Implementing a Creational Pattern: Factory

for

Master of Science

Information Technology

Peter Fedor

University of Denver College of Professional Studies

April 27, 2025

For this assignment I was tasked with implementing a factory creational pattern for the previous weeks calculator. This was achieved by introducing the ParkingChargeStrategyFactory, which creates instances of the StrategyInterface interface. The goal of adding this is to add to the existing functionality of the code by allowing the dynamic selection of the pricing strategy based on the lot. This enables the expansion of the code by allowing the addition of other pricing strategy without changing the overall structure of the existing code.

Design

For my design of the factory, I added a new factory which handles the creation of the StrategyInterface instances. The design ultimately included the StrategyInterface, HourlyCharge and DailyCharge implementations, ParkingLot class, and the transaction manager. The factory pattern essentially allowed for the dynamic selection of charge types. This is where the HourlyCharge and DailyCharges comes in since they are both represented through the StategyInterface which determines the charge type for a certain parking lot. This in turn eliminated the need for assigning pricing strategies directly to the transaction manager, making the code overall more decoupled. This allows for the future expansion of the program, as it can now use different pricing strategies that are not as tightly tied to the rest of the program. I did have issues figuring out how to tie the program together as a whole since I was adding new functionality to it. Since there is no list of existing lots and their determined costs or types, I got a bit lost on how these instances would actually reflect in the working program. This leads to the next portion, my challenges.

Challenges

Once again, I found this assignment to be a bit challenging. Since factory design patterns are new to me, I still do not know if I implemented them in the most effective way. For example, determining if the correct strategy was used for each lot made sense in design, but getting my classes be able to select each type or pricing took me a while to get correct. This all boils down to how difficult it was to refactor my code to accept the new changes. Since my program written in object oriented one was not the exact model we are using in this class, there were some hold ups. Overall, I was able to work through them, and it helped me better understand the material.

Conclusion

The updates made to the code introduce a new level of scalability. Parking charges are now abstracted, allowing for new strategies to be added, expanding the functionality of the system. For example, if I wanted to add a new lot, with a different pricing strategy, I could simply write the strategy for that pricing model and use it wherever I need. I now understand the full potential of using factory design patterns in my code. When used properly, it allows programmers to cleanly integrate new bits of code into existing code while maintaining the integrity of the program.

References

"Design Patterns: Strategy in Java." 2014. Refactoring.guru. 2014. https://refactoring.guru/design-patterns/strategy/java/example.

Refactoring Guru. 2014. "Factory Method." Refactoring.guru. 2014. https://refactoring.guru/design-patterns/factory-method.

"Strategy Pattern | Set 1 (Introduction) - GeeksforGeeks." GeeksForGeeks. 2016. GeeksforGeeks. April 29, 2016. https://www.geeksforgeeks.org/strategy-pattern-set-1/.