Muangkong Yang
CSCD349
Heroes vs Monsters Refactor

1) All data members in each classes are currently set to protected. I will be changing their visibility to private and set up get and set methods.

Before

```
public abstract class DungeonCharacter implements Comparable
{

    protected String name;
    protected int hitPoints;
    protected int attackSpeed;
    protected double chanceToHit;
    protected int damageMin, damageMax;
```

```
public abstract class Monster extends DungeonCharacter
{
    protected double chanceToHeal;
    protected int minHeal, maxHeal;
```

After

```
private String name;
private int hitPoints;
private int numTurns;
private boolean isAlive;
```

2) I will be changing the DungeonCharacter class hierarchy to follow the Strategy Pattern. All monsters and heroes have some same functionality like statuses, and different ones. I will push all commonalities to the highest hierarchy possible with new abstract methods. Heroes and Monster subclasses will define their own values for each. This makes it add in new subclasses without having to change the overall code. Each hero also have their own unique special move, and inside the method is the intended functionality. This also helps lessen the amount of primitives in the constructors and more self explanatory.

Before

```java
public abstract class Hero extends DungeonCharacter
{
    protected double chanceToBlock;
    protected int numTurns;

//-----------------------------------------------------------------
//calls base constructor and gets name of hero from user
  public Hero(String name, int hitPoints, int attackSpeed,
                    double chanceToHit, int damageMin, int damageMax,
                    double chanceToBlock)
  {
    super(name, hitPoints, attackSpeed, chanceToHit, damageMin, damageMax);
    this.chanceToBlock = chanceToBlock;
    readName();
  }
```

```java
public class Thief extends Hero
{

    public Thief()
    {
        super("Thief", 75, 6, .8, 20, 40, .5);


    }//end constructor

    public void surpriseAttack(DungeonCharacter opponent)
    {
        double surprise = Math.random();
        if (surprise <= .4)
        {
            System.out.println("Surprise attack was successful!\n" +
                               name + " gets an additional turn.");
            numTurns++;
            attack(opponent);
        }//end surprise
        else if (surprise >= .9)
        {
            System.out.println("Uh oh! " + opponent.getName() + " saw you and" +
                               " blocked your attack!");
        }
        else
            attack(opponent);


    }//end surpriseAttack method
```

After

```
public abstract class DungeonCharacter
{
    // Refactor 2
    // All Heroes and Monsters have these abstract attributes
    public abstract int originalHitPoints();
    public abstract int attackSpeed();
    public abstract double chanceToHit();
    public abstract int damageMin();
    public abstract int damageMax();
    public abstract void attackMotion();
```

```
public abstract class Monster extends DungeonCharacter
{
    // Some unique functions for Monsters only that differ
    // List them here:
    public abstract double chanceToHeal();
    public abstract int minHealPoints();
    public abstract int maxHealPoints();
```

```java
public abstract class Hero extends DungeonCharacter
{
    public abstract void commandMenu();
    public abstract double chanceToBlock();
    public abstract void attackMotion();
    public abstract void special(DungeonCharacter character);

public class Thief extends Hero
{
    public Thief(final String name)
    {
        super(name, 75);
    }

    @Override
    public int originalHitPoints()
    {
        return 75;
    }

    @Override
    public int attackSpeed()
    {
        return 6;
    }

    @Override
    public double chanceToHit()
    {
        return 0.8;
    }

    @Override
    public int damageMin()
    {
        return 20;
    }
```

```java
public class Skeleton extends Monster
{
    public Skeleton()
    {
        super("Sargath the Skeleton", 100);

    }//end constructor

    @Override
    public int originalHitPoints()
    {
        return 100;
    }

    @Override
    public int attackSpeed()
    {
        return 3;
    }

    @Override
    public double chanceToHit()
    {
        return 0.8;
    }

    @Override
    public int damageMin()
    {
        return 30;
    }
```

```java
@Override
public void special(DungeonCharacter character)
{
    double surprise = Math.random();
    if (surprise <= .4)
    {
        System.out.println("Surprise attack was successful!\n" +
                           this.getName() + " gets an additional turn.");
        this.setNumTurns(getNumTurns() + 1);
        attack(character);
    }//end surprise
    else if (surprise >= .9)
    {
        System.out.println("Uh oh! " + character.getName() + " saw you and" +
                           " blocked your attack!");
        System.out.println();
    }
    else
        attack(character);
}
```

3) I will be using the factory method to create the intended heroes and monsters instead of these switch statements in the Dungeon class.

Before

```
public static Hero chooseHero()
{
    int choice;
    Hero theHero;

    System.out.println("Choose a hero:\n" +
                       "1. Warrior\n" +
                       "2. Sorceress\n" +
                       "3. Thief");
    choice = Keyboard.readInt();

    switch(choice)
    {
        case 1: return new Warrior();

        case 2: return new Sorceress();

        case 3: return new Thief();

        default: System.out.println("invalid choice, returning Thief");
                 return new Thief();
    }//end switch
}//end chooseHero method
```

```
public static Monster generateMonster()
{
    int choice;

    choice = (int)(Math.random() * 3) + 1;

    switch(choice)
    {
        case 1: return new Ogre();

        case 2: return new Gremlin();

        case 3: return new Skeleton();

        default: System.out.println("invalid choice, returning Skeleton");
                 return new Skeleton();
    }//end switch
}//end generateMonster method
```

After

```
public class HeroFactory
{
    public Hero createHeroClass(final String characterName, int type)
    {

        if(type == 1)
            return new Warrior(characterName);
        else if(type == 2)
            return new Sorceress(characterName);
        else
            return new Thief(characterName);
    }
}
```

```
public class MonsterFactory
{
    public Monster generateMonster(int type)
    {
        if(type == 1)
            return new Ogre();
        else if(type == 2)
            return new Gremlin();
        else
            return new Skeleton();
    }
}
```

4) Each hero has their own battleChoices method, so I refactored the method up to the Hero class and to just list the differences in the hero subclasses. I named the subclass method commandMenu(). Instead of having the commandMenu() having to call Utils to readInt() all of the time, I added a heroMenu() in the Hero class to have the code once, and call the commandMenu() within. This also gets rid of the switch statement in the battleChoices method.

Before

```
public void battleChoices(DungeonCharacter opponent)
{
    super.battleChoices(opponent);
    int choice;


    do
    {
        System.out.println("1. Attack Opponent");
        System.out.println("2. Surprise Attack");
        System.out.print("Choose an option: ");
        choice = Keyboard.readInt();

        switch (choice)
        {
            case 1: attack(opponent);
                break;
            case 2: surpriseAttack(opponent);
                break;
            default:
                System.out.println("invalid choice!");
        }//end switch

        numTurns--;
        if (numTurns > 0)
            System.out.println("Number of turns remaining is: " + numTurns);

    } while(numTurns > 0);

}
```

After

```java
public void battleChoices(DungeonCharacter character)
{
   int choice;
   this.calculateTurns(character);

   do
   {
       choice = heroMenu();
       System.out.println();

       switch (choice)
       {
           case 1: attack(character);
               break;
           case 2: special(character);
               break;
       }//end switch

       if(character.isAlive())
       {
           this.setNumTurns(this.getNumTurns() - 1);
           if (this.getNumTurns() > 0)
               System.out.println("Number of turns remaining is: " + this.getNumTurns());
       }
       else
           this.setNumTurns(0);

   } while(this.getNumTurns() > 0);

}//end battleChoices method
```

```java
public int heroMenu()
{
    int choice;
    this.commandMenu();
    System.out.print("Choose an option: ");
    do
    {
        choice = Utils.readInt();
    } while(choice < 1 || choice > 2);

    return choice;
}
```

```java
@Override
public void commandMenu()
{
    System.out.println("1. Attack Opponent");
    System.out.println("2. Surprise Attack");
}
```

5) I will be refactoring the Keyboard class to implement the Scanner class in later versions of java. I named the new class Utils. Every method that needs a user input has also been changed accordingly. I also moved the readString() for the name of character to my chooseHero method when creating the hero from the factory. Likewise for monster.

Before

```
public static String readString()
{
   String str;

   try
   {
      str = getNextToken(false);
      while (! endOfLine())
      {
         str = str + getNextToken(false);
      }
   }
   catch (Exception exception)
   {
      error ("Error reading String data, null value returned.");
      str = null;
   }
   return str;
}
```

After

```java
public class Utils
{
    static Scanner kb = new Scanner(System.in);

    public static int readInt()
    {
        int num = 0;
        do
        {
            try
            {
                num = Integer.parseInt(kb.nextLine());
            }
            catch(NumberFormatException e)
            {
                System.out.print("Not a number.\nChoose a option: ");
            }
        } while(num < 1 || num > 10);
        return num;
    }
```

```java
public static Hero chooseHero(HeroFactory heroFactory)
{
    int choice;

    System.out.println("Choose a hero:\n" +
                       "1. Warrior\n" +
                       "2. Sorceress\n" +
                       "3. Thief");
    do
    {
        choice = Utils.readInt();
    } while(choice < 1 || choice > 3);

    System.out.print("Enter character name: ");
    String characterName = Utils.readString();

    return heroFactory.createHeroClass(characterName, choice);
}
```

6) I've also added a setHitPoints method and set the hitpoints to either 0 if its less than 0 or to the originalHitPoints() if the amount of heal points recovered goes over the original hit points of the character or monster.

Before

```
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
```

After

```
public void addHealPoints(int healPoints)
{
    if (this.getHitPoints() <=0)
        System.out.println("Hitpoint amount must be positive.");
    else
    {
        int tempHP = 0;
        tempHP = this.getHitPoints() + healPoints;
        this.setHitPoints(tempHP);
    }
}

public void setHitPoints(int hitPoints)
{
    if(hitPoints < 0)
        this.hitPoints = 0;
    else if(hitPoints > this.originalHitPoints())
        this.hitPoints = this.originalHitPoints();
    else
        this.hitPoints = hitPoints;
}
```

7) I've also changed the look to subtractHitPoints() method to make it more understandable by passing in the character that is getting hit. I also handle a monster having a chance to heal in the if(character instanceof Monster) so that the monster heals if it is the one getting hit.

Before

```
public void subtractHitPoints(int hitPoints)
{
    if (hitPoints <0)
        System.out.println("Hitpoint amount must be positive.");
    else if (hitPoints >0)
    {
        this.hitPoints -= hitPoints;
        if (this.hitPoints < 0)
            this.hitPoints = 0;
        System.out.println(getName() + " hit " +
                           " for <" + hitPoints + "> points damage.");
        System.out.println(getName() + " now has " +
                           getHitPoints() + " hit points remaining.");
        System.out.println();
    }//end else if

    if (this.hitPoints == 0)
        System.out.println(name + " has been killed :-(");


}//end method
```

After

```
public void subtractHitPoints(DungeonCharacter character, int damage)
{
    if (character.hitPoints < 0)
        System.out.println("Hitpoint amount must be positive.");
    else if (this.hitPoints > 0)
    {
        int tempHitPoints = character.hitPoints - damage;
        character.setHitPoints(tempHitPoints);
        System.out.println(getName() + " hit " + character.getName() +
                           " for <" + damage + "> points damage.");
        System.out.println(character.getName() + " now has " +
                           character.getHitPoints() + " hit points remaining.");
        System.out.println();
    }//end else if

    if(character instanceof Monster)
        ((Monster) character).heal();

    if (character.hitPoints == 0)
        System.out.println(character.getName() + " has been killed :-(");
}
```

8) I moved all attack() methods to DungeonCharacter so that it covers everything in common. I've ended up with just having to add an attackMotion() method to print out the unique motions of each Hero/Monster. I also pass in the character getting attacked to pass them into subtractHtPoints();

Before

```
public void attack(DungeonCharacter opponent)
{
    boolean canAttack;
    int damage;

    canAttack = Math.random() <= chanceToHit;

    if (canAttack)
    {
        damage = (int)(Math.random() * (damageMax - damageMin + 1))
                    + damageMin ;
        opponent.subtractHitPoints(damage);



        System.out.println();
    }//end if can attack
    else
    {

        System.out.println(getName() + "'s attack on " + opponent.getName() +
                           " failed!");
        System.out.println();
    }//end else

}//end attack method
```

After

```java
public void attack(DungeonCharacter character)
{
    boolean canAttack;
    int damage = 0;

    canAttack = Math.random() <= chanceToHit();

    if (canAttack)
    {
        damage = (int)(Math.random() * (damageMax() - damageMin() + 1))
                    + damageMin();
        System.out.print(this.getName());
        this.attackMotion();
        System.out.println(character.getName() + ":");
        this.subtractHitPoints(character, damage);

    }//end if can attack
    else
    {
        System.out.println(this.getName() + "'s attack on " + character.getName() +
                           " failed!");
        System.out.println();
    }//end else

}
```

```java
@Override
public void attackMotion()
{
    System.out.print(" casts a spell of fireball at ");
}
```

9) I also switched on how to keep track of the number of turns a hero has with a new calculateNumTurn() method. This will change the private field numTurn locally so I can refer to it anytime if I need it. Compare battleChoices to see the difference. Note that I have moved all battleChoices to Hero.

Before

```
public void battleChoices(DungeonCharacter opponent)
{
    numTurns = attackSpeed/opponent.getAttackSpeed();

    if (numTurns == 0)
        numTurns++;

    System.out.println("Number of turns this round is: " + numTurns);

}//end battleChoices
```

After

```
public int calculateTurns(DungeonCharacter character)
{
    int numTurns = this.attackSpeed()/character.attackSpeed();
    if (numTurns == 0)
        numTurns++;
    this.setNumTurns(numTurns);

    System.out.println("Number of turns this round is: " + this.getNumTurns());

    return this.getNumTurns();
}
```

```java
public void battleChoices(DungeonCharacter character)
{
    int choice;
    this.calculateTurns(character);

    do
    {
        choice = heroMenu();
        System.out.println();

        switch (choice)
        {
            case 1: attack(character);
                break;
            case 2: special(character);
                break;
        }//end switch

        if(character.isAlive())
        {
            this.setNumTurns(this.getNumTurns() - 1);
            if (this.getNumTurns() > 0)
                System.out.println("Number of turns remaining is: " + this.getNumTurns());
        }
        else
            this.setNumTurns(0);

    } while(this.getNumTurns() > 0);

}//end battleChoices method
```

10) I added in a check to see if the character we are attacking is alive before attacking again. If a monster dies on turn 1 when you still have a total of 2 turns, it will make you attack again when the enemy is already dead.

Before

```
do
{
    System.out.println("1. Attack Opponent");
    System.out.println("2. Surprise Attack");
    System.out.print("Choose an option: ");
    choice = Keyboard.readInt();

    switch (choice)
    {
        case 1: attack(opponent);
            break;
        case 2: surpriseAttack(opponent);
            break;
        default:
            System.out.println("invalid choice!");
    }//end switch

    numTurns--;
    if (numTurns > 0)
        System.out.println("Number of turns remaining is: " + numTurns);

} while(numTurns > 0);
```

After

```
do
{
    choice = heroMenu();
    System.out.println();

    switch (choice)
    {
        case 1: attack(character);
            break;
        case 2: special(character);
            break;
    }//end switch

    if(character.isAlive())
    {
        this.setNumTurns(this.getNumTurns() - 1);
        if (this.getNumTurns() > 0)
            System.out.println("Number of turns remaining is: " + this.getNumTurns());
    }
    else
        this.setNumTurns(0);

} while(this.getNumTurns() > 0);
```