

**Week 4 Assignment: Implementing a Creational Pattern: Factory for**

ICT 4315-1 Object-Oriented Methods & Paradigms II

Jack Hermanson

University of Denver University College

April 27, 2025

Faculty: Nathan Braun, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

This paper discusses the introduction of the factory pattern to the parking lot solution. The factory pattern allowed me to create a “virtual constructor” for the different parking charge strategies that vary based on the lot type (Refactoring Guru 2025). The benefit of this is that, when it’s time to create a parking transaction, the transaction manager will use a factory to get the right type of strategy to produce a transaction with the correct amount charged. The transaction manager itself does not need to worry about the logic behind determining which strategy to use; the factory does this. The logic is abstracted away with any complexity.

I added a method onto the Transaction Manager class named `park`, which accepts a permit (which was scanned entering or exiting a lot), the lot in question, an entry time, and a nullable exit time. This method creates a new Parking Charge Strategy Factory, although in a more advanced system this would probably be dependency-injected into the Transaction Manager class as a singleton, so the `park` method wouldn’t need to create a new one each time. Since the factory doesn’t have any data members, it would also make sense to simply make it static.

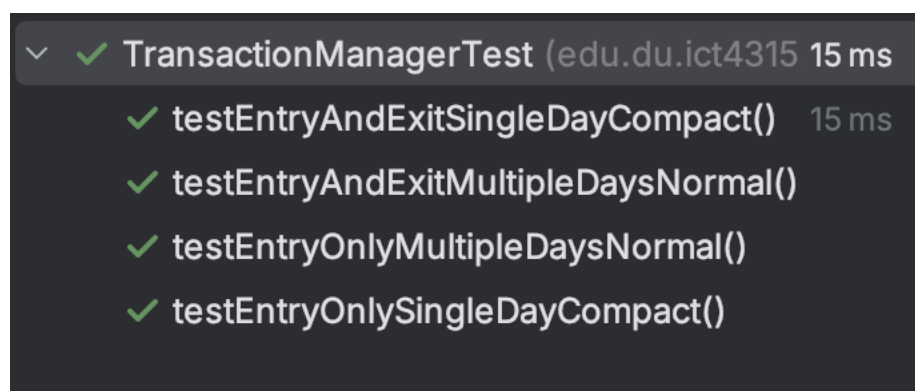
The factory then performs logic to determine which type of strategy to return to the transaction manager based on the type of lot. For added flexibility, I wrote an override, so the programmer can pass in the parking lot type or the entire lot itself into the factory and get the same outcome.

The transaction manager’s `park` method uses that factory to get a strategy. It doesn’t know what type of strategy it needs to use, but it doesn’t need to. The factory created an

instance of a concrete class that implements the strategy's interface, so all it has to do is call the calculate charge method on the concrete class and generate a transaction.

This assignment helped me learn about the factory pattern, and it also made me think about using the builder pattern, but I will save that for later. I think the builder pattern could be useful for situations where certain values should be null, like exit time for an entry only lot. One way I problem-solved was by adjusting the factory from an interface to a concrete class. The instructions said it should be an interface, but I couldn't figure out how to do that in a way that made sense with my code, so I decided to make it a concrete class. Most material I used to help me learn about the factory method showed examples where there were multiple different types of factories implementing an interface, like the strategy pattern. I'm not sure what the best way to do that would be here, without it being too redundant or complex.

All tests pass:



```
✓ TransactionManagerTest (edu.du.ict4315 15 ms)
  ✓ testEntryAndExitSingleDayCompact() 15 ms
  ✓ testEntryAndExitMultipleDaysNormal()
  ✓ testEntryOnlyMultipleDaysNormal()
  ✓ testEntryOnlySingleDayCompact()
```

## References

Refactoring Guru. 2025. *Factory Method*. Accessed April 27, 2025.

<https://refactoring.guru/design-patterns/factory-method>.