Milestone 3

shotgunrobot

2D platformer game written in Java.
https://github.com/rustbyte/shotgunrobot

Names	IDs
Felicia Santoro-Petti	6619657
Daniel Caterson	9746277
Amanda Juneau	6338518
Mark Karanfil	1526294
George Valergas	6095836

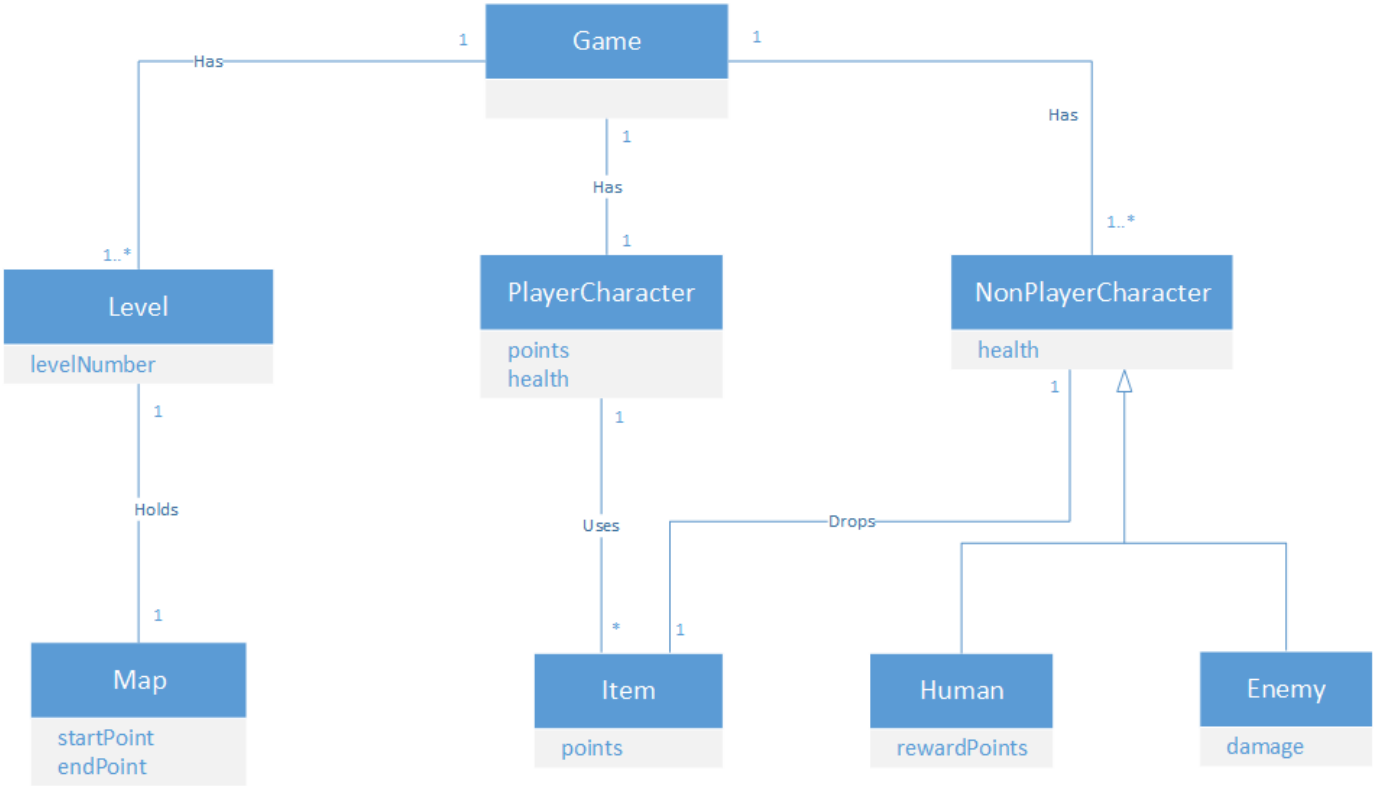
October 29, 2014
We have compiled and assessed this project a believe it to be of a reasonable size for a term project.

Summary of Project

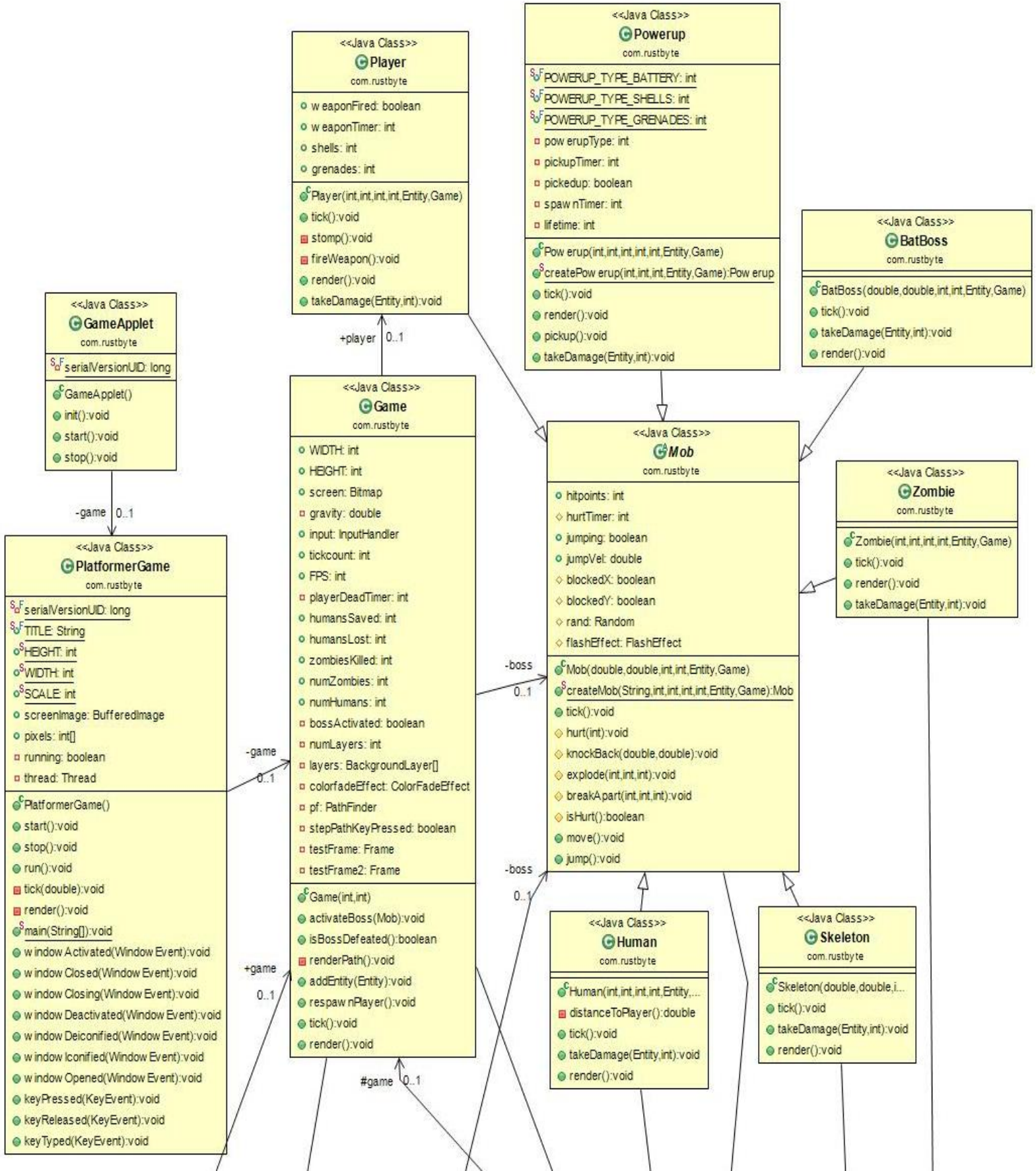
Shotgun Robot is a 2D platformer game written in Java and hosted on Github. The primary contributor goes by the Github username *rustbyte*. He frequently codes live using a service called hitbox. The player assumes the role of a robot with a shotgun, whose purpose is to save humanity from extinction one person at a time, rescuing people as you fight off blood hungry zombies. The game takes place during a zombie apocalypse and does a decent job at ripping all of its concepts from Robocop, Terminator and similar franchises.

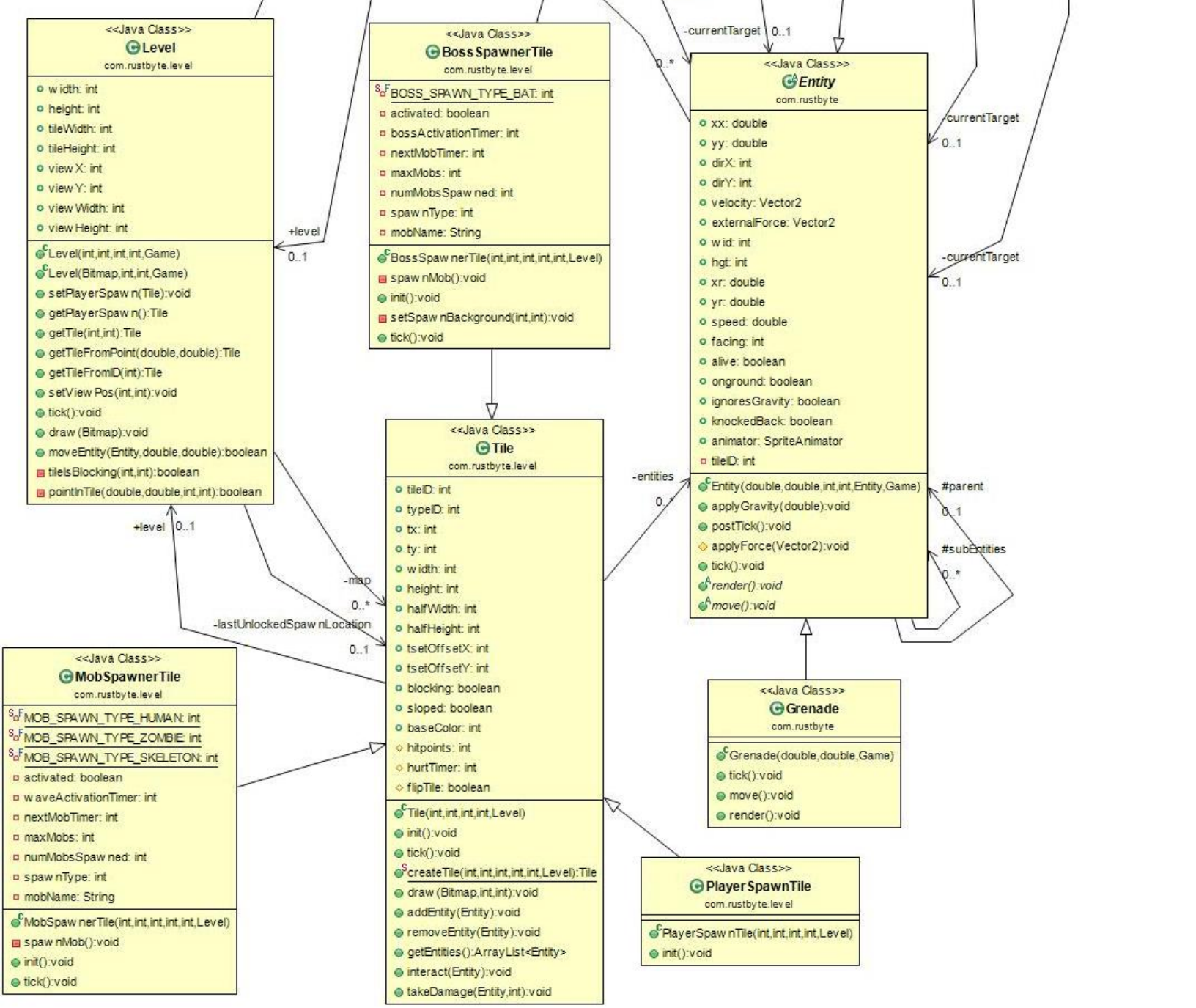
Class Diagram of Actual System

Conceptual Diagram



Actual UML Diagram




<<Java Class>>
Level
com.rustbyte.level
width: int
height: int
tileWidth: int
tileHeight: int
viewX: int
viewY: int
viewWidth: int
viewHeight: int
Level(int,int,int,int,Game)
Level(Bitmap,int,int,Game)
setPlayerSpawn(Tile):void
getPlayerSpawn():Tile
getTile(int,int):Tile
getTileFromPoint(double,double):Tile
getTileFromID(int):Tile
setViewPos(int,int):void
tick():void
draw(Bitmap):void
moveEntity(Entity,double,double):boolean
tileIsBlocking(int,int):boolean
pointInTile(double,double,int,int):boolean
<<Java Class>>
BossSpawnerTile
com.rustbyte.level
BOSS_SPAWN_TYPE_BAT: int
activated: boolean
bossActivationTimer: int
nextMobTimer: int
maxMobs: int
numMobsSpawned: int
spawnType: int
mobName: String
BossSpawnerTile(int,int,int,int,int,int,Level)
spawnMob():void
init():void
setSpawnBackground(int,int):void
tick():void
<<Java Class>>
Entity
com.rustbyte
xx: double
yy: double
dirX: int
dirY: int
velocity: Vector2
externalForce: Vector2
wid: int
hgt: int
xr: double
yr: double
speed: double
facing: int
alive: boolean
onground: boolean
ignoresGravity: boolean
knockedBack: boolean
animator: SpriteAnimator
tileID: int
Entity(double,double,int,int,Entity,Game)
applyGravity(double):void
postTick():void
applyForce(Vector2):void
tick():void
render():void
move():void
<<Java Class>>
Tile
com.rustbyte.level
tileID: int
typeID: int
tx: int
ty: int
width: int
height: int
halfWidth: int
halfHeight: int
tsetOffsetX: int
tsetOffsetY: int
blocking: boolean
sloped: boolean
baseColor: int
hitpoints: int
hurtTimer: int
flipTile: boolean
Tile(int,int,int,int,Level)
init():void
tick():void
createTile(int,int,int,int,int,int,Level):Tile
draw(Bitmap,int,int):void
addEntity(Entity):void
removeEntity(Entity):void
getEntities():ArrayList<Entity>
interact(Entity):void
takeDamage(Entity,int):void
<<Java Class>>
MobSpawnerTile
com.rustbyte.level
MOB_SPAWN_TYPE_HUMAN: int
MOB_SPAWN_TYPE_ZOMBIE: int
MOB_SPAWN_TYPE_SKELETON: int
activated: boolean
waveActivationTimer: int
nextMobTimer: int
maxMobs: int
numMobsSpawned: int
spawnType: int
mobName: String
MobSpawnerTile(int,int,int,int,int,int,Level)
spawnMob():void
init():void
tick():void
<<Java Class>>
Grenade
com.rustbyte
Grenade(double,double,Game)
tick():void
move():void
render():void
<<Java Class>>
PlayerSpawnTile
com.rustbyte.level
PlayerSpawnTile(int,int,int,int,Level)
init():void
+level
0..1
-currentTarget
0..1
0..*
-entities
0..*
#parent
0..1
#subEntities
0..*
-map
0..*
-lastUnlockedSpawnLocation
0..1

UML Diagram Description

The Game class is the common object that brings together all the components relevant to the game. The Game has a player, a level, a pathfinder, and game world entities. It also has a unique boss. The Game class holds information about the content of the game while the PlatformerGame and GameApplet classes have information and methods to run the game.

The Level class is composed of tiles, one of which represents the spawn location of the player character. The level also contains a special tile that marks the level exit and contains a tile that holds the unique boss enemy of the game. There are many different varieties of tiles that extend from the Tile base class, some of which contain additional methods.

The Entity class represents physical objects in the game world. It has methods that allow the entity to behave according to the physics of the game world. All moving objects in the game world inherit from Entity in some way. These include the particle effects and bullet traces used in the game.

The Mob class also inherits from Entity, and represents the entities that can be considered living characters in the game world. Classes for enemies like zombies and skeletons as well as friendly humans inherit from Mob. Enemies and friendly humans have an entity that they target. The Player class also inherits from Mob, as does the Powerup Class.

The game's unique boss inherits from Mob and is tied directly to the Game class. The BossSpawnerTile class also holds the boss in order to place it in the appropriate place in the game world.

Class Descriptions

Game:

The Game class is responsible for the creation of a new instance of the game. It has one Level, one player, one boss, and many Entities. The Game class has an array of different responsibilities. First, it is the class responsible for rendering the contents of the game (the screen), and in this case it means that it loads the layers of the single Level it supports. It also has a method which is responsible for respawning a Player. A Player is spawned initially when the Game is first created, and then whenever a Player is killed by an enemy.
It is also responsible for keeping track of Entity objects, which are in turn anything "tangible" in the game.
Lastly, it responsible for spawning the boss of the Level. This, in turn, a flag must be set to true when the BatBoss is to be activated. This boolean is then checked to verify if the boss has been defeated yet in the Level.

Entity

The Entity class is an abstract class which is responsible for representing the tangible objects within the game, and also the special effects that is seen in game. It is basically anything that you can see in the game other than the tiles or the background. Some examples of tangible objects in the game are grenades, enemies and humans, while miscellaneous special effects could be debris, bullets and particles.
The Entity class contains common methods that are meant to be used by all classes that inherit from the Entity class, such as how to deal with gravity, how to move, how to draw itself, and what to do every tick of the game.

Grenade

The Grenade class inherits from the Entity class abstract class. It is responsible for actions that come with the user using a grenade in game. This includes the animation of a grenade, the animation of an exploding grenade, and the damage a grenade does to Entities.
It also is in direct association with the ParticleEmitter class, which is what emulates the effects of the grenade creating sparks when used in game.

Mob:

The Mob class is a child class of the Entity abstract class. The Mob class represents all types of entities in the game which move and take damage. In turn, Player, Human, Skeleton, BatBoss and Powerup all inherit from the Mob class, each overriding their tick and render methods in a different way.
The Mob class mainly describes how things moves (for example, jumping) and how things react when they take damage. In turn, the Mob class contains methods like explode, knockBack and breakApart, which are used to take care of animations when damage is taken to a Mob object.

Player:

The Player class inherits from the Mob class, since it is an object in the game that is able to move and take damage. A player itself represents the object in game which the user controls with the keyboard. The Player class is what deals with these actions. After every tick, it performs checks to see what the Player object should do next, based on the input provided by the user. In this case, it is simply the key pressed by the user. Each key that it listens to has different actions (for example, some keys mean moving, while others fire a weapon).

Powerup:

The Powerup class is a child class of the Mob class. It is responsible for representing anything that can be picked up be a Player. This could be different things, like ammunition and grenades. Its only purpose is to determine what happens when a Player picks up any of these powerups. For example, if the user picks up the ammunition power-up, it would then add a certain amount of bullets to the current amount of bullets in your inventory.

Tile

The Tile class is responsible for representing the different types of Tiles that can exist in the game. The class holds the information on how to create any type of Tile, how to draw a tile, how to add or remove an Entity on top of a Tile, and how to add Particles (special effects) to a tile. It also keeps track of information on the tile, like the list of all Entities on top of the Tile.
There are many different types of Tiles that inherit from the Tile class. Some of the most notable are PlayerSpawnTile, MobSpawnerTile, and BossSpwanerTile, and of course there are regular tiles such as WallTile, EmptyTile and BrickTile.

BossSpawnerTile

The BossSpawnerTile class is a child class of the Tile class. It is responsible for letting the Game know that the user has reached to the point where they encounter the Bat Boss. It adds the Bat Boss to the Game.

Discrepancies

Conceptual Model	**Actual Model**
Game	Game
Item	Entity
PlayerCharacter	Player
Level	Level
Map	Tile

Many of the classes in the conceptual design mapped to the actual design but there were a few discrepancies between the two designs.

Conceptual Model
NonPlayerCharacter

Actual Model
Mob

NonPlayerCharacter:

A discrepancy between the two designs is the presence of the NonPlayerCharacter class in the conceptual design. We designed the system to have a class for playable characters and a class for non-playable characters. The actual design did not do this. The actual design for the system requires both the playable and non-playable characters to inherit from the same class, Mob. Although it does not change the effect of the overall system, and is only a difference in the system architecture, the two classes should not be inheriting from the same class because of the different behaviors.

Mob:

Mob is a class in the actual design that is not mapped to a class in the conceptual design. This is the parent class of all entities that move and take damage. As stated above, in the conceptual design, we have a class for playable and non-playable characters. Again, this discrepancy is just an architectural difference and does not affect the system.

Reverse Engineering Tools

In order to generate a UML based on the code as is, we decided we were going to use a plugin for Eclipse. The plugin we used is called ObjectAid UML Explorer, and it was very simple to add and use in Eclipse. Once the plugin was added in Eclipse, all it took was creating a new file of type Class Diagram, and then drag and drop all classes we wish to represent in the UML diagram. It makes it very quick and easy to use, giving us our actual class diagram in minutes.
ObjectAid UML Explorer is based on UML 2.0, and is able to depict Java classes, interfaces, enumerations and packages. It takes existing Java source code and transforms it into a UML class diagram. It also gets updated automatically if there are any other changes added to the code once the Class Diagram file is created.

Relationship Between BossSpawnerTile Class and Mob Class

```
public class BossSpawnerTile extends Tile:
	private static final int BOSS_SPAWN_TYPE_BAT = 0x7F006E;

	private boolean activated = false;
	private int bossActivationTimer = 0;
	private int nextMobTimer = 0;
	private int maxMobs = 1;
	private int numMobsSpawned = 0;
	private int spawnType = 0;
	private String mobName;
	private Mob boss = null;

	public BossSpawnerTile(int type, int x, int y, int wid, int hgt, Level lev)
	public void init()
	private void setSpawnBackground(int backOffsetX, int backOffsetY)

	private void spawnMob() {
		if( nextMobTimer <= 0 && numMobsSpawned < maxMobs) {
			level.game.addEntity( boss );
			nextMobTimer = 60;
			numMobsSpawned++;
		}
		if( --nextMobTimer < 0 )
			nextMobTimer = 0;
		}
	}

	public void tick() {
		if( !activated ) {
			// Check distance to player.
			Player p = level.game.player;
			Vector2 playerPos = new Vector2(p.xx, p.yy);
			Vector2 myPos = new Vector2((this.tx * 20) + 10, (this.ty * 20) +
			10);
			Vector2 delta = myPos.sub(playerPos);
			double distance = delta.length();
			if( distance < 200) {
				// Activate!
				activated = true;
				bossActivationTimer = 500;
				boss = Mob.createMob(mobName, (tx * 20) + 10, (ty * 20) +
					10,20,20,null, level.game);
				level.game.activateBoss(boss);
			}
		} else {
			if( --bossActivationTimer < 0 )
				bossActivationTimer = 0;
		}

		if( activated && bossActivationTimer <= 0) {
			spawnMob();
		}
	}
```

```
public abstract class Mob extends Entity:
	public int hitpoints;
	protected int hurtTimer = 0;
	public boolean jumping = false;
	public double jumpVel = -4.45;
	protected boolean blockedX = false;
	protected boolean blockedY = false;
	protected Random rand = new Random();
	protected FlashEffect flashEffect;

	public Mob(double x, double y, int w, int h, Entity p, Game g)
	public static Mob createMob(String mobType, int x, int y, int w, int h, Entity
		ent, Game game)
	public void tick()
	protected void hurt(int time)
	protected void knockBack(double d, double force)
	protected void explode(int partitions, int color, int particleCount)
	protected void breakApart(int partitions, int color, int particleCount)
	protected boolean isHurt()
	public void move()
	public void jump()
```

```
public class Game:
	private boolean bossActivated = false;
	private Mob boss = null;
	...

	public void activateBoss(Mob bossMob) {
		boss = bossMob;
		bossActivated = true;
	}
```

```
public class Tile:
	public Level level;
	...
```

Code Smells and System Level Refactorings

Refused Bequest

As described above, the Mob class represents all entities within the game that move and take damage. Therefore, classes such as Skeleton and Zombie appropriately inherit from Mob. On the other hand, the Powerup class extends from Mob when it should not. By inheriting from Mob, the Powerup class inherits methods and functionality that serve no purpose to its definition. The only method that Powerup uses from the Mob class is move(). We recommend removing the inheritance relationship between Mob and Powerup and extending Powerup from Entity instead.

Hard-coding

The behavior of the various types of enemies as well as the humans is determined via hardcoded string as opposed to interface or inheritance.
Power-ups are also determined in the same fashion.

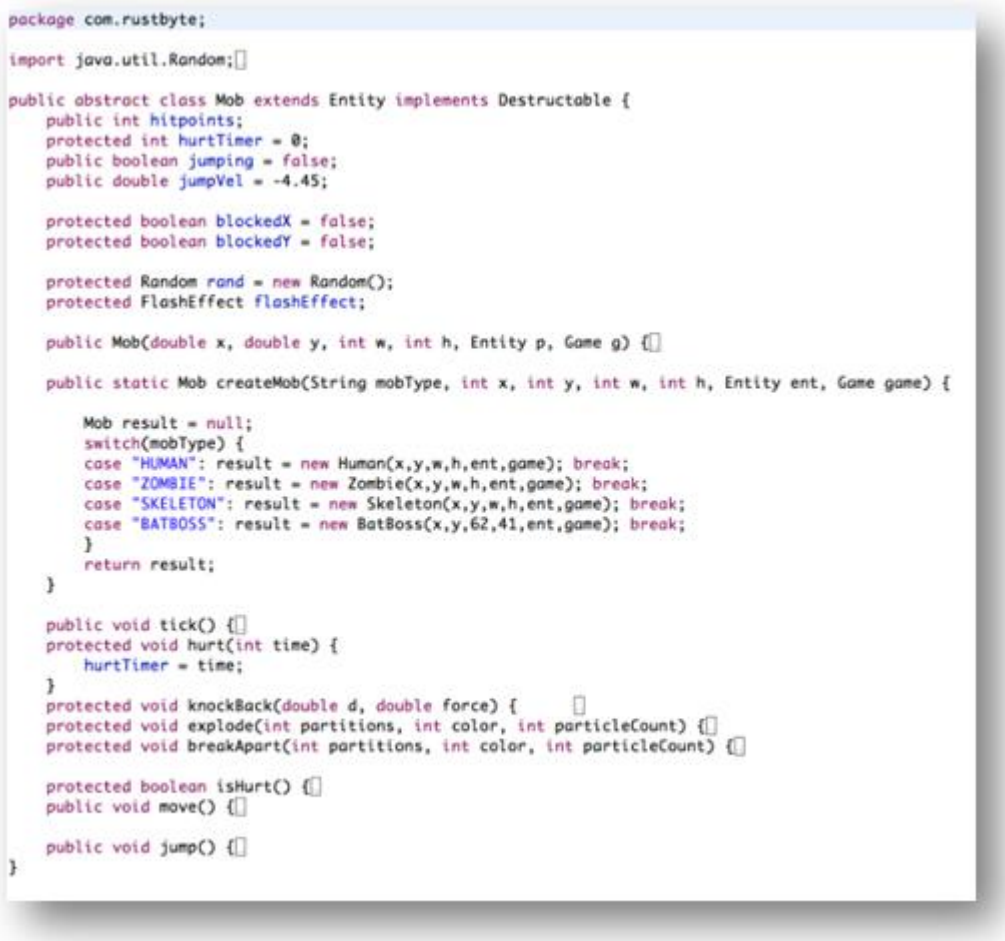
Message Chains

The BossSpawnerTile needs the player location in order to perform a method call, it does so by first calling the level object which then calls the game object which then calls the player which then calls the method to get the players location.

Intended Refactorings

Refactoring 1- Replace Conditional with Polymorphism to Rectify some Hard Coding

The Mob class's createMob method serves as a factory, however it is implemented inefficiently. Currently, the method accepts a String that is used to determine which type of Mob to create. We intend to introduce polymorphism by passing in any object that extends the Mob abstract class.



```
package com.rustbyte;

import java.util.Random;

public abstract class Mob extends Entity implements Destructable {
    public int hitpoints;
    protected int hurtTimer = 0;
    public boolean jumping = false;
    public double jumpVel = -4.45;

    protected boolean blockedX = false;
    protected boolean blockedY = false;

    protected Random rand = new Random();
    protected FlashEffect flashEffect;

    public Mob(double x, double y, int w, int h, Entity p, Game g) {

    public static Mob createMob(String mobType, int x, int y, int w, int h, Entity ent, Game game) {

        Mob result = null;
        switch(mobType) {
        case "HUMAN": result = new Human(x,y,w,h,ent,game); break;
        case "ZOMBIE": result = new Zombie(x,y,w,h,ent,game); break;
        case "SKELETON": result = new Skeleton(x,y,w,h,ent,game); break;
        case "BATBOSS": result = new BatBoss(x,y,62,41,ent,game); break;
        }
        return result;
    }

    public void tick() {
    protected void hurt(int time) {
        hurtTimer = time;
    }
    protected void knockBack(double d, double force) {
    protected void explode(int partitions, int color, int particleCount) {
    protected void breakApart(int partitions, int color, int particleCount) {

    protected boolean isHurt() {
    public void move() {

    public void jump() {
}
```

Refactoring 2 - Rectify Message Chains

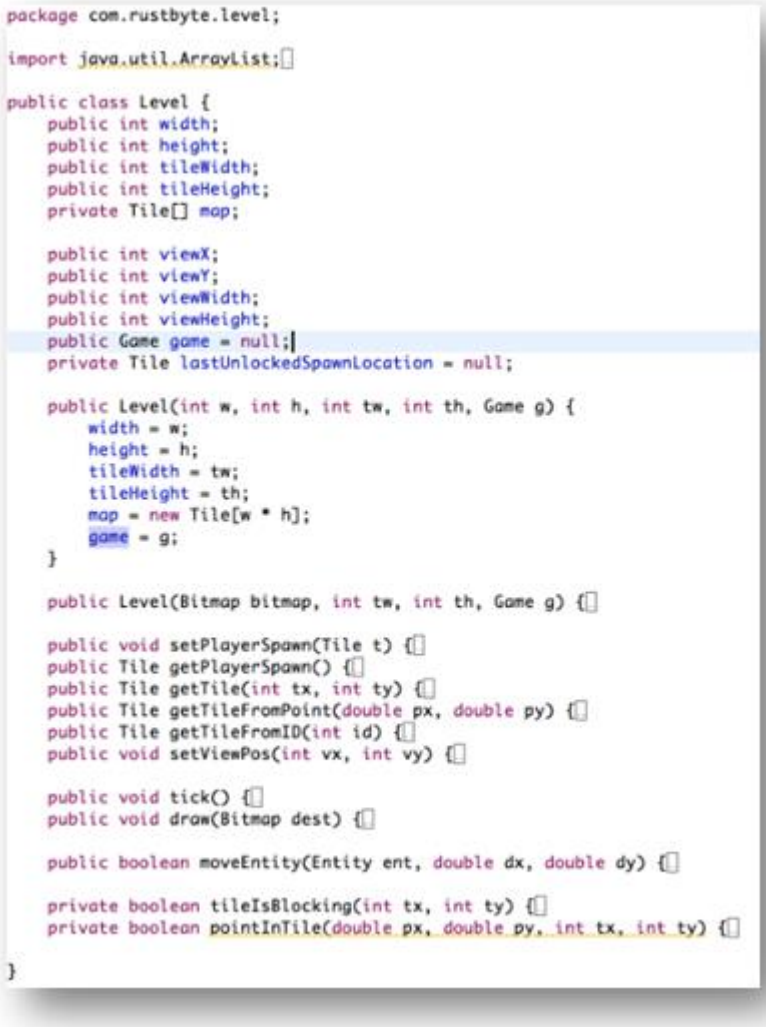
The Level class is acting as a middleman by relaying game information to the BossSpawnerTile class as well as others. As stated in Refactoring 4, the level will be stripped of its game instance, which will force us to make an association between BossSpawnerTile and game. This way, Level will cease to act as a middleman, and some message chaining will be avoided.

Refactoring 3 - Rectify Refused Bequest using Push Down

Move *destructible interface* down to Zombie,Skeleton,Human, Batboss, from Mob AND Entity to solve Refused Bequest.

Refactoring 4 - Rectify Circular Dependency in Level and Game

The level class contains the game object in addition to the game object containing the level. There are two associations made when only one will suffice. 'Level.game' is used in various Tile subclasses, however in all cases, it is only used as an intermediary to get to the game singleton object. We intend to remove the association from level to game. Anything that is broken from this change can be remedied by passing in the game object directly.



```
package com.rustbyte.level;

import java.util.ArrayList;

public class Level {
    public int width;
    public int height;
    public int tileWidth;
    public int tileHeight;
    private Tile[] map;

    public int viewX;
    public int viewY;
    public int viewWidth;
    public int viewHeight;
    public Game game = null;
    private Tile lastUnlockedSpawnLocation = null;

    public Level(int w, int h, int tw, int th, Game g) {
        width = w;
        height = h;
        tileWidth = tw;
        tileHeight = th;
        map = new Tile[w * h];
        game = g;
    }

    public Level(Bitmap bitmap, int tw, int th, Game g) {

    public void setPlayerSpawn(Tile t) {
    public Tile getPlayerSpawn() {
    public Tile getTile(int tx, int ty) {
    public Tile getTileFromPoint(double px, double py) {
    public Tile getTileFromID(int id) {
    public void setViewPos(int vx, int vy) {

    public void tick() {
    public void draw(Bitmap dest) {

    public boolean moveEntity(Entity ent, double dx, double dy) {

    private boolean tileIsBlocking(int tx, int ty) {
    private boolean pointInTile(double px, double py, int tx, int ty) {

}
```