Individual Reports

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

November 9, 2014

A design pattern present in the Java based shooter game, shotgunrobot, is the Strategy pattern.

Why is it needed?

The Strategy pattern can be applied to classes that only differ in behaviour, but the concept of the classes are basically the same. Its purpose is to encapsulate each individual algorithm and make them interchangeable to the system [1].

In the case of shotgunrobot, this pattern is used with the **Mob** class. There are many different classes that inherit from the **Mob** class. Each of these classes that inherit from the **Mob** class are objects in game that can move and take damage, but each **Mob** type has a different behaviour while they exist in game, meaning that each class inheriting from **Mob** implements a different kind of algorithm. The Strategy pattern here is used to determine how each of these Mob objects behaves as the game progresses, (specifically its tick function).

What role does it play?

As mentioned, **Mobs** are entities in the game which move and take damage. In this program, the way that any type of **Mob** object can get instantiated is through the classes **PlayerSpawnerTile**, **MobSpawnerTile** or **BossSpawnerTile**. Depending where you are in the level, the **Level** will initialize all **MobSpawnerTiles** with specific **Mob** type objects.

The different types of **Mob** objects that could be spawned are **Zombies**, **Humans**, **Players**, **Skeletons**, the **BatBoss**, and **Powerups**. The way the **Level** knows which type to spawn is based on the types of **Tiles** that are currently surrounding the player. When the **Level**, is generate this is when it knows where to place types of tiles where on the map, and therefore knowing which type of **Mob** object to spawn.

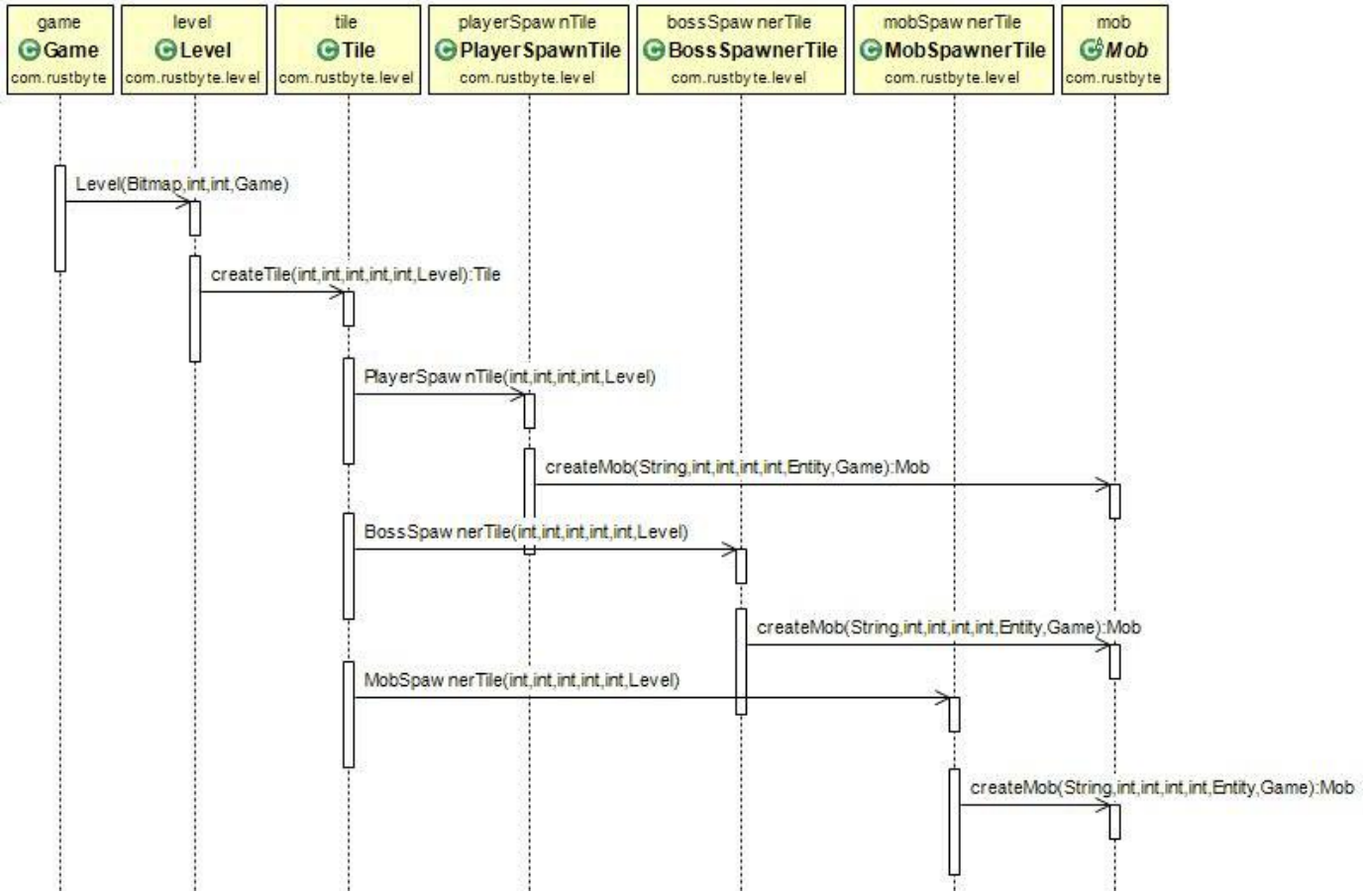
References

[1] "Strategy Design Pattern," SourceMaking, [Online]. Available: http://sourcemaking.com/design_patterns/strategy. [Accessed 6 November 2014].

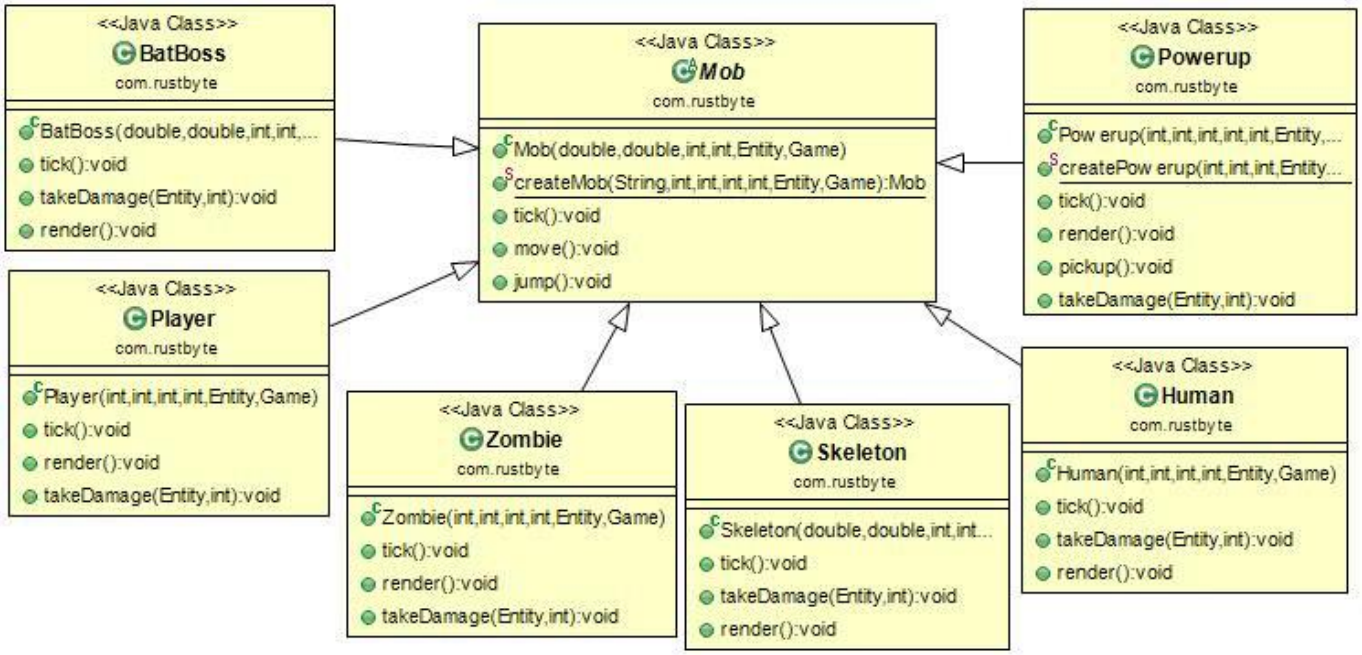
Reverse Engineering Tools

In order to generate a UML diagram and a sequence diagram for the Mob class, I have used the same plugin as we did for Milestone 3, ObjectAid UML Explorer. Once the plugin was added in Eclipse, all it took was creating a new file of type Class Diagram or Sequence Diagram, and then drag and drop all classes/methods we wish to represent in the UML diagram/sequence diagram. 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.

Sequence Diagram



UML Diagram



Code

```
public abstract class Mob extends Entity {
        public static Mob createMob(String mobType, int x, int y, int w,
        int h, Entity ent, Game game) {...}
        @Override
        public void tick() {...}
      }

      public class Player extends Mob {
        @Override
        public void tick() {...}
      }

      public class Human extends Mob {
        @Override
        public void tick() {...}
      }

      public class Zombie extends Mob {
        @Override
        public void tick() {...}
      }

      public class Skeleton extends Mob {
        @Override
        public void tick() {...}
      }

      public class BatBoss extends Mob {
        @Override
        public void tick() {...}
      }

      public class PowerUp extends Mob {
        @Override
        public void tick() {...}
      }
}
```

Daniel Caterson 9746277

Patterns Observed: Strategy and Template Method Patterns

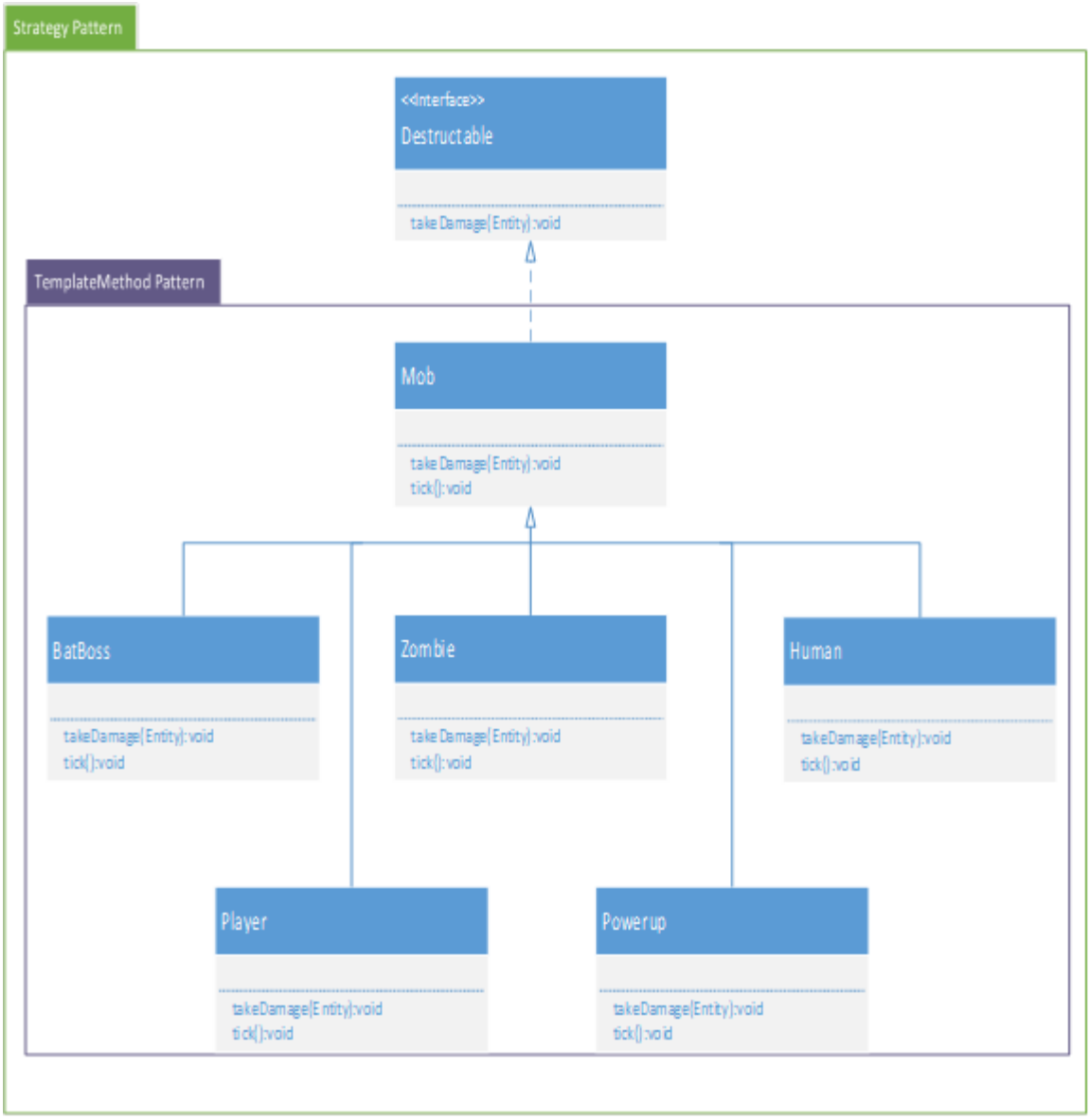
Why are they needed?

To define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. [1]
In shotgun robot, the basic flow of method calls for each subclass (super.tick()--> ishurt()--> takedamage()-->tick()) is the same, however once they reach the tick() function, each subclass needs to handle it in its own way. This "build order" is where the template method pattern shines, allowing the modification of other process that affect a swath of classes without shotgun surgery becoming necessary.

The Strategy pattern can be applied to classes that only differ in behaviour, but the concept of the classes are basically the same. Its purpose is to encapsulate each individual algorithm and make them interchangeable to the system. [2]
The destructible interface is applied to the abstract Mob class, which is the base for Human, Player, Batboss, Zombie, Skeleton and Powerup. Each of those then have to implement the takeDamage() method that was passed down to them. Depending on what class gets instantiated, our algorithm/behavior changes.

Roles

The Template Method pattern allows for some fine-tuned granularity in a mostly homogeneous recipe, all the while providing a means of easy modification. In this case the tick function is just one step of many toward bringing each mob class to life, but one that is unique across the derived siblings. You will spend far more time maintaining code than writing it, so this is a powerful asset. The Strategy pattern allows the unique handling of our single common function across varying classes.The takeDamage function will occur in each class but will be handled differently according to its runtime instantiation. Given both the Inheritance of **Mob** and the ***destructible*** interface, unique behavior is forced into derived classes. Shared attributes from **Mob** ensure that state and general expected behavior is kept, whilst the interface gives each derived class unique behavior.



Sources

[1] SourceMaking, "TemplateMethod," SourceMaking, [Online]. Available: http://sourcemaking.com/design_patterns/template_method.

[2] SourceMaking, "Strategy Pattern," SourceMaking, [Online]. Available: http://sourcemaking.com/design_patterns/strategy.

[3] W. Contributors, "Template Method Design Pattern," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Template_method_pattern.

Code

```
public interface Destructable{
	public void takeDamage(Entity source);
}


public abstract class Mob extends Entity implements Destructable {
	 public static Mob createMob(String mobType, int x, int y, int w,
	 int h, Entity ent, Game game) {...}
	@Override
	public void tick() {...}
	}
}


	public class Player extends Mob {
	@Override
	public void tick() {...}
	public void takeDamage(Entity source) {...}	}
	public class Human extends Mob {
	@Override
	public void tick() {...}
	public void takeDamage(Entity source) {...}
	}
	public class Zombie extends Mob {
	@Override
	public void tick() {...}
	public void takeDamage(Entity source) {...}
	}
	public class Skeleton extends Mob {
	@Override
	public void tick() {...}
	public void takeDamage(Entity source) {...}
	}
	public class BatBoss extends Mob {
	@Override
	public void tick() {...}
	public void takeDamage(Entity source) {...}
	}
	public class PowerUp extends Mob {
	@Override
	public void tick() {...}
	 public void takeDamage(Entity source) {...}
	}
```

Amanda Juneau 6338518
Observed pattern: Strategy Pattern

WHY IS IT NEEDED

The strategy pattern allows you to eliminate duplicate code and it also provides flexibility to change behavior by changing the composed strategy object dynamically, without causing much coupling. It separates behaviour from super and subclasses.[1]

WHAT ROLE DOES IT PLAY

The shotgunrobot project implements the strategy pattern on the Mob class. Many classes inherit from the Mob class, such as Zombie, Human, Skeleton Player and BatBoss. Each of the these classes implement a takeDamage method but each method varies to some degree depending on what kind of character it is for example, a regular enemy, such as a zombie or skeleton, takes damage different than a boss. This is where the strategy pattern comes into play.

The strategy pattern separates the different ways in which a Mob can take damage depending on what kind of Mob it is. The Destructable interface contains a takeDamage method therefore all the classes that implement the interface must define the method. From there, the Mob class should have an instance of the Destructable interface and a method referencing back to the strategies.
The diagram below shows how the strategy pattern would be implement with the classes in the in the give code for shotgunrobot

REFERENCE

[1] Java.dzone.com, (2014). *Design Patterns Uncovered: The Strategy Pattern | Javalobby*. [online] Available at: http://java.dzone.com/articles/design-patterns-strategy [Accessed 10 Nov. 2014]

CODE

```
public interface Destructable {
	public void takeDamage(Entity source, int amount);
}

public abstract class Mob extends Entity implements Destructable {...}
public class Human extends Mob {
	public void takeDamage(Entity source, int amount) {...}
...
}
public class BatBoss extends Mob {
	public void takeDamage(Entity source, int amount) {...}
...
}
public class Skeleton extends Mob {
	public void takeDamage(Entity source, int amount) {...}
...
}
public Zombie(int x, int y, int w, int h, Entity p, Game g) {
	public void takeDamage(Entity source, int amount) {...}
...
}
public Player(int x, int y, int w, int h, Entity p, Game g)   {
	public void takeDamage(Entity source, int amount) {...}
...
}
```

ShotgunRobot – George Valergas 6095836–Factory Method+Chain of Responsibility

The *factory* design pattern is implemented as a static method belonging to the **Mob** class (*See Mob#createMob()*). This factory method wraps the constructors of all the possible subclasses of **Mob**, returning the desired constructor based on a received argument. Within the **MobSpawnerTile** class, the result of Mob#createMob() is passed to the Game#addEntity() method.

This can be considered an implementation of the *Chain of Responsibility* pattern. Each time a mob is created, the object is passed as an entity to the **entities** arrayList on the Game object. Game#addEntity() wraps the arrayList#add method, as well as performing some auxiliary processing such as declaring the entity as "alive", and incrementing the count of humans or Zombies.

References:

Factory: http://www.drdobbs.com/jvm/creating-and-destroying-java-objects-par/208403883

Chain of Responsibility: http://sourcemaking.com/design_patterns/chain_of_responsibility

Code

```
public abstract class Mob extends Entity implements Destructable {
        // preceding code
        public static Mob createMob(String mobType,
        int x, int y, int w, int h, Entity ent, Game game) {
            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;
        }
        // remaining code
}
```

```
public class MobSpawnerTile extends Tile {
//. . .
    private void spawnMob() {
        if(. . .) {
                level.game.addEntity( Mob.createMob(. . .);
                nextMobTimer = 60;
                numMobsSpawned++;
        }
        // . . .
    }
//. . .
}
```

```
public class Game {
        public void addEntity(Entity ent) {
            ent.alive = true;
            entities.add(ent);
            if( ent instanceof Human ) numHumans++;
```