7-2 Project Two Submission

“Programmers want to do a good job. They understand that their goal is to produce effective software, not software that merely passes tests.” (Gelperin and Hetzel, 1988).

To validate the various components of Project One, I aligned each of my unit tests to addressing a specific software requirement and did so as completely as possible. I implemented each unit test to validate a scenario as it relates to data integrity. I think I did an adequate job without being overly verbose. The fifty total unit tests I created were each built to test for positive and negative outcomes on a per field basis. To the best of my knowledge, there is no reasonable way for a user to submit, retrieve, update, or delete data in the system outside of the prescribed, acceptable methods.

For example, if a user attempted to add a new Appointment to the Appointment Service, the new Appointment instance needed to pass validity checks on every field in the object class:

```
// Constructors
public Appointment(String id, Date date, String description) {

    // Validate unique ID
    if (id == null || id.length() < 1 || id.length() > 10) {
        throw new IllegalArgumentException("You must specify a valid ID.");
    }

    // Validate date
    if (date == null || date.before(new Date())) {
        throw new IllegalArgumentException("You must specify a valid date.");
    }

    // Validate description
    if (description == null || description.length() < 1 || description.length() > 50) {
        throw new IllegalArgumentException("You must specify a valid description.");
    }

    this.id = id;
    this.date = date;
    this.description = description;
}
```

The associated JUnit tests also check for validity against multiple possible scenarios:

```
@Test
public void testID() {
    assertTrue(appointment != null);
    assertTrue(appointment.getID() != null);
    assertTrue(appointment.getID().length() > 0);
    assertTrue(appointment.getID().length() <= 10);
}

@Test
public void testID_TooShort() {
    exceptionRule.expect(IllegalArgumentException.class);
    new Appointment("", gc.getTime(), "Merry Christmas!");
}

@Test
public void testID_TooLong() {
    exceptionRule.expect(IllegalArgumentException.class);
    new Appointment("12345678910", gc.getTime(), "Merry Christmas!");
}

@Test
public void testDate() {
    assertTrue(appointment != null);
    assertTrue(appointment.getDate() != null);
    assertTrue(appointment.getDate().after(new Date()));
}

@Test
public void testDate_InPast() {
    exceptionRule.expect(IllegalArgumentException.class);
    GregorianCalendar gc = new GregorianCalendar(1900, Calendar.DECEMBER, 25, 0, 0, 0);
    new Appointment(VALID_ID, gc.getTime(), "Merry Christmas!");
}
```

Overall, I believe that writing these JUnit tests were a valuable experience. I have considerable experience writing unit tests already, but I am not a Java developer by trade, so being able to learn and understand assertations from another perspective helped to broaden my understanding of the concept.

I am confident that my code is technically sound because every line of code was written with the intent of satisfying the acceptance criteria. There is little if any cruft.

```java
@Test
public void testUpdate() {
    if (!cs.exists(VALID_ID))
        cs.add(contact);

    final String NEW_FIRST_NAME = "John";
    final String NEW_LAST_NAME = "Doe";
    final String NEW_PHONE = "5555555555";
    final String NEW_ADDRESS = "NEW ADDRESS";

    // Update entry
    cs.update(VALID_ID, NEW_FIRST_NAME, NEW_LAST_NAME, NEW_PHONE, NEW_ADDRESS);

    // Verify updated entry
    Contact updated = cs.get(VALID_ID);
    assertTrue(updated != null);
    assertTrue(updated.getFirstName().equals(NEW_FIRST_NAME));
    assertTrue(updated.getLastName().equals(NEW_LAST_NAME));
    assertTrue(updated.getPhone().equals(NEW_PHONE));
    assertTrue(updated.getAddress().equals(NEW_ADDRESS));

}
```

I used standard Java naming conventions, grammar, formatting, and datatypes. I validated data integrity as often as was required instead of assuming that the data from an external source would be “good”, and I am always on the lookout for possible Null Exceptions, the most common type of exception in my experience.

While building the Unit Tests, I focused on getting a single test working; then I would revise it as many times as needed until it not only worked but worked in the most efficient way I could manage. Once I got one test working, I ran all the Unit Tests over again to make sure I didn’t create any conflicts in my code coverage.

I believe it was a wise decision to use a HashMap to store collections of data rather than a more common collection data type such as an ArrayList, not only because it can be used to easily look up entries with an ID instead of an esoteric index, but a HashMap also enforces data integrity by

protecting against duplication of unique lookup keys while still allowing the value or record to contain the same information. (geeksforgeeks.org, 2019)

```
private Map<String, Appointment> appointments = new HashMap<String, Appointment>();
```

I would describe my mindset while writing these unit tests as “Cautiously Pessimistic”. I think it is a safe and wise approach to assume all data coming in is *bad*. However, I was also careful to not be overly finicky about the validations. For example, I didn’t put any validation in to protect against XSS (Cross Site Scripting) attacks or use of Unicode characters because the requirements did not specify those as possible error vectors.

It is important that a developer understand the interrogatability of the code they are testing. It is not always clear how a change in one place may affect the execution of code in another.

As Dr. Ahamed says, “Software bugs are a fact of life…even the best programmers can’t write error-free code all the time. On average, well-written programs have one to three bugs for every 100 statements.” (Ahamed, 2009).

When an *inevitable* bug does become introduced it is important to catch it as early as possible before it results in cascading errors which in turn result in the obfuscation of the root cause of the issue.

With the tools I have honed over the duration of this course, I feel confident that I will be able to deliver quality software written with the business needs in the forefront of my mind and the understanding that in the end your code is only as good as the first bug encountered by the user.

Citations

Ahamed, S. S. Riaz. (2009). Studying the Feasibility and Importance of Software Testing: An Analysis. Retrieved April 16, 2021, from https://arxiv.org/ftp/arxiv/papers/1001/1001.4193.pdf

geeksforgeeks.org. (2019, September 30). Difference between hashmap and HashSet. Retrieved April 16, 2021, from https://www.geeksforgeeks.org/difference-between-hashmap-and-hashset/

Gelperin, D. & Hetzel, B. (1988). The growth of software testing. Retrieved April 16, 2021, from https://www.researchgate.net/publication/234808293_The_growth_of_software_testing