
	try
	{
		new Customer().getFullName("John", null);
		fail("Expected IIllegalArgumentException for Last Name");
	}
	catch (IllegalArgumentException ignore)
	{
		//No need to check here
	}
}
```

Decision/Conditional Coverage

This type of coverage tests each condition of all inner boolean expressions inside a decision branch independently. Some outcomes may have already been tested, so a truth table is a good way to identify the test cases that have occurred.

Using the first conditional as an example:

```
	if((null == firstName) || firstName.isEmpty())
```

The table below helps distinguish what outcomes have been tested and which require test cases:

Logic Path	null == firstName	firstName.isEmpty()	|| (or)	Test Case(s)
1	F	F	F	Test Case 1 and 3

2	T	T/F	T	Test Case 2
3	F	T	T	Not tested

From the truth table, we see the following:

- Logic path 1 tests ‘not null’ or ‘not empty’ firstName field and two test cases have covered this path
- Logic path 2 tests ‘null’ firstName. A null Java variable cannot be tested for isEmpty() so both conditions are tested
- Logic path 3 covers a ‘not null’ firstName and an ‘empty’ firstName and has not been tested
- To cover logic path 3, we will need to write another test case which has a ‘not null’ firstName but it is an empty string.

Test case 4 completes the truth table above.

```
@Test Case 4
public void testGetFullName_4()
{
     try
     {
          new Customer().getFullName("", "Doe");
          fail("Expected IIllegalArgumentException for First Name");
     }
     catch (IllegalArgumentException ignore)
     {
          //No need to check here
     }
}
```

To complete this test excercise we repeat test case 4 for the last name field. Testing a ‘not null’ and ‘not empty’ lastName. This is shown in test case 5 below.

```
@Test Case 5
public void testGetFullName_5()
{
      try
      {
            new Customer().getFullName("John", "");
            fail("Expected IllegalArgumentException for Last Name");
      }
      catch (IllegalArgumentException ignore)
      {
            //No need to check
      }
}
```

By using truth tables and knowledge of program structure we have created 5 unit tests to exercise the logic. What should be obvious at this point is that this is a lot of work, in fact, probably more work than it took to write the original code! The pragmatic aspect of unit testing drives when and how much Java code to write exhaustive unit tests on. Here are some general rules of thumb

1. Will this code be used by many other parts of the system?
   Java code that will be called often in the system should have more unit tests due to the broad impacts of a defect.

2. How important is the code?
   Java code that is not used often, but could have significant real-world impacts, like financial calculations, benefits issuance, mis-shipped goods, etc.

3. Will the code be changed often?
   Java code that is in a volatile part of the system that could change often due to policy, market changes, environment changes, business conditions, etc., should have more unit tests.

4. Is the code very complex?
   Java code that is complex scares most good developers due to the laws of unintended consequences. If they change something without fully understanding what the code does it can cause system issues. This type of code should have more unit tests to clarify what is expected.

Use Java Tools

Some of the more popular frameworks for Java unit testing are JUnit and TestNG. Both frameworks are available for free and can be easily integrated in your development environment and build scripts. There are tools both fee and free that generate JUnit tests on large code bases that have low test coverage.

“As the number of units of developed code continues to grow, the number of individual unit tests should also increase.”

As the number of units of developed code continues to grow, the number of individual unit tests should also increase. Unit tests that are related can be grouped together to form a suite of regression tests. Once a new unit of new code has passes its own testing then, regression tests are executed to identify if any defect has been injected into the existing code. Typically, these types of regression tests are executed automatically on the server-side by combining a testing tool with an automated code building tool such as JUnit and CruiseControl.

Conclusion

Pragmatic Java unit testing is a way to build quality into Java custom development projects from the beginning. This is accomplished by understanding the requirements and the solution, using white box testing as a form of validation, and using available Java tools.

As the management, functional, and development team members focus on the three elements to pragmatic unit testing, they will systematically test their code as it is developed and maintained. The list below highlights how developers and functional analyst can work together to increase the quality of Java custom development projects.

The project management and functional teams can improve unit testing by completing the following:

- Write testable requirements
- Review the requirements with the development team
- Review the solution with the functional team
- Allocate the time for unit testing in the project schedule
- Discuss the business/functional priorities with the development team to identify the pieces of code which require more unit testing

The development team members can improve their unit tests by completing the following:

- Clarify requirements with functional analyst
- Incorporate the suggested white-box testing techniques
- Use Java tools to automate unit testing

The earlier a defect is identified in the SDLC, the less costly it is to fix. The pragmatic testing approach will reduce costs, improve quality, and an increase in the likelihood of an on time delivery by identifying defects earlier in the SDLC.

Contributors

Primary contributors

Colleen Donnelly - cmdonnelly@deloitte.com

Colleen Donnelly holds the Certified Software Tester (CSTE) certification. Colleen is a Senior Consultant in the Systems Integration Service Line with 6.5 years of experience focusing on quality improvement in the public sector practice. Colleen earned a Bachelor of Science in Information Science and a related area in Computer Science from the University of Pittsburgh.

Manuel Stuart - mastuart@deloitte.com

Manuel serves as a Technology Practitioner in the Systems Integration Service Line within the Federal Practice. He brings 4 years of experience supporting and developing custom system applications applying the Software Development Life Cycle (SDLC). Manuel earned his Master in Business Administration from American University, and received his undergraduate degree in Computer Science from the University of Arkansas.

Java Community of Practice

The Java Community of Practice is a community of technology professionals whose primary focus is Java based technology and its applications. The community provides a platform for collaboration, exchange of ideas and encouraging contemporary best practices among its members. Our goal is to continuously enhance our capability to deliver technology solutions across different industries.

More information about the Java Community of Practice as well as additional collaboration and Java resources can be found at our KX site here.