Southern New Hampshire University

CS-320: Assignment 7-2: Module 7 Journal

Prepared on: April 22, 2024

Prepared for: Prof. Angelo Luo

Prepared by: Alexander Ahmann

Contents

1 Background

Software testing is the step of the *software development life cycle*[1] that concerns itself with the assessment and ensurance of the quality and corectness of a software product or a computer system. I have written about good software testing in previous papers (Ahmann, 2024a,b,c) and would like to culminate my thoughts about software testing into this final report.[2] I also intend to discuss a case study Java application that I worked on as part of a class on software testing.

[1] Typically shortened to "SDLC."

[2] For the sake of concision, I had to remove parts that went into too much detail.

Software testing is arguably the most important part of software development, as the skilled computer scientist E. W. Dijkstra contends that computer programming is the process of introducing bugs.[3] To put it another way, programming introduces risk in the form of potential defects, and software testing removes such defects before they can introduce risk to the system. New software has the potential to improve the greater society, but if not properly implemented, it can cause more problems.

2 Case Study: Accounting Service

For the Southern New Hampshire University *CS-320* course, I developed a simple Java codebase that acts as an in-memory database for storing contact information and other accounting data. I developed three Java modules: `appointment`, `contact` and `task`, which will store `String` data whose meaning is not of my concern.[4] What is of my concern are constraints defined in the project specifications[5] and implementing them to function as reliably as possible.

2.1 Project Requirements

As discussed earlier, `appointment`, `contact` and `task` Java modules make up the library package. All of them consist of two kinds of source code files: the "abstract type" source file and the "service database" source file. The abstract type would define the entity that the software will keep track of, and the service file is the in-memory database to manage instances of the abstract type. For each of the Java modules, I named the source code files with the following convention, and they are listed in table 1.

Next, I had to write source code to solve the problems described in each of the project guidelines. The `Contact.java`, `Task.java` and `Appointment.java` source code files held a `public class` that would define a higher level entity[6] Each of the "abstract type" classes consist of a unique ID with the data type of a `String`. These classes will store accounting information in a `String` variable— with the `Appointment` abstract type storing a date in the form

[3]To quote Dijkstra exactly: *"[i]f debugging is the process of removing software bugs, then programming must be the process of putting them in."* From the following web page (Retrieved Apr. 16, 2024): https://www.goodreads.com/author/quotes/1013817.Edsger_W_Dijkstra

[4]In the case of the `appointment` module, I stored a `Date` data type in addition to the usual `String` data types.

[5]See CS-320 (n.d.-b), CS-320 (n.d.-c) and CS-320 (n.d.-d).

[6]This is "higher level" when compared to the `int` or `char` data type.

Package	"Abstract Type"	"Service Database"
`appointment`	`Appointments.java`	`AppointmentService.java`
`contact`	`Contact.java`	`ContactService.java`
`task`	`Task.java`	`TaskService.java`

Table 1: `.java` source code by its respective module and kind.

of a Java `Date` data type. Each of these data entries in these classes have a `get*` method, which returns the data type's value, and a `set*` method that changes its value given a new input. The only data entry that does not have a `set*` method is the unique ID, which is supposed to be immutable as defined by the specifications. I used a constructor to initialise each instance of an abstract object.

Finally, I proceeded to write an in-memory class for each of the respective "abstract types" that I am working with. The naming convention for the in-memory database service would be `<abstract type>Service.java`— with `<abstract type>` being a placeholder for the abstract type that I implemented earlier. The in-memory database is a `HashMap<K, V>` data type with the unique ID[7] as the key and the abstract type as its value. Each version of the `*Service` class may have methods to add, update or delete entries in the in-memory `HashMap` database— with all of them having a method to return the value of a particular entry given its unique ID.

This is just a summary of how I accomplished the task of implementing higher-level data types and their respective in-memory database services. I needed a way to test each of these modules, which is where *unit testing* comes into play.

2.2 Unit Tests: Examples

I wrote *unit tests* with the JUnit 5 (n.d.) framework in order to assure that my codebase functioned as specified in the project guidelines.[8,9,10] More specifically, when writing code for `Contact.java`, I was faced with constraints for implementing abstract objects described in CS-320 (n.d.-b).

To test each the classes and methods in a Java source code file, I cre-

[7]This is a `String` data type.

[8]This subsection addresses the following from CS-320 (n.d.): "Describe your unit testing approach for each of the three features."

[9]This subsection (also) addresses the following from CS-320 (n.d.): "What were the software testing techniques that you employed in this project? Describe their characteristics using specific details."

[10]I created another Java module called `test` which stored all the unit tests.

ated a JUnit test with that source code's filename and `Test` appended to it. For example: `Contact.java` would become `ContactTest.java`. In these `*Test.java` files, I would then write methods to assert that the constraints given in the project guidelines are met. Proceeding with the `ContactTest.java` example, I used the following snippet of code to declare an instance of the `Contact` abstract type, and then test it to ensure that it meets the project requirements:

```
1. @Test
2. void testContact() {
3.     Contact contactClass = new Contact("12345",
4.         "Aleksey", "Ahmann", "5555555555",
           "1 Hacker Way");
5.      assertTrue(contactClass.getContactID()
          .equals("12345"));
6.      assertTrue(contactClass.getFirstName()
          .equals("Aleksey"));
7.      assertTrue(contactClass.getLastName()
          .equals("Ahmann"));
8.      assertTrue(contactClass.getPhone()
          .equals("5555555555"));
9.      assertTrue(contactClass.getAddress()
          .equals("1 Hacker Way"));
10. }
11. // Excerpt from ContactTest.java
```

I used the `assertTrue` method that comes with the JUnit 5 (n.d.) library to ensure that the values of the appropriate fields are equal to the input arguments when declaring a new instance of the `Contact` abstract type. Next, I proceeded to write test methods to test how `Contact.java` handles exceptions:

```
1. @Test
2. void testContactIDTooLong() {
3.     Assertions.assertThrows
         (IllegalArgumentException.class, () -> {
4.         new Contact ("12345678909876543 21",
             "Aleksey", "Ahmann",
5.               "5555555555", "1 Hacker Way");
6.         });
7.     }
```

```
8.
9.  @Test
10. void testContactIDIsNull() {
11.     Assertions.assertThrows
          (IllegalArgumentException.class, () -> {
12.         new Contact (null, "Aleksey", "Ahmann",
            "5555555555", "1 Hacker Way");
13.     });
14. }
15. // Excerpt from ContactTest.java
```

The various attributes for the `Contact` are to be equal to or less than some fixed number. In the case of the `Contact` abstract type's unique string, its length must be less than ten (10) characters. I used the `Assertions.assertThrows` and the `IllegalArgumentException.class` constructs that come with the JUnit 5 (n.d.) framework to ensure that an exception is thrown when it is faced with input argument for a unique ID that go beyond ten characters. I wrote the same testing method to assess `String` attributes and ensure that they throw exceptions when given an invalid `String` length.

It would be tedious for me to examine all the test methods and test units that I wrote, so instead I would invite the reader to view the finished codebase submission (Ahmann, 2024d) to view more specific examples for themselves.

2.3 Unit Tests: Code Coverage

Figure 1 depicts the code coverage results after executing test units from the *Eclipse IDE*.[11]

[11] This subsection addresses the criterias 1a.ii, 1b.i, 1b.ii from the guidelines (CS-320, n.d.).

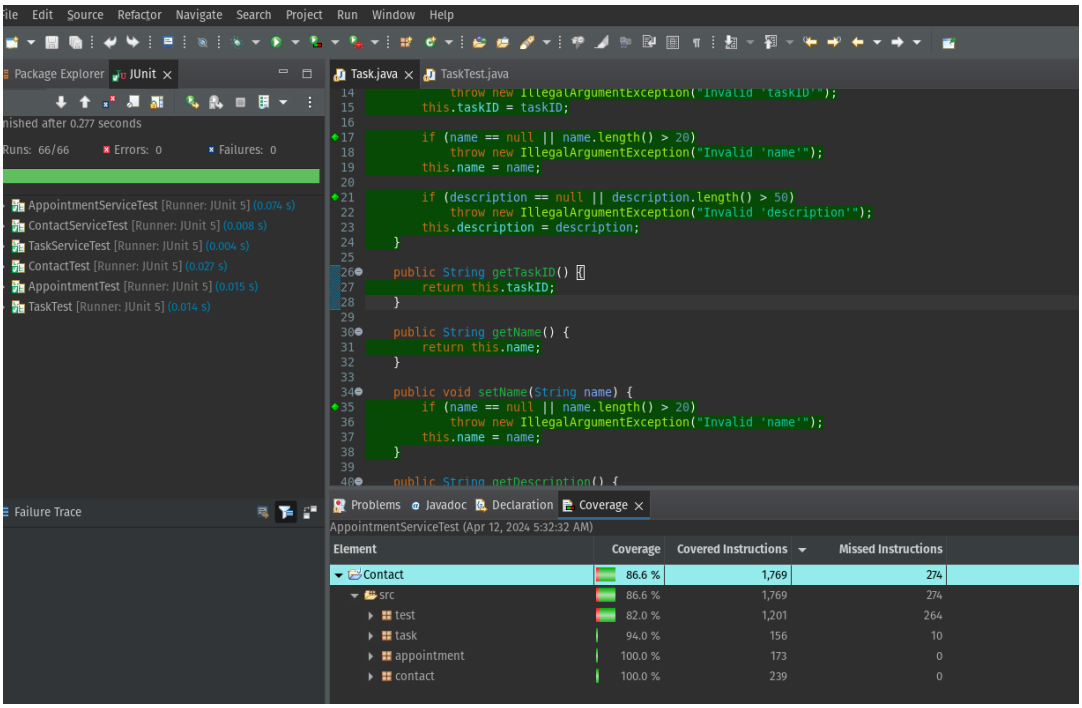

Figure 1: Code Coverage from the *Eclipse IDE*.

The bottom half of the Eclipse IDE shows a table depicting the results of covered instructions and missed instructions when executing the code coverage. The software reported a grand total of 86.6 per cent code coverage — which included the test cases. However, I am interested in only the code coverage of the abstract types and its respective in-memory service database. I used the following formula to calculate the code coverage for all the Java modules with the exception of the test units:

$$\%Y = \frac{100 \times \left(\sum C_{\rho \notin C}\right)}{\left(\sum C_{\rho \notin C}\right) + \left(\sum \neg C_{\rho \notin C}\right)} = \frac{100 \times (156 + 173 + 239)}{(156 + 173 + 239) + (10 + 0 + 0)} \approx 98.3\%$$

In this formula, the $\%Y$ is the per cent code coverage, the $\sum C_{\rho \notin C}$ is the total number of covered instructions that is not counting test units, and the $\sum \neg C_{\rho \notin C}$ is the number of missed instructions that is not counting test units. The logic of this formula is that the sum of the sum of covered instructions, and the sum of missed instructions[12] would add up to the total number of instructions in the codebase. Covered instructions is a subset of the total

[12]Not counting the test units, of course

number of instructions in the codebase, and the general formula would result in a per cent value of the code coverage.

If not taking test units into account, my code coverage is nearly 100 per cent. This near perfect code coverage implies that the test units were able to trigger most execution paths and, through the test units, demonstrate their correctness. Sixty-six (66) test units were executed, and all of them passed.[13] While this may not prove that the code is correct,[14] the unit testing that I employed is enough to show that the software is reliable in most situations.

3 Discussion

3.1 Different levels of testing

The case study was done as part of the requirements for Southern New Hampshire University's CS-320 class on software testing and quality assurance. This course concerned itself mainly with unit testing, which is a kind of software testing that works by testing a software application's individual components.[15] "GeeksForGeeks" (2024) lists software testing at different levels, and identifies the following levels of software testing that differ from what I used in this project:[16]

- *Integration Testing*: this is a level of testing "where individual units are combined and tested as a group." (quoted verbatim from "GeeksForGeeks" (2024))
- *System Testing*: this is a level of testing "where a complete, integrated system/software is tested." (quoted verbatim from "GeeksForGeeks" (2024))
- *Acceptance Testing*: this is a level of testing "where a system is tested for acceptability." (quoted verbatim from "GeeksForGeeks" (2024))

The different levels of testing correspond to the scale at which the software application will be deployed, or whether-or-not the software application does

[13] See the top-left pane in figure 1.

[14] The subject matter of *Provably Correct Systems* concerns itself with using mathematical proofs to give a binary and final proof of a software or information system's correctness. See He et al. (1994).

[15] Paraphrased from "GeeksForGeeks" (2024).

[16] This part addressing the following criteria from the guidelines (CS-320, n.d.): "What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details."

what it is expected to do. For example, I might need to use system testing to observe how a piece of software will respond to different computing systems.

3.2 Flexible Methodology

While writing the case study, I learnt a few useful things. I mainly learnt about the JUnit 5 (n.d.) framework and how it can be used to automatically carry out unit tests and work out the per cent of Java code coverage. As I learnt new things, I refactored my codebase to meet the requirements. The example that I want to discuss is rewriting `set*` methods to return values in order to achieve more code coverage. Consider the following method from `Contact.java`:

```
1. public boolean setAddress(String address) {
2.     if (address == null || address.length() > 30)
3.         throw new IllegalArgumentException
             ("Invalid 'address'");
4.      this.address = address;
5.      return true;
6. }
```

I initially had this method return nothing and defined it as a `public void`. But, I noticed that in order to assess the functionality of this method, it needed to return something so that I can use an `assert*` method to assess it. From previous experience programming in C/C++, I remember that its functions typically return an integer. Functions that were successful typically returned a 0, and if they ran into an error, they would return a non-zero integer — typically an error code mapped to a specific kind of error. I rewrote the `set*` methods to return a `boolean` type. It would return a `true` if executed successfully. If the method did not execute successfully, it would raise an exception and the JUnit 5 (n.d.) test units would make sure that they raised the proper exception.

3.3 Bias and Peer Review

When developing the case study, and indeed other kinds of software or (non-computer) systems, it will likely be the case that there are flaws in my designs and implementations, and it is equally likely that I am biased in my assessment of such flaws.[17] In this context, I will define bias as "the tendancy to

[17] This subsection concerns itself with the following criteria from the guidelines (CS-320, n.d.): "Assess the ways you tried to limit bias in your review of the code. On the software

overlook facts that challenge the quality of a software product or computer system."[18] Such bias can be a result of a conscious effort to dishonestly admit to the limitations of the software or system invented, but it is unconscious and the result of honest mistakes for the most part.

One possible source of bias is the tendency to underestimate the per cent of code coverage that is acceptable before deploying a software product. Software organisations may tolerate a 50 per cent code coverage, or even less, before giving approval to deploy their software product or service. Another possible bias could be a software developer's tendency to "cut corners" and use a technically less secure method of implementing software— like letting a software daemon run under the `root` user instead of giving it proper permissions. The preceeding example was given by Donnie Werner of the Disobey (2018) conference— where he discusses what he thinks is wrong with the state of software development.

To combat bias, software developers can employ the help of third-party companies to assess the quality of their code and identify programming errors. Another way to combat bias would be to make a software product open source, if possible, and subject the source code to an open code review.[19] Allowing the scrutiny of many programmers of varying skill and experience will allow for bugs and design flaws to be quickly worked out, and hopefully eliminated.

3.4 Limitations of the Case Study

Regarding the case study, I think that I can improve the codebase.[20] The three Java modules, `appointment`, `contact` and `task`, seem to have similar functionality in both their "abstract type" and "service database" classes. I think that I can use inheritance or polymorphism to reuse base functionality and avoid having to rewrite the same methods or classes. This is something that I may explore in the future.

---

developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims."

[18] This definition is in part inspired by how journalists define bias; see the following *Wikipedia* entry on Media Bias: https://en.wikipedia.org/w/index.php?title=Media_bias&oldid=1219494727

[19] An example of a good code review forum, which I have used multiple times, is *StackExchange Code Review*: https://codereview.stackexchange.com/

[20] This subsection addresses the following from the guidelines (CS-320, n.d.): "How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate."

4 Conclusion

Software testing is an important step of the *software development life cycle*. This paper discussed a case study of myself writing a simple in-memory database for storing accounting information, and developing unit tests with the JUnit 5 (n.d.) framework to demonstrate software correctness. I also discussed software testing at different levels, how learning new things can change up my workflow and methodology, biases in software development and how peer-review can mitigate them, and limitations of my case study. I also discussed how software testing can benefit self-interested entities by reducing costs and the greater society by reducing risk.

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 4-2: Module 4 Journal*.

Ahmann, A. (2024d). *6-1 Project One Submission: Codebase.zip*.

CS-320 (n.d.). *Project Two Guidelines and Rubric*.

CS-320 (n.d.-b). *CS 320 Module Three Milestone Guidelines and Rubric*.

CS-320 (n.d.-c). *CS 320 Module Four Milestone Guidelines and Rubric*.

CS-320 (n.d.-d). *CS 320 Module Five Milestone Guidelines and Rubric*.

Disobey (2018). *Disobey 2018— Hacker hints for developers— Donnie Werner*. YouTube Video. Retrieved on Apr. 22, 2024 from: https://www.youtube.com/watch?v=MzPVMIWYSpQ

"GeeksForGeeks" (Mar. 19, 2024). *What is Software Testing?* Retrieved on Apr. 21, 2024 from: https://www.geeksforgeeks.org/software-testing-basics/

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

JUnit 5 (n.d.). Retrieved on Apr. 21, 2024 from: https://junit.org/junit5/