Hall of Bones

By Kevin Joseph (214591838), Anton Thomas (211710670), Andrew Laniak-Fuoco (213322482), Yi Ngan (214387989), and Alan Tang (214717581)

https://github.com/kevinj22/Hall-Of-Bones-master

http://www.eecs.yorku.ca/~kevinj22/RPG_API/

Introduction

This project is a role-playing game inspired by titles such as Darkest Dungeon, Final Fantasy and Pokemon. Players explore a brief dungeon battling enemies.

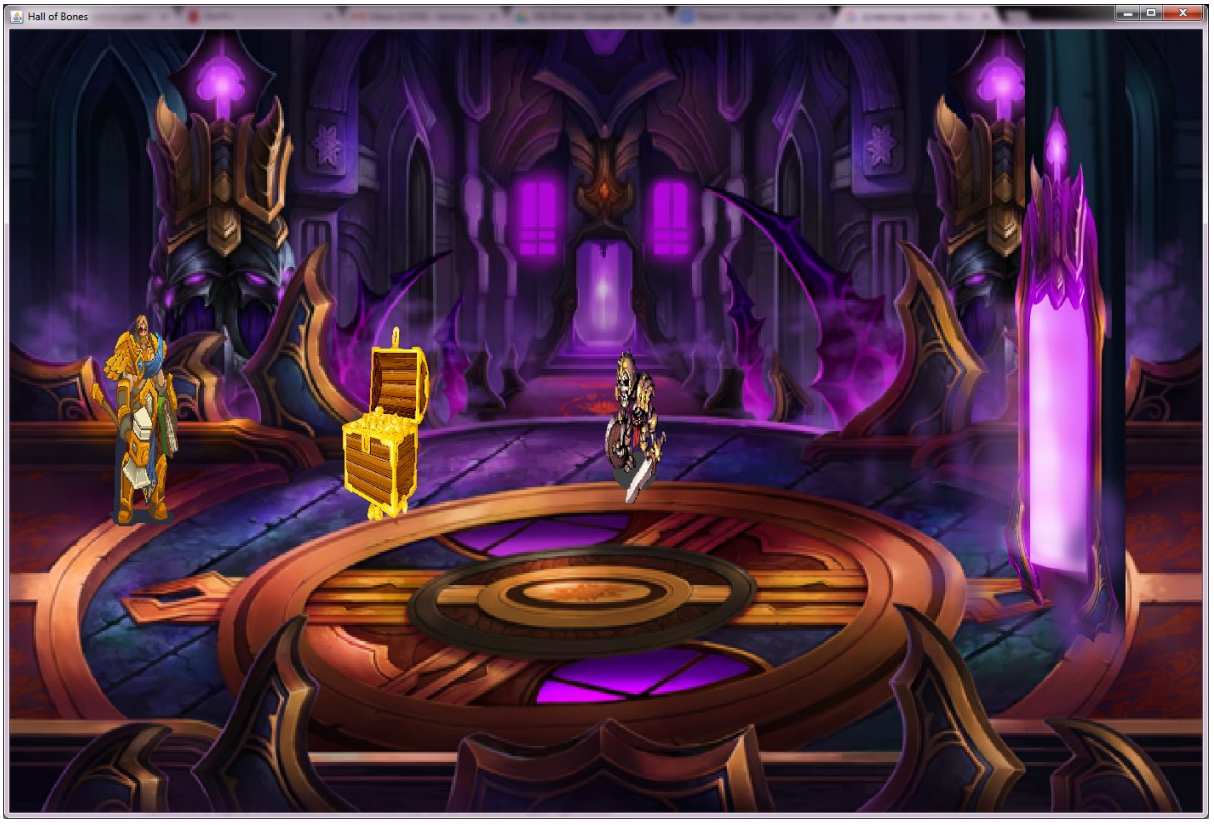
Overview

Players start in the Dungeon view. They can then be launched into the Battle view on collision with an enemy.

Dungeon

The Dungeon view contains these features:

- An enemy to collide with
- A treasure chest to open
- A door to exit the dungeon


Hall of Bones

Battle

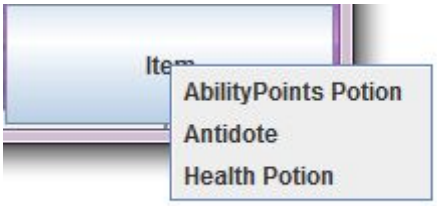
The Battle view contains these features.

- Health bars
- Ability Points bars
- Animations on attacks, status updates, and character indicators

Indicator:

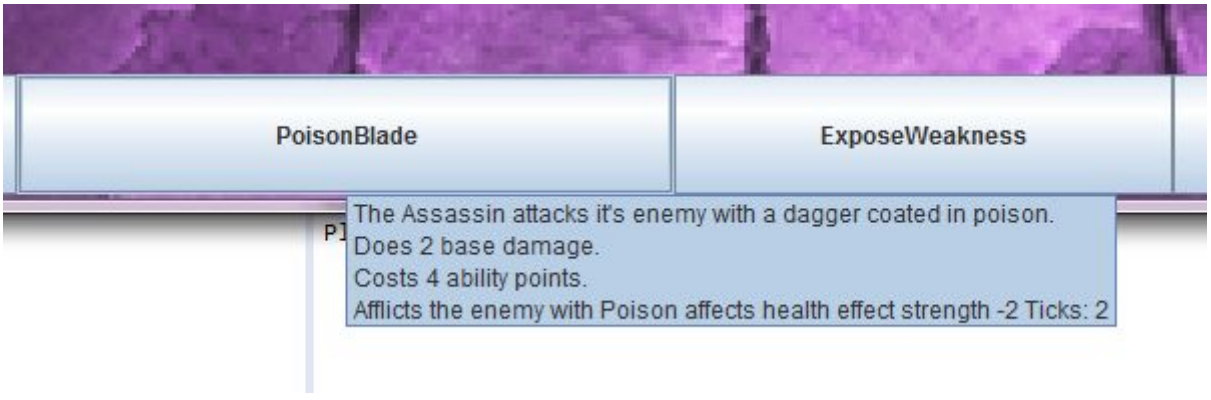
- Stacking status effects

- Ability buttons that update dependent on the character who is currently controlled
- A pop up menu for the inventory selection



- Tooltips on hover over characters and ability buttons that describe the character's stats are / what the ability does


Mage has:
Attack power: 1
Defense rating: 1


PoisonBlade
ExposeWeakness
The Assassin attacks it's enemy with a dagger coated in poison.
Does 2 base damage.
Costs 4 ability points.
Afflicts the enemy with Poison affects health effect strength -2 Ticks: 2

Technical Details and Implementation

Implementation Details

Main JFrame

As this project implements several separate views rather than creating a new JFrame every time a switch is required, we implemented a main JFrame class that handles all switching.

This class contains a JFrame which has a JPanel that implements a CardLayout. The CardLayout allows us to create many JPanels at once and store them within the CardLayout.

```
   /**
     * Create the CardLayout JPanel used in this JFrame.
     */
    private void createThisCardLayout()
    {
        mainContentPane = new JPanel();
        mainContentPane.setLayout(new CardLayout());
        mainCard = (CardLayout) (mainContentPane.getLayout());
    }

     /**
     * Creates all of the panels involved in this game. Does not start any threads in each
model until
     * the GameView startView() function is called in switch views.
     * @throws IOException
     */
    private void createAllGamePanels() throws IOException {
         mainMenu = new MainMenu.MainMenuView(new MainMenu.MainMenuController(),this);
         String filename2 = "/FFIV_PSP_Final_Dungeon_Background_2.png";
        battleView = new BattleView(ImageIO.read(this.getClass().getResource(filename2)),
new HumanPlayer(),new AI(),this);
        String filename3 = "/Dungeon_Background_49.png";
        dungeonView = new DungeonView(ImageIO.read(this.getClass().getResource(filename3)),
this);
         gameViews.put("mainMenu", mainMenu);
         gameViews.put("battle", battleView);
         gameViews.put("dungeon", dungeonView);
         mainContentPane.add(mainMenu, "mainMenu");
         mainContentPane.add(battleView, "battle");
         mainContentPane.add(dungeonView, "dungeon");

    }
```

When it is time to switch we can tell the CardLayout to show that particular JPanel, which in our case are the separate views.

```
     /**
     * Function to switch between views in this game. Makes the current working view
invisible, then
     * calls the GameView startView function on the new view to add all necessary images /
start necessary threads in the new view.
     * @param activePanel
     * @param changeToView
     */
    public void switchViews(JPanel activePanel, String changeToView)
    {
       activePanel.setVisible(false);
       GameView currentView = gameViews.get(changeToView);
       currentView.startView();
       mainCard.show(mainContentPane, changeToView);
    }
```

To ensure we don't have many views / threads operating at once nothing in each JPanel is activated until the respective startView method is called. The startView method will create the necessary components of the upcoming view when it is to be displayed in order to save resources. From the BattleView class:

```
    /**
     * Start the battle. Creates the model and controller, sets all necessary fields, and
starts the BattleState thread.
     */
    public void startView()
    {
       this.startBattle();
    }

     /**
     * Creates the controller and model then starts the battle thread.
     */
    public void startBattle()
    {
              System.out.println("here");
              controller = new BattleController(this);
              model = new BattleModel(controller, human,AI);
              controller.setModel(model);
              model.battle();
              repaint();
              revalidate();
    }
```

Once the current view / controller / model has completed its purpose when it changes view again it is set inactive and invisible.

```
 public void switchViews(JPanel activePanel, String changeToView)
```

```
    {
        activePanel.setVisible(false);
```

The Dungeon

The Dungeon presents an area to allow The Hero, to move around within, a chest to loot for some helpful items, to approach and trigger a fight with the enemy upon the defeat of which the Hero can now get to the door he defends to escape the dungeon.

The ‘Dungeon’ follows MVC code styles to create a pane which contains the background on which all visual objects, ‘Sprites’, are painted. It uses the GridBag Layout to allow for a coordinate based original position to spawn the Sprites. All visual objects, (except the door) such as the Hero, Enemy, and chest as well as the alternate versions of the enemy and chest are turned from images into JPanels through the DungeonChar class

```
public DungeonChar(ImageIcon file)
        {
                Image fileImage = file.getImage();
                Image newFileImg = fileImage.getScaledInstance((int) (1024 / 9.0), 250,
java.awt.Image.SCALE_DEFAULT);
                file = new ImageIcon(newFileImg);
                this.Sprite = new JLabel(file);
        }
```

This class incorporates a Sprite attribute for each image as well as a rectangle bound for each image passed in this manner.

```
public Rectangle dimen(){
                return new Rectangle (Sprite.getX(), Sprite.getY(), Sprite.getWidth(),
Sprite.getHeight());
        }
```

To allow the door to be a visually appealing size it undergoes slightly different scaling.

```
Image newDoorImg = doorImage.getScaledInstance((int) (1024 / 3), 1000,
java.awt.Image.SCALE_DEFAULT);
door = new JLabel(doorImg);
```

All Sprites are spawned into onto the view with Gridbag Constraints to ensure appropriate spacing and order

```
GridBagConstraints c = makeGbc(0,0);
this.add(hero.Sprite, c);
c = makeGbc(1,0);
this.add(chest.Sprite, c);
this.add(chestHidden.Sprite, c);
c = makeGbc(2,0);
this.add(enemy.Sprite, c);
this.add(deadEnemy.Sprite, c);
c = makeGbc(3,0);
this.add(door, c);
```

The view has various getters to allow access to requisite elements for the model and controller.

After collision with the chest and interacting with it the entire view is remodelled to hide the chest but retain proper positions by making use of an invisible object

```
public void chestOpened()
{
	GridBagLayout lay = (GridBagLayout) this.getLayout();
	GridBagConstraints che = lay.getConstraints(chest.Sprite);
	this.hideChest(true);
	this.remove(chest.Sprite);
	this.remove(hero.Sprite);
	GridBagConstraints gbc = this.makeGbc(0,0);
	this.add(chestHidden.Sprite,gbc);
	this.add(hero.Sprite, che);

	repaint();
	revalidate();
}
```

The controller passes important signals from the model to the view, to signify when items should be hidden and when components should be move.

The model contains the methods to implement the keyboard listener which allows for keyboard controls as well as the collision function to send signals upon collision of the bounding rectangles for each sprite. Collision with each element is controlled by separate functions. Each collision stops the timer to prevent an unwanted continuous PRESSED input for the key and then acts based on what the collision is with. The classes contain specific values to allow the collision to be restricted on visual overlap instead of only the bounding boxes for each JPanel as the bounding boxes are much larger than the actual visual sprite. Collision with an enemy causes a switch to the battle view.

```
private void checkEnemyCollisions()
{
	Rectangle r1 = new Rectangle(component.getBounds());
	Rectangle r2 = new Rectangle(DungeonView.getEnemy().dimen());
	if((r2.getX() - 82) < r1.getX() && r1.getX() < (r2.getX() + 82))
	{
		timer.stop();
		// to set background to disappear you would need an instance
therefore would require the menu
		controller.signalStartBattle();
		Return;
	}
}
```

Collision with the chest causes a popup message which informs you of the spoils within you have gained.

```
        private void checkChestCollision()
        {
                Rectangle r1 = new Rectangle(component.getBounds());
                Rectangle r2 = new Rectangle(DungeonView.getChest().dimen());
                if((r1.getX() > r2.getX() - 45))
                {
                        timer.stop();
                        JOptionPane.showMessageDialog(null, "You Found \nAbilityPoints Potion
x1\nAntidote x1\nHealth Potion x1", "Golden Chest", JOptionPane.WARNING_MESSAGE);
                         soundEffectControl.SoundClipPlayer.playSound("/Inventory.wav");
                        looted  = true;
                        controller.signalHideChest();
                }
        }
```

Collision with the door allows one to exit the game.

```
        private void checkDoorCollisions()
        {
                Rectangle r1 = new Rectangle(component.getBounds());
                        Rectangle r3 = new Rectangle(DungeonView.getDoor().getBounds());
                if(r1.getX() > (r3.getX()+136))
                {
                        timer.stop();
                        String message = "Thanks for playing \n Would you like to stay for
        more?";
                        int reply = JOptionPane.showConfirmDialog(null, message, "Exit",
        JOptionPane.YES_NO_OPTION);
                        if (reply == JOptionPane.YES_OPTION) {
                                JOptionPane.showMessageDialog(null, "Freedom of choice is an
        illusion");
                                controller.signalSendToMainMenu();
                        }
                        else
                        {
                                 JOptionPane.showMessageDialog(null, "It was nice while it
                lasted");
                                controller.signalSendToMainMenu()
                        }
                }
        }
```

The keyboard controls are implemented via use of an inner helper class.

```
        private class AnimationAction extends AbstractAction implements ActionListener
        {
                 private Point moveDelta;
                /**Constructor calls to super from implementation of Action Listener
                 * @param key
                 * @param moveDelta
                 */
                public AnimationAction(String key, Point moveDelta)
                {
```

```
                super(key);
                    this.moveDelta = moveDelta;
            }
            public void actionPerformed(ActionEvent e)
            {
                handleKeyEvent((String)getValue(NAME), moveDelta);
            }
        }
```

This implements the actions for each key. The action performed method is repeatedly running when the timer is running to act on the action events to check collision and move the Hero.

```
        public void actionPerformed(ActionEvent e)
        {
            int[] newCoords = this.computeNewCoordinates();
            controller.signalMoveComponent(this.component, newCoords[0],
newCoords[1]);
            if(!looted)
            checkChestCollision();
            if(!battleOver)
            checkEnemyCollisions();
            if(battleOver)
            checkDoorCollisions();
        }
```

Battle

Each Hero class in the game has a integer speed rating and implements the Comparable interface.

public abstract class Hero implements Comparable<Hero>

More about this can be found in the Interface section.

At the start of the battle the Heroes from both the player and the AI party are added to a collection and sorted. The collection is initially sorted in ascending order which implies that a lower speed is better. Thus we reverse the order of this collection.

```
    private void fillInitialQueue()
    {
            // Add human party to array list to be sorted
            TreeMap<String, Hero> party = human.getParty();
            Collection<Hero> p = party.values();
            ArrayList<Hero> sortMe = new ArrayList<Hero>(p);

            // Add AI party to array list to be sorted
            TreeMap<String, Hero> aiParty = AI.getParty();
```

```
            Collection<Hero> aiP = aiParty.values();
            sortMe.addAll(aiP);

            // Sort the complete list
            Collections.sort(sortMe);
            // Since it sorts in ascending order
            // i.e lower speed first
            // Need to reverse the order
            Collections.reverse(sortMe);

            // Add to queue in sorted order
            for(Hero hero : sortMe)
            {
                  gameQueue.add(hero);
            }
    }
```

The battle runs in a separate thread from the main Java Swing thread.

public class BattleState extends Thread

More about this can be found under threads.

The queue pops off whomever is to go first and passes control to either the human player or AI by checking the “controlledBy” field of the current acting hero:

```
currentHero.getControlledBy().equals("AI")
```

From here players and enemies alike have various attacks.

- Items
    - Health Items
    - Ability Point Items
    - Status Effect Removal Items
- Abilities
    - Offensive Abilities
        - Offensive abilities with no status effects
        - Offensive abilities with per turn status effects
        - Offensive abilities with not per turn status effects (one apply that lasts until the duration is over)
    - Defensive Abilities
        - Defensive abilities with no status effects
        - Defensive abilities with not per turn status effects (one apply that lasts until the duration is over)
    - Crowd Control Abilities
        - Stuns the enemy skipping their turn

The battle continues until one party is composed of all dead characters.

```
private void checkBattleStatus()
{
	// Will remove characters from party when they die
	// So if either party is empty, end battle
	int deadCountHuman = partyDead(human.getParty());
	if(deadCountHuman == 4)
	{
		this.gameOver = true;
		System.exit(0);
	}
	int deadCountAI = partyDead(AI.getParty());
	if(deadCountAI == 4)
	{
		this.gameOver = true;
		System.exit(0);
	}
}
```

Highlighted Items and Abilities

Name	Type	Action
HealthPotion	Item	Add 2 to Health
AbilityPotion	Item	Add 2 to Ability Points
PoisonBlade	OffensiveAbility implements StatusEffectAbility	Does a base damage of 2, uses 4 ability points, applies the Poison status effect which is an OffensiveStatusPerTurn Status Effect
ExposeWeakness	OffensiveAbility implements StatusEffectAbility	Does a base damage of 1, uses 4 ability points, applies the LowerDefense status effect which is an OffensiveStatusNotPerTurn Status Effect
ShieldBash	CrowdControlAbility implements StatusEffectAbility, CrowdControlAbility	Does a base damage of 2, uses no ability points, applies CrowdControlStatus which stuns the enemy target for a duration

AI

The method of implementation of the AI is covered in the inheritance section.

Sound

Hall of Bones uses the Java sound API to render sounds such as music and sound effects across platforms. Sounds and music are stored as Microsoft WAV files. The SoundClipPlayer utility class under the soundControl package of the game handles the playback of these files. SoundClipPlayer abstracts the API of javax.sound.sampled.

The SoundClipPlayer API consists of 3 methods:

- playSound(String)
- playLoopingSound(String)
- shutUp()

As their names suggest, playSound and playLoopingSound playback a stored audio file. They take the path to the file as arguments. shutUp stops all playing sounds.

The overall implementation of playSound is as follows:

1. Parse a URL from the path of the file
2. Create an AudioInputStream from the URL using the getAudioInputStream(URL) method in the AudioSystem class.
3. Create a Clip from the AudioInputStream using the getClip() method of AudioInputStream.
4. Store the clip in a static ArrayList
5. Load the AudioInputStream created in Step 2 into the Clip using the open(AudioInputStream) method in Clip.
6. Invoke the start() method of the Clip, which plays the Clip.

The implementation of playLoopingSound is similar to that of playSound, except step 6 uses the loop(int) method instead of the start() method. The argument of loop is the constant Clip.LOOP_CONTINUOUSLY, which does what its name suggests.

Steps 1-5 of the implementation of playSound and playLoopingSound are delegated to the helper method loadAndOpenSound(String) which takes the argument of those methods as its argument. Delegation is an important way to avoid code-duplication, in effect increasing maintainability of code.

Step 4 of the implementation of playSound and playLoopingSound allow for the shutUp() method. This method iterates through the static ArrayList, invoking the stop() method of each Clip, stopping the playback of each Clip. It also removes them from the ArrayList for collection by the JVM garbage collector.

The SoundClipPlayer class is used throughout the three activities of the game (Main Menu, Dungeon, Battle) to provide a more immersive interactive experience. In all three activities, playLoopingSound is used to play music and give the player emotional tension. playSound is used for sound effects such inventory changes in the dungeon; and attacks, and items in battle. User experience is greatly improved with the use of this utility class.

OOP Features

Encapsulation/Information Hiding

Encapsulation: wrapping the data (variables) and code acting on the data (methods) together as a single unit

Encapsulation can be found within the:

1. Item class
2. Status class
3. Abilities class

Let's look at the Item class.

An Item should:

- Use itself
- Manage the inventory which is composed of items
    - Adding items
    - Removing items

In the item class we can find:

All inventory management functions:

```
	/**
	 * Check if the inventory contains the item you are looking for
	 * @param player
	 * @param item
	 * @return
	 */
	public boolean inventoryContains(Item item)
	{
		return player.getInventory().containsKey(item.toString());
	}

	/**
	 * Update the inventory count of an item. If the count is 0 the item is
removed from the inventory.
	 * @param item: item whose count to update
	 */
	public void updateInventory(Item item)
	{
		int currentCount = item.getCount();
		currentCount--;
		item.setCount(currentCount);
		if(currentCount <= 0)
```

```
        {
                player.getInventory().remove(item.toString());
        }
        else
        {
                player.getInventory().put(item.toString(), item);
        }
}

/**
 * Adds the current item to the inventory of the given player.
 * @param Player player
 */
public void addItem()
{
        String k = this.toString();
        Item val = player.getInventory().put(k,this);
        if(val == null)
        {
                player.getInventory().put(k, this);
        }
        else
        {
                val.setCount(val.getCount() + 1);
                player.getInventory().put(k, val);
        }
}

/**
 * Gets an item from the inventory by the item's name.
 * @param itemName, the name of the item you wish to retrieve
 * @return Item: the item you wish to retrieve
 */
public Item getItem()
{
        return player.getInventory().get(this.toString());
}
```

Let's look at the Player class where the inventory of items is stored:

The creation of the defaultInventory uses Item's method to fill the inventory VS the Player's

```
/**
 * Fills the default inventory on party creation.
 */
private void defaultInventory()
{
        // this refers to the items' player
        HealthItem defaultHealth = new HealthItem("Health Potion",2,this);
        defaultHealth.addItem();
```

```
            AbilityItem abilityPointsPotion = new AbilityItem("AbilityPoints
Potion",2,this);
            abilityPointsPotion.addItem();
            StatusItem antidote = new StatusItem("Antidote","Poison",this);
            antidote.addItem();
      }
```

There are also no getters for Items, only the full inventory:

```
public TreeMap<String,Item> getInventory()
public ArrayList<Item> inventoryItems()
```

The encapsulation methods of the Status class and Ability classes are extremely similar to to the item class. The main difference is that Status and Ability affect individual heros and their status / ability maps while Items affect a Player / Party's inventory.

Overloading

First we will covering overloading. The useBattleCommand method is overloaded with three different options. One prompts a human player to pick a target within the GUI, another accepts a predetermined target, and the third is for the AI who pick their own target but also wish to animate the attack.

Predetermined target:

```
/**
* All offensive abilities will be applying damage to a target. This this method
computes
* the damage done to the target by taking the difference between their defense
rating and the
* abilities damage done. If the target's defense rating is higher than the damage
done by this ability
* the damage done is 1. This method also updates the ability points of the current
acting hero.
* @param hero current acting hero
* @param target target hero
* @throws A child of BattleModelException if the defensive ability or item used
cannot increase the statistic that it affects, if you pick a
* target that isn't alive, or if the character doesn't have enough ability points.
*/
      public void useBattleCommand(Hero hero, Hero target) throws
BattleModelException
      {
            hero.checkIfEnoughAP(this);
            int otherDefenseRating = target.getDefenseRating();
            int damageDone = this.getDamage() - otherDefenseRating;
            if(damageDone <= 0)
            {
```

```
                damageDone = 1;
            }
            int otherHealth = target.getHealth();
            int newHealth = otherHealth - damageDone;
            System.out.println("AI" + " health change: " + target.getHealth());
            hero.setAbilityPoints(hero.getAbilityPoints() - this.getPointCost());
            target.setHealth(newHealth);
            System.out.println("AI" + " health Change: " + target.getHealth());
        }
```

Prompt for target then animate:

```
/**
* All offensive abilities will be applying damage to a target. This this method
computes
* the damage done to the target by taking the difference between their defense
rating and the
* abilities damage done. If the target's defense rating is higher than the damage
done by this ability
* the damage done is 1. This method also updates the ability points of the current
acting hero.
* @param controller battle controller to animate.
* @param hero current acting hero
* @throws A child of BattleModelException if the defensive ability or item used
cannot increase the statistic that it affects, if you pick a
* target that isn't alive, or if the character doesn't have enough ability points.
*/
        public void useBattleCommand(Hero hero, BattleController controller) throws
BattleModelException
        {
            hero.checkIfEnoughAP(this,controller);
            Hero target = controller.signalShowTargetOptions();
            if(target == null)
            {
                throw new CancelledTargetException(controller);
            }
            useBattleCommand(hero,target);
            controller.animateBattleCommand(target, this.getAnimationImage(),
true);
        }
```

AI battle command with animation:

```
/**
* All offensive abilities will be applying damage to a target. This this method
computes
* the damage done to the target by taking the difference between their defense
rating and the
* abilities damage done. If the target's defense rating is higher than the damage
done by this ability
```

```
* the damage done is 1. This method also updates the ability points of the current
acting hero.
* @param controller battle controller to animate.
* @param hero current acting hero
* @param target target hero
* @throws A child of BattleModelException if the defensive ability or item used
cannot increase the statistic that it affects, if you pick a
* target that isn't alive, or if the character doesn't have enough ability points.
*/

	public void useBattleCommand(Hero hero, Hero target, BattleController
controller) throws BattleModelException
	{
		useBattleCommand(hero,target);
		controller.animateBattleCommand(target, this.getAnimationImage(),
true);
	}
```

Constructors

There are many typical constructors in this project. Rather than cover the “regular” constructors let’s look at something more interesting.

public abstract class Status implements Cloneable

Above is the declaration of the abstract class Status which implements Clonable

The Status class includes two constructors, one regular parameter constructor and another copy constructor that implements constructor chaining:

```
	/**
	 * Status argument constructor.
	 * @param name: status name
	 * @param statAffected: stat affected
	 * @param effectStrength: effect strength
	 * @param duration: duration of status
	 * @param defaultDuration: default duration of status
	 */
	public Status(String name, String statAffected, int effectStrength, int
duration, int defaultDuration)
	{
		this.name = name;
		this.statAffected = statAffected;
		this.effectStrength = effectStrength;
		this.duration = duration;
		this.defaultDuration = defaultDuration;
		this.time = new Date().getTime();
	}
```

```
	/**
	 * Status copy constructor. Used when updating and adding status effects
heros to create deep copies in clone so references aren't shared in common.
	 * If references were shared in common when the duration of the applied
status is changed the duration of the
	 * status effect applied by a hero's ability would also be changed.
	 * @param other: Status to copy
	 */
	public Status(Status other)
	{
		this(other.name, other.statAffected, other.effectStrength,
other.duration, other.defaultDuration);
		// New time stamp is created in the original constructor
		// This way we can stack status effects
	}
```

So why implement cloneable?

```
	/**
	 * Clone method, since this is an abstract class we can't create instances
of it.
	 * This is used in updating a hero's statuses to create a copy of the
proper run type
	 * so we can still apply polymorphism.
	 *
	 * Why not a static factory method?
	 * Polymorphism doesn't apply to static methods.
	 */
	public abstract Status clone();
```

When status effects are updated in the Hero class I create a copy of that Hero's statuses Map:

```
	/**
	 * The hero must update it's own statuses, the status class only adheres to
a single status.
	 * Thus the hero iterates through it's own statuses and calls the
individual status's update status method
	 * to update the hero's state.
	 * @return boolean: if the hero is crowd controlled no further action can
be made on it's turn
	 */
	public boolean updateStatuses()
	{
		boolean crowdControlled = false;
		if(!this.statuses.isEmpty())
		{
			// Make deep copy to reference from to delete from in for each
loop
			// Otherwise get error
```

```
                    // Not using Iterator because already returning one value from
this.updateStatus
                    // Don't wish to deal with an array of booleans as a return,
so just make deep copy
                    HashMap<String,Status> tmpCopyForDeletion = new
HashMap<String,Status>();
                    for(Map.Entry<String, Status> entry :
this.statuses.entrySet())
                    {
                          tmpCopyForDeletion.put(entry.getKey(),
entry.getValue().clone());
                    }

                    // Operate on the characters status effects using the old keys
generated
                    // When they were applied to the character
                    // While using the tmpCopy so the status effects can be
deleted
                    for(Map.Entry<String, Status> entry :
tmpCopyForDeletion.entrySet())
                    {
                          Status currentStatus =
this.statuses.get(entry.getKey());
                          boolean currentCrowd =
currentStatus.updateStatus(this);
                          if(currentCrowd)
                          {
                                crowdControlled = currentCrowd;
                          }
                    }

             }
             return crowdControlled;
      }
```

To insure that the same Status classes can stack on a hero we must make NEW Status classes. Since I don't wish to check instanceof for every status to create the proper Status which could be placed in the HashMap I implemented the Clonable interface and clone deep copies with a new timestamp created from the copy process.

With the Clonable interface this forces each subclass to implement clone. Thanks to co-variant return types on methods that return classes regardless of which descendent of Status implements the clone method, the return type can be of that class.

```
      /**
       * Clone method, since this is a abstract class we can't create instances
of it.
       * This is used in updating a hero's statuses to create a copy of the
proper run type
       * so we can still apply polymorphism.
```

```
         *
         * Why not a static factory method?
         * Polymorphism doesn't apply to static methods.
         *
         * This method also displays an implementation of co-variant return type:
the class type returned in an overloaded method has been changed
         * @return OffensiveStatusPerTurn deep copy with new time stamp to allow
stacking
         *
         */
        @Override
        public OffensiveStatusPerTurn clone() {
                return new OffensiveStatusPerTurn(this);
        }
```

Why not a static factory method to return a Status type?

Late binding does not apply to static methods, thus I would have to check the instanceof each status to determine which is the proper static method to call.

Static Methods

```
        /**
         * Returns the hero's ability from the hero's ability set by it's key.
         * Why static?
         * Don't need to have an ability instance to get an ability from the Map.
         * @param hero: hero to get ability from
         * @param ability: name to find in hero's HashMap
         * @return Ability
         */
        public static Ability getAbility(Hero hero, String ability)
        {
                return hero.getAbilities().get(ability);
        }
```

Why static?

If I am looking to get an ability from a Hero class' ability Map by the NAME of the ability odds are I don't already have an ability class to call that method on. If I already had an ability class I could just use the overloaded version of this method on the ability instance.

```
        /**
         * Returns the hero's ability from the hero's ability set using the ability
instance.
         * @param hero: hero to get ability from
         * @return Ability
```

```
     */
    public Ability getAbility(Hero hero)
    {
        return hero.getAbilities().get(this.getName());
    }
```

Another static method is the ImagePreparation Static Factory Method:

```
    private static final ImagePreparation instance = new ImagePreparation();

    /**
     * Private constructor, empty by intention as no fields necessary.
     */
    private ImagePreparation()
    {

    }

    /**
     * Get the singleton instance of this class.
     * @return ImagePreparation instance
     */
    public static ImagePreparation getInstance()
    {
        return ImagePreparation.instance;
    }
```

This class cannot directly be a utility class as we wish to use the
new ImageIcon(this.getClass().getResource(path));
method of loading images from the source images folder (NOTE: it uses the “this” parameter which is not available to a static method). Thus it is made into a singleton class which any other class can get the one instance of (not necessary to have many instances of this class) then prepare all of their images with the instance returned.

Example:

```
private static final Image ANIMATION_IMAGE =
ImagePreparation.getInstance().prepImage("/fireball.png", ABILITY_IMAGE_WIDTH,
ABILITY_IMAGE_HEIGHT);
```

Static Variables

There are loads of static variables in this project. The most notable to me are the constants set for positioning various elements within the battle view’s grid bag layout:

```
    // Declare constants for positioning
    // Y for row positioning
    public static final int STATUS_Y = 0;
```

```
public static final int HEALTH_BAR_Y = 1;
public static final int AP_BAR_Y = 2;
public static final int MOVE_TEXT_Y = 3;
public static final int INDICATOR_Y = 4;
public static final int CHARACTER_Y = 5;
public static final int ABILITY_Y = 6;

// Position to indicate grid height and grid width in makeGbc function
public static final int BARS_POS = 1;
public static final int INDICATOR_POS = 0;
public static final int STATUS_POS = 0;
public static final int CHAR_POS = 2;
public static final int MOVE_TEXT_POS = 0;
public static final int ABILITY_POS = 4;
```

Now when it comes time to add the images to JLabels then into the grid bag layout it can be done in a much more readable way VS using magic numbers:

```
TreeMap<String, Hero> party = player.getParty();
            Collection<Hero> p = party.values();
            int gridX = 0;

            for(Hero hero : p)
            {
                  // Create hero images
                  GridBagConstraints c = makeGbc(gridX,CHARACTER_Y,CHAR_POS);
                  JLabel imageLbl = prepImage(hero.getImage());
                  pane.add(imageLbl, c);
                  this.charPos.put(gridX, hero);
                  hero.setPosition(gridX);

            // Other stuff here
            }
```

Mutable and Immutable Classes

I believe all classes implemented are mutable i.e at the very least they have setters.

Why?

Items need to adjust their count
Statuses need to tick their duration down until they have expired.
Hero’s take damage and their statistics change on a regular basis in battle.
Abilities should be able to change their statistics on level up (not implemented).
The view constantly changes in the dungeon, battle, animations appear, text, etc...

Inner Classes

Inner Classes were used in the battle system GUI production to create action listeners that could contain fields or pop up menus.

```
public class AbilityListener implements ActionListener {

	private Hero watchingHero;

	public AbilityListener(Hero hero)
	{
		this.watchingHero = hero;
	}
// Other stuff here
```

We also used inner classes to store various classes abilities and group statues into one class file VS having multiple class files.

Only public static classes were used: provides for easier construction and they don't need to see non-static fields from other inner classes in the same file.

For example:

```
public abstract class SoldierAbility extends Ability {

	private static final String BELONGS_TO_CLASS = "SOLDIER";

	/**
	 * HamString class is an OffensiveAbility. Note the absence of the
useBattleCommand method.
	 * Since this is a pure OffensiveAbility there is no need to redefine this
method, it will immediately call OffensiveAbiltiies
	 * useBattleCommand.
	 * @author Kevin
	 *
	 */
	public static class HamString extends OffensiveAbility {
		public static final String NAME = "HamString";
		private static final int ABILITY_POINTS_COST = 4;
		private static final int BASE_DAMAGE = 6;

		/**
		 * Default constructor that sets the ability point cost and damage
done.
		 */
		public HamString()
		{
			super(ABILITY_POINTS_COST,BASE_DAMAGE);
```

```
                this.setName(HamString.NAME);
        }

        /**
         * Returns the class string that owns these abilities.
         * @return Class string that owns these abilities.
         */
        @Override
        public String getClassOwner() {
                return SoldierAbility.BELONGS_TO_CLASS;
        }

    }

    /**
     * Lunge class is an OffensiveAbility. Note the absence of the
useBattleCommand method.
     * Since this is a pure OffensiveAbility there is no need to redefine this
method, it will immediately call OffensiveAbiltiies
     * useBattleCommand.
     * @author Kevin
     *
     */
    public static class Lunge extends OffensiveAbility {
        public static final String NAME = "Lunge";
        private static final int ABILITY_POINTS_COST = 4;
        private static final int BASE_DAMAGE = 5;

        /**
         * Default constructor that sets the ability point cost and damage
done.
         */
        public Lunge()
        {
                super(ABILITY_POINTS_COST,BASE_DAMAGE);
                this.setName(Lunge.NAME);
        }

        /**
         * Returns the class string that owns these abilities.
         * @return Class string that owns these abilities.
         */
        @Override
        public String getClassOwner() {
                return SoldierAbility.BELONGS_TO_CLASS;
        }
    }

// More stuff here
}
```

Interfaces

The use of cloneable has been covered in constructors.

The use of comparable has been covered in the Introduction Battle section.

In this section we will instead discuss the StatusEffectAbility, StatGet, and StatSet interfaces.

First let's look at StatusEffectAbility

```
/**
 * This interface provides only one method, getAbilityStatus().
 * If this interface is implemented properly in the corresponding abilities
 * useBattleCommand method it should call, clone, and then
 * apply getAbilityStatus to it's enemy.
 * @author Kevin
 *
 */
public interface StatusEffectAbility {
	/**
	 * Applies and returns a clone of an abilities Status effect.
	 * If this interface is implemented properly in the corresponding abilities
	 * useBattleCommand method it should applyAbilityStatus to it's enemy.
	 * @param target target to apply status effect to
	 * @return Cloned Abilities status effect with new time stamp
	 */
	Status applyAbilityStatus(Hero target);
```

To see this in action let's look at the relevant methods from the PoisonBlade ability.

public static class PoisonBlade extends OffensiveAbility implements StatusEffectAbility

```
	/**
	* Calls the regular OffensiveAbility useBattleCommand to apply the base
damage,
	* then applies the status to the target.
	* @param hero: current acting hero
	* @param target: target
	*/
	@Override
	public void useBattleCommand(Hero hero, Hero target) {
		super.useBattleCommand(hero, target);
		// Add status
		// Is cloned within the status
		applyAbilityStatus(target);
	}
```

```
	/**
	* Applies and returns a clone of an abilities Status effect.
	* If this interface is implemented properly in the corresponding abilities
	* useBattleCommand method it should applyAbilityStatus to it's enemy.
	* @param target target to apply status effect to
	* @return Abilities status effect
	*/
	@Override
	public Status applyAbilityStatus(Hero target)
	{
		return this.poison.addStatus(target);
	}
```

So why use a interface when I could have made a separate offensive ability class that applies statuses automatically?

By using this simple interface I can apply multiple inheritance thus I don't produce a million separate classes for a small difference in the application of useBattleCommand.

Next up we will analyze the StatGet / StatSet interfaces which will be used to create HashMaps of anonymous functions within the Hero class. Only the StatGet example will be pasted here, the StatSet is a setter VS getter.

```
/**
 * Interface to make a HashMap of anonymous functions that refer to a Hero classes getters within the Hero class.
 * This allows me to set a string in each status effect of the stat it will effect.
 * Rather than specifically define each stauts affected i.e Hero.getStat("health") vs Hero.getHealth()
 * So many status effects can use the general form of hero.getStat(String stat)
 * Saves me from rewriting the update and apply status functions numerous times.
 * @author Kevin
 *
 */
public interface StatGet {
	/**
	 * Will be used to create anonymous functions within a HashMap of the Hero class that represent the Hero's getter functions.
	 * @param hero: current hero we wish to get stats from
	 * @return A getter function from the hero
	 */
	int statGet(Hero hero);
}
```

With this interface defined we can make the HashMaps of anonymous functions in the Hero class.

```
private static HashMap<String, StatGet> getStats = new HashMap<String, StatGet>();
```

```
private static HashMap<String, StatSet> setStats = new HashMap<String, StatSet>();
static
{
	Hero.getStats.put("health", new StatGet() {public int statGet(Hero hero) {
	return hero.getHealth();}});

	Hero.getStats.put("abilityPoints", new StatGet() {public int statGet(Hero
	hero) { return hero.getAbilityPoints();}});

	Hero.getStats.put("defenseRating", new StatGet() {public int statGet(Hero
	hero) { return hero.getDefenseRating();}});

	Hero.setStats.put("health", new StatSet() {public void statSet(Hero hero,
	int val) { hero.setHealth(val);}});

	Hero.setStats.put("abilityPoints", new StatSet() {public void statSet(Hero
	hero, int val) { hero.setAbilityPoints(val);}});
	Hero.setStats.put("defenseRating", new StatSet() {public void statSet(Hero
	hero, int val) { hero.setDefenseRating(val);}});
}
```

So what is the point of all of this?

To answer this question we have to look at the one the Status's children. OffensiveStatusPerTurn will do fine.

```
	/**
	 * ApplyStatusEffect adjusts the appropriate statistic on the hero
depending on the status effect.
	 * Since this is a OffenseStatus the amount is SUBTRACTED to the hero's
statistic VS a BuffStatus in which it is ADDED.
	 * Reduces the duration of the status effect, if the duration is 0 it will
be removed in    updateStatus.
	 * @param hero: who to apply the status effect to
	 */
	@Override
	public void applyStatusEffect(Hero hero)
	{
		int duration = this.getDuration();
		int effectStrength = this.getEffectStrength();
		String statAffected = this.getStatAffected();
		int statToChange = hero.getStat(statAffected);
		int changedStat;
		if(effectStrength > 0)
		{
			changedStat = statToChange - effectStrength;
		}
		else
		{
			changedStat = statToChange + effectStrength;
		}
```

```
            hero.setStat(statAffected, changedStat);
            System.out.println(this.toString() + " being updated. Stat affected:
" + this.getStatAffected() + " before: " + statToChange + " after: " +
changedStat);
            duration--;
            this.setDuration(duration);
            // If duration is 0 remove the status
            if(duration == 0)
            {
                  this.removeStatus(hero);
            }
      }
```

Since these anonymous functions were defined in HashMap with the keys composed of the name of each statistics rather than define the EXACT getters and setters for each new status ability, I just define a String adhering to the statAffected in the status:

```
      public static class Poison extends OffensiveStatusPerTurn
      {
            public static final String NAME = "Poison";
            private static final String STAT_AFFECTED = "health";
```

This prevents me from having to make specific status effects classes for each statistic; instead I can just define a string.

Abstract Classes

There are many abstract classes in the project.

1. Player
2. BattleCommand
3. Ability
4. Item
5. Status
6. Hero

And more….

So why are these abstract, below we've covered a few classes and their reasons.

Status:

```
/**
 * Status abstract class.
 * Why abstract?
 * Don't know what type of status is applied yet, does it tick per turn or is it a
one
 * time application?
 * @author Kevin
```

```
 *
 */
```

Player:

```
/**
 * Player is an abstract class that will be extended by HumanPlayer and AI player.
 * It serves as a wrapped that will hold a participating entities inventory and
party of heros.
 * To adhere to encapsulation the Heros composing the party and the Items
composing the inventory will manage all
 * Heros / Items / Inventory / Party actions. This class is just to store these
Maps.
 * Why abstract?
 * 1) We don't know the party composition yet, the AI and the Human player have
access to different Classes
 * 2) Having just one class Player for both AI and Human would not adhere to
encapsulation : we need to separate the AI's functions from a Human's
 * during battle
 * @author Kevin
 *
 */
```

Item:

```
/**
 * The item abstract class defines the abstract method useItem which is overridden
by
 * each specific subclass of Item. Beyond this to implement Encapsulation is
manages all Item related inventory control.
 * It is also abstract so the class cannot be used to create a stand alone object.
 * Why? The Classes field that the item affects has yet to be defined.
 * @author Kevin
 *
 */
```

Inheritance

There are numerous examples of inheritance in this project. The following are a few:

1. Both HumanPlayer and AI extend Player, an abstract class that declares some fields, implementations of common methods for both classes, and some abstract methods
2. Both Item and Ability extend BattleCommand.

   Let's follow Ability and it's chain of inheritance. The following is an example of some of the hierarchies of BattleCommand's inheritance tree
   1. BattleCommand

2. Ability
    a. OffensiveAbility
        i. HamString
        ii. PoisonBlade
        iii. FireBall
    b. SoliderAbility
3. Status -> OffensiveStatusPerTurn

Inheritance also played a large part in how the computer controlled characters acted in a fight.

The Monster class inherited from the Player class in order to have both human and computer characters act on the same variables so they could all be controlled with the same methods.

From here the Monster class is then inherited by a number of sub-types (Punisher, Defender ect.) which are used to define how the Monster will act during combat. Each specific Monster class (SkeletonBoss, SkeletonArcher ect.) inherits from a Monster sub-type.

With this setup the only work that needs to be put into creating an individual Monster is defining it's ability list and combat relevant stats (player specific stats are unused for Monsters).

Polymorphism

There is an abundance of polymorphism in use in this project.

Consider the ability PoisonBlade.

PoisonBlade inherits from OffensiveAbility which inherits from Ability which inherits from BattleCommand.

PoisonBlade is stored in a HashMap<String, Ability> in the Hero class per particular hero.

When the battle commences and you input a command, it is retrieved as an Ability.

```
Ability abilityToUse = Ability.getAbility(watchingHero, cmd);
```

On that ability useBattleCommand is called.

```
abilityToUse.useBattleCommand(watchingHero, targetedChar);
```

Here polymorphism is used to determine the RUN TYPE not the declared type of the invoking class. Thus if the abilityToUse is the PoisonBlade ability then PoisonBlade's useBattleCommand is called.

Same idea applies to the Item class.

An item is selected during battle, returned as the parent type Item.

```
	for(int i = 0; i<menuItems.size(); i++)
	{
		JMenuItem currentItem= menuItems.get(i);
		Item item = items.get(i);
		currentItem.addActionListener((ActionEvent e) -> {
		try {
			// Try and use the item
			// It useItem throws MaximumStatException if your character is
at the maximum of said stat
			item.useBattleCommand(currentHero,controller);
			itemPopUp.setVisible(false);
			controller.signalDisplayAbilityUsed(currentHero, item);
			updateOnSuccessfulEvent(currentHero);
			gameState.resumeThread();
			} catch (BattleModelException itemException) {
				itemException.sendControllerErrorSignal();
					}
			 });
		}
	}
	controller.signalItemPopUp(itemPopUp);
```

The RUN TYPE of the item is determined on invocation when we call

```
it.useBattleCommand(currentHero, null);
```

For instance if the item is of class HealthItem, then HealthItem's useBattleCommand is called.

Same idea applies to the Status class.

Hero's store a HashMap of Status. When it comes time to update each individual Status the RUN TYPE's update status is called.

```
boolean currentCrowd = currentStatus.updateStatus(this);
```

Generics

A generic method was used inside the Monster class to give variability to the enemies behaviours. The method will take in an ArrayList containing any object type. Based on the list's size it will pick a random element from that list. The return type matches the object type of the ArrayList.

```
public static <T> T pickRandom(ArrayList<T> options) {
	Random random = new Random();
	int pick = random.nextInt(options.size());
	T choice = options.get(pick);
	return choice;
}
```

This method made it simple to make random choices whenever the computer controlled enemies had more than a single option, or had a possibility to have more than one.

This method working with the other random methods which defined the enemies behaviours removed the ridged feeling it had when the project first started.

Array and ArrayList/Collections

Collections are used throughout this project to manage Items, Statuses, Heros, Abilities, and StatusBars on the View.

Perhaps the most interesting collect is the queue is used to determine the turn order during battle.

```
private Queue<Hero> gameQueue = new LinkedList<Hero>();

private void fillInitialQueue()
{
	// Add human party to array list to be sorted
	TreeMap<String, Hero> party = human.getParty();
		Collection<Hero> p = party.values();
		ArrayList<Hero> sortMe = new ArrayList<Hero>(p);

		// Add AI party to array list to be sorted
		TreeMap<String, Hero> aiParty = AI.getParty();
		Collection<Hero> aiP = aiParty.values();
		sortMe.addAll(aiP);

		// Sort the complete list
		Collections.sort(sortMe);
		// Since it sorts in ascending order
		// i.e lower speed first
```

```
                // Need to reverse the order
                Collections.reverse(sortMe);

                // Add to queue in sorted order
                for(Hero hero : sortMe)
                {
                        gameQueue.add(hero);
                }
 }
```

Once a hero takes their turn if they are re added to the queue.

```
if(currentHero.getHealth() > 0)
{
                // Hero takes turn…..
                gameQueue.add(currentHero);
}
```

If the queue pops a hero who is dead they are not re added to the queue

Exceptions

There are various exceptions implemented in this project. They all adhere to the BattleModel.

All BattleModel exceptions inherit from the abstract class BattleModelException which defines the abstract method:

```
        /**
         * Abstract method as we don't know what signal is sent yet.
         */
        public abstract void sendControllerErrorSignal();
```

Beyond the two normal exception constructors all BattleModelExceptions have a third constructor:

```
        /**
         * Construct this exception with a controller to send the error signal to.
         */
        public BattleModelException(BattleController controller)
        {
                super();
                this.controller = controller;
        }
```

This allows them to send the controller a signal to send a signal to the view to fire off the exception message.

For example:

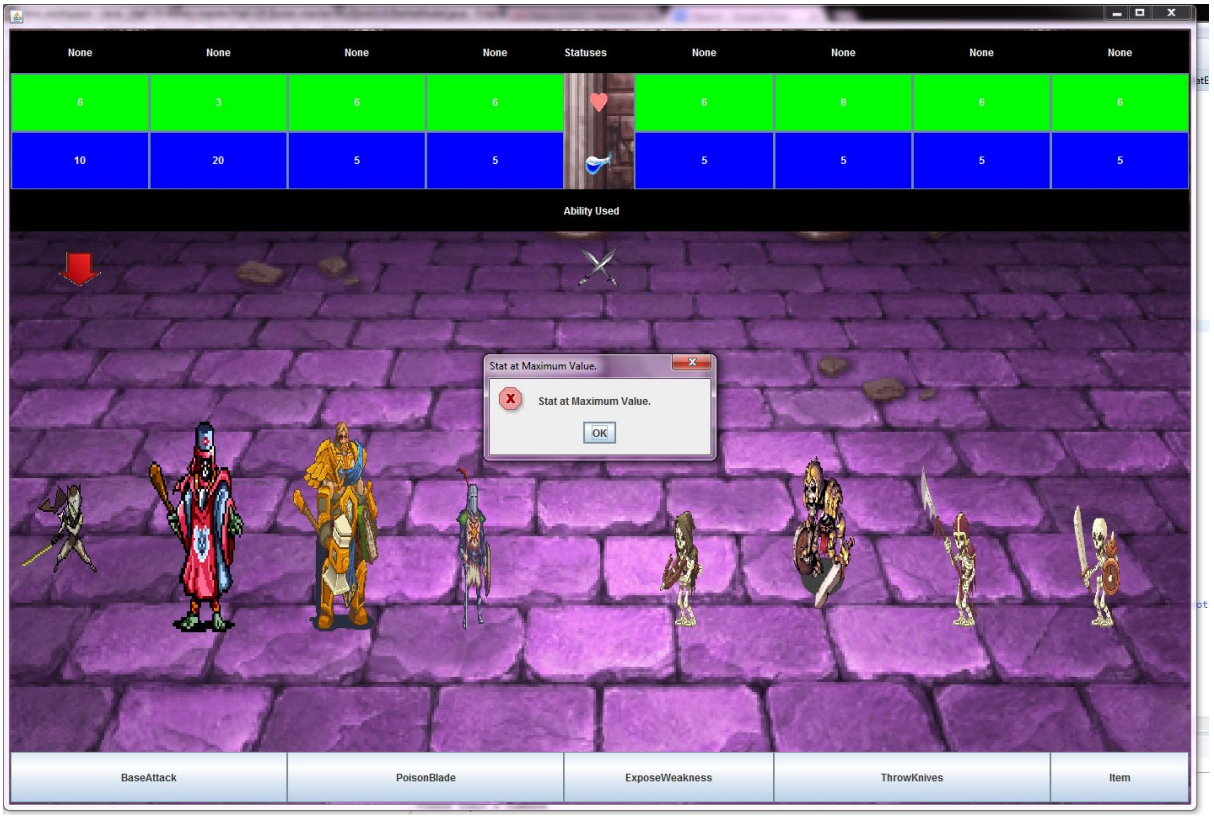
```
public class MaximumStatException extends BattleModelException

	/**
	 * Send message to controller to signal view to display error popup.
	 */
	@Override
	public void sendControllerErrorSignal() {
		this.getController().sendMaxStatExceptionSignal();
	}
```

Then in the BattleModel…. Lines 452 …

```
			try{
				abilityToUse.useBattleCommand(watchingHero, controller);
					// Update statuses
					gameState.resumeThread();
				controller.signalDisplayAbilityUsed(watchingHero,
abilityToUse);
				controller.signalUpdateStatuses();
					// Animate Attack
					updateOnSuccessfulEvent(watchingHero);
			}
			catch(BattleModelException battleException)
			{
				battleException.sendControllerErrorSignal();
			}
```

Due to polymorphism we don't have to write a chain try catch block, rather just a single catch with a single line.



File I/O

All image files for this project are stored within an added source folder: images.

The path's to each individual image for the Hero's are set within the specific class i.e

```
public class Soldier extends Hero{
	private static final String IMAGE = "/human_warrior.gif";
```

Since most of the images are GIFs or transparent PNGs they are loaded, and generally resized with a function similar to:

```
private JLabel prepImage(String path)
	{
		ImageIcon ii = new ImageIcon(this.getClass().getResource(path));
		Image image = ii.getImage(); // transform it
		Image newimg = image.getScaledInstance((int) (1024 / 9.0), 250,
java.awt.Image.SCALE_DEFAULT); // scale it the smooth way
		ii = new ImageIcon(newimg);  // transform it back
		JLabel imageLbl = new JLabel(ii);
		return imageLbl;
	}
```

Why all of this work?

Can’t directly resize an ImageIcon. While an Image is resizable it does not display gif animations.

Outside of images no other files are loaded.

Functional Programming

Use a lambda function to generate the pop up menu.

```
	currentItem.addActionListener((ActionEvent e) -> {
			try {
			 // Try and use the item
			 // It useItem throws MaximumStatException if your character is at the maximum of said stat
				item.useBattleCommand(currentHero,null);
				itemPopUp.setVisible(false);
				battleGUI.displayHeal(currentHero);
				unPauseOnSuccessfulEvent(currentHero);
				gameState.resumeThread();
				} catch (MaximumStatException e1) {
					JOptionPane.showMessageDialog(battleGUI,"Stat at Maximum Value.", "Stat at Maximum Value.", JOptionPane.ERROR_MESSAGE);

				}
		});
```

Model/View/Controller (MVC) and Threads

View

MVC was implemented within the Battle System MVC. Let’s start with the view: **BattleView**.

A view should only deal with displaying components. It is a GridBagLayout JPanel in which the view as designed to contain:

7 Rows
9 Columns

There are LOADS of display functions to be found in the view. Let’s look at perhaps the largest one.

For example:

```
	/**
```

```
        * Add all characters and their associated displays i.e their ability used,
indicators, and status used bars.
        * @param player player whose characters you wish to display
        * @param ai AI whose characters you wish to display
        */
       private void addCharacters(Player player, AI ai)
       {
              TreeMap<String, Hero> party = player.getParty();
              Collection<Hero> p = party.values();
              int gridX = 0;

              for(Hero hero : p)
              {
                     addAllDisplayBarsPerCharacter(hero, gridX);
                     gridX ++;
              }

              TreeMap<String, Hero> aiParty = ai.getParty();
              Collection<Hero> aiP = aiParty.values();
              gridX = 5;
              for(Hero hero : aiP)
              {
                     addAllDisplayBarsPerCharacter(hero, gridX);
                     gridX ++;
              }

              revalidate();
              repaint();
       }
```

Here we can see the view placing the hero and it's many related view components at an increasing gridX position in the GridBagLayout.

```
       /**
        * Adds the character and their associated displays i.e their ability used,
indicators, and status used bars.
        * @param hero the hero you wish to add
        * @param gridX which column you wish to display the character
        */
       private void addAllDisplayBarsPerCharacter(Hero hero, int gridX)
       {
              // Create hero images
              GridBagConstraints c = makeGbc(gridX,CHARACTER_Y,CHAR_POS);
              JLabelWithToolTip imageLbl =
ImagePreparation.getInstance().getToolTipJLabel(hero,
hero.getImage(),gridX,CHARACTER_WIDTH,CHARACTER_HEIGHT);
              this.toolTipLabels.add(imageLbl);
              this.add(imageLbl, c);
              this.charPos.put(gridX, hero);
              hero.setPosition(gridX);
```

```
                // Create the turn indicators
                // NOTE: they go above the characters
                c = makeGbc(gridX,INDICATOR_Y,INDICATOR_POS);
                JLabel indicator =
ImagePreparation.getInstance().attachImageIconToJLabel(INDICATOR_PATH,INDICATOR_WI
DTH,INDICATOR_HEIGHT);
                indicator.setVisible(false);
                this.add(indicator,c);
                indicatorPos.put(gridX, indicator);

                // Create JLabels for status effect text
                c = makeGbc(gridX,STATUS_Y,STATUS_POS);
                StatusBar currentStatusBar = new StatusBar(hero,"None");
                this.statusBars.add(currentStatusBar);
                this.add(currentStatusBar, c);

                // Create JLabels for the ability text
                c = makeGbc(gridX,MOVE_TEXT_Y,MOVE_TEXT_POS);
                AbilityNotifier currentNotifier = new AbilityNotifier(hero);
                this.abilityNotifiers.put(gridX, currentNotifier);
                this.add(currentNotifier, c);

                revalidate();
                repaint();
        }
```

Thanks to the static variables mentioned in the static variables section this code is much more readable with the constants. You can see above that each character and their respective bars are placed in their designated positions by the view.

The view is also responsible for starting animation **threads** and displaying error messages.

```
        private Thread animator; // In GameViewPanel

        // Rest from BattleView which extends GameViewPanel
        /**
         * Show Not Enough Ability Points Error Message
         */
        public void showNotEnoughAbilityPointsMessage()
        {
                JOptionPane.showMessageDialog(this,"Not Enough AP.", "Not Enough
AP.",                 JOptionPane.ERROR_MESSAGE);
        }

        /**
         * Start battle down animation.
         * @param startX x starting position
         * @param attackAnimationStartY y starting position
         * @param animationImage image to animate
         */
```

```
      public void startBattleDownAnimation(int startX, int attackAnimationStartY,
Image animationImage)
      {
            pane.startDownAnimation(startX,
attackAnimationStartY,animationImage);
       }
```

These animations are signaled to the view by the BattleController.

Controller

The controller sets ability listeners on buttons as well as sends various signals to the view to update or display messages.

Following the above example of displaying an error message the related signal to the view from the controller can be found:

```
     /**
     * Send the signal to the display not enough ability points error message.
     */
    public void sendNotEnoughAbilityPointsSignal()
    {
       view.showNotEnoughAbilityPointsMessage();
    }

     /**
     * Send the signal to the view to animate the appropriate battle command.
     * @param targetedChar target to animate over
     * @param animationImage image to animate
     * @param down whether to animate up or down
     */
    public void animateBattleCommand(Hero targetedChar, Image animationImage,
boolean down)
    {
       int startX = getAnimationStartingX(targetedChar);
       if(down)
       {
            view.startBattleDownAnimation(startX, attackAnimationStartY,
animationImage);

       }
       else
       {
            view.startBattleUpAnimation(startX,
BackGroundPane.B_HEIGHT,animationImage);
       }
    }
```

A more interesting aspect of the controller is the setting of each Hero's ability buttons AbilityListeners. Below we can see the controller setting the AbilityListener of each button displayed for the current hero:

```
    /**********************************************************************/
    //========================= Action Listeners =========================

    /**
     * Add the action listeners to the ability buttons of the current acting hero
     * @param addHero the hero whose ability buttons to customize
     */
    public void addActionListeners(Hero addHero)
    {
       if(model.isFirstGo())
       {
              for(int i = 0; i < 4; i ++)
              {
                     view.addAbilityListener(i, model.new
AbilityListener(addHero));
              }
             view.addAbilityListener(4, model.new ItemListener());
       }
       else
       {
              for(int i = 0; i < 4; i ++)
              {
                     view.setAbilityListener(i, model.new
AbilityListener(addHero));
              }
       }
    }
```

The rest of the controller is more of the same; signal sending to the view and adding action listeners to the buttons. Next up….

Model and Threads

The battle model handles all data management within the MVC. It updates both the human and AI parties health / statuses / ability points and continually checks to determine if the battle is still raging.

The majority of the work within the BattleModel is done within the BattleState **thread** and the AbilityListeners defined within the Model.

First lets look at the **BattleState thread**. The battle state thread:

* 1) Determines whose turn it is, then passing control to them
* 2) Updating all hero statistics on / before / during actions.

* 3) Triggering animations and pausing the state until they are finished.
* 4) Sending many signals to the controller to update the view when actions are taken.
* 5) Ends when one party is completely composed of dead characters.

If it is the player's turn the BattleState thread pauses until a successful action completion.

```
try {
     pauseThread();
     } catch (InterruptedException e) {
           e.printStackTrace();
     }
// Waits until thread is unpaused by an Ability or Item listener
     System.out.println("Please input a command");
     checkForPause();
```

The successful action completion is signaled in an AbilityListener or ItemListener. From the AbilityListener inner class:

```
try{
     abilityToUse.useBattleCommand(watchingHero, controller);
     gameState.resumeThread();
     controller.signalDisplayAbilityUsed(watchingHero,
abilityToUse);
     controller.signalUpdateStatuses();
     updateOnSuccessfulEvent(watchingHero);
}
          catch(BattleModelException battleException)
{
     battleException.sendControllerErrorSignal();
}
```

If the useBattleCommand throws an exception it will be caught and the BattleState thread will NOT RESUME. This allows the player to make "checked" mistakes.

Once this is done the BattleThread continues to delegate whose turn it is and to provide time for animations to complete:

```
if(!firstGo)
 {
       try {
            BattleState.sleep(DISPLAY_SLEEP_TIME);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
       controller.signalRemoveAbilityUsed(currentHero);
 }
```

In the event of the AI's turn the BattleState also sleeps to give their animations time to display. Otherwise their turn would happen immediately and nothing would display.

```
if(currentHero.getControlledBy().equals("AI")
{
                    try {
                    BattleState.sleep(DISPLAY_SLEEP_TIME);
                    } catch (InterruptedException e) {
                    e.printStackTrace();
}
```

Event-Driven Programming

The Dungeon acts on event driven programming where the event is a keypress instead of onscreen button. All movement and outcomes within the dungeon view are controlled by this event. As such without any key input nothing can occur.

Nothing happens in the BattleModel without event-driven programming. If no button is pressed then the BattleModel will never resume. On each button pressed a series of actions is executed.

JUnit Testing

There are numerous tests throughout this project. All JUnit tests can be found within the JUnit package. Here are a few:

```
/**
 * Poison Stacking Test
 */
@Test
public void testPoisonStack()
{
	Soldier hero = new Soldier(HumanPlayer.CONTROLLED);
	Status poison = new OffensiveStatusesPerTurn.Poison(-2, 2, 2);
	poison.addStatus(hero);
	// Sleep so time stamp is different
	try {
		Thread.sleep(10);
	} catch (InterruptedException e) {
		e.printStackTrace();
	}
	poison.addStatus(hero);
	hero.updateStatuses();
	int actual = hero.getStatuses().size();
	int expected = 2;
	assertEquals(expected, actual);
}
```

```
	/**
	 * LowerDefense test to check if armor ISN'T lowered a second time on tick
	 */
	@Test
	public void testLowerDefenseTick()
	{
		Soldier hero = new Soldier(HumanPlayer.CONTROLLED);
		Status lowerDefense = new
OffensiveStatusesNotPerTurn.LowerDefense(-3, 2, 2);
		lowerDefense.addStatus(hero);
		// Get defense rating before applied
		int expected = hero.getDefenseRating() +
lowerDefense.getEffectStrength();
		hero.updateStatuses();
		hero.updateStatuses();
		// Get after application
		int actual = hero.getDefenseRating();
		assertEquals(expected, actual);
	}
	/**
	 * CrowdControlStatus change tick test. Should still be true.
	 */
	@Test
	public void testCrowdControlStatusTick()
	{
		Soldier hero = new Soldier(HumanPlayer.CONTROLLED);
		Status stunned = new CrowdControlStatus("CC",3, 2, 2);
		stunned.addStatus(hero);
		boolean actual = hero.updateStatuses();
		actual = hero.updateStatuses();
		// Get after application
		boolean expected = true;
		assertEquals(expected, actual);
	}

	/**
	 * Check if Heal ability throws MaxStat.
	 * @throws MaximumStatException
	 */
	@Test(expected = MaximumStatException.class)
	public void testHealAbilityThrows() throws BattleModelException
	{
		Paladin hero = new Paladin(HumanPlayer.CONTROLLED);
		DefensiveAbility ability = (DefensiveAbility)
Ability.getAbility(hero, "Heal");
		Hero target = null;
		ability.useBattleCommand(hero, target);
	}

	    /**
	 * PoisonBlade damage subtraction on application. .
```

```
         * @throws MaximumStatException
         */
        @Test
        public void testPoisonBladeDamage() throws BattleModelException
        {
                Assassin hero = new Assassin(HumanPlayer.CONTROLLED);
                Assassin target = new Assassin(HumanPlayer.CONTROLLED);
                OffensiveAbility offensiveAbility = (OffensiveAbility)
Ability.getAbility(hero, "PoisonBlade");
                offensiveAbility.useBattleCommand(hero, target);
                int actual = target.getHealth();
                int expected = target.getMaxHealth() -
OffensiveAbility.evaluateDamage(target, offensiveAbility);
                assertEquals(expected,actual);
        }
```