Southern New Hampshire University

CS-320: Assignment 4-2: Module 4 Journal
Prepared on: March 31, 2024
Prepared for: Prof. Angelo Luo
Prepared by: Alexander Ahmann

Contents

1 Background

As I have discussed in earlier papers (Ahmann, 2024a,b), *software testing* is an important part of developing good and reliable software products. Software testing helps organisations achieve their selfish goals of maximising profits by reducing the potential costs associated with fixing software errors, and it altruistically helps the greater society by preventing potential injuries or deaths caused by defective software. In this paper, I will discuss my attempts at developing unit tests for the Java programming language with *JUnit*.

2 Case Study: Java Accounting

This class assigned us a practical programming project which involves creating a basic accounting system in the *Java* programming language which, at the moment, keeps track of contact and task information. It accomplishes this with an in-memory database that holds this information in a Java class that acts as an analog to the notion of a `struct` in C++.

The `Contact` and `Task` classes would hold private `String` variables storing attributes associated with each particular instance of the thing that it is meant to represent. `get*()` and `set*(*ID)` methods[1] written to access or modify them if needed.[2] Next, the `ContactService` and `TaskService` serves as an in-memory database which stores their respective `Contact` or `Task` objects in a `Hashtable` key-value data structure. These `*Service` classes would have four methods: an `add*(**args)` function, an `update*(**args)` method, a `delete*(**args)` method, and a `get*()` method.

2.1 Implementation

The following pseudocode shows how the `Contact` and `Task` constructs work:

```
public class Construct {

    private String attrib1;
    private String attrib2;

    ...

    private String attribn;

    public Construct(String attrib1, String attrib2 ...
        String attribn) {
        if attrib* == null || attrib* > max(len(attrib*)) {
            throw IllegalArgumentException();
        }

        this.attrib1 = attrib1;
        this.attrib2 = attrib2;

        ...

        this.attribn = attribn;
    }

    private String getAttrib*() {
```

[1]The asterisk is a placeholder for a specific case of the more general notion of the thing in the accounting system.

[2]Most `String` attributes can be modified with its respective `set*(*ID)` method. The exception is the ID of the entity.

```
        return attrib*;
    }

    private void setAttrib*(String x) {
        this.attrib* = x;
    }
}
```

Like previously stated, the class is treated like a C++ `struct` and more primitative data types like `String` are used to store the entity's characteristics. The traditional `get*()` and `set*(String x)` methods in objected-oriented programming are used to access and update the `Strings` in the in-memory database.

Regarding the `*Service` classes, the following pseudocode describes their general functionality:

```
public class Service {

    private Hashtable<String, Construct> constructDatabase
        = new Hashtable<String, Construct>();

    public static void addConstruct(String ID, String attrib1,
        String attrib2 ... String attribn) {
        if (constructDatabaae.get(ID) != null)
            throw new Exception();
        constructDatabase.put(ID, new Task(ID, attrib1,
            attrib2 ... attribn));
    }
}
```

Here, the "`Service`" class is declared and I implemented a high-level pseudocode of its `addConstruct` functionality. The class defines a `Hashtable` database consisting of a `String` as its key and a `Construct` as its value. The key is the unique `ID` of the construct, and the `Construct` is a generalisation for `Contact` or `Task`.

The `addConstruct` first checks the `Hashtable` database to see if the `ID` is already entered in. If it is, then it throws an exception. It otherwise would add a new instance of the construct with the HashTable's `.put(key, value)` method, which has the unique ID as the key and the `Construct` as the value.

Next, I have written the pseudocode to describe the functionality to update the attributed of the construct:

```
[... snip ...]

    public static void updateConstruct(String ID,
        int field, String data) {
        Construct currentConstruct = constructDatabase.get(ID);
        if (currentConstuct == null)
            throw new Exception();
        switch (field) {
        case 1:
            currentConstruct.setField1(data);
            break;
        case 2:
            currentConstruct.setField2(data);
            break;

        ...

        case n:
            currentConstruct.setFieldN(data);
        default:
            break;
        }
    }

[... snip ...]
```

To update the field, the `updateConstruct(**args)` method takes three arguments: the ID of the construct, an integer called `field` that stands for a particular attribute of the construct, and another string called `data` that will replace the current value of the attribute. If the unique ID of the construct being queried does not exist, it will throw an exception. Otherwise, it will execute a `switch` that is conditional on the field integer, and then will use the construct's respective `set` method to update the field.

Next, I have written the following pseudocode to describe the functionality of the delete functionality of the `*Service` database:

```
[... snip ...]
```

```
    public void deleteConstruct(String ID) {
        Construct currentConstruct = constructDatabase.get(ID);
        if (currentConstruct == null)
            throw new Exception();
        constructDatabase.remove(ID);
    }

[... snip ...]
```

Here, the `deleteConstruct` method takes in the ID of a particular instance of the construct, the it attempts to retrieve it from the `Hashtable` through its `.get(key)` method. If it returns a `null`, then it throws an exception. Otherwise, it will remove the instance of the construct with its `.remove(key)` method, parameterised by the construct's unique ID.

Finally, I have written a `getConstruct()` method. But I will not be discussing it in this paper because it was not mentioned in the guidelines in the project.

2.2 Writing JUnit Test Cases

I used Java's `JUnit` library to write test units to demonstrate the correctness of the code that I wrote.[3] I do not want to go through all of my test cases in this subsection, but I will go over excerpts of one of them to give the reader an idea of what I was doing. I decided to go with `Contact.java` to demonstrate examples of test units that I wrote. I began by importing necessary libraries:[4]

```
import java.io.*;
import java.util.*;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;

import contact.Contact;
import contact.ContactService;
```

[3]I use the term "demonstrate" as opposed to "prove" because the former entails using empirical methods and the latter entails using mathematical proof. The difference being that empirical methods are not final, where as mathematical proof is final. See He et al. (1994).

[4]Excerpt from Ahmann (2024c, `Contact.java`).

I then proceeded to write testing methods with the `@Test` annotation:[5] The following test unit assures that the get*(String *ID) methods are working properly.

```
    @Test
    void testContact() {
        Contact contactClass = new Contact("12345", "Aleksey",
            "Ahmann", "5555555555", "1 Hacker Way");
        assertTrue(contactClass.getContactID().equals("12345"));
        assertTrue(contactClass.getFirstName().equals("Aleksey"));
        assertTrue(contactClass.getLastName().equals("Ahmann"));
        assertTrue(contactClass.getPhone().equals("5555555555"));
        assertTrue(contactClass.getAddress().equals("1 Hacker Way"));
    }
```

I also needed to test for invalid parameters. The following tests for invalid `firstName` inputs:

```
[... snip ...]

    @Test
    void testContactIDIsNull() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            new Contact (null, "Aleksey", "Ahmann",
            "5555555555", "1 Hacker Way");
        });
    }

    @Test
    void testFirstNameTooLong() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            new Contact ("12345", "FirstNameIsTooLong1234567890",
                "Ahmann", "5555555555", "1 Hacker Way");
        });
    }
```

[5]Through feedback from a previous submission given by Prof. Luo, I have used `@BeforeEach` to initialise instances of `*Service` classes. They recommended the following resource for implementing it (Retrieved on Mar. 31, 2024):
https://www.educative.io/answers/what-is-the-before-annotation-in-junit-testing

```
[... snip ...]
```

I do not want to discuss every possible test method, as that will make the paper unnecessarily long and redundant. The important thing to know is that there is a maximum length for each `String` type in a class, and inputs cannot be `null`. I used `Assertions.assertThrows()` to make sure that `IllegalArgumentExceptions` were thrown when `String` inputs became too long or were null.

3 Discussion

The software requirements were defined in CS-320 (n.d.-a,n), and they were assessed with the test units written to compared the observed behaviour to the expected behaviour of the software.[6] The quality of the *JUnit* tests that I wrote met a fair standard of decency, as I was able to work out all possible illegal paths based on the specifications, and test for them.[7]

Furthermore, the JUnit tests ensure the technical soundness of my code by acting like a second measuring device to ensure that it is functioning properly. For example, consider the following code from `Contact.java` that creates a “Contact” construct to store contact information:

```
[... snip ...]

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        if (firstName == null || firstName.length() > 10)
            throw new IllegalArgumentException("Invalid 'firstName'");
        this.firstName = firstName;
    }

[... snip ... ]
```

[6]This answers the question “[t]o what extent was your testing approach aligned to the software requirements? Support your claims with specific evidence.”

[7]This addresses the criteria to “[d]efend the overall quality of your JUnit tests for the contact service and task service.” and answers the question “how do you know that your JUnit tests were effective on the basis of coverage percentage?”

The test methods `testFirstNameTooLong()` and `testFirstNameIsNull()` tests both these methods by comparing the output of `getFirstName` to its expected output, and making sure that the `setFirstName` method is changing the value of the first name as long as it is within the specifications of being less than or equal to ten (10) and not `null`.[8]

References

Ahmann, A. (2024a). *~~CS 330~~ CS-320: Assignment 1-3: Module 1 Journal.*

Ahmann, A. (2024b). *~~CS 330~~ CS-320: Assignment 2-2: Module 2 Journal.*

Ahmann, A. (2024c). *CS-320: Assignment 4-1 Codebase.*

CS-320 (n.d.-a). *CS 320 Module Four Milestone Guidelines and Rubric.* Southern New Hampshire University.

CS-320 (n.d.-b). *CS 320 Module Four Milestone Guidelines and Rubric.* Southern New Hampshire University.

He, J. et al. (1994). Provably Correct Systems. *In: Langmaack, H., de Roever, WP., Vytopil, J. (eds) Formal Techniques in Real-Time and Fault-Tolerant Systems.* FTRTFT ProCoS 1994 1994. Lecture Notes in Computer Science, vol 863. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-58468-4_171

[8]This answers the question "[h]ow did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate," along with the other question of "[h]ow did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate."