A Game Engine for AI Competitions

by

Alexander Schmitz

Matriculation Number: 394540
Degree: Information Systems Management (Wirtschaftsinformatik)

Bachelor Thesis

Technische Universität Berlin
Faculty IV - Electrical Engineering and Computer Science
AOT
Distributed Artificial Intelligence Laboratory

February 21, 2022

Supervisor:
Dr.- Ing. Stefan Fricke

Contents

2.3 The Java Persistence API

Applications need to store persistent data if they want to access the same objects over a long period of time. Persistent data is stored in non-volatile memory and can be accessed throughout multiple executions of the application [7]. That also prevents data loss in case of a crash and restart of the system. As there is no need to store all data persistently, there is transient data which can only be accessed during the runtime of an application and is lost after it terminates.

Object-relational mapping (ORM) is a technique to convert data from the object-oriented approach (OOA) into other data formats, e.g. relation data models [8]. In objects, data is often stored as non primitive types where as the tables in a database require primitive types (int, varchar, etc.) to store data. ORM provides a layer in between to automate the conversion and relieve the developer from doing so.

The Java Persistence API (JPA) is a technology to manage the persistence of objects and the object-relational mapping in a Java application. JPA specifies a standard to use annotations or XML files for the mapping of objects to a relational model. The JPA allows for more simplicity in the persistence of entities by taking a notation-driven approach. **Entities** can be simple Java objects which receive the *@Entity* annotation. They don't need to implement any additional interfaces or methods. The persistence of the attributes of the objects is then handled by the JPA automatically, where possible. **Attributes** can be mapped individually by the *@Column* annotation, or be declared *@Transient* if they should not be mapped to the database. Additionally attributes can receive annotations specifying the multiplicity (one to many, many to many, etc.) or other relationship characteristics.

3 Design

The use cases for our application can be split up in two phases. The first one is during the implementation of their game. In this phase the developer defines the rules and custom behaviour needed for the execution of their game. During the runtime the perspective on the user changes. Now the user will connect to our application to play the game they implemented in the framework. Similarly the requirements for this project are split up in two phases and are separated in the requirements during development and requirements during runtime.

3.1 Development requirements

In this section we want to define the requirements for the development phase. We consider the development phase, the phase during which a developer defines the rules for his game. This includes defining the starting state, a method to check if the game reached an end state and a method which defines legal transitions. Transitions in the context of a game can also be called moves, while the state of a game can be the positions of the pieces on a chess board for example.

As a framework this application should support any functionality which isn't directly related to the game while allowing flexibility if custom behaviour is desired. Relevant

functions for this application are:

- The communication between server and client
- Serializing and Deserializing the game state
- Creating persistent game history
- Implementing a new game

The communication between client and server has to be platform independent. The client can be programmed in any programming language, as long as it supports HTTP Websocket communication. Furthermore the connection has to be reliable, as to not disconnect during the execution of a critical use case. Disconnects in the context of a game execution can result in time loss for a player or losing a game by time-out. This has to be prevented.

To further guarantee a platform independent communication, the state of a game, or the request by the clients have to be serialized and deserialized. This process is handled by the framework rather than by the developer. However if necessary, the serialization behaviour can be overwritten. This can be useful if the game requires a more complicated state which has to be send to the clients or if clients receive incomplete information about the game state.

To allow machine learning possibilities or general data collection, the game history has to be stored persistently. The game history includes necessary information about a completed game and especially a sequence of game state (or sequence of moves) as well as the outcome of the game.

Finally the developer has to have the capability of implementing his game into the application. In the simplest case, a developer would have to define the rules of his game, manage the state of his game and define the end of his game. In more complex games or if more complex behaviour is required, the developer has the options to define custom serialization behaviour, custom turn behaviour and custom notifications.

3.2 Runtime requirements

After the development phase, users can now communicate with the server and interact with it. They have a set of methods to accomplish the exchange of data.

- Create a new game instance
- Join a game instance
- Receive updates when the state of the game changes
- Send moves to the server to change the state of the game
- Get information about existing games and completed games
- For a completed game, get the sequence of moves

The process of creating a new game, requires the client to send the game type. The game type can be any game implemented on the server. Additionally the start state of the created game can be set, to play a game from a custom starting point. This allows more troubleshooting for the AIs and more interesting states can be simulated. Finally the client can decide whether he wants to play an opponent or the MCTS bot on the server. After the creation, the client will receive a confirmation about the creation and meta data about the game created.

To join a game the client has to specify the game ID of the game he wants to join. After joining he will receive a confirmation if the action was successful or an error if not.

After a client has joined a game, they are subscribed to any updates about the game state. The first update is received once the game is full, i.e. the minimum amount of players has joined. At this point the game is started and the clients will receive updates about any changes in the game state.

Once a game has started, all subscribed clients (i.e. players) can send messages to the server with a move they want to perform. A move is a transition from one game state into another game state. Whether a move results in a change of the game state depends on the developer. They need to decide if the move is legal which includes checking if the correct player tried to move and if the move results in a legal state.

The client can request data about current and completed games. Current games are games which are either currently running or which haven't yet started. A client can observe current games and receive the same updates about state changes as the players would, but an observer is not allowed to make a move. For a game that hasn't started, the client can join this game and become a player. Completed games are games whose state is in a final state.

Figure 3.1 shows a deployment diagram of the application. It describes the necessary components and their interaction. It shows the interaction between software and hardware and additionally shows the distribution of tasks. Any game related logic is managed in the game engine, which handles state changes, persistence and notifying the players. The communication between clients and the game engine runs through the websocket endpoint which manages the authentication process of registered players.

3.3 Additional requirements

Central storage of collected data in a database system. For the realisation of the data storage requirement, usually a central database system is necessary to simplify the data access. Additionally storing data in a single location helps with data retrieval and querying specific data.

The application is required to run on a standard web server. The application is supposed to run on internal servers but should also be able to be hosted on other web servers.

The framework should be easily understandable and extendable. To make the application more understandable there need to be intuitive interfaces and a simple communication protocol. Additionally examples and guidelines are desirable to help the future developer in creating their game. The amount of coding performed by a developer should be reduced to a minimum to save time and effort. Additionally, hidden functionality to the developer should not produce incoherent or inconsistent results. To make the application more

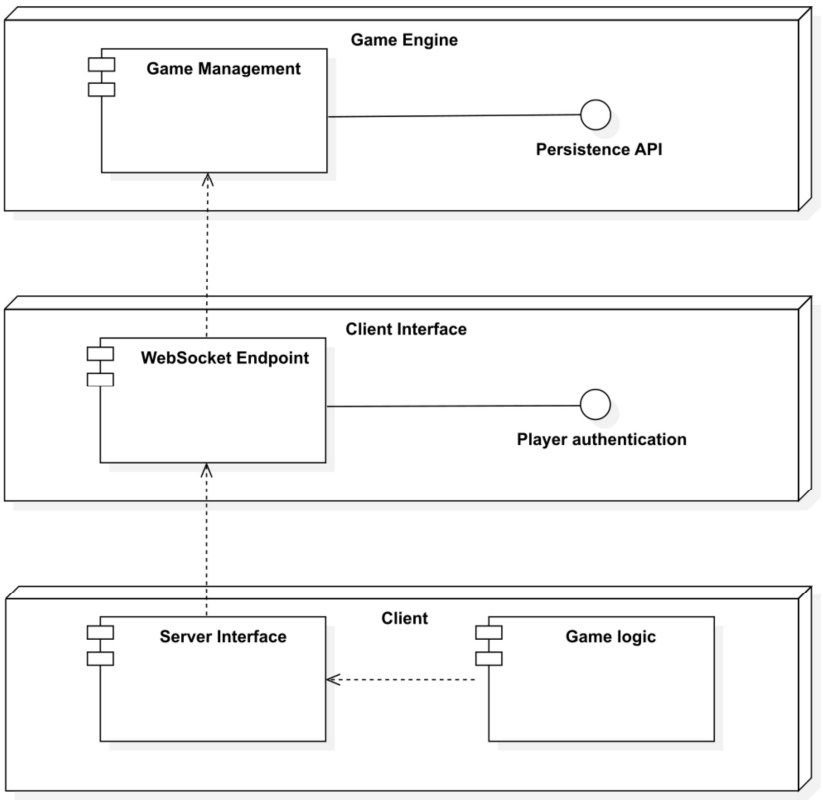

Figure 3.1: Deployment diagram of the game server

extendable, the use of easy to use interfaces and patterns is required. Patterns should be common patterns and interfaces should be well explained.

3.4 Limitations

As mentioned in Section 2.1 the concept of a game is rather abstract and creating a game engine for this abstract concept is impractical if not impossible. Therefore we pose some restrictions to the definition, which are further explained in the upcoming section about the choice for restricted game design 3.7.3. This restriction poses some limitations as to what this application was meant to do.

The focus is on turn based games with certainty and complete information. However uncertain games can be created without any restrictions on the core functionality. Only the quality of the MCTS will suffer under uncertainty. Additionally games without complete information can be implemented but require more care of the developer. This framework was not meant for real time applications and the behaviour in such cases is uncertain.

Furthermore the framework does not include any visualisation tools, as this is the task of the developer. The abstract nature of the game does not allow for the creation of a visualisation tool and can only be done once a concrete game has been implemented.

3.5 Use Case Analysis

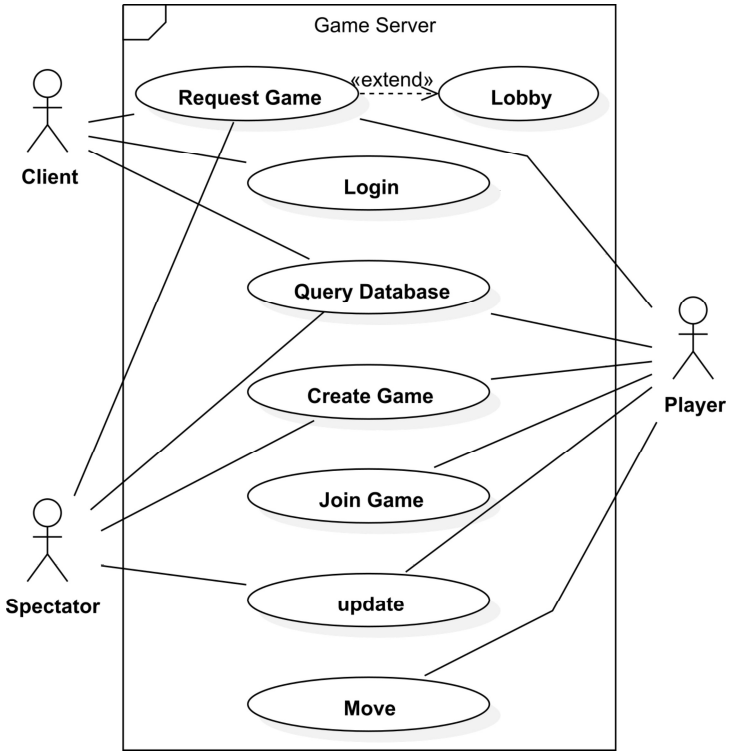

Figure 3.2: A high level use case diagram showing the scenarios for the interaction between client and server

Based on the requirements from section 3 we can create use cases for the client during the run time of the server. Use case diagrams show the functionality of the system and the interaction with the clients. They present a high level analysis of the requirements from the viewpoint of a user without going into details.

In figure 3.2 we can see 3 different actors which can interact with the server. The request they can make are described in the requirements but they will be formerly described in the section about the low-level design. A client has restricted access to the request they can perform. However through the login they can be "promoted" to either player or spectator. Whether they are considered player or spectator depends on the role they take in a game instance.

The methods described in the use case diagram come with certain conditions. First an actor has to have a connection with the server. This can be a connection through the internet or a local connection.

3.6 Architectural Design

In this section we would like to show the aspects of the software architecture in this project in regard to a object-oriented approach. With this we can document an important milestone in the implementation process and create a detailed concept of the framework.

The process of creating a framework for a game engine can be challenging but it can be simplified by taking advantage of design principles. The first tool we will use is splitting up the process in different design levels, starting from the most abstract level down to the most detailed one.

The architectural design level is the most abstract level of the design phase. It describes the system as a whole and focuses on the environment for the system more than the modules which the system contains.

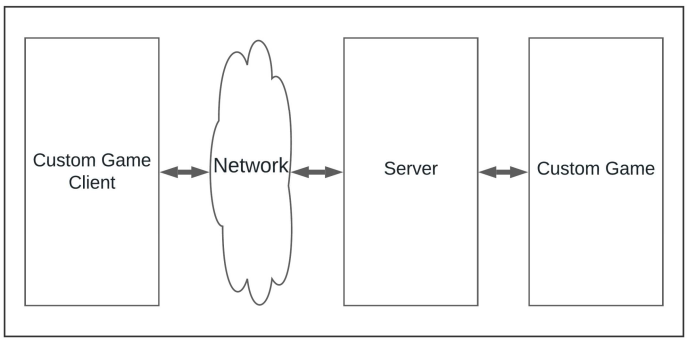

Figure 3.3: The high level architecture of the game server

As shown in Figure 3.3 the high level design is a simple client-server communication. The server provides services, in our case game state management, player management, game persistence, etc. to the clients which can access these services through requests. The client can be any application using the HTTP Websocket protocol and knowing the communication standard of the server. The server is the central structure to store and manage data remotely. Having a central structure to manage game states, the clients don't need to store or manage the state themselves. On a request they can access the state and can be sure that the state is true. The server allows multiple client connections and through the server they then can indirectly communicate. This does not mean, that clients can send messages to the server who then relays this message to another client. Rather a client can change data in the server through a request and this change can then be perceived by other clients. In our case, a request to change the game state by one client triggers a message to other clients.

3.6.1 Observer Pattern

During the programming process, it is very common to encounter problems which other people have solved before. Therefore it is useful to define the problem and use provided solutions in the form of software patterns. The observer pattern proved to be especially interesting for our task, as it creates a good solution for the game interface. The observer pattern, also known as the pub-sub (publish and subscribe) pattern is a behavioural object-oriented design pattern defined by IBM researchers in [9]. In it there are two important classes: the subject and the observer. The subject maintains a list of observers and can trigger a function to notify them about any state changes. Any observer can register themselves to the subject during runtime and can receive updates. The subject

does not have to know any functionality of the observers only that they exist and that they have an *update()* method.

Whenever a player changes the state of a game, the game will notify any registered observers about the changes. This includes other players, who can then calculate their next move to the update. Other observers are: clients which spectate a game, i.e. spectators, or the persistence interface which stores the game states. With the observer interface, the developer can create other functionalities which attach to a game.

3.6.2 User Interface

The communication between clients and server has to be bidirectional. The context of multiplayer games requires a back and forth of messages from the client to the server requesting information or state changes and from the server to the client in form of request fulfills or to push game state updates. Additionally the communication protocol has to be platform independent. The user interface of the application server uses the WebSocket protocol as it fulfils both requirements.

The user interface will handle any communication related logic, such as the parsing of incoming messages and the deserialization of outgoing messages. The interface then forwards the requests to the appropriate classes handling either the persistence logic, game logic or the game management.

To accomplish the parsing of the messages send by the client and appropriate parser should be used. Desirably the used library automates as much as possible of the serialization and deserialization process. Once the messages are parsed, packet objects representing the client requests can be created which are simpler to handle in a Java application. The parsing step also accomplishes the syntax checking of the send messages before any other method tries to access the unparsed message.

3.7 Detailed design

In figure 3.4 the use cases *login, create game, update* and *move* are described, specifying the order of the messages. We assume that after the login, each message from the client to the server contains authentication and that the Websocket Endpoint checks the authentication with the lobby manager. Additionally any message from the client is not written as function with arguments but rather describes the purpose of the message. The message has to be serialized by the Websocket Endpoint first before the server is able to fulfill the request. Any message described with a function containing arguments are method calls in between modules in the server or if targeted towards the client they are parsed Java objects. The condition for the loop to stop is that the game has either been aborted or has reached a final state. The loop starts with the outgoing update message rather than an incoming message because the first update is send after the game has started. Only then the client is supposed to send their moves.

Instead of the creation of a new game instance, the client could request information about the existing games. They could then choose one of the games to join.

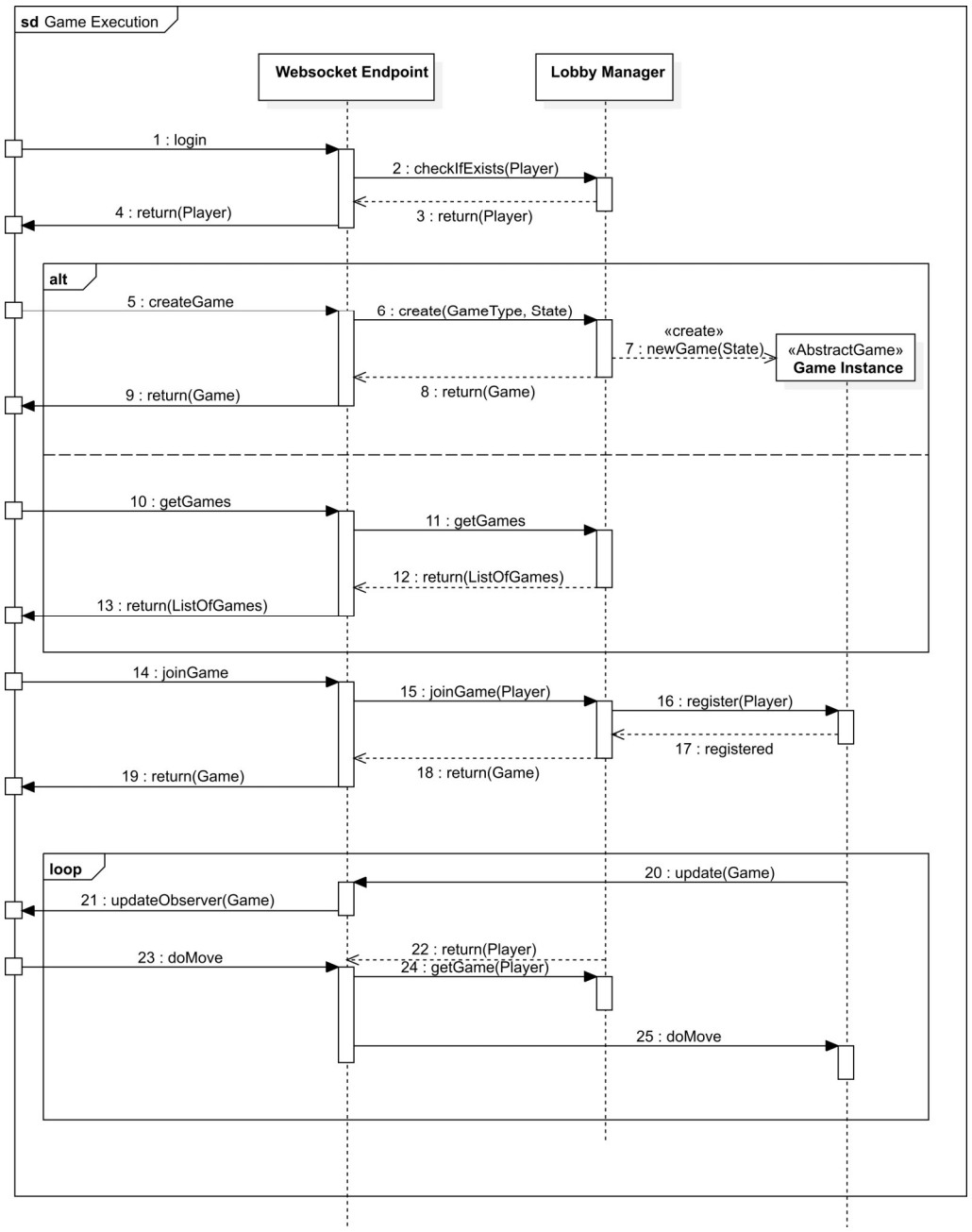

Figure 3.4: A typical interaction between server and clients during runtime

3.7.1 Messages

During the runtime, the server and client communicate through the websocket protocol with messages. These messages contain a description of the purpose of the message as well as the content needed to execute the desired task. The client has following set of methods which he can request the server to perform:

- **Login** The first interaction with the server should be a login message. The client needs to send a username which is not in use by another player. The server then

sends a success or error message informing the client whether or not the request was successful. In case of a success, the client will receive a unique id which will serve as authentication for future messages. In most of the following messages, the client needs to include the set of name and id if he wants to perform a request.

- **Lobby** Any client can request the status of the lobby. The server will reply with the set of active games in the lobby.

- **Create** A client can request to create a new game instance. On creation, the server replies with a success message containing the id of the newly created game. Any other client can now see the game in the lobby and join it as long as it is not full. A creation message from the client has to contain the correct header and in addition information about the game they want to create. This information has to contain at least the name of the game type they want to create.

- **Join** To join a game instance, the client has to send the game id of the game he wants to join and his authentication. The websocket interface relays this request to the lobby manager which returns the game instance for the given id. If the game is not already full, the player is added to the set of active players and the client is notified.

- **Move** During the runtime of a game instance, clients can send their moves to the server. The lobby manager first checks again if the client is registered as a player and if the player is registered as an active player in the game. The move is then send to the game instance, which can either execute the move and then update the registered observers or throw an Exception if the move was not legal. In the later case, the player that requested the illegal move is notified.

- **Database** The database request gives access to information about completed games. The request can contain additional query elements specifying the type of the game or the number of games they want to request.

The login is necessary to perform any game related request in future. By sending the set of username and id the server is able to identify a player and grant him access to specific tasks. When a player wants to join a game, the server checks if the player is already registered for this game. Then during the runtime of a game, the server checks if the correct player requests to make a move. This excludes on the one hand players which are not even registered for the game as well as players who are registered but not allowed to move.

Although the username and session would be enough to uniquely identify the client as an existing player, we need to add the id in the authentication process. When a client disconnects and reconnects, the Java Websocket API assigns a new session to this client and they would not be identifiable to the server by the username. The username is part of the game object and visible to to other players. The id therefore serves as password to the client if they want to login after a disconnect.

3.7.2 Lobby management

In the lobby, players can come together to create and join games. The lobby manages several aspects of the business logic of the server. It forms a layer between the communication between the user and the game instances and stores information about the players and the games. Request for creating or joining a game and request about information about games are handled by the lobby manager.

During the login procedure the lobby manager stores information about the players username, the player id and the websocket session. For any future request, the lobby manager provides methods to check the users authentication. During the development phase, when a developer creates a new game class they register the game class at the manager. During the runtime a user can then create a new instance of this game with an appropriate request.

3.7.3 Game definition

As explained in Section 2.1 defining a game is rather difficult. For the sake of this application, we have to create our own definition of what is acceptable as a game. We want to create the abstract idea of a game in a program therefore we have to specify the attributes a game needs. Looking at the definition by Salen and Zimmermann we can adapt three main ideas of what a game needs to be considered a game:

- Games have "a system in which players engage in artificial conflict". To keep track of the system we can store the state of the system. The state of a system is an abstract concept which defines all relevant game aspects. This can be the pieces on a chess board or the cards in each players hand during Poker.

- Games are "defined by rules". Rules define what is legal in the context of a game. For chess they define the movement of the pieces for example.

- Games result "in a quantifiable outcome". This requires a game to have an end or an end state. After the outcome the game is over and neither player can continue playing.

These three ingredients: the state, the rule set and the outcome are essential for any game implemented in this framework. The task of the future developer is to keep track of the current state of his game and define the moves any player can make to transition to the next state. Finally the game needs to define when a final state is reached and what to do with the outcome.

To enforce this behaviour we create an abstract class *AbstractGame* which has to be extended by the developer when he implements his custom game.

3.7.4 MCTS

For a player who is unfamiliar with the game he is about to play on the server or if a user wants to test their AI against a benchmark, they have the possibility to play against a Bot. Creating an AI for an abstract game is a challenge and most of the algorithms

commonly used in AIs depend on a well defined game logic to come up with a move. A heurisitic function evaluating the state of a game is a common tool for algorithms like mini-max or alpha-beta search. Based on the value of each game state they can choose an optimal move to make.

We do not want to create the burden of defining complicated algorithms if the user wants to play against an AI. Also we cannot assume that the developer has enough knowledge about their game to define complicated heuristics. Therefore we need an AI which can make an educated guess about a good next move without the ability to evaluate the state this move leads to. In the following part we will describe the process of using Monte Carlo Tree Search (MCTS) [10] to create a capable AI. Fortunately MCTS does not require an evaluation function for a given state of the game. It is a best-first search based on the random exploration of the game tree. By using the results of previous explorations, it can successively find better moves. Fundamentally MCTS only needs to know three things about a game to find the next move.

- The current state of the game
- The set of transitions (moves) from the current state which result in legal states
- If the current state is a final state

Two of these points are already defined by the developer when he defines the rules for his game. First, the MCTS Bot will receive the current state of the game whenever it is its turn to move. Second, each game has to have a function which checks if the current state is a final state. This function can then also be used by the Bot. Therefore the developer only need to define one additional method which returns a set of all legal moves.

MCTS traverses the game tree through the nodes. Each node represents a state of the game and are formed during the search algorithm. A node i has following attributes:

- The game state
- A parent Node
- A collection of child Nodes
- A counter for the visits v_i
- A counter for the wins w_i

Furthermore the MCTS algorithm is made up from 4 phases which it loops through.

Selection During the selection the tree is traversed from the root node and the most promising child node is selected through the UCT-value of this child. The UCT (Upper Confidence bound applied to Trees) formula evaluates the trade-off between exploration and exploitation of a given node.

$$UCT_i = \frac{w_i}{v_i} + C * \sqrt{\frac{\ln n}{v_i}} \tag{0.1}$$

With C is a constant to adjust the amount of exploration, usually $C = \sqrt{2}$ and n is the amount of visits of the parent Node.

Expansion In the expansion process, a new child Node is added to the promising child Node of the selection process.

Simulation During the simulation process we simulate a random sequence of game states until we reach a leaf node, i.e. a final state of the game, starting from the child Node from the expansion step.

Backpropagation The backpropagation starts by evaluating the final state reached in the simulation process. The evaluation only takes in account whether or not the game ended in a draw or if not, the winner. If it ended in a win, the new childs wins counter and visit counter gets updated, aswell as every parent Node of it. If the game did not end in a win, only the visits counter gets increased.

This process can be theoretically repeated infinitely, but one main advantage of the MCTS is that it can also be stopped at any point in the process and return the best move it found till now.

4 Implementation

In this section we will discuss the implementation process of the project. During the process we take in account the design choices from the previous section and elaborate the technologies used to accomplish the required functionalities. We will also discuss the chosen ones and compare them to other available technologies.

4.1 Observer pattern

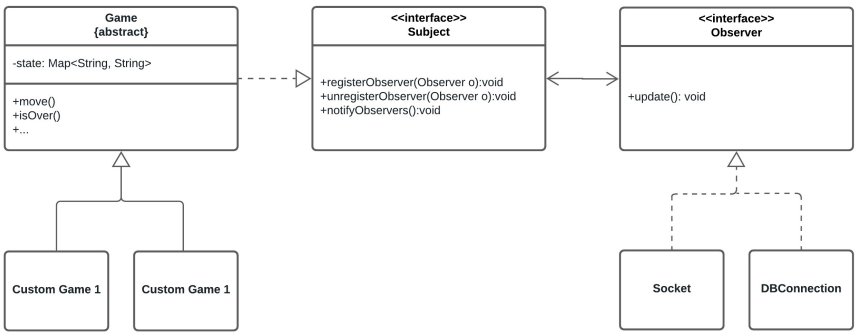

Figure 4.1: The software design pattern used for the pub-sub implementation

The observer pattern requires two interfaces: the observer and the subject. The *java.util* library provides an existing *Observer* interface and classes implementing this interface have to implement the *update(Observable o, Object arg)* method. Usually this interface would accomplish the desired tasks for this application, however we will need additional methods in our case. There is a special case in our application in which we don't want to broadcast to all the observers. We therefore created our own adaptation of the *Observer*

interface with the possibility to update a subset of observers. This goes against the desired behaviour of the pattern, in which the subject does not know about the observers during the runtime. It is however useful in a certain scenario which we will illustrate in the example in the next paragraph.

In this application, the only subjects are game instances. For certain game implementations, it is desirable to not update each player about the change of a state. For example in a 2 vs. 2 player game where two players need to cooperate against the other team, the cooperation could remain secret from the other team in a turn. Once a turn is completed every observer can then be informed about any changes. Therefore the interface defines two additional methods:

- `public void notifyObservers(List<Observer> o)`
- `public void notifyObserver(Observer o)`

The extension of the observer pattern transforms the pattern into a direct communication between certain observers. The logic about the communication can be modified by the developer when he defines the rules for his game or he can choose to ignore it and use the normal observer pattern, which should cover most use cases.

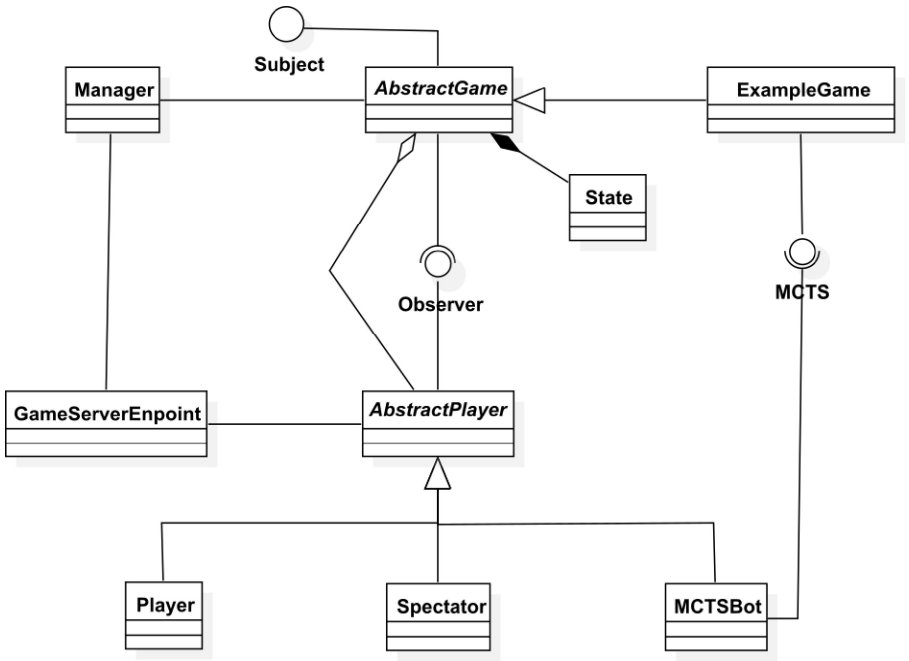

Figure 4.2: A reduced class diagram of the most important classes in the application

Figure 4.2 shows a class diagram of the most important classes in the game server. A more complete Diagram can be found in the Appendix.

4.2 Game Interface

The game interface serves as exchange point for requesting state changes and getting updates about state changes. As the class diagram in figure 6.1 suggests, the class

`AbstractGame` is a central piece of the application. It takes the role of an observable in the observer pattern by implementing the `Subject` interface. Additionally it implements the `Cloneable` interface from the `java.lang` library and the `JsonSerializer` from the `com.google.gson` library. The `Cloneable` interface is only required by the MCTS Bot as to not change the game state of the actual game instance when performing the tree search. More on this will be covered in the section about the MCTS Bot 4.5. The other two interfaces will be covered below.

4.2.1 Subject

In the role of the subject in the observer pattern, the `AbstractGame` defines the required methods as shown in the following listing.

```
1  public void registerObserver(Observer o) {
2      observerList.add(o);
3  }
4
5  public void unregisterObserver(Observer o) throws Exception {
6      if (!observerList.remove(o)) {
7          throw new Exception("Observer not in ObserverList");
8      }
9  }
10
11 public void notifyAllObservers() {
12     for (Observer o: observerList){
13         o.update(this);
14     }
15 }
```

Listing 1: Implementation of the Observer Interface

4.2.2 Serialization

The `AbstractGame` class comes with an already implemented serialization method. The serialization parses the Java Object into a corresponding Json representation which is send to the clients through the Websocket. The `com.google.gson` library is one of the most popular Json libraries. It is an open-source developed library which can handle the serialization of complex data types and generics in Java. Another popular library is `com.fasterxml.jackson` which offers similar functionalities and performance and is included in all *JAX-RS* applications and the *Spring* framework. The choice for the `gson` library was taken as it generally is easier to use with just two methods: `toJson()/fromJson()` while allowing simple customisability if desired.

At first we used the standard deserialization method of the *gson* library. For this we create a new *Gson* object `Gson gson = new Gson()` which can then be used to transform a game object such as the `TicTacToeExample ttt` into a json `String`: `String jsonGame = gson.toJson(ttt)`. However since Java 9 the automatic deserialization can cause errors. The *gson* library uses Java reflection, which is an API to modify internal properties

of a program. "For example, it's possible for a Java class to obtain the names of all its members and display them." [11] Before Java 9 reflection could be used to make private fields public which allows *gson* to access these fields and put them into the json object. Now we can no longer use the reflection to access these fields, as it throws a `java.lang.reflect.InaccessibleObjectException`. The `JsonSerializer` interface is used to serialize the relevant fields of the game object. In the method `public JsonElement serialize(AbstractGame src, Type typeOfSrc, JsonSerializationContext context)` we can define the attributes of the game object which should be serialized into a `JsonElement`. This includes the meta data such as the maximum number of players, the id of the game and the active players. Fortunately the *gson* library still allows the automatic serialization of a *Map*, notably the game state, into a Json object.

It is possible that the reflection can cause exception for a future developer depending on the objects they store in the game state. Therefore they can overwrite the existing `serialize()` method.

4.2.3 Game logic

The `AbstractGame` class provides the abstract game logic to the application. The abstract game logic is behaviour which all games, for which this application was designed, have in common. The specific game logic is defined by the developer when they create their game type. To accomplish the abstract behaviour of the game, the `AbstractGame` class provides several functions:

- `executeMove(AbstractPlayer player, String move)` is called when the game is required to complete a turn. This includes calling the abstract method `move(String move)` itself, ending the turn with `endTurn()`, saving the `move` in the history and checking whether or not the game is over with `isOver()`. If it is over the game is send to the persistence controller to be stored in the database. The listing **??** shows the behaviour of this method in detail.

- `endTurn()` This method rotates through the player list. For example the player list has three players in the order [A, B, C]. Player A just finished their move, then player B is assigned as current player. Player B then finishes their move, then player C becomes the current player etc.

- `isOver()` and `isDraw()` are getters for the `boolean` attributes `over` and `draw` once they are set to `true`. Otherwise these methods will call the abstract methods `checkIfOver()` and `checkIfDraw()` to check if the game has reached a final state. The two abstract methods have to implemented by the developer.

- Additional methods which can be classified as abstract game logic manage the player list. This includes adding and removing players with `public void addActivePlayer( AbstractPlayer player)throws MaximumPlayerNumberExceededException` and `public void removeActivePlayer(Player player)`.

```
1  public abstract void move(AbstractPlayer player, String move)
       throws IllegalMoveException;
2
3  public void executeMove(AbstractPlayer player, String move) throws
       IllegalMoveException{
4      try {
5          move(player, move);
6          moveHistory.add(move);
7          endTurn();
8          notifyAllObservers();
9          if (isOver()) {
10             Controller controller = Controller.getController();
11             controller.persistGame(this);
12         }
13     } catch (IllegalMoveException e) {
14         throw(e);
15     }
16 };
```

Listing 2: The move turnus in the AbstractGame class

4.2.4 Expandability

In this part we want to discuss some choices that were made in the implementation of the Game Interface which grant the developer additional freedom in their game design. With the standard serialization of the game, only the `State` object is send to the observers. Therefore if the game needs contain private information, additional attributes can be defined in the game implementation. This allows for games with imperfect information. Any information that should be send to the observers can be stored in the `State` object.

Theoretically games which do not classify as turn based games can be implemented, although this has not been thoroughly tested as it is not the main focus of this project. The methods from the `AbstractGame` class can all be overwritten. In this case the methods `executeMove()` or `endTurn()` can be altered to not end turns. The next step would be to ignore the `currentPlayer` in the rules and allow any player to make a move at any point in time.

4.3 Entity Management

In the previous section we discussed classes related to the game logic. In this section we will discuss the implementation of the game management. This includes storing a list of currently running games, i.e. the lobby, creating games, keeping a list of available game types and observer management. Most of the game management logic is done trough methods in the `Manager` class. This class was implemented according to the *Singleton* pattern.

The *Singleton* pattern is a creational pattern where a given class is only instantiated once in an application. The *Singleton* pattern is an easy to implement pattern and simplifies accessing the classes methods. However giving multiple classes access to a single object bears the risk of collisions. Therefore special care has to be taken to make it thread-safe.

4.3.1 Game Management

A list of existing game instances is stored in a lobby in the `Manager` game manager object. It additionally manages a list of `AbstractPlayer` and a list of connected clients by storing the `Session` in a list. All three lists are synchronized through the `Collections.synchronizedSet()` method from the `java.util` library.

After the development of a game type and at the start of the execution, a developer has to register their game at the manager. They can call the method `public void addGameType(Class<? extends AbstractGame> gameClass)` which stores the game in the `Map<String, Class<? extends AbstractGame>>` gameMap where the simple name of the underlying class is used as key and the class itself as value. The simple name of a class is the class name without the package specification. For the before mentioned class `java.util.Collections` the method `getSimpleName()` would return a `String` containing "Collections". The registration of a game class is also important for the serialization of the game objects into according Json objects. The `com.google.gson` library provides a `GsonBuilder` class to which objects implementing the `JsonSerializer` can be registered. The manager has an `GsonBuilder` gsonbuilder attribute to which any new game types have to be registered through the method `gsonBuilder.registerTypeHierarchyAdapter(AbstractGame.class, game)`. This configures gson for the serialization and deserialization of inheritance structures, in particular the inheritence of a custom game type from the `AbstractGame` class. When a client requests a multitude of games, the lobby which potentially contains instances of several different classes is then able to be serialized to a `JsonObject`.

During runtime if a client request the creation a new game they have to specify the exact name stored in the gameMap. The method `public AbstractGame addGameToLobby(String gameName)` then tries to create a new instance of the corresponding class, store it in the lobby and return the new instance.

```
1 public void addGameType(Class<? extends AbstractGame> gameClass) {
2     try {
3         AbstractGame game = gameClass.getConstructor().newInstance
              ().getNewInstance();
4         gsonBuilder.registerTypeHierarchyAdapter(AbstractGame.
              class, game);
5     } catch (Exception e) {
6         e.printStackTrace();
7     }
8     gameMap.put(gameClass.getSimpleName(), gameClass);
9 }
```

Listing 3: Registering a custom game at the game Manager

4.3.2 Authentication

For the authentication of already connected players or spectators, the manager provides following methods:

- `public Player getPlayerByUsername(String username)`
- `public Player getPlayerBySession(Session session)`
- `public Player getPlayerByID(Long playerID)`

They can be used in different scenarios by the websocket endpoint. For example requesting the player by a given username is important during the login procedure to check if a user with this name already exists.

4.4 Websocket

The interface to the clients is started when the method `WebSocketServer.runServer()` is called. This creates constructs a new `org.glassfish.tyrus.server.Server` instance.

Glassfish is an open-source application server which includes implementations of all Java APIs [12]. As we want to create an application server supporting the WebSocket protocol, we need a server supporting the JPA. *Tyrus* is a standard implementation of the JPA which we can use for the communication via the Websocket protocol.

The *Tyrus* server can have multiple endpoints to which clients can connect. In our case we create a single `GameServerEndpoint` which handles the communication with clients. As described in the section about JPA, we can impement 4 methods which will be annotated with `@OnOpen, @OnMessage, @OnClose, @OnError` to handle the requests by the clients. When a websocket connection is opened, we receive a `Session` object. The WebSocket session represents the conversatin between the client and our websocket endpoint. The session is created when the HTTP handshake which promotes the HTTP connection to a HTTP WebSocket connection is completed. It can then be used to send messages to the clients without receiving a request by them before. Therefore we have a bidirectional communication where either participant can initiate a conversation. We additionally add the session to the game manager, to keep a list of connected clients.

During the closing of the websocket the session is lost can no longer be used. The websocket endpoint then removes the session from the game managers session list.

During the usual communication between the client and server enpoint, messages will be handled in the method annotated with `@OnMessage`. We first try to parse the incoming `String message` to a `com.google.gson.JsonObject`. If the send message does not have a correct json syntax we return a corresponding error message to the client detailing the syntax error. If the syntax is correct the websocket endpoint tries to serialize the `JsonObject` to one of the known data packets. For this step we first check the *type* of the Json, which lets us know the request a client wants to make. The type is an enum specifying the intent of the message, which were described in section **??**. The exact types are defined in the `AbstractPacket` class and are shown in following listing:

```
public static enum Type {
        INVALID(-1), LOGIN(0), GETGAMES(1), CREATEGAME(2), JOIN(3)
           , MOVE(4), DATABASE(5);}
```

Once the message has been serialized in an according `Packet` type the request can be executed. First we check if the client is already registered at the game manager. This allows use to verify if the client has already logged in and if yes which role they have (player, spectator). If they are not logged in, usually the first request should be to log in.

During the login procedure the client has to send a username, which can be any `String` containing between 3 and 20 letters. We then verify if the username has already been taken by checking the list of players registered at the game manager. If they username is not taken, we create a new `Player` object containing the session, the username and a user id and add them to the game managers player list. If the username is already taken, there is possibility that an already registered player resent a login request (either resending it in the same session or after a disconnect). If they provided the matching id to the username the session for this player will be updated. If not they will receive an error that the username is already taken.

From the login onward a client is intended to provide authentication for the communication. This includes the creation of a new game instance, where a `String` containing the name of the game is send to the game manager. If the name of the game can be attributed to an existing game type a new instance is created and the player receives a confirmation containing game information, such as the initial state and meta information.

The database access is also managed through this endpoint. Although a websocket is not the ideal means for a database query, using the existing websocket endpoint was the closest solution. In the client request, a client has the possiblity to include query attributes such as the game type and the amount of entries they want to receive.

Shown below is a sample of a Json request received by a client. The type lets the server endpoint know that the client requests to create a new game instance. The username and playerID are used for the authentication and finally the gameName is relayed to the game manager to try to create a new game instance of the `TicTacToeExample.class`.

```
{
    "type":2,
    "username":"Ernie",
    "playerID":"75ea17ba-c0db-4acb-9a78-8f24caf5cd50",
    "gameName":"TicTacToeExample"
}
```

Listing 4: A sample Json message to create a new game

4.5 MCTS

As stated before, if the Developer wants the MCTS Bot to play their game, they need to additionally define the set of legal moves from the current game state. They define

the moves with the *listMoves()* method of the interface *MCTS*. In other words if MCTS capability is desired, the game needs to implement this interface and implement the inherited method in such a way, that for a given *State state* it returns a list of non isomorphic moves.

The Bot extends the *AbstractPlayer* class and behaves to the *AbstractGame* as an *Observer* in the same way as other players. It receives updates about a changed game state in its *update()* method where it then calculates its next move. To not change the state of the game it is subscribed to, it first creates a deep copy of the game object. A deep copy of an object which references to other objects, not only creates a copy of the object but also of every referenced object. Therefore the original object and the copy now refer to different objects. In contrast, a normal copy keeps the original references and changing the references in the copy will change the references in the original. This additional step is necessary, as to not accidentally notify the players subscribed to the original game.

At this stage the Bot creates the root Node out of the current state of the game and starts the MCTS process as described in Section 3.7.4. In the simulation step we assign a score of 1 if the game is won, 0 if the game is lost and 0.5 if the game resulted in a draw. It is debatable if considering a draw as 0 points is advantageous to the result or not.

4.6 Example

In this section we present an example of a game implementation and walk though the process in the eyes of the developer. This allows us to summarize the classes and methods described in the earlier sections and illustrate the intended use. In figure 6.1 there is already an example class called `TicTacToeExample` and we will show how the creation process.

4.6.1 Modelling

The first step of creating a custom game hosted by the game server is modelling the game. This includes two steps: modelling the state of the game and defining the rules.

As described earlier, in this application the state is represented by a map, which offers great flexibility for the developer. For the game of TicTacToe the state can be represented by a 1D array of size 9 of `int`, where each field of the array represents a cell on the playing field. If a cell is empty, the corresponding field of the array is an 0, a cross is represented by a 1 and a nought by a 2.

The rules say that players take turns in putting down their marks and that a player can only put a mark in an empty field.

A final state is reached once either player puts 3 marks in a row, column or on the diagonal or whenever there are no more free places to put down a mark.

4.6.2 Creating the class

The next step in the implementation is creating a new class for their game. This includes implementing the methods defined by the superclass `AbstractGame` and its interfaces. First

we create the new class `TicTacToeExample` and its constructor:

```
public class TicTacToeExample extends AbstractGame{

    static int winComb[][] = {{0,1,2},{3,4,5},{6,7,8},{0,3,6},
    {1,4,7},{2,5,8},{0,4,8},{2,4,6}};

    public TicTacToeExample() {
        super();
        getState().put("board", new int[]{0,0,0,0,0,0,0,0,0});
    }
}
```

Listing 5: The state definition for a sample Tic Tac Toe game

The static variable `winComb` lists all possible variations which marks of the same type can have to win the game. This will be later used in the `checkIfOver()` method. The constructor calls the constructor of the superclass, which instantiates the state variable. The state can be accessed through the getter method where we save the initial state of the board under the key "board". If desired additional properties of the state can be saved here.

4.7 Defining the rules

Now that we have created a game class and defined the initial state, we can move on to the rule definition. In other words, we create a function to check if a given move can bring us from our current state to the next legal state. First we have to parse the move which is passed to the `move()` method as `String`, then we can check the move for the actual game conditions. With the board represented as a 1D array of size 9, the first condition is that the given move requests a placement within the boundaries of 0 and 8. Then we can check if the cell is not yet occupied and if the correct player is trying to move.

```
public void move(AbstractPlayer player, String move) throws
    IllegalMoveException {
    int cell = 0;
    int[] board = (int[]) getState().get("board");
    try {
        cell = Integer.parseInt(move);
    } catch (NumberFormatException e) {
        throw new IllegalMoveException();
    }
    if (cell < 0 || cell >= board.length ||
            board[cell] != 0 || !player.equals(getCurrentPlayer())
                ) {
        throw new IllegalMoveException();
    }
```

```
13      int currentPlayer = getActivePlayerList().indexOf(
            getCurrentPlayer());
14      board[cell] = currentPlayer + 1;
15      getState().put("board", board);
16  }
```

Listing 6: The definition of legal moves in Tic Tac Toe

4.7.1 Defining the end states

The last step in the game definition is defining the final states, i.e. when the game is over. For this we implement the methods `checkIfOver()` and `checkIfDraw()`. In the former we iterate over `winComb` and take the sets of 3 numbers at a time. We then check if the cells from the board with the corresponding indices all contain either 1s or 2s. Checking for a draw is easier as the game is a draw when the `int[]` from `getState().get("board")` does not contain any zeroes.

```
1   public boolean checkIfOver() {
2       int[] board = (int[]) getState().get("board");
3       for(int[] combination: winComb){
4           if (board[combination[0]] == board[combination[1]]
5               && board[combination[1]] == board[combination[2]]
6               && board[combination[1]] != 0){
7                   setWinner(getActivePlayerList().get(board[
                        combination[0]]-1));
8                   return true;
9               }
10      }
11      if (checkIfDraw()) {
12          return true;
13      } else {
14          return false;
15      }
16  }
17
18  public boolean checkIfDraw() {
19      int[] board = (int[]) getState().get("board");
20      for (int cell: board) {
21          if (cell == 0) {
22              return false;
23          }
24      }
25      return true;
26  }
```

Listing 7: CheckIfOver and checkIfDraw detect the final game state

Note that we call `checkIfDraw()` from within `checkIfOver()` as a game is over when it is a draw. We still need the two functions as game is not necessarily a draw when it is over.

4.7.2 Bots

To add the option to play against a bot, the class `TicTacToeExample` needs to implement the `MCTS` interface. Then we need to implement the `listMoves()` function which should return a `List<String>` of legal moves which can be reached from a current state. In this case it returns the index of any cell containing 0 from the board. For example if we consider our board state `int[] board = {0,2,1,1,2,1,2,1,0}`, `listMoves()` returns `["0", "8"]`.

```
1  public List<String> listMoves() {
2      List<String> legalMoves = new ArrayList<>();
3      int[] board = (int[]) getState().get("board");
4      for (int i = 0; i < 9; i++) {
5          if (board[i] == 0) {
6              legalMoves.add(String.valueOf(i));
7          }
8      }
9      return legalMoves;
10 }
```

Listing 8: Implementation the MCTS interface for AI capability

4.7.3 Registering the game

The final task before a user is able to play the game is registering it to the `Manager`. In the main class, before starting the server we get the manager instance and register the game through the `addGameType(Class<? extends AbstractGame> gameClass)` method with `manager.addGameType(TicTacToeExample.class)`.

5 Evaluation

In this section we will evaluate the application created in the implementation process by comparing it to common metrics. As metrics we use static code analysis which shows the quality of the code. They are mostly used to spot weak spots when it comes to applying best practices in the development process.

5.1 Metrics

For the metrics evaluation we used an IDE code analysis plugin called *Project Usus* [13]. It is an open source eclipse plugin which provide tools to analyse object-oriented code to

find potential weak spots. Additionally this plugin allows us to find complexities in the code.

- **Class size** takes into account the number of static and non-static methods in each class. (Hotspots rating > 12)
- **Cyclomatic complexity** can be used as a quality metric and indicates the complexity of a project by counting the number of possible branches in the execution path. (Hotspots rating > 4)
- **Lack of cohesion of classes** (LCOC) describes the number of unconnected groups of classes in a package. (Hotspots rating > 1)
- **Method length** Counts the number of statements in each method (not lines of code). (Hotspots rating > 9)

Table 0.1: Static code metrics of the completed project

Indicator	Avg. rating	Hotspots	total
Class size	6.7	3	22 classes
Cyclomatic complexity	2.3	6	217 methods
Lack of cohesion of classes	1.6	3	10 packages
Method length	4.5	14	217 methods

After comparing the average rating to the benchmarks, we can count the hotspots. Hotspots are methods, classes or packages which do not meet the benchmarks. In general those are complex classes, in this project especially the `AbstractGame` and `GameServerEndpoint` . These are the two main access points for the required functionalities and as they have the attributes with the most important information about the server state, they are often accessed by other classes.

To visualise the centrality of those two classes, the *Project Usus* plugin provides a feature which plots the classes into a graph. The intensity of the association between the classes is then calculated with the amount of relations the classes have to each other. A high amount of references results in string bonds between classes. The graph then runs a physics simulation where the strength of the bonds represents the strength of springs between the classes. Stronger springs create a greater attraction and pull the classes closer.

Figure 5.1 show the results of the *Usus* simulation. This highlights again the weight of the central classes `AbstractGame`, `GameServerEndpoint` and `Manager`. This is not unexpected as they have respectively the most functionality for the game logic, client communnication and game management.

6 Conclusion

The trade-off between ease to use and extensibility is not always easy to find. Implementing new functionality has always the risk of restricting the breadth of possibilities for a future

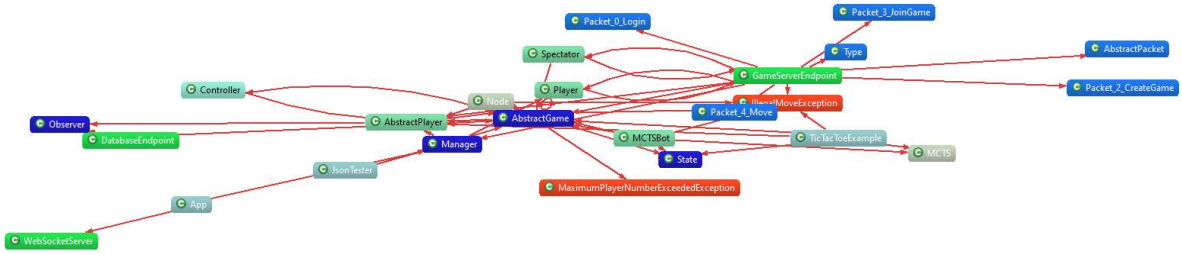

Figure 5.1: Class graph created by *Usus* plugin visualising the interactions between classes

developer. Often the design of this application had to be reiterated when it became apparent that a new functionality restricted the use cases of the developer.

The target audience of this application are developers who are enthusiastic about AI techniques and who are able to implement a game in a framework even if it is a bit bare. The game engine was designed with this in mind and it is possible to abuse it to achieve different behaviour than it was designed for. The line between allowing freedom to the developer and restricting the use cases too much is fine and we rather deviate in the direction of additional freedom than restriction. In this sense the goal of creating a platform to host AI competitions was achieved. The requirements posed for the development phase such as the runtime phase have been fulfilled by using a design driven software development approach.

The developer has the highest amount of possibilities when it comes to implementing a game while on the other side it is possible to implement a game by defining the bare minimum of what makes a game a game. On the other side players have the freedom to host their AIs written in the programming language of their choice on a platform of their choice. They also just have a bare minimum of necessities to being able to communicate with the server.

The evaluation showed that the application turned out more complex than expected, which can be attributed to the abstractness of the most parts of this project.

Overall, the goals from section 1.1 are accomplished. The process of creating a new game requires three definitions from the developer. The initial state, the rule set and detecting the final state. However, this is probably only the case for the simplest of games and in many cases the developer is tasked to create custom serialization for the client interface for example. Therefore making use of the extensibility of the framework is probably more often a necessity than a choice.

6.1 Future work

For the automation of the competitions a tournament system would be a good addition to the application. With a similar level of abstractness as the game interface, a tournament interface could be created which allows different forms of competitions, such as tournaments in a knock-out form or a swiss-system (often used in chess competitions).

Extending the MCTS Bot would also be an interesting challenge. Although it would probably make more sense to host an AI on a different dedicated server than on the server hosting the game engine. Although the idea of creating a capable AI for a multitude of

board games (or other game styles) probably does not have many real life applications as a specialised AI will always be better. Nonetheless similar to the general uses a MCTS AI has to a multitude of game, an AI using unsupervised learning could be applied to different game types and over time learn patterns to improve itself. In addition to playing against itself, it could use data from other games which are collected on the game server to learn new tactics.

Bibliography

1. 2022. Available also from: `https://boardgame.io/documentation/#/`.
2. BURNS, B. *Darkstar: The java game server*. O'Reilly Media, Inc., 2007.
3. ELIAS, G. S. et al. *Characteristics of games*. MIT Press, 2012.
4. MELNIKOV. *The websocket protocol*. IETF, 2011. Available also from: `https://www.rfc-editor.org/rfc/rfc6455.html`.
5. VOS, J. *JSR 356, Java API for WebSocket*. 2013. Available also from: `https://www.oracle.com/technical-resources/articles/java/jsr356.html`.
6. CHEN, B.; XU, Z. A framework for browser-based Multiplayer Online Games using WebGL and WebSocket. In: *2011 International Conference on Multimedia Technology*. 2011, pp. 471–474. Available from DOI: `10.1109/ICMT.2011.6001673`.
7. BALZER, S. Contracted persistent object programming. In: *PhD Workshop, ECOOP*. 2005, vol. 12.
8. WAMBLER, S. *Mapping objects to relational databases: O/R mapping in detail*. [N.d.]. Available also from: `http://www.agiledata.org/essays/mappingObjects.html`.
9. GAMMA, E. *Design patterns*. Pearson Education India, 1995.
10. CHASLOT, G. et al. Monte-carlo tree search: A new framework for game ai. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2008, vol. 4, pp. 216–217. No. 1.
11. MCCLUSKEY, G. [N.d.]. Available also from: `https://www.oracle.com/technical-resources/articles/java/javareflection.html`.
12. 2022. Available also from: `https://projects.eclipse.org/projects/ee4j.glassfish`.
13. *Project Usus*. 2022. Available also from: `http://www.projectusus.org`.

List of Figures

Listings

Appendix

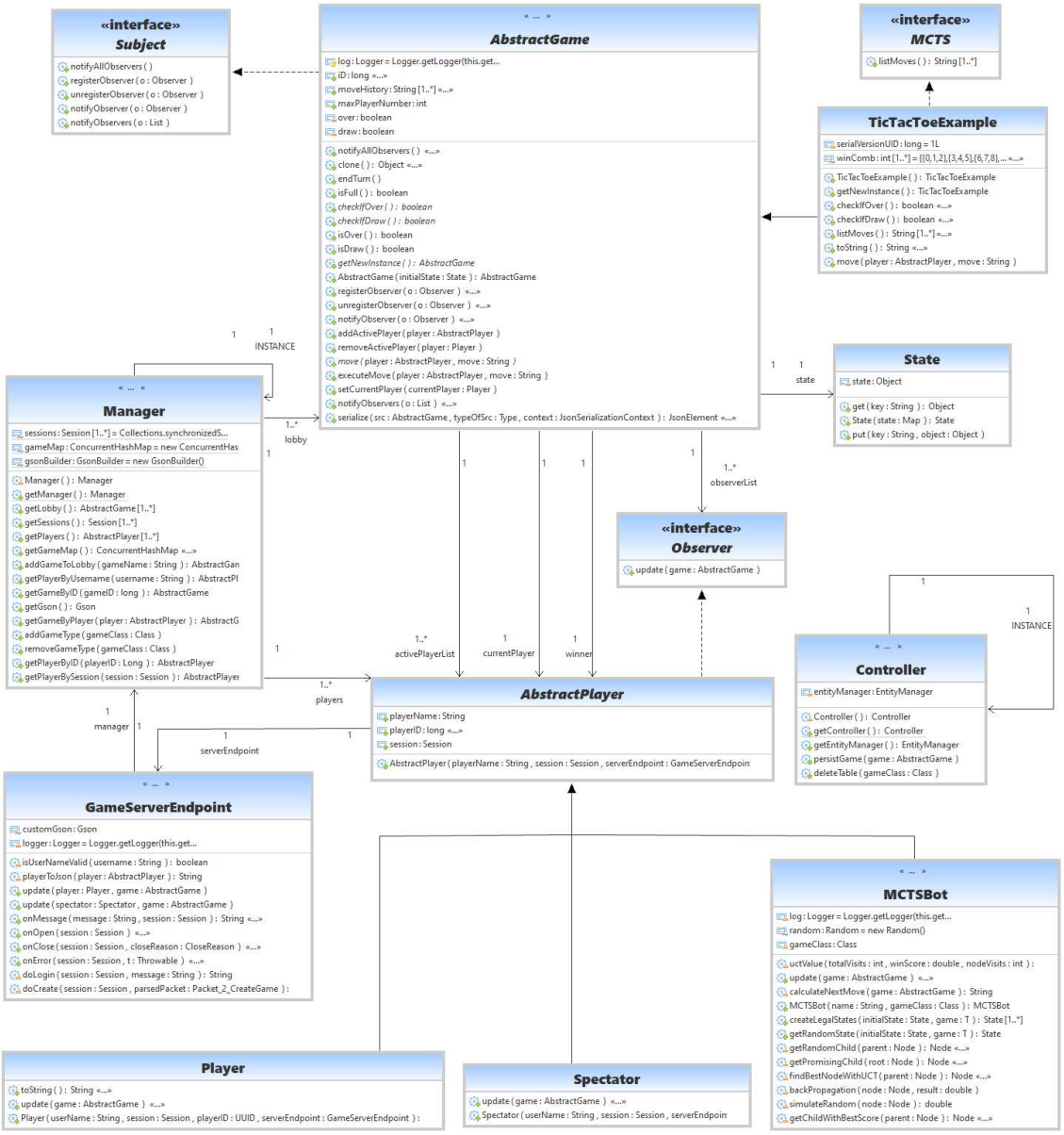

Figure 6.1: The full class diagram of the game server