Gill, Harjasleen
Glushchenko, Ephrem
Hanson, Denis

Team Based Assignment: Refactoring Write-up

1. **Comments** – The comments were removed from the original classes as they were a code smell. The comments were unnecessary and were explaining what was written and provided information that was not needed to understand the code. The only bit of comments in the classes that are left now is // same. They are at the end of methods to show which methods were not changed from the original source.

Offending Code

```
/*-------------------------------------------------------
addHitPoints is used to increment the hitpoints a dungeon character has

Receives: number of hit points to add
Returns: nothing

This method calls: nothing
This method is called by: heal method of monsters and Sorceress
-------------------------------------------------------*/
    public void addHitPoints(int hitPoints)
    {
        if (hitPoints <=0)
            System.out.println("Hitpoint amount must be positive.");
        else
        {
            this.hitPoints += hitPoints;
            //System.out.println("Remaining Hit Points: " + hitPoints);

        }
    }//end addHitPoints method

/*-------------------------------------------------------
subtractHitPoints is used to decrement the hitpoints a dungeon character has.
It also reports the damage and remaining hit points (these things could be
done in separate methods to make code more modular ;-)

Receives: number of hit points to subtract
Returns: nothing

This method calls: nothing
This method is called by: overridden versions in Hero and Monster
-------------------------------------------------------*/
    public void subtractHitPoints(int hitPoints)
```

New Code

```
public boolean guard() {
    return Math.random() <= this.block;
    // same
}
```

2. **Naming** - Some method and parameter names were renamed (sometimes shortened) to improve understanding of the code. And are more consistent with the theme and do not require excessive comments for explanation.

Offending code

```
public void subtractHitPoints(int hitPoints)
    {
        if (defend())
        {
            System.out.println(name + " BLOCKED the attack!");
        }
        else
        {
            super.subtractHitPoints(hitPoints);
        }


    }//end method
```

New Code

```
public void subHitScore(int hitScore) {
    if (guard()) {
        System.out.println(getName() + " BLOCKED the attack!");
    } else {
        super.subHitScore(hitScore);
    }
    // same
}
```

3. **Logic and implementing updated Java API** – Better logic was used to design and update the project and outdated tools from java API were removed. The Keyboard.java class was an example of outdated tools from java API, so it was removed and replaced with built-in functions.

Offending code

```
//-----------------------------------------------------------------
//  Returns a space-delimited substring (a word) read from
//  standard input.
//-----------------------------------------------------------------
public static String readWord()
{
   String token;
   try
   {
      token = getNextToken();
   }
   catch (Exception exception)
   {
      error ("Error reading String data, null value returned.");
      token = null;
   }
   return token;
}
```

New Code

```
import java.util.Scanner;

public String name() {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter character name: ");
    String name = scanner.nextLine();
    return name;
}
```

4. **Dead Code** – deleted unused code from the program. The method *public void heal()* in the original Monster.java class was not implemented and thus was removed to refactor the class.

Offending code

```
public void heal()
{
    boolean canHeal;
    int healPoints;

    canHeal = (Math.random() <= chanceToHeal) && (hitPoints > 0);

    if (canHeal)
    {
        healPoints = (int)(Math.random() * (maxHeal - minHeal + 1)) + minHeal;
        addHitPoints(healPoints);
        System.out.println(name + " healed itself for " + healPoints + " points.\n"
                           + "Total hit points remaining are: " + hitPoints);
        System.out.println();
    }//end can heal
```

5. **Program to Interface, not Implementation** – The Actions interface was created to define the role a hero character has. The benefit of an interface is that it achieves loose coupling.

New Code

```
public class WarriorAction implements Actions {

    public void setActions(DungeonCharacter enemy, DungeonCharacter hero) {
        if (Math.random() <= .4) {
            int damage = (int) (Math.random() * 76) + 100;
            System.out.println(hero.getName() + " lands a CRUSHING BLOW for " + damage + " damage!");
            enemy.subHitScore(damage);
        }
        else {
            System.out.println(hero.getName() + " failed to land a crushing blow");
            System.out.println();
        }
        //same
    }
}
```

6. **Program to Interface, not Implementation** – The *strikeBack* interface was created to define the role a monster character has. The benefit of an interface is that it achieves loose coupling.

   New Code

```
public class GremlinStrike implements strikeBack {

    public void strike(DungeonCharacter enemy, DungeonCharacter monster) {
        System.out.println(monster.getName() + " jabs his kris at " + enemy.getName() + ":");
        // same
    }
}
```

7. **Factory Method** – The *HeroFactory* was created to create different types of warriors and to assign them a role. This helps us maintain our code as it makes it simpler to add new types of warriors without worrying about class explosion.

   New Code

```
public interface HeroFactory {
    public static Hero addCharacter(int type) {
        Hero hero = null;
        if (type == 1) {
            hero = new Hero("Warrior", 125, 4, .8, 35, 60, .2);
            hero.setActions(setActions(type));
            return hero;
        } else if (type == 2) {
            hero = new Hero("Sorceress", 75, 5, .7, 25, 50, .3);
            hero.setActions(setActions(type));
            return hero;
        } else if (type == 3) {
            hero = new Hero("Thief", 75, 6, .8, 20, 40, .5);
            hero.setActions(setActions(type));
            return hero;
        }
        return hero;
    }

    public static Actions setActions(int type) {
        if (type == 1) {
            return new WarriorAction();
        }
        if (type == 2) {
            return new SorceressAction();
        }
        if (type == 3) {
            return new ThiefAction();
        } else {
            return new WarriorAction();
        }
    }
}
```

8. **Factory Method** – The *MonsterFactory* was created to create different types of monsters and to assign them a role. This helps us maintain our code as it makes it simpler to add new types of monsters without worrying about class explosion.

New Code

```
public interface MonsterFactory {
    public static Monster addCharacter(int type) {
        Monster monster = null;
        if (type == 1) {
            monster = new Monster("Oscar the Ogre", 200, 2, .6, 30, 50);
            monster.setStrike(setStrike(type));
            return monster;
        }
        if (type == 2) {
            monster = new Monster("Gnarltooth the Gremlin", 70, 5, .8, 15, 30);
            monster.setStrike(setStrike(type));
            return monster;
        }
        if (type == 3) {
            monster = new Monster("Sargath the Skeleton", 100, 3, .8, 30, 50);
            monster.setStrike(setStrike(type));
            return monster;
        }
        return monster;
    }

    public static strikeBack setStrike(int type) {
        if (type == 1) {
            return new OgreStrike();
        }
        if (type == 2) {
            return new GremlinStrike();
        }
        if (type == 3) {
            return new SkeletonStrike();
        } else {
            return new OgreStrike();
        }
    }
```