Refactoring Changes made to “Dungeon” Project

Kelsey McMahon, Joshua Merrell, Alan Moss, Michael Nall

1. Setting up project with classes in the source folder and reformatting the code for better readability.

2. Removing deprecated code to make the project compatible: Line 287 and 386 on Keyboard class, changed to Float.parseDouble() and Double.parseDouble.

```
285     try
286     {
287        value = (new Float(token)).floatValue();
288     }
289     catch (Exception exception)
290     {
291        error ("Error reading float data, NaN value returned.");
292        value = Float.NaN;
293     }
294     return value;
295  }
296
297  //-----------------------------------------------------------------
298  //  Returns a double read from standard input.
299  //-----------------------------------------------------------------
300  public static double readDouble()
301  {
302     String token = getNextToken();
303     double value;
304     try
305     {
306        value = (new Double(token)).doubleValue();
307     }
308     catch (Exception exception)
309     {
```

3. Removing any unused variables or references: Line 72 in the Dungeon class the variable “Hero” was not being used and was removed.

```
69     public static Hero chooseHero()
70     {
71         int choice;
72         Hero theHero;
73
74         System.out.println("Choose a hero:\n" +
75                            "1. Warrior\n" +
76                            "2. Sorceress\n" +
77                            "3. Thief");
78         choice = Keyboard.readInt();
```

4. Implementing the Strategy Pattern to remove inheritance in favor of composition by adding the DungeonCharacter interface and removing the DungeonCharacter class. This helps make the code easier to understand, removes duplicated methods, and makes it easier for future edits or additions to the classes.

```java
public interface DungeonCharacter
{
    public String getName();

    public int getHitPoints();

    public int getAttackSpeed();

    public boolean isAlive();

    public void addHitPoints(int hitPoints);

    public void subtractHitPoints(int hitPoints);

    public void attack(DungeonCharacter opponent);

    public void battleChoices(DungeonCharacter opponent);

}
```

5. Adding a Creational Pattern to the object creation part of the program by adding the Monster Factory to create the different monster characters. Adding a Factory Creational Pattern helps to encapsulate the object creation portion of the project.

```java
public class MonsterFactory
{

    // Ogre constants
    public static final int OGRE_ID = 1;
    public static final String OGRE_CLASS_NAME = "Oscar the Ogre";
    public static final int OGRE_HP = 200;
    public static final int OGRE_ATTACK_SPEED = 2;
    public static final String OGRE_ATTACK_STRING = "%s slowly swings a club toward's %s.\n";
    public static final double OGRE_HIT_CHANCE = 0.6;
```

6. Adding a Creational Pattern to the object creation part of the program by adding the Hero Factory to create the different hero characters. Adding a Factory Creational Pattern helps to encapsulate the object creation portion of the project.

```java
public class HeroFactory
{
    // Warrior constants
    public static final int WARRIOR_ID = 1;
    public static final String WARRIOR_CLASS_NAME = "Warrior";
    public static final int WARRIOR_HP = 125;
    public static final int WARRIOR_ATTACK_SPEED = 4;
    public static final String WARRIOR_ATTACK_STRING = "%s swings a mighty sword at %s:\n";
    public static final double WARRIOR_HIT_CHANCE = 0.8;
    public static final int WARRIOR_MIN_DAMAGE = 35;
    public static final int WARRIOR_MAX_DAMAGE = 60;
```

7. Reformatting main (Dungeon.java) to include the factory classes for making the Hero character and Monster character. This helps implement the factory pattern and makes the code easier to understand.

```
31      public static DungeonCharacter chooseHero()
32      {
33          int choice;
34          // Hero theHero;
35
36          System.out.println(
37              "Choose a hero:\n" +
38              "1. Warrior\n" +
39              "2. Sorceress\n" +
40              "3. Thief\n");
41          choice = Keyboard.readInt();
42          return HeroFactory.makeHero(choice);
43      }//end chooseHero method
44
```

8. Removing the Monster and Hero subclasses and creating an interface called SpecialAttack to be implemented by the different attack classes (SupriseAttack, HealSpecial, CrushingBlow). Removing the sublasses makes future changes and additions easier because you only have to make changes to one class (Hero, and Monster).

```
1   public interface SpecialAttack
2   {
3       /**
4        * Return the name of the attack to be used by Hero#battleChoices
5        */
6       public String getAttackName();
7
8       /**
9        * Called by Hero#specialAttack
0        * @param attacker the Hero that is doing the attack
1        * @param target the monster to be attacked
2        */
3       public void doAttack(Hero attacker, DungeonCharacter target);
4   }
5
```

```
1   public class SurpriseAttack implements SpecialAttack
2   {
3       protected static final double DEFAULT_MIN = 0.4;
4       protected static final double DEFAULT_MAX = 0.9;
5
6       protected String attackName;
7       protected double min;
8       protected double max;
9
0       public SurpriseAttack(String attackName, double min, double max)
1       {
2           this.attackName = attackName;
3           this.min = min;
4           this.max = max;
5       }
6
7       public SurpriseAttack(String attackName)
8       {
9           this.attackName = attackName;
0           this.min = DEFAULT_MIN;
1           this.max = DEFAULT_MAX;
2       }
3
```