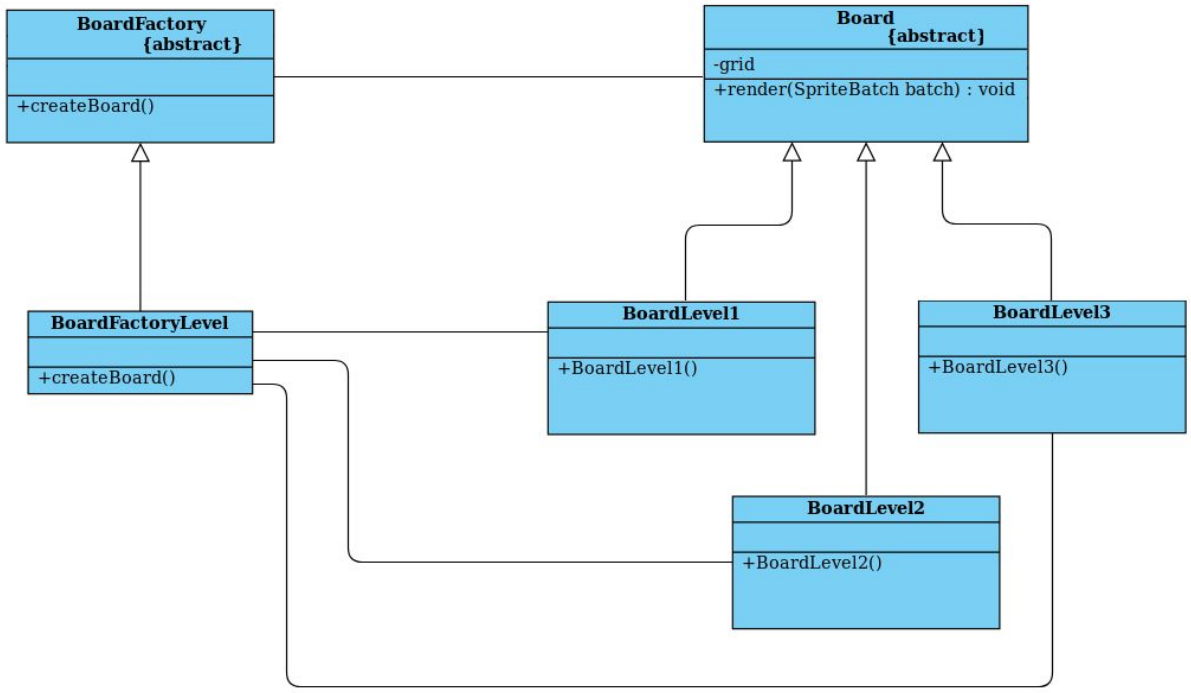
Exercise 1: Design Patterns:

1. **Abstract Factory Pattern:**



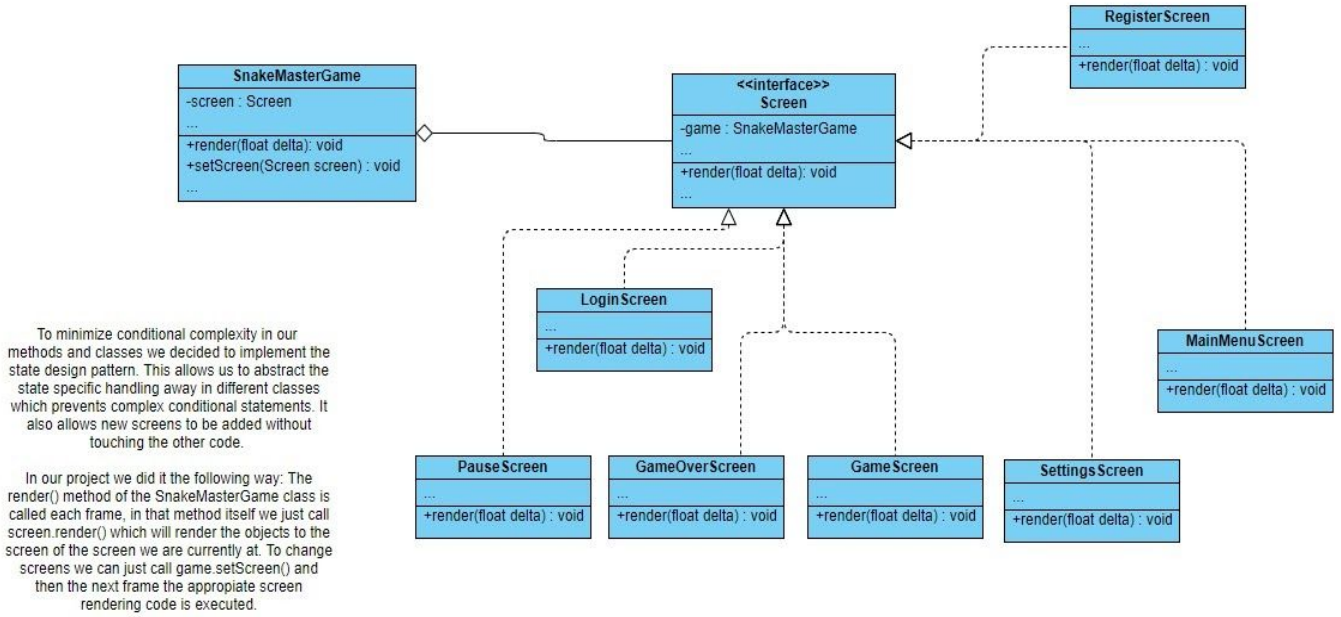
To isolate other classes from the specific details on how to create a specific board for a level, we implemented the Factory Design pattern. Our classes implementing this pattern neither have to know the details of a board nor the creation of it.

We define an abstract class *BoardFactory* (top left) which has an abstract method *createBoard(int level)*. This method gets implemented by a specific factory namely our *BoardFactoryLevel* (bottom left). This structure of an abstract parent factory and a specific child factory makes it convenient to create other types of factories that involve the creation of different types of boards. An example would be a factory for a board that is not used in a level, but rather for a 'free play' mode not involving levels. The *Board* class (top right) is the product of this factory setup. What the user wants are the boards for the specific levels. These are implementations of the *Board* abstract class (bottom right). A call to a constructor of a specific boards happens in the *createBoard(int level)* method of the *BoardFactoryLevel* class. With this, the user only needs to specify the level to get the desired board as an object returned. When calling this method, a set of *if conditions* evaluates which of the specific

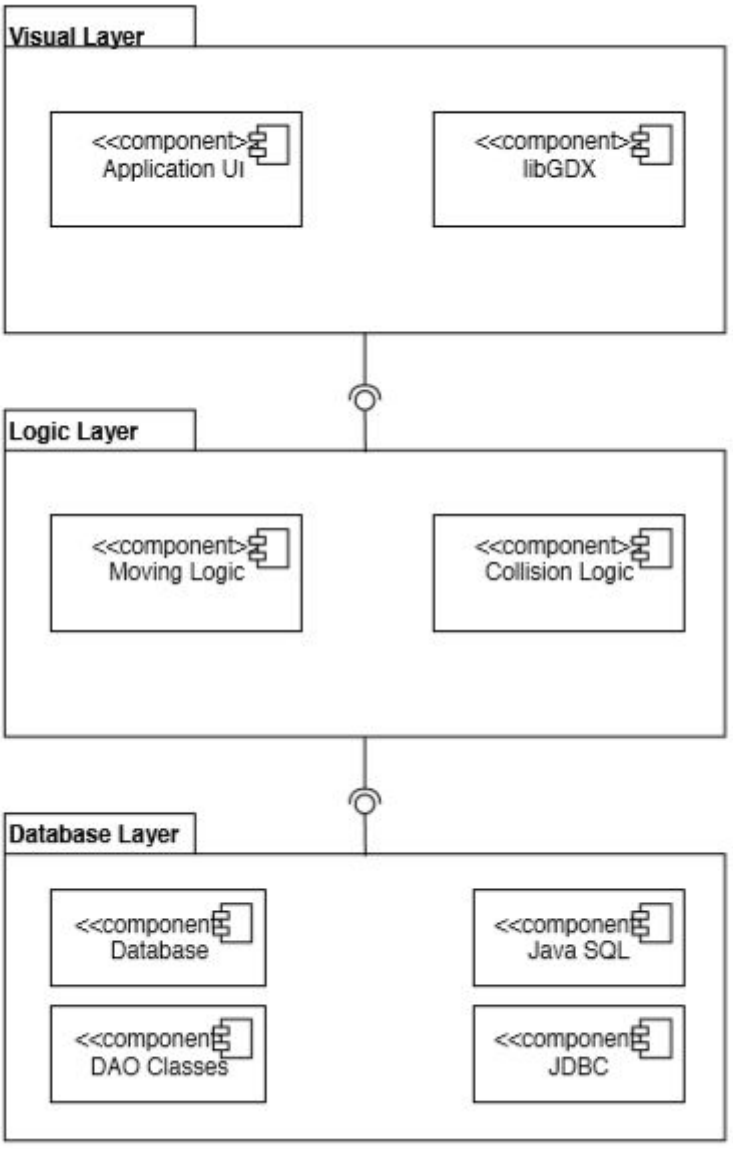
board needs to be instantiated. All the details of this board are in its own board class. This directly brings us to the reasons why we choose this design pattern, namely to increase cohesion among the specific *Board* classes and, second, to reducing coupling between these Board classes and other classes using them. A third reason for us to use this design pattern is to not let classes anticipate in the creation of boards which would make the code more complicated.

In our previous attempt to realize the board creation process, we only had one board class that needed specific details as parameters on how big the board should be and with which sprite it should be tiled with. This approach was not flexible for boards that needed more that one sprite. This case meant that different *Sprite* objects as parameters needed to be passed to the board. But boards that would not use this extra field could cause bugs in the long run. Using a approach with many imports for the single *Board* class would also not work, because different boards (contained of single or multiple sprites) would need different constructors anyway. With our Factory Design pattern all of this is abstracted away by only having to specify a level to get a board returned by a unique constructor for that board. This makes the board classes more coherent and thus less coupled to others. Using different classes in our first attempt would still mean a class would anticipates in the creation process. As mentioned before, this was something we wanted to overcome. Altogether the Factory Design pattern filled our needs in a solid and cohesive manner.

2. State Pattern:



Exercise 2: Software Architecture:



When we started designing our system there was no clear system architecture that we had in mind as our main goal was to just see a moving object. But as we started to put everything together, research libraries that we will use and implement a database to manage our users and data, a clear layer structure had emerged. In our current state of the system we have three main parts which are firstly the visual component that the user sees, then the logic unit in which the system decides what needs to be displayed and in the bottom - data management component which deals with storing the users information and its achievements.

Visual Layer

The **visual layer package** is responsible for everything the user sees on the screen. That includes the Login, Register, Main Menu, Settings, Game and Pause screens which are the different states in the **application UI**. Additionally, we implemented a Setup class which contains all the constants used like fonts, textures, button styles. They are from the external **libGDX** library which we use to create and manage the screens. The screen changing is implemented with the State design pattern which enables efficient screen changes and minimizes conditional complexity. When a certain event occurs and the internal state of the game changes, the game will change its behaviour to that of the state it is now in. Together with the screens in the **application UI component** there are the Food and Snake objects which have a graphical representation but are closely intertwined with the **Logic layer package** as it needs to update the snakes position every frame and the location of the food if collision is detected.

Logic Layer

The **logic layer package** doesn't need to take care of which screen is currently 'active', as the State design pattern delegates the behaviour to the right concrete state. Secondly it handles all the keyboard inputs for the snake movement in the **moving logic component**, collision detection, score incrementation in the **collision logic component**, and, lastly, it decides where to spawn a new Food object so it is not on top of a snake. The Board is created by a BoardFactory which follows the Factory design pattern. The board is made out of squares which can represent different 'tiles'. This design choice made it easy to extend our levels by introducing squares other than Grass, for example, Walls, which could introduce some variability to the game. In addition to gameplay logic, the package is also responsible for managing the transactions between the **Database package**. It makes requests to the database to check for valid logins, save user scores and register new users.

Database Layer

The **database layer package** acts as a data storage for the system. On system start up the connection with the **Database** is initialized through the **JDBC component**. The database holds all user credentials with names, passwords, which are securely hashed with randomized salts, and scores . Whenever there is a request from the **Logic layer** to get some user information, for example, personal highest score, there are specific data access objects (**DAO classes**) that are responsible for these **Java SQL** transactions. All queries are made with prepared statements in order to increase the security and integrity of the system.