University Parking System: Implementing a Creational Pattern (Factory)

for

Master of Science

Information Technology

Kassandra Vega Lucero

University of Denver College of Professional Studies

April 27, 2025

Faculty: Nathan Braun, MBA

Dean: Michael J. McGuire, MLS

Table of Contents

Background 3

Class Implementations 3

Implementation Challenges 6

Unit Tests 6

Unit Test Visuals 7

Development Challenges and Reflection 10

References 11

Background

The University is constantly adding enhancements to develop a parking system that allows students and community members to register with the University's Parking Office to park in their parking lots. The parking system's development has been completed in stages, with this stage creating and implementing a factory, which is a type of creational pattern. The factory takes a superclass that has multiple subclasses and returns a single charge strategy subclass.

Class Implementations

The factory implementation builds on the prior week's parking charge strategies. There is the HourlyRateStrategy which calculates the hours spent in the parking lot then multiplies them by the baseRate. The strategy then goes through the remaining factors rendering discounts and surcharges. The PerEntryStrategy functions similar to the hourly—only there is no hourly calculation. The strategy uses a fixed entry charge that is adjusted based on the factors considered. The factors considered in the parking strategies are car type, day of the week, special days, and overnight stays.

Since each parking lot can have different methods for how the charges are accrued, the system need a way to create different parking charge strategies without having to make extensive changes to the system each time. Once strategies are made such as the HourlyRate and PerEntry indicated above, the system has the factory provide the correct strategy base on the string used in the parameter. The current strategies use the strings "HOURLY" and "PER_ENTRY" to identify the strategy. There is also ParkingChargeStrategyFactoryImpl which implements the factory interface. As long as a valid strategy is used, it will implement or use the creation logic. If the strategy used in the parameter is invalid, an argument exception that is thrown.

Once properly implemented, the TransactionManager and ParkingLot classes work together to accrue the appropriate charges. When a parking charge is created, the Transaction manager class should have the factory indicate the parking strategy to use. When calculating the charges for a parking lot, the factory is used to indicate the parking strategy that would be used for the charge calculation, and then the ParkingChargeStrategy interface would follow through to calculate the charge.

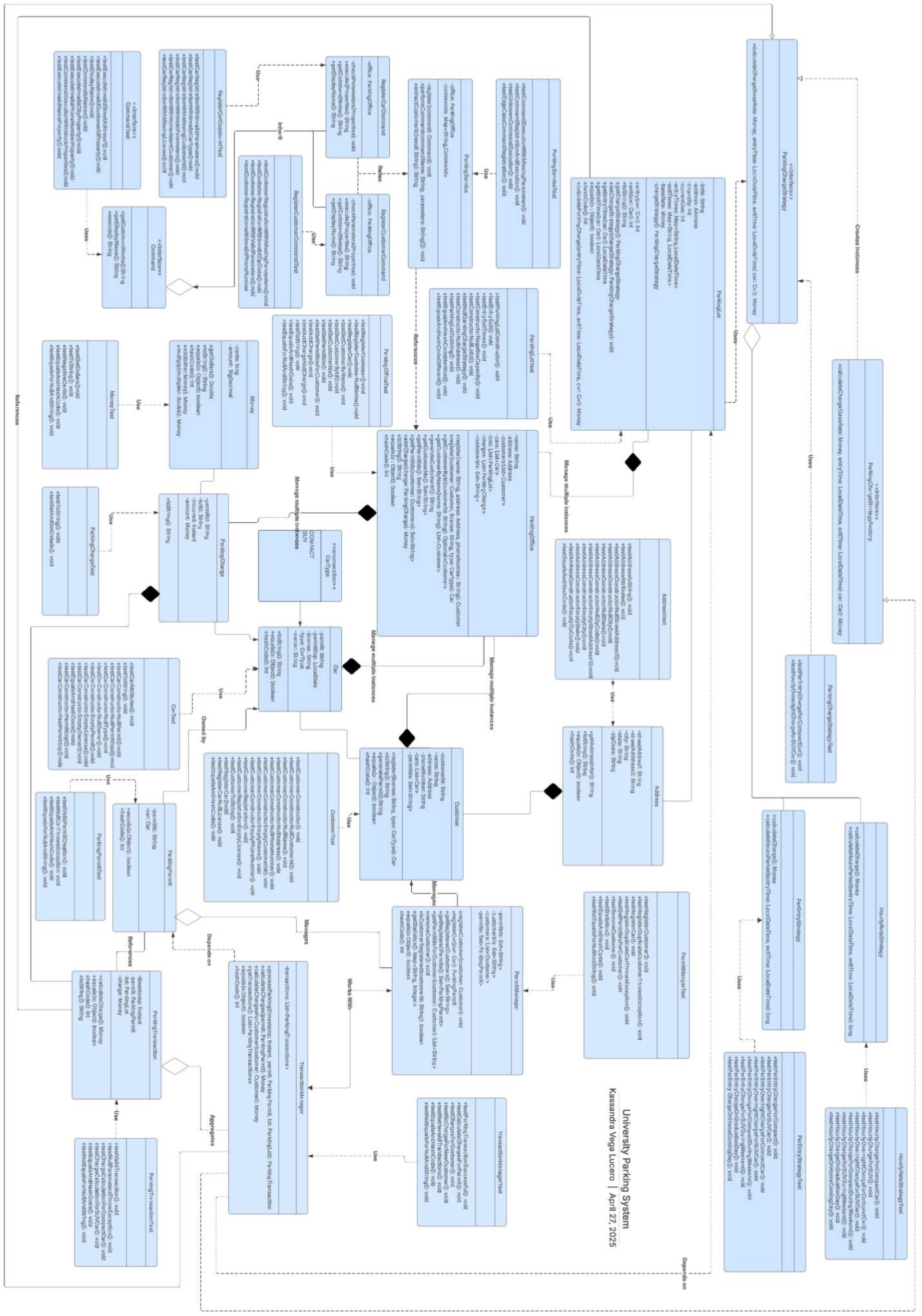


Figure 1. UML Diagram of updated University Parking System.

Implementation Challenges

The challenges I faced while implementing the factory was having to learn what a factory method was. After learning, it seemed fairly simple to implement. The examples I found used a main method approach to show the implementation of the factory. Since this was a newer topic for me, I thought it had to be implemented in a main method, and was overthinking the testing approaches for the factory. Eventually, I realized the factory could be implemented very similarly to the other classes. When it came to testing, the other challenges I faced were incorporating the factory into the TransactionManager as well as the ParkingLot.

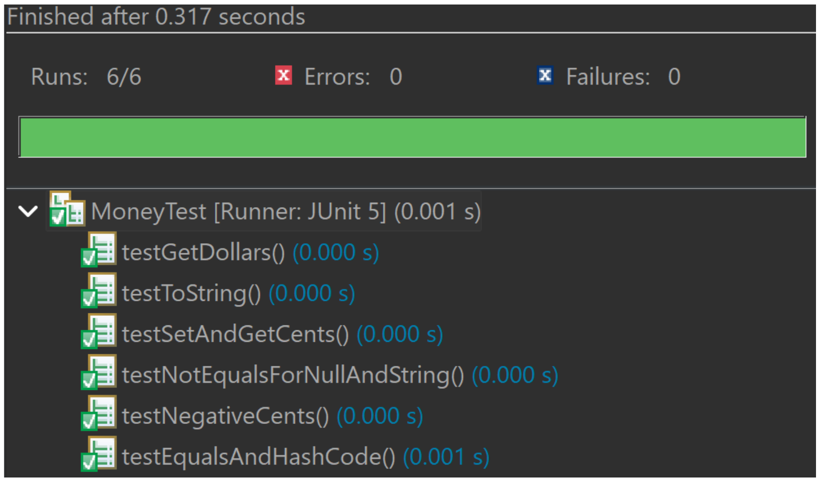
Due to my prior challenges getting my other classes to compile, it was not possible to test whether my code was working as I intended it to. I spent over 12 hours working solely on troubleshooting components in those classes. After earlier guidance, I ensured there were no typos in my imports. I also attempted alternative methods for the existing code to see if there was a workaround. My research has led me to narrow it down to an error with my JDK causing an incomplete build path and issues iterating through an ArrayList for the transactions.

Unit Tests

With every code addition, more tests must be added for smooth operations. There were already tests for the parking charge strategies, so I added tests for the factory. This required testing to identify the correct parking strategy for the factory. Tests were added for the TransactionManager and ParkingLot, with no concrete way of telling whether they worked or not due to the compilation issues. I removed a test for the ParkingLot class that was not working due to a connection with the TransactionManager error.

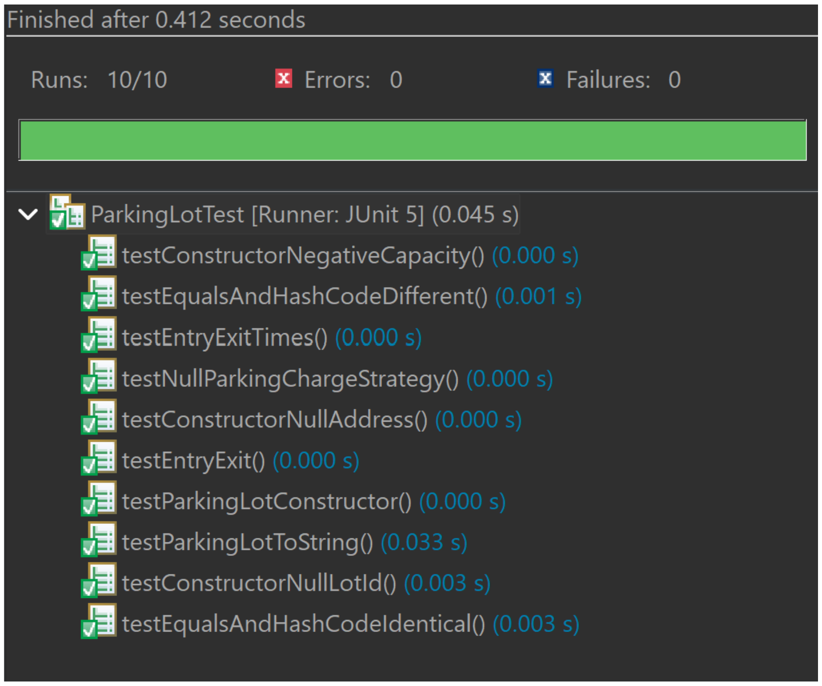
Unit Test Visuals

The images below show a combination of successful unit tests for the classes updated in or implemented for the parking charge calculator as well as those that were inconclusive based on the systems compilation status.



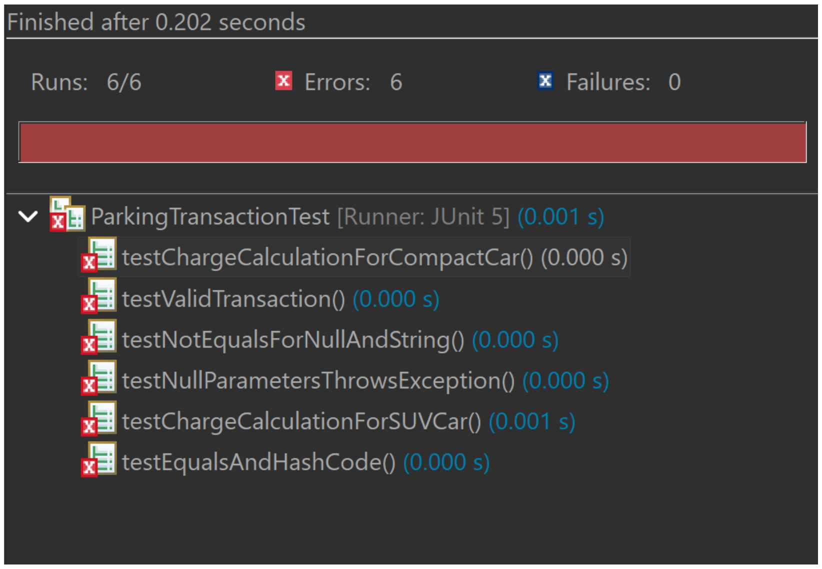
```
Finished after 0.317 seconds

Runs:  6/6          Errors:  0          Failures:  0

MoneyTest [Runner: JUnit 5] (0.001 s)
    testGetDollars() (0.000 s)
    testToString() (0.000 s)
    testSetAndGetCents() (0.000 s)
    testNotEqualsForNullAndString() (0.000 s)
    testNegativeCents() (0.000 s)
    testEqualsAndHashCode() (0.001 s)
```

Figure 2. JUnit results for Money.java



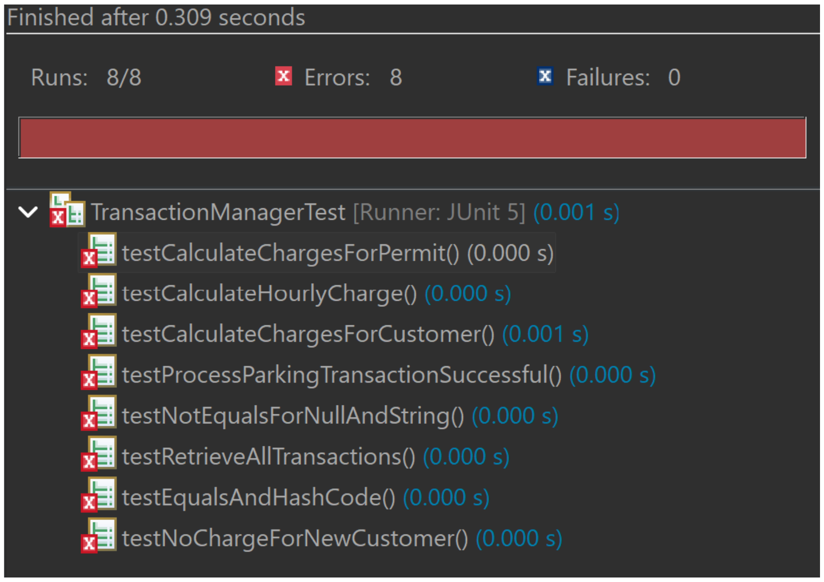
```
Finished after 0.412 seconds

Runs:  10/10        Errors:  0          Failures:  0

ParkingLotTest [Runner: JUnit 5] (0.045 s)
    testConstructorNegativeCapacity() (0.000 s)
    testEqualsAndHashCodeDifferent() (0.001 s)
    testEntryExitTimes() (0.000 s)
    testNullParkingChargeStrategy() (0.000 s)
    testConstructorNullAddress() (0.000 s)
    testEntryExit() (0.000 s)
    testParkingLotConstructor() (0.000 s)
    testParkingLotToString() (0.033 s)
    testConstructorNullLotId() (0.003 s)
    testEqualsAndHashCodeIdentical() (0.003 s)
```

Figure 3. JUnit results for ParkingLotTest.java



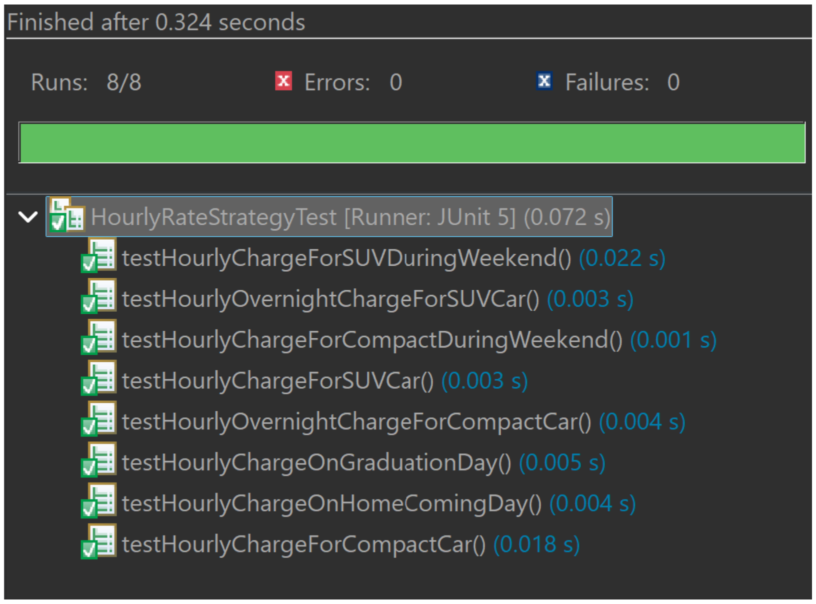
```
Finished after 0.202 seconds

Runs:  6/6          Errors:  6          Failures:  0

ParkingTransactionTest [Runner: JUnit 5] (0.001 s)
    testChargeCalculationForCompactCar() (0.000 s)
    testValidTransaction() (0.000 s)
    testNotEqualsForNullAndString() (0.000 s)
    testNullParametersThrowsException() (0.000 s)
    testChargeCalculationForSUVCar() (0.001 s)
    testEqualsAndHashCode() (0.000 s)
```

Figure 4. JUnit results for ParkingTransactionTest.java which is no longer working.



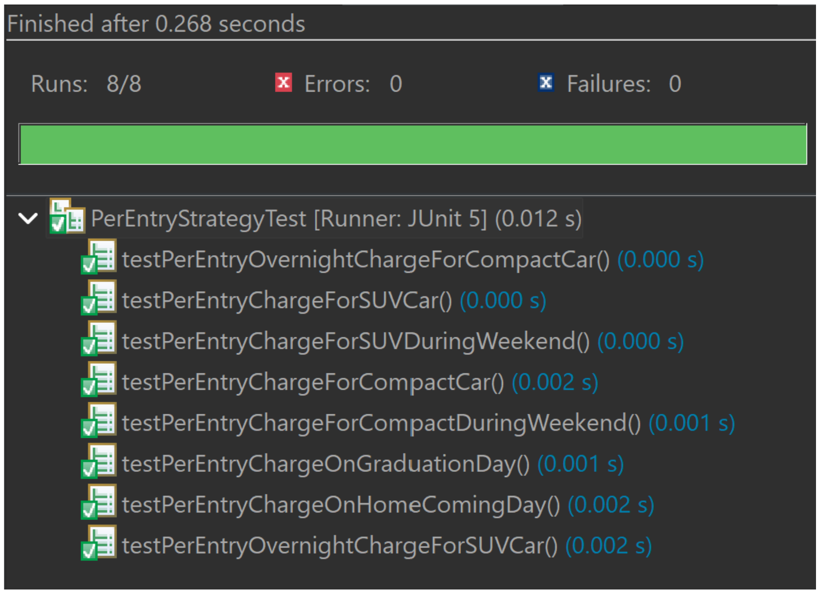
```
Finished after 0.309 seconds

Runs:  8/8          Errors:  8          Failures:  0

TransactionManagerTest [Runner: JUnit 5] (0.001 s)
    testCalculateChargesForPermit() (0.000 s)
    testCalculateHourlyCharge() (0.000 s)
    testCalculateChargesForCustomer() (0.001 s)
    testProcessParkingTransactionSuccessful() (0.000 s)
    testNotEqualsForNullAndString() (0.000 s)
    testRetrieveAllTransactions() (0.000 s)
    testEqualsAndHashCode() (0.000 s)
    testNoChargeForNewCustomer() (0.000 s)
```

Figure 5. JUnit results for TransactionManagerTest.java which is no longer working.



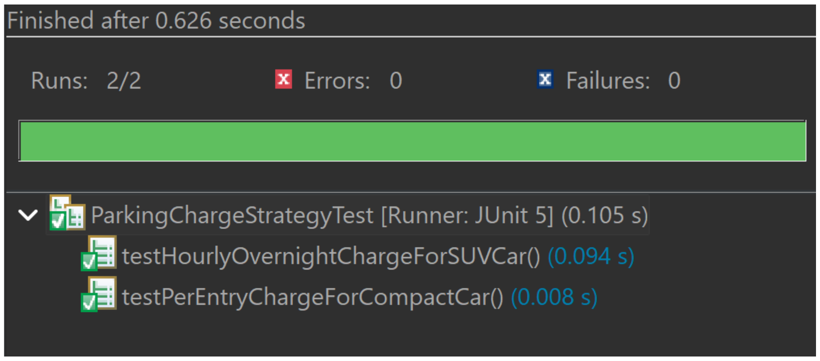
```
Finished after 0.324 seconds

Runs:   8/8          Errors:   0          Failures:   0

HourlyRateStrategyTest [Runner: JUnit 5] (0.072 s)
    testHourlyChargeForSUVDuringWeekend() (0.022 s)
    testHourlyOvernightChargeForSUVCar() (0.003 s)
    testHourlyChargeForCompactDuringWeekend() (0.001 s)
    testHourlyChargeForSUVCar() (0.003 s)
    testHourlyOvernightChargeForCompactCar() (0.004 s)
    testHourlyChargeOnGraduationDay() (0.005 s)
    testHourlyChargeOnHomeComingDay() (0.004 s)
    testHourlyChargeForCompactCar() (0.018 s)
```

Figure 6. JUnit results for HourlyRateStrategyTest.java



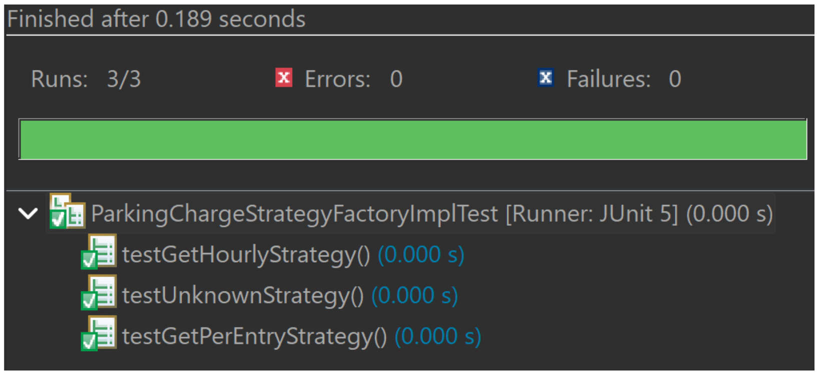
```
Finished after 0.268 seconds

Runs:   8/8          Errors:   0          Failures:   0

PerEntryStrategyTest [Runner: JUnit 5] (0.012 s)
    testPerEntryOvernightChargeForCompactCar() (0.000 s)
    testPerEntryChargeForSUVCar() (0.000 s)
    testPerEntryChargeForSUVDuringWeekend() (0.000 s)
    testPerEntryChargeForCompactCar() (0.002 s)
    testPerEntryChargeForCompactDuringWeekend() (0.001 s)
    testPerEntryChargeOnGraduationDay() (0.001 s)
    testPerEntryChargeOnHomeComingDay() (0.002 s)
    testPerEntryOvernightChargeForSUVCar() (0.002 s)
```

Figure 7. JUnit results for PerEntryStrategyTest.java



```
Finished after 0.626 seconds
Runs:  2/2        Errors:  0        Failures:  0
ParkingChargeStrategyTest [Runner: JUnit 5] (0.105 s)
    testHourlyOvernightChargeForSUVCar() (0.094 s)
    testPerEntryChargeForCompactCar() (0.008 s)
```

Figure 8. JUnit results for ParkingChargeStrategyTest.java



```
Finished after 0.189 seconds
Runs:  3/3        Errors:  0        Failures:  0
ParkingChargeStrategyFactoryImplTest [Runner: JUnit 5] (0.000 s)
    testGetHourlyStrategy() (0.000 s)
    testUnknownStrategy() (0.000 s)
    testGetPerEntryStrategy() (0.000 s)
```

Figure 9. JUnit results for ParkingChargeStrategyFactoryImplTest.java

Development Challenges and Reflection

After navigating the challenges, I am more confused than when I began regarding my prior implementation and why they now have so many errors. Although I got some components from the factory to test successfully, the failures in the other tests took a toll on my confidence in having a successful system by the end of this class. I want to make many changes to my code to enhance exceptions and make it cleaner, but with my limited time between jobs, doing so may be a challenge.

References

Copilot User. 2023. "Java Evolution: Builder Misuses and Modern Alternatives." Copilot User. Published June 1, 2023. https://copilotuser.com/java/builder-pattern/modern-practices/2023/06/01/java-evolution-modernizing-the-builder-pattern.html.

Geekific. 2021. “The Factory Method Pattern Explained and Implemented in Java | Creational Design Patterns | Geekific.” Published May 29, 2021. YouTube tutorial video. https://www.youtube.com/watch?v=EdFq_JIThqM&t=49s.

GeeksforGeeks. 2023. “Factory Method Design Pattern in Java.” GeeksforGeeks. Updated January 3, 2025. https://www.geeksforgeeks.org/factory-method-design-pattern-in-java/.