CPSC 501

Assignment 1

Version Control and Refactoring

Se Yeon Sim 10102631
Jonathan Hudson
October 7, 2022

I've chosen to refactor parking lot java project. The code does not contain an inheritance structure at the start of the project. I've added some inheritance structure while completing the project. The code contains an inheritance structure for different types of parking slot, and different types of parking charge strategy.

The version controlled code is hosted in a public repository on Gitlab, which can be found here:
https://gitlab.cpsc.ucalgary.ca/sysim/CPSC501A1

Refactoring 1: Replace Error Code with Exception

SHA Number: 4b19bb6c, 5196b127

1. What code in which files was altered.
   - ParkingFullException.java class was added, extending the Exception class. I've chosen to use a custom exception class rather than Exception class itself in order to give more descriptive information as to why the exception was thrown. The class incudes a single constructor.(SHA Number: 4b19bb6c)

```
public class ParkingFullException extends Exception {
    public ParkingFullException(String message){
        super(message);
    }
}
```

   - InvalidVehicleNumberException.java was added, extending the Exception class. I've chosen to use a custom exception class rather than Exception class itself in order to give more descriptive information as to why the exception was thrown. The class incudes a single constructor. (SHA Number: 5196b127)

```
+ public class InvalidVehicleNumberException extends Exception{
+     public InvalidVehicleNumberException(String message){
+         super(message);
+     }
+ }
```

- In ParkingLot.java, park method was changed to throw a ParkingFullException. (SHA Number: 4b19bb6c)
  - getNextAvailableCarSlot() method and getNextAvailableMotorBikeSlot() was changed from return of null to throw ParkingFullException with ParkingFull error message.
- In ParkingLot.java, removeVehicle method was changed to throw a InvalidVehicleNumberException. (SHA Number: 5196b127)
  - getMotorBikeSlotByVehicleNumber(String VehicleNumber) and getCarSlotByVehicleNumber(String VehicleNumber) was changed form return of null to throw InvalidVehicleNumber with error message.

2. What needed to be improved? That is, what "bad code smell" was detected? Use the terminology found in the Fowler text.
   - There was a need to make error indication more concise rather than just throwing NullPointerException. Previously, getNextAvailableCarSlot() / getNextAvailableMotorBikeSlot() and getMotorBikeSlotByVehicleNumber(String VehicleNumber) / getCarSlotByVehicleNumber(String VehicleNumber) return null if no parking slot is found. There is no formal code smell in textbook which refers to NullPointerException. However, I assumed throwing NullPointerException does not provide sound information as to why the Exception was thrown.
3. What refactoring was applied? What steps did you follow? Use the terminology and mechanics outlined in the Fowler text.
   - The refactoring applied was to replace the error code with the exception.
   - I have taken following steps to refactor:
     1. I found all method which returns null pointer error.
     2. I created FullParkingException class and InvalidVehicleNumberException class.

3. Inside the method, instead of returning null pointer errors when no parking slot is found, I changed to throw newly created exception classes.
4. Changed the method signature so that it contains information about the exception being thrown.

4. What code in which files was the result of the refactoring.
   - ParkingFullException.java and InvalidVehicleNumberException.java are added as the result of the refactoring
   - In ParkingLot.java
     - Throwing InvalidVehicleNumberException in getMotorBikeSlotByVehicleNumber(String VehicleNumber)/ getCarSlotByVehicleNumber(String VehicleNumber) methods are the result of refactoring
     - Throwing ParkingFullException in getMotorBikeSlotByVehicleNumber(String VehicleNumber) / getCarSlotByVehicleNumber(String VehicleNumber) methods are the result of refactoring

5. How was the code tested?
   - I added additional a unit test to make sure the ParkingFullexception is always thrown in specific condition. This unit test involves parking vehicle when there is no parking slot available or when all parking slots are taken by other vehicles.
   - I also added a unit test to make sure InvalidVehicleNumberException is always thrown. This unit test involves calling getMotorBikeSlotByVehicleNumber(String VehicleNumber) / getCarSlotByVehicleNumber(String VehicleNumber) method when there is no parked vehicle or when there is no parked vehicle with given VehicleNumber.

6. Why is the code better structured after the refactoring?
    - The code is better structured after the refactoring because error codes are quite primitive and provide not enough information as to what went wrong. The new exception classes provide more descriptive information on what went wrong and shows the root cause of the error.
7. Does the result of the refactoring suggest or enable further refactorings?
    - I cannot think of further refactoring with the result of the refactoring.

Refactoring 2: Replace Conditional with Polymorphism

SHA Number: ed4d2a44

Branch: refactorParkingCharge

1. What code in which files was altered.
    - Created Parent Class of ParkingCharge.java and Subclasses of CarParkingCharge.java and MotorBikeParkingCharge.java matching the branches of the conditional. In Subclasses, created a shared method and move code from the corresponding branch of the conditional to it.
    - In ParkingLot.java removeVehicle(ParkingTicket ticket) was altered. Replaced conditional with the relevant method call getParkingCharge(hours).
    - In ParkingTicket.java, new attribute private ParkingCharge chargeStrategy is created which assigns chargeStrategy depending on ParkingSlot taken. And getChargeStrategy method is created.
2. What needed to be improved? That is, what “bad code smell” was detected? Use the terminology found in the Fowler text.
    - The bad code smell of “Switch Statements” was detected. The code had a complex switch operator. The Switch structure is typically spread throughout many methods which makes the code difficult to extend.

3. What refactoring was applied? What steps did you follow? Use the terminology and mechanics outlined in the Fowler text.
   - Refactoring applied was to Replace Conditional with Polymorphism
   - I have taken following steps to refactor:
     1. I created hierarchy of classes. Parent class of ParkingCharge.java was created along with subclasses of CarParkingCharge.java and MotorBikeParkingCharge.java
     2. Replace Type Code with State/Strategy: subclasses of ParkingCharge is created depending on VehicleType.
     3. For each hierarchy subclasses of ParkingCharge.java, define the getParkingCharge method and copy the corresponding conditional branch to corresponding subclass method.
     4. Delete branch from the conditional
     5. Repeat replacement until the conditional is empty. Then delete the conditional and declared the ParkingCharge. Java abstract
4. What code in which files was the result of the refactoring.
   - CarParkingCharge.java was newly created extending ParkingCharge.java class. This subclass override getParkingCharge method.

```java
public class CarParkingCharge extends ParkingCharge{
    @Override
    public double getParkingCharge(int hours) {
        double parkingCost = 5.0;
        if (hours > 1) {
            parkingCost += (hours - 1) * 3;
        }
        if (hours > 48 ){
            parkingCost -= (hours/48) * 20;
        }
        return parkingCost;
    }
}
```

```java
public class MotorBikeParkingCharge extends ParkingCharge{
    @Override
    public double getParkingCharge(int hours) {
        double parkingCost = 3.0;
        if (hours > 1) {
            parkingCost += (hours - 1) * 2;
        }
        if (hours > 48 ){
            parkingCost -= (hours/48) * 18;
        }
        return parkingCost;
    }
}
```

- MotorBikeParkingCharge.java was newly created extending ParkingCharge.java. This subclass override getParkingCharge method.

```java
+
+ abstract class ParkingCharge {
+     public abstract double getParkingCharge(int hours);
+ }
```

- Abstract class ParkingCharge.java was created with abstract getParkingCharge method.

```java
    public ParkingTicket(int slotNumber, String vehicleNumber, VehicleType
vehicleType, Date date) {
        super();
        this.slotNumber = slotNumber;
        this.vehicleNumber = vehicleNumber;
        this.setVehicleType(vehicleType);
        if(this.vehicleType==VehicleType.CAR){
            this.chargeStrategy = new CarParkingCharge();
        } else if (this.vehicleType == vehicleType.MOTORBIKE) {
            this.chargeStrategy = new MotorBikeParkingCharge();
        }
        this.date = date;

    }
```

- In ParkingTicket.java, new chargeStrategy parameter is created. This creates subclass of ParkingCharge depending on VehicleType
- In ParkingLot.java Switch and case code is deleted and replaced with polymorphism

5. How was the code tested?
   - The code was mainly tested with a series of changing Junit tests testing various parking hours with different parking charge strategy.
6. Why is the code better structured after the refactoring?
   - The code is better structured after the refactoring because this technique adheres to the Tell-Don't-Ask principle. Instead of asking an object about its state and then performing actions based on this, it's much easier to simply tell the object what it needs to do and let it decide for itself how to do that. Since this use polymorphism, It emphasize the benefits of Object-Oriented Structure.
   - After refactoring, it removed duplicated code.
   - If I need to add a new ParkingCharge Strategy, all I need to do is add a new subclass without touching the existing code.
7. Does the result of the refactoring suggest or enable further refactorings?
   - Yes, I can apply same refactoring technique for ParkingLot.java. I used a lot of conditional statement in ParkingLot.java. Hence, this file contains a lot of duplicated code, I can remove duplicated code by replacing conditional with polymorphism.
   - I can also replace temp (parkingCost) with Query to improve code readability. After refactoring, temp variable resulted in long expression that uses multiple method calls.

Refactoring 3: Extract SubClass

SHA Number: eb7a2c0, 71077205

1. What code in which files was altered.
   - Created CarSlot extending ParkingSlot
   - Created MotorBikeSlot extending ParkingSlot
   - Made ParkingSlot.java abstract class and added ParkingSlotType attribute
   - Created enum ParkingSlotType.java
   - In ParkingLot.java, most of the method that uses ParkingSlot.java was changed to either CarSlot or MotorBikeSlot
   - In ParkingTicket.java ParkingChargeStrategy gets assigned depending on ParkingSlotType
2. What needed to be improved? That is, what "bad code smell" was detected? Use the terminology found in the Fowler text.
   - Large Class was the bad code smell. This can be improved by extracting subclass since part of the behaviour of the large class can be implemented in different ways.
3. What refactoring was applied? What steps did you follow? Use the terminology and mechanics outlined in the Fowler text.
   - The refactoring applied was Extract Subclass
   - I have taken following steps to refactor:
     1. I created a new CarSlot.java and MotorBikeSlot.java from ParkingSlot.java
     2. I created enum ParkingSlotType.java
     3. Created new additional data of ParkingSlotType to ParkingSlot.java
     4. Find all class to the constructor of the parent class. When the functionality of a subclass is necessary, replaced the parent constructor with the subclass constructor
     5. Move the necessary fields from parent class to subclass.

4. What code in which files was the result of the refactoring.
   - In ParkingTicket.java

```
    public ParkingTicket(int slotNumber, String vehicleNumber, VehicleType
vehicleType, ParkingSlotType slotType, Date date) {
        super();
        this.slotNumber = slotNumber;
        this.vehicleNumber = vehicleNumber;
        this.slotType = slotType;
        this.setVehicleType(vehicleType);
        if(this.vehicleType==VehicleType.CAR){
        if(this.slotType==ParkingSlotType.CARSLOT){
            this.chargeStrategy = new CarParkingCharge();
        } else if (this.vehicleType == vehicleType.MOTORBIKE) {
        } else if (this.slotType == ParkingSlotType.MOTORBIKESLOT) {
            this.chargeStrategy = new MotorBikeParkingCharge();
        }
        this.date = date;
@@ -58,6 +62,10 @@ public class ParkingTicket {
        return chargeStrategy;
    }

    public ParkingSlotType getSlotType() {
        return slotType;
    }
```

   - Created enum ParkingSlotType
   - Made ParkingSlot class abstract add new parameter of parkingSlotType
   - Created subclasses of ParkingSlot.java: CarSlot and MotorBikeSlot
   - In ParkingLot.java
     - Find all class to the constructor of the parent class. When the functionality of a subclass is necessary, replaced the parent constructor with the subclass constructor
5. How was the code tested?

- Code was tested with a series of Junit tests testing constructor of MotorBikeSlot and CarSlot depending on VehicleType. Tested getSlotType() whether it returns correct ParkingSlotType.

6. Why is the code better structured after the refactoring?
   - The code is better structured because if there were to be a new type of parking slots added, code would only need to change in a single place.
7. Does the result of the refactoring suggest or enable further refactorings?
   - I cannot think of further refactoring as the result of the refactoring.

Refactoring 4: Extract Method

SHA Number: a1aa7ddb

1. What code in which files was altered.
   - In ParkingLot.java initializeParkingSlots method was altered
2. What needed to be improved? That is, what "bad code smell" was detected? Use the terminology found in the Fowler text.
   - InitializeParkingSlots method was quite long. The more lines found in a method, the harder it's to figure out what the method does. As such, the code smell for this refactoring was long method.
3. What refactoring was applied? What steps did you follow? Use the terminology and mechanics outlined in the Fowler text.
   - The refactoring applied was to extract method. I chose to create new methods of initializeMotorBikeSlots and initializeCarSlots.
   - I have taken following steps to refactor:
     1. Created a new methods of initializeMotorBikeSlots and initializeCarSlots and name it in a way that makes it purpose self-evident.
     2. Copy the relevant code fragment to initializeMotorBikeSlots and initializeCarSlots methods. Delete the fragment from initializeParkingSlots method and put a call for the new method there instead.

3. I pass the variables that are declared prior to the code that I'm extracting to the parameters of new method in order to use the values previously contained in them.

4. What code in which files was the result of the refactoring.

- In ParkingLot.java

```
     for (int i = 1; i <= numberOfMotorBikeParkingSlots; i++) {
    initializeMotorBikeSlots(numberOfMotorBikeParkingSlots,motorBikeSlots);
    initializeCarSlots(numberOfCarParkingSlots,carSlots);
    return true;
}

public void initializeMotorBikeSlots(int numOfSlots, List<MotorBikeSlot> motorBikeSlots){
     for (int i = 1; i <= numOfSlots; i++) {
         motorBikeSlots.add(new MotorBikeSlot(i));
     }
     System.out.printf("Created a motorbike parking lot with %s slots %n",
mberOfMotorBikeParkingSlots);

     for (int i = 1; i <= numberOfCarParkingSlots; i++) {
     System.out.printf("Created a motorbike parking lot with %s slots %n", numOfSlots);
}
public void initializeCarSlots(int numOfSlots, List<CarSlot> carSlots){
     for (int i = 1; i <= numOfSlots; i++) {
         carSlots.add(new CarSlot(i));
     }
     System.out.printf("Created a car parking lot with %s slots %n", numberOfCarParkingSlots);
     return true;
     System.out.printf("Created a car parking lot with %s slots %n", numOfSlots);
}
```

5. How was the code tested?

- The code was tested with the same set of Junit test as previous one because none of the functionality should have changed. These tests includes testing initialization of parking lot.

6. Why is the code better structured after the refactoring?

- The code is better structured because the longer a method or function is, the harder it becomes to understand and maintain it. Now I have clear and understandable code.

7. Does the result of the refactoring suggest or enable further refactorings?

- No, I cannot think of any further refactoring as a result of the refactoring

Refactoring 5: Rename Method

SHA Number: b819e64f

1. What code in which files was altered.
   - ParkingLot.java
     - Park() method was renamed to parkVehicleInParkingLot()
     - removeVehicle() was renamed to removeVehicleFromParkingLot()
   - ParkingSlot.java
     - removeVehicleSlot() was renamed to removeVehicleFromParkingSlot()
     - placeVehicleSlot was renamed to placeVehicleInParkingSlot()
2. What needed to be improved? That is, what "bad code smell" was detected? Use the terminology found in the Fowler text.
   - There was a code smell of several methods having similar name across various classes, perhaps causing some confusion.
3. What refactoring was applied? What steps did you follow? Use the terminology and mechanics outlined in the Fowler text.
   - The applied refactoring was to rename method within ParkingLot class and ParkingSlot class. These methods were renamed to have name include a send of what is happening and where it is happening.
4. What code in which files was the result of the refactoring.
   - In ParkingLot.java

```
-     public ParkingTicket park(Vehicle vehicle) throws ParkingFullException
  {
+     public ParkingTicket parkVehicleInParkingLot(Vehicle vehicle) throws
  ParkingFullException {
```

```
-     public double removeVehicle(ParkingTicket ticket) throws
  InvalidVehicleNumberException{
+     public double removeVehicleFromParkingLot(ParkingTicket ticket) throws
  InvalidVehicleNumberException{
```

- In ParkingSlot.java

```
public void removeVehicleSlot() {
public void removeVehicleFromParkingSlot() {
```

```
public void placeVehicleSlot(Vehicle parkVehicle) {
public void placeVehicleInParkingSlot(Vehicle parkVehicle) {
```

5. How was the code tested?
   - The code was tested with the same set of Junit tests from the previous refactors, however many calls tests needed to be changed to reflect these renames. Carefully renaming all the method calls in the unit tests, and ensuring the result were the same was key in testing.
6. Why is the code better structured after the refactoring?
   - As mentioned above, renaming structured the code better as it gave a sense as to what the specific method was doing in context of the class it was in.
7. Does the result of the refactoring suggest or enable further refactorings?
   - No, I cannot think of any further refactoring as a result of the refactoring