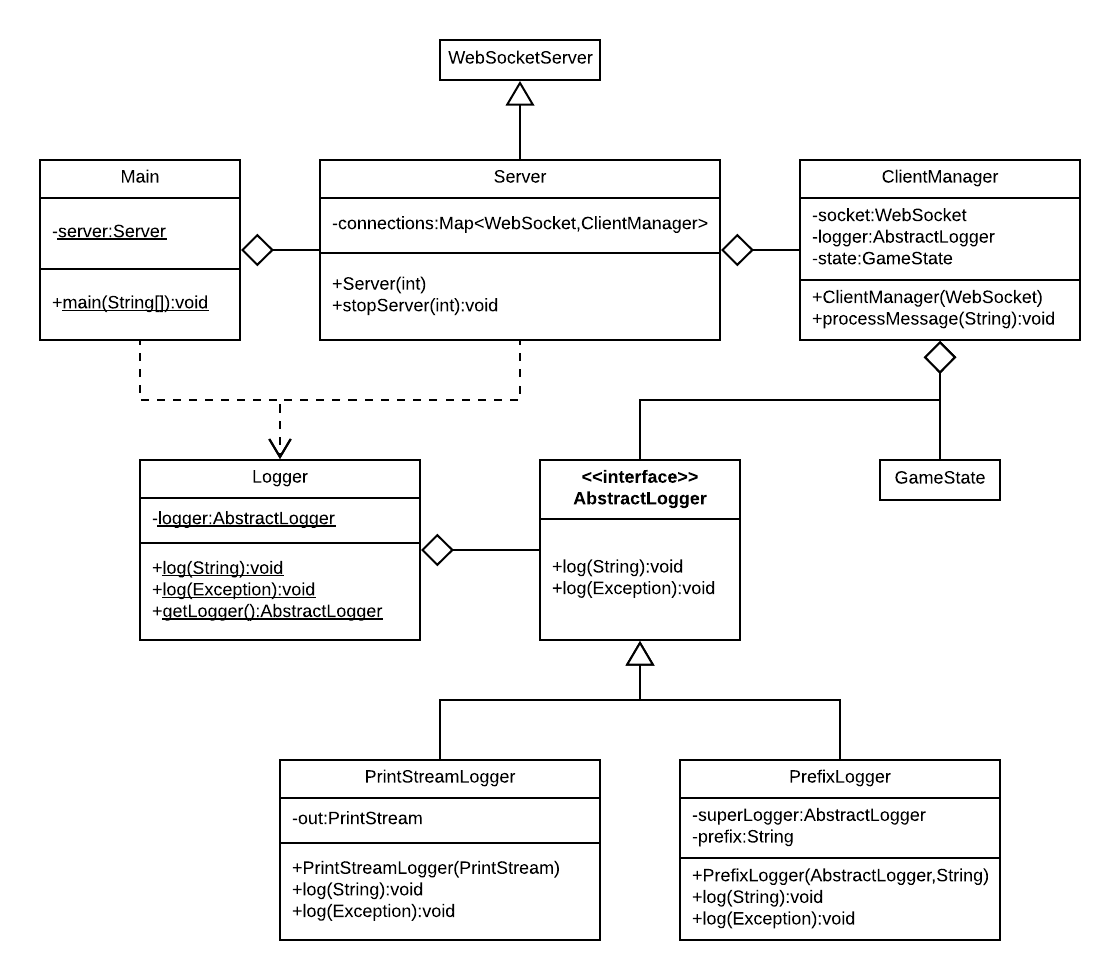
Game server main architecture

The game server is the part of the server that communicates with the users that are actually playing the game. There are two major tasks of the game server. It needs to send the level with its problems to the client. It also needs to check the formal specifications of the player. But the game server also needs to keep track of the score of the players in order to implement adaptive difficulty. The decision has been made to make the game server in Java, so there needs to be an object oriented approach.

# Alternatives

## The main idea

The first and main idea for an architecture is the following. This architecture is more of a guideline than an actual solution.



Standard logging is done through the console output. But in order to prepare for a change to file output, there is the *AbstractLogger* interface. This can log strings and exceptions. Two useful implementations of this are the *PrintStreamLogger*, which simply logs everything onto a *PrintStream*, and the *PrefixLogger*, which takes an *AbstractLogger* and adds a prefix to whatever is logged. The *Logger* class contains a static logger that can be used from anywhere. This is currently a *PrintStreamLogger* which has the *PrintStream System.out*, which logs everything to the console, but this can easily be changed.

The *Main* class has the main method, which creates a *Server* and then starts reading from the console until the program terminates. If it turns out that there will be a lot of console commands, the responsibility of reading them can be moved to another class.

A *Server* is a *WebSocketServer*, which is a class from the Java *WebSocket* library. This already handles all the connections. The only thing left to do is define what needs to happen when a new connection is opened, when a connection is closed, when a message is received from a connection and when an error occurs. Since this gives the *Server* class enough responsibility as it is, actually handling the messages from the connections is done in a *ClientManager* object. There is one *ClientManager* object for each connection. The connections are stored as *WebSockets*, so a *Map* from *WebSockets* to *ClientManagers* is stored in the *Server*.

A *ClientManager*, as previously stated, needs to handle all the messages from one connection. In order to be able to send messages with the connection, every *ClientManager* needs to keep track of the *WebSocket* which defines the connection. Furthermore, every *ClientManager* is given their own logger, such that logged messages from different *ClientManagers* are distinguishable. This is an example where a *PrefixLogger* is useful. Also, every *ClientManager* needs to keep track of some *GameState* in order to determine what to do when a message is received. Finally, the *ClientManager* class has a *processMessage* method, which actually processes a message from the connection.

The *GameState* should not stay in one class, but specifying how this will be distributed over multiple classes is not relevant to the main architecture.

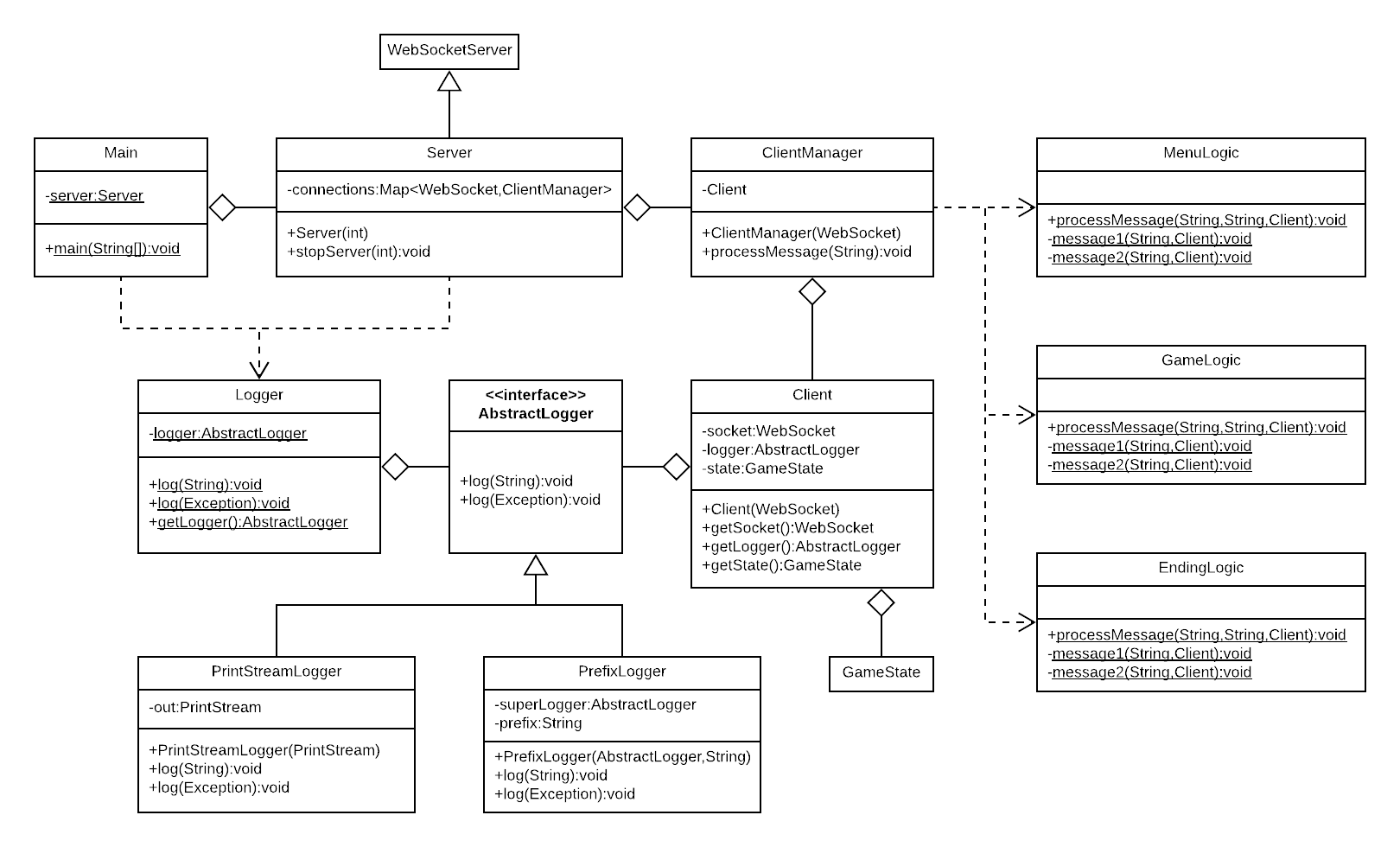
Apart from the classes in the diagram, there are a few classes filled with static methods in order to, for example, read entries from the database, compare specifications with the Haskell API, and read from the settings.json file, which contains the settings for the *GameServer*. But since these classes are not relevant to the main architecture, they will not be considered in this document.

## An improvement

There is one big problem with this architecture. The *ClientManager* class will contain all message processing, but there will be a lot of different types of messages and they all need a method for handling that type of message. So this architecture unfortunately does not scale very well with the amount of different commands that will be defined.

Something that makes it hard to come up with a solution is that processing a message requires access to the *WebSocket*, the *AbstractLogger* and the *GameState* corresponding to the connection. These objects can not be static, since there will be multiple connections, each with their own *WebSocket*, *AbstractLogger* and *GameState*. Therefore, pointers to these objects will need to be sent around.

One solution is the following architecture.

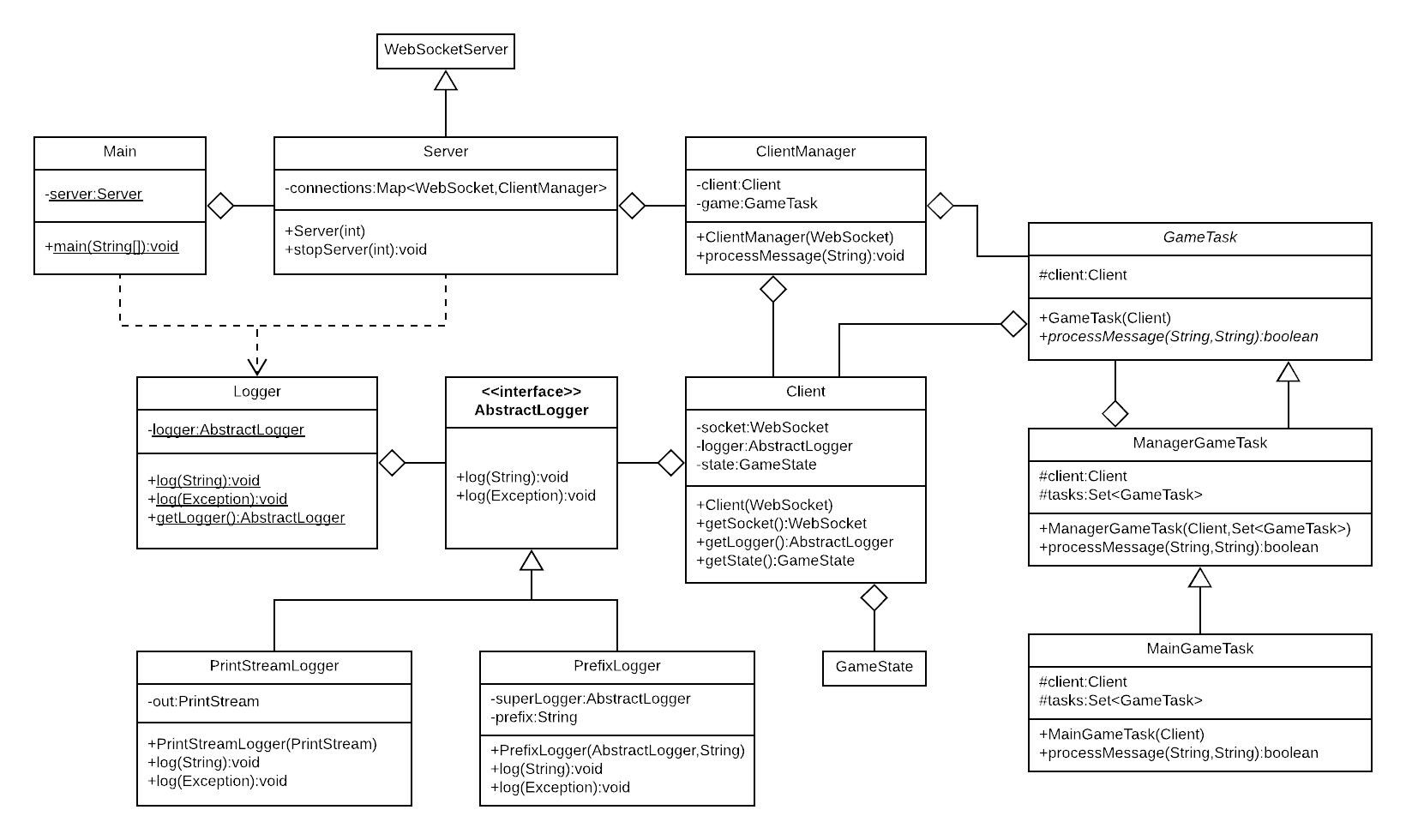


The *WebSocket*, *AbstractLogger* and *GameState* objects are wrapped into one object: a *Client* object. Now every *ClientManager* only needs to keep track of the corresponding *Client* object. As concluded earlier, pointers to the *WebSocket*, *AbstractLogger* and *GameState* will need to be sent around, so wrapping these objects into a single object makes this a bit cleaner, as only one pointer will need to be sent around.

The main idea of this architecture is to distribute the message handling over classes with static methods that handle messages. As these methods are static, the *Client* pointer needs to be given as a parameter. The goal of this is to be able to distribute the different kinds of messages through different classes. One example of this would be to have a class that handles all messages that are specific to the menu screen of the game, such as verifying the user and starting the game. Another such example would be to have a class that handles all messages that are specific to the ending screen of the game, such as storing the score and restarting the game. All these classes can have their own main *processMessage* method that will check if any of the message handlers in the class apply to the given message. The *ClientManager* now only needs to parse the command of any incoming message and send this, together with a pointer to the corresponding *Client* object, to the main *processMessage* method of all message handling classes.

## An alternative

An alternative solution is the following architecture.



Instead of the message handling methods being static, they are inside *GameTask* objects that the *ClientManager* initializes and keeps track of. The *GameTask* object is abstract. It needs a *Client* pointer on construction, such that it can store this pointer, such that the *Client* pointer does not need to be sent around for every message that gets processed. Examples of implementations of the *GameTask* object would be a *MenuTask*, which would be similar to the *MenuLogic* class in the previous architecture, and an *EndTask*, which would be similar to the *EndingLogic* class in the previous architecture. The *processMessage* method of the *GameTask* class returns a boolean, which indicates whether or not the message was actually handled.

The abstraction of *GameTask* objects allows for a *ManagerGameTask*. Such a *GameTask* could distribute messages over multiple other *GameTasks*. It could use the return value of the *processMessage* method to determine when to stop distributing the message. For example, there could be a *MainGameTask* object, which would send any message from the client to the *MenuTask* and the *EndTask*. This would make the *MainGameTask* object a *ManagerGameTask*. Now *ClientManager* objects only need to initialize and keep track of a *MainGameTask*.

Since *ManagerGameTasks* are *GameTasks* themselves, a *ManagerGameTask* could potentially manage other *ManageGameTasks*. For example, apart from *MenuTask* and *EndTask*, there might be a *GameLogicTask* for when the user is actually playing the game. This *GameLogicTask* could then be a *ManagerGameTask* itself, which distributes messages from the client to the *SpecificationTask* and the *ScoreTask*. This way, the *GameTasks* form a graph, which is a rooted tree. The *MainGameTask* is the root, the *ManagerGameTasks* are the nodes and the other *GameTasks* are the leaves. This way, no matter how many different messages there will be, there do not need to be any classes that have too many of them.

# The decision

One reason to pick the first solution is that it does not require any initialization. The second solution requires the *ClientManager* to initialize the *MainGameTask*. Also, for every *ManagerGameTask*, all the *GameTasks* it is supposed to manage need to be initialized. On top of that, all *GameTasks* take up a bit of space, so the first solution is a bit more space efficient than the second solution. Also, remark that the *ManagerGameTask* idea from the second solution could also be applied to the first solution. There could be a *MainLogic* class which has a static *processMessage* method that distributes the message to *MenuLogic*, *GameLogic* and *EndingLogic*.

One reason to pick the second solution would be that the *processMessage* calls become much cleaner, as the *Client* object does not need to be sent around. This also makes the *processMessage* calls cheaper in terms of space, as well as time. But the main advantage of the second solution over the first is that it is more object oriented. As stated at the top of this document, the game server will be implemented in Java, so there will need to be an object oriented approach. The static methods in the first solution are very similar to functions. It is basically what a functional approach would look like. But the *GameTasks* in the second solution are very object oriented. The tree structure of the *GameTasks* also works better in an object oriented approach, as there are actual objects that represent the vertices of the tree.

For these reasons mostly, the decision has been made to implement the second solution in the game server.