Drone Ingredient Delivery System: Phase IV Design Changes Report

Dr. Mark Bomi Moss

CS 2340: Objects and Design

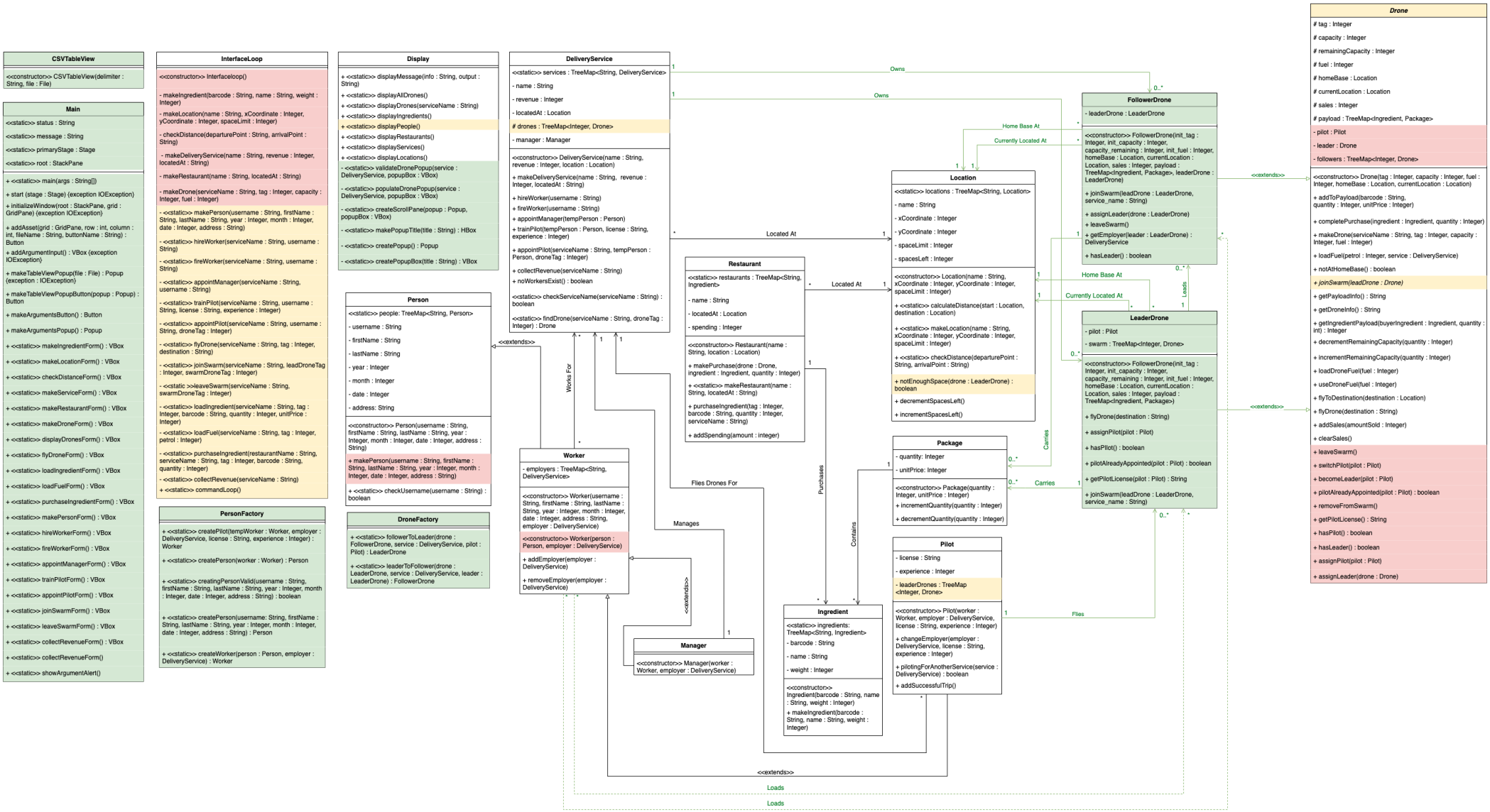
Reetesh Sudhakar, Yash Gupta, Kunal Daga, Sebastian Jaskowski

July 16th, 2022

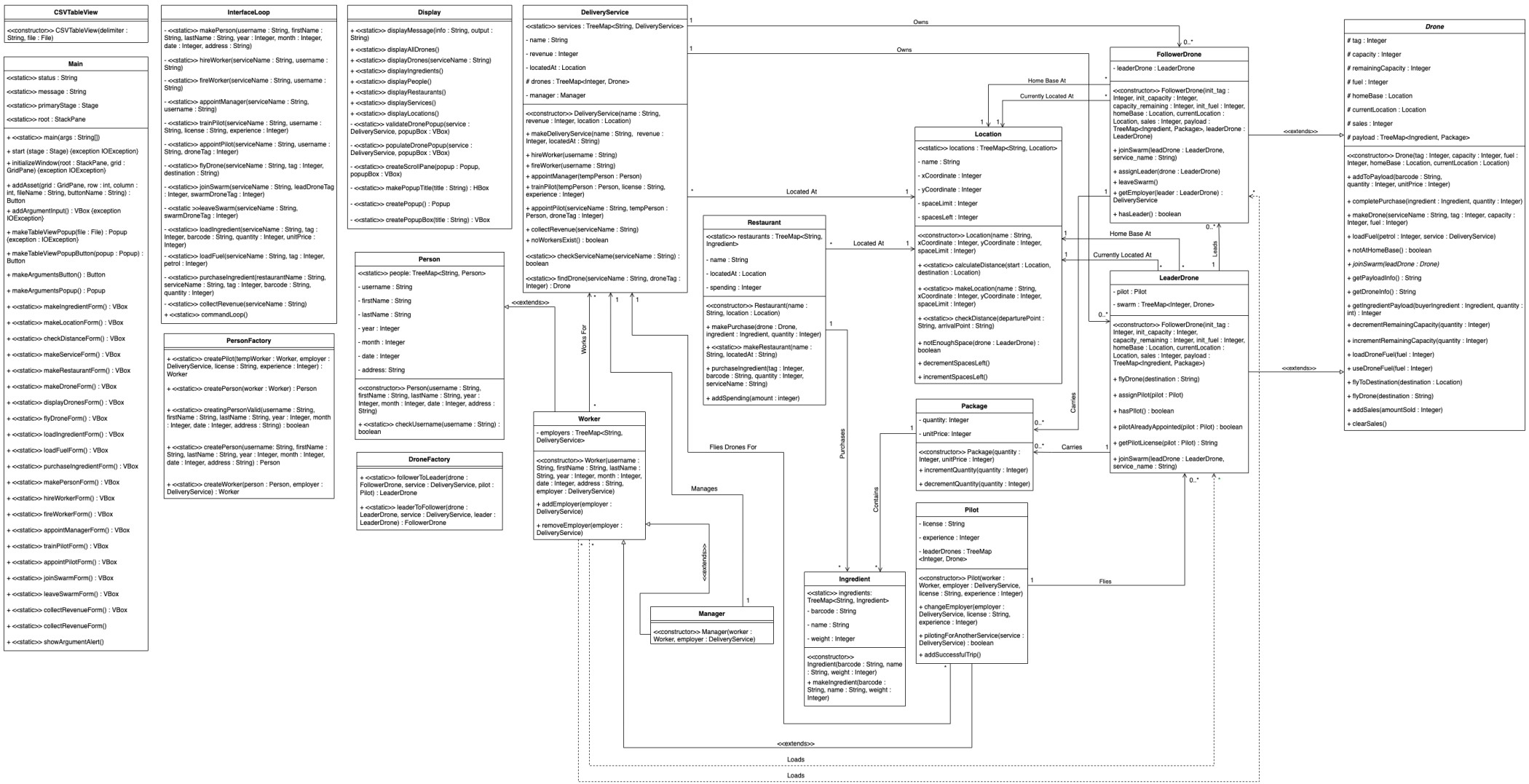
Summary:

With the completion of our Phase III Submission, we have successfully implemented all of the design requirements originally specified in the problem description for the ingredient delivery system. However, as with any software project, there is always room for continual iterative development. For this reason, in Phase IV, we implemented a few changes that aim to add new features to our software as well as improve the structure of our code in order to ensure it better aligns with established software development principles such as those outlined in GRASP and SOLID principles. Three primary changes were implemented during this phase of the iterative process: the implementation of a factory method pattern for the instantiation of drones and persons (pilots, workers, and persons), a redesign for the Drone class, as well as the implementation of a graphical user interface (GUI). These changes improve the scalability and design of the overall system, as well as ease of use and clarity with the visual interface. The system design implementations with the factory method and drone redesign were intended to clean up code, improve cohesion and modify the use of a controller class to achieve a low representational gap. Within the GUI, users have the option to enter arguments manually or with the aid of text fields and combo boxes to improve input validation, as certain commands are only enabled based on dependencies needed for their execution. As the GUI improves communication of changes with errors, successful changes, and displays, the design changes in the background work to improve the system's overall functionality. A summary of these changes is presented in this report, and we hope to demonstrate how these changes improve the functionality, scalability, and form of our code. The two design class diagrams shown summarize the design changes that were implemented in this phase with associations of classes, as well as changes made to methods in existing and new classes. Changes are colorized, with the final class diagram shown as well.

Design Class Diagram: Changes



Design Class Diagram: Phase IV



Factory Method Pattern

One of our first changes was the creation of two new classes called DroneFactory and PersonFactory to align with the Factory Class/Method design pattern. DroneFactory is a class that we use to convert a FollowerDrone to a LeaderDrone and vice-versa. PersonFactory is a class that we use to convert between instances of the Person family of classes (Person, Worker, Pilot, and Manager). This design pattern allows for cleaner business logic and separates code to better align with the GRASP design principles. Due to this change, our classes have higher cohesion. For example, when switching a drone to follow another within a swarm to a drone that has a direct pilot, rather than having to change attribute variables within a Drone class, the DroneFactory class is used to allow for a seamless change between the two classes. This change led to the Drone class being implemented as an abstract class, allowing the code to be simplified when switching between leader and follower drones, as exemplified in the code below.

```
public static LeaderDrone followerToLeader(FollowerDrone drone, DeliveryService service, Pilot pilot
) {
    LeaderDrone newDrone = new LeaderDrone(drone.getTag(), drone.getCapacity(), drone.
getRemainingCapacity(), drone.getFuel(),
            drone.getHomeBase(), drone.getCurrentLocation(), drone.getSales(), drone.getPayload());
    newDrone.assignPilot(pilot);
    service.drones.put(newDrone.getTag(), newDrone);
    return newDrone;
}

public static FollowerDrone leaderToFollower(LeaderDrone drone, DeliveryService service, LeaderDrone
leader) {
    FollowerDrone newDrone =  new FollowerDrone(drone.getTag(), drone.getCapacity(), drone.
getRemainingCapacity(), drone.getFuel(), drone.getHomeBase(), drone.getCurrentLocation(), drone.
getSales(), drone.getPayload(), leader);
    service.drones.put(newDrone.getTag(), newDrone);
    return newDrone;
}
```

Figure 3: FactoryDrone Class

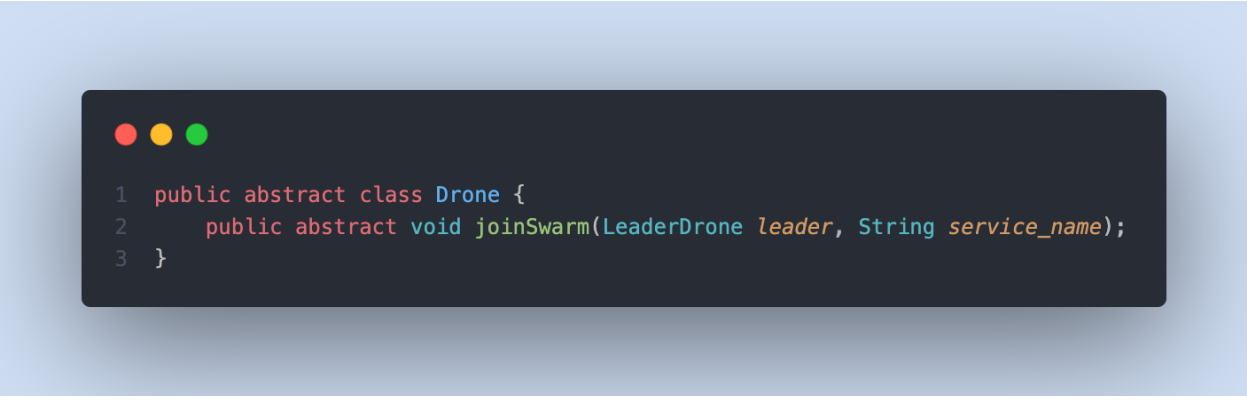
Furthermore, the *DeliveryService* class now calls *PersonFactory.createPilot()* to delegate the responsibility of creating a *Pilot* from a *Worker* to the *PersonFactory* class. Through this separation of code, our *DeliveryService* class now is only responsible for methods that can be directly attributed to a *DeliveryService* and all other methods are moved to other classes. This also allows for easy change in the future if the way our objects are created changes, potentially as new business requirements are discovered.

FollowerDrone and LeaderDrone Implementation

In Phase 3, we used a single *Drone* class that contained all the functionality of handling the leader and follower mechanisms. We implemented this using two attributes, one being a reference to a pilot and one being a reference to a drone. This required one attribute being null at all times, and lots of control flow to handle both cases in the InterfaceLoop class. This required a lot of casting and checking.

In our new design, we separated *Drone* into *LeaderDrone* and *FollowerDrone*, so that logic would be separated more naturally. *LeaderDrone* objects have a reference to a *Pilot*, and *FollowerDrone* objects have a reference to their *LeaderDrone*, separating their functionality. This system is simpler, has higher cohesion, and also solves separation of concerns. As a result, the *InterfaceLoop* was far simpler, allowing the respective drone classes to contain more of the implementation logic. This also led us to make the *Drone* class abstract, which is a quasi-interface that all has common drone functionality and from which both *FollowerDrone* and *LeaderDrone* inherit. From a design perspective, this makes sense since in our system there will never be an object of *Drone* type, but rather of either *FollowerDrone* and *LeaderDrone* to distinguish between what is piloting the drone. Furthermore, this also allows for further expansion of the system by allowing the Drone class to be abstract. If, for example, the system

required to have several other types of drones, this design change would aid in expanding the different types of drones that could be used for this system, making it more scalable.



```
1  public abstract class Drone {
2      public abstract void joinSwarm(LeaderDrone leader, String service_name);
3  }
```

Figure 4: Abstract Drone class and abstract method joinSwarm()

Graphical User Interface (GUI)

In addition to changes that improved the structure of our code, we also added a GUI to our software to enhance the user experience. The GUI provides a simplification of usage as it relates to the system. First, there is a button titled “View Commands”, allowing users to check what parameters are needed when entering arguments manually, as was done using the command line in previous iterations. The structured format design pattern was used to ensure that confusion was minimized as it relates to argument input and usage with the displays. Furthermore, the “Arguments” button allows users to input arguments into TextFields and ComboBox objects. These objects have validations based on the existence of certain objects: for example, if a delivery service does not exist, then any input fields that require a delivery service for its execution are immediately disabled until they are instantiated by other commands (e.g., make_person, make_ingredient, etc.). The landing page of the user interface and Arguments window are shown below.

Figure 5: Graphical User Interface Landing Page

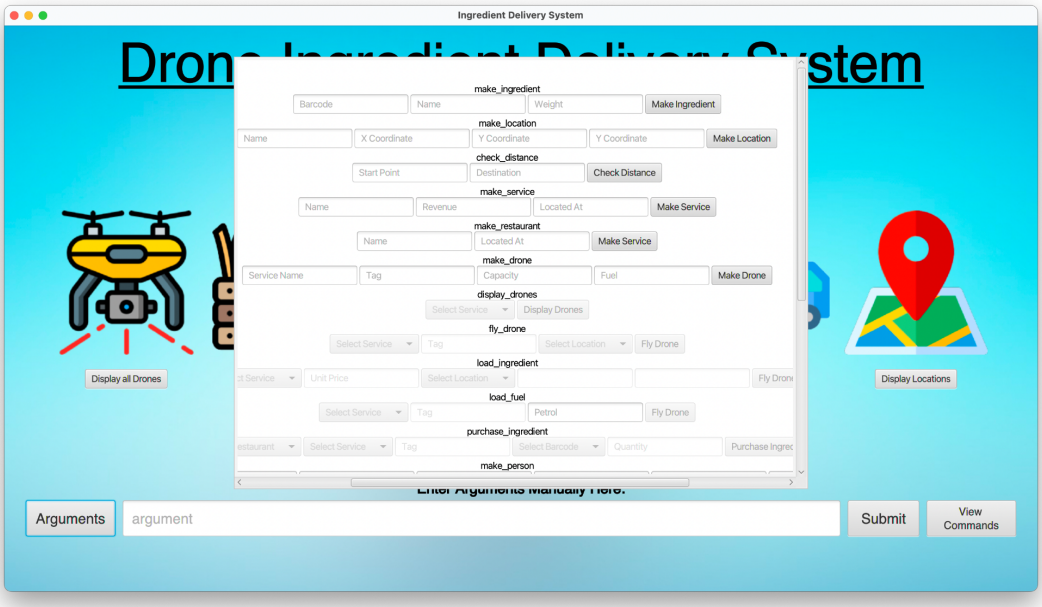


Figure 6: Graphical User Interface Argument Input Window

The addition of this user interface raised questions about the Boundary classes of our system and how exactly our users interact with our system. Since we had an existing controller

class, InterfaceLoop, in our previous phase, we adapted that class to act as a controller class between the user interface related classes and our domain classes. With the inclusion of a manual input, the InterfaceLoop class was extremely beneficial in maintaining the functionality of this text field in the GUI. The graphical user interface elements perform calls to InterfaceLoop, which in turn calls the necessary methods in the respective domain classes. While we believe this solution works is sufficient for this current iterative phase, we acknowledge that it has propagated some technical debt due to creating a class with low cohesion. In a future iteration, we would likely consider splitting up the InterfaceLoop class into multiple controller classes that are related to specific domains, such as Restaurants, Delivery Services, and Drones. In this way, not all calls from the GUI elements end up traversing through a singular class.

In addition to the InterfaceLoop class, we also took the pre-existing Display class, which was previously used for displaying success and error messages to the console, and adapted it to display success and error pop ups within the GUI, as well as display information requested by the user by returning Popup objects when specific buttons were pressed using event-driven programming. Within the Main class, static variables for status and output messages were used to ensure that only a singular Alert object was instantiated and displayed within the GUI to simplify the code. This singleton-like pattern helps ensure that the proper errors and success messages are displayed. The replaceable instance concept of the singleton design pattern was used within the Main class utilizing a switch statement to ensure that the proper alert message was shown when any command was executed. In this way, all feedback to the user from our system traverses through this second controller class that manages output from our system. The implementation of a GUI aids in usability of the overall system, while also allowing users to view information in a digestible and efficient manner.

References

"Factory Method." *Refactoring Guru*, 2014, refactoring.guru/design-patterns/factory-method.

Kollár, Lajos, and Nóra Sterbinszky. *Case Study in System Development - Notes*. Hungary, Debreceni Egyetem, 2014, gyires.inf.unideb.hu/GyBITT/07/. Accessed 10 July 2022.

Larman, Craig. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Upper Saddle River, N.J., Prentice Hall Ptr, 2005.

Stencel, Krzysztof, and Patrycja Węgrzynowicz. "Implementation Variants of the Singleton Design Pattern." *On the Move to Meaningful Internet Systems: OTM 2008 Workshops*, 2008, pp. 396–406, 10.1007/978-3-540-88875-8_61. Accessed 17 May 2022.

"Structured Format Design Pattern." *Ui-Patterns.com*, ui-patterns.com/patterns/StructuredFormat.