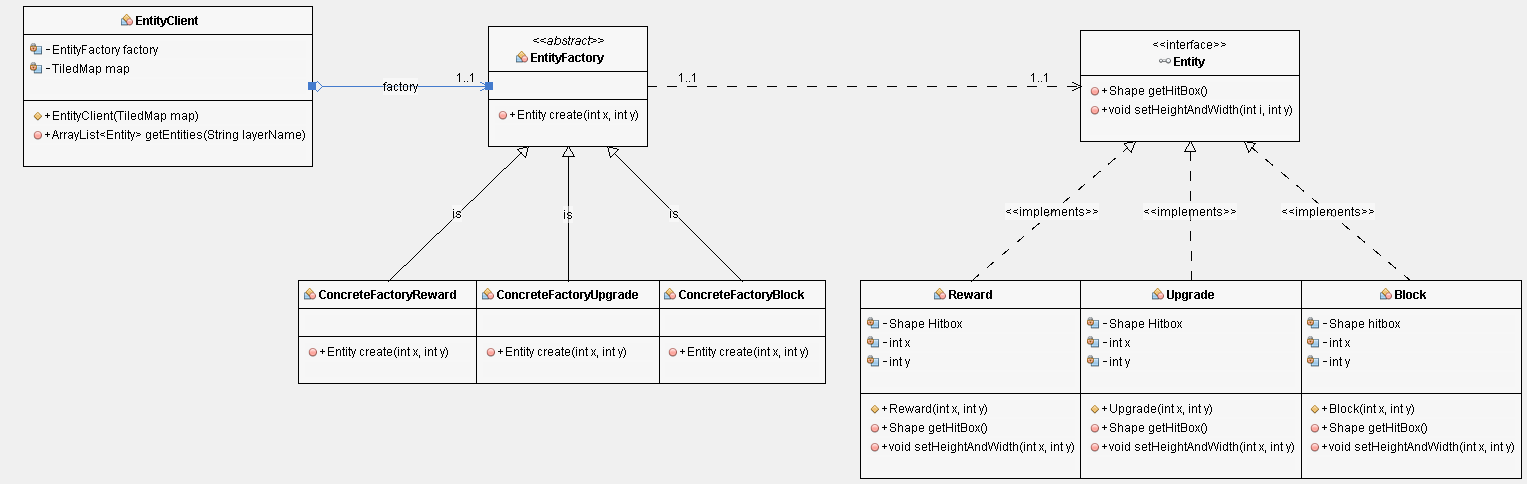
**Creation of the Entities – Factory Method**

**Problem:**

I have an EntityClient that wants to be returned, according to the layer of the map it needs, an ArrayList of entities. The problem is that, giving one of the possible layer names of the map (Walls, Rewards or Upgrades) an ArrayList of different object types is returned (Block, Reward, Upgrade).

**Solution: Factory Method pattern**

The EntityClient, according to the needed layer of the map, it creates the correct ConcreteFactory. Each of the Concrete factory has the same EntiryFactory interface, providing a “create” method. According to the name of the factory, the correct Entity will be returned aReward, an Upgrade or a Block.



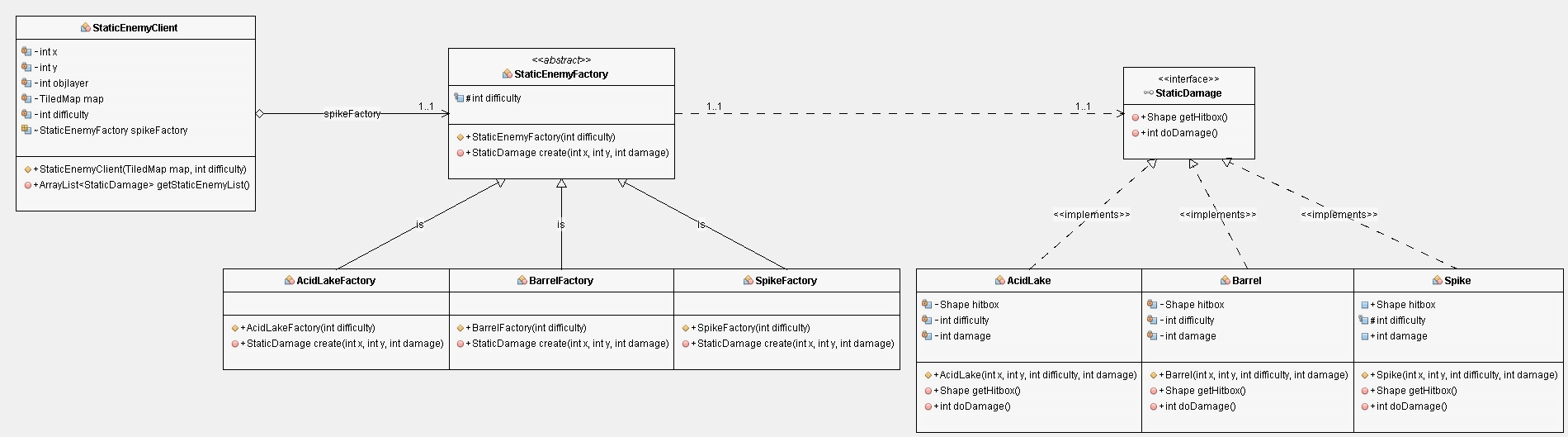
**Creation of Static Damage objects – Factory Method**

**Problem:**

I have a StaticEnemy client that, depending on the ID of the tile in the map, wants to be returned an ArrayList of objects. Those objects could be AcidLakes, Barrels and/or Spikes and they are all present in the same ArrayList thanks to the fact they share the same interface, StaticDamage.

**Solution: Factory Method**

There’s a superclass interface, StaticEnemyFactory, that allows creating objects, but allows the subclasses to alter the type of objects that will be created. In this case, the objects I want to create are AcidLake, Barrel, Spike.



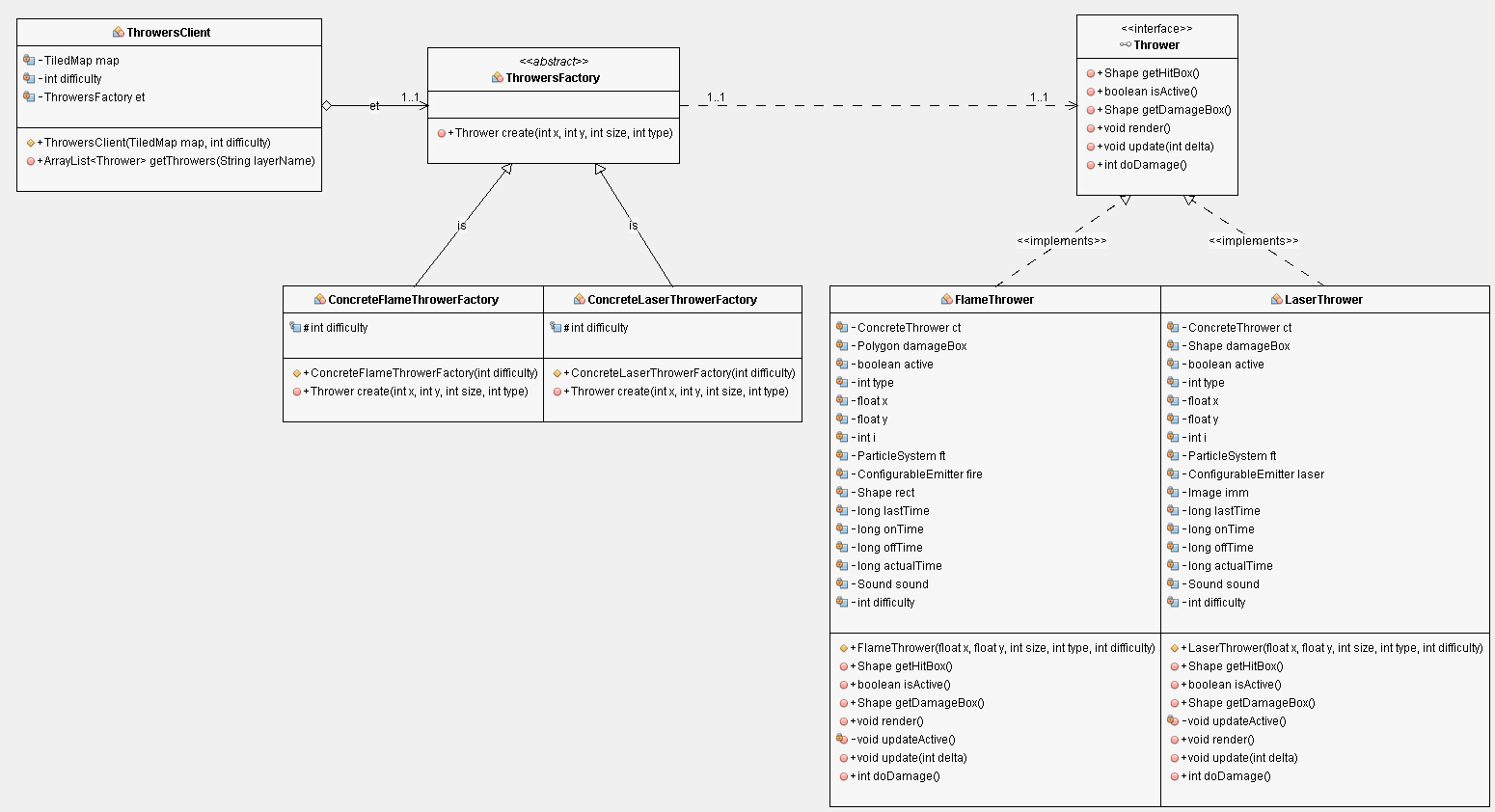
**Creation of Throwers objects – Factory Method**

**Problem:**

I have a Throwers client that, depending on the layer name of the map, wants to be returned an ArrayList of objects. Those objects could be FlameThrowers and/or FireThrowers and they are all present in the same ArrayList thanks to the fact they share the same interface, Thrower.

**Solution: Factory Method**

There’s a superclass interface, ThrowersFactory, that allows creating objects, but allows the subclasses to alter the type of objects that will be created. In this case, the objects I want to create are Flame Throwers and Laser Throwers.



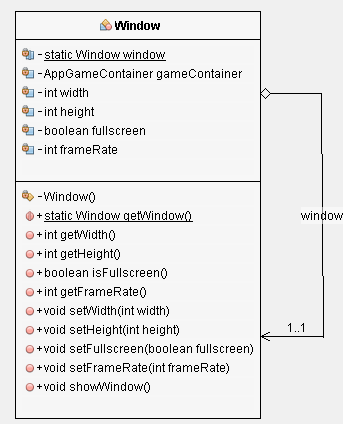
**Creation of the Window – Singleton**

**Problem:**

I need to have a single instance of the Window class, in order to avoid error by the JWJGL library that cannot manage multiple windows at the same time.

**Solution: Singleton**

I can access an already present instance of Window or create a new one if still not present, using the getWindow() method.



**Creation of the Menu - Command**

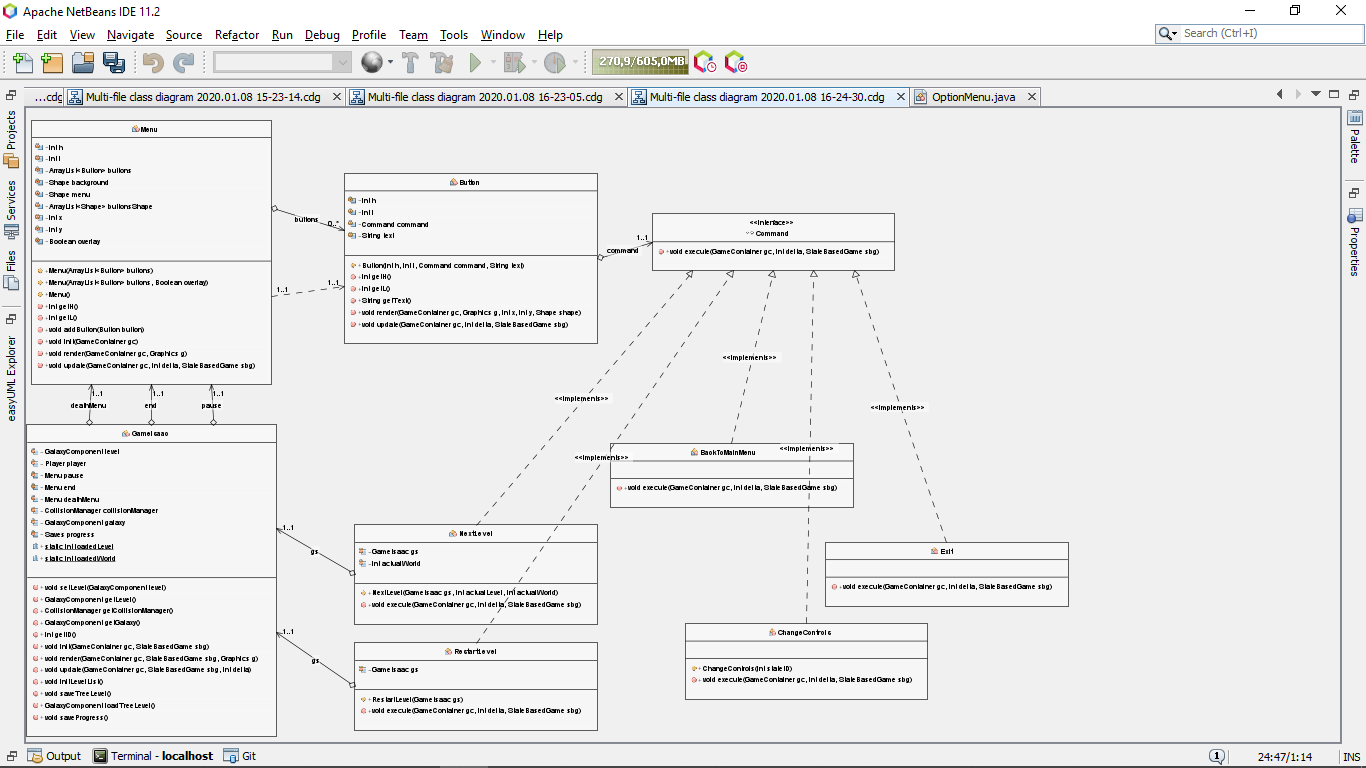
**Problem:**

I need to create some menus that perform the same actions, since I don’t want to repeat the code for each menu I used the Command pattern.

**Solution: Command**

I created a bunch of commands in order to avoid the duplication of the operation’s code in many classes or make menus dependent on buttons.

Then I put a single field into the base Button class that stores a reference to a command object and make the button execute that command on click.



**Communication between the current level and the player – Mediator**

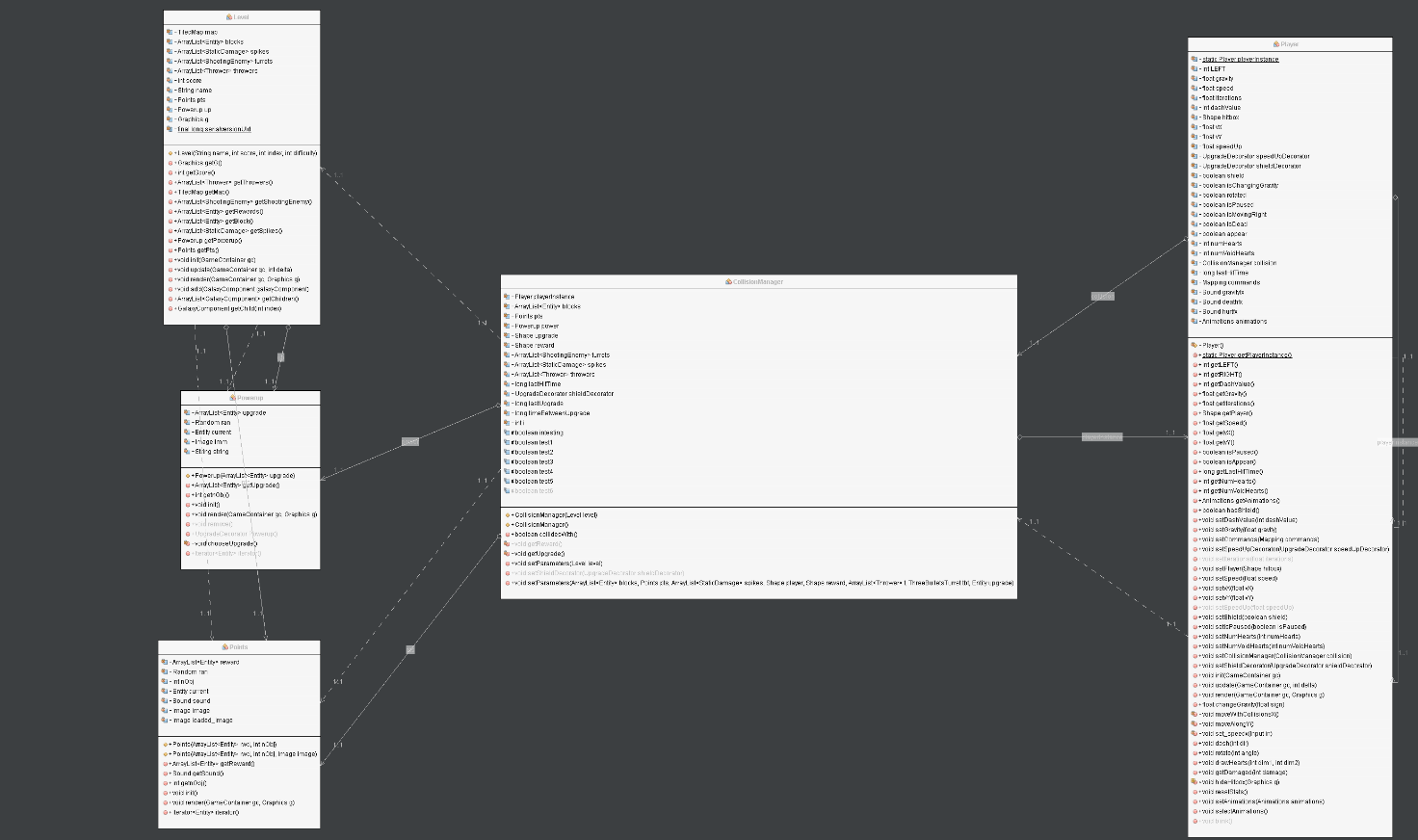
**Problem:**

I need to ease the UML of the Level class and the Player, the Player needs to signal its movements to the classes: Level, Points and Powerup, its actions trigger behaviour of these classes.

**Solution: Mediator**

The mediator “Collision Manager”, checks the collision between the player’s Shape and the Shape(s) of all the elements always present in the level: Thrower(s), Turret(s), StaticEnemies (AcidLakes and Spikes), Blocks. When the Player’s Shape collides with one or more of these elements undergo a damage or its movements are stopped.

Since when the Player collects a Reward, or an Upgrade must appear another one of these entities, the CollisionManager asks to the classes Points and Powerup if there is another element that has to spawn and, then, this element.



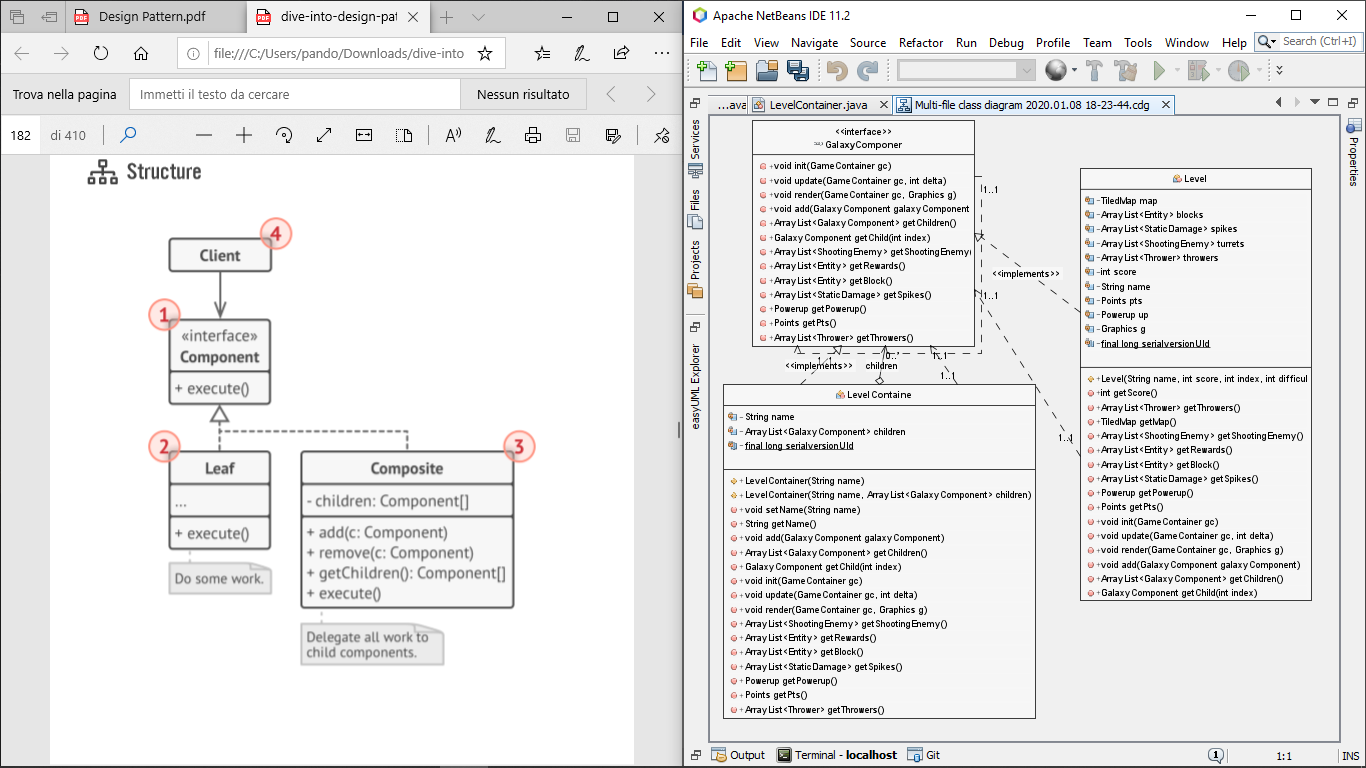
**Creation of the tree of levels – Composite**

**Problem:**

I wanted to represent the level list as a tree structure where the root is the galaxy, the branch of the root are the worlds and each leaf is a world’s level.

**Solution:**

One of the Composite pattern benefit is that you don’t need to care about the concrete classes of objects that compose the tree, so we can add or remove level in a modular way without changing the structure.



**Animation Skins – Template Method**

**Problem:**

After I implemented some skins for the Player I noticed that all these classes have a lot of similar code

**Solution:**

To avoid the repetition of code I used the Template Method that suggest to create an abstract class that implements the methods that have to be inheritate by every subclass and other optional methods with some default implementation that can be overridden if needed.

