University Parking System: Using the Observer Pattern to Assess Parking Charges

for

Master of Science

Information Technology

Kassandra Vega Lucero

University of Denver College of Professional Studies

May 4, 2025

Faculty: Nathan Braun, MBA

Dean: Michael J. McGuire, MLS

Table of Contents

Background

The University is developing a parking system for students and community members. Users must register with the University Parking Office to use the parking lots. The system takes several factors into account when calculating the user's charges, such as the type of car used, the kind of charge used for the parking lot, and other factors. The enhancement added to the parking system during this week's development was the observation pattern. The observation pattern is used to assess the parking charges. Other design changes were also implemented to gear towards a more efficient system, along with the enhancement.

Class Implementations

Adding the observer pattern meant adding a ParkingLotObserver class as well as making changes to the TransactionManager and ParkingLot classes. The ParkingLot is the observable, which means it maintains a list of observers as well as any triggering events. The ParkingObserver is the observer, which means it listens to any events and interacts with the TransactionManager. The TransactionManager serves as what seems to be an independent actor that processes the parking transactions based on any events received by the observers. Implementing the system aims to attain a decoupled system and ensure flexibility for future implementations.

The classes were implemented in separate packages to maintain similar classes together, making it easier to understand and make future changes. Using examples shared in the class discussion, such as the one I found on The Code Bean (2023), as a reference for the structure, I was able to modify the basic structure until the observers worked. In prior developments, TransactionManager had a different structure and layout. The changes were made to ensure proper usage due to previous issues I had with compilations. To aid in implementing the observer pattern I also used a blog by Salas (2023) who dives into explanations and examples of the observer pattern. It was easier to visualize and locate errors when I compared the parking system to the weather example and the publisher analogy.

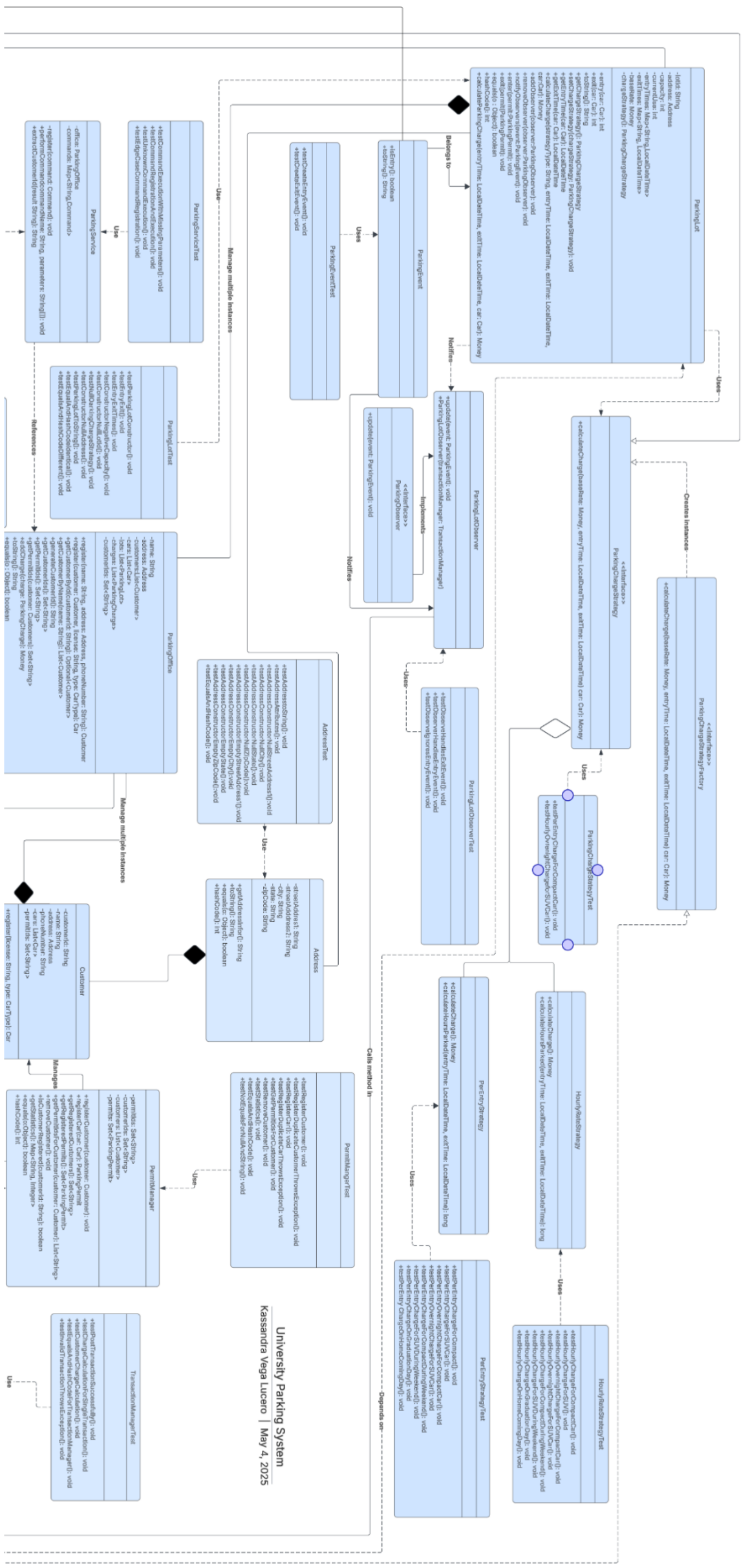


Figure 1. UML Diagram of the updated University Parking System (top half).

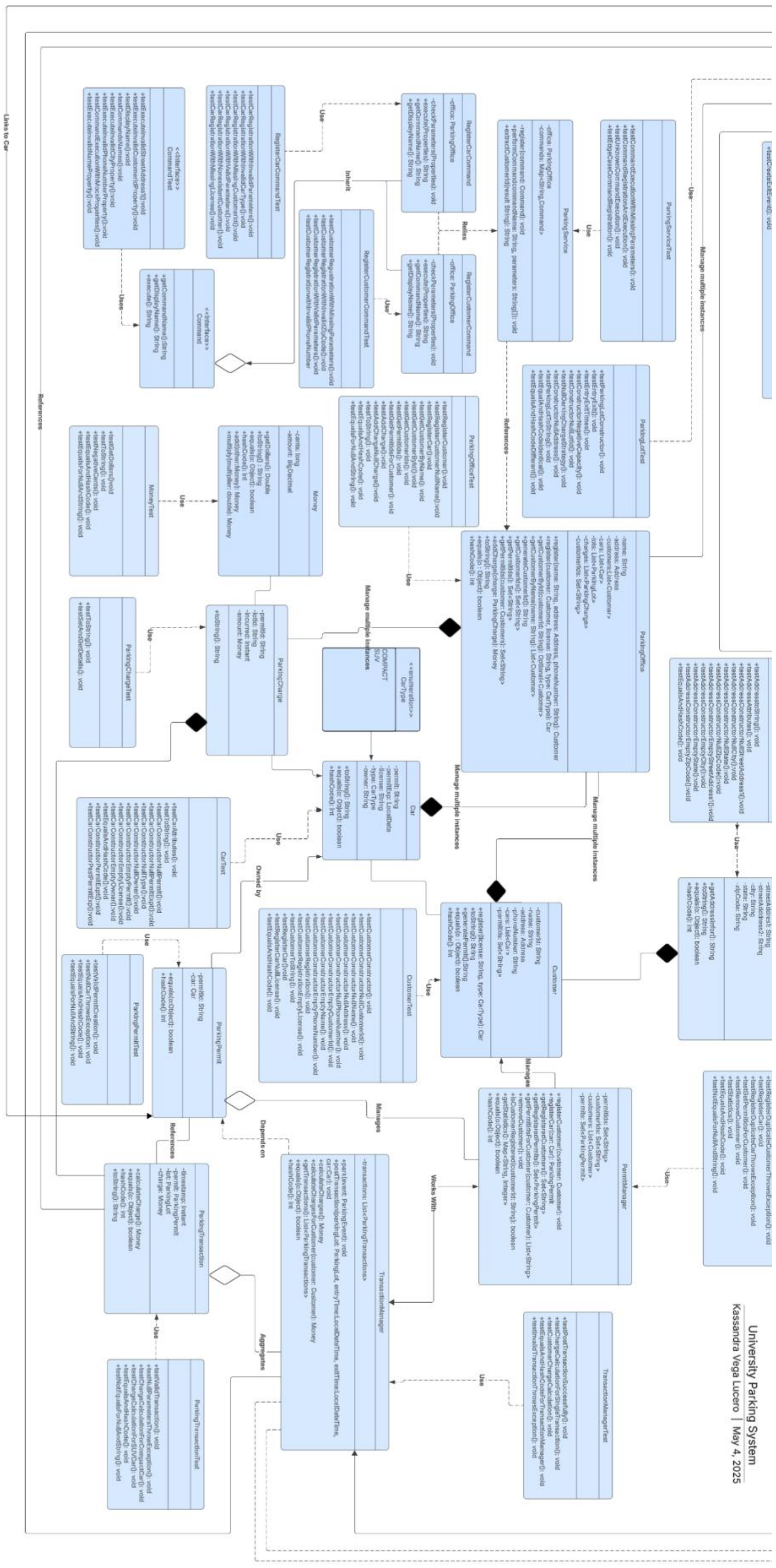


Figure 2. UML Diagram of the updated University Parking System (bottom half).

Implementation Challenges

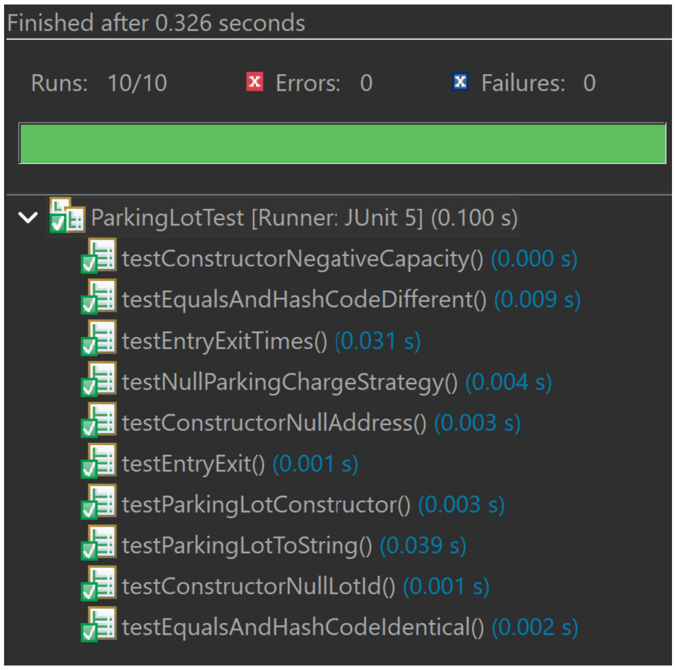
Developing the system's enhancements involved several challenges. The first challenge was getting the system to compile again. Over the last two weeks, compilation issues have been recurring, and it has felt impossible to troubleshoot. After trying several things, something must have clicked that got the system to compile, but the origin or cause is still unidentified. After the parking system was compiled, the next challenge that needed to be overcome was the added bugs that went undetected due to the prior issue. This portion took about eight hours, since the debugging process also changed the design structure for the TransactionManager and parts of the ParkingLot class.

The development of the Observer pattern involved quite a bit of trial and error. My initial attempt at creating the observation pattern and its relationships with the ParkingLot and TransactionManager classes involved creating transactions without using the parking charge strategies. The approach I tried at first also involved nesting too many different classes when trying to calculate the charges, which made the expected values differ from the resulting values. Through the unit tests, I was also able to uncover another difficult bug.

When testing, I saw an example of using @BeforeEach in the testing file. When I applied this to the TransactionManagerTest class, I noticed that I was getting a null argument exception. This led to creating and running several print statements to deduce where the issue originated. While the values were being created properly, including them within the @BeforeEach was causing the values to be converted back to null values. To resolve the issue, I included the implementation within each test. This resolved the problem, but it was a bit more tedious to use.

Unit Test and Visuals

The images below show a couple of tests that were ran to ensure new implementations as well as some of the updates made to prior classes still resulted in successful tests. Some of the tests picture below such as Figure 3 and Figure 4, are tests that were not successful last week. Since the ParkingEvent class was added to facilitate parking transactions, there are tests to test the functionality of a car entering and exiting the lot. This is later testing in conjunction with the observer to ensure a car entering would not cause a transaction to be recorded. Transactions should only be recorded after a car leaves in case the car stayed overnight.

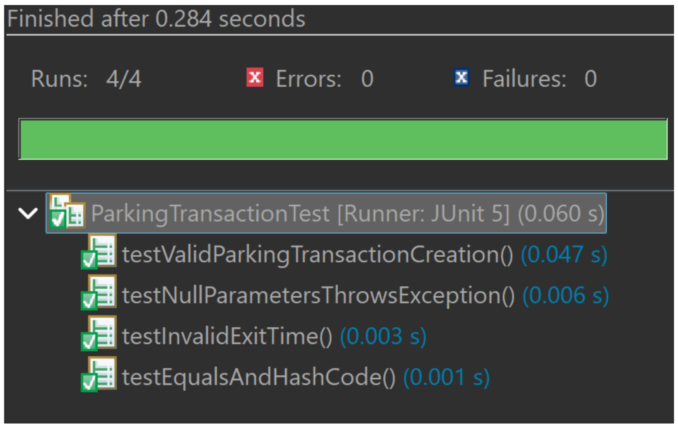


```
Finished after 0.326 seconds

Runs: 10/10        Errors: 0        Failures: 0

ParkingLotTest [Runner: JUnit 5] (0.100 s)
    testConstructorNegativeCapacity() (0.000 s)
    testEqualsAndHashCodeDifferent() (0.009 s)
    testEntryExitTimes() (0.031 s)
    testNullParkingChargeStrategy() (0.004 s)
    testConstructorNullAddress() (0.003 s)
    testEntryExit() (0.001 s)
    testParkingLotConstructor() (0.003 s)
    testParkingLotToString() (0.039 s)
    testConstructorNullLotId() (0.001 s)
    testEqualsAndHashCodeIdentical() (0.002 s)
```

Figure 3. JUnit results for ParkingLotTest.java

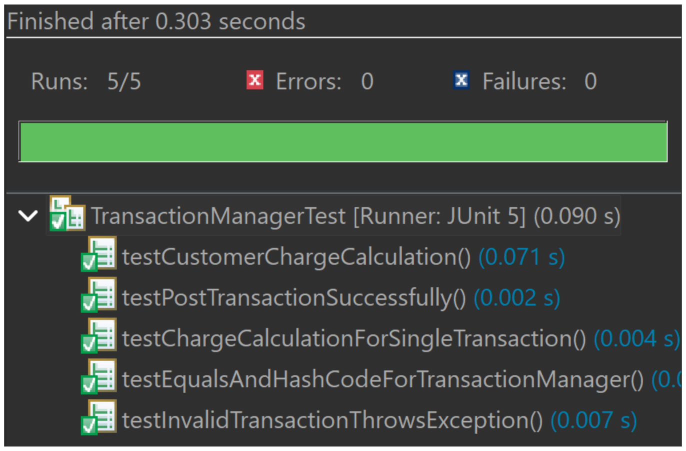


```
Finished after 0.284 seconds

Runs:  4/4        Errors:  0        Failures:  0

ParkingTransactionTest [Runner: JUnit 5] (0.060 s)
    testValidParkingTransactionCreation() (0.047 s)
    testNullParametersThrowsException() (0.006 s)
    testInvalidExitTime() (0.003 s)
    testEqualsAndHashCode() (0.001 s)
```

Figure 4. JUnit results for ParkingTransactionTest.java are working again.

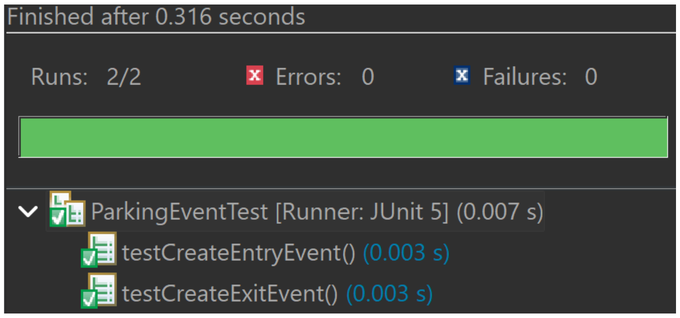


```
Finished after 0.303 seconds

Runs:  5/5        Errors:  0        Failures:  0

TransactionManagerTest [Runner: JUnit 5] (0.090 s)
    testCustomerChargeCalculation() (0.071 s)
    testPostTransactionSuccessfully() (0.002 s)
    testChargeCalculationForSingleTransaction() (0.004 s)
    testEqualsAndHashCodeForTransactionManager() (0.
    testInvalidTransactionThrowsException() (0.007 s)
```

Figure 5. JUnit results for TransactionManagerTest.java are working again.

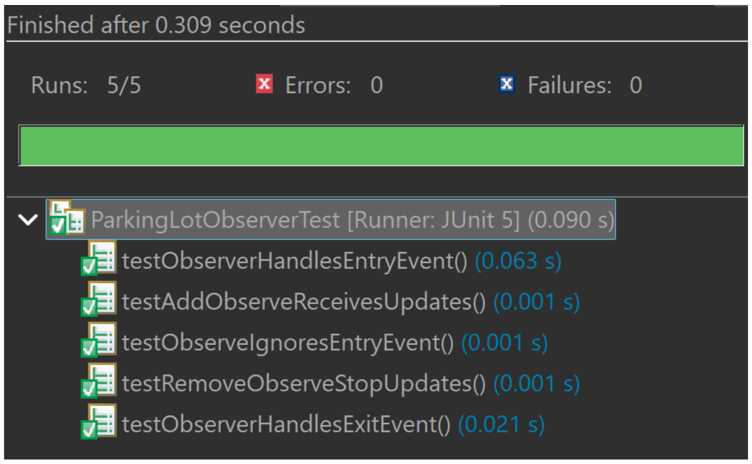


```
Finished after 0.316 seconds

Runs:  2/2        Errors:  0        Failures:  0

ParkingEventTest [Runner: JUnit 5] (0.007 s)
    testCreateEntryEvent() (0.003 s)
    testCreateExitEvent() (0.003 s)
```

Figure 6. JUnit results for ParkingEventTest.java



```
Finished after 0.309 seconds

Runs:  5/5        Errors:  0        Failures:  0

ParkingLotObserverTest [Runner: JUnit 5] (0.090 s)
    testObserverHandlesEntryEvent() (0.063 s)
    testAddObserveReceivesUpdates() (0.001 s)
    testObserveIgnoresEntryEvent() (0.001 s)
    testRemoveObserveStopUpdates() (0.001 s)
    testObserverHandlesExitEvent() (0.021 s)
```

Figure 7. JUnit results for ParkingLotObserverTest.java

Reflection

Implementing the observer pattern in the parking system was a bit of a challenge. Once it was up and running, I saw the benefits of using it to maintain the system's organization by allowing for triggers to be introduced and opening the door to the use of features for user design. While I am not too comfortable with implementing the pattern from memory, I do anticipate using it again in the future.

References

Salas, Fernando. 2023. “Exploring Design Patterns: Inside the Observer Pattern.” Medium. Published August 15, 2023. https://blog.stackademic.com/exploring-design-patterns-inside-the-observer-pattern-d4416c829370.

The Code Bean. 2023. "Observer Design Pattern: Implementation in Java." Medium. Published October 13, 2023. https://medium.com/@thecodebean/observer-design-pattern-implementation-in-java-d7d263fbd0e3.