Exercise 2 - Theory in practice

1. What is the difference between aggregation and composition? Where are

composition and aggregation used in your projet? Describe the classes and explain how these associations work.

Aggregation means that the two classes complement eachother, but that they can function perfectly fine on their own. Composition means that one class does not function without having the others.

To explain this, we need two objects that consist of different parts. As an example, let us take a car and a car dealership. In the case of the car, its parts make up the car: there is no car without an engine, a seat and wheels. The car and its parts form a composition.

In the other case, the car dealership, its parts do not compose the object: if one of the cars gets stolen or is replaced with a completely different car, the dealership's function is not hindered. The dealership and its cars have form an aggregation.

Composition and aggregation are used in the following relationships:

1. Grid - Tile. A grid contains sixteen tiles that can be moved around. This relation is a composition, because without the tiles the grid would serve no purpose.

2. Grid - TileHandler. A grid uses a tilehandler to control its tiles. This relation is a composition, because without a tilehandler the tiles cannot be moved and the grid would, again, serve no purpose.

3. GameScreen - GameWorld, GameRenderer, InputHandler. The GameScreen is the main game screen where everything is rendered. It contains the game loop and controls all other classes. The GameRenderer is needed to render all the objects and the GameWorld is required to control all these objects. It also keeps track of the current game state. The InputHandler is used to manage our input events. This is a composition, because without all these classes the GameScreen does not function.

2. Draw the class diagrams for all the hierarchies in your source code.

Explain why you created these hierarchies and classify their type. Are there hierarchies that should be removed? Explain and implement any necessary change.

Our game has several hierarchies. For the UML class diagrams, please see

the file "2048_v1.0.0_all_hierarchies.png" in this same folder.

1. The Restart and ContinueButton extend the abstract SimpleButton. This is an "Is-a" hierarchy, which is created because the RestartButton and ContinueButton would otherwise share code: the only code they would not share is their onClick method. Thus our solution was to create an abstract SimpleButton class to hold the shared code, forcing the derived buttons to define their own onClick method.
2. The Grid and AnimatedGrid relation is an "Is-a" hierarchy. This is done to adhere to the Model-View-Controller design pattern.
3. The Tile and AnimatiedTile relation also is an "Is-a" hierarchy, which is also created to adhere to the Model-View-Controller design pattern.

Considering the lectures, we do not feel that any of our hierarchies should be removed, because they each serve their own well-defined purpose.

3. Where do you use if or case statements in your source code?

Refactor them so that they are not necessary anymore and describe your refactoring. If they cannot be refactored, explain why.

The following classes contain **if** or **case** statements:

Handlers package:

- AssetHandler;
- Buttonhandler;
- CoordinateHandler;
- InputHandler;
- PreferenceHandler;
- ProgressHandler;
- TileHandler;

Buttons package:

- SimpleButton;

Game package:

- GameRenderer;
- GameWorld;
- Launcher;

Gameobjects package:

- AnimatedTile;
- Tile;
- Grid.

Some of these **case** and **if** statements can be refactored, as we have actually done in v2.0.0.