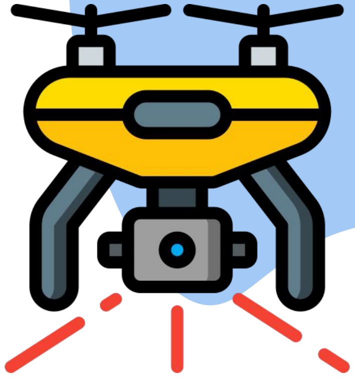
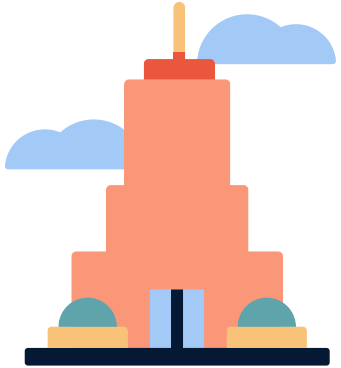
PHASE IV DESIGN CHANGES: CS 2340

Group 13: Reetesh Sudhakar, Yash Gupta, Sebastian Jaskowski, Kunal Daga



DESIGN CHANGES FROM PHASE III

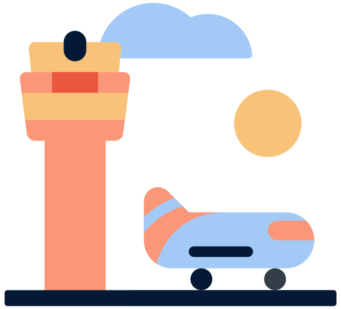


GUI

Added a GUI to replace the CLI and improve interactivity with the system

Factory Methods

Added a *PersonFactory* and *DroneFactory* class for shifting objects

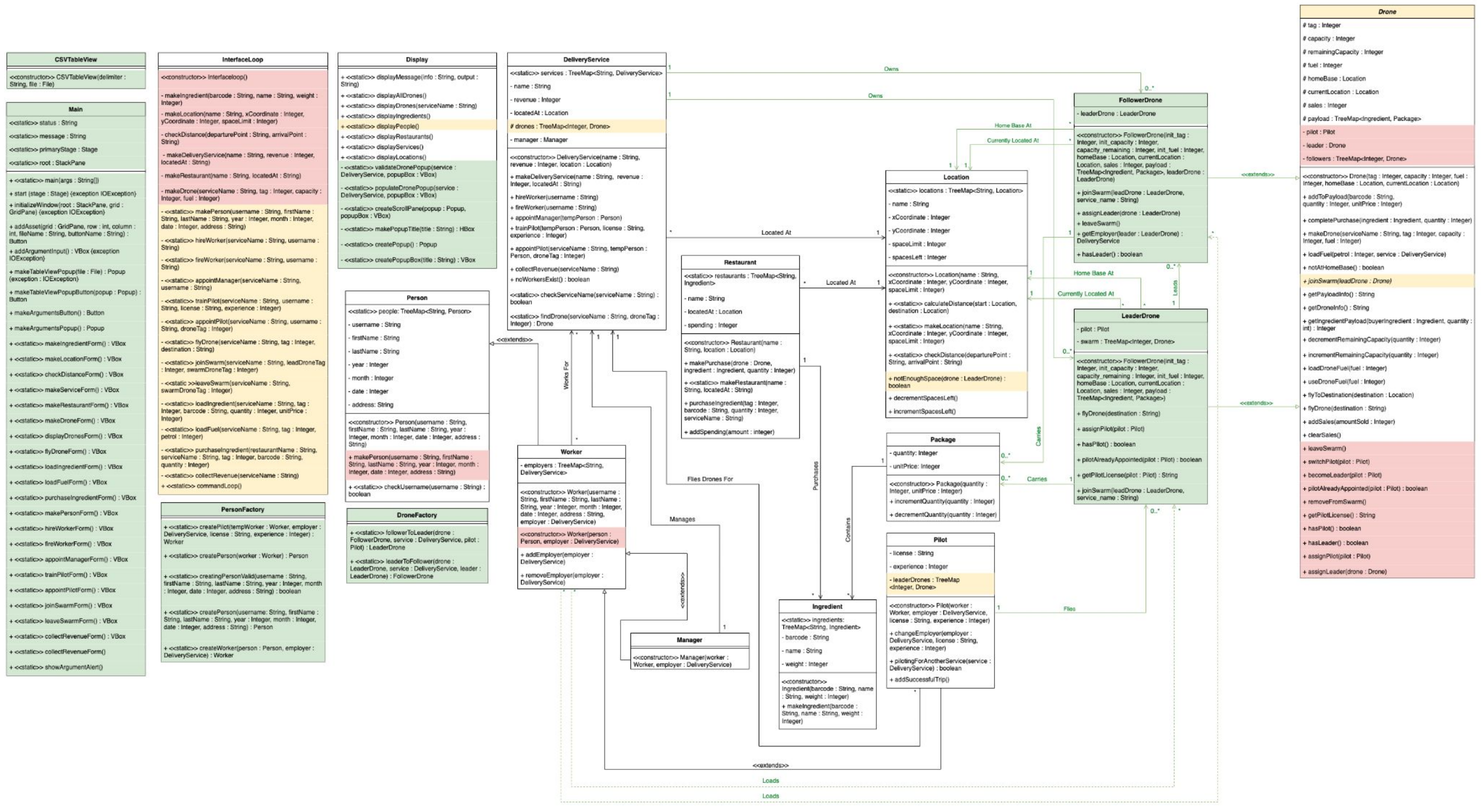


Drone Abstraction

Shifted from a singular *Drone* class to a hierarchy with *FollowerDrone* and *LeaderDrone* classes

Summary of Changes

Design Class Diagram: Changes

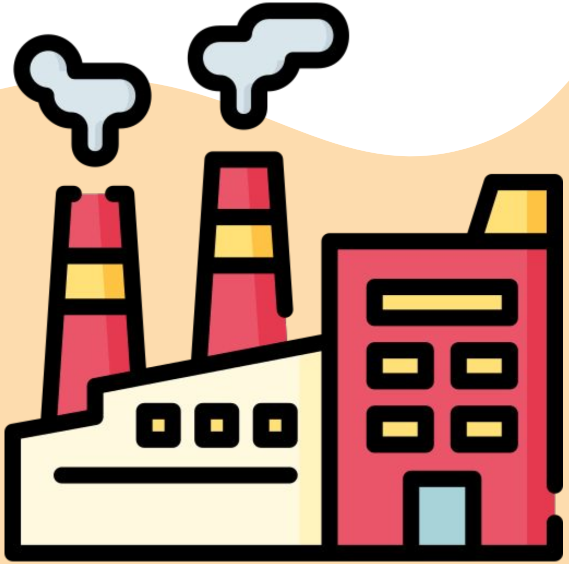


01

Factory Methods

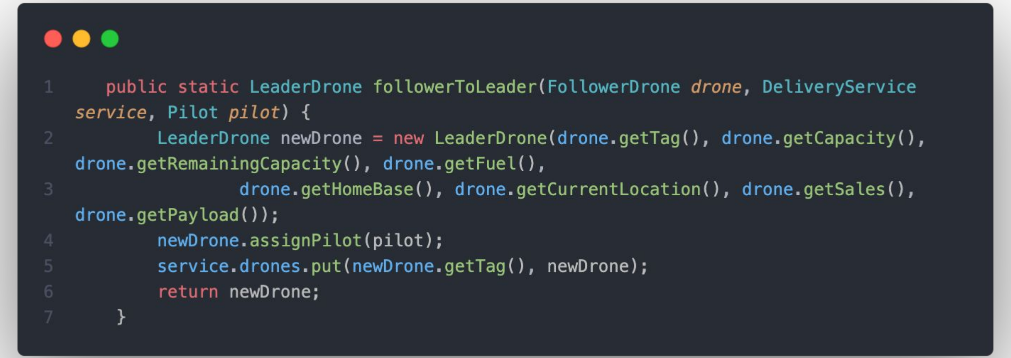
Implementation of
PersonFactory and *DroneFactory*

Purpose of Factory Methods

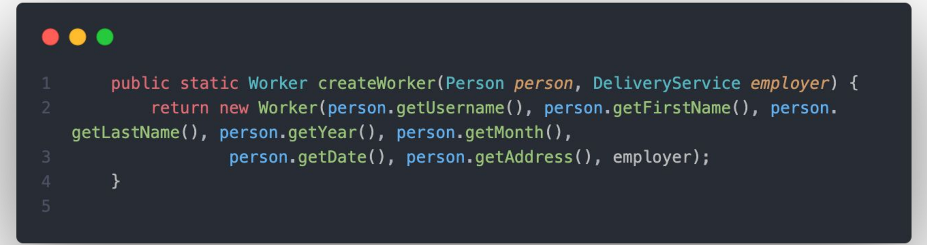


Factory Methods are **creational class patterns** that use methods to create objects without having to specify the exact class of the object.

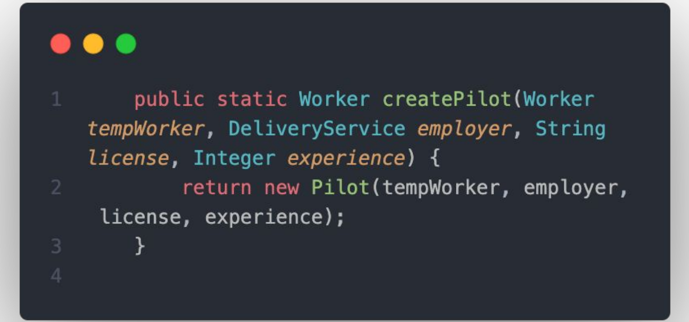
Source Code: Factory Classes



```java
    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;
    }
```



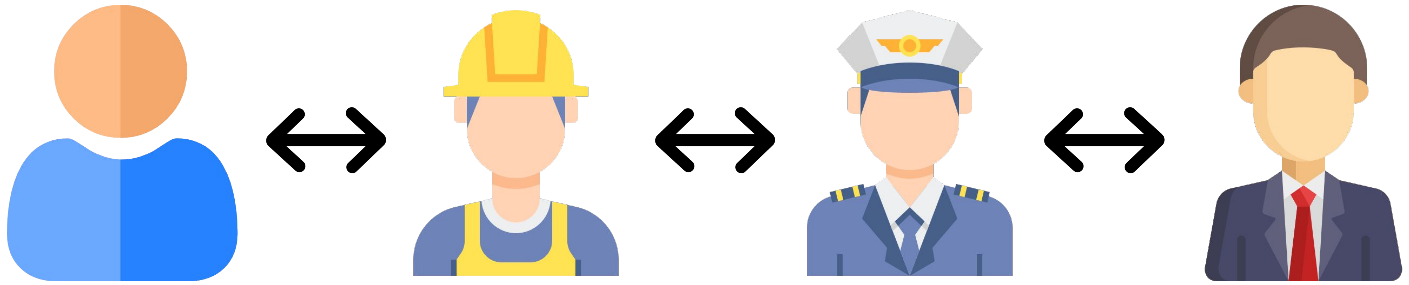
```java
    public static Worker createWorker(Person person, DeliveryService employer) {
        return new Worker(person.getUsername(), person.getFirstName(), person.
getLastName(), person.getYear(), person.getMonth(),
                person.getDate(), person.getAddress(), employer);
    }
```

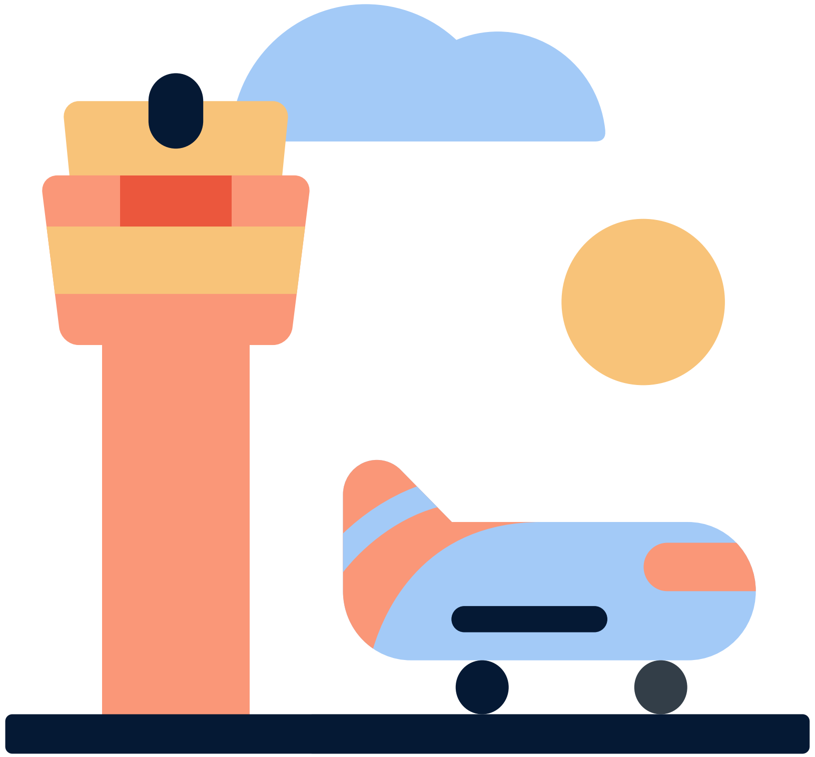


```java
    public static Worker createPilot(Worker
tempWorker, DeliveryService employer, String
license, Integer experience) {
        return new Pilot(tempWorker, employer,
 license, experience);
    }
```

Impact of the Factory Method Design

- **Easy conversion** between classes
    - Worker → Pilot, Worker → Person, etc.
- Promotes **low coupling** by eliminating the need to bind application-specific classes to the code
- Enables **high-cohesive** classes that are only responsible for performing actions that directly relate to them, not other classes
- Code interacts with the **resultant** classes, preventing any issues with conversions between objects





02

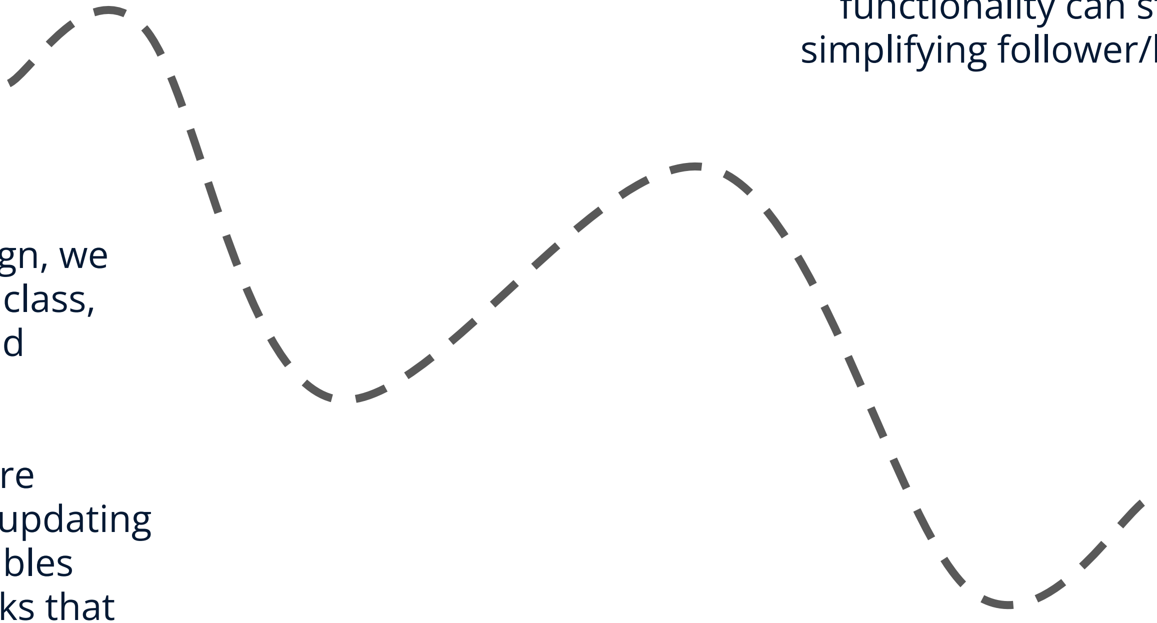
Drone Abstraction

Creation of subclasses
LeaderDrone and *FollowerDrone*

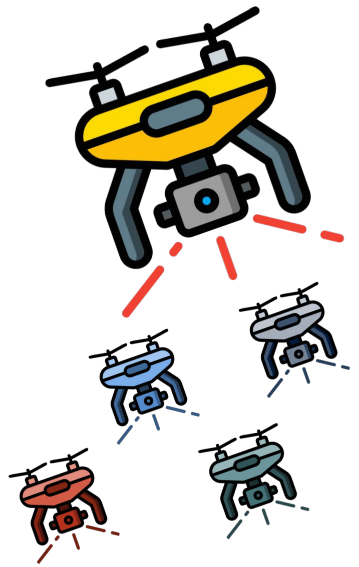
From Old to New

In our Phase III design, we only had one *Drone* class, which had *leader* and *followers* attributes.

These attributes were frequently null and updating these instance variables involved many checks that could still induce errors.

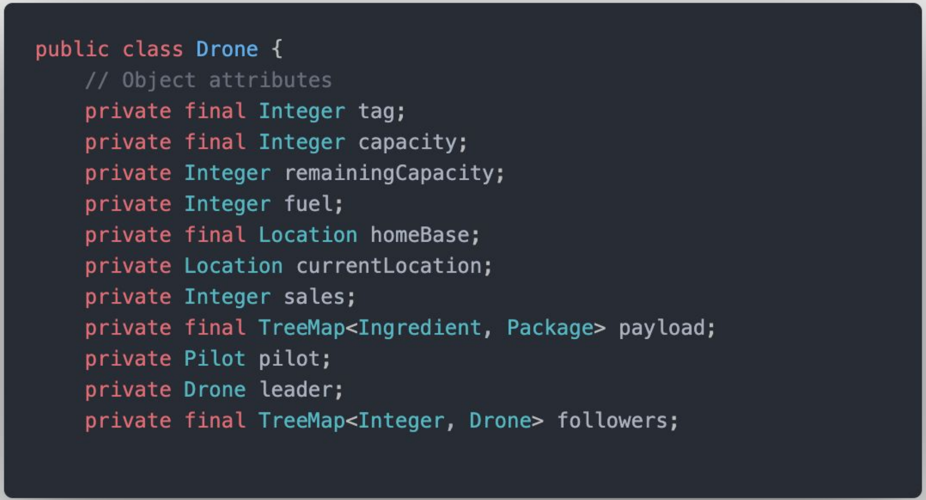


Our Phase IV design had a single **abstract** *Drone* class, which means functionality can still be shared, while simplifying follower/leader conversions.



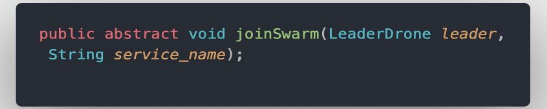
Source Code: Drone Abstraction

Phase III

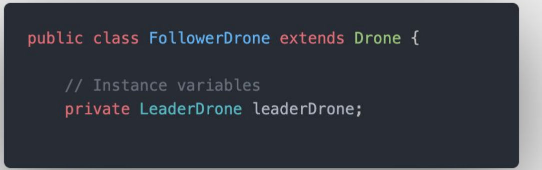


```java
public class Drone {
    // Object attributes
    private final Integer tag;
    private final Integer capacity;
    private Integer remainingCapacity;
    private Integer fuel;
    private final Location homeBase;
    private Location currentLocation;
    private Integer sales;
    private final TreeMap<Ingredient, Package> payload;
    private Pilot pilot;
    private Drone leader;
    private final TreeMap<Integer, Drone> followers;
```

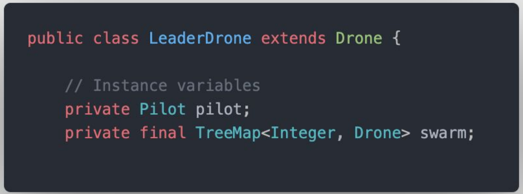
Phase IV



```java
public abstract void joinSwarm(LeaderDrone leader,
 String service_name);
```



```java
public class FollowerDrone extends Drone {

    // Instance variables
    private LeaderDrone leaderDrone;
```

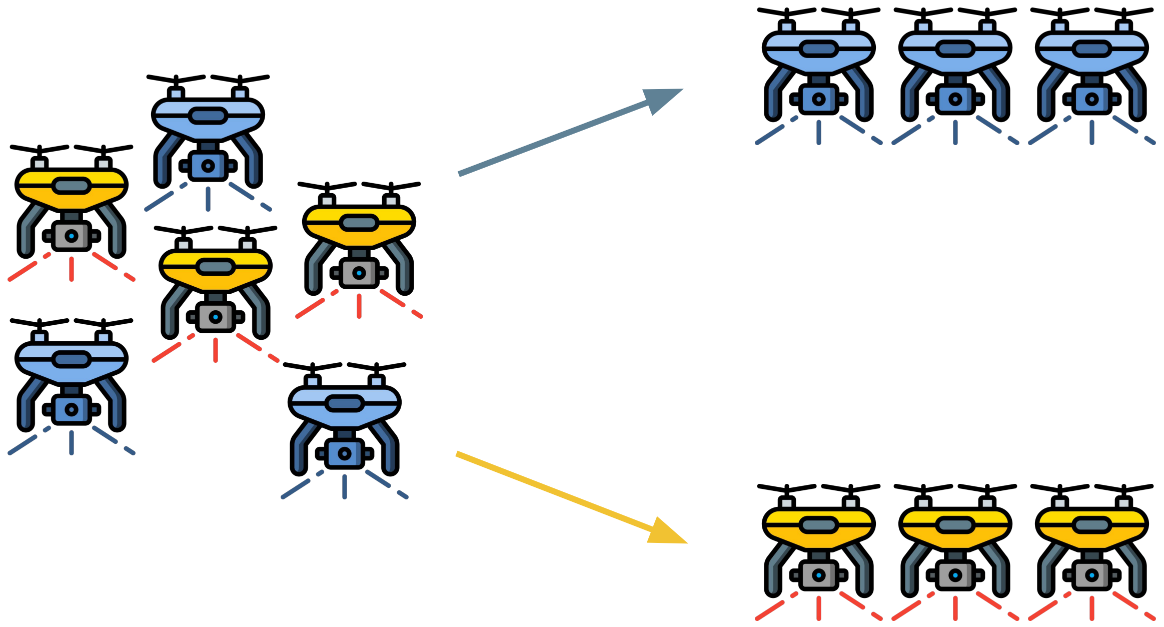


```java
public class LeaderDrone extends Drone {

    // Instance variables
    private Pilot pilot;
    private final TreeMap<Integer, Drone> swarm;
```

Benefits of Abstraction in our Code

Abstraction of the *Drone* class achieves the following:

- Increases **cohesion** and **separation of concerns**
- Allows for more **concise** methods when creating leader/follower specific methods
- Simplifies **conversion** between leader and follower drones to simple casting rather than complicated custom methods



03

GUI

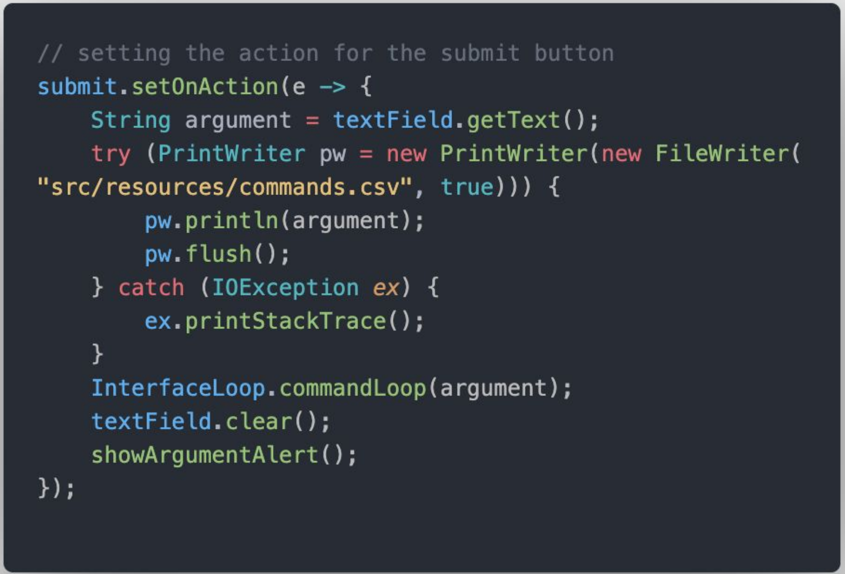
Redesigning the way clients interact with our system

Purpose of a GUI

GUIs are Graphical User Interfaces that typically make it much easier to use the software. Clients without a computing background will find it much easier to use a visual system as opposed to a Command Line Interface (CLI).

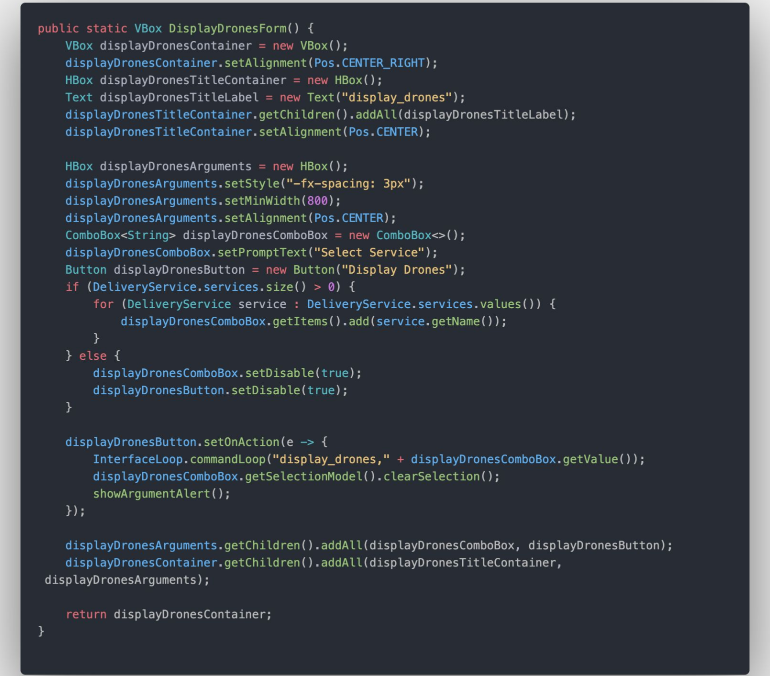
Source Code: GUI

Event-Driven Design



```java
// setting the action for the submit button
submit.setOnAction(e -> {
    String argument = textField.getText();
    try (PrintWriter pw = new PrintWriter(new FileWriter(
"src/resources/commands.csv", true))) {
        pw.println(argument);
        pw.flush();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    InterfaceLoop.commandLoop(argument);
    textField.clear();
    showArgumentAlert();
});
```

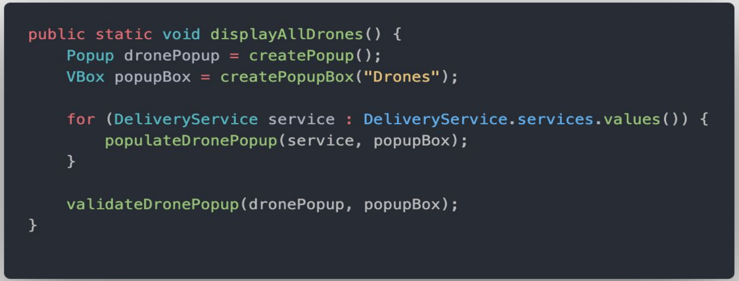
Form Creation



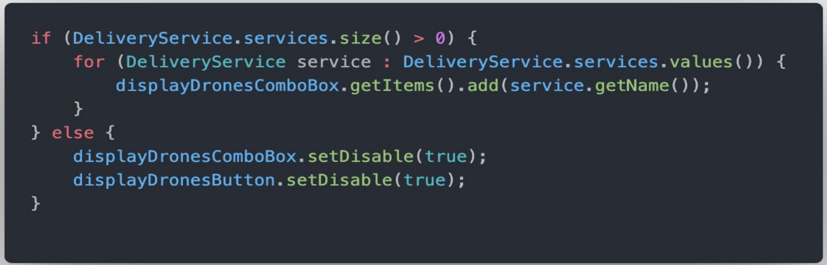
```java
public static VBox DisplayDronesForm() {
    VBox displayDronesContainer = new VBox();
    displayDronesContainer.setAlignment(Pos.CENTER_RIGHT);
    HBox displayDronesTitleContainer = new HBox();
    Text displayDronesTitleLabel = new Text("display_drones");
    displayDronesTitleContainer.getChildren().addAll(displayDronesTitleLabel);
    displayDronesTitleContainer.setAlignment(Pos.CENTER);

    HBox displayDronesArguments = new HBox();
    displayDronesArguments.setStyle("-fx-spacing: 3px");
    displayDronesArguments.setMinWidth(800);
    displayDronesArguments.setAlignment(Pos.CENTER);
    ComboBox<String> displayDronesComboBox = new ComboBox<>();
    displayDronesComboBox.setPromptText("Select Service");
    Button displayDronesButton = new Button("Display Drones");
    if (DeliveryService.services.size() > 0) {
        for (DeliveryService service : DeliveryService.services.values()) {
            displayDronesComboBox.getItems().add(service.getName());
        }
    } else {
        displayDronesComboBox.setDisable(true);
        displayDronesButton.setDisable(true);
    }

    displayDronesButton.setOnAction(e -> {
        InterfaceLoop.commandLoop("display_drones," + displayDronesComboBox.getValue());
        displayDronesComboBox.getSelectionModel().clearSelection();
        showArgumentAlert();
    });

    displayDronesArguments.getChildren().addAll(displayDronesComboBox, displayDronesButton);
    displayDronesContainer.getChildren().addAll(displayDronesTitleContainer,
 displayDronesArguments);

    return displayDronesContainer;
}
```

How the GUI Impacted our Code

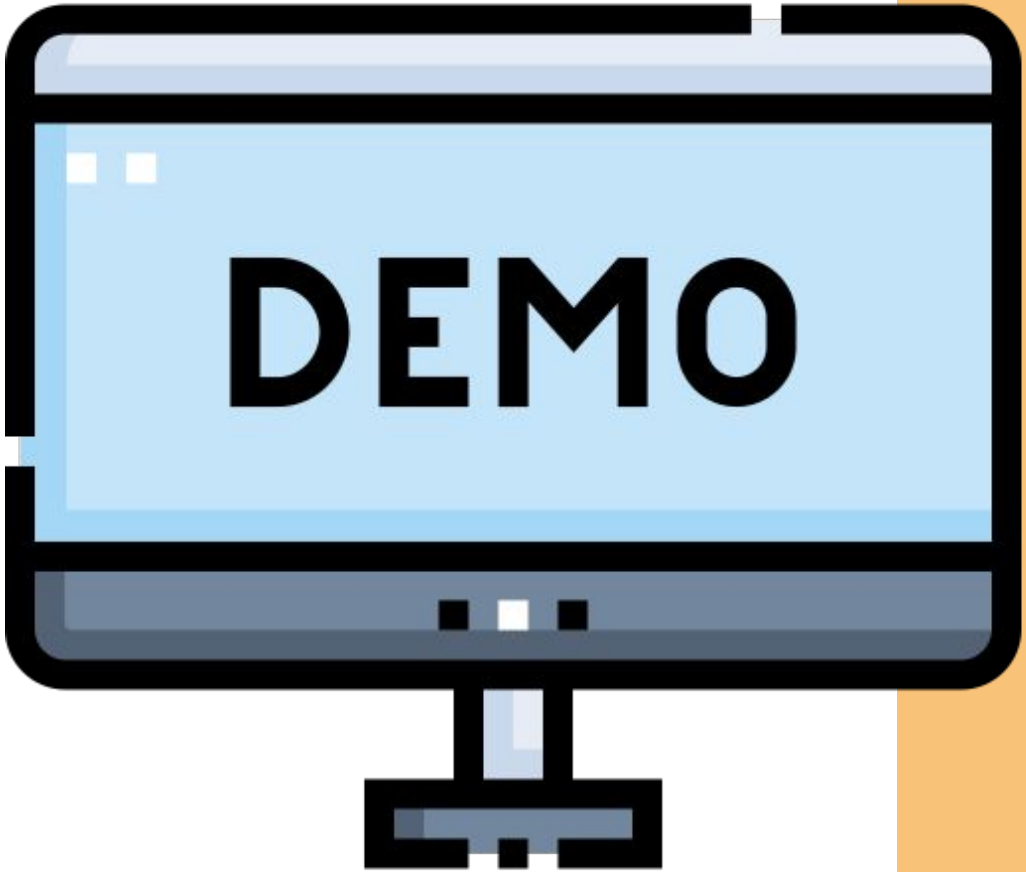
- Event-driven programming (buttons and displays)
- Redesigning the Displays (instantiating *Popup* objects, instead of printing to the command line)
- Input validation (disabling buttons for invalid arguments)
- *InterfaceLoop* class is not instantiated: static methods



```java
public static void displayAllDrones() {
    Popup dronePopup = createPopup();
    VBox popupBox = createPopupBox("Drones");

    for (DeliveryService service : DeliveryService.services.values()) {
        populateDronePopup(service, popupBox);
    }

    validateDronePopup(dronePopup, popupBox);
}
```



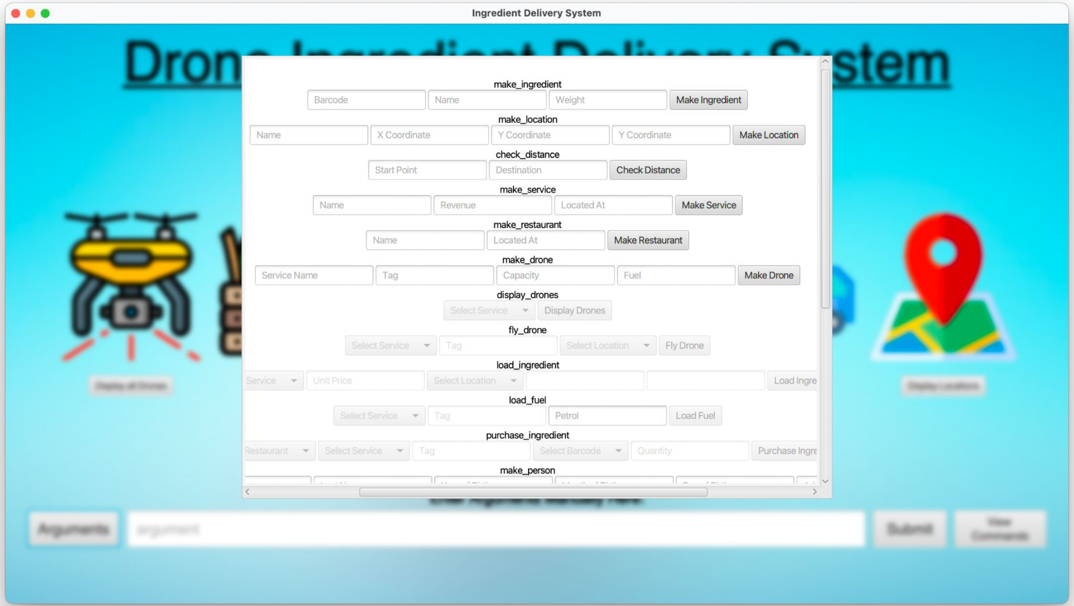
```java
if (DeliveryService.services.size() > 0) {
    for (DeliveryService service : DeliveryService.services.values()) {
        displayDronesComboBox.getItems().add(service.getName());
    }
} else {
    displayDronesComboBox.setDisable(true);
    displayDronesButton.setDisable(true);
}
```



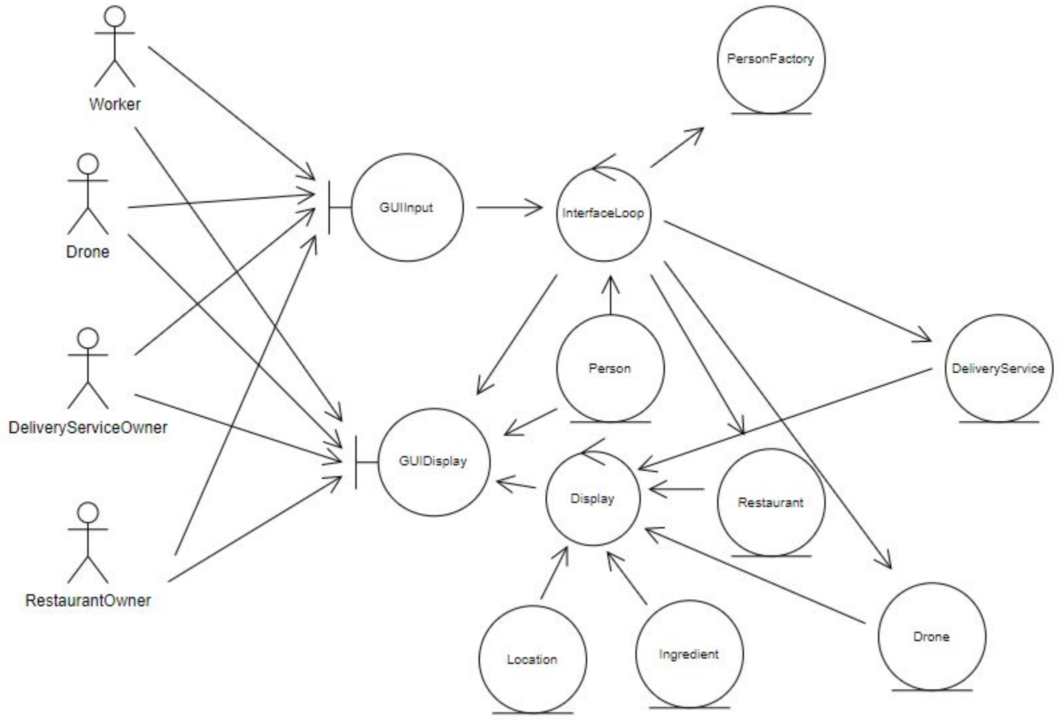
It's demo time...

Advantages and Impacts of the GUI

- Data can be **displayed** easily
- GUIs are more **intuitive**, resembling visual file representation systems
- Design choices can **guide users** towards the important details
- Clients can easily find **available commands** and their parameters



The Robustness of our Creation



The interaction between the user interface and domain classes is funneled through two controller classes.

- ***InterfaceLoop*** is responsible for user inputs that change the state of our system
- the ***Display*** class manages the output of feedback and data to the GUI

04

FUTURE CHANGES

What would we do with more iterations to improve the design further?

Future Changes & Ideas

- Optimising purchases so that restaurants can get ingredients from any *DeliveryService* that is nearest to them
- Separating out controllers to delegate responsibility across classes
- Moving Form instantiation to a separate class (*Main* responsibility)
- Add more types of Workers and Drones (and Drone-related interfaces) that add more functionality
- Add roles so different users of the client have different permissions within the system.

THANKS!
QUESTIONS?

2725 sloc,4278 lines,183 commits,45341 additions,14378 deletions,??? hours spent