Submitted in fulfilment of the requirements for the degree of Bachelor of Computer Science, Honors

♣♥♦♠

Author: Siraj Ahmadzai

Supervisor: Professor Jean-Pierre Corriveau

Honors Project (COMP 4905A)

♣♥♦♠

Carleton University, 1125 Colonel By Dr, Ottawa, ON K15 5B6

January 17, 2019

Abstract

The goal of the project is to implement the card game Machiavelli using a reactive and proactive design pattern for handling multiplayer events on the server.

To develop the game, A simple client server architecture was developed using java, and javaFX was used to create the UI. Then events were specified, and event handlers were created to separate app from event handling logic. In the end, two separate versions based on a reactor and proactor demultiplexing and dispatching system were developed to register the event handlers and call upon them from the event loop to handle the client requests.

It was found that both approaches to event driven programming provided great benefits in the handling of clients. They provided separation of concerns and modularity by separating the application control flow from the event handling mechanisms.

The reactor waits for an event and then uses a demultiplexer which it passes it to the event handler for further processing. Since all of it is done synchronously, the demultiplexer blocks the rest of the awaiting events until the resources are available. On the other hand, the proactor works mostly in a similar way but starts processing each event asynchronously and instead of an event handler, uses a completion handler to notify the handler upon completion of a task. This provides improved performance where a lot of long-term operations need to be performed.

The game being a turn-based strategy game allowed the server to be analyzed under both kinds of concurrent processing techniques to highlight their strengths and liabilities in a practical manner.

Acknowledgements

I would like to thank Professor Jean-Pierre Corriveau for supervising me on this project. Their continuous guidance and support since the start of the project is what made this project possible. I would also like to express my deep gratitude to the department of Computer Science at Carleton University and all my professors and instructors for providing me the opportunity to hone my skills and acquire the knowledge which allowed me to accomplish what I have.

Finally, I would like to thank all my friends and family for their constant support, prayers and encouragement that helped me throughout university to strive to be a better version of myself.

Table of Contents

List of Figures

List of Tables

Introduction

General

Machiavelli is an Italian card game, derived from Rummy, usually played by 2 to 5 players. It is a party game played by many across the world. Its turn-based strategy nature allows for a great platform to be built to explore different event driven programming techniques. The motivation for this project is to develop a system to explore the reactor and proactor design patterns for event driven programming using a client/server architecture.

Scope

There are many variations of the game played throughout the world, but this project focuses on a simple version with the most basic set of rules where each player is given 15 cards at the start. The one to deal all the cards from their hand following the set of valid moves is the winner.

The system was designed to have separate modules for the client and server and follows an MVC paradigm. It tries to separate and specifically focus on the event handling mechanisms to explore the Reactor and Proactor patterns.

Background

Overview

This section gives a brief introduction to the ideas that are explored in this project. The reader should have a good understanding of the general programming principles along with the concepts described in this section before proceeding with the rest of the paper.

Event Driven Architectures

Event driven architecture promotes the development of efficient and robust concurrent and networked systems, using the production, detection, consumption, and reaction to events.

The architecture is divided in layers with a major focus on separation of concerns between the application functionalities and the actual event handling mechanisms.

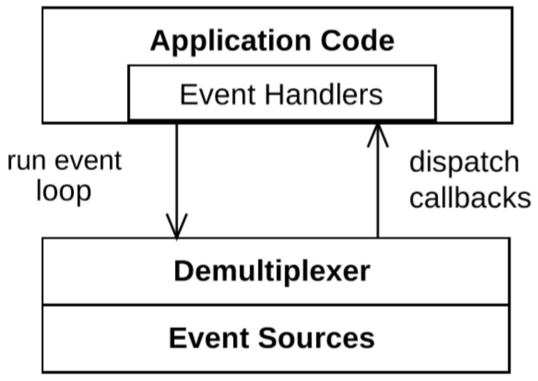


FIGURE 1: Layered Architecture of an event driven system.

Components of an Event driven architecture

The architecture can be built on four logical layers, an input layer which senses and registers an event, a layer which dispatches it to an appropriate handler, the event handling layer and the layer containing the actual application logic.

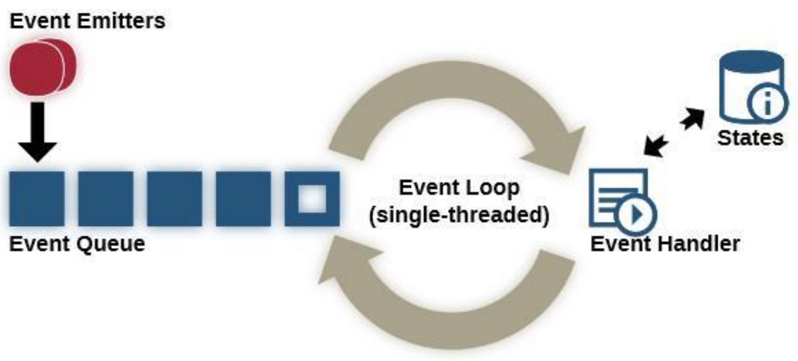


FIGURE 2: Event driven architecture

Event Source
Senses, Identifies and retrieves the events from the requests that arrive at the system. Indication events occur at the source which signal the system that a request has arrived for processing.

Demultiplexer
Waits for an event on the event source and passes the event to the event handler that is registered with the demultiplexer to handle it.

Event Handler
Processes the event using the application operations in response to the call made by the demultiplexer.

Application Code
The program logic which is used to process an event. This contains the operations that are implemented to serve the request.

Concurrency Strategies

Blocking I/O

The caller is blocked until resources are free to process their request and return the result. Only 1 request is served at any time. The rest are blocked until they can be served. This wastes resources.

Non-blocking Synchronous

Like blocking I/O where a single request is served at any moment except instead of blocking them, they are notified that resources to process their request are not available and they can try again later.

Non-blocking Asynchronous

A new thread is created to process each request concurrently so that multiple requests are being served at any given moment without blocking any of them.

Event driven systems can both be synchronous and asynchronous and use callbacks to notify on completion of the task requested.

Benefits of an Event Driven System

Separation of concerns

The functionality of modules is separated and contained in well-defined components each with a well-defined task.

Inversion of control

Helps in building an enhanced preemptive, reactive and a real-time system that is better able to handle unpredictable and non-linear behavior.

Increased Reusability

Modularity in processing of different events allows the same event handlers to be used elsewhere.

Improved Portability

Separation of the event handling mechanisms from event processing loop allows for greater portability as the handling mechanism can be reused on other platforms.

Reactor and Proactor Patterns

The reactor and proactor are event handling patterns that provide a way to implement the concurrency strategies given above. Both propose a different solution to initiate, receive, demultiplex and dispatch events in an event driven architecture.

Reactor Pattern

The reactor pattern allows the event-driven systems to receive requests simultaneously and serve them in a serial manner by demultiplexing them and dispatching them synchronously.

The actual event processing part is a single threaded component by nature of the design. The reactor allows indication events to occur which are picked up by the system to register a request. The reactor then returns the control to the caller immediately after either processing the request and returning with the result or with a notification that resources required to complete the request are currently unavailable.

Participants

Reactor
It's the core component of this design. Has an interface to register and remove event handlers from the system. Runs the main event loop to read indication events, provides inversion of control, uses the demultiplexer to get an event from its handle and dispatches the proper callback to process it.

Handle
It's an identifier for an event source which can provide indication events for requests that arrive from internal or external sources.

Event Handler
It’s an interface specification which provides a set of hook methods that can be used by the dispatcher to handle events.

Synchronous Event Demultiplexer

Blocks indication events until the resources are available to process their requests.

Concrete Event Handler

Implementation of the event handler interface. It specializes the interface for serving a particular request and implements the hook methods required for its processing.

Structure

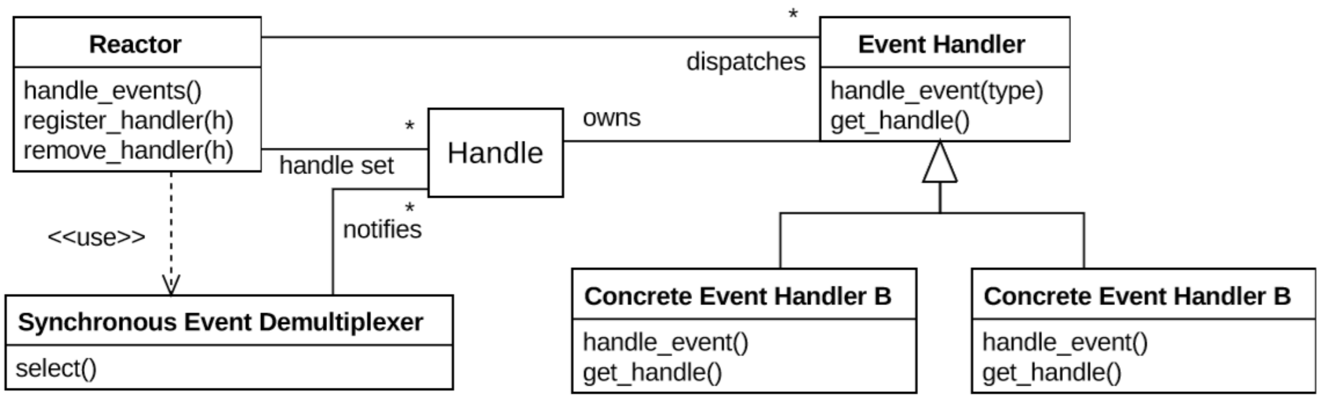


FIGURE 3: Structure of a reactor system

The Reactor registers all the appropriate event handlers and provides a mechanism to run the event loop. The event loop works with all the sources of events and then synchronously calls the Event Handler hook methods in its handle_events() method.

Dynamics

The control of the system alternatively flows between the Reactor and the Event Handlers.

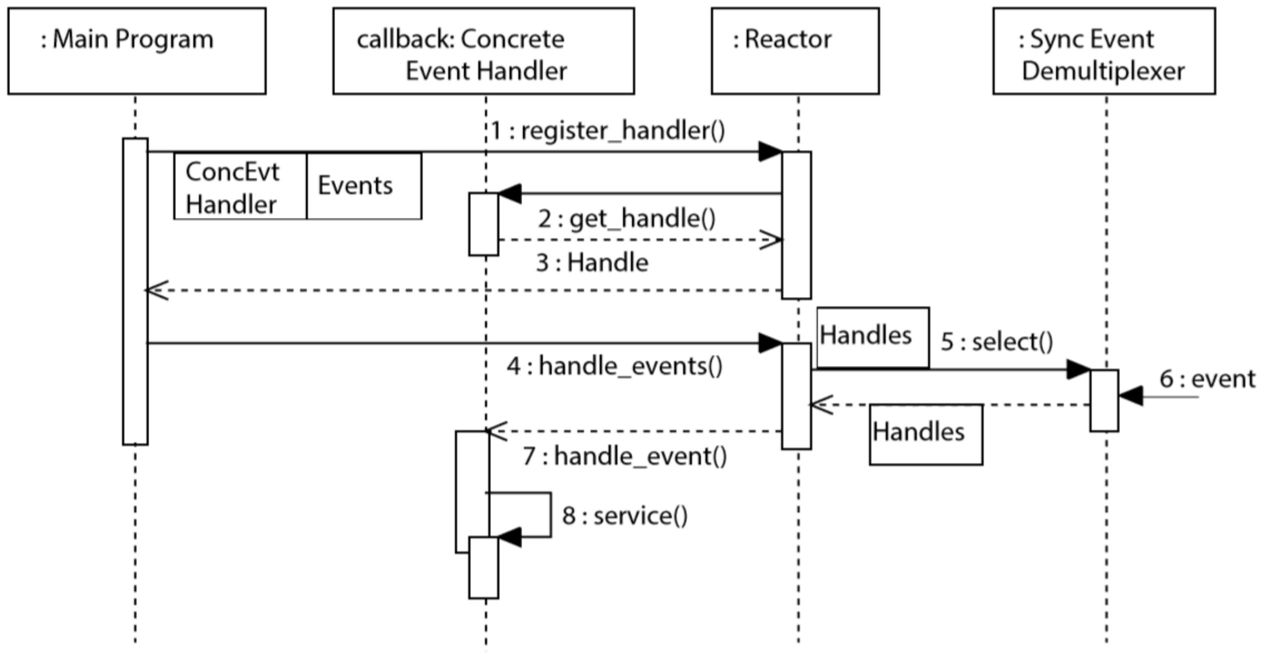


FIGURE 4: Inversion of control as the control is taken from the usual flow and given to the event handler.

The reactor’s responsibility is to wait for requests, listen for their indication events, demultiplex and dispatch them.
The event handlers are called through the hook methods they provide to the reactor for handling the event they provide service for.

Demultiplexing Mechanisms

select()
Its portable but inefficient. With O(n) descriptor selection, limited to 1024 descriptors, stateless.

poll()
It allows better and more fine-grained control of events, but still O(n) descriptor selection, stateless.

epool()
It keeps info, dynamic descriptor set, efficient with O(1) descriptor selection, only available on Linux platforms.

kqueue()
It’s a more general mechanism, O(1) descriptor selection, only on Mac OS X and FreeBSD systems

WaitForMultipleObjects()
works on multiple types of synchronization objects, only on Windows

ACE and Boost libraries supply a common interface to choose the best Reactor implementation depending on the execution platform support

On Places where there is no support for native synchronous demultiplexing, multiple threads can be used within the implementation to emulate this.

Advantages and Disadvantages

Advantages	Disadvantages
Increase separation of concern	Non-Preemptive
Better Modularity, Reusability, and Configurability	Decreased Scalability
Better Portability	Development Complexity
Lightweight concurrency control	

Table 1: Advantages/Disadvantages of Reactor based event handling

Proactor Pattern

The proactor pattern allows the event-driven systems to receive requests simultaneously and serve them in a robust and efficient manner by demultiplexing them and dispatching them asynchronously.

Instead of the **indication events** like in reactor, it waits for **completion events.**

The actual event processing part is a multithreaded component by nature of the design. The proactor creates a separate thread and returns the control to the caller immediately indicating that the request is being processed. The completion event is then used to notify the caller when the operation finishes.

Participants

Proactor
It's the core component of this design. Has an interface to register and remove event handlers from the system. Runs the main event loop to read indication events, provides inversion of control, uses the demultiplexer to get an event from its handle and dispatches the proper callback to process it.

Handle
It's an identifier for an event source which can provide completion events for requests that arrive from internal or external sources.

Completion Handler
It’s an interface specification which provides a set of hook methods that can be used to process the results of an asynchronous operation.

Asynchronous Event Demultiplexer
Blocks completion events until they are added to a completion queue for the results to be sent back to the callers.

Concrete Completion Handler

Implementation of the completion handler interface. It specializes the interface for serving a particular request and implements the hook methods required for its processing.

Completion Event Queue

Keeps the completion events temporarily until they are demultiplexed and dispatched to the completion handlers.

Asynchronous Operation

Operations which might run for a long duration and can be used as a service.

Asynchronous Operation Processor

Creates a new thread to execute each asynchronous operation, then generates a completion event and adds it to the completion event queue.

Initiator

Starts the event handling, by registering the proactor and the completion handlers with the asynchronous operation processor, to process the asynchronous operation.

Structure

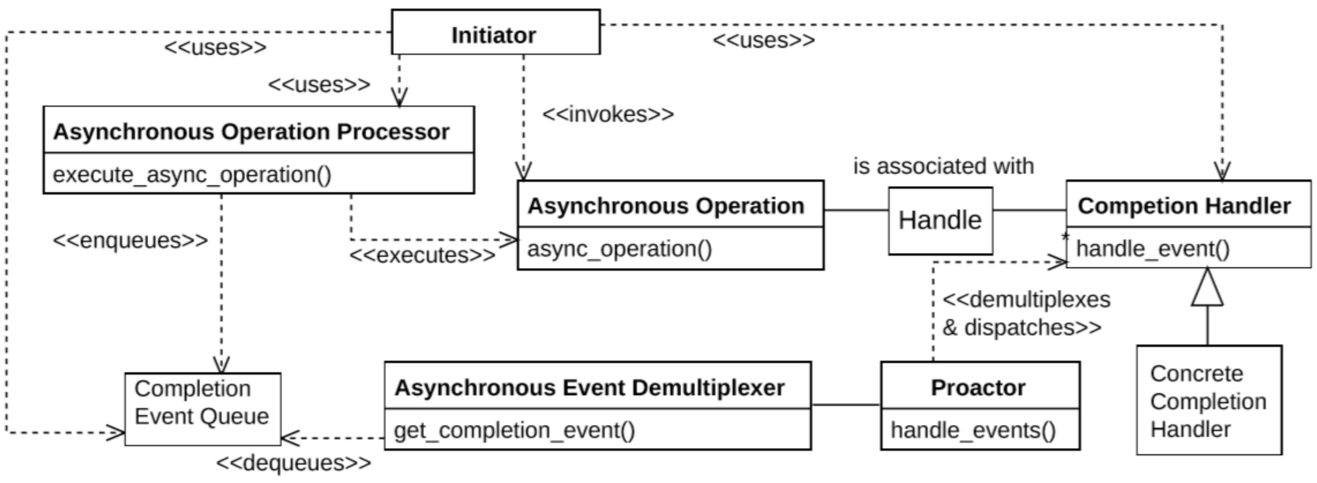


FIGURE 5: Structure of a Proactor System

The Proactor pattern allows all the operations to be invoked asynchronously. The Proactor keeps a queue of the completed operations and demultiplexes and dispatches the hook methods of the completed events' handlers in the event loop.

Dynamics

The operations are invoked asynchronously in separate threads while the control of the main thread flows between the proactor and the completion event handlers.

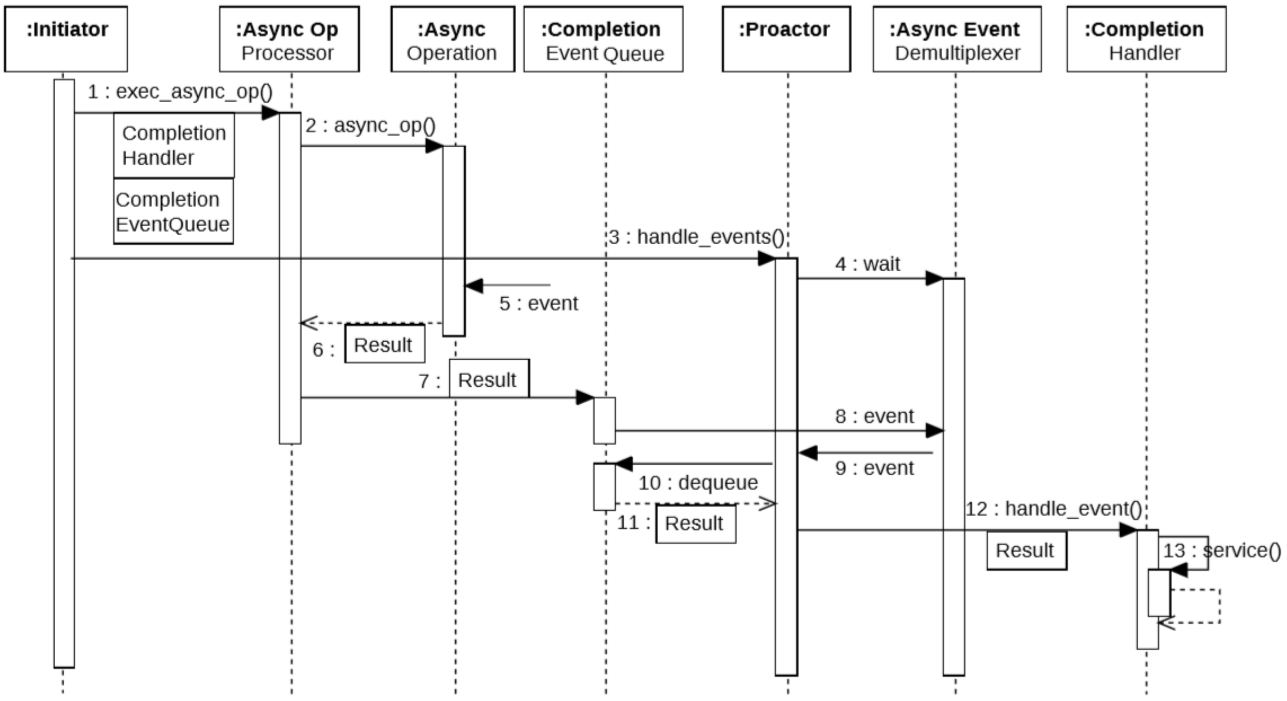


FIGURE 6: The async operation is executed immediately by the initiator, and its result is added to the completion event queue.

The Initiator immediately calls the Asynchronous operation processor to complete the operation asynchronously and registers the completion handlers with the proactor.
The proactor's responsibility is to run an event loop and dispatch event completion handlers for the completed events in the queue.
The completed event handlers are called through the hook methods they provide to the proactor for handling the event they provide service for.

Every application is split into long-duration asynchronous operations, and completion handlers which process the results of these long-duration operations

Advantages and Disadvantages

Advantages	Disadvantages
Increase separation of concern	No control over scheduling of operations
Better Modularity, Reusability, and Configurability	Platform dependent efficiency
Better Portability	Development Complexity
Lightweight concurrency control	
Encapsulation of concurrency mechanisms	
Concurrency Policy independent from threading policy	
Simplified Synchronization	

Table 2: Advantages/Disadvantages of Proactive Event Handling

Methodology

Overview

This section gives a brief look at how different components of the system were proposed to be created. The paradigms that were considered and the approach that culminated into the success of the project.

System Specification

Machiavelli is an Italian card game, derived from Rummy, usually played by 2 to 5 players, but can be played by more. It is a party game played by many across the world.
Its appearance among card games can be traced to the Second World War.

Play requires two decks of 52 standard playing cards, excluding the jokers. The main objective of the game is to play all the cards in your hand.

The system will be implementing the above game using two event driven programming techniques to study their differences as well as their strengths and weaknesses.

Functional Requirements

ID	Name	Specification
FR-1	Render	The system should be able to render all the game elements
FR-2	Game State	The system should be able to validate the state of game at each moment
FR-3	Server Listening	The system should be able to act as a server and listen for client connections
FR-4	Client Connection	The system should be able to accept a set number of clients to start a game.
FR-5	Card Management	The system should allow a player to manage the cards they have on hand and make them able to put any on the table in a way which is valid by the game rules
FR-6	Game Rules	The system should be able to enforce the rules of the game
FR-7	Error Messages	The system should be able to display appropriate error messages in case the user does something invalid
FR-8	Turn Management	The system should be able to let each player play their hand on their turns only.
FR-9	Turn Memorization	Until a move is finalized, the system should be able to keep track of what the player is doing so that it can be reverted if the player changes their mind.
FR-10	Reorganization	The system should allow the user to reorganize any card on the table.

Table 3: Functional Requirements of the system

Non-Functional Requirements

ID	Name	Specification
NFR-1	Security	Each player should be able to authenticate securely and be in control of their own gameplay.
NFR-2	Reliability	The system should work in a consistent and reliable way to make sure the gameplay is not affected by disconnects or server blocking.
NFR-3	Response Time	The system should be able to update its internal state and all the clients connected within
NFR-4	Efficiency	The system should be able to timely notify the clients of the result of their requests and perform all operations in a non-blocking manner.
NFR-5	Concurrency	The system should respond to requests made by all the clients connected regardless of any of them running a long operation

Table 4: Non-Functional Requirements of the system

System Design

The specially designed to explore the event driven approach for handling requests. So, a simple **client/server architecture** was designed and expanded upon to create the required functionality.

Since the system must be graphical, **MVC** approach was chosen to separate the presentation layer from the backend logic.

The system must allow concurrent requests to be handled so a basic event driven approach was designed which was later expanded upon by the **proactor** and **reactor** designs.

The system must be a turn-based game, so to keep track of all the turns and cards during play, the **memento pattern** was used.

Game states were created to signal each player for their turn to play or their turn to wait for the other players to finish. All the signaling and communication between the clients and the server was performed with the help of **commands**.

System Architecture

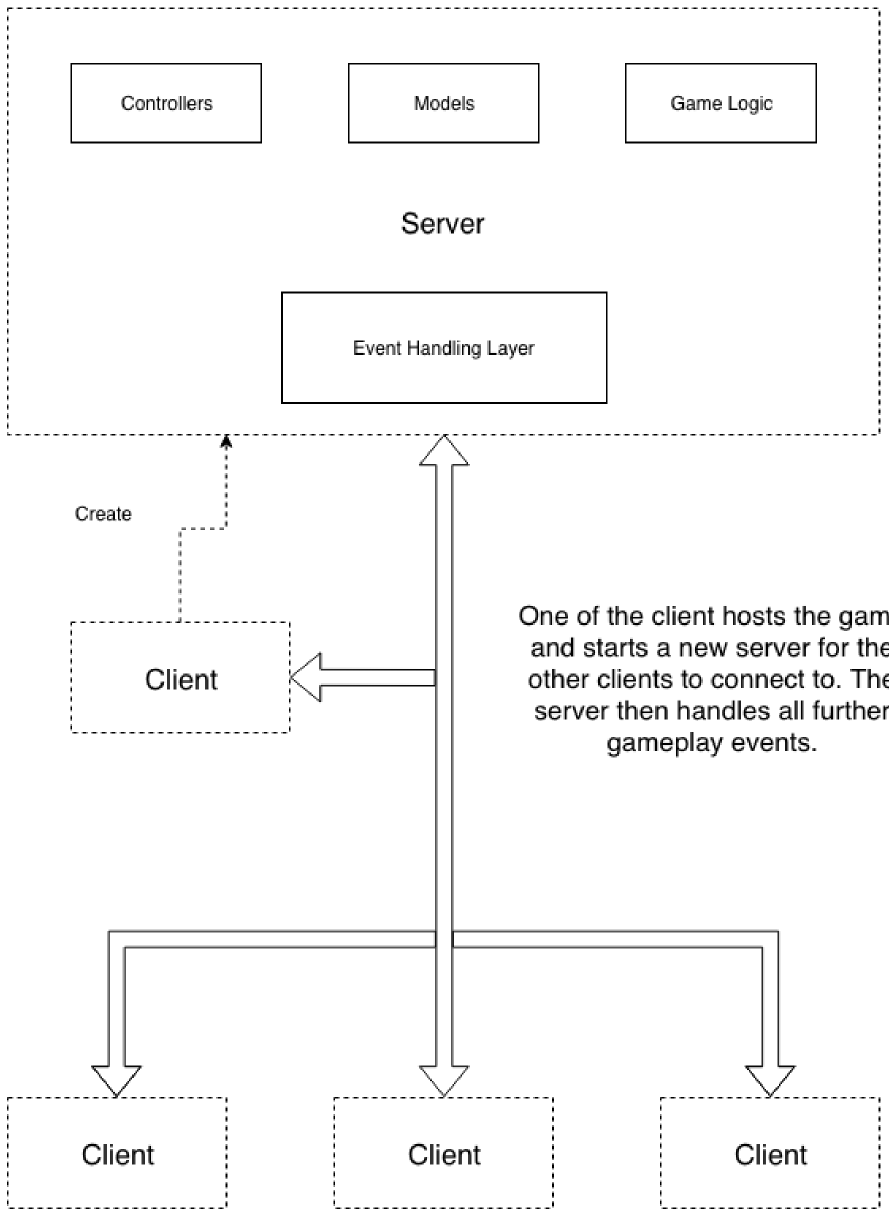

Controllers
Models
Game Logic
Server
Event Handling Layer
Create
Client
One of the client hosts the game and starts a new server for the other clients to connect to. The server then handles all further gameplay events.
Client
Client
Client

FIGURE 7: Overall architecture of the proposed system

The system designed to follow such a client and server architecture where a client chooses to be a “host” for the game and creates a server for others to connect to. The server module then initializes and starts listening and serving incoming requests.

Thus, everyone has the option to either join a game or start their own and let others join it.

Modules

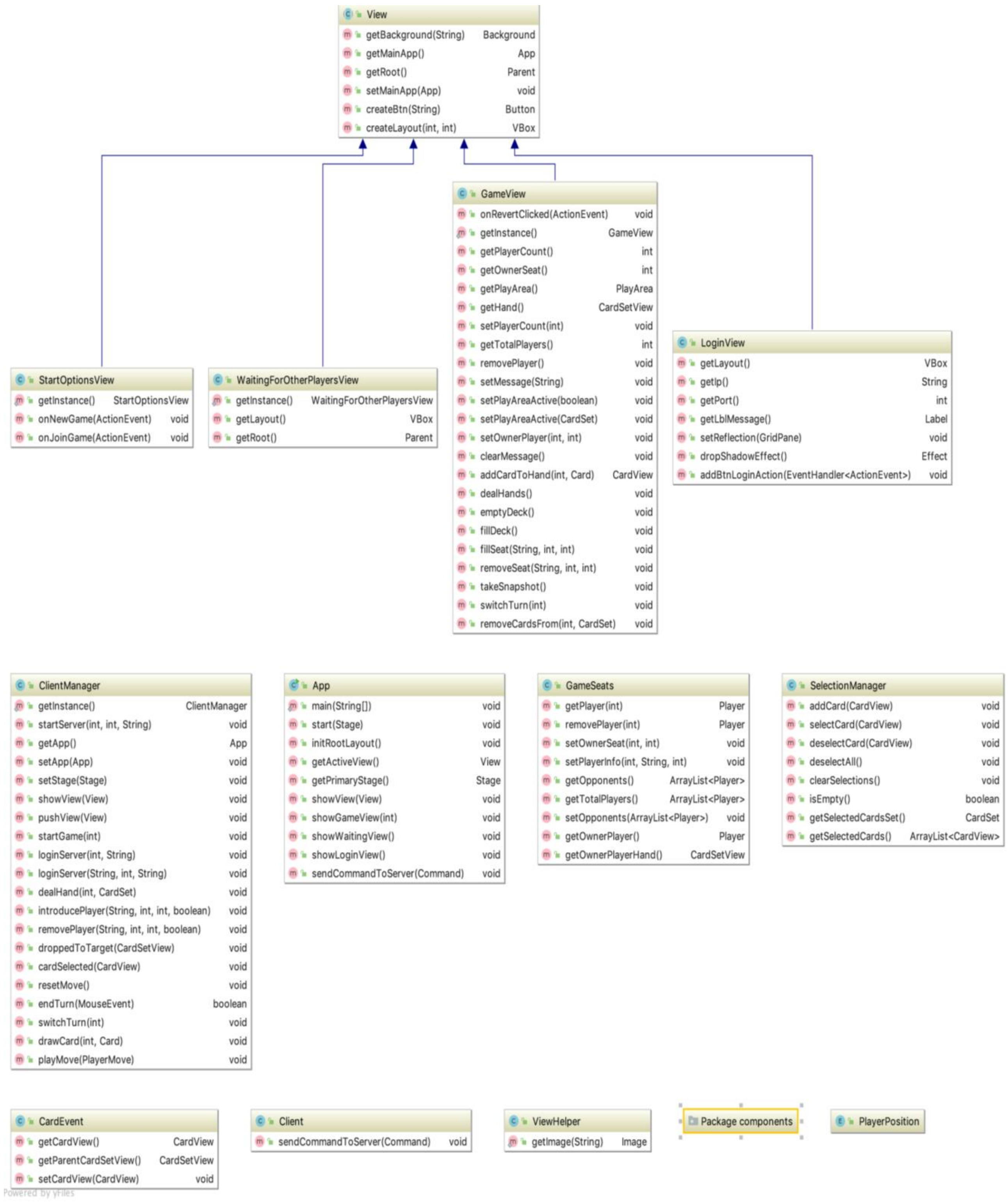

View
getBackground(String) Background
getMainApp() App
getRoot() Parent
setMainApp(App) void
createBtn(String) Button
createLayout(int, int) VBox
GameView
onRevertClicked(ActionEvent) void
getInstance() GameView
getPlayerCount() int
getOwnerSeat() int
getPlayArea() PlayArea
getHand() CardSetView
setPlayerCount(int) void
getTotalPlayers() int
removePlayer() void
setMessage(String) void
setPlayAreaActive(boolean) void
setPlayAreaActive(CardSet) void
setOwnerPlayer(int, int) void
clearMessage() void
addCardToHand(int, Card) CardView
dealHands() void
emptyDeck() void
fillDeck() void
fillSeat(String, int, int) void
removeSeat(String, int, int) void
takeSnapshot() void
switchTurn(int) void
removeCardsFrom(int, CardSet) void
LoginView
getLayout() VBox
getIp() String
getPort() int
getLblMessage() Label
setReflection(GridPane) void
dropShadowEffect() Effect
addBtnLoginAction(EventHandler<ActionEvent>) void
StartOptionsView
getInstance() StartOptionsView
onNewGame(ActionEvent) void
onJoinGame(ActionEvent) void
WaitingForOtherPlayersView
getInstance() WaitingForOtherPlayersView
getLayout() VBox
getRoot() Parent
ClientManager
getInstance() ClientManager
startServer(int, int, String) void
getApp() App
setApp(App) void
setStage(Stage) void
showView(View) void
pushView(View) void
startGame(int) void
loginServer(int, String) void
loginServer(String, int, String) void
dealHand(int, CardSet) void
introducePlayer(String, int, int, boolean) void
removePlayer(String, int, int, boolean) void
droppedToTarget(CardSetView) void
cardSelected(CardView) void
resetMove() void
endTurn(MouseEvent) boolean
switchTurn(int) void
drawCard(int, Card) void
playMove(PlayerMove) void
App
main(String[]) void
start(Stage) void
initRootLayout() void
getActiveView() View
getPrimaryStage() Stage
showView(View) void
showGameView(int) void
showWaitingView() void
showLoginView() void
sendCommandToServer(Command) void
GameSeats
getPlayer(int) Player
removePlayer(int) Player
setOwnerSeat(int, int) void
setPlayerInfo(int, String, int) void
getOpponents() ArrayList<Player>
getTotalPlayers() ArrayList<Player>
setOpponents(ArrayList<Player>) void
getOwnerPlayer() Player
getOwnerPlayerHand() CardSetView
SelectionManager
addCard(CardView) void
selectCard(CardView) void
deselectCard(CardView) void
deselectAll() void
clearSelections() void
isEmpty() boolean
getSelectedCardsSet() CardSet
getSelectedCards() ArrayList<CardView>
CardEvent
getCardView() CardView
getParentCardSetView() CardSetView
setCardView(CardView) void
Client
sendCommandToServer(Command) void
ViewHelper
getImage(String) Image
Package components
PlayerPosition
Powered by yFiles

FIGURE 8: Clients Module

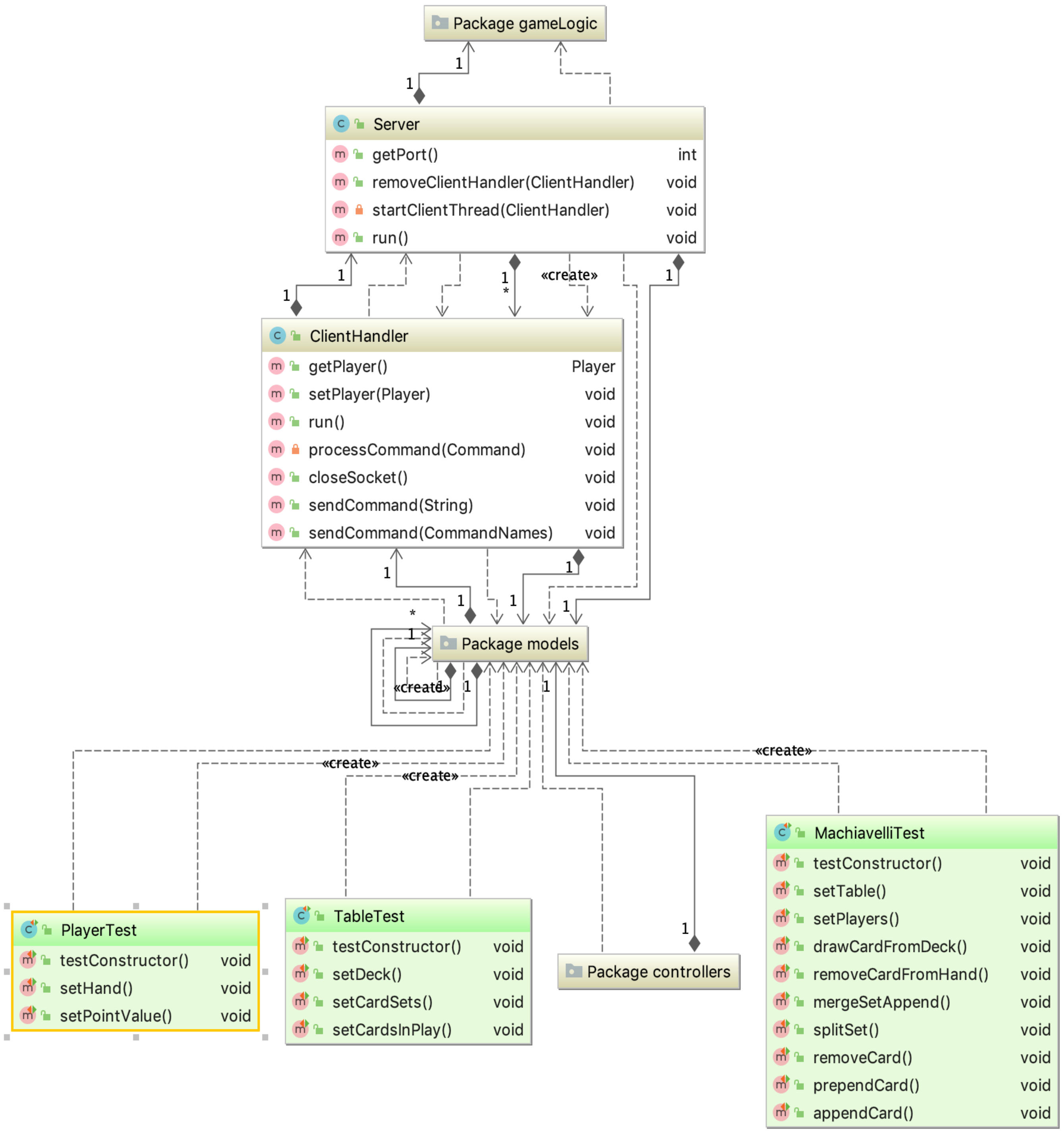
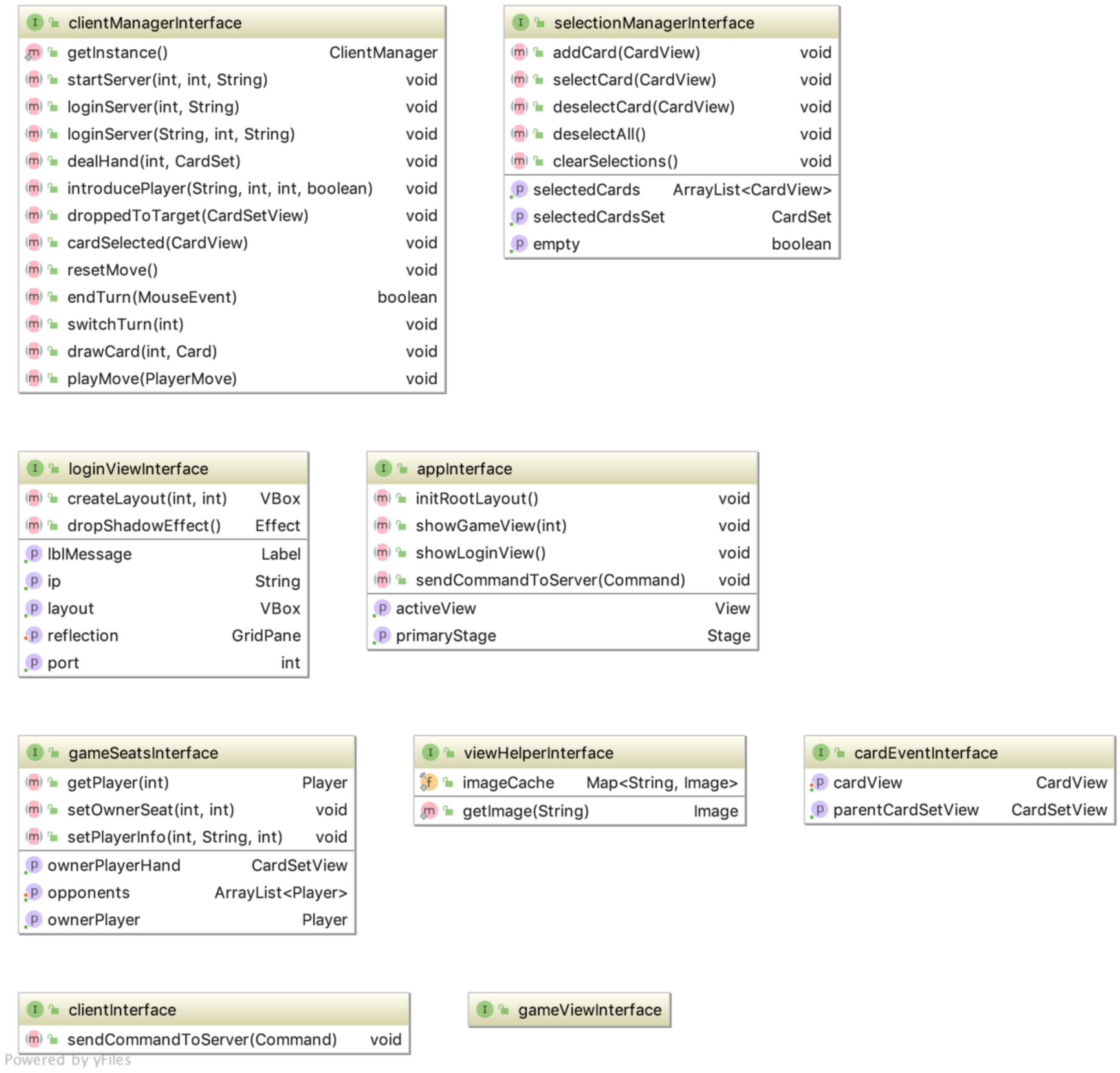


FIGURE 9: Server Module



clientManagerInterface	
getInstance()	ClientManager
startServer(int, int, String)	void
loginServer(int, String)	void
loginServer(String, int, String)	void
dealHand(int, CardSet)	void
introducePlayer(String, int, int, boolean)	void
droppedToTarget(CardSetView)	void
cardSelected(CardView)	void
resetMove()	void
endTurn(MouseEvent)	boolean
switchTurn(int)	void
drawCard(int, Card)	void
playMove(PlayerMove)	void

selectionManagerInterface	
addCard(CardView)	void
selectCard(CardView)	void
deselectCard(CardView)	void
deselectAll()	void
clearSelections()	void
selectedCards	ArrayList<CardView>
selectedCardsSet	CardSet
empty	boolean

loginViewInterface	
createLayout(int, int)	VBox
dropShadowEffect()	Effect
lblMessage	Label
ip	String
layout	VBox
reflection	GridPane
port	int

appInterface	
initRootLayout()	void
showGameView(int)	void
showLoginView()	void
sendCommandToServer(Command)	void
activeView	View
primaryStage	Stage

gameSeatsInterface	
getPlayer(int)	Player
setOwnerSeat(int, int)	void
setPlayerInfo(int, String, int)	void
ownerPlayerHand	CardSetView
opponents	ArrayList<Player>
ownerPlayer	Player

viewHelperInterface	
imageCache	Map<String, Image>
getImage(String)	Image

cardEventInterface	
cardView	CardView
parentCardSetView	CardSetView

clientInterface	
sendCommandToServer(Command)	void

gameViewInterface

Powered by yFiles

FIGURE 10: Interfaces Module

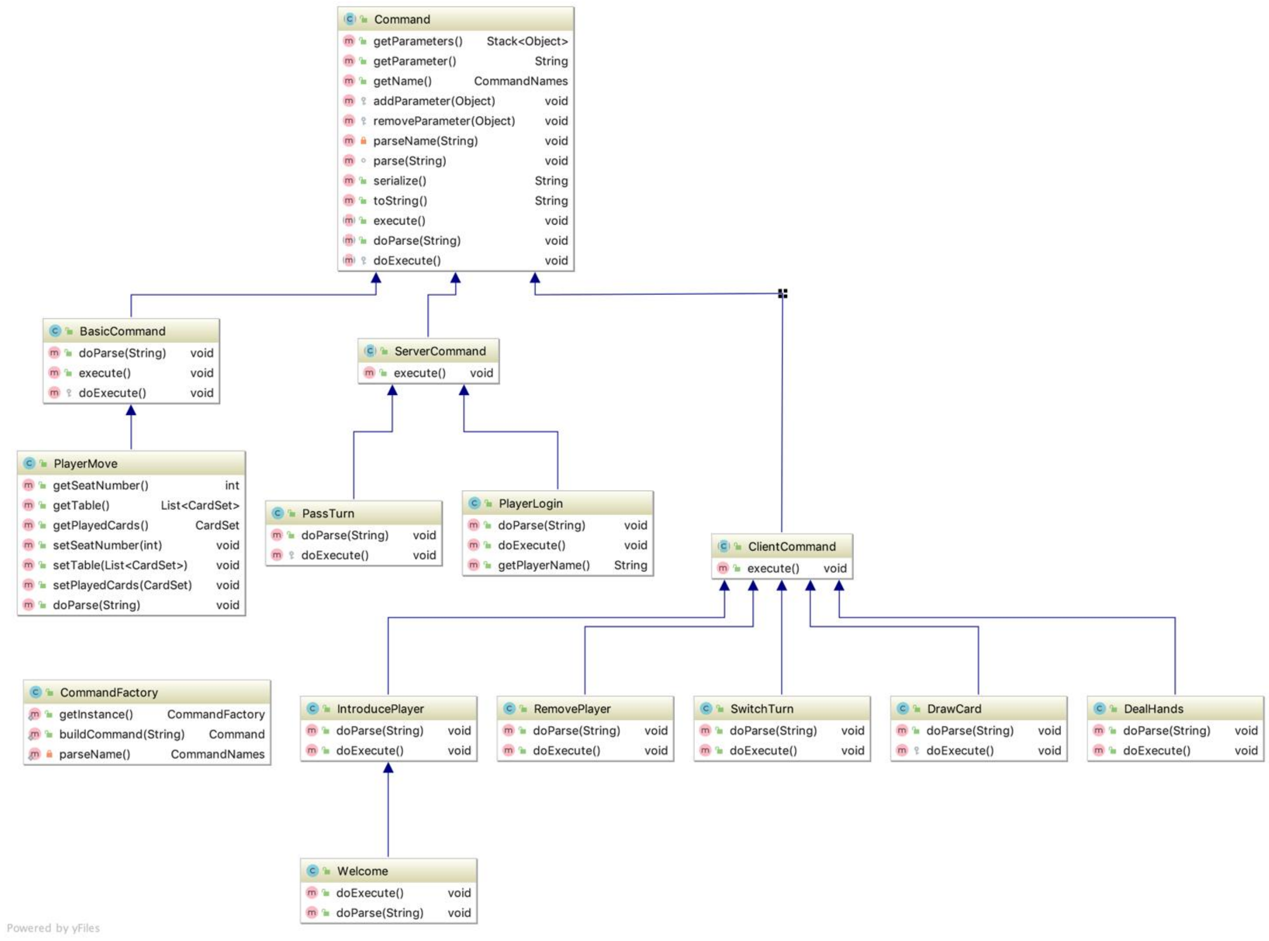


FIGURE 11: Commands Module

Dependency Diagrams

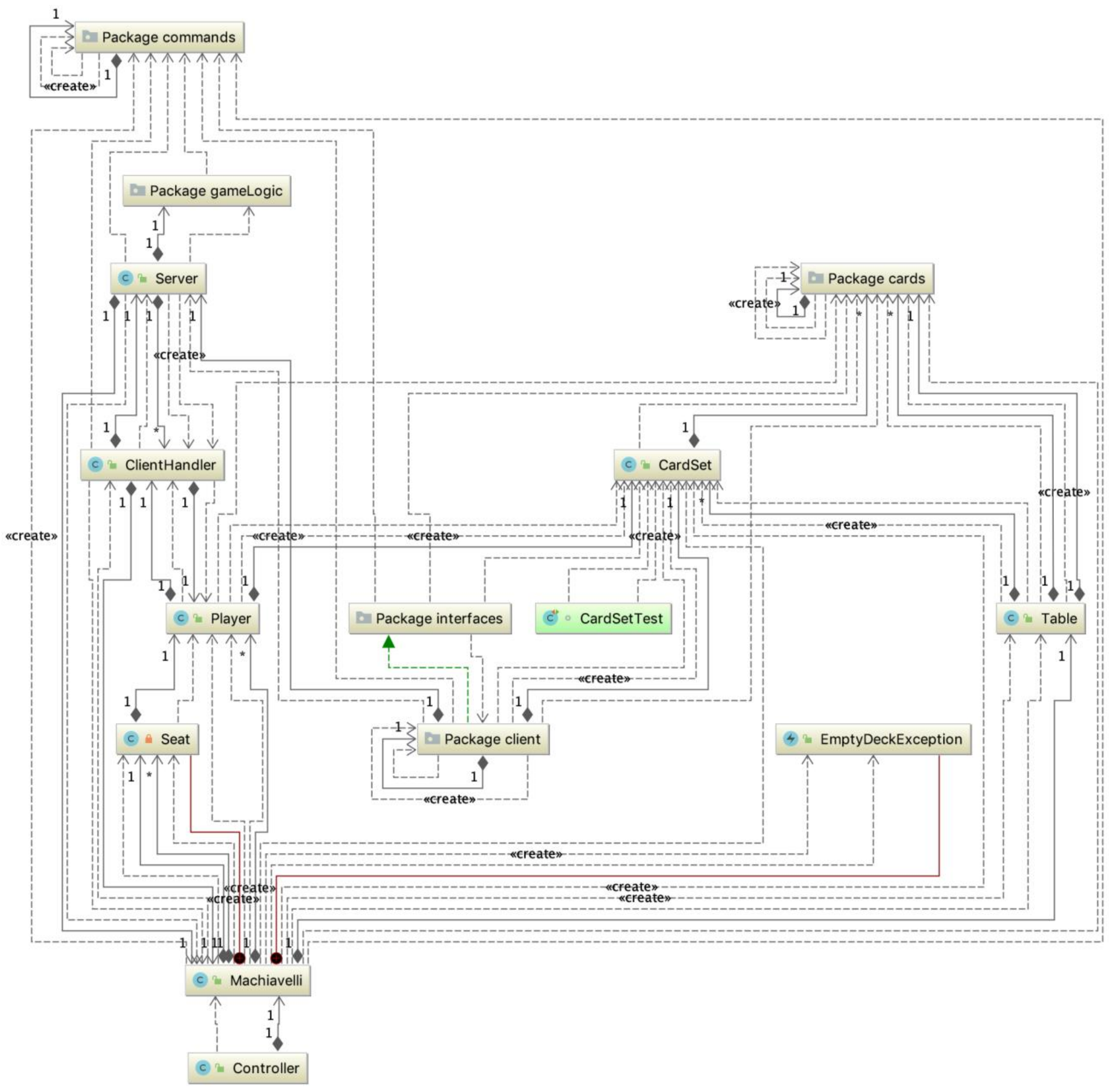


FIGURE 12: SERVER-SIDE Dependencies

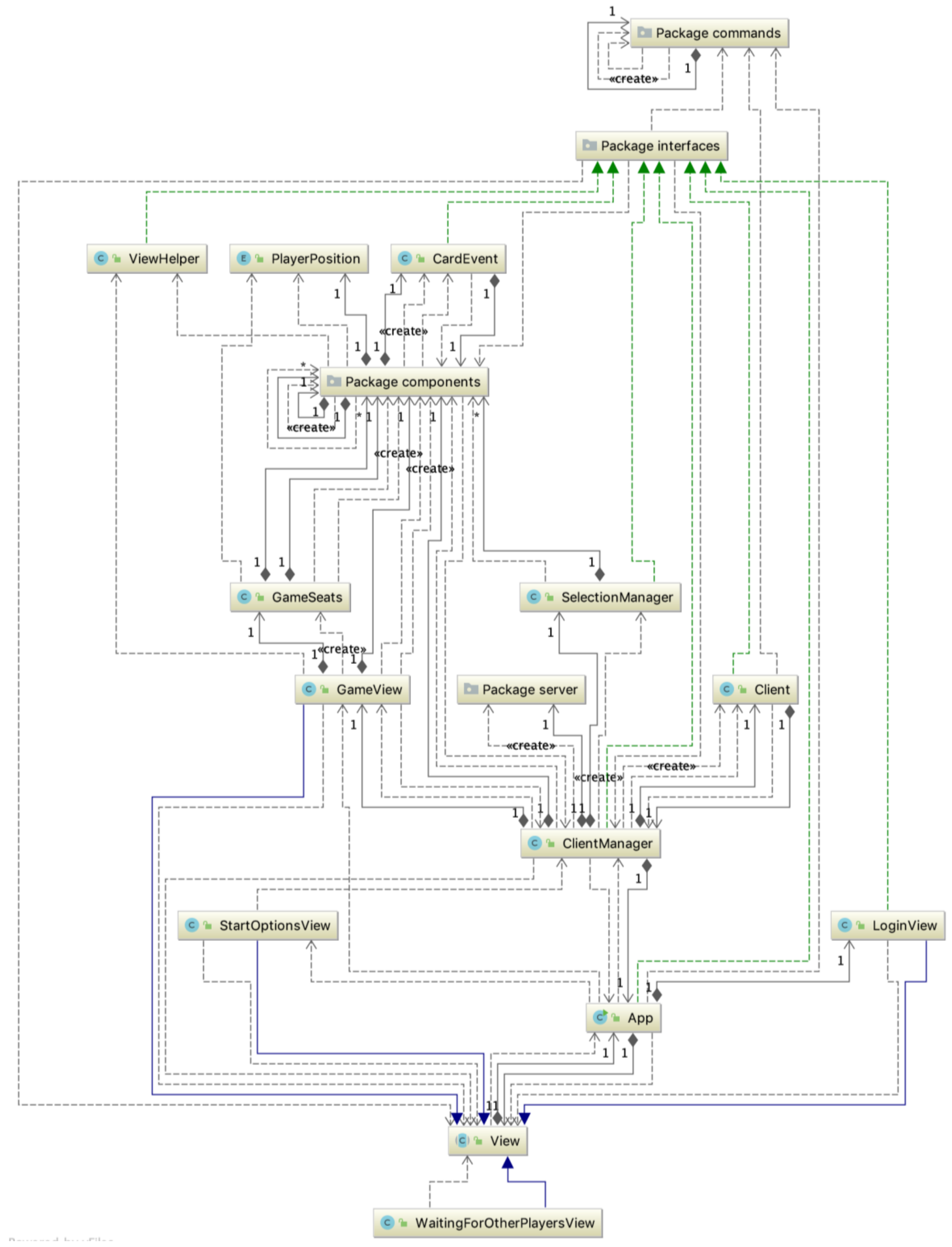


FIGURE 13: CLIENT-SIDE Dependencies

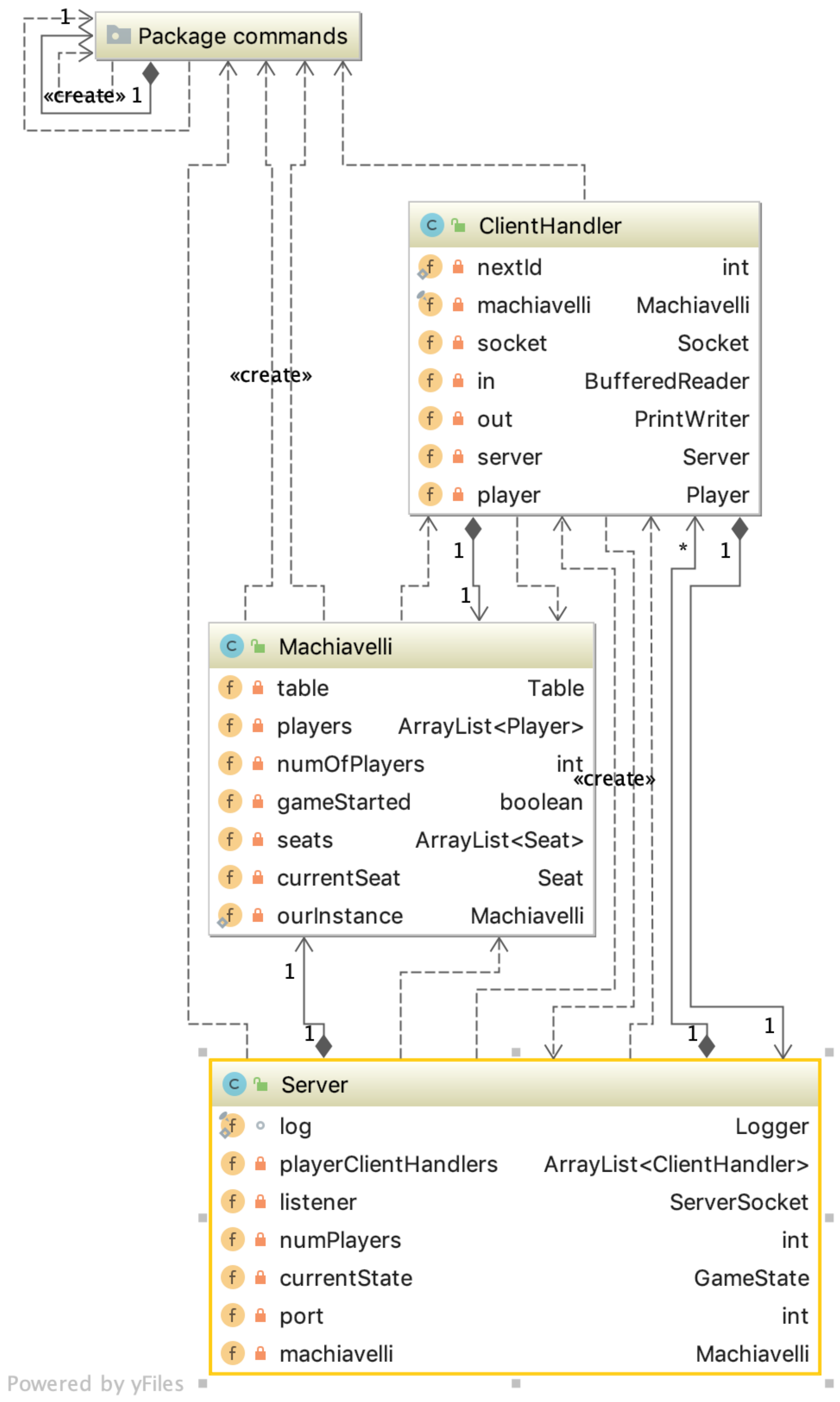


FIGURE 14: Client Handler

OOP Patterns

Each module and the entire UI was developed by keeping the MVC architecture in mind. The core logic of the gameplay and the role of each object is designed to be contained in the models. The views allow user interaction and all of it is managed by the controllers. The following other OOP design patterns were identified to be used in the development for the project.

Pattern	**Reason**
Memento	To keep track of the cards in each move and while reorganizing the ones from the table so the move can be reverted.
Flyweight	To cache images and other resources to save memory
Command	To ease the communication between the client and server
Factory	To create command objects by parameters received from the client or the server without specifying the exact class of the command
Singleton	To keep a single instance of an object and reuse it when needed.

Table 5: OOP Design Patterns followed in the development

Implementation

This section gives a brief account of the implementation of the project and how the system was coded. It talks about how each layer of the system was developed in succession to make the code modular and easy to debug while being true to the design specifications and our original intention.

The system was implemented using java. Java FX was used for the interface as it provided a strong working ground to quickly get rolling with an MVC architecture.

Tools of the trade

- Java
- JavaFX for UI
- CSS for designing UI and layouts
- IntelliJ Idea Form designer

The major focus of the development was to create the base working system as soon as possible so we could focus on studying and applying the event driven approaches and analyze how they worked in a real-time setting.

Client Server

The implementation process started with the development of the client and server. A basic communication platform was developed that allowed the client to connect to the server and request for services using commands.

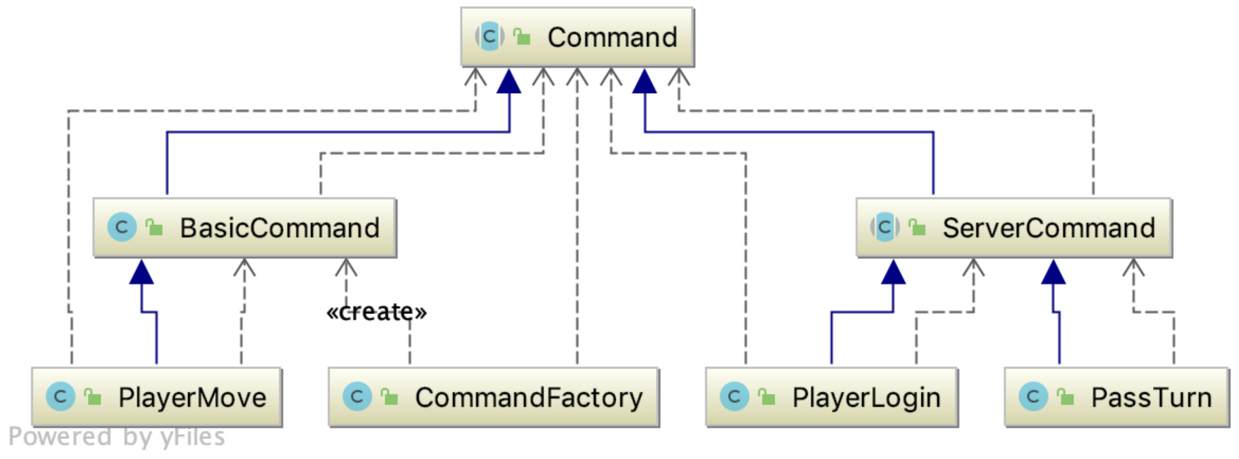


FIGURE 15: Possible commands for the server

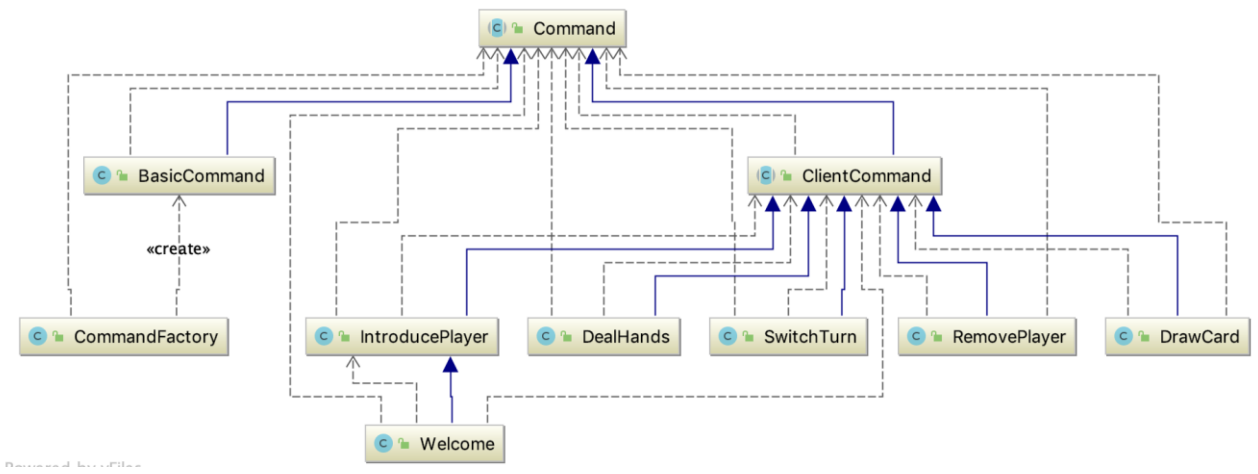


FIGURE 16: Possible Commands for the Client

Models

All the game entities were represented as models and each model deals with all the core logic and functionality of the entity. The class Machiavelli contains all the core of the gameplay mechanisms and its responsible for validating the rules of the gameplay and the win lose conditions.

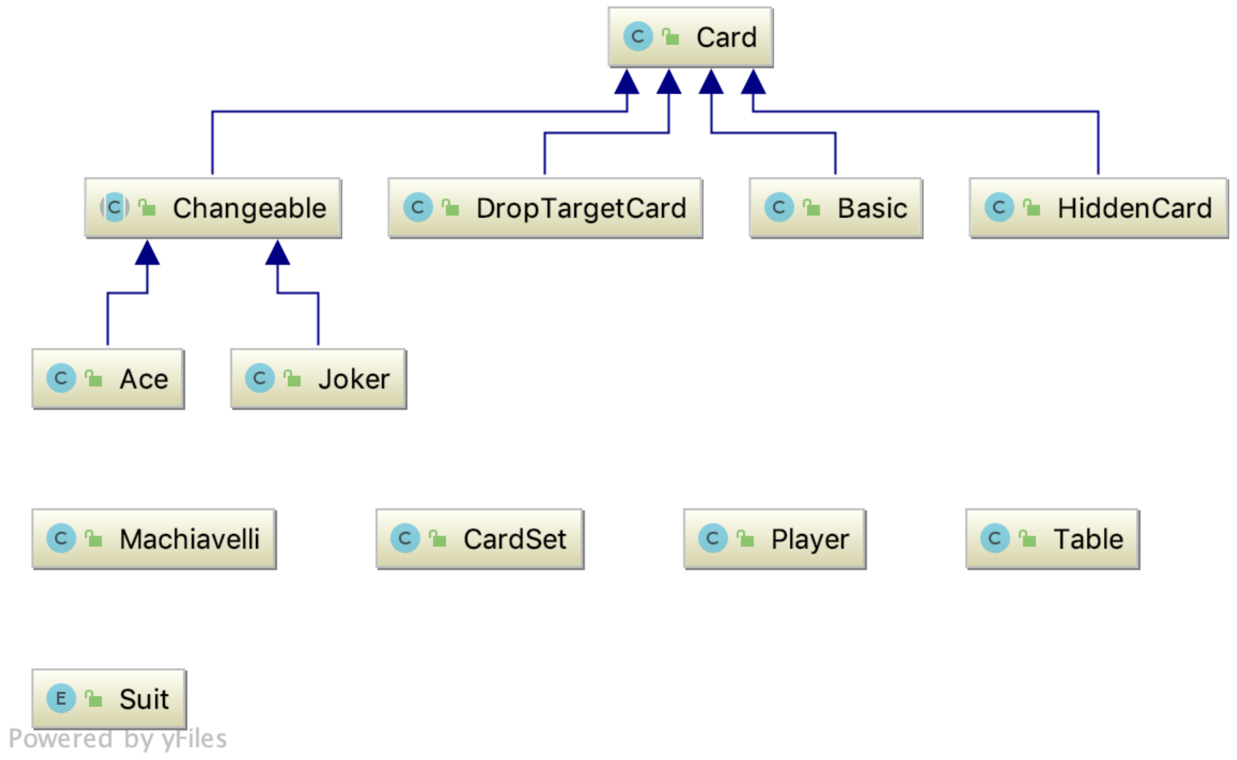


FIGURE 17: Models

Views

After getting the communication system ready, application UI was developed. The "Views" for each component of the UI are developed using CSS and FXML. Since the client program is used to both host a game and join other servers, all the UI is created in the client module.

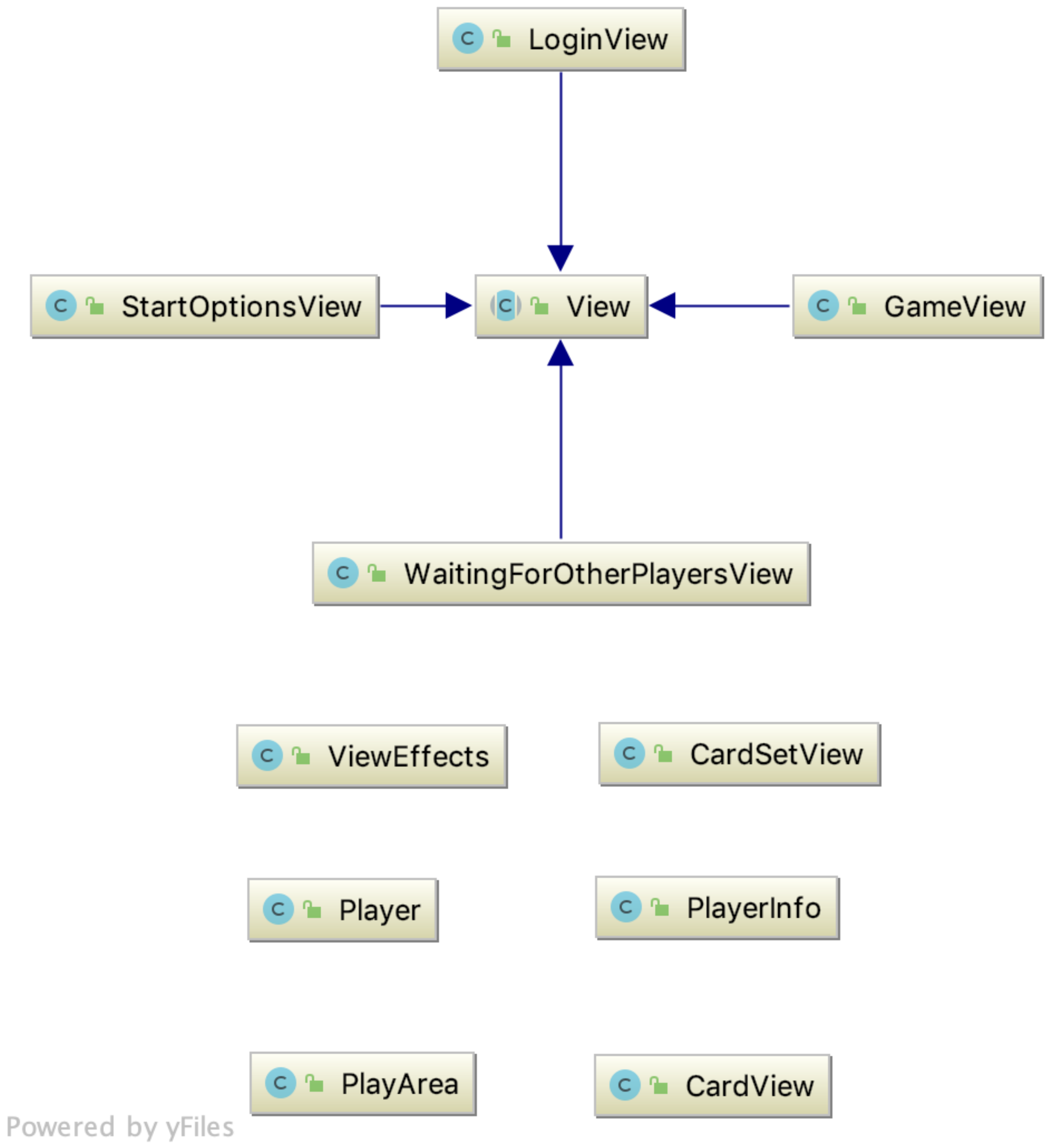


FIGURE 18: Views

Reactor Pattern

This implementation uses a mixture of **Concurrent and Re-entrant Reactor** design. In this design, the server class acts as the reactor, but it creates a new thread for each event handler to improve performance. Further since each player will be constantly connected to the server, so a re-entrant design allows each client handler to run its own separate event loop.

The server class opens a port for each connected client and creates a Client Handler. Whenever a client connects, the server checks for a slot on the table's queue and if its available, adds the client to the queue. Else it responds to the client with a TABLE_IS_FULL command.

```
while (true) {
   System.out.println("Accepting next Client..");
   ClientHandler clientHandler = new ClientHandler(listener.accept(), this);

   if (machiavelli.isTableFull())
   {
      clientHandler.sendCommand(Command.CommandNames.TABLE_IS_FULL);
   }
   else
      {
      machiavelli.addPlayer(clientHandler);
      playerClientHandlers.add(clientHandler);
      startClientThread(clientHandler);
      machiavelli.startGame();
   }
}
```

The main event loop of the server.

Here the reactor class is listening on the handle (TCP Port object in this case) for a new client. If there is space on the table to handle the new client, a ClientHandler is created and started in a new thread.

Implementation of Components

Reactor

The server class acts as the reactor and runs the main event loop.

Handle

The TCP port object serves as the handle and generates indication events to be processed by the handler.

Synchronous Event Demultiplexer

We use multiple threads to emulate the demultiplexing mechanism and to improve performance. Each handler thread runs in a separate thread but reacts to the event information synchronously.

Concrete Event Handler

The client handler class acts as the event handler for a client. Since the client is kept connected during the game, it runs a separate event loop for serving the requests of that particular client.

```
public void run() {
   try {
      while (true) {
         String cmdString = in.readLine();          //Reading from the handle
         Command cmd = buildCommand(cmdString);       //Identifying the event
         processCommand(cmd);                       //Calling the hook
      }
   } catch (Exception e) {
      . . .
   }
}
```

The ClientHandler’s event loop in ClientHandler.run() method

Proactor Pattern

This implementation uses a Proactor design with **Asynchronous Completion Handlers.** So, the Completion handler also acts as the initiator and invokes the hook for the asynchronous operation.

The main event loop of the server is the same as before with it creating a Client Handler in separate thread for each new client. It uses the port its listening on as a handle for accepting connections. Once it has connected a client and added it to the list of connected clients, it resumes its task for listening for connections while the connected client is forwarded to the Client Handler's event loop.

```
public void run() {
   try {
      while (true) {
         String cmdString = in.readLine();          //Reading from the handle
         Command cmd = buildCommand(cmdString);      //Identifying the event
         processCommand(cmd);                        //Calling the hook
      }
   } catch (Exception e) {
      . . .
   }
}
```

The ClientHandler's event loop in ClientHandler.run() method

Implementation of Components

Components are implemented similarly except for a few changes in the roles they perform in the system.

Proactor

The server acts as a Proactor and it provides a login event loop for new clients.

Handle

The TCP Ports act as handles for generating information events.

Concrete completion Handler

The Client class acts as the concrete completion handler. It also ac

Invoker and Asynchronous Operation Processor

The ClientHandler.processCommand function acts as the Asynchronous Operation Processor. It creates a new thread and starts processing the request immediately. It then uses the Commands specified to generate the respective completion event and process it. Thus, eliminating the need for a separate invoker.

```
private void processCommand(Command cmd) {
   new Thread(new Runnable() {  //Create a new thread
      @Override
      public void run() {
         ...
         cmd.execute();      //Execute the command
         ...
      }
   }).start();              //Start the processing immediately and return control
}
```

The ClientHandler's processCommand method

Completion Event Queue

Since we are using a design with asynchronous completion handlers, the completion events are handled asynchronously by the thread on which they were dispatched upon.

Discussion

Overview

This section gives a brief overview of the differences found in implementing both patterns and how they effected the overall system. It also gives a few insights as to where each type of system might be more useful.

Event Handler vs Event Completion Handler

Both of their task is to run a loop, sense event information, and demultiplex and dispatch appropriate hook methods for handling those events.

But the difference lies in the nature of the events. Which greatly effects their behavior.
The event handler in reactor design simply listens for event information for operations that haven't been performed yet. Then it synchronously dispatches the hooks for processing an event. After it has finished processing, the client is notified of the results and the reactor is free to react to other events.
In the game the server listens on the port (which acts as a handle) for events. When a client is connected, it creates a ClientHandler in a new thread and passes the handle of the client to the ClientHandler.
After this, the difference in implementation comes into play. In reactor, the command is received, the Client handler forwards the control to its processCommand method and blocks all the other events until the processCommand method returns.
This reactive approach makes the processing synchronous so there is a lower overhead for concurrency, it improves modularity while keeping the complexity of the whole process relatively low.
In the Proactor implementation, the ClientHandler listens on the handle and starts processing a request immediately when received. It creates a new thread and passes the request to the new thread and resumes listening on the handle. The new thread acts as the invoker as well as the event completion handler which greatly improves performance and concurrent processing capability.

Characteristics of Scenarios where Reactor and Proactor designs might be useful

This type of systems is implemented where a lot of concurrent event driven requests need to be served. Reactive and Proactive approach provides modularity, scalability, and concurrency while keeping the system modular and highly responsive.

The reactive approach is suitable for

- A synchronous system which processes where the order of operations and scheduling greatly matters.
- Systems which require the application code to remain separate from the request handling functionality to keep the application modular
- Concurrent systems which need to be cross platform while still maintaining a low overhead for concurrency.

The proactive approach is particularly suitable for

- Systems that are required to serve long running tasks.
- Concurrently processing systems which are required to be pre-emptive.
- Systems need to be highly scalable
- Systems where the concurrency policy needs to be separate from threading policy.

Future Implementations

The game currently doesn’t allow more than 4 players to be connected. Future experimentations could be done by trying to make the game be able to handle spectators. ClientHandler could be extended to allow many users to connect as viewers to the game while allowing a set number of players to play. This might display the capabilities of the proactive pattern as a more scalable version of event handling or the capability of the reactive system to have a low overhead in case of restrictive environments.

References

Bates, B., Sierra, K., Freeman, E., & Robson, E. (2009). *Head First Design Patterns.* O'Reilly Media.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Schmidt, D. (2018, January 4). *The Reactor and Singleton Pattern*. Retrieved from Youtube: https://www.youtube.com/watch?v=pmtrUcPs4GQ

Schmidt, D. C. (n.d.). *Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events.* Retrieved from vanderbilt.edu: https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2.* John Wiley & Sons.

Vitellaro, S. (2018). *Reactor and Proactor Examples of event handling patterns.* Retrieved from http://didawiki.cli.di.unipi.it: http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/tdp/tpd_reactor_proactor.pdf

Game resources from:
https://code.google.com/archive/p/vector-playing-cards/downloads