When it came to unit testing for the Contact, Task, and Appointment services there were requirements. I was asked for the Contact service to be created by an object’s properties comprised of a (**String**) unique ID that can’t be null and cannot be longer than 10 characters, a (**String**)first name that can’t be null and cannot be longer than 10 characters, a (**String**) last name that can’t be null and cannot be greater than 10 characters, a (**String**) phone number that can’t be null and has to be a length of 10 characters, and a (**String**) address that can’t be null and cannot be greater than 30 characters.

if (variable == null || variable.length() > 10){  
 throw new IllegalArgumentException("variable must not be null and cannot be greater than 10 characters long...");  
 }  
 this.variable= variable;

This code snippet above shows how to put a constraint on an Object’s property to keep inline what the client's needs.

@Test  
@DisplayName("Test Contact ID too long")  
void testContactIDTooLong(){  
 Assertions.*assertThrows*(IllegalArgumentException.class, () -> {  
 new Contact(longID, firstName, lastName, number, address);  
 });  
}

This code snippet above shows how we can test the constraints of the   
Contact service and result in a pass/fail to see if the code is working properly. JUnit5 shows that we annotate a unit test with @Test before a method call and @DisplayName annotation will show in the console the description of the test (Bechtold et al., n.d.).

Now when it comes to writing the JUnit test and keeping it technically sound and efficient, is to write code in a way that is self-contained (i.e., can be independent from the entire system) and keep code easy to read and if code can’t self-describe itself, then commenting should be used to explain what’s happening. The following code below shows how to properly setup code for testing starting from the creation of the object, the service used for that object and testing the object and its corresponding service. The last screenshot is testing the coverage of contact

(**Object**)

public class Contact {

private String firstName = "scotty";

private String ID;

public String getFirstName() {  
 return firstName;  
 }

public void setFirstName(String firstName) {  
 if(firstName == null || firstName.length() > 10){  
 throw new IllegalArgumentException("Field cannot be empty or over 10 characters...");

}  
this.firstName = firstName;

}

public String getID() {  
 return ID;  
 }

public Contact(String ID){  
 if (ID == null || ID.length() > 10) {  
 throw new IllegalArgumentException("ID can't be null and can't be greater than 10 characters...");  
 }  
 this.ID = ID;  
 }

public Contact(String ID, String firstName){  
 if (firstName == null || firstName.length() > 10) {  
 throw new IllegalArgumentException("firstName can't be null and can't be greater than 10 characters...");  
 }  
 this.ID = ID;

this.firstName = firstName;  
 }

}

(**Object Service**)

public class ContactService {

List<Contact> contacts = new ArrayList<>();

String uniqueID(){  
 Random rand = new Random();  
 int randomID = rand.nextInt(1000000000)+1;  
 return String.*valueOf*(randomID);  
 }  
  
 void addContact(){  
 Contact contact = new Contact(uniqueID());  
 for(Contact id : contacts){  
 if(id.getID().equals(uniqueID())){  
 throw new IllegalArgumentException("ID Has to be unique to

user...");  
 }  
 }  
   
contacts.add(contact);  
   
System.o*ut*.println("Contact has been added...\n" + "-".repeat(100));  
   
}

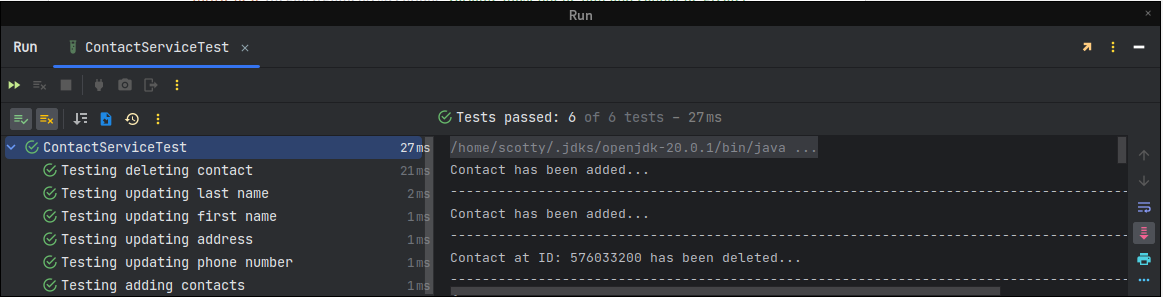
void showAllContacts() {  
 for(Contact contact: contacts){  
 System.*out*.println("{ " +  
 "\n\tID: " + contact.getID() +  
 ", \n\tFirst Name: " + contact.getFirstName()+  
 ", \n\tLast Name: " + contact.getLastName() +  
 ", \n\tPhone Number: " + contact.getPhoneNumber() +  
 ", \n\tAddress: " + contact.getAddress() +  
 "\n}" + ((contacts.indexOf(contact) < contacts.size()-1)?",":""));  
 }  
 }

}

(**Object Testing**)

//*TODO ContactServiceTest ✓*  
*class* ContactServiceTest {;  
 ContactService cs;

@BeforeEach  
 void init(){  
 cs = new ContactService();  
 }  
  
 @Test  
 @DisplayName("Testing adding contacts")  
 void testAddContact() {  
 cs.addContact();  
 cs.showAllContacts();  
 }

@Test  
 @DisplayName("Testing updating first name")  
 void updateFirstName(){  
 cs.addContact();  
 for(Contact contact: cs.contacts){  
 cs.updateFirstName(contact.getID(),"Lisa");  
 Assertions.*assertThrows*(IllegalArgumentException.class, ()->{  
 cs.updateFirstName(contact.getID(),"SomeLongNAmeHere");  
 });  
 }  
 cs.showAllContacts();  
  
 }  
}In my testing, I would keep methods modular for reusability and for independent testing. This technique used was unit testing. I, however, didn’t use integration testing in this project since the project didn’t require a main driver class, and everything was tested independently from one another. Unit testing tests code in small chunks while integration testing requires that you test the entire code together for example, you would make a main class that would call the other classes to work together and view its behavior (*Software Testing Techniques*, 2021). In my mindset, I would create problems that otherwise would need to be solved. For me, since I created the code for the project, I know how to keep the program from crashing, but someone may not. Keeping this in mind, I needed to think like a foreigner to the project and essentially break the program. By doing this, I can find ways to keep the program functioning as intended and lessen time having to refactor code and cut cost and time down due to mitigation of bugs.

**Resources**

Bechtold, S., Brannen , S., Link, J., Merdes , M., Philipp, M., de Rancourt, J., & Stein, C. (n.d.). *JUnit 5 User Guide*. Junit.org. <https://junit.org/junit5/docs/current/user-guide/>

*Software Testing Techniques*. (2021, February 22). GeeksforGeeks. <https://www.geeksforgeeks.org/software-testing-techniques/>#