7-2 Project Two

Cory Remick

CS-320 Software Test Automation

Joseph Rangitsch

April 16, 2022

Contents

[**Introduction** 3](#_Toc101013138)

[**Summary** 3](#_Toc101013139)

[**Alignment to Requirements** 3](#_Toc101013140)

[**Effective Tests** 7](#_Toc101013141)

[**Technically Sound Code** 8](#_Toc101013142)

[**Efficient Code** 8](#_Toc101013143)

[**Reflection** 10](#_Toc101013144)

[**Testing Techniques** 10](#_Toc101013145)

[**Caution** 13](#_Toc101013146)

[**Bias** 14](#_Toc101013147)

[**Conclusion** 15](#_Toc101013148)

[**References** 16](#_Toc101013149)

# **Introduction**

This report is for Grand Strand Systems to review the testing approaches used in the mobile application project. We will cover the techniques used, testing quality, and the testing experience and mindset used on the project.

# **Summary**

## **Alignment to Requirements**

The testing approach used is white box testing through unit testing using JUnit 5. Knowledge of the inner workings of the code is used to inform the tests to write. Test cases were selected to ensure that the typical use for a method operates as expected, that branches were exercised, and variable boundaries were tested. This set of tests together ensures that the foundation of the code performs as expected. Once the foundation was established, further tests were added to ensure the project requirements were met. These requirements listed what fields were required, which were changeable and not, and the limitations on what the variable could be, such as length or default value.

As an example of testing a typical use case, the contact service is required to allow the class consumer to add new contacts to a list.

**public** **void** add(Contact contact) {

Contact existingContact = get(contact.getId());

**if** (existingContact == **null**) {

contacts.add(contact);

}

**else** {

**throw** **new** IllegalArgumentException("'contact' already in service.");

}

}

The test for this code must ensure that it does what it should do. We attempt to add a new contact to the service and then ask for what we added back and compare the return value to what was submitted to ensure they are the same object. This verifies that the service does indeed hold the object added. One of the tests for the code above is below.

/\*

\* Test add method

\*/

@Test

@DisplayName("Add Contact")

**void** testAdd() {

ContactService service = **new** ContactService();

Contact contactAdd = **new** Contact("TestId1", "John", "Doe", "1234567890", "1 Main St. Nowhere, FL 12345");

service.add(contactAdd);

Contact contactReturn = service.get(0);

*assertSame*(contactAdd, contactReturn);

}

We also check branches to ensure that parts of the code are tested. There is a check in the add code above to see if the contact already exists "**if** (existingContact == **null**)." We added a test for the add method to ensure that adding a duplicate does indeed throw an exception. This is an example of killing two birds with one stone with one test. One bird is to ensure that the branch is covered, and the other is to make sure the design requirement for no duplicates is enforced.

/\*

\* Test cannot add duplicate contact

\*/

@Test

@DisplayName("No Duplicate Contacts")

**void** testNoDuplicates() {

ContactService service = **new** ContactService();

Contact contactAdd = **new** Contact("TestId1", "John", "Doe", "1234567890", "1 Main St. Nowhere, FL 12345");

service.add(contactAdd);

*assertThrows*(IllegalArgumentException.**class**, () -> {

Contact contactDup = **new** Contact("TestId1", "Jane", "Doe", "0987654321", "2 Main St. Nowhere, FL 54321");

service.add(contactDup);

});

}

We also tested variable boundaries to ensure that each case worked as expected. For example, an override of the equals() method was later used to test the results of the update() method. But a test of equals() was required to prove it behaved as expected before it could be used to prove that update() worked as expected. The equals() method returns a Boolean value that can either be 'true' or 'false.' If we had only tested for the positive case that it returns 'true,' that would not inform us that it works for the negative case. So, two tests were necessary for the same method. The code and tests are below.

// support equality

@Override

**public** **boolean** equals(Object o) {

**if** (**this** == o) **return** **true**; // object references match therefore same object

**if** (o == **null** || getClass() != o.getClass()) **return** **false**; // different types, so not equal

Contact contact = (Contact) o; // cast object to Contact

**return** id.equals(contact.getId()) // compare fields, only equal if all equal

&& firstName.equals(contact.getFirstName())

&& lastName.equals(contact.getLastName())

&& phone.equals(contact.getPhone())

&& address.equals(contact.address);

}

/\*

\* Test equals() method works for true case

\*/

@Test

**void** testEqualsObject() {

Contact testContact1 = **new** Contact("TestId1", "John", "Doe", "1234567890", "1 Main St. Nowhere, FL 12345");

Contact testContact2 = **new** Contact("TestId1", "John", "Doe", "1234567890", "1 Main St. Nowhere, FL 12345");

*assertTrue*(testContact1.equals(testContact2));

}

/\*

\* Test equals() method works for false case

\*/

@Test

**void** testNotEqualsObject() {

Contact testContact1 = **new** Contact("TestId1", "John", "Doe", "1234567890", "1 Main St. Nowhere, FL 12345");

Contact testContact2 = **new** Contact("NotTestId1", "John", "Doe", "1234567890", "1 Main St. Nowhere, FL 12345");

*assertFalse*(testContact1.equals(testContact2));

}

Lastly, we tested specific design requirements such that fields did not exceed certain lengths, and required values were specified for each field. For example, the requirements specify that a Task name cannot be longer than 20 characters, and there must be a value. It cannot be null. We created two tests to ensure that both conditions were checked correctly, one test to pass something longer than the required length and the other test to pass a null value.

**public** Task setName(String name) {

**if** (name == **null** || name.length() > 20) {

**throw** **new** IllegalArgumentException("Invalid argument 'name'.");

}

**this**.name = name;

**return** **this**;

}

/\*

\* Name cannot be longer than 20 characters

\* Test with 30 characters

\* Expect IllegalArgumentException

\*/

@Test

@DisplayName("Name Not Longer Than 20 Char")

**void** testNameNotLongerThan20() {

*assertThrows*(IllegalArgumentException.**class**, () -> {**new** Task("TestId1", "012345678901234567890123456789", ""); });

}

/\*

\* Name cannot be null

\* Test with null

\* Expect IllegalArgumentException

\*/

@Test

@DisplayName("Name Not Null")

**void** testNameNotNull() {

*assertThrows*(IllegalArgumentException.**class**, () -> {**new** Task("TestId1", **null**, ""); });

}

## **Effective Tests**

All of this means nothing if we could not test everything that needed to be tested. While we did not achieve 100% coverage, we did achieve over 90% in each of the four classes.





Some branches of edge cases were not tested. For example, in the equals() method, equality testing of the same object references and differing types was not exercised. However, the lines "&& name… and && description…" were covered but not highlighted properly, leaving some false reporting over the percentage covered. Therefore, due to uncommon use cases and tool limitations, it is unreasonable to expect 100% coverage to be an indicator of code quality. Ninety percent and over seems to be a reasonable number given the results.

// support equality

@Override

**public** **boolean** equals(Object o) {

**if** (**this** == o) **return** **true**; // object references match therefore same object

**if** (o == **null** || getClass() != o.getClass()) **return** **false**; // different types, so not equal

Task task = (Task) o; // cast object to Contact

**return** id.equals(task.getId()) // compare fields, only equal if all equal

&& name.equals(task.getName())

&& description.equals(task.getDescription());

}

## **Technically Sound Code**

Some design requirements were not written explicitly in code through conditional statements but in class architecture. For example, the condition that the identifier for Contact and Task classes be required and not null. To facilitate that, the classes were designed without default constructors. This forces the implementer to plan how these parameters will be filled in during instantiation. Notice the test cannot create an object without specifying an Id.

// constructors

// no default constructor provided because id is required and must be unique

**public** Task(String id) {

**this**(id, "", "");

}

**public** Task(String id, String name, String description) {

setId(id);

setName(name);

setDescription(description);

}

/\*

\* Test constructor works for common case

\*/

@Test

**void** testTaskConstructId() {

Task testTask = **new** Task("TestId1");

*assertNotNull*(testTask); // I would rather test assertDoesNotThrow but platform only has JUnit 5.0

}

/\*

\* Test constructor works for common case

\*/

@Test

**void** testTaskConstructStringStringString() {

Task testTask = **new** Task("TestId1", "Task 1", "This is a description.");

*assertNotNull*(testTask); // I would rather test assertDoesNotThrow but platform only has JUnit 5.0

}

## **Efficient Code**

The code written is also efficient in that the "don't repeat yourself" (DRY) principle is used. For example, there is a get() method implementation in the TaskService and ContactService classes to iterate the list to find an object by Id. The add methods of each service need to check if the object already exists before adding the object parameter to the list. Rather than reimplement the iterator pattern in the add method, it calls the class's get() method and inspects the return value to determine if the object is already in the collection. This separation means we do not have to repeat testing the get() and add() methods to ensure they iterate correctly.

**public** **void** add(Task task) {

Task existingTask = **this**.get(task.getId());

**if** (existingTask == **null**) {

tasks.add(task);

}

**else** {

**throw** **new** IllegalArgumentException("'task' already in service.");

}

}

**public** Task get(String id) {

Iterator<Task> taskIter = tasks.iterator();

Task foundTask = **null**;

**while** (taskIter.hasNext()) {

Task task = taskIter.next();

**if** (task.getId().equals(id)) {

foundTask = task;

**break**;

}

}

**return** foundTask;

}

We see that the testing approach works to achieve confidence in the foundation of the code by ensuring that functionality works for the common use case. Also, testing ensures that design requirements are met via required fields, field sizes, and proper default values. The code uses object-orientated principles of encapsulation and privacy to ensure that implementations of the classes cannot instantiate an object without an identifier. Furthermore, we have reasonably assured the code functions as expected due to the tests covering over ninety percent of the lines written. Lastly, the code follows the don't repeat yourself principle to reduce the burden of introducing multiple points of failure and increasing the testing burden.

# **Reflection**

## **Testing Techniques**

The testing approach used is white box testing through unit testing using JUnit 5. Knowledge of the inner workings of the code is used to inform the tests to write. The techniques used are Statement and Decision tests (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). However, we also use a boundary value and equivalence partition technique (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015) found under black-box testing to test that parameters meet design documentation requirements. Black-box testing is typically created not from knowledge of the code but the requirements of the design documents. However, even though these techniques enforce design requirements, they are made based upon the knowledge of the code for unit testing, so they are white-box approaches. We will next discuss the techniques used.

The Statement technique is employed to ensure that we have tested the execution of as many of the program's statements as possible to execute as expected (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). Another term for this is "code coverage." We noted in the last journal that we achieved over 90% code coverage for the tests employed. This means that the tests we wrote executed over 90% of the code statements written. In some cases, we could have written more tests to cover edge cases that would cover more branches, especially in the equals() method, but doing so would not have added much more value to the quality of the code. Also, we noted some limitations of the JUnit 5 tool in Eclipse to count the correct number of lines of code exercised when dealing with a multi-line logical expression statement. This led to a code coverage value reported less than what was executed. Therefore achieving 100% was impossible without refactoring the code to match the test tool's expectations.

The decision technique was employed to exercise conditional statements such as IF ELSE branches (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). Not much needs to be explained, other than to say that the tests are adequate if one test's parameters execute the IF branch and another test's parameters cause the ELSE branch to execute. Eclipse's Code Coverage tool using JUnit to exercise the code was used to confirm if the branch was taken or not.

Boundary value techniques determine if the parameters used are appropriately checked before they are used (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). For example, integers can be positive, zero, or negative. The code that is written may only want to deal with zero and positive integers. Therefore, it is necessary to test that the code can adequately deal with a negative value by returning an expected error code, exception, or message. This technique and Equivalence partitioning were used to ensure that specific fields met design requirements of not being over a particular field length and were not null. In Equivalence partitioning, if a test was successful in detecting the field was over the required length, then it is assumed it will be successful for any value greater than the length tested. Therefore, we need not write tests for every possible parameter value (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015).

Lastly, the State testing technique was employed. This technique requires the test to exercise a transition from one state to another (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). The only state that I can think of that might qualify for this is the requirement that no two identifiers be used in the same list of objects. In this case, the test would be testing that a particular state of adding another object is not possible from a state in which the list already has an existing Id.

There were certain techniques not employed in unit testing. One such approach, Static Analysis, was not used. This technique involves running a tool that reads the code as input and compares the statements against a list of rules that check the code for coding errors (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). This technique can detect structural design flaws and deprecated API choices and recommend best coding practices and standards.

Another technique not used was a decision table. The decision table maps out every possible combination of dependent variables to ensure the code handles the combinations (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). So far, the design requirements do not specify dependencies between fields or classes, so this technique is not necessary.

Use Case testing was also not used. In Use Case testing, design documents specify users' specific roles and how they interact with the system (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). The design documents for this project do not specify actors and how they interact with the classes under test. Therefore, testing the code using a sequence of events was not used.

The design documents also did not specify flow charts or control flow graphs that map out how a program should execute (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). It is unnecessary to test whether the program adheres to the expected flows without those specifications.

Lastly, we did not employ experience-based techniques such as Error Guessing and Exploratory testing. In Error Guessing, the tester uses their experience to create a test that will likely cover an expected problem (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). As it was enough to cover the expected statements and branches, additional tests to cover theoretical problems seemed unnecessary. In Exploratory testing, the tester anticipates how the program might flow in unexpected ways and create tests to cover those cases (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015). As the design documents do not specify how the classes will be employed, this type of testing is premature.

## **Caution**

As a software developer, I used a mindset of writing clean, efficient, and performant code that met the feature requirements. By clean, I mean writing code that is easy to read, and the reviewer can understand what it is doing. By efficient, I mean using code reuse and standard platform APIs to reduce the amount of code written and tested. Lastly, by performant, I mean code that executes as quickly as possible while maintaining the other clean and efficient goals.

As a tester, though, I had to switch to a mindset of proving that the functionality works as expected. One area that required caution was to examine how the collection classes Contacts, Appointments, and Tasks related to their respective data object classes, Contact, Appointment, and Task. The tests for the collection classes required properly constructed objects to be passed to the methods of the collections for add, update, and delete. For example, the add methods require checking for duplicate identifiers before adding an object.

/\*

\* Test cannot add duplicate contact

\*/

@Test

@DisplayName("No Duplicate Contacts")

**void** testNoDuplicates() {

ContactService service = **new** ContactService();

Contact contactAdd = **new** Contact("TestId1", "John", "Doe", "1234567890", "1 Main St. Nowhere, FL 12345");

service.add(contactAdd);

*assertThrows*(IllegalArgumentException.**class**, () -> {

Contact contactDup = **new** Contact("TestId1", "Jane", "Doe", "0987654321", "2 Main St. Nowhere, FL 54321");

service.add(contactDup);

});

}

Tests were written not just to test adding an object to the collections but also to test adding duplicate objects and ensuring the code returned a proper exception.

## **Bias**

Writing tests for code I wrote allowed for the possibility of bias to impede the testing process. To overcome that bias, I ensured that the specific requirements outlined in the design documents were tested. I also ensured that over 90% of the code was covered by testing. However, upon reflection, I still allowed some bias to creep into testing the project. I made some decisions on whether it was worth it or not to write additional tests for things that were not specified in the design documents. For example, an equals() method was not specified. Therefore, I only ensured that it worked for the typical case of comparing properties on both objects.

// support equality

@Override

**public** **boolean** equals(Object o) {

**if** (**this** == o) **return** **true**; // object references match therefore same object

**if** (o == **null** || getClass() != o.getClass()) **return** **false**; // different types, so not equal

Task task = (Task) o; // cast object to Contact

**return** id.equals(task.getId()) // compare fields, only equal if all equal

&& name.equals(task.getName())

&& description.equals(task.getDescription());

}

I did not add additional tests for the various branches by comparing the same object, null objects, and different classes. Instead, I relied on reviewing my code as sufficient to ensure it worked as expected.

**Discipline**

I maintained discipline as a tester to go through the effort to duplicate tests from one project to another. For example, the tests I wrote for the Contacts project were copied into the Tasks project and updated for the different names and possibly different field lengths. The code for both projects was very similar. It would have been easy to think that since they follow the same pattern with different names, I could save time by only testing what was different in the two projects. Nevertheless, I tested all the same things in each project. I discovered that the tests were necessary, as I discovered that copy and paste errors had crept into the Tasks project by failing to update several variable names. It was through testing that these problems were found and corrected.

# **Conclusion**

As a tester, I endeavored to employ the appropriate testing techniques based on the project's requirements. I maintained quality by ensuring that the requirements were met and that over 90% of the code was tested. I strived to remove bias from testing though some still were discovered in a small number of edge cases. I continued to write tests as a discipline even when I questioned whether it was warranted.

# **References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition).* BCS The Chartered Institute for IT. Retrieved April 3rd, 2022, from https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb