Week 5 Assignment: Using the Observer Pattern to Assess Parking Charges for

ICT 4315-1 Object-Oriented Methods & Paradigms II

Jack Hermanson

University of Denver University College

May 4, 2025

Faculty: Nathan Braun, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

This paper discusses the implementation of the observer pattern in the parking lot system. The observer pattern, at a high level, involves two roles: publishers and subscribers (Refactoring Guru 2025). In this assignment, I named them with the convention of observable and observers, respectively. Classes that are observable maintain a list of observers, and when something happens, it notifies those observers.

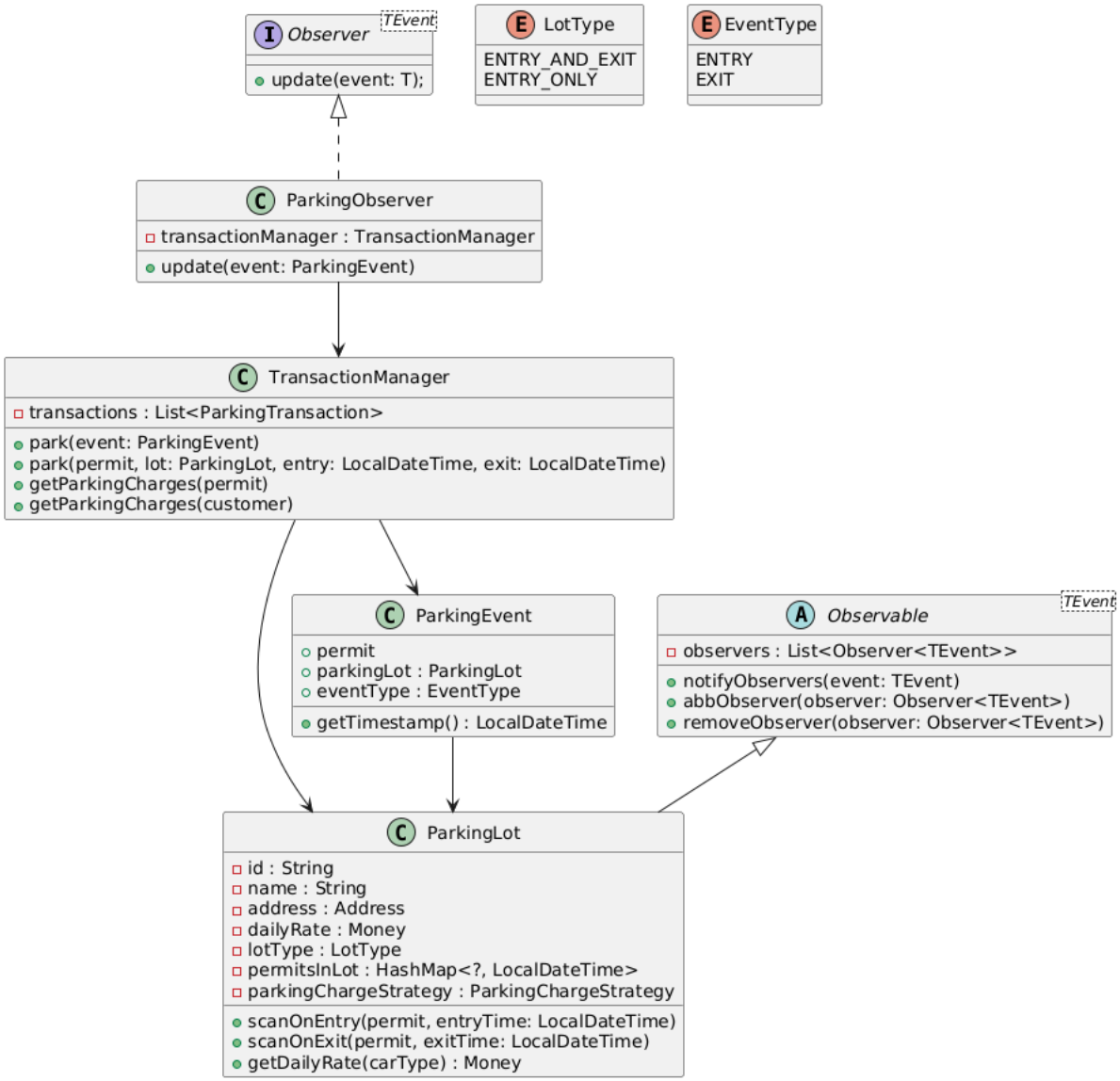
I started by creating an interface for Observers. This is a simple class that takes a generic parameter, TEvent, to tell it what type of events it expects to be notified about. This type is used in its only method, update, which accepts an event of that type.

The Observable class takes that same generic parameter so it knows what types of events its observers are expecting. It has a list of observers, methods to add and remove an observer, and a method to notify all observers. The notify observers method takes an event of type TEvent, as previously mentioned, and it knows that all of its observers are able to handle that type of event.

This all set the stage for modifying the Parking Lot class to extend Observable. Once I had the Parking Lot class extend observable, the observable pattern methods were already in place for me, and I did not have to manually implement anything specific for that class regarding notifying observers. This is handy because other classes in the future can also extend the Observable class and do the same thing without duplicate code. The only modifications I needed to make were to add functionality in the scan on exit/entrance methods to notify subscribers of a parking event. Now all observers know when something occurs. These observers could be pieces of functionality that notify a user when their permit has been

charged, add charges to a user's statement, update a database for statistical tracking, update a counter outside the parking lot to let visitors know how many spots remain, etc.

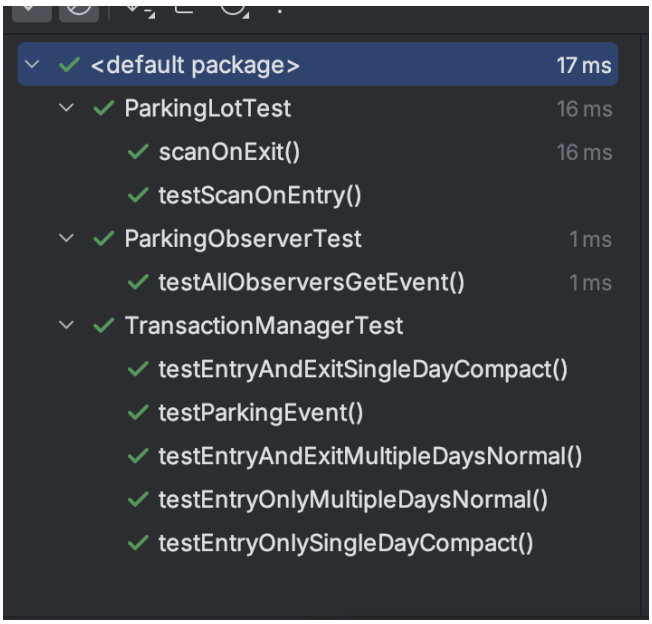
When the parking observer receives an event, it passes it along to the transaction manager so that it can record a new transaction for this event if necessary. Other components of the system could implement the update method to do other things as previously mentioned, like notify the user or update a display.



The UML class diagram shows that the Parking Observer class implements the Observer class and its update method. It also has a reference to the Transaction Manager so that it can do transactions when the update method is called. The Parking Lot class extends the Observable class, so it automatically has a list of observers and methods to add, remove, and notify them. Both the Observable and Observer classes take a generic parameter for event type.

One challenge I ran into was feeling like the code I was writing for Parking Lot to become observable was too specific and should be generalized. I started writing the methods on it, then decided to create an abstract class for Observable, which would provide Parking Lot with all of its benefits right out of the box. This felt like a clean solution and extensible for other observers in the future.

The code compiles and all tests pass:



References

Refactoring Guru. 2025. "Observer." Accessed April 30, 2025. https://refactoring.guru/design-patterns/observer.