ASSIGNMENT 3

DESIGN PATTERNS

CSE2115 Software Engineering Methods
Team 90

Made by:

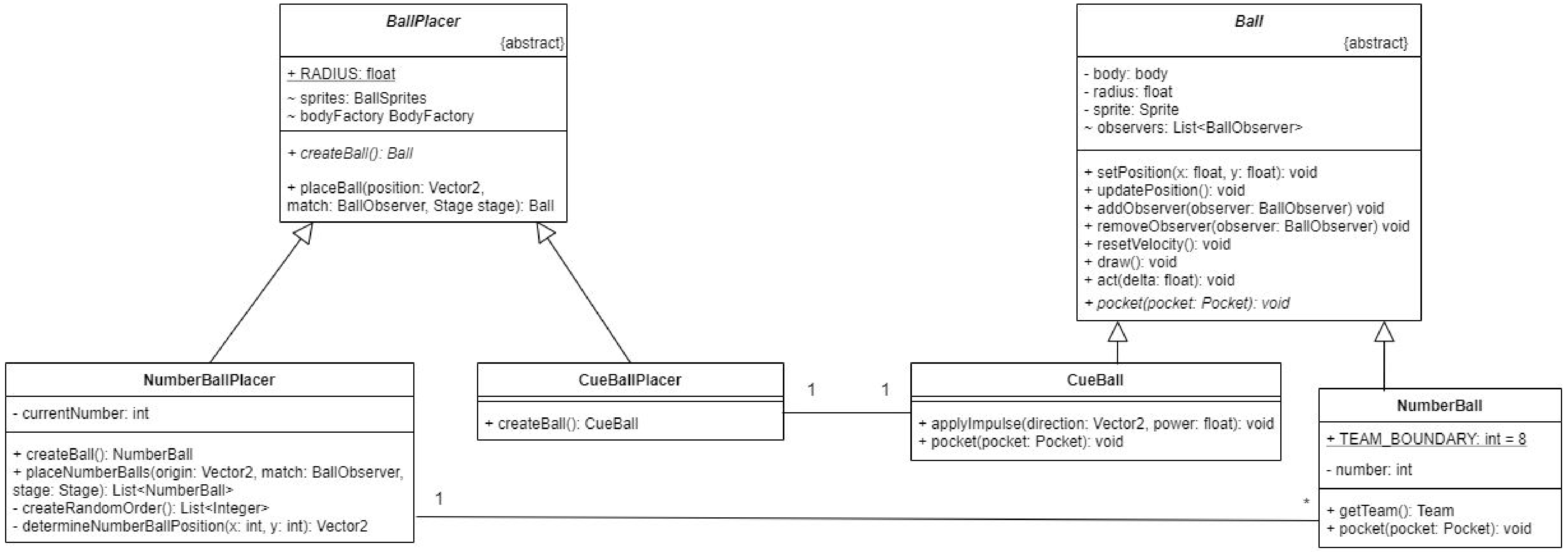
Victor Roest
Tim Numan
Maikel Houbaer
Jesse Nieland
Eduard Filip

CONTENTS

Exercise 1 - Design patterns

1. Factory method

We applied the Factory Method design pattern because we wanted the BallPlacer class to delegate responsibility for creating specific type of balls to several helper subclasses. Using this design pattern provides a way to implement subclasses and connect parallel class hierarchies. In the diagram you can see the class hierarchies are parallel and this is not only the case for the classes itself: also their test classes are implemented using a parallel class hierarchy. Furthermore, using this design pattern also allows us to add different types of balls in the future, for example for a special gamemode, without having the change code in already existing classes.
In the class diagram below, you can see how exactly we implemented this. Our 'factory method' is the method createBall, which is implemented by the subclasses who decide which specific class to instantiate. The operation done with the 'output' of this factory method is placeBall, which places the ball on the table, at a certain position, in the given match and on the right stage.


BallPlacer
{abstract}
+ RADIUS: float
~ sprites: BallSprites
~ bodyFactory BodyFactory
+ createBall(): Ball
+ placeBall(position: Vector2, match: BallObserver, Stage stage): Ball
Ball
{abstract}
- body: body
- radius: float
- sprite: Sprite
~ observers: List<BallObserver>
+ setPosition(x: float, y: float): void
+ updatePosition(): void
+ addObserver(observer: BallObserver) void
+ removeObserver(observer: BallObserver) void
+ resetVelocity(): void
+ draw(): void
+ act(delta: float): void
+ pocket(pocket: Pocket): void
NumberBallPlacer
- currentNumber: int
+ createBall(): NumberBall
+ placeNumberBalls(origin: Vector2, match: BallObserver, stage: Stage): List<NumberBall>
- createRandomOrder(): List<Integer>
- determineNumberBallPosition(x: int, y: int): Vector2
CueBallPlacer
+ createBall(): CueBall
1
1
CueBall
+ applyImpulse(direction: Vector2, power: float): void
+ pocket(pocket: Pocket): void
NumberBall
+ TEAM_BOUNDARY: int = 8
- number: int
+ getTeam(): Team
+ pocket(pocket: Pocket): void
1
*

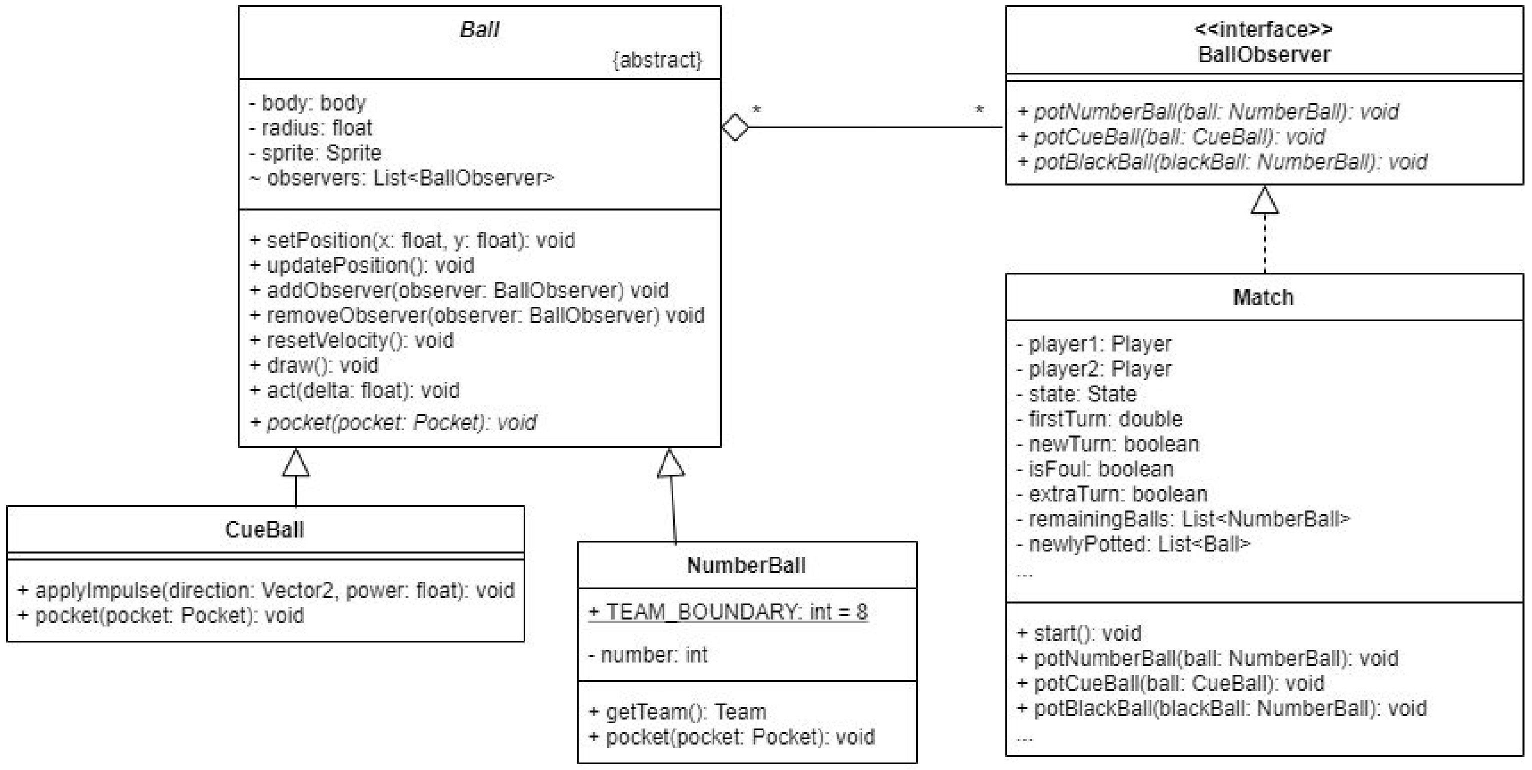
There are a few minor differences comparing this class diagram to the class diagram for this design pattern on the slides (07-Design Patterns - Part II: slide 48), but those are all easy to explain:

1. **Extra attributes & methods:** the picture on the slides is only used to explain the (theory behind) the design pattern, but in real life there are of course more attributes and methods than just the factoryMethod and 1 operation.
2. **Product is abstract class instead of interface:** Our ‘product’, the ball is an abstract class and not an interface. As explained on slide 47 of the same lecture, the product can be one of the two: an abstract class or an interface. We had to make it an abstract class, because it needs attributes and method implementations that are shared by the different kind of balls and it needs to extend a LibGDX class named Image, which is both not possible with an interface.

2. Observer

We chose to implement the Observer design pattern, because we strive for loosely coupled design between objects that interact within our game. In this case, these are the Match (observer) and the Ball (subject) objects. When the ball changes state, the match should be notified and updated automatically. This is very convenient, since this way the match doesn’t have to poll each ball over and over again to check its state, but can await a message of all the balls. Other advantages are that we can reuse our observers independently of each other, changes to (the code of) either the object or an observer will not affect the other and that we can add and remove observers at any time, which could be great for integration/system testing or (wild idea) spectating matches of others.

See the class diagram below for how exactly we implemented the design pattern in our code. Our subject (Ball) has a notify method (pocket) which notifies the ballObservers the ball has been pocketed. It calls one of the update methods of the BallObservers. We have three update methods, which all lead to a different reaction of the match: a method for potting a regular number ball (score it, determine possible foul and next turn), a method for potting the white ball (give foul and next turn) and finally a method for potting a special kind of number ball: the black ball (number 8). If this last update method is called, the match should end the game and determine the winner.


Ball
{abstract}
- body: body
- radius: float
- sprite: Sprite
~ observers: List<BallObserver>
+ setPosition(x: float, y: float): void
+ updatePosition(): void
+ addObserver(observer: BallObserver) void
+ removeObserver(observer: BallObserver) void
+ resetVelocity(): void
+ draw(): void
+ act(delta: float): void
+ pocket(pocket: Pocket): void
*
*
<<interface>>
BallObserver
+ potNumberBall(ball: NumberBall): void
+ potCueBall(ball: CueBall): void
+ potBlackBall(blackBall: NumberBall): void
CueBall
+ applyImpulse(direction: Vector2, power: float): void
+ pocket(pocket: Pocket): void
NumberBall
+ TEAM_BOUNDARY: int = 8
- number: int
+ getTeam(): Team
+ pocket(pocket: Pocket): void
Match
- player1: Player
- player2: Player
- state: State
- firstTurn: double
- newTurn: boolean
- isFoul: boolean
- extraTurn: boolean
- remainingBalls: List<NumberBall>
- newlyPotted: List<Ball>
...
+ start(): void
+ potNumberBall(ball: NumberBall): void
+ potCueBall(ball: CueBall): void
+ potBlackBall(blackBall: NumberBall): void
...

There are a few minor differences comparing this class diagram to the class diagram for this design pattern on the slides (06-Design Patterns - Part I: slide 44), but those are all easy to explain:

1. **Notify (pocket) method instead of update for subject:** On the slides there is literally one method, update, and both the observer and concrete subject have this method. This is a small mistake in the slides, for the subject this should actually be notify instead of update: like in slide 46 of the same lecture. The notify method we implement for the subject is pocket, which updates the observer by notifying the ball is potted.
2. **Extra attributes & methods:** the picture on the slides is only used to explain the (theory behind) the design pattern, but in real life there are of course more attributes and methods than just the notify and update method(s).
3. **Subject is abstract class instead of interface:** Our 'subject', the ball is an abstract class and not an interface. We had to make it an abstract class, because it needs attributes and method implementations that are shared by the different kind of balls and it needs to extend a LibGDX class named Image, which is both not possible with an interface.

Note: in the diagram we left out some attributes/methods of the Match class, because those are not related to the design pattern and having all wouldn't fit on the picture. To show this, we used '...', similar to what was done in the lecture slides. Everything needed for the design pattern is in the picture and is much more clear in this way.

Exercise 2 - Software Architecture

When we made our system requirements, we realized that, in order to keep it modular, we needed both a client and a server. We need the server to allow for a secure login and communication with the database, which stores the users and the scores. We didn't want the client to directly access the data in the database, mainly for security reasons, as having the db password on the client would be a huge security flaw. Therefore the client/server architectural pattern was the best choice for us. An alternative could be MVC, but we chose to not implement MVC because the complexity of the project wasn't high enough that this would offer any substantial benefits.
The client and the server are different processes that have well-designed interfaces and also play different roles. In our case, the client represents the game itself (more details in the next paragraph), while the server contains the services necessary to authorize the user and to store his score after playing a pool match. This architecture helps us have some key quality attributes: scalability, security, testability, reusability, maintainability and availability.

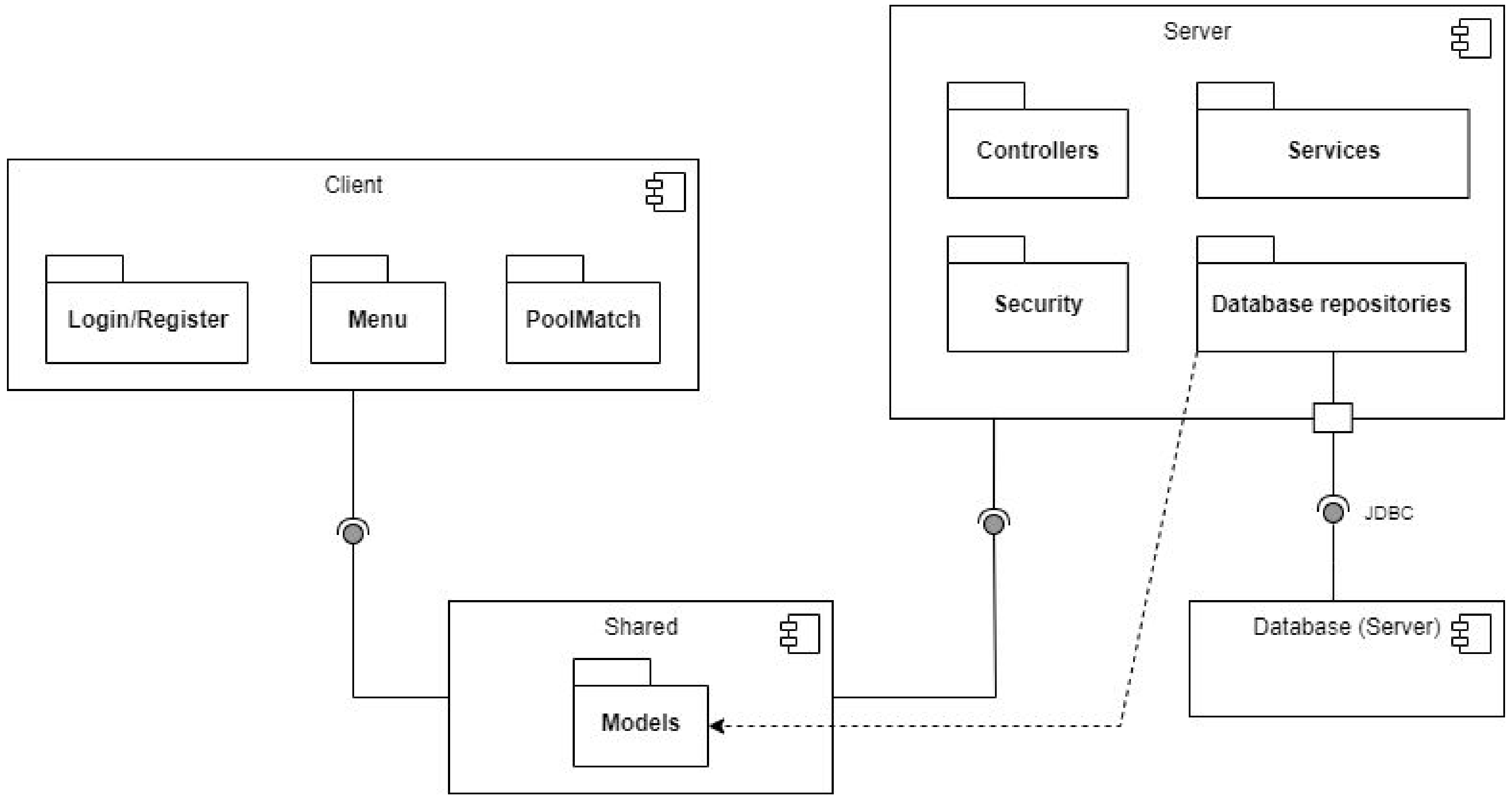
The client side can be easily divided into three sub-systems: Login/Register, Menu and Pool Match. It uses a 'Main program and subroutine' architectural pattern, as we have the main class 'DesktopLauncher' which launches the client with the GUI elements, and then the three subsystems which may have more divisions depending on their complexity.
The Login/Register subsystem is the one that first communicates with the server in order to create a new user or validate an existing one. We have two screens here: register and login. For register, we ask the user to enter a username and a password twice, after which we send a request to the server to create an account with the details he just provided. Then the user can login. If the user is already registered, he can immediately login without the need of registering again. Most part of this subsystem is represented by the GUI, as the logic only consists of sending some messages to the server when login or register buttons are clicked.
The menu subsystem is simplistic in order to make it user-friendly. After the user logs in, he has the menu from which he can choose to play a pool match, to see top scores, to tweak some game settings and to log out in order to either close the game or let someone else play. Here we have a lot of GUI classes which contain the necessary code in order to build the visual with which the user interacts. Since the backend of this subsystem mostly has to do with the user interacting with the visual part of the game, it was not necessary to make separate classes (controllers) that handle only that part.

The Pool Match subsystem is the most complex one, since this represents the pool game itself. In order to make it easy to develop and maintainable, we kept it as modular as possible:
Everything related to what the user sees and his interaction with the game itself is put the class ‘MatchScreen’, which can found in the gui (java) package. In this way, every issue that we experience visually (rendering the images of the colors) should be solved there.
All objects needed for a poolmatch (cue, balls, pockets etc.) are placed in the ‘gameobjects’ (java) package. Each object has a draw method and a act method which represent the visual part of that object. These methods are used in the render method of MatchScreen.
The other methods in each object shape the behaviour of them, like rotating the cue or applying an impulse to the white cue ball. If inheritance was possible in order to reduce duplicates (for instance making a cue ball and the number balls based on the abstract ball class), we implemented it.
Finally, the state of a pool match is monitored by the classes in the ‘matchstate’ (java) package. The most important class is ‘Match’ which stores the current state of the game and determines the turns following the rules and fouls. Next to that we have enumerations for State and Team and a WorldContactListener which listens to the collisions between balls. If a ball is potted, the match is notified by the ball itself, as described in exercise 1.2 about the observer design pattern.

The server side uses a ‘Main program and subroutines’ as well, since it is not really big and complex and enables modularity and reusability. The main program is called ‘Main’, which runs the server. The subsystems are the following: firstly we have the security classes which enable us to make a secure authentication using tokens and also not allow any random user to store scores or request info on other users. Then, the controllers, which contain the messages that are sent between the client and the server such as saving a new user and sending the top scores. Thirdly, we have the database repositories, which represent the transition between the higher level structure (our class objects) and lower-level database (in our case the queries), allowing us to store data in the database. The database and database repositories communicate with each other via the JDBC interface. Finally, we have a service part, which handles how the details of the user are handled by the server.

Next to a module for the client and the server, we also have a ‘shared’ module which contains the models used for storing data in the database as well as communication between the client and the server (login data, scores). Having them in the shared module prevents code duplication, while allowing the client and server to use the same classes for a stable and clean communication.

On the next page, you can see the component diagram that describes our architecture.



Note: these packages in the UML component diagram do not necessarily correspond to actual Java packages but "UML packages": a collection of logically related UML elements. Java packages are not grouped on logic, but on the functionality/type of the classes (GUI, game objects, matchstate etc.).