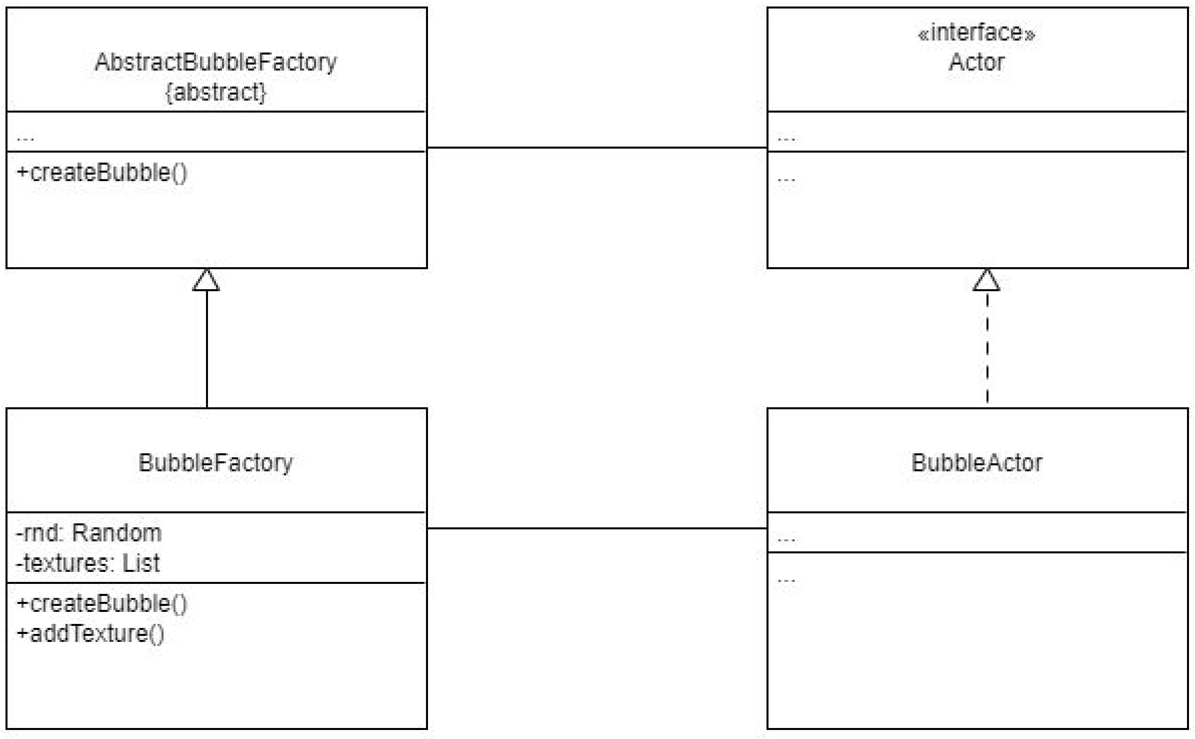
Assignment 3 - Group 15

Exercise 1 – Design Patterns

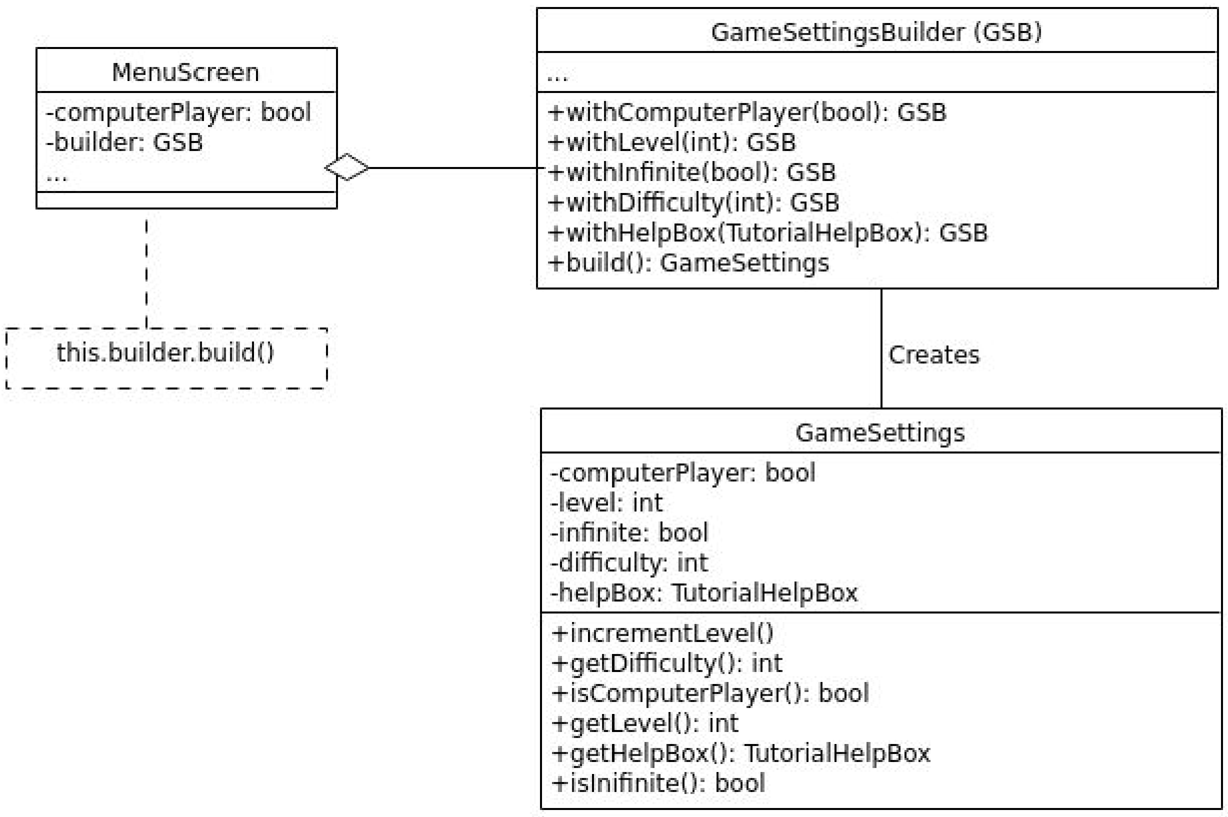
Factory bubble pattern

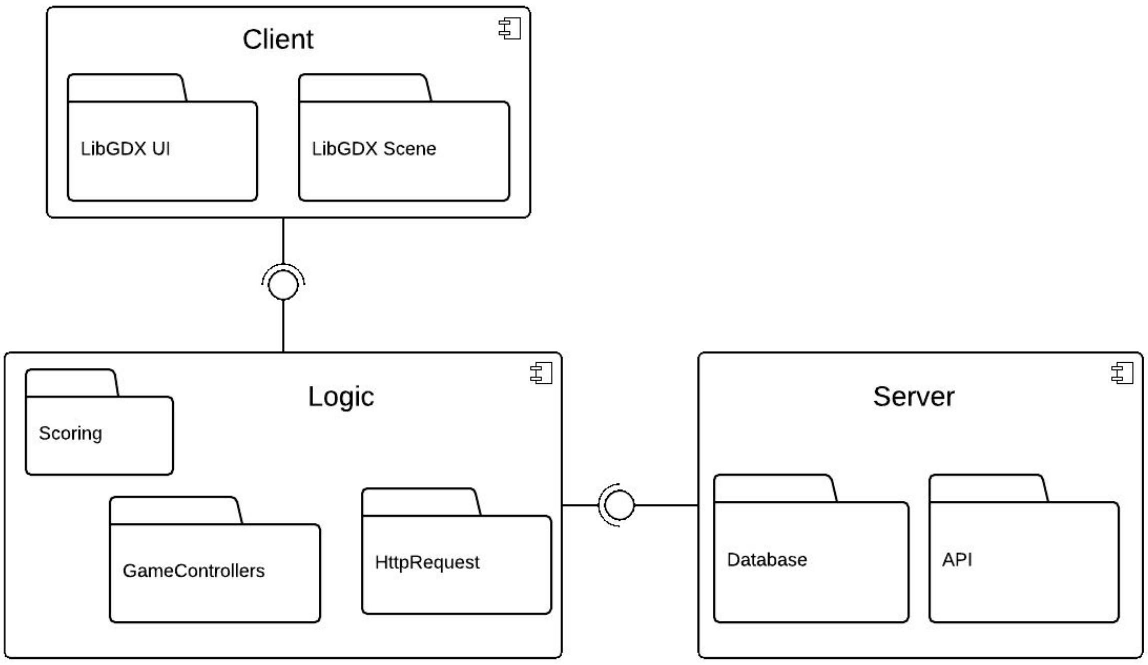
In the game we need to create arbitrary bubbles for different objects such as the shooter at the top of the screen or the central hexagon. Both objects do not need to know how the bubbles are created nor how to create them themselves therefore we created a BubbleFactory class. We applied the design pattern Factory Method such as the BubbleFactory class has an interface consisting of a single method called createBubble() which will be used by the different objects to create arbitrary bubbles. In the implementation we decided to return a BubbleActor object to make it easier for the shooter/hexagon to use its methods.



Builder pattern

When we're setting up the game, there's a lot of different settings that need to be applied. Because the GameSettings object has many fields that we need to set, we decided to apply the Builder pattern here. As far as applicability goes, we think this is the perfect place for the use of such a design pattern, because the creation of the GameSettings object is seperate from the different components that go into its creation, and we want to have a higher degree of freedom over the creation process. Additionally, the builder pattern really improves the readability of the construction of GameSettings objects.
The way we decided to apply this design pattern was by using a class called GameSettingsBuilder, the sole purpose of this class is to build a single GameSettings object. The different public methods of the GameSettingsBuilder class are used to set the fields in the final GameSettings object. Finally, after setting the different attributes, the programmer uses the build() method to create the final GameSettings object.




Client
LibGDX UI
LibGDX Scene
Logic
Scoring
GameControllers
HttpRequest
Server
Database
API

Exercise 2 – Software Architecture

We use a 3-tier layer architecture, this means each layer performs a specific role and has a unique responsibility within the overall application architecture.
This means the layers are designed as independent modules, and from the top downwards they are called:

1. Client
2. Logic
3. Server

The motivation behind this encapsulation of the layers is simple: it allows us to freely work on separate parts of the project without getting in each other's way. We chose this architecture over the ports and adapters because we didn't want to make the implementation complex when incorporating different libraries into the game, e.g. libGDX. At first we wanted to go for a simple client-server architecture but we decided to add a new layer of abstraction in order to separate the logic from the UI and improve the testability of our code.

We will now go into depth on the individual layers:

Client

The first layer we can distinguish is the client layer, this layer is responsible for the interactive side of the application (e.g. Showing the UI, handling input). Seeing as we chose to use LibGDX as the framework for our game, we've structured the client layer in a way that fits this framework best.
This means that we use LibGDX not only for showing the different UI screens, but also for the drawing of the game itself. Furthermore, the client layer is also where the input from the user is coming in and subsequently handled in the Logic layer.

Logic

The second layer is the logic layer, its main job is to handle the game logic. This includes setting up the game by creating the necessary objects, creating the individual bubbles, and parts of the code we use to control the bot are also a part of this layer. The logic layer takes care of game state as well, e.g. not letting you shoot bubbles when another bubble is still traveling or keeping track of the score.
Another important part of the logic layer is handling the interaction with the layer below (server), and wraps this up nicely so the data can be used without having to

know the server implementation specifics. Moreover this layer implements an HTTP client which is used to communicate with the server API.

Server

The third layer is called the Server layer. The server layer is responsible for persistent storage of information and handling the API calls from the clients.
The server consists of a SQL based database, namely SQLite, which stores user credentials, scores, badges and game history. For the API we use a RESTful architecture, which uses HTTP coding to facilitate the transmission of messages between client and server. The server offers various API endpoints that are directly mapped to communicate with the database, e.g. the login endpoint will check the user credentials in the database.
The database component is a wrapper around the JDBC driver which allows to communicate with the database and perform queries to insert or read data. Each SQL query sent to the database is prepared to prevent injection attacks.