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. All screens and their actors. A screen contains several actors (controllable and drawn objects). All these relations are compositions, because without actors a screen would be empty and not function properly.

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_v2.0.0_all_hierarchies.png" in this same folder.

1. All buttons extend LibGDX' TextButton. This is an "Is-a" hierarchy, which is created because we use these buttons in several menus and thus

wanted to prevent code duplication.
2. TwentyFourtyGame extends LibGDX' Game. This is an "Is-a" hierarchy, which is created to hook into LibGDX' events.
3. Grid extends LibGDX' Actor. This is again an "Is-a" hierarchy, which is created to easily at our grid into the relevant screens.
4. Tile extends LibGDX' Actor. Again this is an "Is-a" hierarchy, which is created to easily draw our tiles from inside the grid.
5. ScoreDisplay extends LibGDX' Group. This is also an "Is-a" hierarchy, which is created to easily draw and position our score tiles.
6. InputHandler and LocalInputHandler both extend LibGDX' InputListener. This is again an "Is-a" hierarchy, created to write our own keyDown method upon a key pess.
7. All our screens extend the abstract Screen class. This is a polymorphism hierarchy, created 1) to prevent code duplication and 2) to manage all our screens in the ScreenHandler which keeps a stack of openend Screen objects.

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:

Screens package:

- ClientScreen;
- GameScreen;
- HostScreen;
- MultiGameScreen;
- ScreenHandler;
- WaitScreen.

Net package:

- Networking.

Handlers package:

- TileHandler;
- ProgressHandler;
- PreferenceHandler;
- LocalInputHandler.

Gameobjects package:

- Tile;
- Grid.

Game package:

- Launcher.

The case statements we use cannot be refactored, since we rely on LibGDX to receive input events and LibGDX hands us an integer keycode. We are not aware of any method in Java that would enable us to not use a **case** statement.

The **if** statements we have left, also cannot be refactored. They are general **if** statements used to check for conditions, e.g.:

```
if (score > highscore) { highscore = score; }
```

We felt like going overboard with removing these **if** statements would ruin our code.