

CPSC 501 Assignment 1

Refactoring

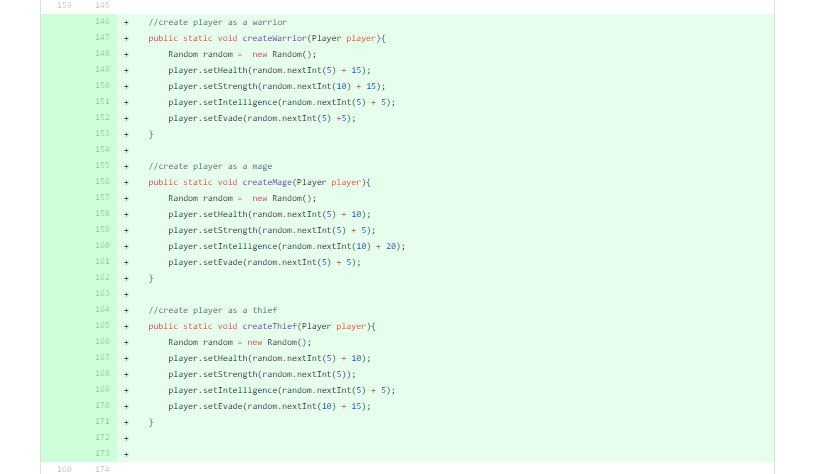
Johnny Phuong Chung | 10036448 | Oct 13 2017

# Refactoring #1 – Extract method

In the original version of the refactored code, the most glaring “bad code smell” was the very **long method** for the Driver class, in which the main method was responsible for almost all of the game logic:



As such, the first and most logical refactoring technique applied was **Extract Method**, wherein the large method was decomposed into smaller methods. Specifically, the conditional logic for the Player’s class creation were grouped together and separated into the createWarrior method, createMage method, createThief methods (refer to GitHub version control logs):

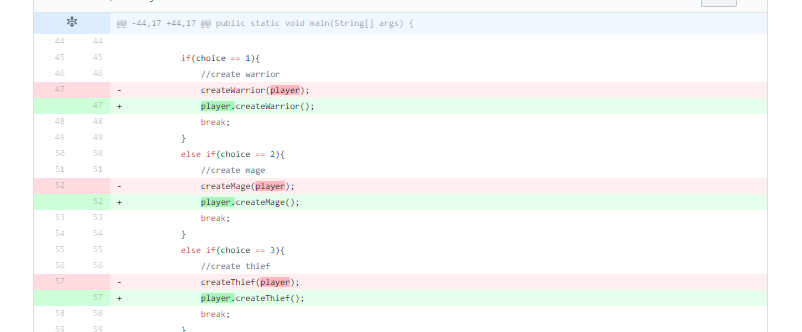


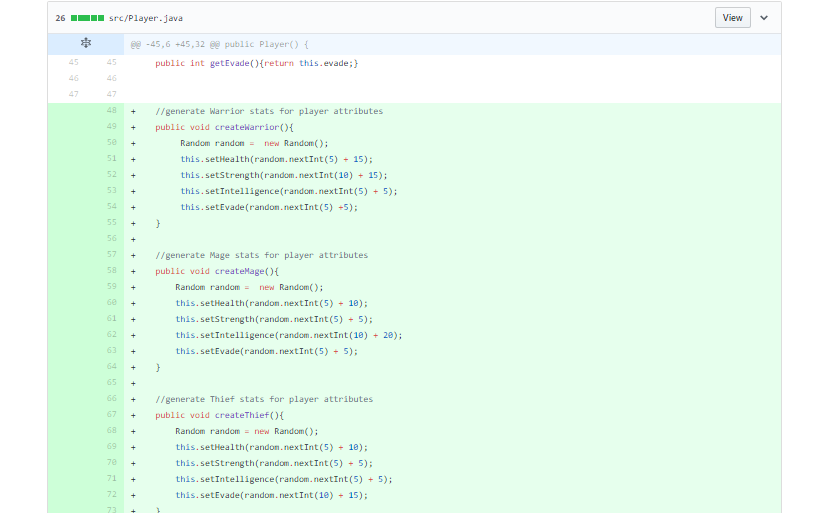
This resulted in a smaller main method that was no longer trying to do as many things at once, and this refactoring also suggested that the following refactoring was possible: **Move Method**.

## Refactoring #2 – Move method

Following the first refactoring, another bad code smell was apparent in the code that needed to be improved: **Feature Envy**. Specifically, the logic for the Player’s class creation in the Driver class was using too many instance variables and methods of the Player class itself.

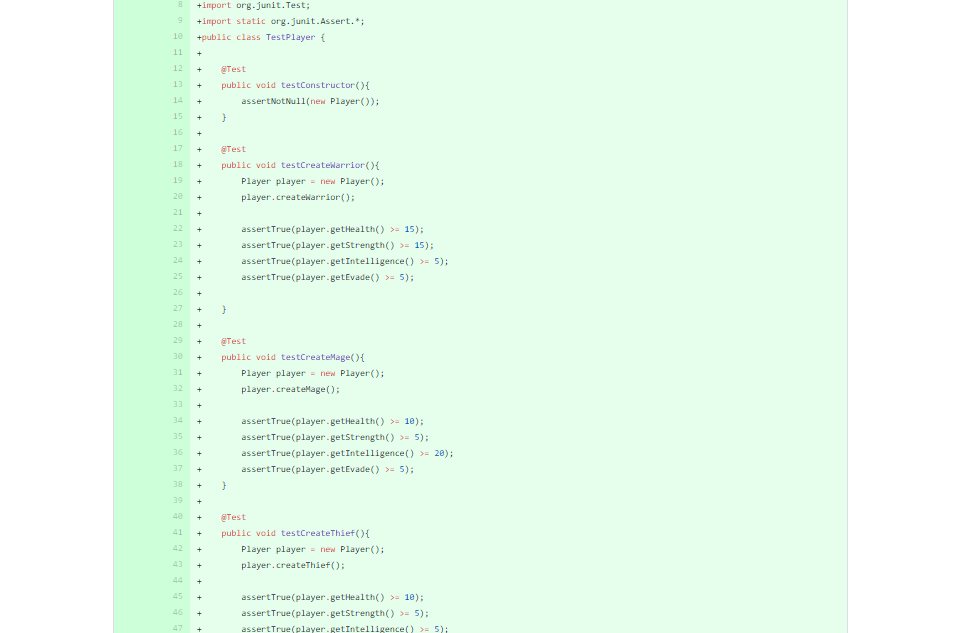
Therefore, it was apparent that the code could be improved with a **Move Method** refactoring. That is, the createWarrior, createMage, and createThief methods were moved to the Player class (refer to GitHUb version control logs):





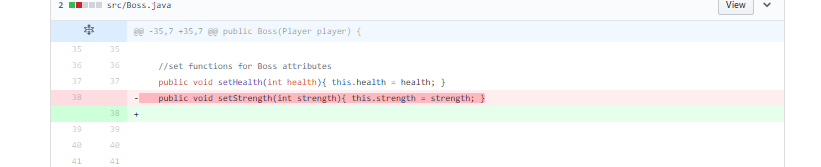
From this structural refactoring, the **Feature Envy** of the Driver class for the Player class is resolved. Additionally, the refactoring had an additional effect of reducing the size of the Driver class, thus also reducing another bad code smell: **Large class**, of which the Driver class was certainly guilty of.

The moved methods would then be tested with JUnit and the TestPlayer class, with specific unit tests for the proper stat propagation for each Player class:



## REFACTORING #3 – Remove Setting Method

Upon reviewing the Boss class, it was apparent that the setting method for the strength instance variable was not ever being used, and was thus removed as per the **Remove Setting Method** refactoring (refer to GitHub version control logs):

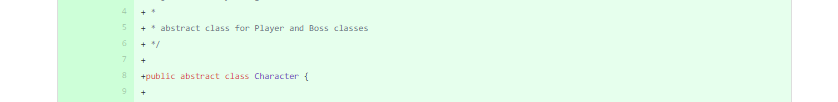


This refactoring did remove extraneous code from the Boss class, but as a result highlighted possible warning signs of a another bad code smell: if not for the displayAttributes method, the Boss class would be a **Data Class** (a class with no behaviour outside of set/get methods).

## Refactoring #4, #5, #6 – Extract SuperClass, PULL UP METHOD, PULL UP Field

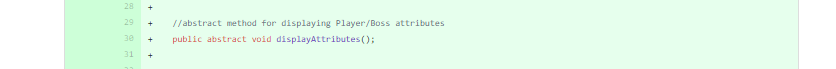
The next major code smell that was noticed was the **duplicate code** in both the Player and Boss classes. Specifically, the instance variables were mostly the same and constructors for each class was doing similar things. This suggested that there was a potential class hierarchy that could be created.

As such, the **Extract Superclass** refactoring technique was used, wherein the Character abstract superclass was created, and Player and Boss classes were then declared as extensions of Character. Both the Player and Boss classes also had a displayAttribute method with the same signature, and thus the Character superclass was given an abstract method with that name and signature (refer GitHub version control logs):

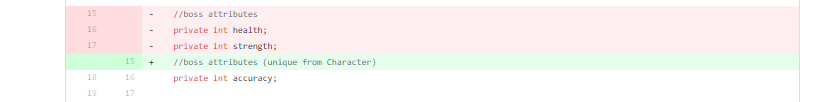


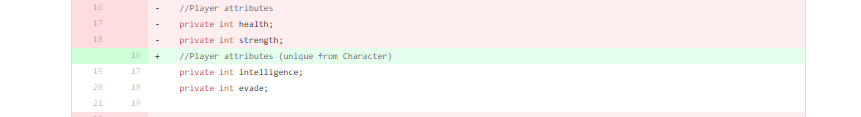


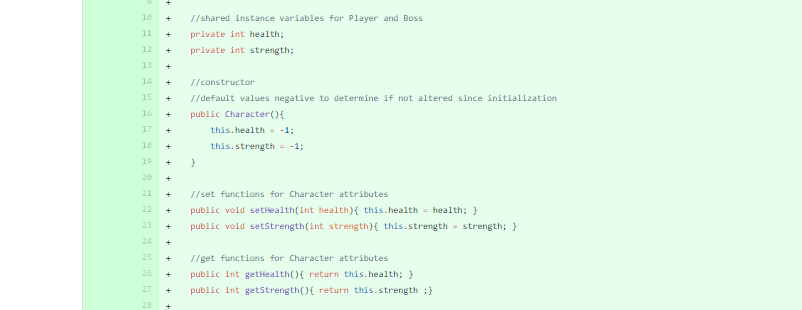




As part of the **Extract Superclass** refactoring, two additional refactorings were also utilized: **Pull Up Method** and **Pull Up Field,** wherein the identical fields and methods of the Player and Boss subclasses were pulled up to the Character superclass:



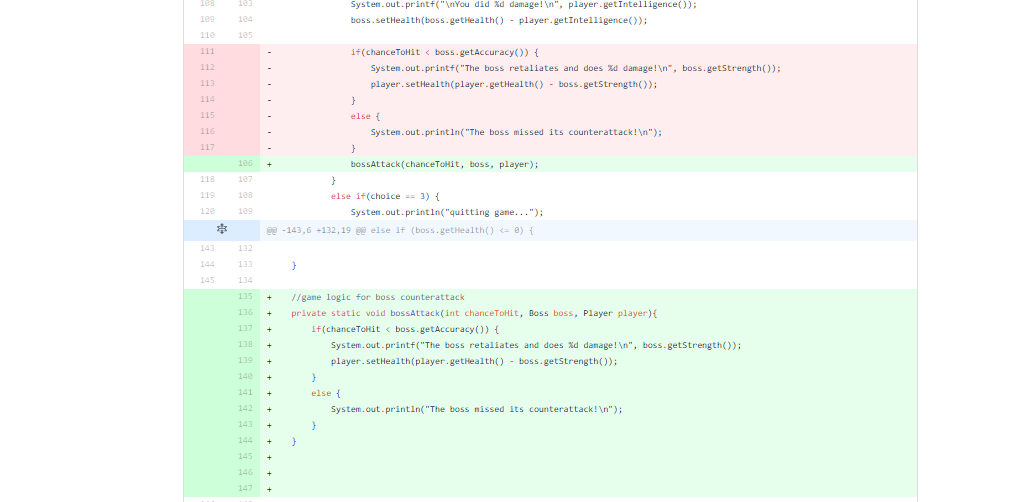




These refactorings thus resulted in a less duplicated code between the Player and Boss classes, both in terms of identical fields and identical methods.

## Refactoring #7, #8 – Extract Method, MOVE METHOD

Referring back to the code smell of the Boss class being too close to a **Data Class**, the Driver class was then reevaluated for possible behaviour that could be delegated to the Boss class. Such delegable behaviour was found in the conditional logic for the boss attacks in the Driver main, which was also still guilty of being a **Long Method**. As such, the **Extract Method** refactoring was first used to separate the bossAttack method from the Driver main method (refer to GitHub version control logs):

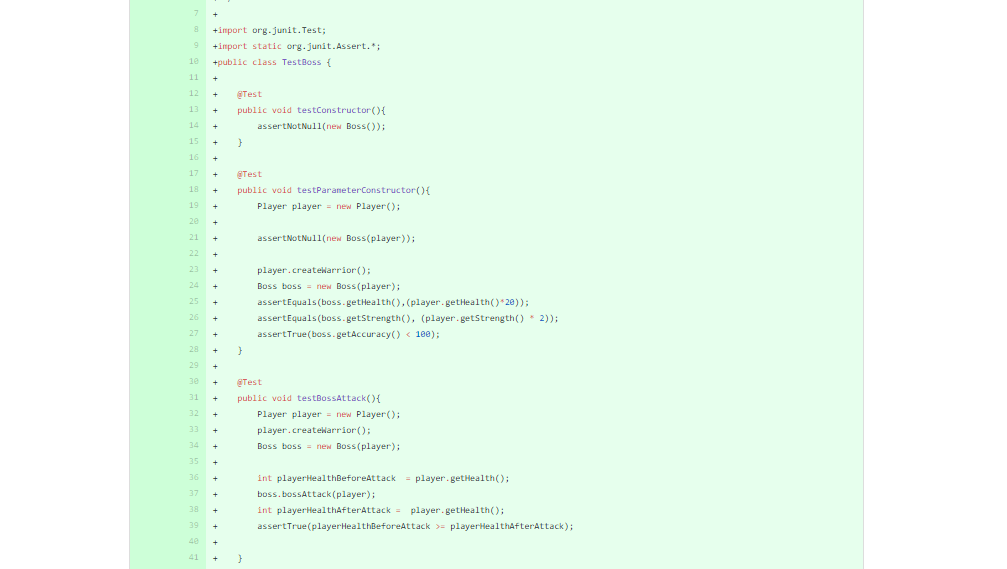


Following the **Extract Method**, a subsequent **Move Method** from the Driver class to the Boss class was then implemented to improve the behaviours of the Boss class as a separate class:



The combination of these refactorings thus resulted in a smaller main method for the Driver, a smaller overall Driver class, and prevents the Boss class from being more than just Data class.

The Boss class was then given unit tests to test its constructors and the bossAttack method itself, to ensure that the game logic was intact after the extraction and moving of methods.

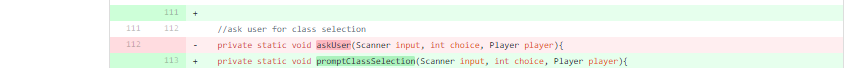


## REFactoring #9, #10 – Extract Method, RENAME METHOD

For the last series of refactorings, the Driver main method was given one last review to reduce its **Long Method** code smell. From this review, it became apparent that the prompt logic for the conditional class creation could be clumped together, and thus separated from the main method using **Extract Method** refactoring (refer GitHub version control logs):



Following this, an additional **Rename Method** refactoring was applied to better convey the purpose of the newly extracted method:



From these reafactorings, the Driver main method was again reduced in size and made more readable, thus facilitating future debugging or refactoring if necessary.

## CONCLUSION – List of Refactorings

While the original version of the code was certainly functional, it was mired with Driver main that was **long method**, which in turn caused the Driver to be a **large class** that had major **feature envy**, and the Player and Boss classes had **duplicate code** that made them inelegant as their own separate classes. From the application of the refactorings mentioned previously, it is clear that this object oriented program is now better suited for iterative development, and less prone to decay should it be revisited at a later date.

Following is a complete listing of the first and last versions of the refactored code, as well as all intermediate refactorings. For more details, refer to the GitHub version control logs:

