Software Verification Reflection

Word Count
648

Date
22/08/2024

Lecturer
Christopher Meudec

Student
Mantas Macionis (C00242178)

Academic Year
2023/2024

Introduction

The development process for this iteration focused on creating robust and reliable Gate and Stay classes, with a key objective being to achieve 100% branch coverage through a structured approach: black-box testing, class implementation, and white-box testing. Below is a reflection on the challenges faced, strategies utilized, and insights gained throughout the process.

Black-Box Testing

The first task was creating black-box tests for the Gate and Stay classes without knowing their internal structure. This phase involved anticipating various input scenarios and edge cases. For example, in the Stay class, a critical test case was ensuring the class threw an IllegalArgumentException when entryDateTime was after exitDateTime. Covering edge cases such as this, was crucial in revealing potential flaws in logic that might not surface during standard use.

Using JUnit within the IntelliJ IDE allowed for a standardized approach to structuring the tests efficiently, making it easy to verify expected outcomes. The black-box testing phase was essential in setting a solid foundation for the implementation phase by identifying possible issues early.

Class Implementation

After black-box testing, I moved on to implementing the Gate and Stay classes. Design decisions were guided by SOLID principles, especially the Single Responsibility Principle (SRP). Each class was designed with a specific role in mind, making them modular and testable. For example, the Stay class managed relationships between gates and time intervals, while the validation logic was handled by a separate functional interface, CarParkValidator. This separation reduced coupling and made the code easier to maintain and test.

One of the key challenges during implementation was handling dependencies, particularly in the Stay class, where ensuring that gates belonged to the same car park was necessary. The creation of the CarParkValidator interface allowed for flexible gate validation, highlighting the importance of managing dependencies effectively. Additionally, defensive programming practices, such as validating inputs in constructors, were crucial in preventing the creation of invalid objects that could cause issues later.

White-Box Testing

The most challenging phase was white-box testing to achieve 100% branch coverage. This process required meticulous testing of all possible code paths to ensure that every condition was accounted for.

A particularly memorable challenge was testing the equals method in both classes, which required thorough coverage of various comparison scenarios, such as self-comparison, null comparison, and comparisons with different object types. For example, in the Stay class, tests were created for every possible combination of attributes (entryGate, exitGate, entryDateTime, exitDateTime, and charge) to ensure all logical paths were covered.

Testing constructors also required careful attention, especially in handling invalid inputs. For instance, the Stay constructor was tested with scenarios involving null values and negative charges to ensure that appropriate exceptions were thrown. This testing was critical in validating the robustness of the classes and ensuring that no invalid states could exist.

Insights and Conclusion

This iteration highlighted the importance of thorough testing and validation in software development. Edge cases, often overlooked, proved critical in ensuring the application's robustness. The iterative approach—starting with black-box tests, followed by class implementation, and concluding with white-box tests—was effective in building reliable, maintainable code.

One key takeaway from this process is the value of writing tests before implementation. The black-box tests served as a clear specification of what the classes needed to achieve, guiding the implementation and ensuring that the resulting code was robust. White-box testing then acted as a final check, ensuring no logical paths were left untested.

This experience of achieving 100% branch coverage will significantly influence my future development practices. Specifically, it has reinforced the importance of integrating testing throughout the development lifecycle, not just as a final step. Moving forward, I plan to utilize a more rigorous test-driven development (TDD) approach, writing tests before code to ensure that the code is designed to be testable from the outset.