Nate Hiblar, Eddie Joseph, Colton Cronquist
March 6th, 2020
Team Based Assignment: Refactoring!

Refactors:

1) Monster creation moved to factory pattern.
    a. This allows for easy addition and removal of new or old monsters. It de-couples the concrete monster from the abstract dungeon character. This is designing to an "interface" – dungeon character is an abstract class.

```
5  public abstract class DungeonCharacter implements AttackBehavior {
```

2) Hero creation moved to factory pattern.
    a. This allows for easy addition and removal of new or old hero's. It de-couples the concrete hero from the abstract dungeon character. This is designing to an "interface" – dungeon character is an abstract class.

```
5  public class HeroFactory {
6
7      public Hero newHero(String charRace, String charClass, String name){
8
9          switch(charRace) {
10             case "Aarakocra":
11                 return new Aarakocra(name, 14, 3, 16, charClass, new ShortSword());
12             case "Bugbear":
13                 return new BugBear  (name, 19, 1, 17, charClass, new GreatSword());
14             case "Elf":
```

3) Replaced kb class with scanner.
    a. A class for accepting input is deprecated. Using java's built in scanner class isolates behavior to allow for easier testing.

Completely unnecessary …

```
38    //-----------------------------------------------------------------
39    public static boolean getPrintErrors()
40    {
41       return printErrors;
42    }
43
44    //-----------------------------------------------------------------
45    //  Sets a boolean indicating whether input errors are to be
46    //  printed to standard output.
47    //-----------------------------------------------------------------
48    public static void setPrintErrors (boolean flag)
49    {
50       printErrors = flag;
51    }
52
```

4) Created user input validation class.

a. Removes code from the driver class to a single input class that does one and only one thing. All input validation is handled in one place reducing the size of the driver class.

This code snip is from the hero class. Hero class should not accept user input.

```
55 ------------------------------------------------------------*/
56   public void readName()
57   {
58       System.out.print("Enter character name: ");
59       name = Keyboard.readString();
60   }//end readName method
61
62 /*------------------------------------------------------------
```

5) Fields changed to private and added getters and setters.
   a. This is black box design to hide the fields of classes from the outside world.

```
public class Hero extends DungeonCharacter implements AttackBehavior {

    private String charRace;
    private String charClass;
    private SpecialAttack specialAttack;
```

6) Attack behaviors use strategy patterns
   a. This allows for dependency injection. The attack behavior in an interface referenced by the hero object. Multiple hero's can have the same attack or have the behavior of their attack changed at run time if the hero abilities change during game play.

```
5  public interface AttackBehavior {
6      int attack(DungeonCharacter enemy);
7      String getName();
8  }
```

```
6  public class ShortSword implements AttackBehavior {
7
8      public int attack(DungeonCharacter enemy) {
9          boolean doesHit = Dice.d20() - enemy.getAC() > 0;
10         if (doesHit) {
11             int dam = Dice.d6();
12             enemy.subtractHitPoints(dam);
```

```
public class Bandit extends Monster{

    public Bandit() {
        super("Jerry the Bandit", 10, 1, 13, new ShortSword());
    }
```

7) Special attacks use strategy pattern.

a. This allows for dependency injection. The attack behavior in an interface referenced by the hero object. Multiple hero's can have the same attack or have the behavior of their attack changed at run time if the hero's abilities change during game play.

```
5  public interface SpecialAttack {
6      int specialAttack(DungeonCharacter enemy);
7      String getName();
8  }
```

```
6  public class Fireball implements SpecialAttack {
7      @Override
8      public int specialAttack(DungeonCharacter enemy) {
9          int dam = Dice.d6() * 3;
10         enemy.subtractHitPoints(dam);
```

8) Changed names of methods and fields
   a. This creates readable code that is easier to follow. Names of methods describe their behaviors, and field names represent the attributes they hold.

```
hero = heroFactory.newHero(GetUserInput.getRaceFromUser(), GetUserInput.getClassFromUser(), GetUserInput.getNameFromUser());
```

```
again = GetUserInput.getPlayAgainUser();
```