**ICT 4315: Object Oriented Methods and Programming II**

**Portfolio Assignment**

Master of Science

Information and Communications Technology

Daphne Monroe Goujon

University of Denver University College

June 3, 2022

Faculty: Michael Schwartz

Director: Cathie Wilson, M.S.

Dean: Michael J. McGuire. MLS

**Abstract**

The following paper is a presentation of the Parking System Application I have created for ICT 4315, Object Oriented Methods and Programming II. This presentation will walk through the application's intended use and requirements, the implementation decisions made to meet those requirements, what I have learned from this project and the difficulties I faced in implementation, and a reflection to discuss what I may have done differently and enhancements I would make to the project.

My Parking System Application was created in Eclipse IDE. To provide visual aid for this presentation, I have included UML diagrams and screenshots from my Eclipse environment, in addition to discussion about Eclipse's capabilities and plugins that aided in creating this application.

## Table of Contents

## Introduction

The assignment instructions for the Parking System Portfolio Assignment were as follows:

The University has a need to develop an Object-Oriented Parking System. Each assignment will build towards creating the parking system.

- The University has several parking lots and the parking fees are different for each parking lot.
- Some parking lots scan permits only on entry. Charge is on entry, plus daily charges if car is parked overnight.
- Some parking lots scan permits on entry and exit, and charge is based on total hours parked, upon exit.
- Customers must have registered with the University parking office in order to use any parking lot. Customers can use any parking lot. Each parking transaction will incur a charge to their account.
- Customers can have more than one car, and so they may request more than one parking permit for each car.
- The University provides a 20% discount to compact cars compare to SUV cars.
- For simplicity, assume that the Parking Office sends a monthly bill to customer and customer pays it outside of the parking system.
- Each week you will need to submit an updated Class Diagram along with the other deliverables for the assignment.

Using these instructions, I created my Parking System Application that utilizes several classes to fulfill the above parameters.

## Design

### Parking

The foundational classes for my application exist in the edu.du.ict4315.parkingsystem.parking package as shown in Figure 1.

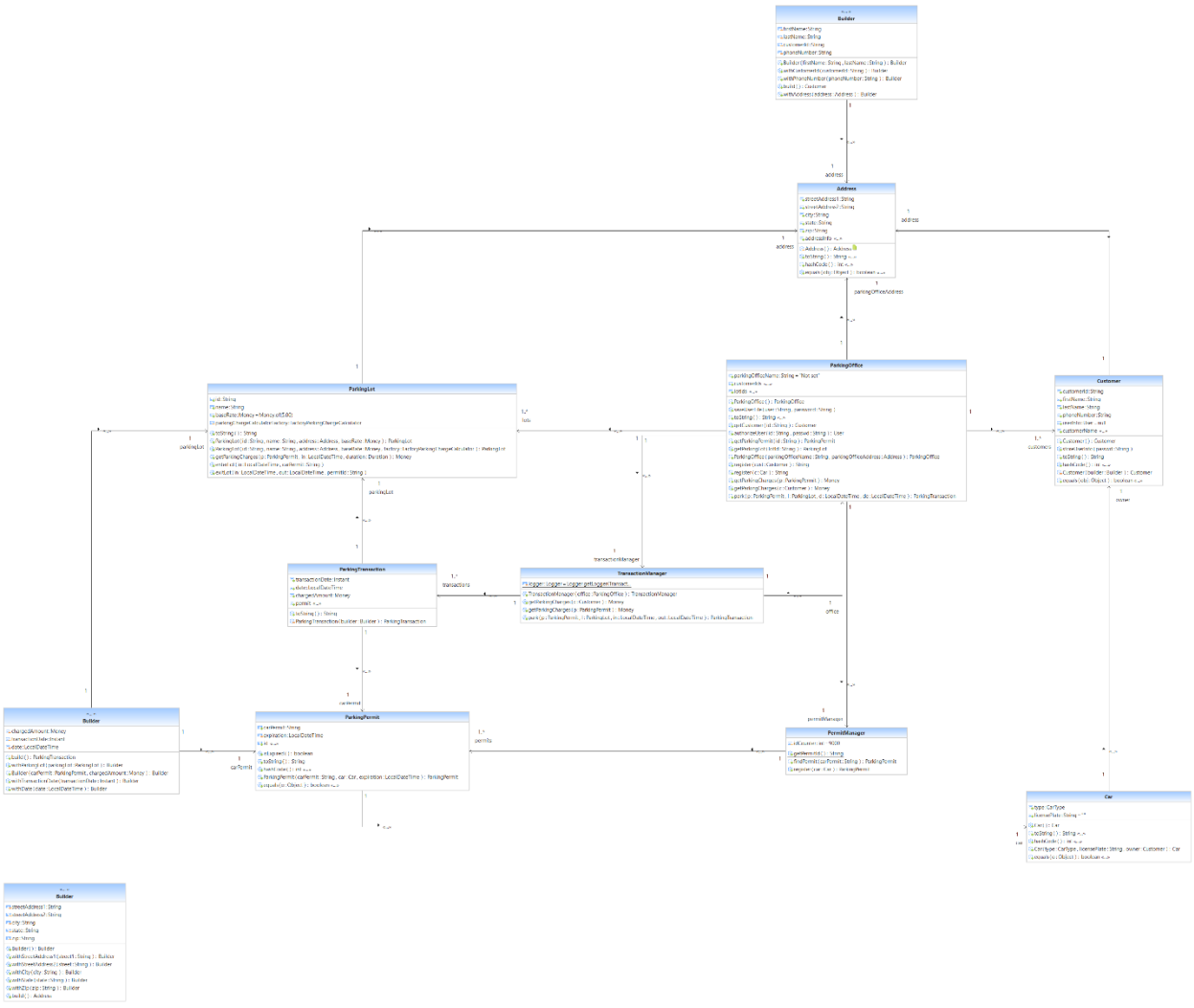

*Figure 1. Parking Package Class Diagram*

The classes in this package create the objects that most classes in the other packages use for their methods. The Address class creates address objects, which the Parking Office, Parking Lot and Customer classes use to set their addresses rather than initializing an address variable in each class repeatedly.

A central class in the entire system is the Parking Office class. This class keeps the list of registered customers as well as the various parking lots. Parking Office uses delegation for most of its methods so that it is not overloaded with too many calculations. The class contains many methods including methods to register both customers and their cars, retrieve the list of parking charges for either a specified customer or parking charge from the Transaction Manager Class, load and save the csv files containing user and parking lot information using the File Loader class, a park method that retrieves transactions from the Transaction Manager class, methods to search for registered customers or cars using their unique IDs (or permits), and a host of utility methods to return the Parking Office object's information.

The Address class and Customer class both use Java's Builder pattern to allow the created object some variance. In the Address class, this is done because addresses may vary depending on the residence or building with which they are associated, like an address that has a second street address line for an apartment/unit number. The Customer Builder Pattern is created with the customer's first and last names as required parameters. This is because a customer cannot register with the Parking Office without a name, but they can do so without an address or phone number.

The Car class represents the vehicles that the customers will register with the Parking Office to receive a parking permit to use the parking lots and accumulate parking charges. The

Car Type enumerated class contains two variables, SUV and COMPACT, which represent the two types of cars a customer may register and assists with calculating the 20% discount for compact cars. When a car is registered in the Parking Office class, it receives a parking permit created by the Permit Manager class.

Permit Manager creates permits and keeps the list of created permits. This class uses an idCounter integer variable initialized to begin at 9000, and increments the integer by one for each new permit created. The returned string is formatted to read as "P9000." This class also sets the permit expiration date to one year from the issuance date and adds the permit to the permits list. The findPermit() method is what Parking Office delegates to for it's own findPermit() method. It takes in the permit String, iterates through the list, and returns the permit information (if it exists in the list) maintained by the Parking Permit class.

Parking Permit is the class that takes what Permit Manager creates and associates it to a vehicle and thus a customer. This is a simple but important class for creating and calculating parking charges, which is where the class Parking Lot becomes relevant.

Another central class to this application, the Parking Lot class represents the parking lots on campus for which this application was created. Parking Lot has basic methods to retrieve the parking lots' information in addition to the very important method getParkingCharges(). This method is what calculates the cost for each scanned permit. The method calculation is delegated to the Factory Parking Charge Calculator class, which I will discuss later in this presentation. The method takes in a parking permit, the time the permit was scanned in and out in the form of LocalDateTime objects, the duration between the in and out times and then returns a Money object which represents the parking charge.

The Transaction Manager class uses the getParkingCharges() method to create a Parking Transaction object. Transaction Manager is also where the full methods to retrieve the list of parking charges for either a customer or permit. These methods take in either a Customer or Parking Permit object, filter through the list of all transactions to retrieve the transactions associated with the respective object and add them together to return a Money object representing the total charges. These methods are useful in totaling a customer's charges each month for billing purposes. The version that takes in a Customer is the primary method the parking office would use to calculate a customer's total charges. If a customer has multiple registered vehicles, this method calculates the total across all the customer's cars, whereas the version that takes in only a Parking Permit calculates the total for only the vehicle associated with that permit.

Parking Transaction is the class that creates the objects returned from the Transaction Manager park() method. This class is also constructed using the Builder Pattern. The information in a Parking Transaction object includes a parking permit, a Money object representing the amount charged, a transaction time variable of type Instant, a parking lot, and a transaction time variable of type LocalDateTime; parking permit and amount charged being required.

## Decorator and Factory

Throughout the term, we learned and experimented with several design patterns to create parking charges. For my Parking System Application, I decided to implement the Decorator and Factor Design Patterns shown in Figure 2. When a parking lot scans a parking permit, it uses similar variable to parking transaction to calculate the charges and return a

Money object. This method is delegated to the Factory Parking Charge Calculator interface. The classes that implement this interface include Factory Flat Rate, Factory Compact Parking Charge, and Factory Weekend Charge. These classes all contain one method, getCalculator(), which then retrieves the various parking charge decorator objects to calculate parking charges based on various factors like the type of car or day of the week.

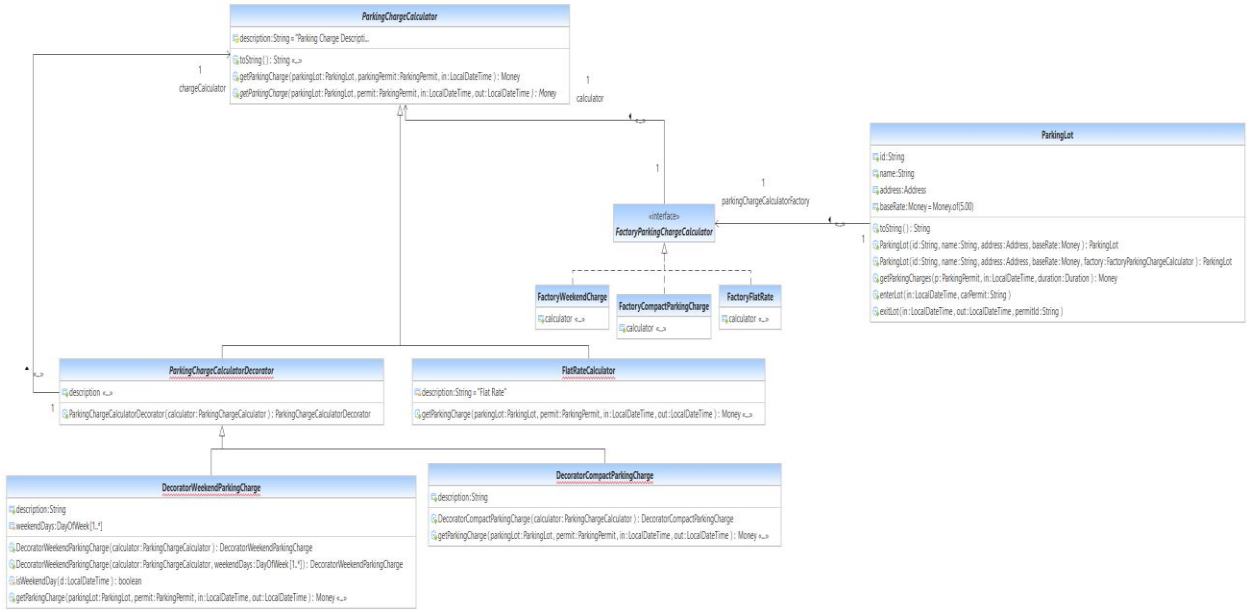


*Figure 2. Decorator and Decorator Factory Class Diagram*

Parking Charge Calculator Decorator is an abstract class that extends the abstract Parking Charge Calculator class. Three decorator classes then extend the Parking Charge Calculator Decorator class, Decorator Compact Parking Charge, Decorator Weekend Parking Charge and Flat Rate Calculator. These classes contain the calculations that the Parking Lot class will then return as a parking charge. The Decorator Weekend Parking Charge contains a list consisting of the two weekend days, using Java's java.time.DayOfWeek enum import. A private boolean method then compares the day of week from the permit's LocalDateTime in-time variable to this list and if the date is either a Saturday or Sunday, the charge is reduced by 50%,

regardless of car type. The Decorator Compact Parking Charge gets the car type from the scanned parking permit, and if the car is a compact car, the charge is reduced by 20%.

The decorator factories wrap the flat rate— which is just the parking lots' base rate— with the calculators, and the csv file, parking_lot_factories, assigns the type of factory each lot will use to the parking lots found in the parking_lots_du csv file. The compact charge is the basis for all lots as required in the assignment parameters, and the weekend charge is an accessory, that only some lots may use, which wraps both the compact charge and flat rate.

## Server, Clients, Command and Response

Through the exploration of different server-client connection methods, I have implemented the Object Writing Server, Object Client, and Handle Remote Client classes as illustrated in Figure 3. The server and client classes communicate to receive Parking Request

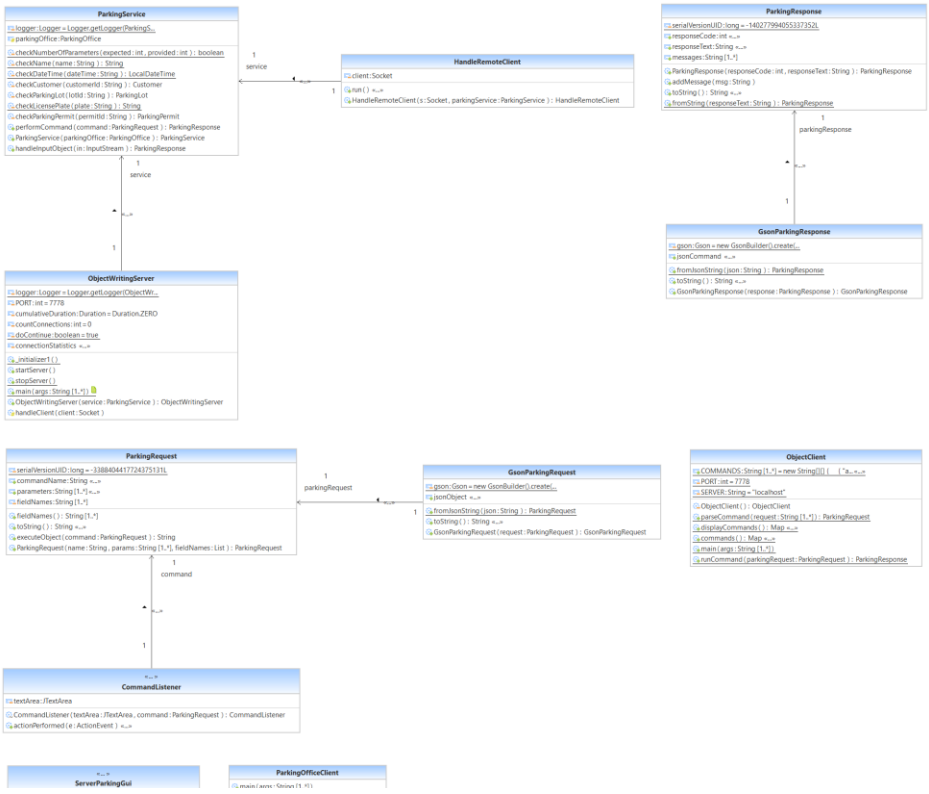


*Figure 3. Server, Client, Command Class Diagram*

command objects, perform the command, and return a Parking Response string. The commands currently supported are CUSTOMER, CAR and STOP.

CUSTOMER takes the parameters firstname and lastname and performs the register(Customer) method in Parking Office, assigning the new customer an ID and adding the customer to the list of registered customers. Similarly, the CAR command takes in the parameters of the car license plate and customer ID and performs the register(Car) method to assign a permit to the car and add it to the list of registered permits. The ObjectClient class parses the command to ensure that it is a valid command based on the array of commands and runs the command to receive a Parking Response. The Parking Response class contains the performCommand() method. This is method takes in a Parking Request object, checks the command parameters, performs the command (if applicable), and adds messages for Parking Response to return.

Object Writing Server is a multithreaded class that has methods to start and stop the server, handle the client, and gets and prints the connection statistics. The Handle Remote Client class allows for multithreading through the implementation of Java's java.lang.Runnable interface.

The Gson Parking Request and Gson Parking Response classes wrap the Parking Request and Parking Response classes respectively to convert Parking Request objects to Gson from Json and vice versa.

The Server Parking Gui class constructs the GUI (shown in Figures 4 and 5) used to take in the Parking Request Commands from the graphical interface a user would use to register themselves and their vehicles.

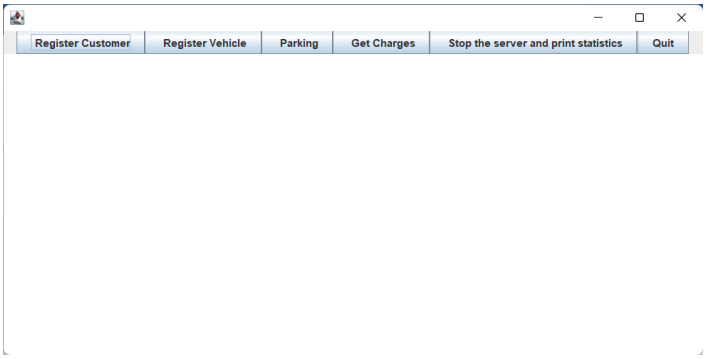


*Figure 4. Server Parking GUI*

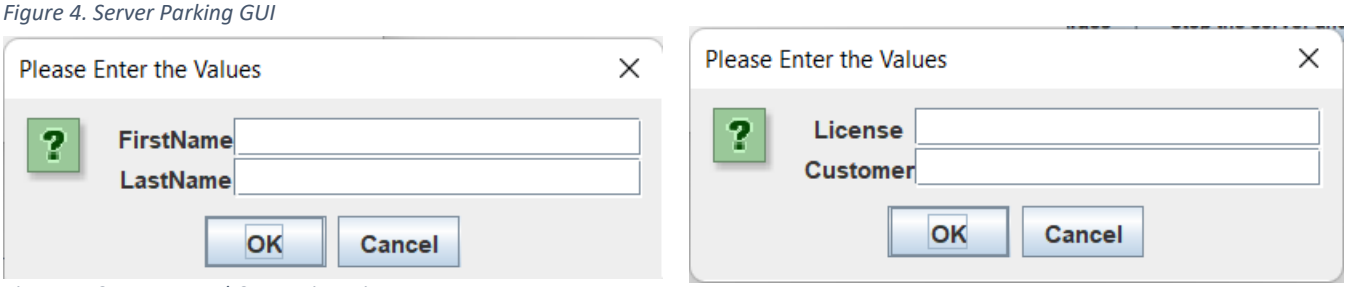


*Figure 5. Customer and Car Registration Prompts*

## Support

The package, edu.du.ict4315.parkingsystem.support, contains all of the supporting classes for the Parking System Application shown in Figure 6 below. The only one of these classes that I changed from the provided starter code is the Id Maker class. I wanted my customer IDs to be the first initials of the customer's first and last names followed by a digit, so I added some short lines of code to make this happen. Now, the Id Maker makeId() method takes in a Customer object, gets the customer name, parses through it to pull the first initials of each name String and puts them together capitalized, and combines that with the AtomicInteger counter variable to assign a unique customer ID.

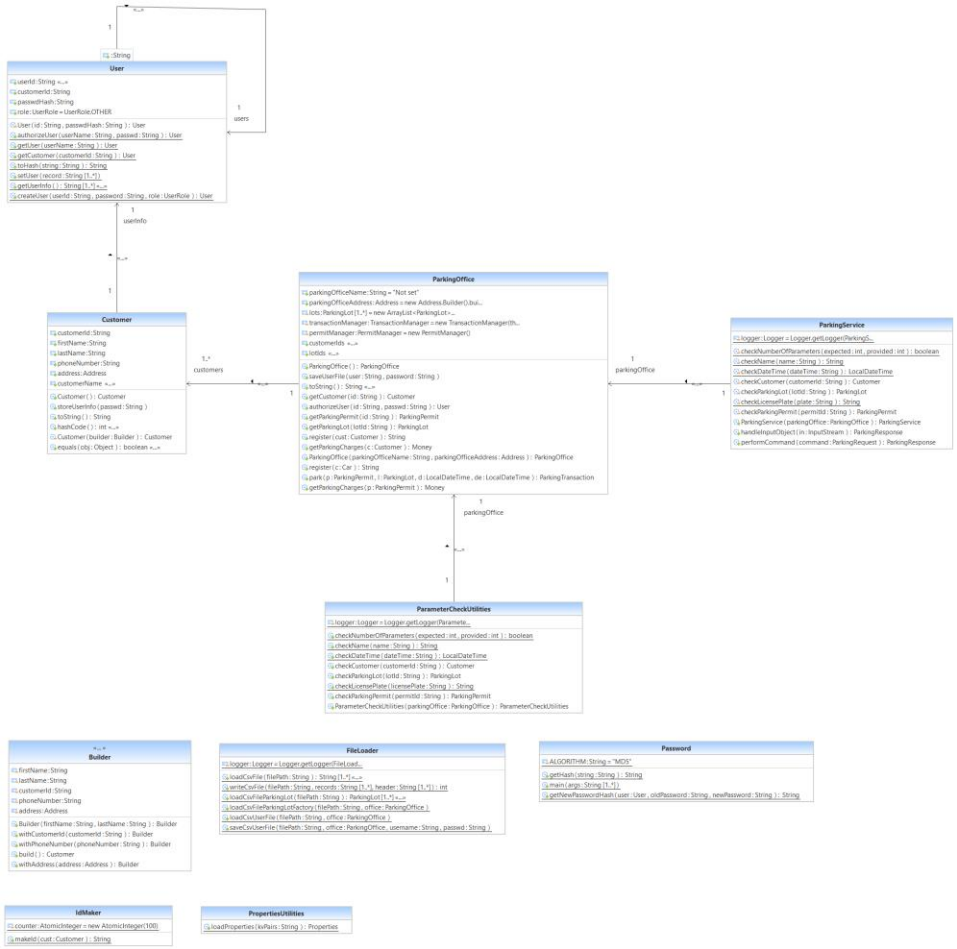


*Figure 6. Support Package Class Diagram*

**Reflection**

This application was incredibly challenging and yet incredibly rewarding. I learned a great deal about Object Oriented Programming through the creation of my Parking System Application. As someone fairly new to OOP and Java, I had a lot to learn and each week presented a new opportunity to do so. I had a hard time seeing the big picture in the beginning, which did make it more difficult to understand how each piece of the assignment was going to work together to complete the application. Almost every concept was new to me and thus required a significant amount of time dedicated to researching to try and understand the concept before implementing them. Once implemented, then there was time dedicated to troubleshooting and testing. Eclipse IDE provides quite a few helpful utilities for debugging and troubleshooting, which were very helpful.

Given more time to learn and work, I would have added more parking charge decorators and factories to add some complexity and variety to the ways in which a parking charge is calculated. I also would have written in code to support commands that retrieve parking charges based on the getParkingCharge() methods in Parking Office. I would have included and proxy and DBMS to really enhance the application. I also would have taken more time to learn and truly understand the more confusing concepts. However, this term ending does not mean I am done tinkering with my application, as I now have a great application with which I can practice java concepts.

Bringing my Parking System Application together was so incredibly rewarding. Troubleshooting and solving the problems I encountered solidified to me that even though I felt like I was just cramming and regurgitating each week, concepts and lessons were sticking and I

did learn quite a bit. This has been one of the first programming projects I've completed that has me excited about continuing to practice my coding skills.

**References**

Abdrazak, Almas. 2019. “Inheritance , the Good ,the Bad, the Ugly.” *Medium*. Medium. January 13. https://medium.com/@almas337519/inheritance-the-good-the-bad-the-ugly-1bd1bf249813.

Baeldung. 2021. “Different Serialization Approaches for Java.” *Baeldung*. Baeldung. October 8. https://www.baeldung.com/java-serialization-approaches.

Bateman, Robert. 2022. “Sample Mobile App Privacy Policy Template.” *TermsFeed*. TermsFeed. March 13. https://www.termsfeed.com/blog/sample-mobile-app-privacy-policy-template/.

Bigelow, Stephen J. 2021. “CI/CD Pipelines Explained: Everything You Need to Know.” *SearchSoftwareQuality*. TechTarget. May 13. https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know.

Bloch, Joshua. 2018. *Effective Java, 3rd Edition*. Addison-Wesley.

Digital Team. 2022. “Privacy Policy.” *University of Denver*. University of Denver. https://www.du.edu/site-utilities/privacy-policy.

Freeman, Eric, and Elisabeth Robson. 2020. “Chapter 2. Keeping Your Objects in the Know: The Observer Pattern.” Essay. In *Head First Design Patterns*, 2nd ed. Beijing: O'Reilly.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

GeeksForGeeks. 2021. “Serialization and Deserialization in Java with Example.” Edited by Kumar Sagar and Vandana Kumari. *GeeksforGeeks*. GeeksforGeeks. October 7.

https://www.geeksforgeeks.org/serialization-in-java/#:~:text=Serialization%20is%20a%20mechanism%20of,stream%20created%20is%20platform%20independent.

Horstmann, Cay S. 2019. *Core Java® Volume I—Fundamentals, Eleventh Edition*. New York: Prentice-Hall. https://du-primo.hosted.exlibrisgroup.com/permalink/f/1vj5nol/01UODE_ALMA511025711720002766

Kadir, Abdul. 2019. "The Strategy Pattern Explained Using Java." *FreeCodeCamp.org*. freeCodeCamp.org. June 16. https://www.freecodecamp.org/news/the-strategy-pattern-explained-using-java-bc30542204e0/.

Pankaj. 2016. "Observer Design Pattern in Java." *JournalDev*. JournalDev. July 2. https://www.journaldev.com/1739/observer-design-pattern-in-java.

Pankaj. 2020. "Builder Design Pattern in Java." *JournalDev*. JournalDev IT Services Private Limited. February 24. https://www.journaldev.com/1425/builder-design-pattern-in-java.

Pedamkar, Priya. 2021. "Iterator in Java: Retrieving Elements Using the Iterator Method." *EDUCBA*. EDUCBA. November 8. https://www.educba.com/iterator-in-java/.

Pettit, Sten. 2022. "How to Get to Continuous Integration." *Atlassian*. Atlassian. https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration.

Rehkopf, Max. 2022. "What Is Continuous Integration." *Atlassian*. Atlassian. https://www.atlassian.com/continuous-delivery/continuous-integration#:~:text=Continuous%20integration%20(CI)%20is%20the,builds%20and%20tests%20then%20run.

Sacolick, Isaac. 2022. “What Is Ci/CD? Continuous Integration and Continuous Delivery Explained.” *InfoWorld*. InfoWorld. April 15. https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html.

“System Integration: Types, Approaches, and Implementation Steps.” 2021. *AltexSoft*. AltexSoft. March 10. https://www.altexsoft.com/blog/system-integration/.

Wanpal, Nikhil. 2016. “An Opinionless Comparison of Spring and Guice - DZone Java.” *Dzone.com*. DZone. October 6. https://dzone.com/articles/an-opinionless-comparison-of-spring-and-guice.

Wong, Henry; Oaks, Scott. 2004. *Java Threads, 3rd Edition.* O'Reilly Media, Inc.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.