Kasra Pratt

Prof Springer

CS320

8/11/2025

Grand Strand Systems Summary and Reflections Report

The three features I developed for Grand Strand Systems are the Task Service, Appointment Service, and Contact Service. Each feature includes a core class responsible for data validation, and a service class for managing collections of said core classes. My unit testing approach was focused on testing using JUnit 5 in a white-box manner. I emphasised constructor validation and operational testing for service classes. Tests were isolated, with each test focused on a single behavior. Assertions were checked across expected outcomes, including exceptions. Testing for core classes mainly focused on covering normal paths, boundary conditions, and error scenarios. Testing for services verified state changes in the data collections.

For the Task section, I tested the Task class by creating instances with valid inputs and null values to ensure validation was enforced. For the Task Service, I tested addTask by inserting a valid instance of a task into the service and retrieving it via index; delTask was tested by adding then removing a task and verifying with an IndexOutOfBoundsException on retrieval; and updateTask was tested by modifying the name and description fields post-addition.

For the Appointment section, I tested the Appointment class with valid cases, then with invalid cases like null ID/Description/Date fields, oversized ID/description fields, and past dates using assertThrows for exceptions. For the Appointment Service, I tested addAppointment for successful insertion and duplicate ID rejection, and deleteAppointmentByID for removal and non-existent ID handling.

For the Contact section, I tested the Contact class with valid, null, and invalid length inputs. For Contact Service, I tested append by adding and comparing toString outputs; pop by

adding multiple contacts, removing one by ID, and checking for size reduction; and update by modifying a field like firstName and verifying inequality to the original.

My approach was heavily influenced by the software requirements, which specified constraints such as field lengths, non-null values, and future dates; and service behaviors such as unique IDs, no duplicates, and successful delete/update. For example, in TaskTest.java, the testTaskOverLimit method directly reflects the requirements for ID <=10, name <=20, and desc <= 50 by throwing an IllegalArgumentException on oversized inputs. This covered boundary requirements explicitly, similarly in appointmentTest.java, where testInvalidDateInPast aligns with the “future dates only” requirement. ContactServiceTest.java verified collection management with testPop by asserting true on pop, then checking size equals 0, aligning with unique ID-based deletion. Overall, 100% of requirements were tested with evidence in exception messages like “invalid” matching class validations.

Writing the JUnit tests was straightforward due to the modular design; they achieved 90-95% coverage, covering all if/else paths in validation and service loops with a balance of positive and negative tests, where all constructors have valid/invalid test cases and services test state mutations. I used @Test annotations for isolation and refined my code through iterations of failures.

I employed unit testing, boundary value analysis, and equivalence partitioning. Unit testing isolated components, boundary value analysis tested limits such as string lengths, and equivalence partitioning grouped inputs into valid/invalid classes. I did not employ integration testing, black box testing, or fuzz testing. Integration testing would verify services interact correctly. Black-box treats code as unknown, focusing on inputs and outputs. Fuzz testing inputs random data to find crashes, often for security. Unit testing is ideal for early defect detection for

modular projects. Boundary and equivalence partitioning are better for validation-heavy applications by reducing test volume while implying high confidence for edge cases.

I adopted a detail-oriented mindset as a software testing, meticulously covering all validation paths to avoid any oversights. To limit bias, I relied on objective criteria such as requirements to avoid any assumptions about correctness. For example, I used a random string generator for unbiased oversized inputs instead of hardcoded ones I may favor. I'd also mitigate bias with peer reviews or TDD, but bias could still creep in, like skipping duplicate ID tests in the appointment service if I "knew" the logic. Discipline in quality is crucial for software engineers. It is the foundation to build reliable and maintainable systems; Cutting corners invites bugs that erode user trust and inflate cost.