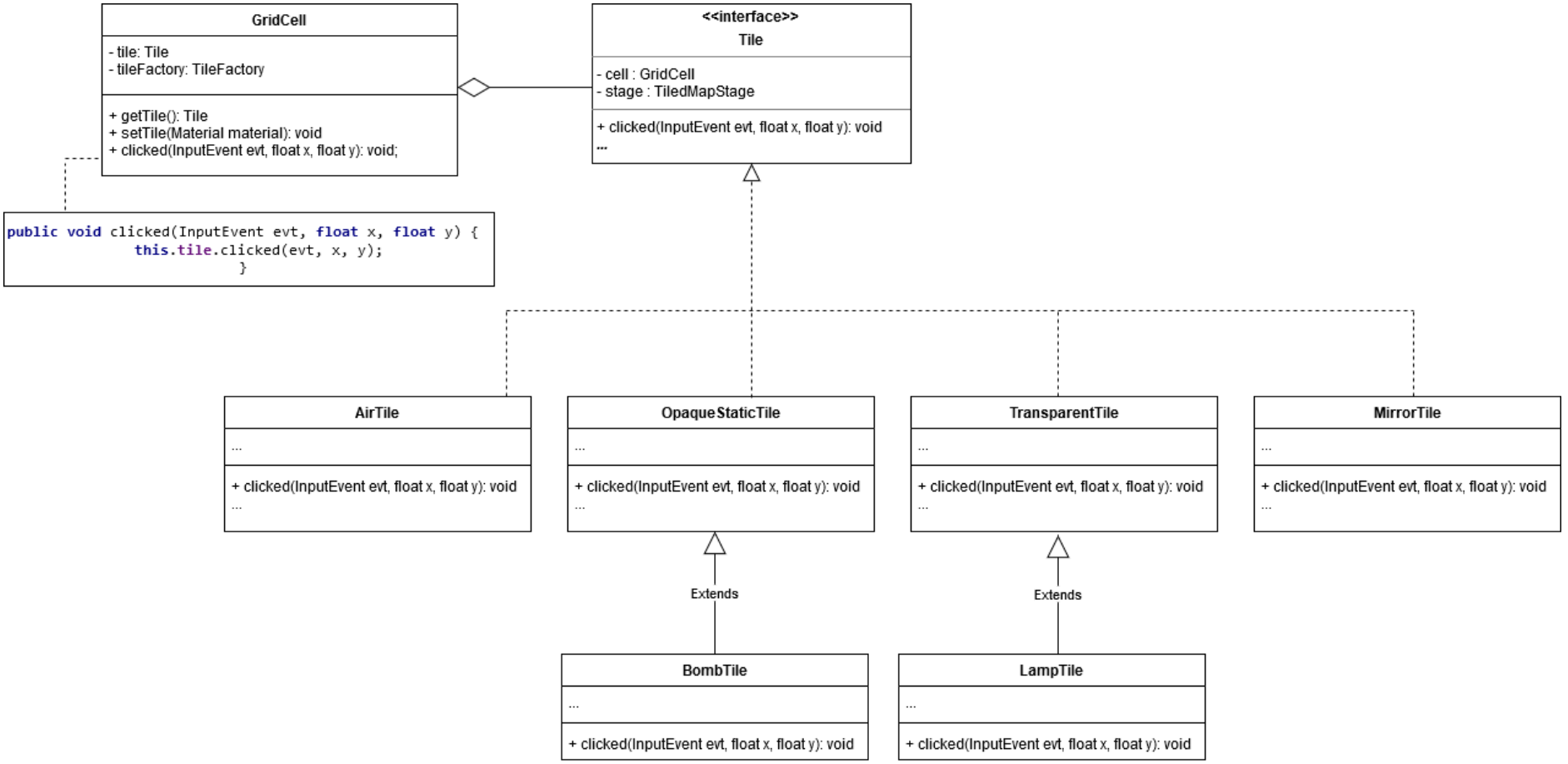
Assignment 3

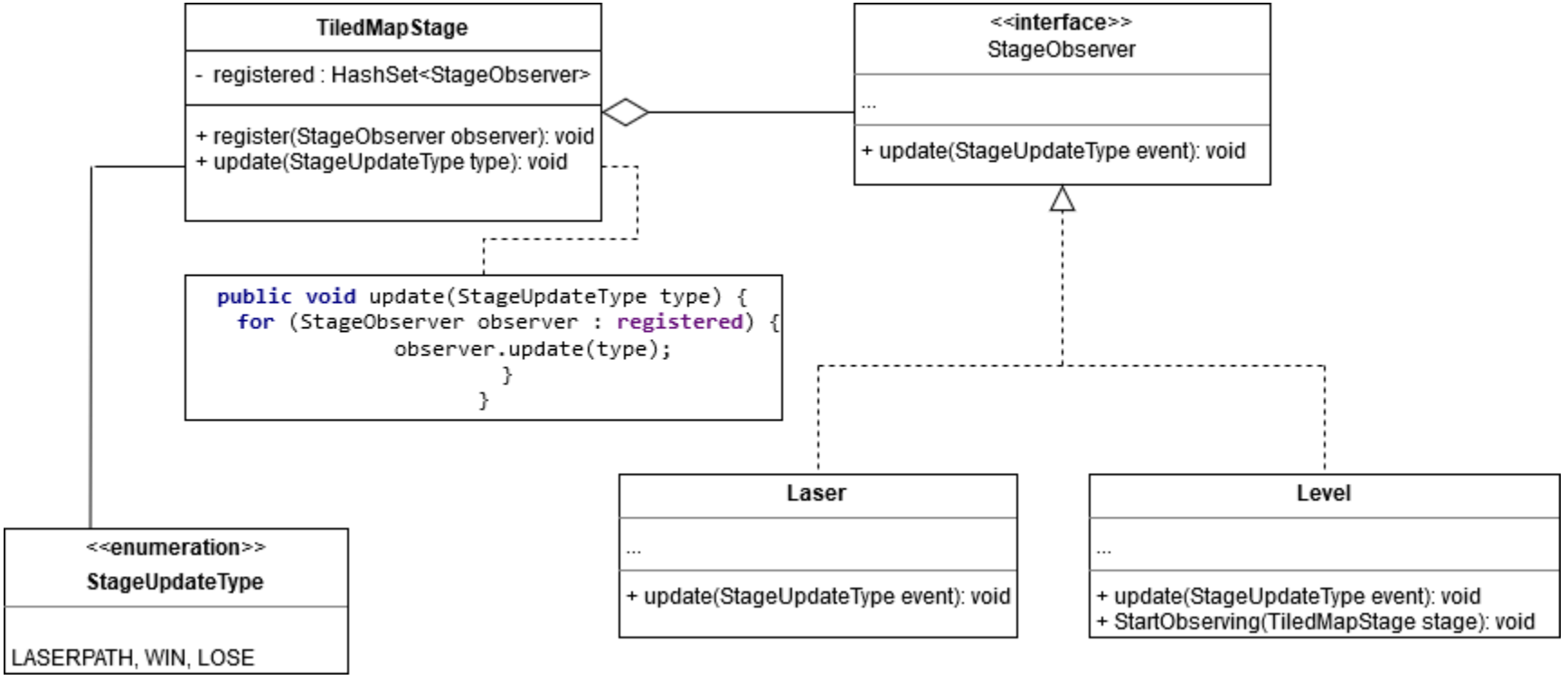
Exercise 1

First design pattern: State



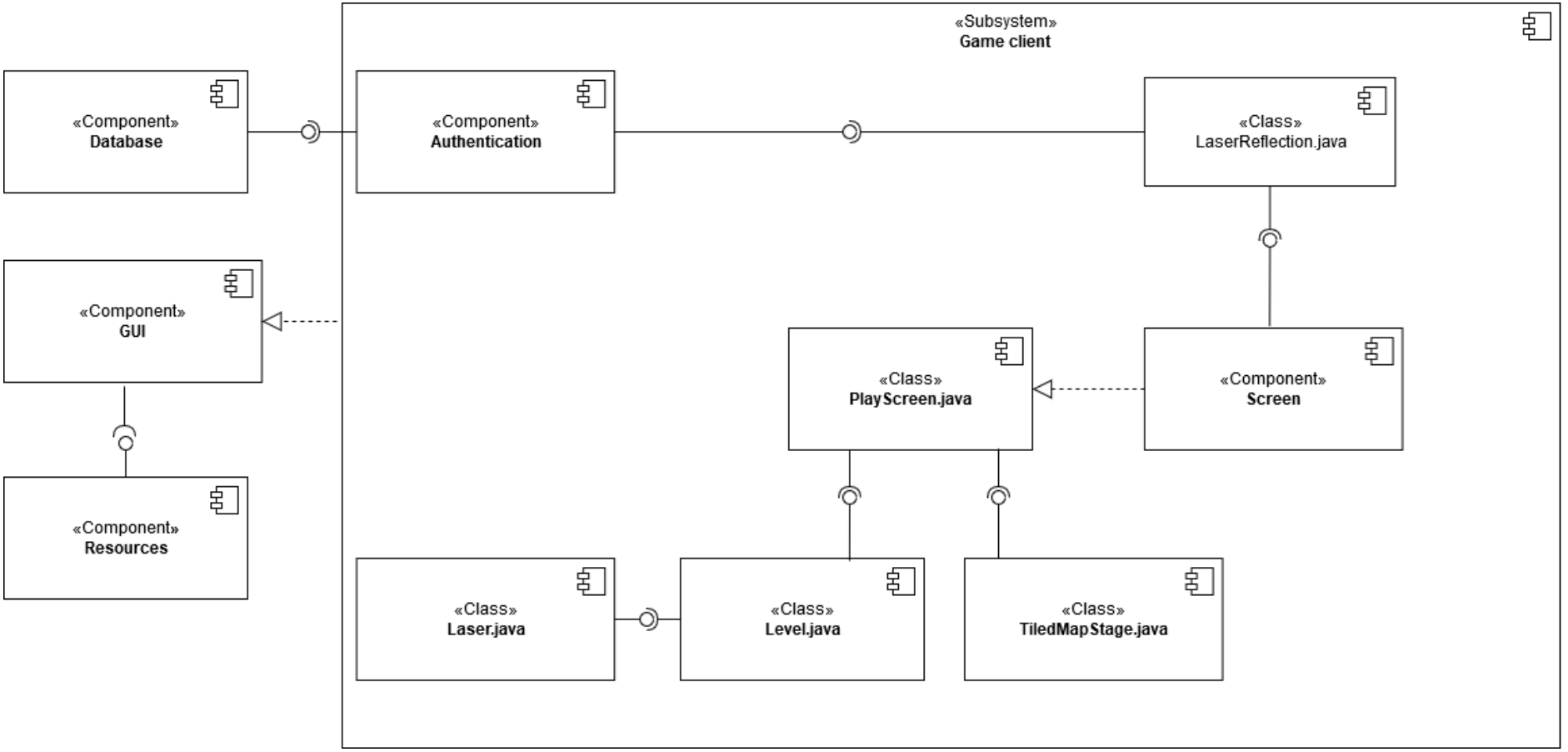
Description: The state pattern has been used to handle the logic related to the user input and a cell on the grid. Our game is divided into cells on a grid, every cell having a Tile of its own. The user can click on a cell and, depending on what Tile occupies this cell, the logic pertaining to this Tile will be executed. Firstly, how is this state pattern implemented? As we can see in the class diagram, every GridCell has an instance of a Tile (State). Whenever the clicked event gets triggered by user input, the GridCell will call the clicked method of its current Tile. The Tile interface has several implementations (States), which have a different implementation of their clicked method. The reason why we implemented this design pattern is that we needed to differentiate the behaviour of a GridCell depending on it's Tile. It furthermore avoids us having to use big if-else blocks and, consequently, make the code easier to maintain. If we want to add new tiles in the future, we can do so without having to change existing code. A good example of a state change would be when an empty tile is clicked. It then will change in to a mirror tile which uses different logic to handle further clicks.

Second design pattern: Observer



Description: The observer pattern has been used to facilitate event handling in our game. Our game has several events, such as placing a mirror, hitting a lamp or hitting a bomb. Any of these events can cause several changes in our game. Placing a mirror (or changing its rotation), can change the path of the laser if it is placed in line with the laser's trajectory. Hitting a lamp can cause the stage to be won and hitting a bomb will lose you the game. Firstly, how is this pattern implemented? If we look at the class diagram once again, we will see that our TiledMapStage keeps track of all observers registered to it. The TiledMapStage creates and keeps track of our stage. It is our subject in the observer pattern. The register method will register any observer in the set of the stage. The update method of the TiledMapStage will, for every observer registered, call their update method with an event type (StageUpdateType enumeration). Our concrete observers are at this point the Laser class and the Level class. The observers will all receive the update but will, depending on the event, react accordingly. For example, the Laser class is interested in LASERPATH events and will act upon these events while the Level class will be interested in knowing about WIN and LOSE events. The reason for implementing this design pattern is that we do want interaction between our observers and the subject, but we don't want strict coupling. We want the option to add other observers down the line without changing the code for the current subject and observers. With the observer pattern we can achieve this, by having new observers just implement the StageObserver interface.

Exercise 2


«Subsystem»
Game client
«Component»
Database
«Component»
Authentication
«Class»
LaserReflection.java
«Component»
GUI
«Class»
PlayScreen.java
«Component»
Screen
«Component»
Resources
«Class»
Laser.java
«Class»
Level.java
«Class»
TiledMapStage.java

Components:

Database: The database stores the information on our users. It stores the username and password, which we need for authentication and it stores the high score of a user. This high score is required on determining which levels a player can play.

Authentication: The authentication component requires information from the database to do the authenticating. This component provides the user with the possibility to login to his/her account. If the user does not have an account yet, then a possibility to register a new account is given to the user. After logging in, the user gets access to the rest of the system to play the game. If the user does not successfully login, then he/she does not get access to the game.

GUI: The GUI provides for the users the menus and visual feedback on their monitors. Through using the GUI it can go through the many menus and start a game. It can also log out of the game and exit the game if he/she wants to. In our diagram it shows that our entire game client depends on the GUI. Which makes sense, as our users require visual feedback.

Resources: Our game requires several resources. We chose to implement our game by splitting our grid into tiles. By using the program Tiled, we have created several levels which are saved as tmx files created through a spritesheet. Furthermore we use a skin to give our GUI a theme.

Screen: The game has several screens. The LaserReflection class, being the main point of our game, can assign to itself one screen at a time. When a screen gets assigned to the LaserReflection class, it will execute the logic that is accompanied with that screen. In that sense, the different screens in this component provide logic to accompany our GUI.

Classes:

LaserReflection.java: This class is the manager of our game, the main point. It holds the information for the current screen that is set. By having an entity above an actual instance of the screens, we can have multiple screens. When one wants to switch between screens, the only thing that happens is that the screen of LaserReflection.java gets set to the screen that we want to change to.

PlayScreen.java: This is one of the screens belonging to the Screens component. We highlight it because it plays a crucial role. On creation it will keep a list of Level objects with their own separate Laser object. Upon loading a Level it will create a TiledMapStage object for this specific Level. This means that this screen is basically the screen where the user will play the game on.

TiledMapStage.java: When the user starts a Level to play the game, the PlayScreen.java will have to setup a Level and a TiledMapStage instance for this specific Level. The Level class will create a Laser instance which is the manifestation of a laser in our game. It handles everything related to the laser

(trajectory, reflection etc.) The TiledMapStage handles the logic of mirrors and other objects in our level. It will assign to every cell in the grid of our level a Tile with an object in it. After doing this, it will attach a listener to every tile so that user input will be processed depending on what this tile is. This means that this class creates our grid of cells and after that keeps track of the logic happening in our stage.

We highlight these classes as we find them to be, together, crucial components of our architecture. Furthermore our architecture is a clear example of an MVC architecture. As also clearly can be seen from our diagram, we have a separation between our view (GUI), the model, which is responsible for handling the data (Database) and the controller, which responds to the user's input and handles the logic. Afterwards, the controller will pass the data back to the model. Our choice for an MVC architecture was motivated by several components. Firstly, we might want to develop this game on different platforms. The MVC architecture will helps us by making it possible to reuse code. We just have to cater a certain view for our designated platform. Furthermore, the development of our game was easier because of this separation. We split our group in several groups who all worked separately on either Model, View or Controller concurrently. These arguments were convincing enough for us to choose for this architecture.