# MVC UML - GC 5

Andrea Sgobbi, Roberto Scardia, Jonatan Sciaky, Luca Simei

May 2023

## 1 Preface

The networking stack is very similar to that seen during lessons: our main Client/Server objects are called ***LocalClient*** and ***LocalServer*** (note that, unlike the example we've seen, we are not using multiple LocalServer instances). A *LocalClient* either communicates via RMI to a *LocalServer*, or uses a ***ServerStub*** to emulate RMI writing on a socket. These two objects act mostly as "routers" and handle the initial setup of the connection.

Since we are implementing multiple concurrent games, our controller is split up in a ***LobbyController*** and a ***GameController***. Brokerage between the two controllers is handled by *LocalServer*.

The ***GameModel*** is largely the same as the previous version, with extra methods to notify the clients of model changes and to serialize the model in order to implement persistence.

Our message-based communication protocol allows us to have different `onMessage()` methods being directly called by the server: these methods represent the only *GameController/LobbyController* entry points (same goes for the View running on the client)

## 2 GameController

The controller receives messages of two types: ***ChatMessage*** and ***MoveMessage*** (includes selected board coordinates, insertion order and insertion column). It's guaranteed by the server that the messages come from *Clients* within the game the *GameController* is handling.

Chat is relayed by the model, depending on its destination, while Move needs various extra legality checks, after which it gets processed by the main logic of the controller: remove the selected tiles from the board, insert them into the shelf, then check for CommonGoals, see if the game is over and see if the board needs to be refilled. Any changes made to the Model will then automatically be propagated to the clients as different *ModelMessages*.

# 3 LobbyController

The *LobbyController* manages creating/joining lobbies and starting the game once they are full. Possible incoming lobby messages are 4: ***RequestLobby***, asking to display all of the current lobbies, ***RecoverLobby***, asking to join a lobby recovering from a server crash, ***CreateLobby***, asking to create a lobby, and ***JoinLobby***, asking to join a specific lobby.

## 3.1 Lobby

Lobbies are identified by their unique lobbyID, which is used by Views to choose which lobby to join. It is also required that nicknames be globally unique, attempts to take a nickname already in use (or join full lobbies) will result in a Bad Response from the server. Lobbies are removed when a game ends, along with their model saved on disk.

To implement persistence, we also have a *RecoveryLobby* which contains model information read from disk (we save the model after each turn). These are filled when receiving RecoverLobby messages. Once a RecoveryLobby is full, we follow the same procedure used for normal lobbies to start the controller and convert the Recovery-Lobby to a normal Lobby. From then on, their behaviour is identical.

## 3.2 Concurrent Games

Once a login causes a lobby to be full, the *LobbyController* will create a new instance of the *GameController*, a new instance of *GameModel*, link the model up to all the clients and return the new controller to the server. The server, using a Client->Controller mapping, is then able to forward all messages from clients within that lobby to the new controller.