Nathan Jackson
30000224

CPSC 501 – Assignment 1 - Refactoring

The code I've chosen to refactor is some object-oriented code representing a parking garage. The code contains an inheritance structure for various vehicle types, a parking garage containing levels, rows, and spots, and some entry/exit gates to the parking garage to handle parking tickets.

The version controlled code is hosted in a public repository on GitHub, which can be found here: https://github.com/natejacko/CPSC501-Assignment1

Refactoring #1 – Bus class refactor

GitHub hash codes: ae0b08a, 36c9c16, e2985ca, 363b9dc, da88067

This is by far the largest refactor I did on this assignment, and the explanation may be slightly excessive for this one. But to be comprehensive, I have gone into some detail.

1. **What needed to be improved?**
   There were several things which needed to be improved in the Bus.java file. The main code smell however was an incredible amount of duplicated code. The whole parking functionality (in ParkingGarage, ParkingGarageLevel, and ParkingSpot) was nearly the same for both a bus and a vehicle, though since a bus did not inherit from a vehicle and was the only vehicle type which occupies multiple spots, it led to all the duplicated code. An additional code smell that can be detected with this code is shotgun surgery to an extent; if a change in the vehicle class was needed, the bus class would also need to change to mirror this.
2. **What refactoring was applied?**
   The main refactoring applied was to extract class, where most of the code and functionality of the Bus class was extracted into the Vehicle class (which Bus was changed to inherit from). By doing this, I was able to move up the "parkingSpotsNeeded" field into Vehicle class to allow all sub-classes to inherit (and use) this value, which will make parking consistent across all vehicles. The rest of the refactoring mainly just involved systematically removing the duplicate code from the parking classes and testing after each incremental change, ensuring not to break anything.
3. **What code in which files was altered?**
   The (largest) refactoring I've done involved altering quite a few files:
   i. Bus.java
      - All the following methods were removed from being declared in the Bus class, as it was inheriting them from the Vehicle class:
        - `public ParkingSpotSize getParkingSpotSize()`
        - `public void receiveParkingTicket(ParkingTicket pt)`
        - `public ParkingTicket getParkingTicket()`
        - `public boolean payParkingTicket(double amount)`
        - `public void park(ParkingSpot ps)`
        - `public void leaveParkingSpots()`
      - Additionally, the method `public int getParkingSpotsNeeded()` was removed along with the member variable `protected int parkingSpotsNeeded` as they were extracted into the parent Vehicle class.
      - After refactoring, the Bus class looks like the following:

```
public class Bus extends Vehicle
{
    public Bus(String licensePlate, int parkingSpotLength) { … }
    protected boolean canParkInSpot(ParkingSpot ps) { … }
}
```

ii. CompactCar.java, NormalCar.java, Motorbike.java, Vehicle.java, Car.java
- All the above files needed to be altered to accommodate for moving code from the Bus class to the Vehicle class. The main change was adding the member variable `protected int parkingSpotsNeeded` to Vehicle (thus sub-classes needed to instantiate it).
- Additionally, I applied extract class to CompactCar and NormalCar by adding a new super-class, Car. This was more so to group the cars in the event of future changes.
- After refactoring, the Vehicle class added the following:

```
…
protected int parkingSpotsNeeded;
…
public int getParkingSpotsNeeded() { … }
…
```

- After refactoring, an abstract Car class extending Vehicle was added, with a protected constructor:

```
protected Car()
{
    parkingSpotsNeeded = 1;
}
```

- After refactoring, the NormalCar and CompactCar classes now extended Car and added a call to `super()` in their constructor

iii. ParkingSpot.java, ParkingGarage.java, ParkingGarageLevel.java, ParkingGarageEntranceGate.java
- All the above files needed to be heavily refactored as a result of moving code from the Bus class to the Vehicle class, which resulted in heavily duplicated code, namely the following:
  - ParkingSpot
    - `private Bus currentBus;`
      - Removed
    - `public boolean isSpotFree()`
      - Removed check for `currentBus == null;`
    - `public boolean spotFreeAndCanFitBus(Bus b)`
      - Removed
    - `public boolean parkBus(Bus b)`
      - Removed
    - `public void freeUpSpot()`
      - Removed `currentBus = null;`
  - ParkingGarage
    - `public boolean parkBus(Bus b)`
      - Removed
    - `public boolean removeBusFromGarage(Bus b)`
      - Removed
  - ParkingGarageLevel
    - `public boolean parkVehicle(Vehicle v)`
      - Moved functionality from `parkBus(Bus)` into this method
    - `public boolean parkBus(Bus b)`
      - Removed
    - `private boolean parkVehicle(int rowIndex, int spotIndex, Vehicle v)`
      - Moved functionality from `parkBus(int, int, Bus)` into this method
    - `private boolean parkBus(int rowIndex, int spotIndex, Bus b)`
      - Removed
  - ParkingGarageEntranceGate
    - `public void getParkingTicketForBus(Bus b)`
      - Removed

4. **How was the code tested?**
   The code was mainly tested with a series of changing JUnit tests testing various parking outcomes in the garage (enough spots, not enough spots, filling spots with various vehicle types, freeing up spots, etc.). Due to the incremental changes which took place in this refactoring, the unit tests were also periodically changed to reflect the new changes.
5. **Why is the code better structured after the refactor?**
   The code is much better structured as there is far less duplicated code in both the vehicle class hierarchy, as well as the parking garage functionality. If there were to be a new vehicle added (perhaps which utilized many parking spots), or if the way parking spots were allocated, code would only need to change in a single place, instead of in two.

Refactoring #2 – Long method refactor

GitHub hash code: 3fe8fb3

1. **What needed to be improved?**
   After the previous refactoring of the Bus class, the `public boolean parkVehicle(Vehicle v)` method was quite long, and wasn't just handling parking, but also looking for valid parking spots. As such, the code smell for this refactor was long method.
2. **What refactoring was applied?**
   The refactoring applied was to extract method. I chose to create a new method `private int getNextFreeSpotInRow(int rowIndex, Vehicle v)` which does the scanning for a free parking spot (or spots) in a given row with a given vehicle. This gives some more meaning to the `parkVehicle` method, it simply will only ask the `getNextFreeSpotInRow` for a free spot, and if there is one available, it will park in these spot(s).
3. **What code in which files was altered?**
   As mentioned above, the method `private int getNextFreeSpotInRow(int rowIndex, Vehicle v)` was added to the file ParkingGarageLevel.java, and the method `public boolean parkVehicle(Vehicle v)` was modified to call(s) to this new private method until a valid parking spot (or spots) was found.
4. **How was the code tested?**
   The code was tested with the same set of JUnit tests as refactor #1, as none of the functionality should have changed, simply the code was moved around. These tests included testing various permutations of parking in the garage. As the newly added method was intended to be private as well, the JUnit tests were performed on only the public method (the caller).
5. **Why is the code better structured after the refactor?**
   The code is better structured, as there is now more logical meaning behind the `parkVehicle` method. It simply only handles parking when given a valid spot. Despite the newly added method `getNextFreeSpotInRow` only being used in this one spot currently, if there were to be new functionality added to the parking garage level, such as a query system for finding available parking spots, this new method can be utilized without needed to park a vehicle.

Nathan Jackson
30000224

Refactoring #3 – Better error handling

GitHub hash code: cf97e3d

1. **What needed to be improved?**
   After refactoring #2, there was a need to make error indication more concise. The newly added `getNextFreeSpotInRow` method in the ParkingGarageLevel class uses a -1 error code to indicate no parking spots found (this code was a remainder from the original pre-refactored code, but has become more apparent due to refactoring #2). No formal code smell in the textbook refers to integer error codes, however integer error codes are generally a bad thing to have in code, and especially in the case of the `getNextFreeSpotInRow` method where the integer return is supposed to indicate which spot is available.
2. **What refactoring was applied?**
   The refactoring applied was to replace the error code with an exception, and then wrap all calls to this method in try/catch blocks. To perform this, a new exception class was added "NoFreeParkingSpotsException", which is thrown in the `getNextFreeSpotInRow` method rather than -1 when no free spot(s) can be found with the given vehicle.
3. **What code in which files was altered?**
   The refactoring involved changing two code files:
   i. NoFreeParkingSpotsException.java
      - This class was added, extending the Exception base class. I chose to use a custom exception class rather than Exception itself, to be more descriptive as to why the exception was thrown. The class includes a single constructor:

```
NoFreeParkingSpotsException(String s)
{
        super(s);
}
```

   ii. ParkingGarageLevel.java
      - The `getNextFreeSpotInRow` method was changed to throw a NoFreeParkingSpotException rather than -1:

```
return -1;
throw new NoFreeParkingSpotsException("Cannot find enough free parking spots for
vehicle in row " + (rowIndex + 1));
```

      - The `parkVehicle` method was changed from checking the return of `getNextFreeSpotInRow` != -1, to instead wrapping this call in a try/catch block:

```
try
{
        int spot = getNextFreeSpotInRow(row, v);
        return parkVehicle(row, spot, v);
}
catch (NoFreeParkingSpotsException e)
{
        System.out.println(e);
}
```

4. **How was the code tested?**
   The code running the previous bus refactoring JUnit tests, as well, I added an additional unit test to make sure the exception was always hit. This unit test involved parking several cars, then removing a select few, such that there existed enough spots in the parking garage to fit a bus, but there were not enough spots in a continuous row. Since the exception is thrown in a private method I cannot call directed, when caught in the public method, I am logging the message to the console and running to unit test to make sure the exception message is displayed.

5. **Why is the code better structured after the refactor?**
   The code is better structured after the refactor because error codes are quite primitive and indicate little reason as to what went wrong. In this specific case, if the code did not check for the -1 return, it would have tried to park in index -1, causing an unhandled exception ArrayIndexOutOfBounds, crashing the program. The new exception shows the actual root causing for the error occurring.

Refactoring #4 – Renaming to be more concise

GitHub hash code: [7820833]

1. **What needed to be improved?**
   Due to the functionality changes caused in the previous refactors, there was a code smell of several methods having the same name across various classes, perhaps causing some confusion.
2. **What refactoring was applied?**
   The applied refactoring was to rename various method within Vehicle, ParkingGarage, ParkingGarageLevel, and ParkingSpot classes. These methods were renamed to have the name include a sense of what is happening. For example, instead of saying "parkVehicle" in the ParkingGarage class, which may imply parking in a parking spot, I've instead renamed it to "parkVehicleInGarage".
3. **What code in which files was altered?**
   The following code files had methods renamed:
   i. Vehicle.java
      - `public void park(ParkingSpot ps)` changed to `public void parkVehicle(ParkingSpot ps)`

   ii. ParkingGarage.java
      - `public boolean parkVehicle(Vehicle v)` changed to `public boolean parkVehicleInGarage(Vehicle v)`

   iii. ParkingGarageLevel.java
      - `public boolean parkVehicle(Vehicle v)` changed to `public boolean parkVehicleOnLevel(Vehicle v)`
      - `private boolean parkVehicle(int rowIndex, int spotIndex, Vehicle v)` changed to `private boolean parkVehicleInSpots(int rowIndex, int spotIndex, Vehicle v)`

   iv. ParkingSpot.java
      - `public boolean parkVehicle(Vehicle v)` changed to `public boolean parkInSpot(Vehicle v)`
4. **How was the code tested?**
   The code was tested with the same set of JUnit tests from the previous refactors, however many calls in the unit tests needed to be changed to reflect these renames. Carefully renaming all the method calls in the unit tests, and ensuring the results were the same was key in testing.
5. **Why is the code better structured after the refactor?**
   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.

Nathan Jackson
30000224

Refactoring #5 – Replacing temporary variables

GitHub hash code: 30aa47f

1. **What needed to be improved?**
   The final refactoring done for this assignment was to cleanup the parking garage exit gate and parking ticket. The code smell for this, though not as obvious as refactoring #2 is also long methods. Though the size of the methods in the above-mentioned classes are not nearly as long as refactoring #2, the amount of temporary calculations done within these methods complicates things too much and causes messy looking methods.
2. **What refactoring was applied?**
   The refactoring applied was to replace temporary variables with queries. The variables replaced in the ParkingGarageExitGate class were for converting minutes into half hour increments, and for getting total parking ticket price. The variables replaced in the ParkingTicket class were for getting current time in milliseconds and for converting milliseconds to minutes.
3. **What code in which files was altered?**
   The two code files modified were:
   i. ParkingGarageExitGate.java
      - Two new methods were created `convertMinutesToHalfHour` and `getParkingTicketPrice` to perform calculations originally calculated in `getParkingTicketPrice`. The changed code is as follows:

```
public double getParkingTicketPrice(ParkingTicket pt)
{
	long ticketTime = pt.getElapsedTicketTime();
	int halfHourAmount = convertMinutesToHalfHour(ticketTime);
	return getParkingTicketPrice(halfHourAmount);
}
private int convertMinutesToHalfHour(long minutes)
{
	return (int) Math.ceil(minutes / 30.0);
}
private double getParkingTicketPrice(int halfHourAmount)
{
	return halfHourAmount * parkingGarage.getParkingRatePerHalfHour();
}
```

   ii. ParkingTicket.java
      - Two new methods were created `getCurrentTimeMilli` and `convertMilliToMinutes` to perform calculations originally calculated in `getElapsedTicketTime` and `ParkingTicket`. The changed code is as follows:

```
public ParkingTicket()
{
	startTime = getCurrentTimeMilli();
}
public long getElapsedTicketTime()
{
	long endTime = getCurrentTimeMilli();
	long elapsedTimeMinutes = convertMilliToMinutes(endTime) -
convertMilliToMinutes(startTime);
	return elapsedTimeMinutes;
}
private long getCurrentTimeMilli()
{
	return Instant.now().toEpochMilli();
}
```

```
private long convertMilliToMinutes(long milli)
{
	long seconds = milli / 1000;
	return seconds / 60;
}
```

4. **How was the code tested?**
   The code was tested by adding a variety of JUnit test cases testing the parking ticket calculations, including less than and greater than a minute/30 minutes/1 hour parking ticket times. As running these unit tests takes a long time, and the methods changed are private, I also did some testing of manually entering parking ticket start/end times to test the price calculations.
5. **Why is the code better structured after the refactor?**
   The code is better structured as the methods are more readable in what they do, and single calculations are isolated to small method calls with a specific meaning.