Peer review #2 - GC11

Samuele Pischedda, Angelo Prete, Gabriele Raveggi, Andrea Sanvito

06/05/2024

General Overview

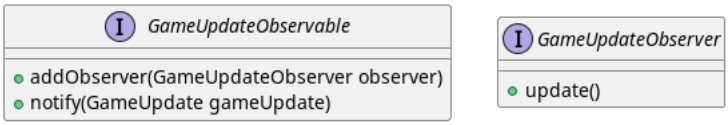
In this document we describe the UML of our Codex Naturalis java implementation. In particular, we will explain how we implemented the controller of the MVC architectural pattern and the client-server communication protocol.

Introduction

The project has been extended by adding network functionalities. The game uses a double connection, one for joining and leaving lobbies (the ”main menu” connection) and the other one for in-game actions and for the chat. Actions from the client to the server and updates from the model to the client have been implemented using the command pattern.

1 Model

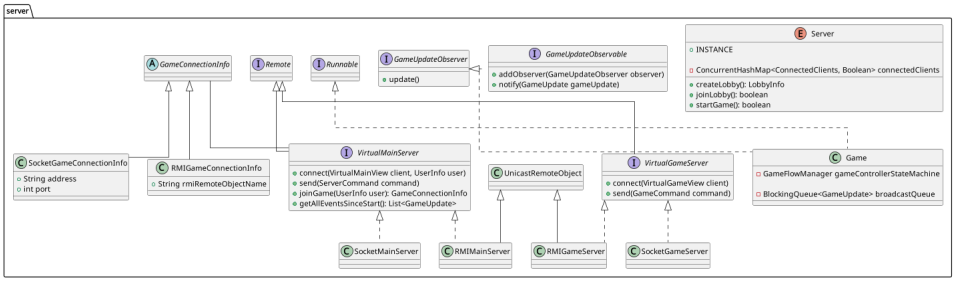
In particular, the updates are generated in the model and sent to all the classes that need them through the use of the observer-observable pattern.



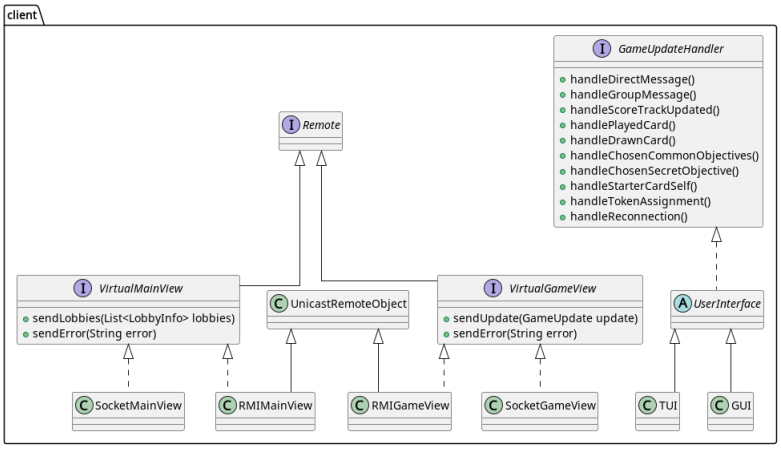
All the model classes that produce updates must implement the `GameUpdateObservable` interface. The `GameModel` constructor now takes a `List<GameUpdateObserver>`, so that all the `GameUpdateObservable` can register them.

2 Network

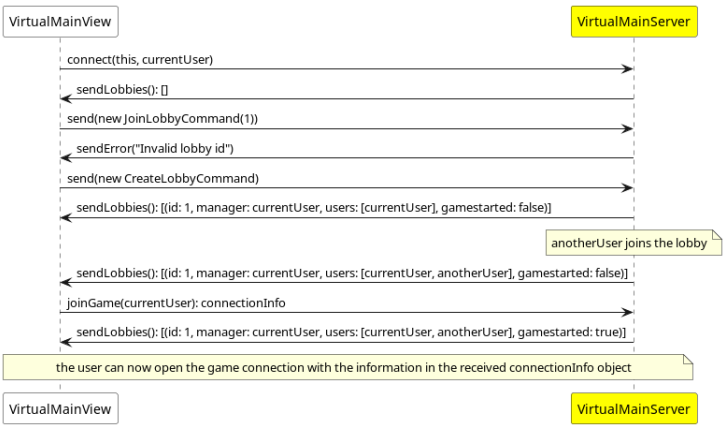
The network communication is implemented using classes that extend interfaces shared by both the RMI and socket implementation. The main connection to the server (used for creating lobbies and starting games) is opened when the client is started and it is kept alive until the client is closed. The `RMIMainServer` and `SocketMainServer` classes, that implement the `VirtualMainServer` interface, are used when the client is connecting to the server for the first time.



We kept the RMI implementation trivial and made the socket implementation mimic the RMI behaviour. Once the connection is established (via the RMI Registry or through a socket), the clients must call the `connect(...)` method that take their `VirtualMainView` as parameter. This interface is the client's dual of `VirtualMainServer` and is implemented by `RMIMainView` and `SocketMainView` classes. The interface offers methods to show errors on the view or to send the list of updated lobbies.



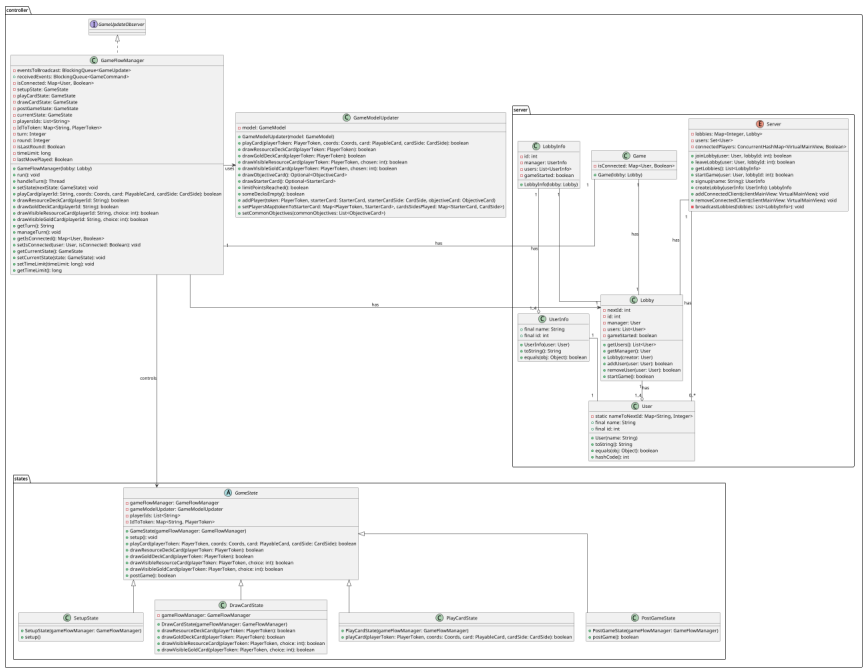
By sending a `ServerCommand` (done by calling the method on the `VirtualMainServer`) a client can create, join or leave a lobby. When the client is ready he can call the `joinGame` method and receive a `ConnectionInfo` object that holds the information needed to open the second connection.



The second connection has the same structure as the first: there is a`VirtualGameServer` and a `VirtualGameView`, both implemented as RMI and socket classes. The clients can send `GameCommand` objects to the controller by calling the `send(...)` method of `VirtualGameServer` and they can receive `GameUpdate` objects when the server calls on the `VirtualGameViews` the `sendUpdate(...)` method. Once the clients receive the update object, they can call the `execute(...)` method on them by passing an object that implements `GameUpdateHandler` as parameter. This makes the implementations of GUI and CLI independent, as they can simply define the `GameUpdateHandler` interface.

3 Controller

The controller package has been updated by implementing a main controller class as a runnable, which is run by a thread started in the main server as soon as the game begins. The class uses a state machine designed through the State Pattern. A class to update the model (`GameModelUpdater`) was added.



3.1 GameFlowManager

This class represents the game itself. It's a runnable, whose `run()` method is divided into three main phases:

- **pre-game (or setup) phase**: allows users to choose their tokens, the side to play of their starter card and the objective card. It also creates all entities needed for the game to happen.
- **in-game phase**: loops through the players' turns, starting two threads each turn: one to actually handle the turn, and the other to keep a timer, to forbid players from going beyond a `limitTime`.
- **post-game phase**: handles the final phase of the game, where the winner is declared and all clients are notified of the results.

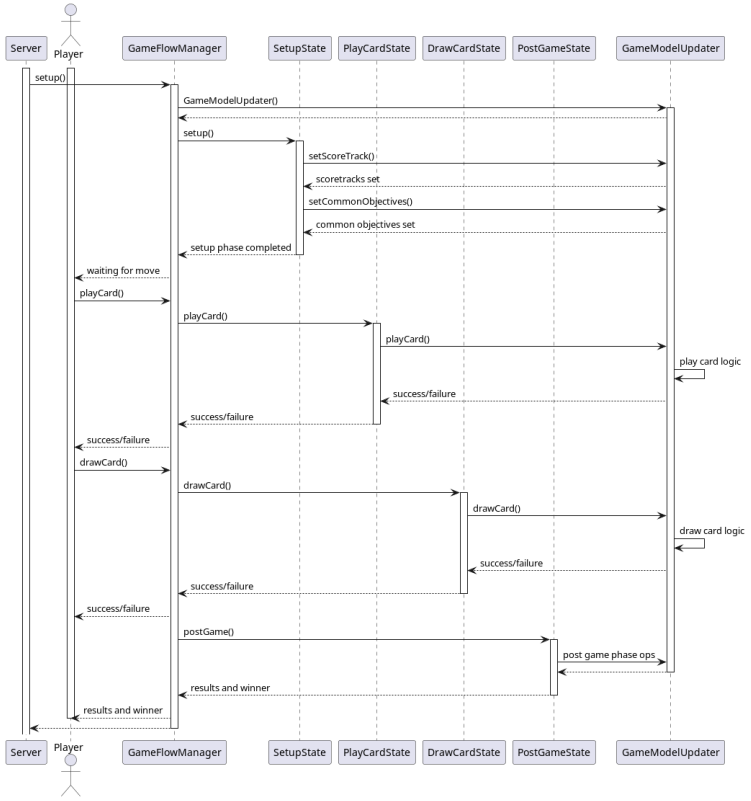
It also keeps track of the current turn, the current round (composed by more turns), and the current player.

3.2 GameState

The information of the GameFlowManager state is kept through a bunch of `GameState` objects, following the State Pattern. This way, players can make their moves only when the `GameFlowManager` is in the correct state.

All the methods in the abstract class `GameState` return a boolean. By default, they return false. Each state will override its own methods. The return value is used by the controller to understand whether the action was successful or not. In particular, the states used are:

- `SetupState:` when the `GameFlowManager` is created, its `currentState` is set to this state. Here, the only overridden method is `setup()`, which handles the initial part of the game.
- `PlayCardState`: state where a player needs to play a card. It overrides the `playCard` method;
- `DrawCardState`: state where a player needs to draw a card. It overrides the four `drawCard` methods;
- `PostGameState`: through this state, the `GameFlowManager` handles the final part of the game. It overrides the `postGame` method.



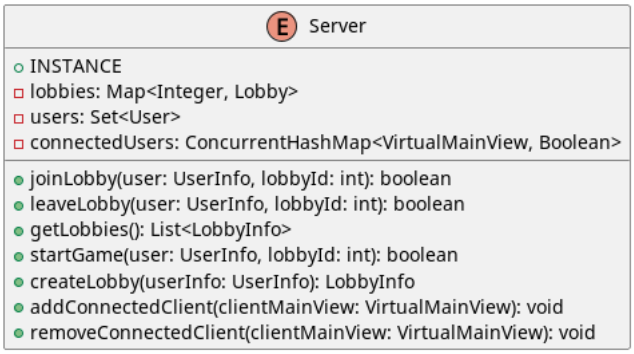
3.3 GameModelUpdater

This class is the section of the controller whose logic allows to update the model, following players' moves. More precisely, the more important methods are:

- `playCard`: it allows a player to place a card on his board. The method checks whether the player can play that card in that position and updates all structures accordingly.
- `drawPlayableCard`: this method is split in four sub-methods, one for each `Deck` type. If the player is drawing from the visible card, an integer is required to discriminate which card he is selecting.
- `drawObjectiveCard`: used in the setup phase by the `GameFlowManager`.
- `drawStarterCard`: used in the setup phase by the `GameFlowManager`.

3.4 Server

The `Server` enum, implemented using the singleton pattern, is the module that contains all the lobbies and users logic. It's implemented as an enum, so it can be called from anywhere using it's only value `Server.INSTANCE`. Therefore, it doesn't have synchronization issues when constructing it and when retrieving the instance (that a static class would have).



The server also keeps track of connected clients to send updates. The collection used is a `Map` from `VirtualMainView` to `Boolean`, `true` when the user is not in-game, so we know what clients should receive the main menu updates.