Chain Of Command

Advanced Java Programming

Design Patterns

Outline

- The Software Engineering Problem
- The Current System
- The GoF description of the CoR pattern
- Implementation of the CoR pattern
- Advantages and Disadvantages of the CoR Pattern
- Versioning & File Structure
- Extended Version (Template Pattern)
- Summary

The Software Engineering Problem

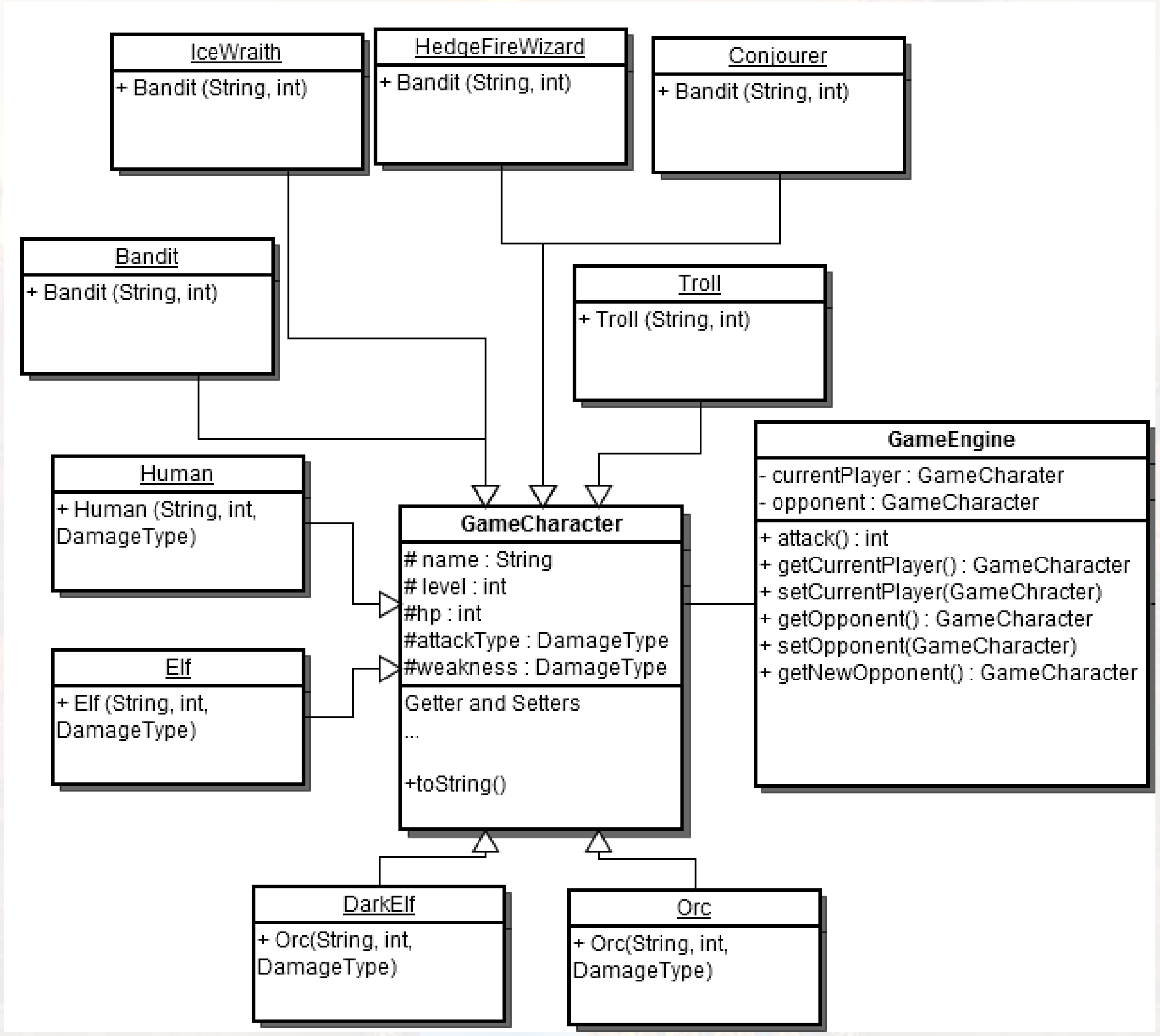
Al K. Holic lead developer of *Sand Storm's: World Of Swordcraft* is looking for a programming solution to help with calculating attack damage to the player or NPC . The development team so far has come up with a solution to calculate the amount of damage based on the level of the enemy; However they need a way to calculate the damage of hits with different effects. These are:

- Mêlée Damage
- Ranged Damage
- Nature Damage (Poisons)
- Fire Damage
- Frost Damage
- Shock Damage
- Shadow Damage

The Software Engineering Problem

- Each Damage type will need to be dealt with differently depending on the class of the player. ie.:
  - A Troll has a weakness to fire.
  - A Rogue player would have a weakness to ranged attacks

The Current System


IceWraith
+ Bandit (String, int)
HedgeFireWizard
+ Bandit (String, int)
Conjourer
+ Bandit (String, int)
Bandit
+ Bandit (String, int)
Troll
+ Troll (String, int)
Human
+ Human (String, int, DamageType)
Elf
+ Elf (String, int, DamageType)
GameCharacter
# name : String
# level : int
#hp : int
#attackType : DamageType
#weakness : DamageType
Getter and Setters
...
+toString()
GameEngine
- currentPlayer : GameCharater
- opponent : GameCharacter
+ attack() : int
+ getCurrentPlayer() : GameCharacter
+ setCurrentPlayer(GameChracter)
+ getOpponent() : GameCharacter
+ setOpponent(GameCharacter)
+ getNewOpponent() : GameCharacter
DarkElf
+ Orc(String, int, DamageType)
Orc
+ Orc(String, int, DamageType)

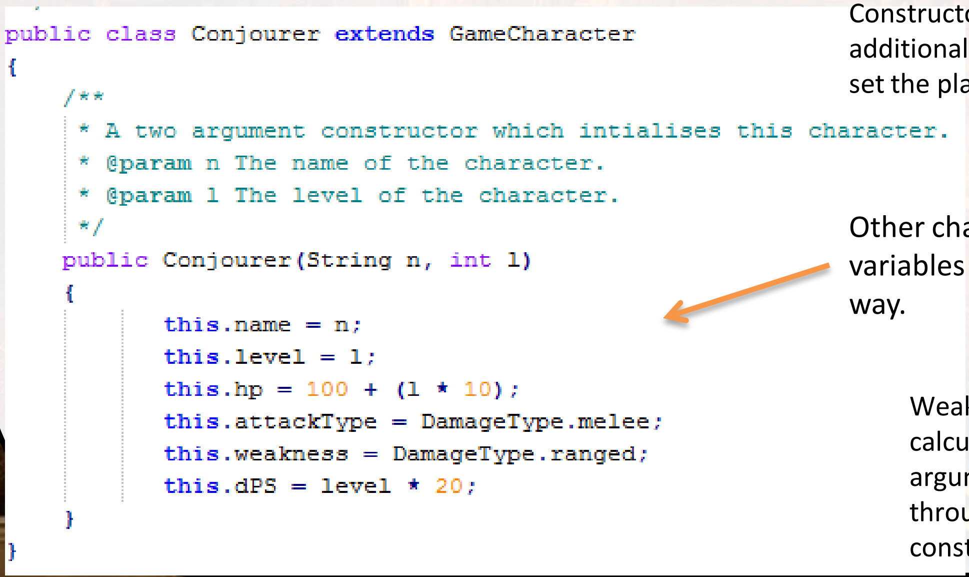
The Current System – GameChracter

- The GameChracter class is abstract. It represents one character which can be both a player and a non playable character (NPC).
- Each character has a name, a level, Health points (HP), an attack type and a weakness.
- The level of a Character determines that characters HP and its damage per second (DPS).
- AttackType and Weakness are enumerations and are pre set.
- toString() returns the character information.

The Current System – Charcters

- The player will be able to choose a race (Human, Elf, Orc and Dark Elf) respective of all the classes that extend the GameCharacter class.
- There is currently also two NPCs – Bandit & Troll
- Player and NPC objects contrast slightly due to the ability a player can select their class. Which in turn effects the characters attackType and Weakness.

NPC

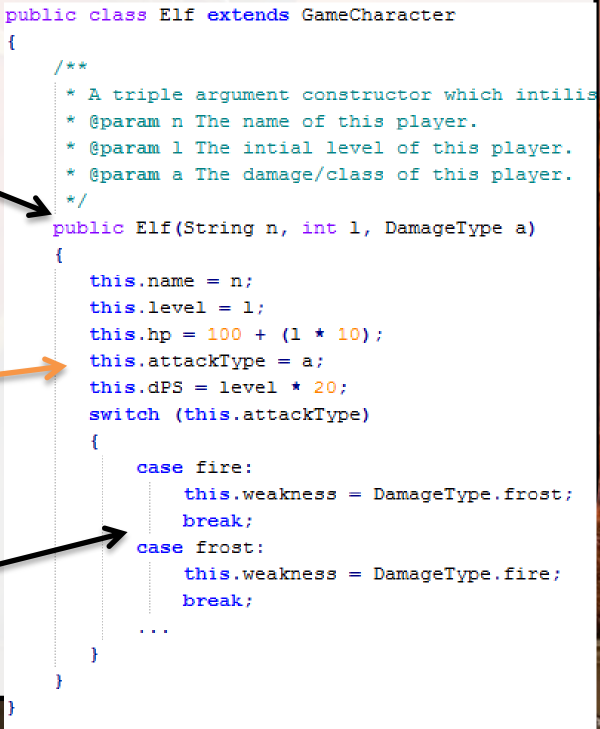


```
public class Conjourerer extends GameCharacter
{
    /**
     * A two argument constructor which intialises this character.
     * @param n The name of the character.
     * @param l The level of the character.
     */
    public Conjourer(String n, int l)
    {
        this.name = n;
        this.level = l;
        this.hp = 100 + (l * 10);
        this.attackType = DamageType.melee;
        this.weakness = DamageType.ranged;
        this.dPS = level * 20;
    }
}
```

Constructor takes additional argument to set the players 'class'.

Other character variables set the same way.

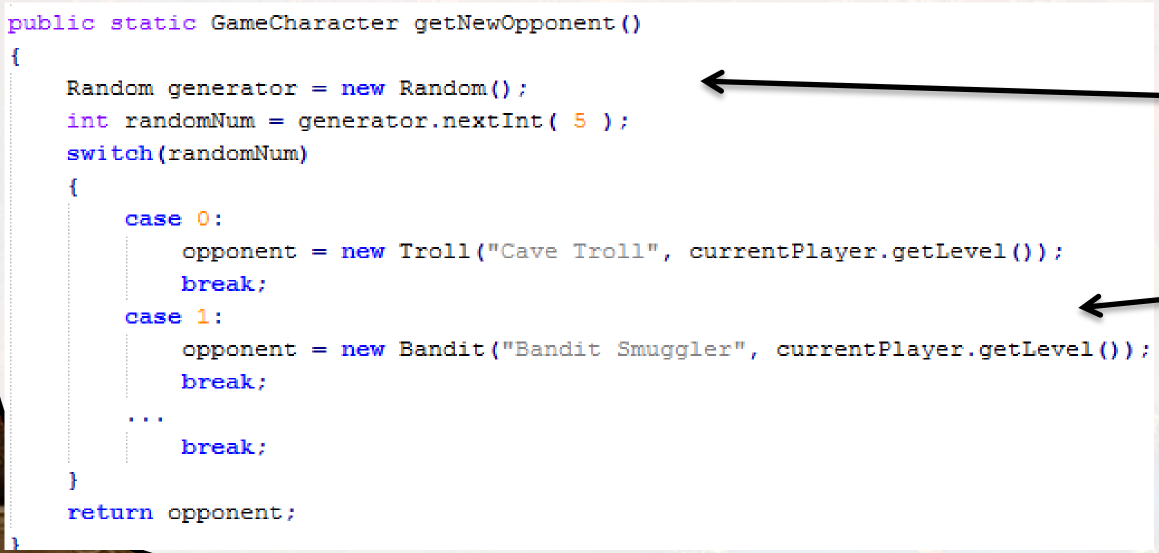
Weakness then calculated from the argument passed through the constructor.



```
public class Elf extends GameCharacter
{
    /**
     * A triple argument constructor which intilis
     * @param n The name of this player.
     * @param l The intial level of this player.
     * @param a The damage/class of this player.
     */
    public Elf(String n, int l, DamageType a)
    {
        this.name = n;
        this.level = l;
        this.hp = 100 + (l * 10);
        this.attackType = a;
        this.dPS = level * 20;
        switch (this.attackType)
        {
            case fire:
                this.weakness = DamageType.frost;
                break;
            case frost:
                this.weakness = DamageType.fire;
                break;
            ...
        }
    }
}
```

The Current System – GameEngine

- The GameEngine Class will be used to implement all the game time calculations.
- It will hold the current player and opponent.
- It will create a new opponent based on the players level – getOpponent()
- It will also perform the calculations when a player and opponent are pitched against each other – attack()



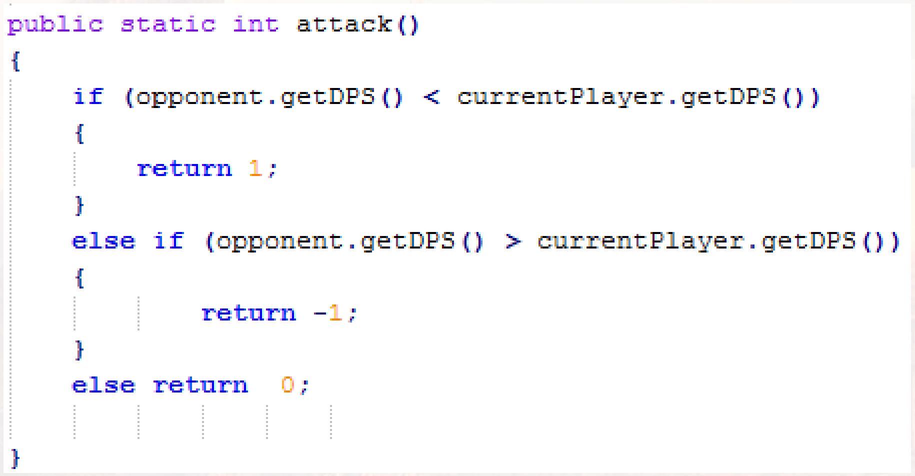
```
public static GameCharacter getNewOpponent()
{
    Random generator = new Random();
    int randomNum = generator.nextInt( 5 );
    switch(randomNum)
    {
        case 0:
            opponent = new Troll("Cave Troll", currentPlayer.getLevel());
            break;
        case 1:
            opponent = new Bandit("Bandit Smuggler", currentPlayer.getLevel());
            break;
        ...
            break;
    }
    return opponent;
}
```

The getOpponent() method creates a random integer.

Which is then used to create a new opponent levelled at the same level of the player.

The Current System – GameEngine

attack() Method



```
public static int attack()
{
    if (opponent.getDPS() < currentPlayer.getDPS())
    {
        return 1;
    }
    else if (opponent.getDPS() > currentPlayer.getDPS())
    {
            return -1;
    }
    else return  0;

}
```

Returns an int which is in relation to the outcome of the current player playing against an NPC.

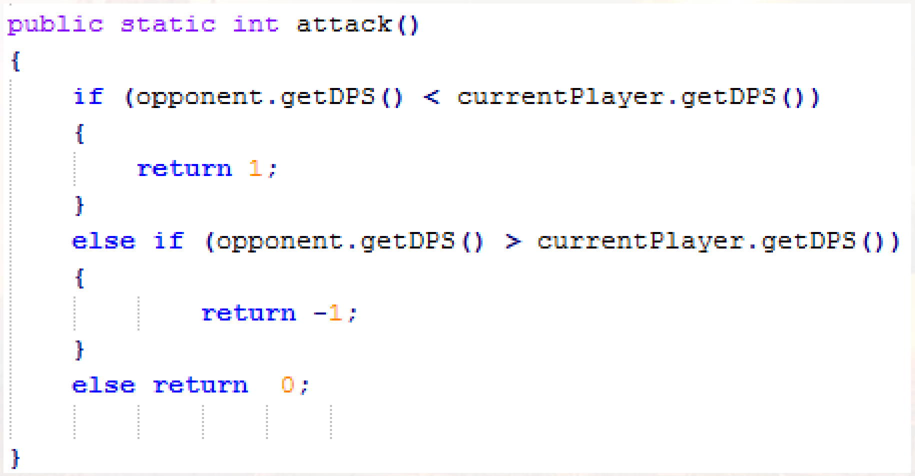
1- Win

0 – Draw

-1 - Loss

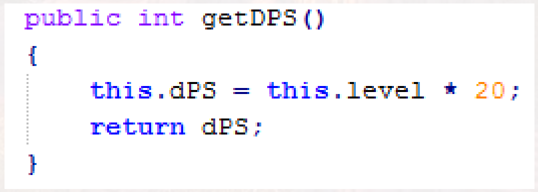
The Current System – The **Problem**

GameEngine.attack() Method



```
public static int attack()
{
    if (opponent.getDPS() < currentPlayer.getDPS())
    {
        return 1;
    }
    else if (opponent.getDPS() > currentPlayer.getDPS())
    {
            return -1;
    }
    else return  0;
}
```

GameCharacter.getDPS() Method

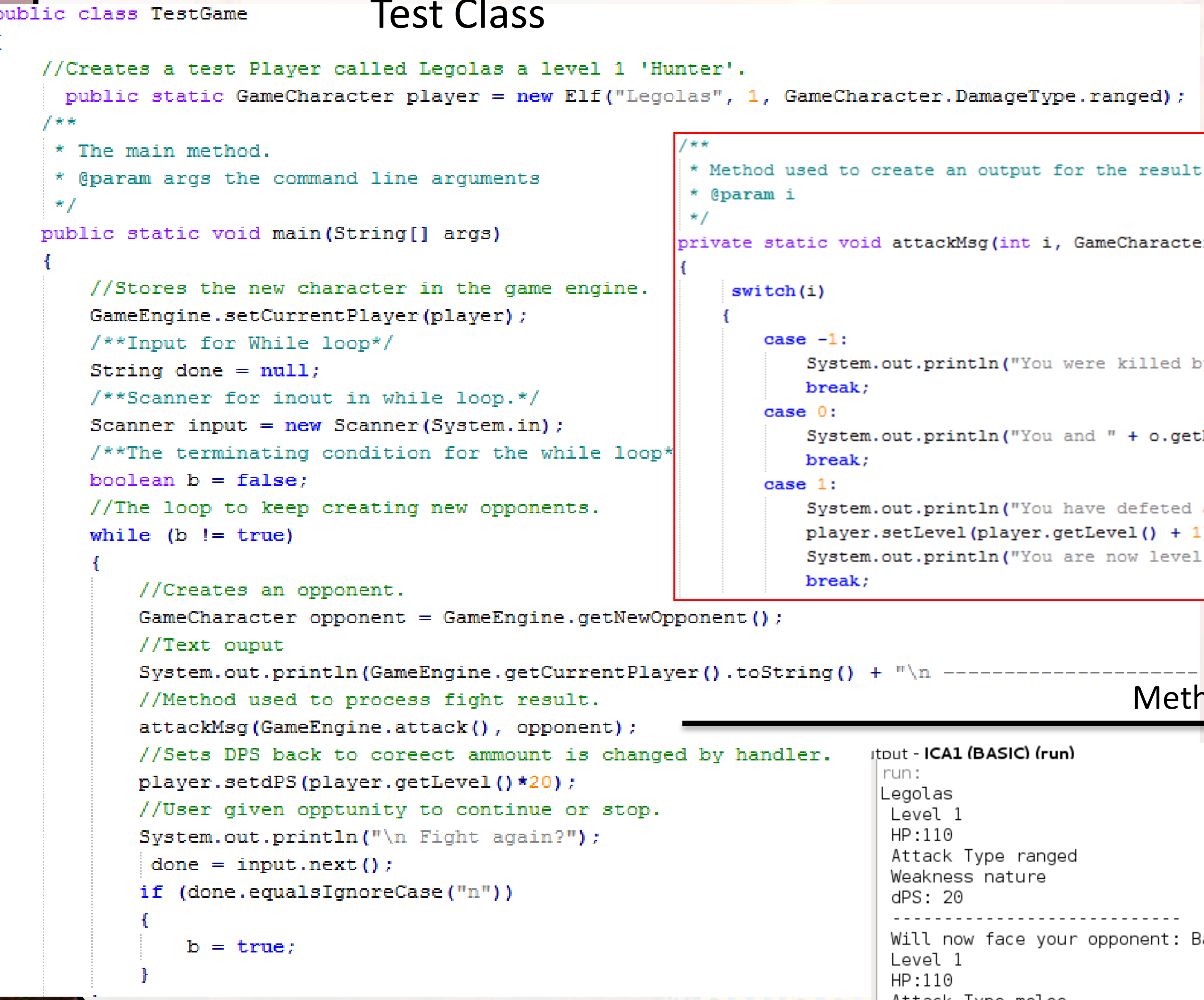


```
public int getDPS()
{
    this.dPS = this.level * 20;
    return dPS;
}
```

- The problem with the system currently lies with how the attack method works.
- It calls the getDPS() method, which works on the level of the character it is called for.
- An opponent is currently always the same level as the player, and so their DPS is always the same. The means every fight results in a draw. Which makes a rubbish Game!

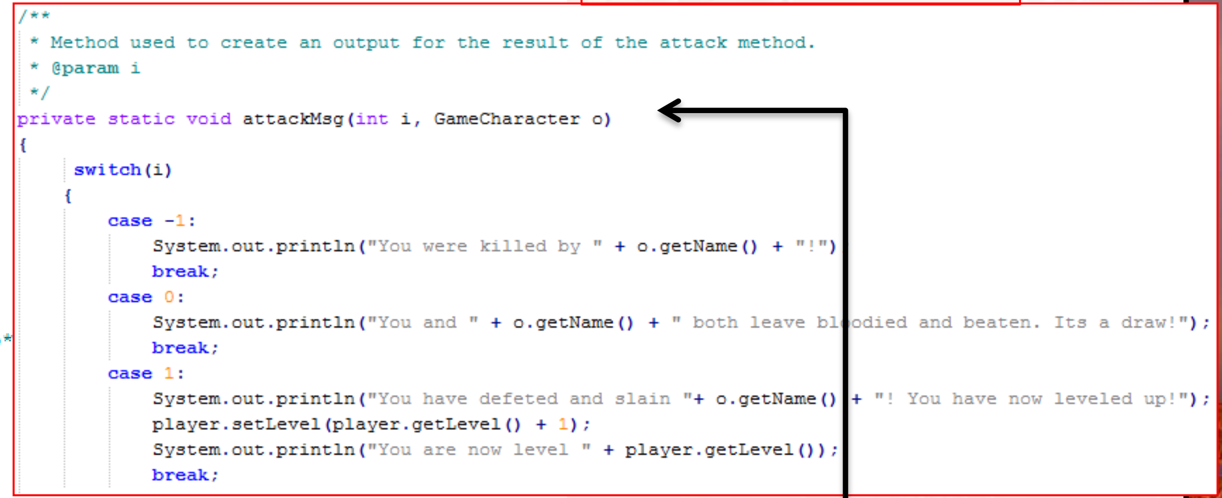
The Current System – The **Problem**

Test Class



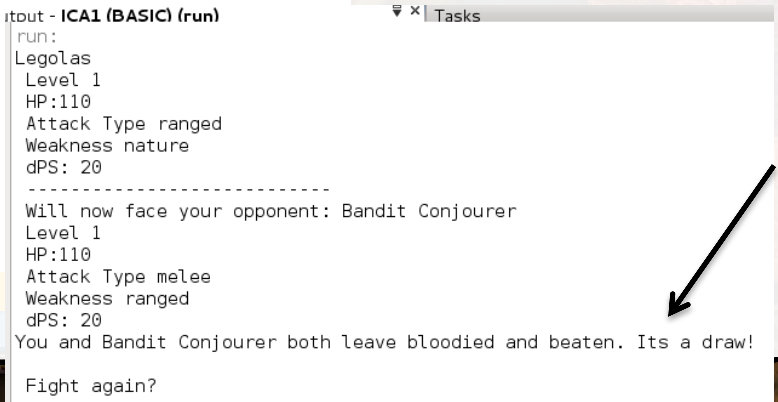
```
public class TestGame
{
  //Creates a test Player called Legolas a level 1 'Hunter'.
    public static GameCharacter player = new Elf("Legolas", 1, GameCharacter.DamageType.ranged);
  /**
   * The main method.
   * @param args the command line arguments
   */
  public static void main(String[] args)
  {
      //Stores the new character in the game engine.
      GameEngine.setCurrentPlayer(player);
      /**Input for While loop*/
      String done = null;
      /**Scanner for inout in while loop.*/
      Scanner input = new Scanner(System.in);
      /**The terminating condition for the while loop*
      boolean b = false;
      //The loop to keep creating new opponents.
      while (b != true)
      {
          //Creates an opponent.
          GameCharacter opponent = GameEngine.getNewOpponent();
          //Text ouput
          System.out.println(GameEngine.getCurrentPlayer().toString() + "\n ---------------------
          //Method used to process fight result.
          attackMsg(GameEngine.attack(), opponent);
          //Sets DPS back to coreect ammount is changed by handler.
          player.setdPS(player.getLevel()*20);
          //User given opptunity to continue or stop.
          System.out.println("\n Fight again?");
           done = input.next();
          if (done.equalsIgnoreCase("n"))
          {
              b = true;
          }
```

attackMsg() method



```
/**
 * Method used to create an output for the result of the attack method.
 * @param i
 */
private static void attackMsg(int i, GameCharacter o)
{
    switch(i)
    {
        case -1:
            System.out.println("You were killed by " + o.getName() + "!")
            break;
        case 0:
            System.out.println("You and " + o.getName() + " both leave bloodied and beaten. Its a draw!");
            break;
        case 1:
            System.out.println("You have defeted and slain "+ o.getName() + "! You have now leveled up!");
            player.setLevel(player.getLevel() + 1);
            System.out.println("You are now level " + player.getLevel());
            break;
```

Method called



```
run:
Legolas
 Level 1
 HP:110
 Attack Type ranged
 Weakness nature
 dPS: 20
 ----------------------------
 Will now face your opponent: Bandit Conjourer
 Level 1
 HP:110
 Attack Type melee
 Weakness ranged
 dPS: 20
You and Bandit Conjourer both leave bloodied and beaten. Its a draw!

 Fight again?
```

Draw!

The Current System – The **Problem**

- As was previously shown; despite the *Bandit Conjurer* having a weakness to ranged attacks our *Elf Hunter* still drew with the Bandit.
- This is because the attack types are not handled in relation to the weaknesses of the characters.
- The fix? ...

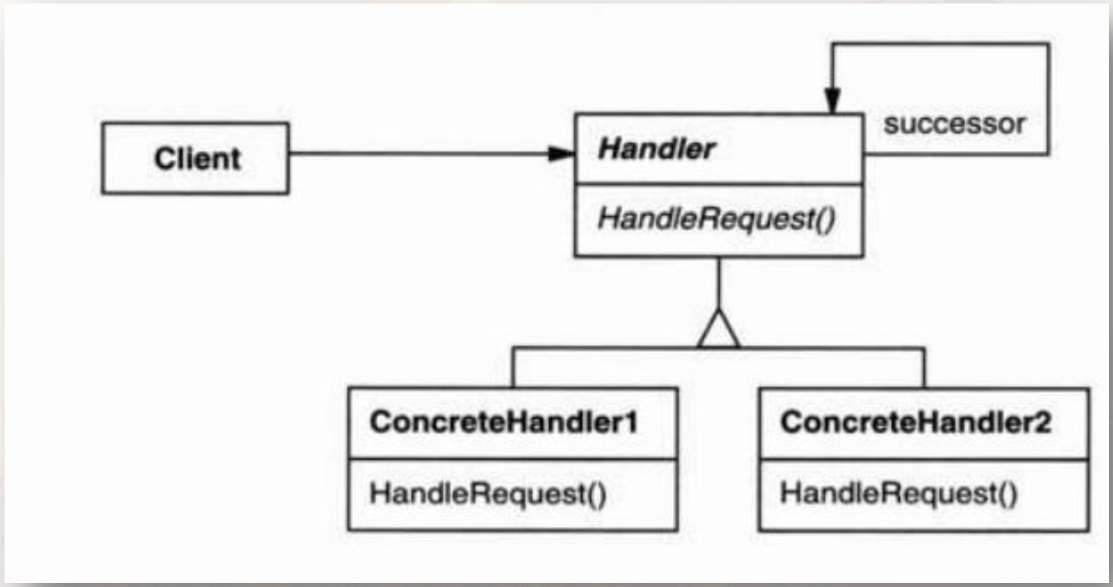
Chain of Responsibility (GoF) – The **Fix**

" Avoid[s] *coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. There is a potentially variable number of "handler" objects and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings. "*

Design Patterns: Elements of Reusable Object-Oriented Software , (Gamma, Helm, Johnson, Vlissides, 1994, pg.223)

- The description given by the gang of four states that the Chain of Responsibility pattern is used to handle requests through a 'chain' of 'handler' objects. The request will be passed through each handler until such a time that a handler can deal with the request.

Chain of Responsibility (GoF)



Design Patterns: Elements of Reusable Object-Oriented Software , (Gamma, Helm, Johnson, Vlissides, 1994, pg.223)

The client which creates a notification of an even happening pushes it out to the Handlers. The concrete handlers implement/extend the Handler. Each concrete handler will contain code to deal with a certain notification. The successor is then passed back to the handler, with the code from the concrete handler being implemented.

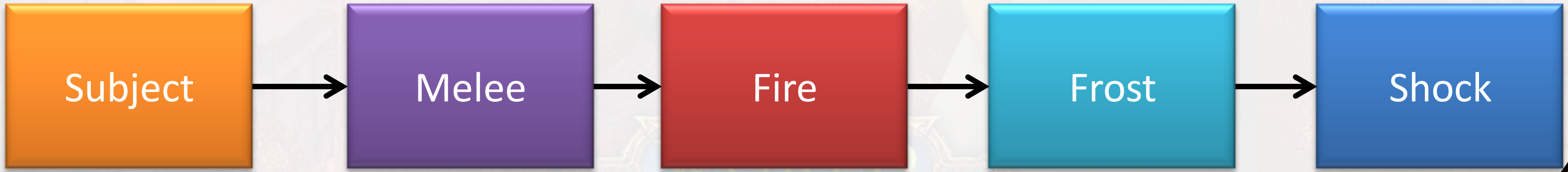
Notification

$\longrightarrow$

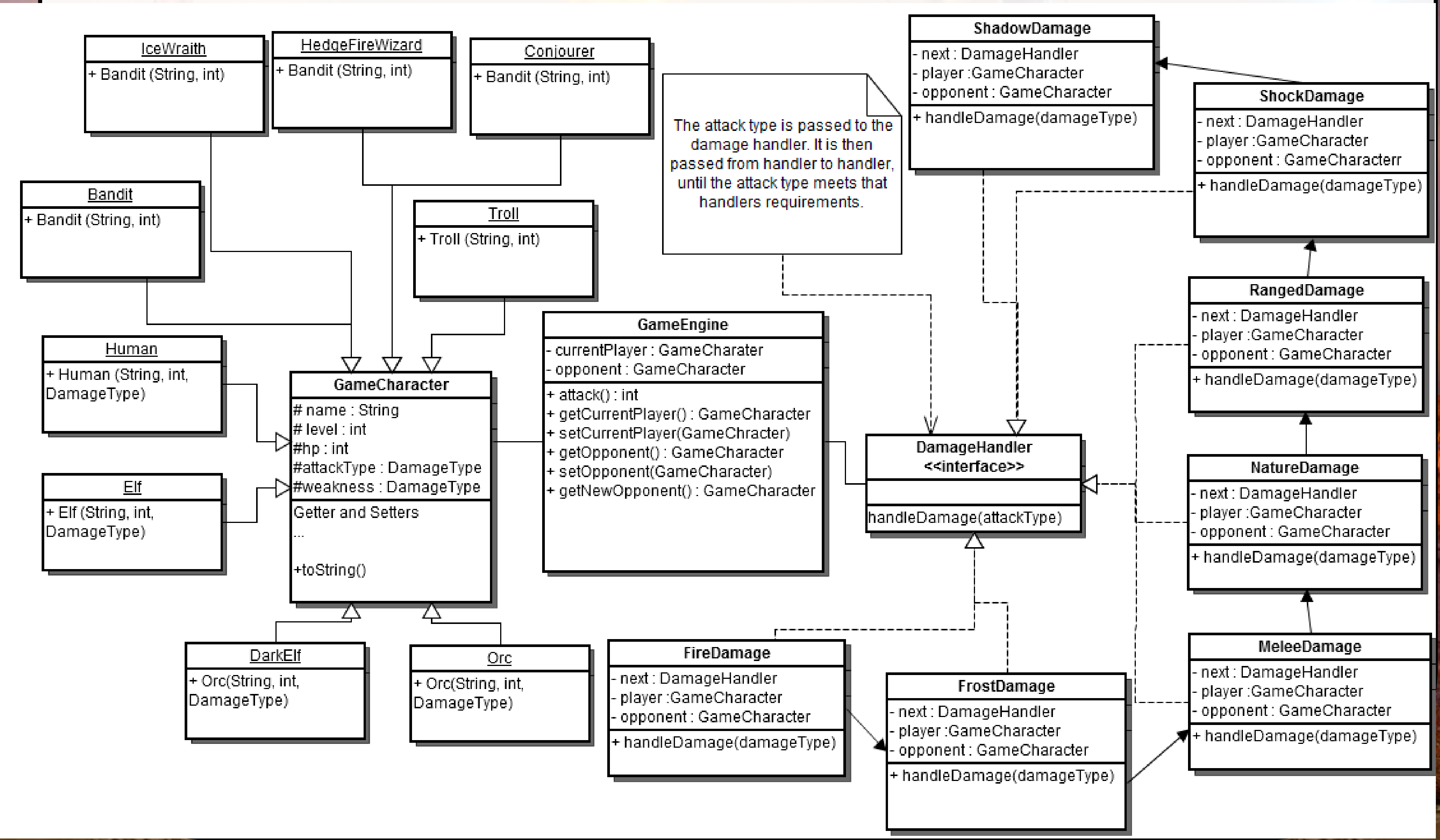
Implementation

To fix the constant draw outcome with the current system, I will use the Chain of Responsibility (CoR) to make best use of the attackTypes and weakness'. By implementing damage handlers, when an attack happens an attack notification will be pushed out by the method. It will pass through the damage handlers. The handler for that attack type will then re calculate the DPS by the combatants based on the weakness' of the characters.

Here is a simple example of how a notification will be passed through handlers:



Implementation



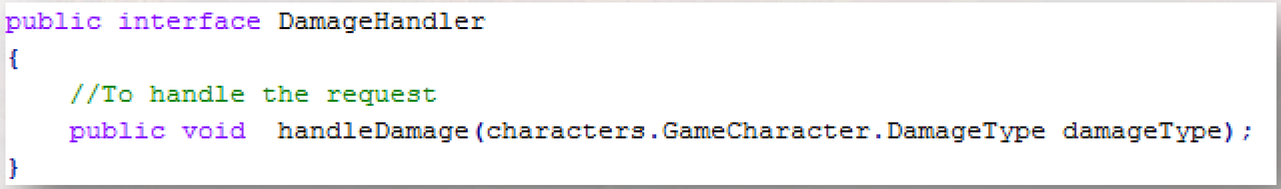
The attack type is passed to the damage handler. It is then passed from handler to handler, until the attack type meets that handlers requirements.

Implementation

- I have created a new package and filled it with a handler interface and concrete handlers to implement this interface.
- The package contains these new classes:
  - *DamageHandler* – The interface.
  - *FireHandler, FrostHandler, MeleeDamage, NatureDamage, RangedDamage, ShockDamage & ShadowDamage* – Concrete Handlers

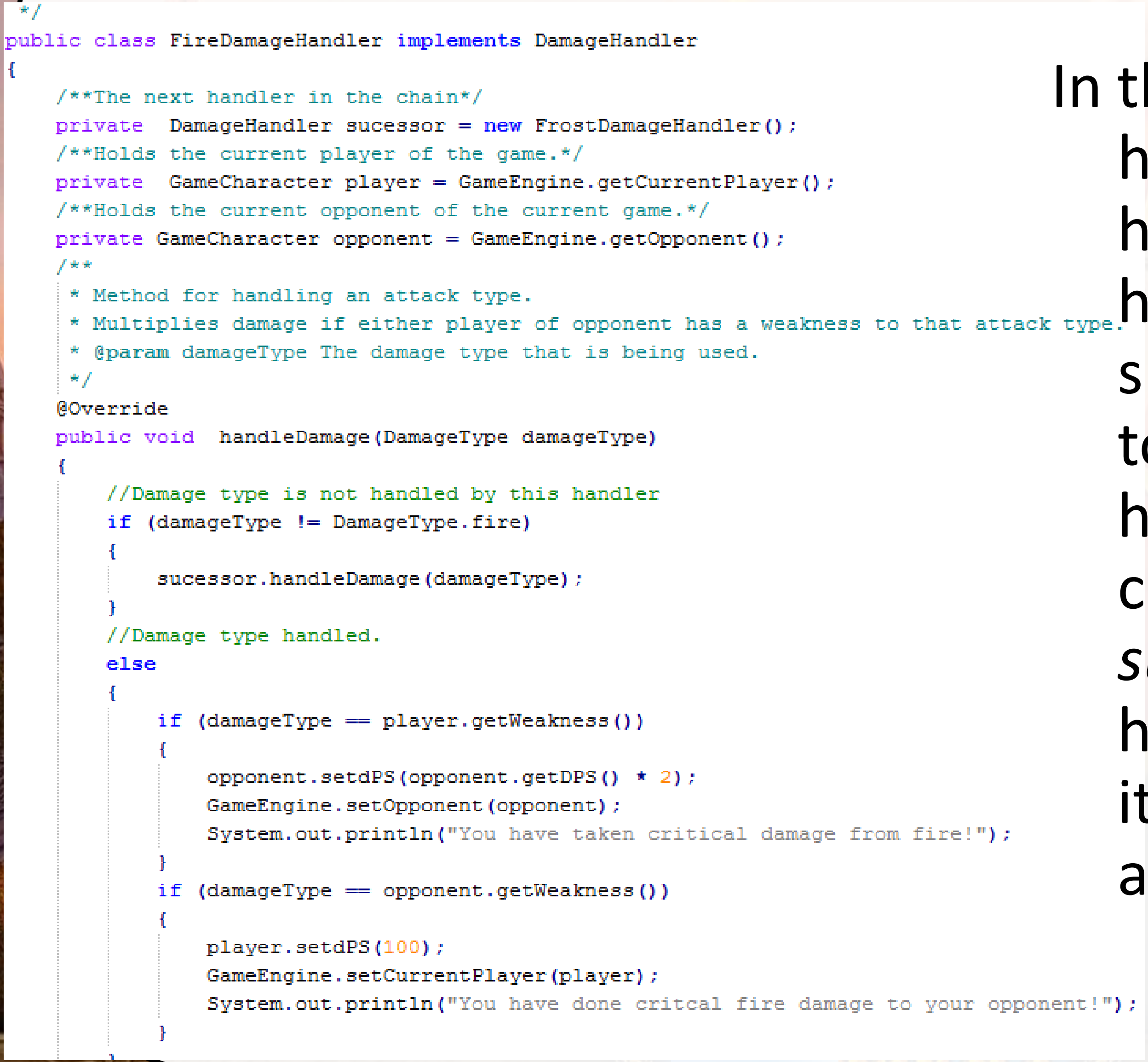
DamageHandler

- DamageHandler is a simple interface that sets the foundation for all the handlers implementing it.
- *handleDamage()* is the method which takes the attack type and performs the handle procedure.



```
public interface DamageHandler
{
    //To handle the request
    public void  handleDamage(characters.GameCharacter.DamageType damageType);
}
```

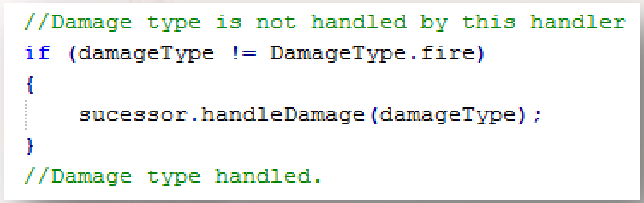
Concrete Handlers



```
 */
public class FireDamageHandler implements DamageHandler
{
    /**The next handler in the chain*/
    private  DamageHandler sucessor = new FrostDamageHandler();
    /**Holds the current player of the game.*/
    private  GameCharacter player = GameEngine.getCurrentPlayer();
    /**Holds the current opponent of the current game.*/
    private GameCharacter opponent = GameEngine.getOpponent();
    /**
     * Method for handling an attack type.
     * Multiplies damage if either player of opponent has a weakness to that attack type.
     * @param damageType The damage type that is being used.
     */
    @Override
    public void  handleDamage(DamageType damageType)
    {
        //Damage type is not handled by this handler
        if (damageType != DamageType.fire)
        {
            sucessor.handleDamage(damageType);
        }
        //Damage type handled.
        else
        {
            if (damageType == player.getWeakness())
            {
                opponent.setdPS(opponent.getDPS() * 2);
                GameEngine.setOpponent(opponent);
                System.out.println("You have taken critical damage from fire!");
            }
            if (damageType == opponent.getWeakness())
            {
                player.setdPS(100);
                GameEngine.setCurrentPlayer(player);
                System.out.println("You have done critcal fire damage to your opponent!");
            }
```

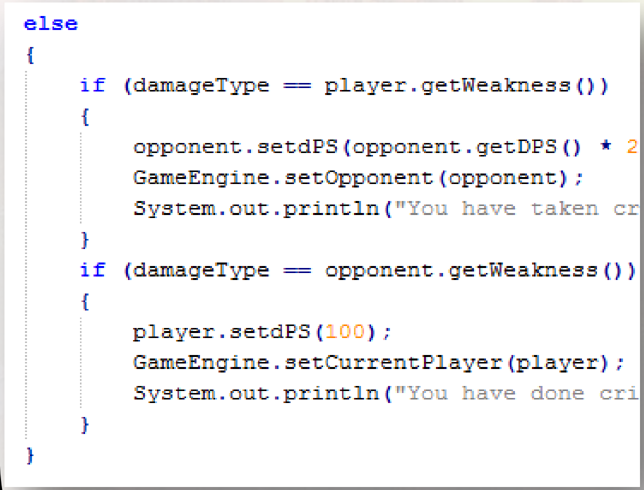
In the CoR pattern I have implemented, I have made it so each handler knows its sucessor as oppose to creating a chain of handlers in a test class. This is held by *successor.* Each handler also creates itself its own player and opponent.

Concrete Handlers – *handleDamage()*



```
//Damage type is not handled by this handler
if (damageType != DamageType.fire)
{
    sucessor.handleDamage(damageType);
}
//Damage type handled.
```

The first if statement simply says "is this anything to do with with me? No? The other guy can have this then!". So the damage type will be passed to the next handler in the chain.



```
else
{
    if (damageType == player.getWeakness())
    {
        opponent.setdPS(opponent.getDPS() * 2
        GameEngine.setOpponent(opponent);
        System.out.println("You have taken cr
    }
    if (damageType == opponent.getWeakness())
    {
        player.setdPS(100);
        GameEngine.setCurrentPlayer(player);
        System.out.println("You have done cri
    }
}
```

Should this handler be able to handle the damage type it then processes it. It doubles the character with the advantages damage and outputs an appropriate message. For changes made to the local characters, the Game Engine must also be updated with the changed character. Which is done using *GameEngine.setCurrentPlayer()*.

Advantages of CoR

- The CoR pattern is classed as a *best practice*. Meaning it consistently achieves superior results, and is much a standard.
- Reduced coupling. Because handler objects are completely separate, and run at different times they have no knowledge of what happens to a request if they cannot handle it. Each handler will only ever know its successor.
  - By having a loose coupling, there is less chance of a change to one object which in turn may affect another object causing a *ripple effect* through the application.
  - Re use if code in a tight coupled application would also be much harder; thus making the application much harder to maintain.
- Reduced workload on the CPU – Handling several objects at the same time means a higher processor usage. On a larger application, the observer pattern could present a problem.

Advantages of CoR

- Flexibility – The chain can be changed at run time. Although this was not implemented in this project, it would have been feasible to skip part of the chain by beginning with a separate handler that dealt with that. By doing so, a message would get straight to the place it need to be.
- By adding an *addHandler()* method to the handler interface, it would have been possible to add more handlers to the chain. As long as the successor is named for the end concrete handler.

Disadvantages of CoR

- The GoF described a disadvantage of the CoR pattern to be “Receipt isn't guaranteed” (pg.226). This arguably isn’t always an occurrence, certainly this is so in this project. Because an enumeration of *damageType* mean’t that no matter the outcome of an attack, a handler would catch it.
  - Incidentally, this disadvantage is not necessarily always a drawback. It could be the solution to avoiding a runtime error. However its usefulness is far outweighed by needing to catch a message.

Disadvantages of CoR

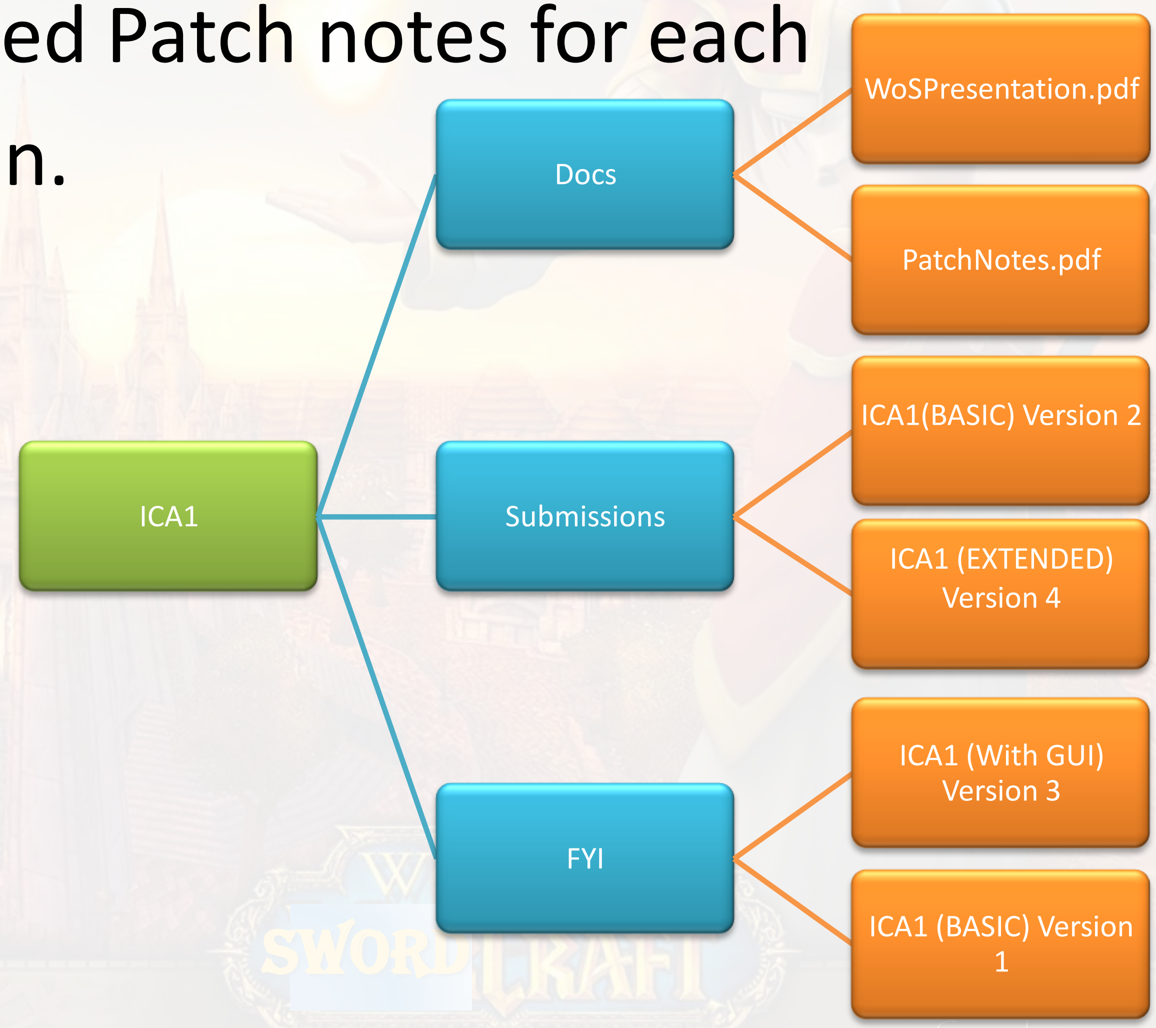
- Time – With the observer pattern, a notification is passed out to every object at the **same** time. Where as with a chain if the handler for a certain request is towards the end of the chain, time is wasted.
  - Although for a small chain the time is miniscule, if the chain was made for an application of thousands of handlers a wait time for a CPU to process each handler could be noticed.

Versioning

- I have submitted four versions of my project.
  - Version 1 is the original problem
  - Version 2 is a simple console game for my BASIC **submission**
  - Version 3 implements a GUI but works on the same principle. And is for your interest.
  - Version 4 implements the Template Pattern which will be my EXTENDED **submission**.
- Version 1 and 2 have been evaluated in this presentation.

Versioning – Folder Structure

- Sandstorm Entertainment Ltd. have also released Patch notes for each version.



ICA Extension

"We like what you have done with our game! Its come leaps and bounds... Just one thing, its boring! No one will want to play a one click game. No! We want out players to love this game! So we need you to allow players to have more than one type of attack. Our main competitors over at *Bethesda* added it to their latest game *Skyrim*, but we know our game is better. Get too it!"

Al K Holic Sandstorm Lead Developer

ICA Extension

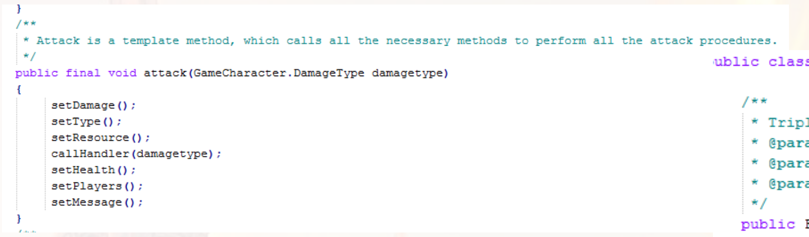
Al K Holic wants a way to have two types of attacks from users. For example, one attack is shadow, but the other is fire. **But** to make the game harder, he wants players who choose both of the same attack to be better at it.

The following slides discuss the ‘in’s and outs’ of Version 4.

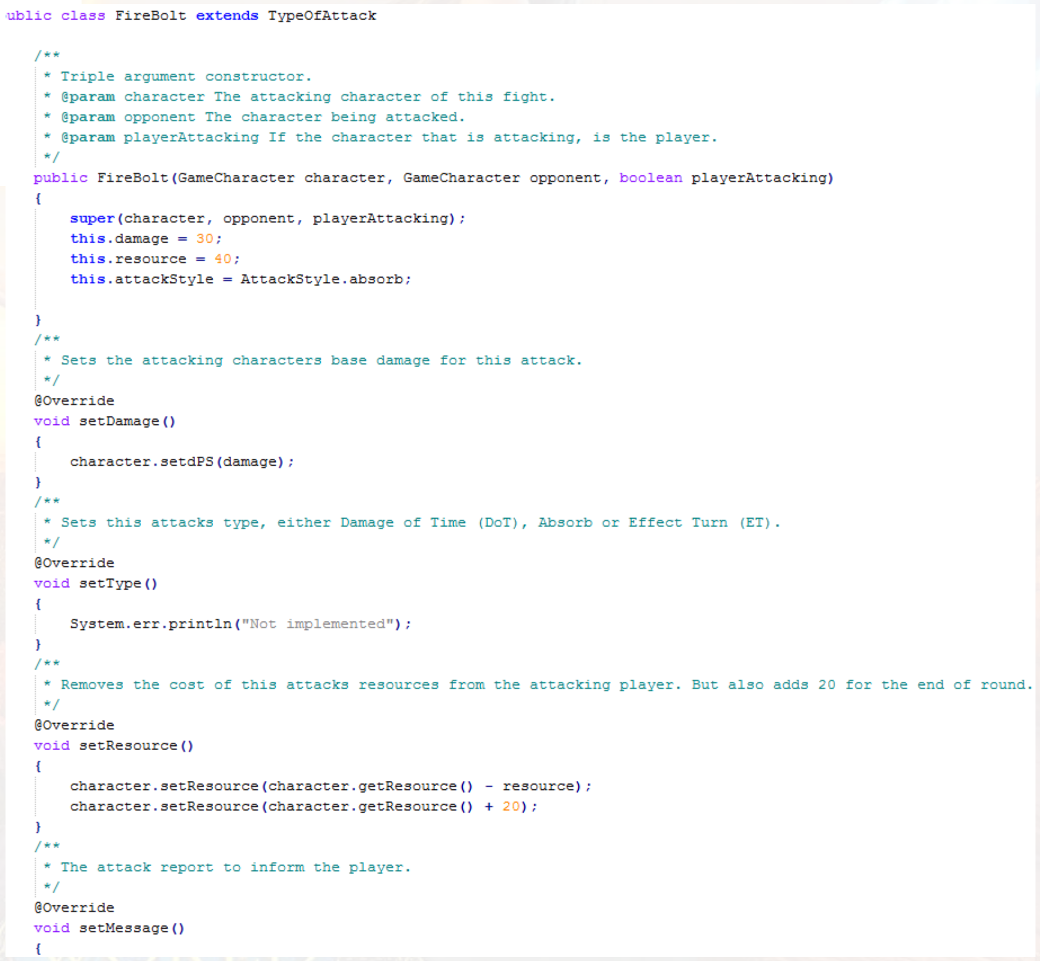
ICA Extension

To have more than one attack the easiest way to implement it us to use the Template Pattern. I have done this my making a package *typesOfAttack* which contains an interface *TypeOfAttack* and classes of each type off attack to implement it.

Template Pattern



```
}
/**
 * Attack is a template method, which calls all the necessary methods to perform all the attack procedures.
 */
public final void attack(GameCharacter.DamageType damagetype)
{
        setDamage();
        setType();
        setResource();
        callHandler(damagetype);
        setHealth();
        setPlayers();
        setMessage();
}
```



```
ublic class FireBolt extends TypeOfAttack

    /**
     * Triple argument constructor.
     * @param character The attacking character of this fight.
     * @param opponent The character being attacked.
     * @param playerAttacking If the character that is attacking, is the player.
     */
    public FireBolt(GameCharacter character, GameCharacter opponent, boolean playerAttacking)
    {
        super(character, opponent, playerAttacking);
        this.damage = 30;
        this.resource = 40;
        this.attackStyle = AttackStyle.absorb;

    }
    /**
     * Sets the attacking characters base damage for this attack.
     */
    @Override
    void setDamage()
    {
        character.setdPS(damage);
    }
    /**
     * Sets this attacks type, either Damage of Time (DoT), Absorb or Effect Turn (ET).
     */
    @Override
    void setType()
    {
        System.err.println("Not implemented");
    }
    /**
     * Removes the cost of this attacks resources from the attacking player. But also adds 20 for the end of round.
     */
    @Override
    void setResource()
    {
        character.setResource(character.getResource() - resource);
        character.setResource(character.getResource() + 20);
    }
    /**
     * The attack report to inform the player.
     */
    @Override
    void setMessage()
    {
```

Above is the abstract class which is the template of each type of attack. And on the right, FireBolt – an attack type that implements typeOfAttack.

It overrides certain abstact classes in that are in type of attack.

Summary – The CoR Pattern

- You should use this pattern when notifications or events are required to be handled in specific ways.
- The decorator pattern could of been another suitable method for the original problem. It could have been possible to wrap each character in an attackType object, and implemented from there. Although the CoR pattern was more suitable. Due to its much easier way to modify.