Software Engineering Methods

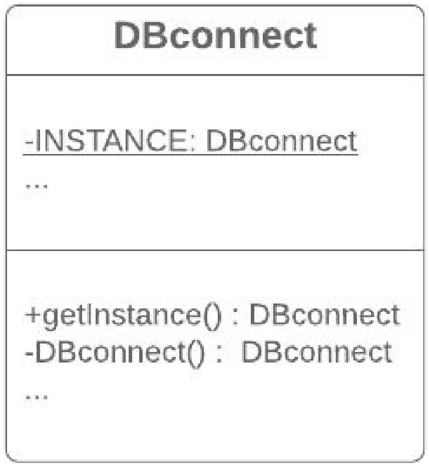
Assignment 3

Snake-Group-1

Table of Contents

Exercise 1) Design Patterns

- Class Diagram Singleton

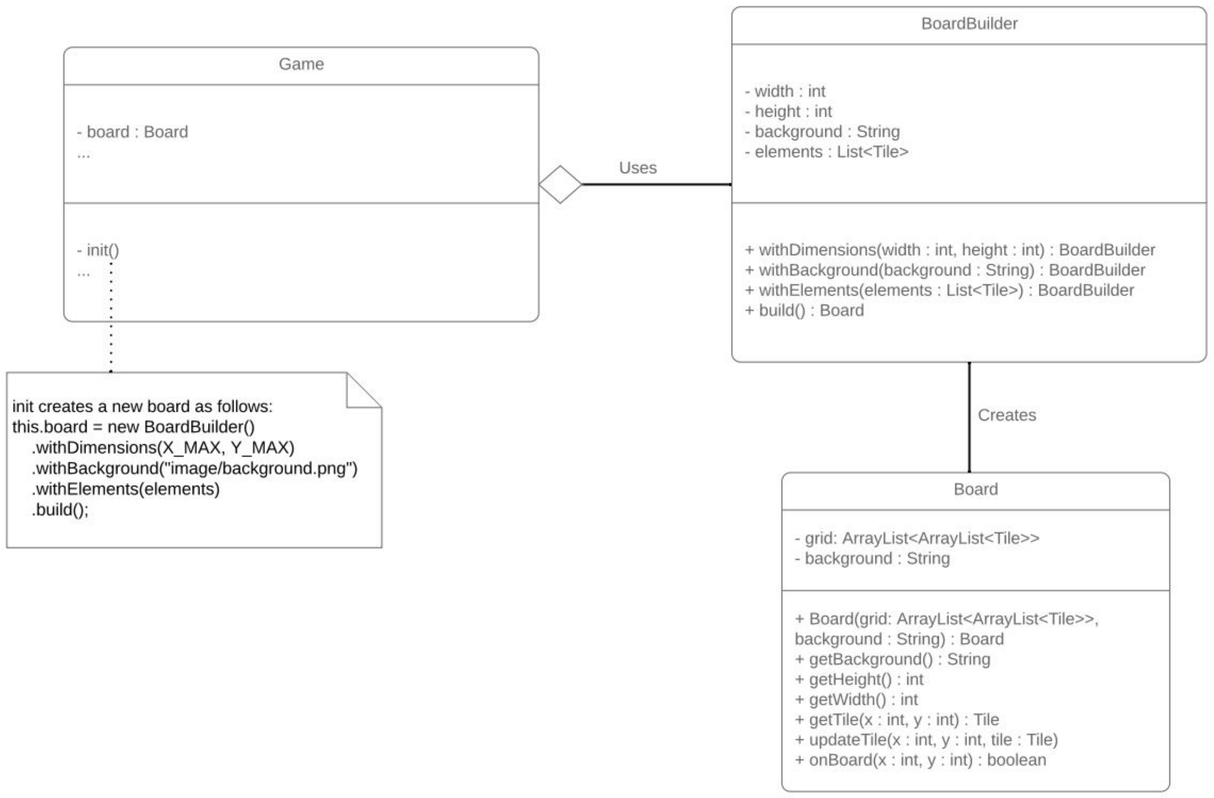


Description Singleton

Setting up a connection with the database is computationally costly. That's why we choose to implement a singleton pattern for the database connection which is encapsulated in DBconnect. At any point in the game, there is only one instance of the DBconnect class. Subsequent calls only retrieve a new reference to the connection.

DBconnect has static reference to the only DBconnect instance. The constructor is private. The method getInstance is used to retrieve an instance of DBconnect. This method creates a new object of DBconnect and assign it to INSTANCE if and only if no other instance of DBconnect already exists. When INSTANCE is already instantiated the method will return the instance.

- ## Class Diagram Builder Pattern


Game
- board : Board
...
- init()
...
init creates a new board as follows:
this.board = new BoardBuilder()
.withDimensions(X_MAX, Y_MAX)
.withBackground("image/background.png")
.withElements(elements)
.build();
Uses
BoardBuilder
- width : int
- height : int
- background : String
- elements : List<Tile>
+ withDimensions(width : int, height : int) : BoardBuilder
+ withBackground(background : String) : BoardBuilder
+ withElements(elements : List<Tile>) : BoardBuilder
+ build() : Board
Creates
Board
- grid: ArrayList<ArrayList<Tile>>
- background : String
+ Board(grid: ArrayList<ArrayList<Tile>>, background : String) : Board
+ getBackground() : String
+ getHeight() : int
+ getWidth() : int
+ getTile(x : int, y : int) : Tile
+ updateTile(x : int, y : int, tile : Tile)
+ onBoard(x : int, y : int) : boolean

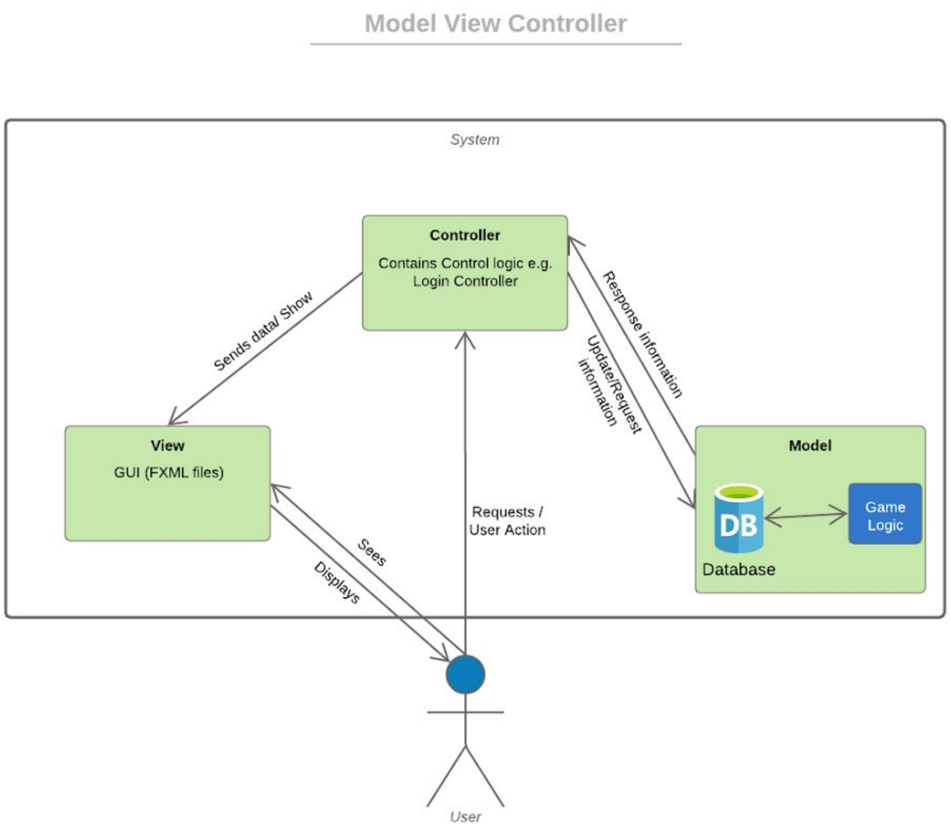
Description Builder

We want to be able to create boards with a different layout. A different layout means that walls and fruits are placed on different positions. To be able to create different boards in a convenient way we implemented the Builder pattern.

The abstract builder class is not represented in the diagram since we only needed one concrete builder - BoardBuilder. Although the Game class does not have an instance of a BoardBuilder, it does use the BoardBuilder class as can be seen in the comment in the diagram above.

We are using the BoardBuilder class to create objects of type Board. These objects contain a grid element which is used by the Game class (and the CollisionManager class which is not represented above for simplicity) for various purposes: drawing the sprites, checking collisions, etc. The reason for choosing the Builder pattern is that we want to be able to have different types of boards (for example boards that vary in dimension or which have different arrangements for the walls).

Exercise 2) Software Architecture



Components

The ‘View’ component is basically GUI part of our application. ‘View’ is responsible for representing the information in user understandable format. We are using JavaFX for this part and we have been using fxml files or coding (using JavaFX library) for GUI components. ‘View’ is a user interface. User can see the data via ‘View’ (Game Screen) and also ‘View’ displays data using model (‘database’) to the user.

The ‘Controller’ is a component that acts as a mediator between view and model. The user interacts with GUI via ‘View’ and the request is handled by a controller. It is responsible to control the data transmission between the model and the view. We have implemented controllers under the name ‘--controller’ in our code. We usually have one controller for each fxml file. We have separated GUI logic from controlling logic thoroughly.

The ‘Model’ is a component that represents data and core logic (game logic) of application. It gets the request from the controller and update database / retrieves data from the database. In our case, we are using EWI server provided by the course for our database. It is basically MySQL. The database contains all the data related to the game such as scores, player names and password.

Why Model-View-Controller over other architectural patterns

We have chosen Model-View-Controller as an architectural pattern of our application due to the following reasons:

- **Main Program and Subroutines:** our code is not structured hierarchically and not focused on functions. Although code is reusable and modular, the program is not divided into subroutines and there is only one thread of control running at any given time.
- **Client and Server**: we do not have a client and server communicating through messages. The only behaviour similar to this is the connection to the database, but that connection is only used for logging the user in and updating the leaderboard, not the whole application.
- **Layered Architecture**: classes are not grouped in layers. Although logic for the individual components is separated, they do not necessarily follow the least knowledge principle nor do they communicate through APIs.
- **Multi-Tier Pattern**: runtime objects do not operate in separate tiers. Although the database is hosted on a dedicated server, it is the only component running in something that could be called a tier.
- **Ports and Adapters**: although business logic is isolated form the services used across the entire application, the only component that can truly be aligned with this approach is the database connection.
- **Pipe-and-Filter**: our application does not filter/buffer data. There is no data stream flowing through the components and being altered.
- **Service-Oriented Architecture**: although the components are implemented separately, each with their own functionalities, they do not necessarily follow this architecture as there is no registry for the services to use for discovering the other services.
- **Publish-Subscribe**: there is not one single bus that publishers and subscribers use. When data is passed around, it is done so directly to the receiving class.
- **Event-Driven Architecture**: the application is not organized as a finite state machine with events triggering transitions from one state to another.