POLITECNICO DI MILANO

Council of four

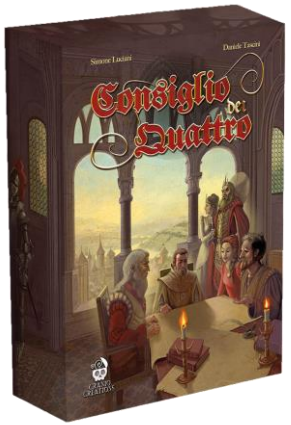




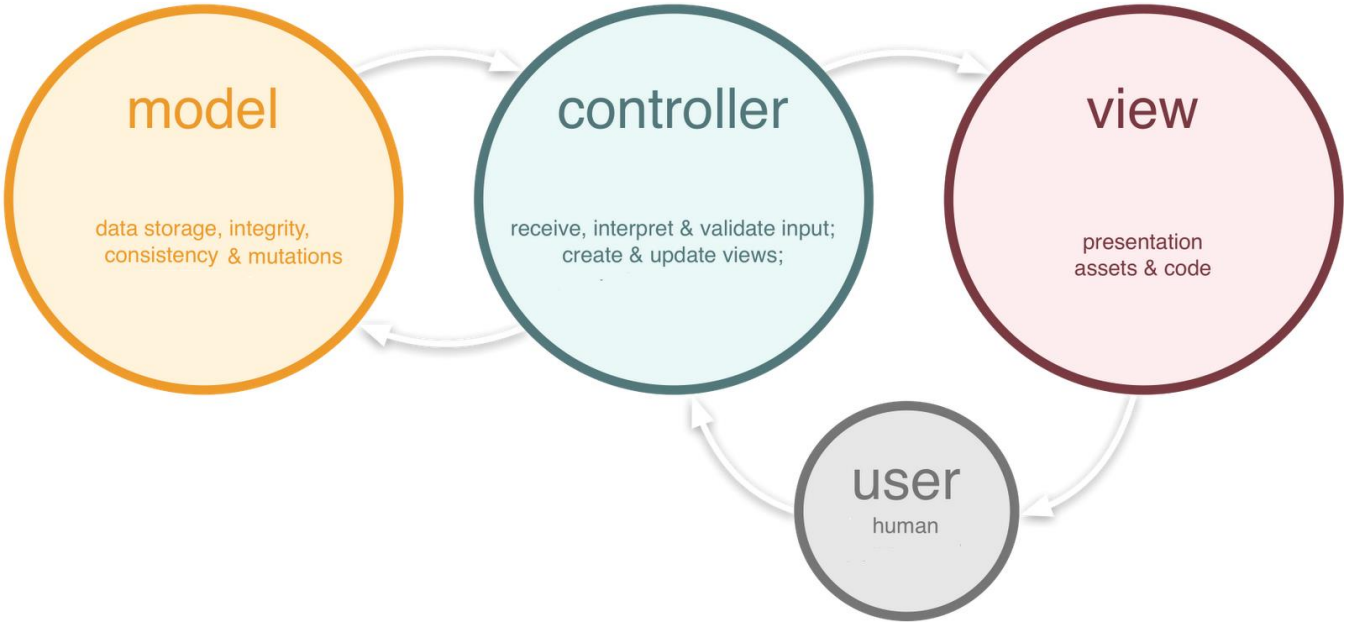
Software Engineering - Project

Academic year 2015/2016

Gianola Giovanni
Ionata Valentina
Leveni Filippo



The whole architecture of the application follows the **MVC** pattern, the model contains all the information to store, the state of the current player turn, the controller manages game initialization, rules and the machines that allowed the succession of states, the view is composed by two parts: a remote view on the server that connected the local controller with the remote client view, that allowed the exchange of messages.

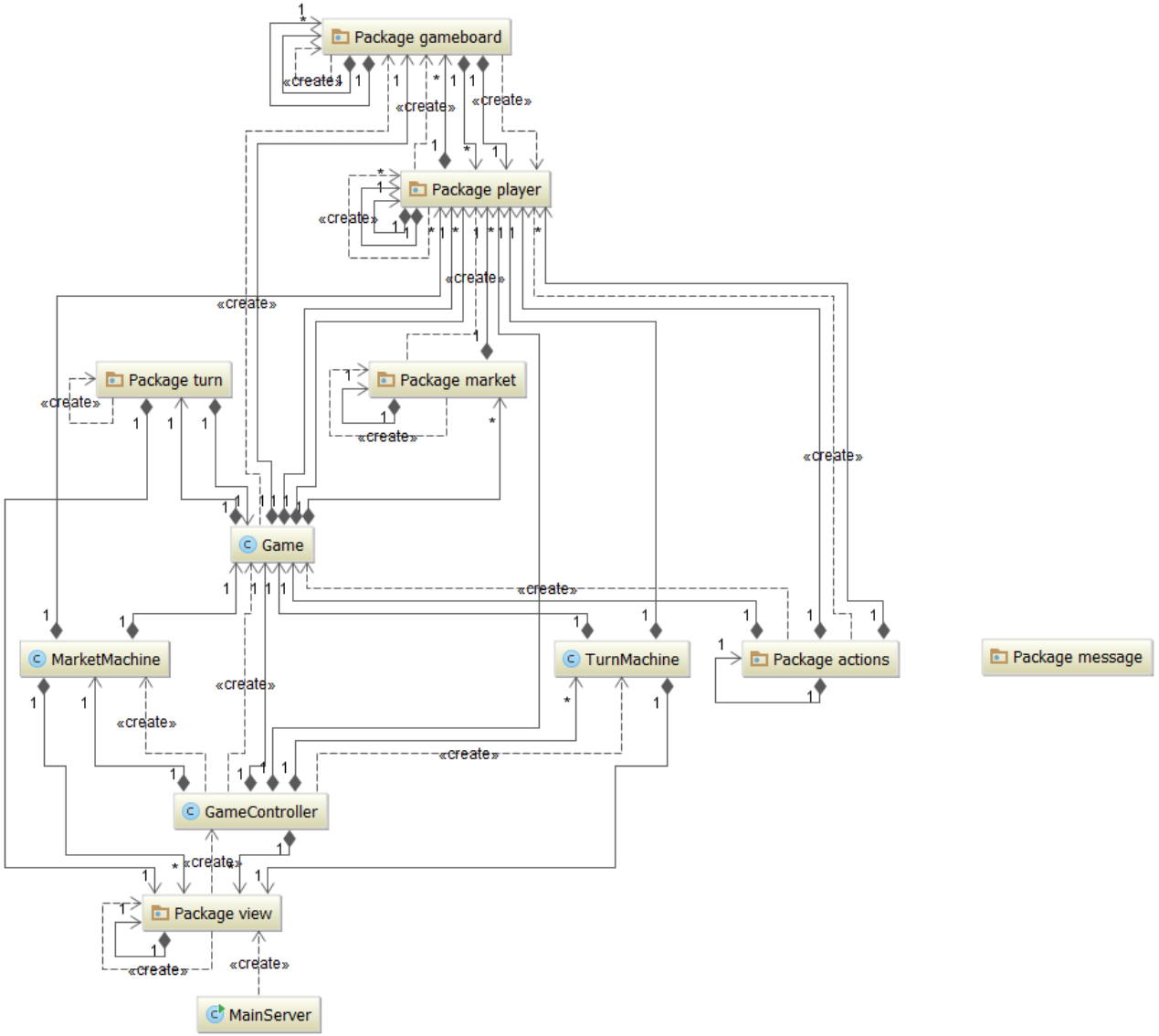


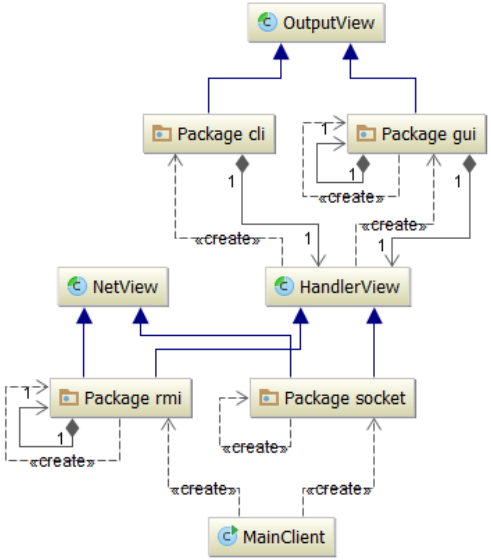
The **Observer/Observable pattern** is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

We used this pattern for the communications either in server and in client.

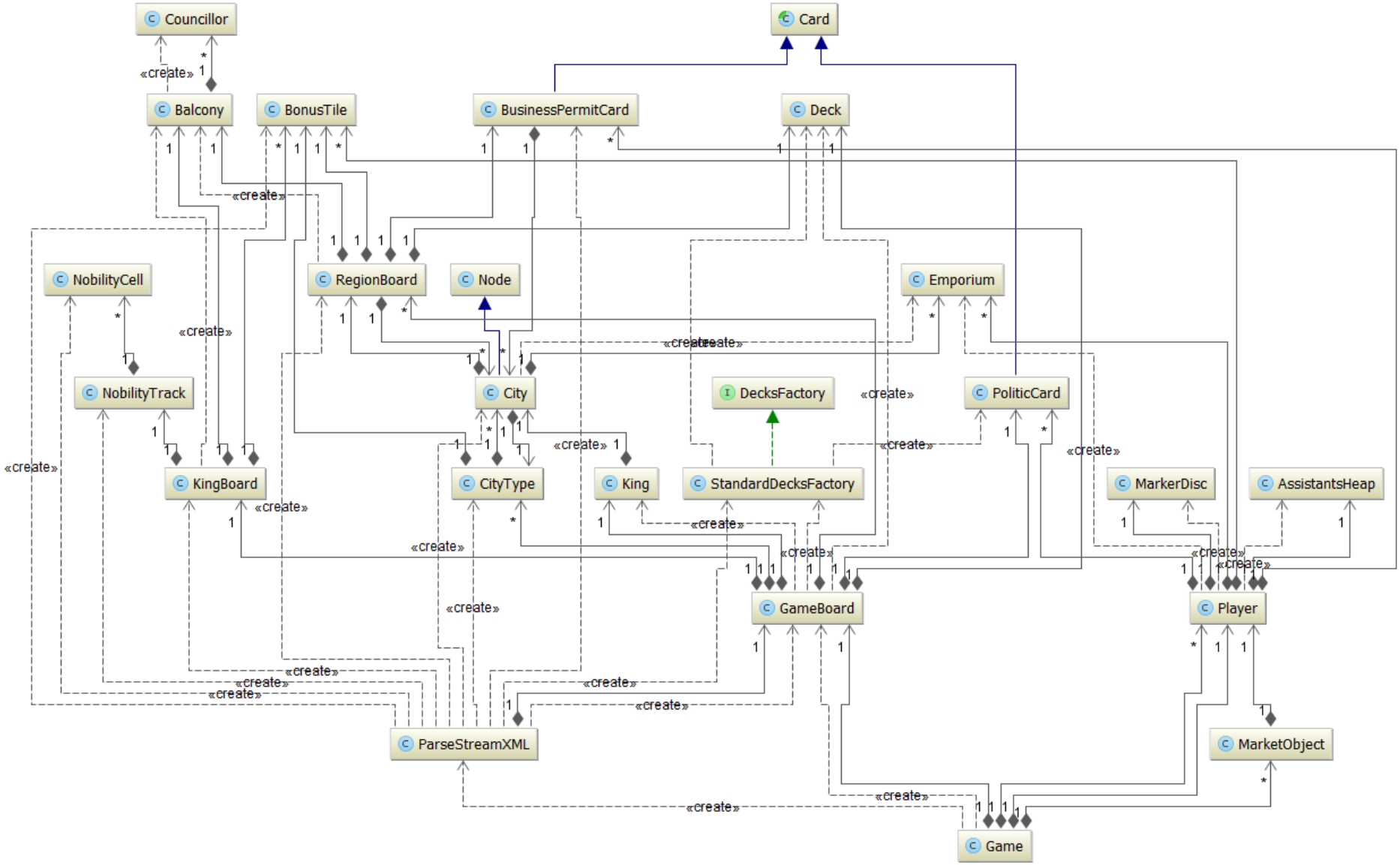
- View observe Model
- Model notify View (Game)
- Controller Observe View
- View Notify Controller (SocketView, RMIView)

MVC – Class Diagram

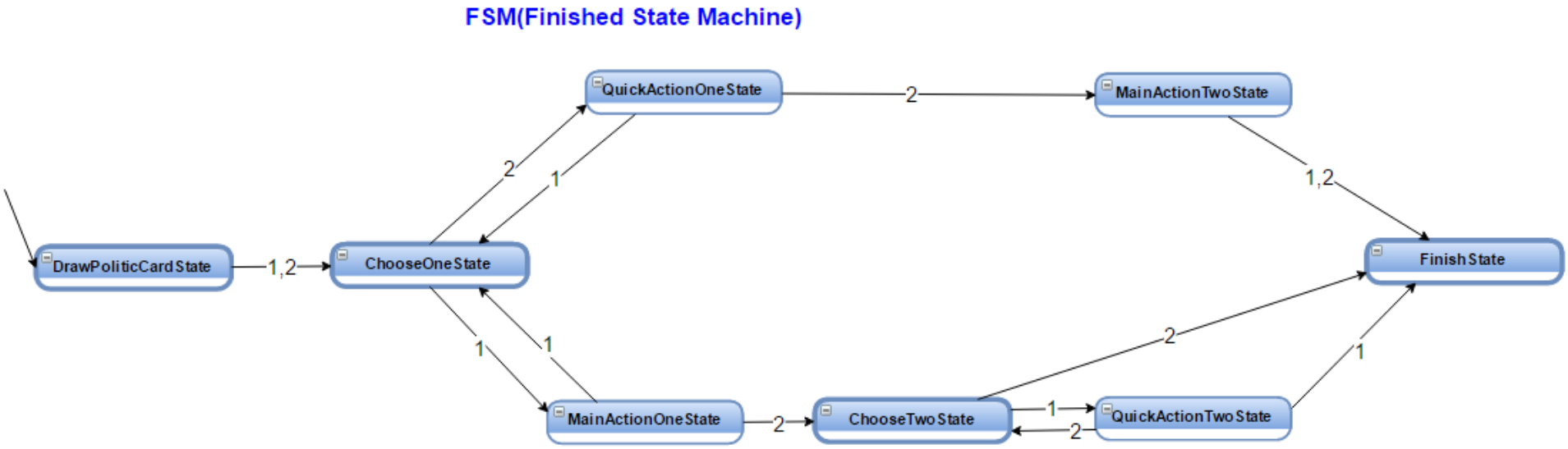




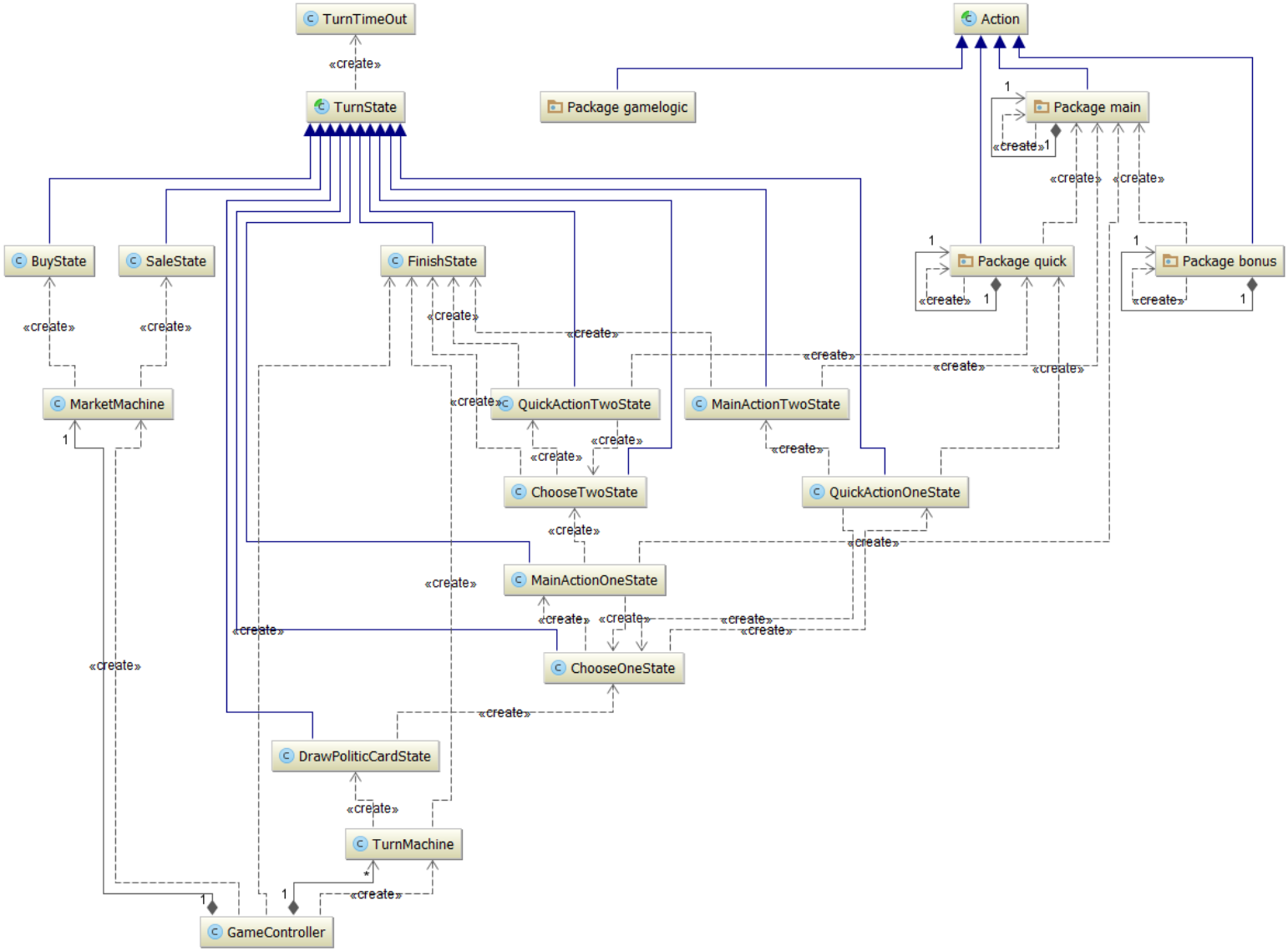
Server - Model - GameBoard



the whole game is managed by a **FSM** (Finished State Machine) implemented with a **State pattern**



Server - Model - GameTurn

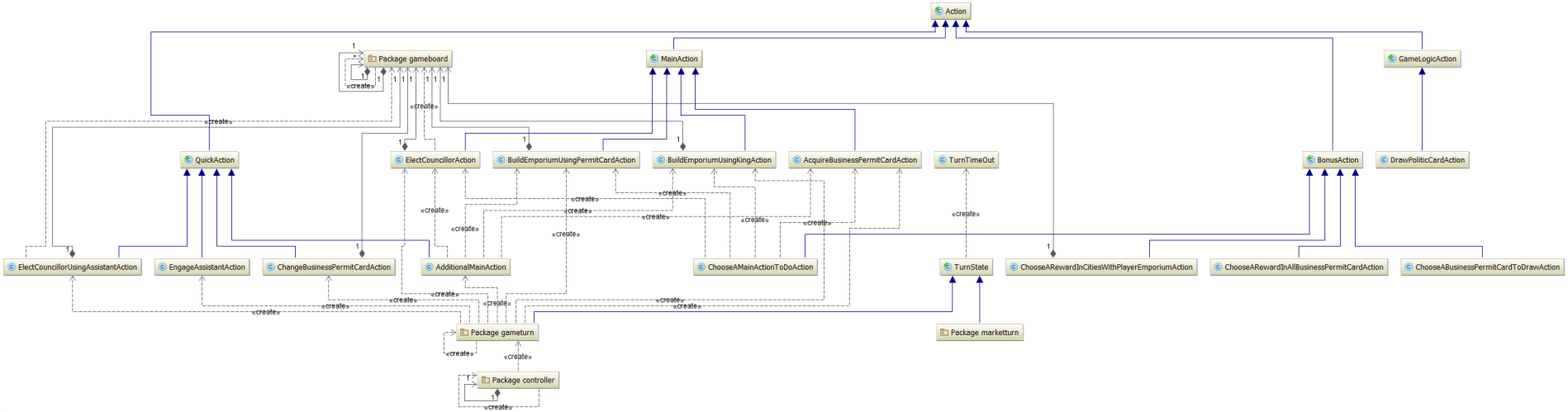


Actions are implemented very similarly to a **Command pattern**, and they determine the following state of the state machine based on their execution. After the execution, actions are also able to generate a pair of public broadcast messages to notify to all players what the current player is doing.

Actions:

- MainAction
- QuickAction
- GameLogicAction
- BonusAction

Server - Model - Actions

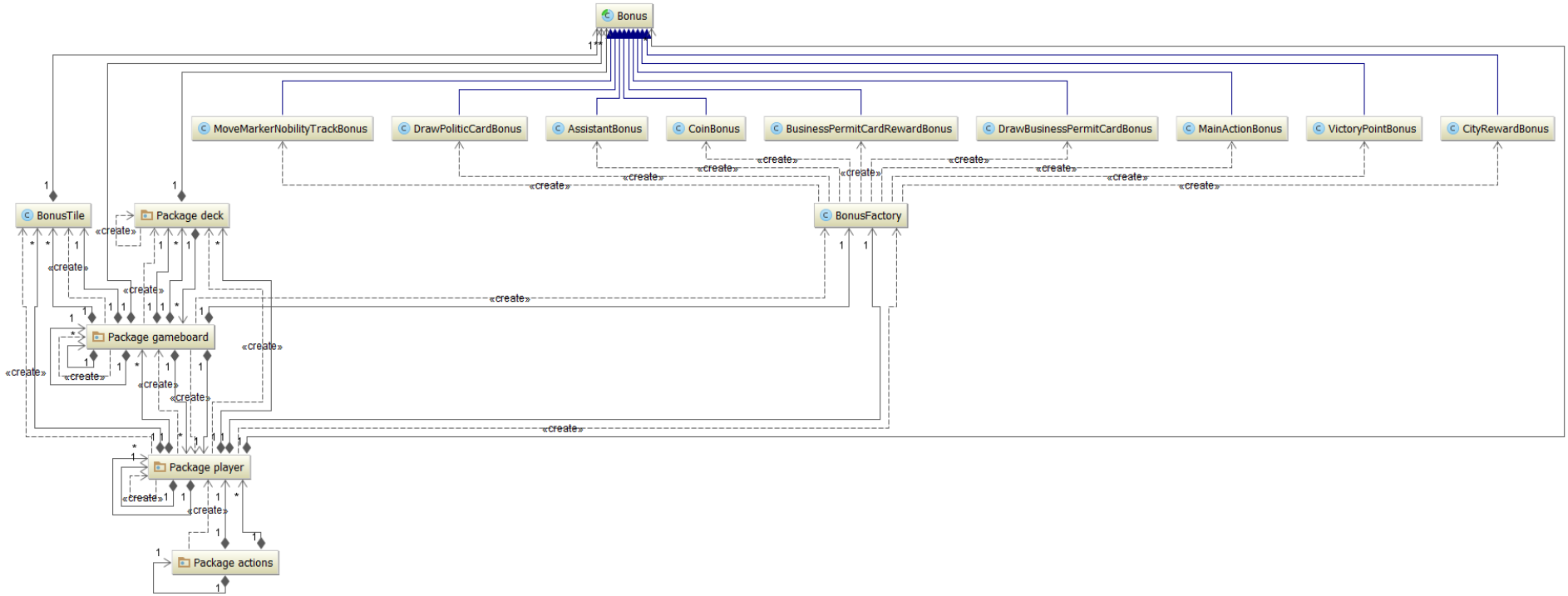


the **factory method pattern** is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.

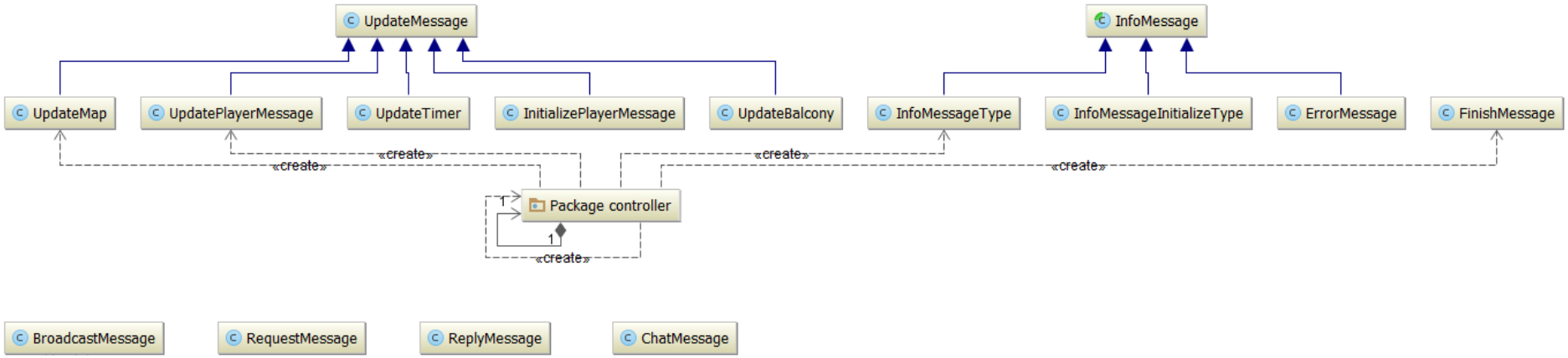
We used Factory Pattern either for create different bonuses and different decks

- Politic Card Deck
- Business Permit Card Deck

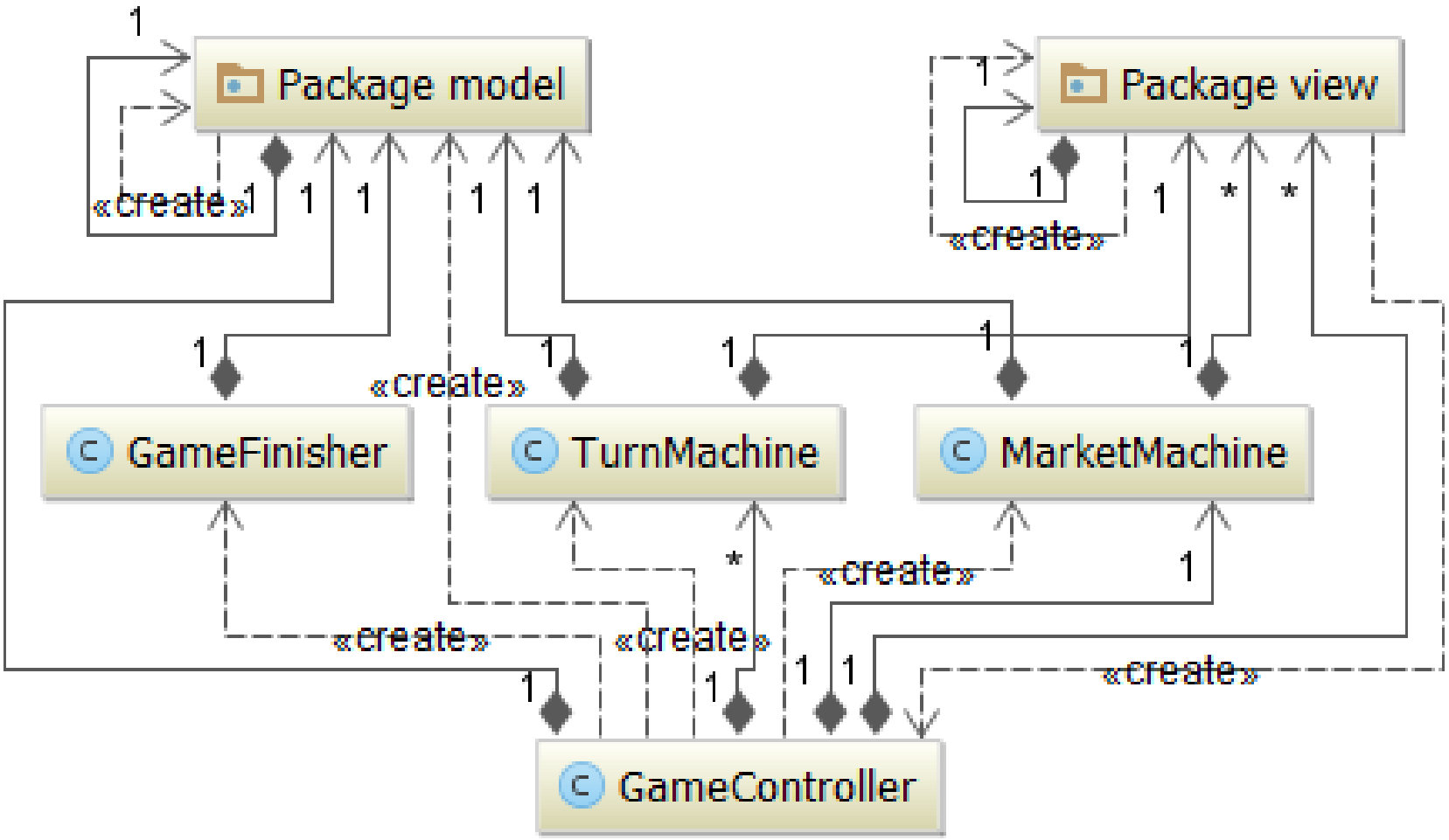
Server - Model - Bonus



Server - Model - Messages



Server - Controller



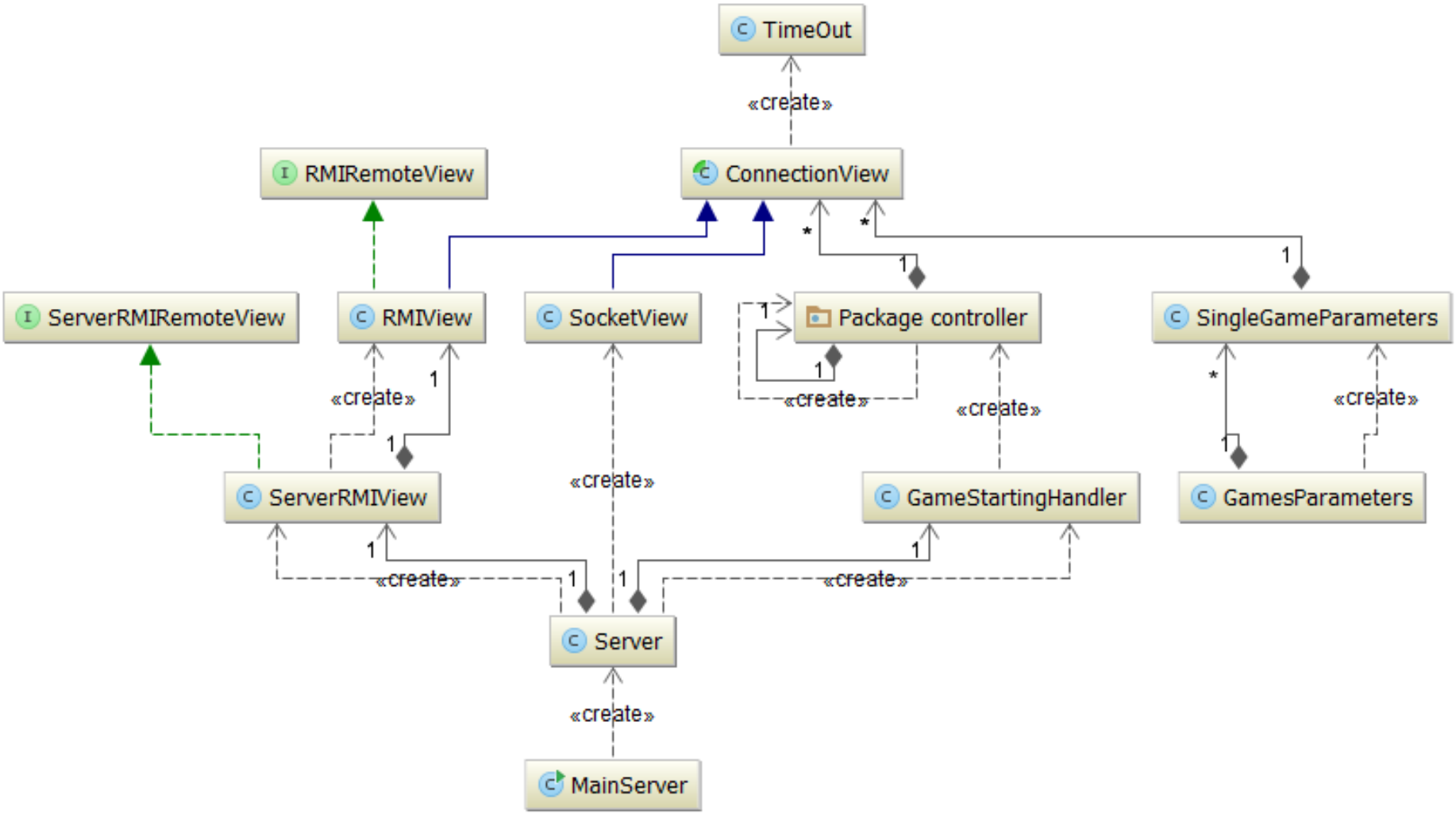
Server - View - Singleton

The Singleton class is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

It is responsible for maintaining, in a hashmap of information of all games at initialization, such as:

- Game Number
- List of Player
- List of View
- King Name
- Map Number
- Player Number

Server - View

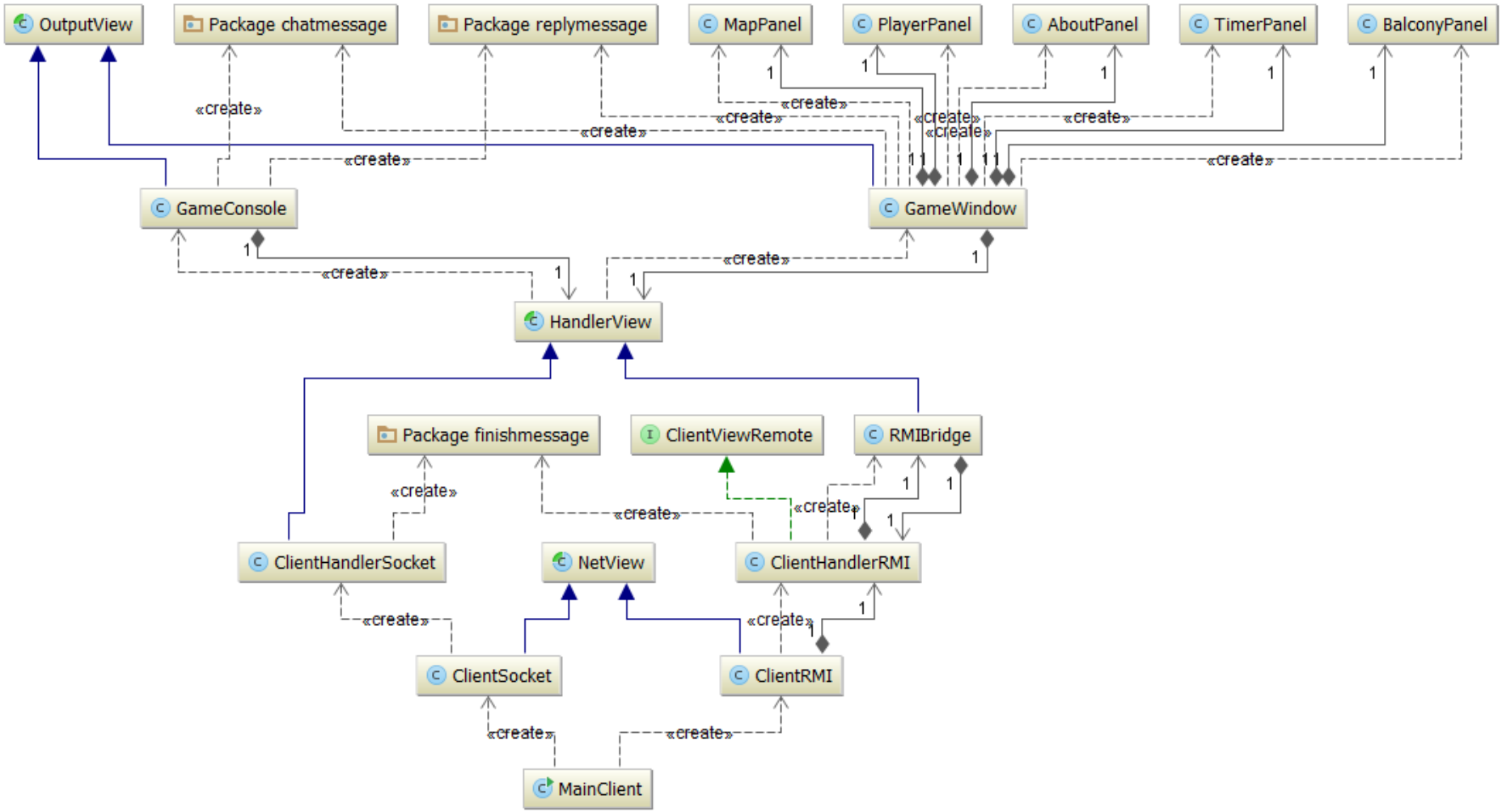

TimeOut
«create»
RMIRemoteView
ConnectionView
ServerRMIRemoteView
RMIView
SocketView
Package controller
SingleGameParameters
«create»
«create»
«create»
«create»
«create»
ServerRMIView
GameStartingHandler
GamesParameters
«create»
«create»
«create»
Server
«create»
MainServer
1
*

Client - Bridge Pattern

Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently.

We used a bridge between the client handler with the final player

Client


OutputView
Package chatmessage
Package replymessage
MapPanel
PlayerPanel
AboutPanel
TimerPanel
BalconyPanel
GameConsole
GameWindow
HandlerView
Package finishmessage
ClientViewRemote
RMIBridge
ClientHandlerSocket
NetView
ClientHandlerRMI
ClientSocket
ClientRMI
MainClient
«create»
1

Additional Functionality - Chat

We also implemented a very simple, intuitive ever active **chat** through which users can exchange messages in real time; messages are identified by the name of the player and the color chosen at the beginning of the game. Before the match starts the chat is not active, and if the player tries to write he is notified with an error message.