Peer review from OOPP-Group77 to TopDownBois

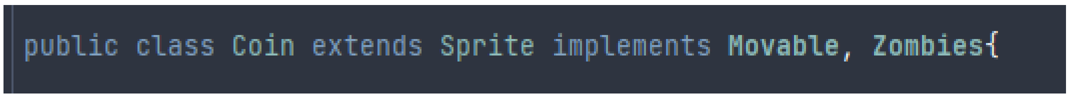
Our first impression of the game was that it is a fun idea and the animations look cool. You get an idea of what the game is about by just looking at the class names of the project. Also, the code uses a consistent coding style. Camel-case is used throughout all of the code.

Singleton pattern is used for View, ZombieObserver and MovableSubject. Since the use of singleton pattern makes it harder to test and control who has access to the instance, maybe consider using dependency injection instead.

The project uses a factory pattern for creating zombies. However, we question whether the ZombieFactory really follows the Single Responsibility Principle. In our opinion, factories should only create and return new instances of some class, not maintain any application state as is done in ZombieFactory. This could be located elsewhere.

The code contains javadoc comments for all methods which is good, but lacks comments that would have been a good addition elsewhere. For example, to make classes easier to understand and analyze, javadoc comments should be added describing the class' responsibility. Also, some additional comments within the method bodies would make the code easier to grasp. As a final note, the developers may want to add @author-tags to their javadoc comments, to give proper credit and also know who to ask about any given code snippet.

The naming of some classes and interfaces are confusing. The class TiledTestTwo is an example of this. The name indicates that the class should be some sort of test class, however it seems to be some sort of main class with a lot of responsibilities. Another example is the Zombies interface. The java naming conventions says that an interface should preferably be an adjective if the interface describes a behaviour. An interface can also be a noun for example List in the java collection library. A better name for Zombies would be for example PlayerPositionObserver because that is the functionality of the only method playerLocation(). This poor naming is very apparent when looking at the class signature for Coin.



```
public class Coin extends Sprite implements Movable, Zombies{
```

A much better interface-naming is the movable interface, which is very appropriate. Lastly, the ZombieObserver is wrongly named because the ZombieObserver is actually a ZombieObservable (which is why it contains a list of observers). It should probably be named ZombieSubject to keep consistent naming with the other observable MovableSubject.

The source code contains one package – *weapons* – but is not split into packages besides this. We would recommend creating a few distinct packages that groups classes with common responsibilities to make the code more structured and potentially easier to reuse, and also easier to understand. This would also make unnecessary dependencies (dependencies out from the package) more clear.

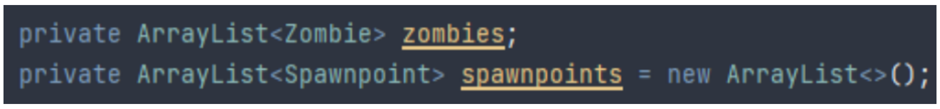
There is an attempt at an MVC pattern, however it is wrongly implemented because there are several instances where the model knows about the view. An example is Projectile which accesses the view and adds itself to the list of sprites. One of the main principles of MVC is that the Model should be isolated, which is clearly violated. One improvement to decouple view from model would be to move Sprite from the model to View. View would be responsible for attaching appropriate sprites for the objects in the model. Also consider dividing the classes into packages (model, view and controller) to make it easier to understand what classes belong to which part of the MVC.

One dependency which we question is the one from Player to PlayerController. It does not seem right that the model (Player) should have a controller as a field, as it makes the backend very coupled to the frontend and hinders reusability of the code.

The class RemindTask inside of Player could be private.

The abstract class Firearm is a good abstraction. It is properly named and has a good set of fields and methods. This makes it easy to add new weapons, however consider using dependency injection instead of creating the firearm inside of the constructor of the Player class. The application has multiple bad abstractions, mostly due to its overuse of class inheritance. For example, it is logical to assume that a Coin **is not a** Sprite. It would be much better to favor composition so Coin would instead **have a** Sprite (if Coin at all should have a Sprite since it should not know of the view).

Using composition will make the application much easier to maintain and extend because an object isn't constrained by a too narrow interface. Lastly, try to always use a broader static type than the actual dynamic type. This is in adherence to the Interface Segregation Principle. An easy fix would be to always use the less specific List type instead of ArrayList when possible.



```
private ArrayList<Zombie> zombies;
private ArrayList<Spawnpoint> spawnpoints = new ArrayList<>();
```

As far as we can see, there are no tests for the code. Tests should preferably be implemented before writing the code.

The project makes use of the observer pattern which is good to achieve a higher abstraction, decrease coupling and make the code more reusable. However, the current implementation of the observer pattern falls a bit short in reaching the proper abstraction level. In our opinion, the observers should implement some ObserverInterface and the observable implement some ObservableInterface. Then, perhaps the Main class could be responsible for adding observers to the observable's list of observers. Currently, observers such as Zombies do not implement any *onNotification()*- or *update()*-method called upon *notify()* from the observable. Instead, in CollisionController's *checkCollision()*-method, the observer's are called with the appropriate behaviour by checking their concrete class instance. This misses out on the opportunity of polymorphism that observer pattern is meant to provide, and increases coupling. Furthermore, observers such as Zombie now add themselves to the observable's list, instead of having the Main-class do it. This further increases coupling and risks reusability and maintainability.