**Southern New Hampshire University**

Joshua Wozny

[joshua.wozny@snhu.edu](mailto:joshua.wozny@snhu.edu)

June 3, 2023

**CS-320: Journal: Software Testing Techniques**

Developing software requires identifying and implementing system requirements. A requirement is a feature that must exist in a software system to meet the consumer's expectations. Using real-life examples rather than abstract descriptions is often be useful for adding details to the requirement description (Boni Garcia, 2017). Through requirements gathering, we’ve identified that the application requires contact, task, and appointment entities and their associated service to manage each. A EntityService API was created to standardize how specific entity services manage their respective types. The following requirements have been identified for each:

* Entity (abstract Class)
  + All entities must have a String id
  + id may not be null, length can’t be greater than 10 characters
* Entity Service API
  + addEntity(entity) ensuring that no 2 entities within the service have the same id
  + deleteEntity(id)
  + updateEntity(entity) for entity that exists within the service
  + getEntity(id) to return an entity that exists in the service by Id
  + entityExists() to return if an entity exists within the service
* Contact (extends Entity)
  + All contacts must have a non-null String firstName, lastName, phone, and address
  + firstName’s and lastName’s length can’t be greater than 10
  + phone must contain only digits (0-9) and it’s length can’t be greater than 10
  + address’ can’t be greater than 30
* Contact Service
  + Uses EntityService API to add, delete and update contacts
  + updateFirstName(id, newFirstName) where firstName is valid
  + updateLastName(id, newLastName) where lastName is valid
  + updatePhone(id, newPhone) where phone is valid
  + updateAddress(id, newAddress) where address is valid
* Task (extends Entity)
  + All contacts must have a non-null String name and description
  + name’s length can’t be greater than 20
  + description’s length can’t be greater than 50
* Task Service
  + Uses EntityService API to add, delete and update tasks
  + updateName(id, newName) where name is valid
  + updateDescription(id, newDescription) where description is valid
* Appointment (extends Entity)
  + All contacts must have a non-null Date appointmentDate and String description
  + appointmentDate can’t be before the current date
  + description’s length can’t be greater than 50
* Appointment Service
  + Uses EntityService API to add, delete and update appointments
  + updateAppointmentDate(id, newAppointmentDate) where appointmentDate is valid
  + updateDescription(id, newDescription) where description is valid

Unit testing with JUnit is used to ensure that all requirements have been met within our implementation. JUnit is an unit testing framework for Java that can be used to integrate testing into the development process – especially when combined with an IDE and a build system like Maven.Each unit test tests a specific requirement of a class, method, or function. The goal is to ensure that requirements are met, and bugs are caught during the earliest stages of development. Tests are re-run after all changes to the code to ensure that no breaking changes have been made. Generally, the unit tests used in this project have used White Box testing, since the inner workings of the software being tested are known (Boni Garcia, 2017).

In the most recent exercises, developing the contact, task, and appointment services, JUnit was used to ensure all tasks, contacts, appointment, and associated services met requirements. Using JUnit was integral to verifying that these work as intended. Each property of the Entities, contact, task, and appointment was tested to ensure that valid strings were allowed, and that invalid strings were disallowed. This was accomplished through the use of comparisons (id==null, task.description.length()>50, appointment.appointmentDate\ .before(newDate()!=true) to throw exceptions for values outside of the expected range. With 100% code coverage I am confident that these services function as intended.

I used best practices to ensure technically sound code, using inheritance for Contacts, Tasks and Appointments by extending inheriting from Entity, an abstract class used to reuse code for each class that requires an Id. This helped standardized and simplified how Ids are treated.

I also used an EntityService class to standardize how each classes Service manages itself by providing an API for adding, updating, and deleting it’s objects. Regression testing was conducted to ensure that each change (through the addition of new classes or through refactoring) did not break previously functional code. Our unit tests performed integration tests between each entity and the EntityService API through their individual service. Integration testing ensures that individually unit tested components interact with each other as expected (Boni Garcia, 2017).

Using the correct data structures helped to make sure that the code is efficient, using HashMap to store contacts and tasks in their associated services. Using the HashMap structure made the Services easier to implement and makes keeping unique values for Ids straightforward since the Id was used as the key of the HashMap, preventing non-unique values within each service.

Using clean coding practices, good commenting, and self-documenting code through naming conventions, the current code base is in good shape and ready to be added to and get integrated into a larger application. As more code is developed, additional refactoring may be needed to ensure that the application can continue to be extended, and will be easy to understood by other developers. The unit tests will also aid other developers to understand the software requirements as they clearly outline what these classes are designed to do .

Other testing techniques which should be considered to ensure that the implementation is error free and meets client expectations. Static testing with a thorough code review by another, more experienced, developer is a great way to ensure best practices are being used. End-To-End testing using Black Box testing verifies that the entire system as a whole behaves how we expect it to, regardless of the internal implementation (15 testing methods all developers should know | CircleCI, n.d.).

**References**

15 testing methods all developers should know | CircleCI. (n.d.). Circleci. https://circleci.com/blog/testing-methods-all-developers-should-know/?utm\_source=google.

Boni Garcia. (2017). Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language. Packt Publishing.