Project Multitris

Group 23
Marcus Dicander
Måns Olson
Tomas Alaeus
Daniel Boström
Oscar Olsson

Table of Contents

1. Introduction

1.1. Purpose

This document details the design and architecture considerations of the Multitris project. The entire project is specified in detail, and the specification allows software engineers to implement the project accordingly.

1.2. Scope

The Design Document details the design and architecture of the Multitris project. Not included are the project overview and the project requirements; these are both descibed in the Requirements Document (RD). For further information about the RD, see the "Related Documents" section below.

1.3. Expected readership

This document is intended for the software engineers tasked with implementing the Multitris project.

1.4. Multitris version history

Version	**Summary**	**Date**	**Authors**
Multitris 1.0	First version of the Design Document (DD)	2008-03-07	Oscar Olsson, Marcus Dicander, Tomas Alaeus, Daniel Boström, Måns Olson

1.5. Summary

The Multitris project is described as a client-server architecture system designed with Java, with separate subsystems for the Client, Server, and Central Server. The graphical user interface is detailed and represented by prototype images. The system behaviour is outlined in a set of diagrams, including state charts and interaction diagrams. Each class and its methods are described in the Javadoc documentation format. For each requirement in the RD, a functional test case is specified to help evalute the system after implementation.

1.6. Related documents

The Requirements Document (RD) specifies the project overview and the project requirements. It is the basis for the Design Document, and is referenced throughout this document.

1.7. Glossary

Table 2.

Term	**Explanation**
Java	A platform-independent programming language.
JRE 1.5	Java Runtime Environment 1.5, which provides a virtual machine for running Java (2).
LWJGL	Lightweight Java Game Library (4), a library for games creation in Java.
Java Webstart	A standardized way to launch Java applications from the Internet.
OpenGL	Open Graphics Library (3), a standard for hardware accelerated graphics.
Central server	The central server is a server that keeps a list of game servers waiting for players to join. There is only one central server.
Game server	The game server is the program that handles the game lobby and game sessions. There is one game server for each game being played.
Game	A game is the time period from the time that the server is launched until it is shut down. This includes the lobby in which players can talk and wait for other players, and any game sessions.
Game session	A game session is the time period when the user actually plays the game.
Game lobby	The game lobby is a process that allows players to communicate and wait for others during games, between game sessions. In a sense, it is a virtual meeting room.
Game board	A playing field with a grid of a given size that can hold bricks.
Brick	A fundamental game element which can connect to other bricks.
Piece	Four connected bricks that can be controlled by a player. Pieces can also collide with each other and bricks.
Powerup	A special brick that can be activated by a player to alter the game board or gameplay.

2. System Overview

2.1. General Description

The system is a client-server architecture including three subsystems: the Game Client, the Game Server, and the Central Server. An active choice was made to implement the project in the object-oriented language Java, and in this document is documentation of the standard Javadoc format to support that choice.

The Game Client uses internal States to represent the different modes of operation inherent in the system. One state represents the menu mode, one state represents the lobby mode, in which players communicate with each other while waiting for a game to start, and one state represents the games session mode.

Each Game Client instance communicates with a Game Server for singleplayer and multiplayer sessions. Available Game Server instances can be accessed via the Central Server, whose primary task is to keep a list of Game Servers.

2.2. Overall Architecture Description

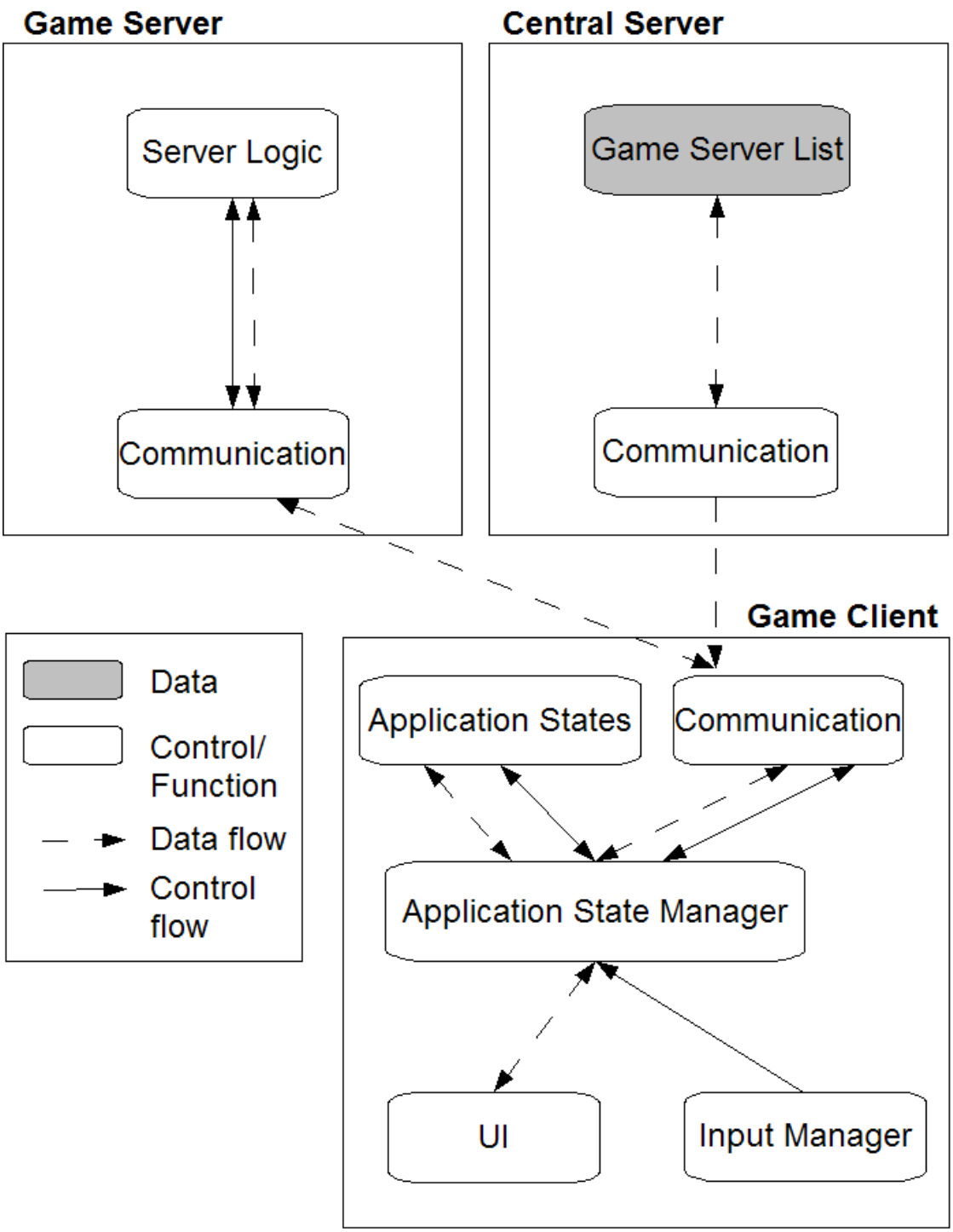


Fig. 2.1

Multitris is a multiplayer game system organized into a client-server architecture. Figure 2.1 is a Box-Line diagram that shows the overall architecture. The system consists of three major parts; the Central Server, the Game Server, and the Game Client. The Central Server keeps a list of Game Servers, which is publicly available to any Game Clients. Clients can the Central Server to find Game Servers. A Client can then request a connection to one of the Game Servers in the list.

The Game Server's task is to keep track of a Game Session. A number of Clients can connect to a Game Server to participate in a Game Session (i.e. playing together). To change the state of the game, for example by making a move, the Client sends a request to the Game Server.

Each of the three major parts consists of a number of subsystems. The Central Server has a Game Server List and a Communication interface. Game Servers that wish to be added or removed to the list can communicate this via the Communication interface. The Central Server can also notify Clients of available Game Servers via the Communication interface.

The Game Server has a Communication interface that is used to communicate with the Game Client and Central Server. Incoming requests are passed on to the Server Logic subsystem. The Server Logic gathers any action requests from the Game Clients, and then notifies each Client of all action requests made via the Communication interface. Further, the Server Logic subsystem can request the Game Server to be added or removed from the Central Server's Game List via the Communication interface.

The Game Client consists of a Communication interface, a Application State Manager, a User Interface subsystem, and an Input Manager. Player input is handled by the Input Manager, which is passed on to the Application State Manager. The Application State Manager then passes on the data to another system, depending on the input. The Client regularly recieves a list of requested actions from the Game Server, and these are then passed on to the Application State Manager which makes changes to the Application States accordingly.

2.3. Detailed Architecture

Local player action

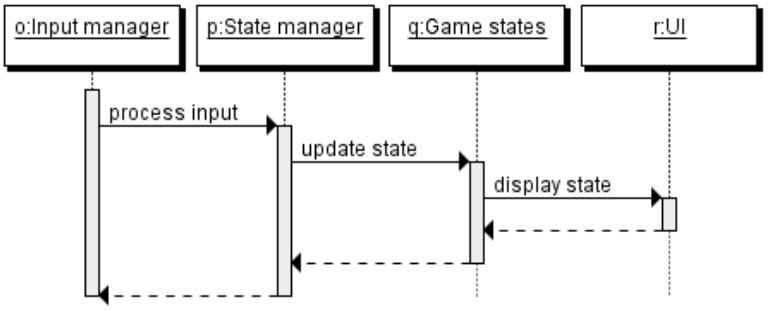


Fig. 2.2

Remote player action

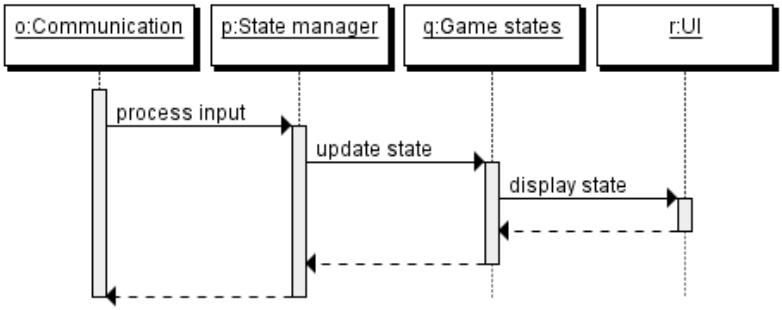


Fig. 2.3

Game server

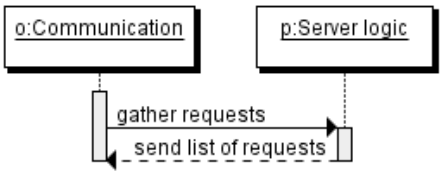
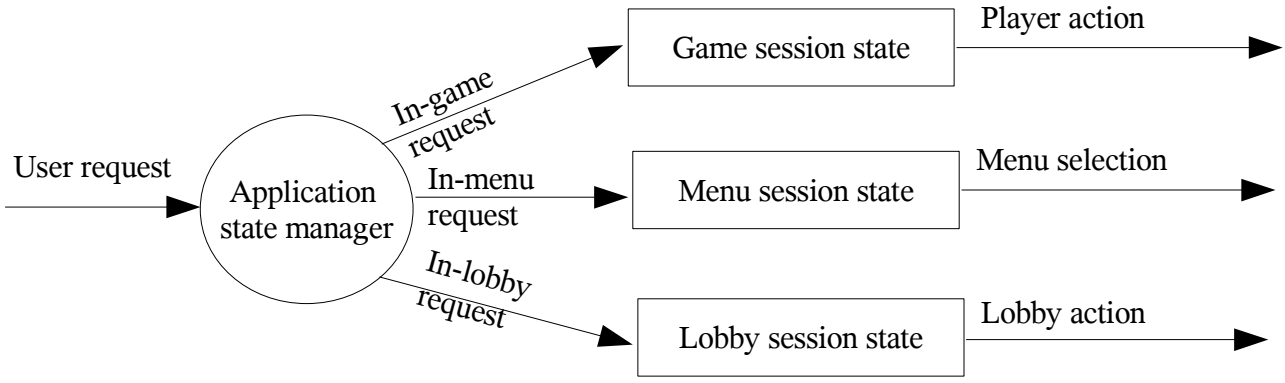


Fig. 2.4

Fig. 2.5



Figures 2.2 through 2.4 above are sequence diagrams showing control and data flow in the system. Figure 2.5 is a Box-Line diagram of how decisions are made in the Application State Manager. The process is explained in detail below.

The Application State Manager's responsibility is to provide a single interface to the different Application States. Any change requested by either the User or the Game Server is routed through the Application State Manager. Depending on the type of change, the Application State Manager will pass on the request to a different Application State, which will in turn handle the request.

At all times, User input is passed on to the currently active Application State. Important Application States are the Menu State, the Lobby State, and the Game Session State. The Menu State uses User input to provide menu functionality. The Lobby State is active when the User is waiting for a Game Session to start. When it does, the Game Session State will be the active state.

Possible requests are movement requests from the User (for example navigating a menu, or moving a piece while in a Game Session), and requests to open or close a Game Server for additional users. In some cases, the requests will need to be passed on via the Communication interface to the server.

3. Design Considerations

3.1. Assumtions and Dependencies

3.1.1. Software and hardware dependencies

We intend to use the Java platform, with the Lightweight Java Game Library wrapper. This is a tool for developing games in Java and provides access to the OpenGL library.

The game can be launched via any web browser with a Java Webstart plugin. As for operating systems, cross-platform compability is always an issue. However, OpenGL and LWJGL are confirmed to work on three major operating systems, namely Windows XP, Mac OS X and Linux. The system will thereforerun on all these platforms. At least one user, the host, will need to have a specific port open for a multiplayer session to work.

3.1.2. End-user characteristics

We target *casual gamers*, i.e. people who play games on a non-regular basis. Our intended user is a male between 15-35 with access to a computer network. Ideally, the user has enough technical knowledge to play games via his browser, and will have a Java Runtime Environment JRE installed on his machine. The user runs either Windows XP, Mac OS X, Linux and has a computer that supports hardware-accelerated graphics. Additionally, the user should be able to read and understand English. Familiarity with the game Tetris (1) is also recommended. The user should ideally use the Internet (reads blogs, search portals, or chats) to find web games.

3.1.3. Minimal system environment specifications

Supported operating systems: Windows XP, Mac OS X, Linux.
1.0+ GHz Intel Pentium processor or equivalent
512 MB of RAM memory
NVIDIA GeForce 4MX or equivalent
50 MB free hard drive space
JRE installed, version 1.5 or higher

3.1.4. Optimal system environment specifications

Supported operating systems: Windows XP, Mac OS X, Linux.
2.0+ GHz Intel Pentium processor or equivalent
1 GB of RAM memory
NVIDIA GeForce FX 7600 or equivalent
50 MB free hard drive space
JRE installed, version 1.5 or higher

3.1.5. System evolution

There are certain fundamental assumptions on which the system is based. These include:
- The operating system on which the system is run supports Java.
- For the clients, support for OpenGL via LWJGL is also assumed.
- A Java Runtime Environment of version 1.5 or higher is installed.
- The TCP/IP protocol is used for network communication.
- For users wishing to host games, the proper network port is open.

As the system environment evolves, these are some of the changes to the system we may have to make:
- New operating systems may not support OpenGL, in which case a different rendering engine may have to be used.
- New hardware may not support OpenGL, in which case a different rendering engine may have to be used.

As the user's needs change, these are some changes to the system we may have to make:
- Users may want to communicate globally, in which case a chat subsystem may have to be added to the central server. Initially, we will only allow players to communicate within games.
- Users may not want to launch a separate game, in which case conversion to a Java Applet may be required. Java Applets are played from within the web browser.

3.2. General Constraints

The hardware and software environment constraints are outlined in the above section. For the purpose of standardization, the game is built using OpenGL graphics technology. This will have a major impact on system design as the system will have to comply to a set of given standards.

The Central Server is a major component of the system. The distributed system packages will contain information on how to access the Central Server, and all clients will thus be able to communicate with each other. Therefore, it is a vital resource that must be stable enough for continuous operation. It can be instanced several times, but the primary global instance must be available at all times.

Since the system is a Java application that can be started via Webstart, the system will need to have a digital certificate for access to network functions. The reason is that unsigned Webstart applications are run in a so-called “sandbox” mode with very limited access to the client platform.

4. Graphical User Interface

4.1. Menu Overview

When the application starts the user is presented with the main menu (figure 4.1). The user may choose between the following options:

Host game
Lets the user start a network game. The user is then promted for a name and the maximum number of players allowed in the session (figure 4.2). The game then enters a ”waiting for players” lobby state (figure 4.3). At any time after this, the user may start the game session.

Join game
Prompts the user for a name (figure 4.4). The user is then presented with a list of network games that he can join (figure 4.5). The user may then select a game to join, upon which he enters the lobby and waits for the game to start (figure 4.5).

Singleplayer
Let the user start a singleplayer game session (figure 4.6).

Help
Presents the user with help documentation for the game (figure 4.7).

Exit
Exits the game.

Once in a game session, the user can access the menu by pressing ”Escape” on his keyboard.

4.2. Functionality description

During the game, the user can control the falling pieces by moving them left or right with the corresponding arrow keys, accelerate downward movement with the arrow down key, or rotating them by pressing the up arrow. In a multiplayer session the users control one piece each and they all share the

same game board. The numbers 1 through 3 can be used to activate the corresponding powerups (figure 4.6). The user can also see the names of other users and the team's score, in the top left corner.

Control of a piece ends when the piece reaches the bottom of the board or on top of another fixated brick. Bricks cannot intersect each other. If two bricks collide, they will either be removed with a penalty to the score or be moved away from each other, depending on how long the players try to force them together.

If a row is completely filled with bricks, that row will be removed resulting in all the above bricks falling down until they reach the bottom or a fixated brick. A player that completes a row with one or more powerup bricks will receive those powerups unless his powerup stack is full (figure 4.6). If the non completed rows stack up to the top of the screen, the game session ends.

4.3. Forms

Main menu

[Button]	Host game; go to *Host game* triggers: *hostGame()*
[Button]	Join game; go to *Join game* triggers: j*oinGame()*
[Button]	Singleplayer; go to *Game session* triggers: startSingleplayer*()*
[Button]	Help; go to *Help* triggers: showH*elp()*
[Button]	Exit triggers: exit*()*

References functional requirement 6.1.1 #6
Accessed at application startup
Displayed by *setStateActive(ApplicationStateManager.MENU, true)*

Host game

[Text field]	Nickname
[Text field]	# of players
[Checkbox]	Private?
[Button]	Create game; go to *Host game – waiting for players* triggers: hostG*ame()*
[Button]	Back; go to *Main menu* triggers: *leaveState()*

References functional requirement 6.1.2 #8 and #9
Accessed from *Main menu*
Displayed by *hostGame()*

Host game - waiting for players

[List]	Player list
[Chat area]	Chat
[Button]	Launch game; go to *Game session* triggers: startMultiplayer*()*

[Button] Back; go to *Main menu*
triggers: *leaveState()*
References functional requirement 6.1.2 #8, #9 and #11
Accessed from *Host game*
Displayed by *hostGame()*

Join game

[Text field] Nickname
[Button] Find games; go to *Join game – game list*
triggers: *listGames()*
[Button] Back; go to *Main menu*
triggers: *leaveState()*
References functional requirement 6.1.2 #10
Accessed from *Main menu*
Displayed by *joinGame()*

Join game – game list

[List] Game list
[Button] Join game; go to *Join game – waiting for players*
triggers: *joinGame()*
[Button] Back; go to *Join game*
triggers: *cancel()*
References functional requirement 6.1.2 #10
Accessed from *Join game*
Displayed by *listGames()*

Join game – waiting for players

[List] Game list
[Chat area] Chat
[Button] Back; go to *Join game - game list*
triggers: *cancel()*
References functional requirement 6.1.2 #10 and #11
Accessed from *Join game – game list*
Displayed by *joinGame(String address)*

Game session

[Text display] Score
[List] Player list
[List] Powerup list
References functional requirement 6.1.1 #1 through #4
Accessed from *Join game – waiting for players*, *Host game – waiting for players* and *Main menu*
Displayed by startMultiplayer*()* and *startSingelplayer()*

Help

[Text display] Help document

[Button] Back; go to *Main menu*
triggers: *cancel()*

References functional requirement 6.1.1 #5
Accessed from *Main menu*
Displayed by *showHelp()*

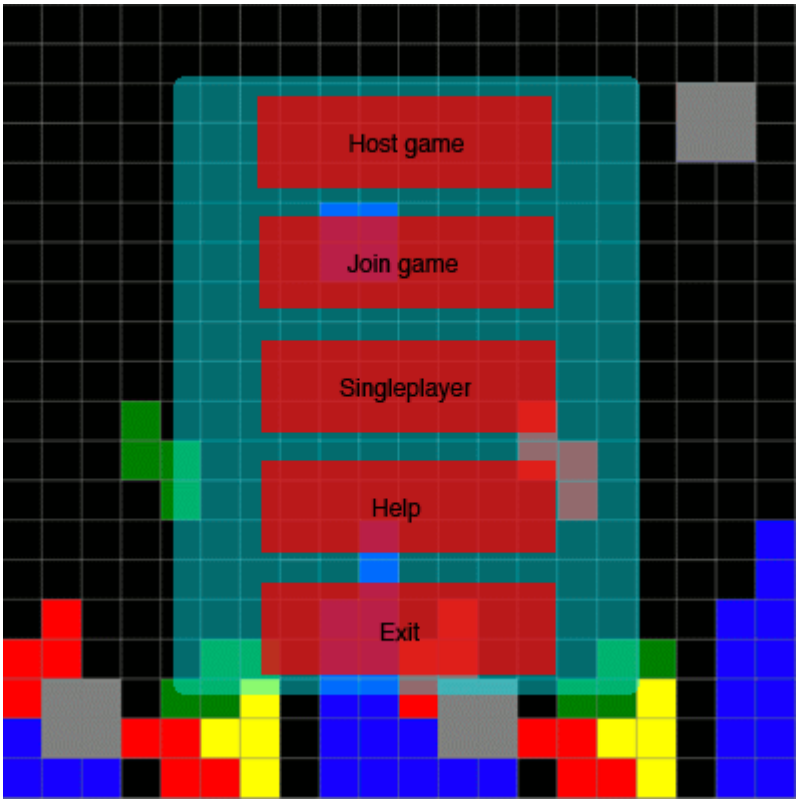


Fig. 6.1

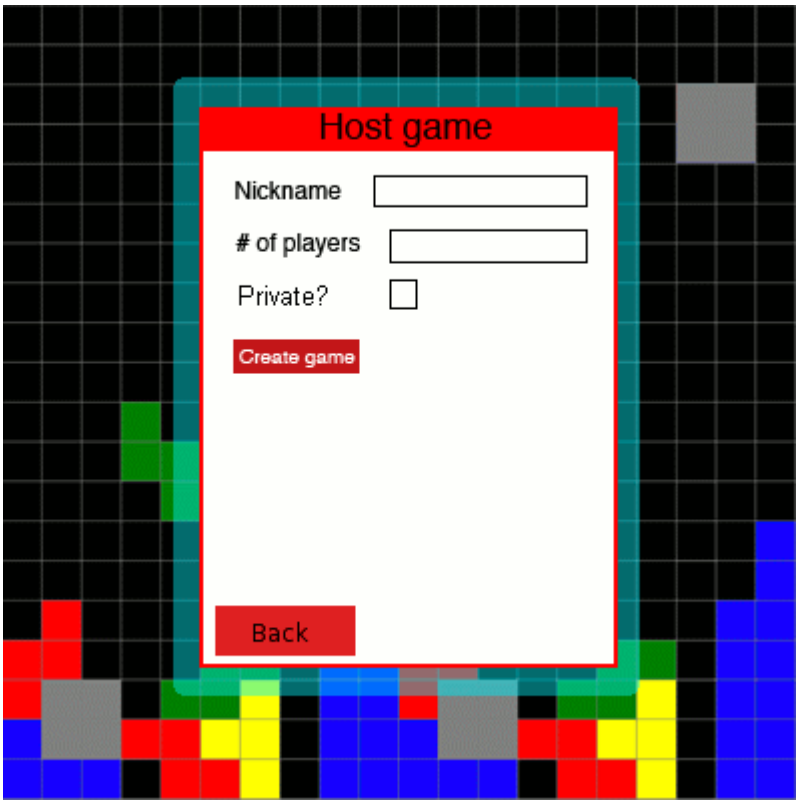

Host game
Nickname
# of players
Private?
Create game
Back

Fig. 6.2

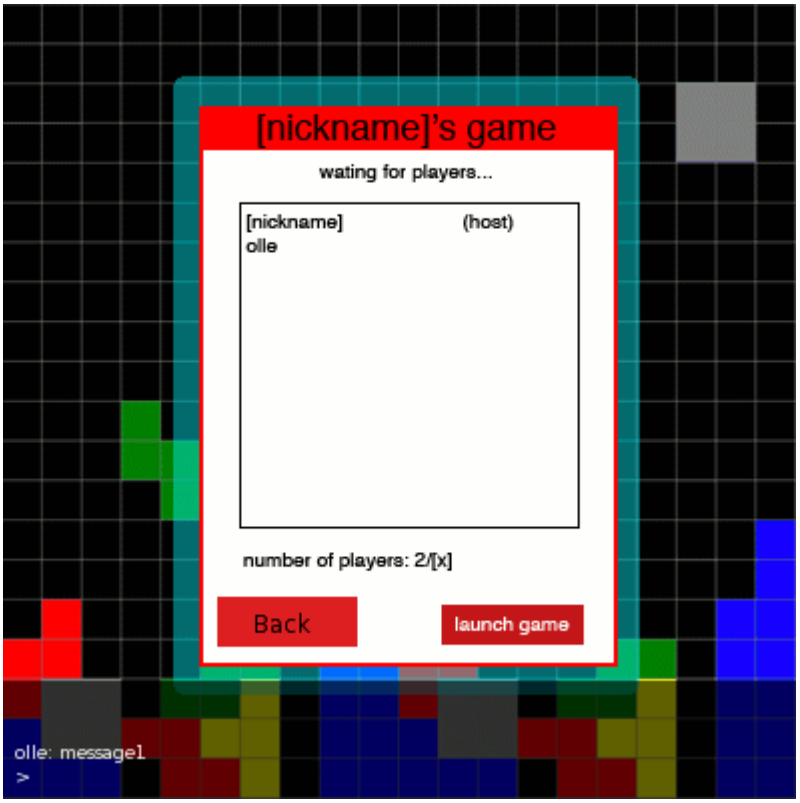

[nickname]'s game
wating for players...
[nickname] (host)
olle
number of players: 2/[x]
Back
launch game
olle: message1
>

Fig. 6.3

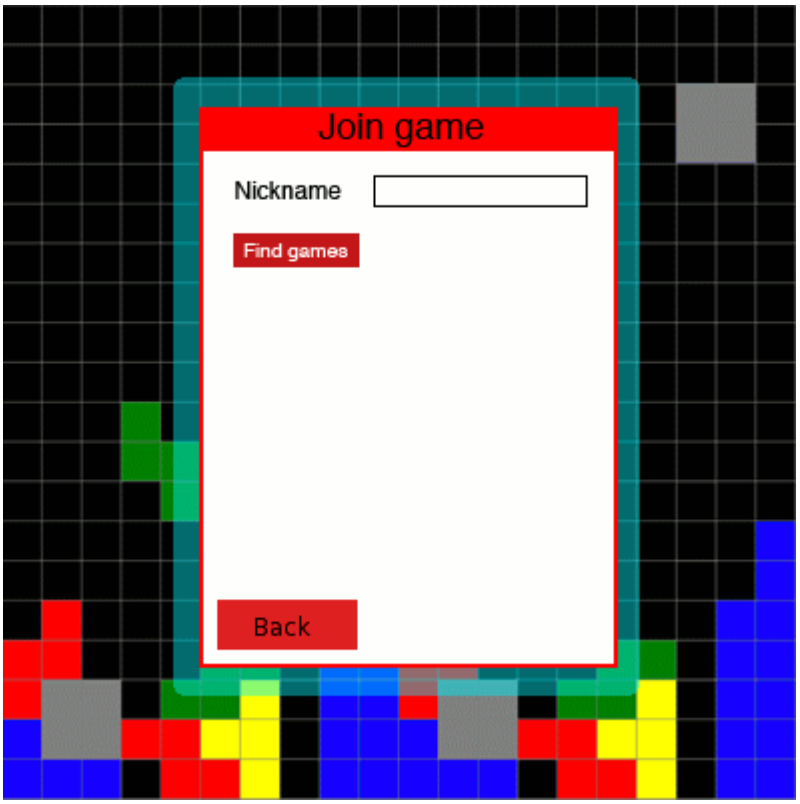

Join game
Nickname
Find games
Back

Fig. 6.4


Join game
nisse 2/5 players
per 1/5 players
nisse 5/5 players
gustaf 2/3 players
Waiting for host to start game...
Private game
Back
Join game
nisse: message 1
per: message 2
>

Fig. 6.5

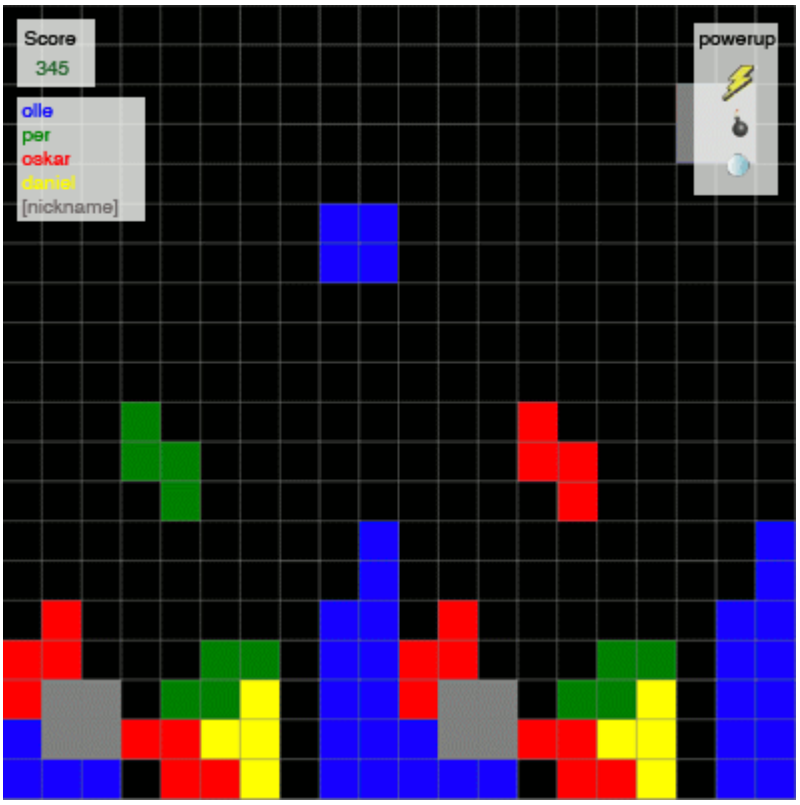

Score
345
olle
per
oskar
daniel
[nickname]
powerup

Fig. 6.6


Help
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Morbi urna ipsum, adipiscing in, facilisis quis, volutpat dictum, dui. Etiam venenatis turpis convallis est. Mauris arcu augue, commodo ut, aliquam nec, ornare a, est. Quisque laoreet mauris ac neque. Duis nec enim. Quisque at ante non elit egestas pulvinar. Sed id nisi. In scelerisque tempor nulla. Nunc urna. Aenean rutrum quam et diam. Pellentesque lacus enim, suscipit gravida, consequat
Back

Fig. 6.7

5. Design Details

5.1 Class Responsibility Collaborator (CRC) Cards

Class ApplicationStateManager	
Responsibilities - Redirects input depending on the current ApplicationState.	**Collaborators** InputManager ApplicationState

Abstract Class ApplicationState	
Responsibilities - Superclass for the different states.	**Collaborators** ApplicationStateManager

Class MenuState	
Responsibilities - Renders menu. - Handles menu actions.	**Collaborators** ApplicationStateManager

Class LobbyState	
Responsibilities - Renders the lobby. - Handles lobby actions.	**Collaborators** ApplicationStateManager ClientCommunication

Class GameSessionState	
Responsibilities - Renders the current game session. - Handles game session actions.	**Collaborators** ApplicationStateManager ClientCommunication

Class GameLogic	
Responsibilities - Handles game logic.	**Collaborators** Player Board

Class Board	
Responsibilities - Stores fixed Bricks.	**Collaborators** GameLogic Brick

Class Brick	
Responsibilities - Knows which player it belongs to. - Knows brick type (powerup?).	**Collaborators** Board

Class Piece	
Responsibilities - Has a list of bricks contained in the piece.	**Collaborators** Brick PieceGenerator Player

Class PieceGenerator	
Responsibilities - Generates pieces for players.	**Collaborators** Piece Player

Class Player	
Responsibilities - Keeps track of the player's current piece. - Keeps track of the player's powerups. - Keeps track of player info.	**Collaborators** PieceGenerator GameLogic Piece

Class InputManager	
Responsibilities - Interprets and forwards player actions.	**Collaborators** ApplicationStateManager ControllerMap

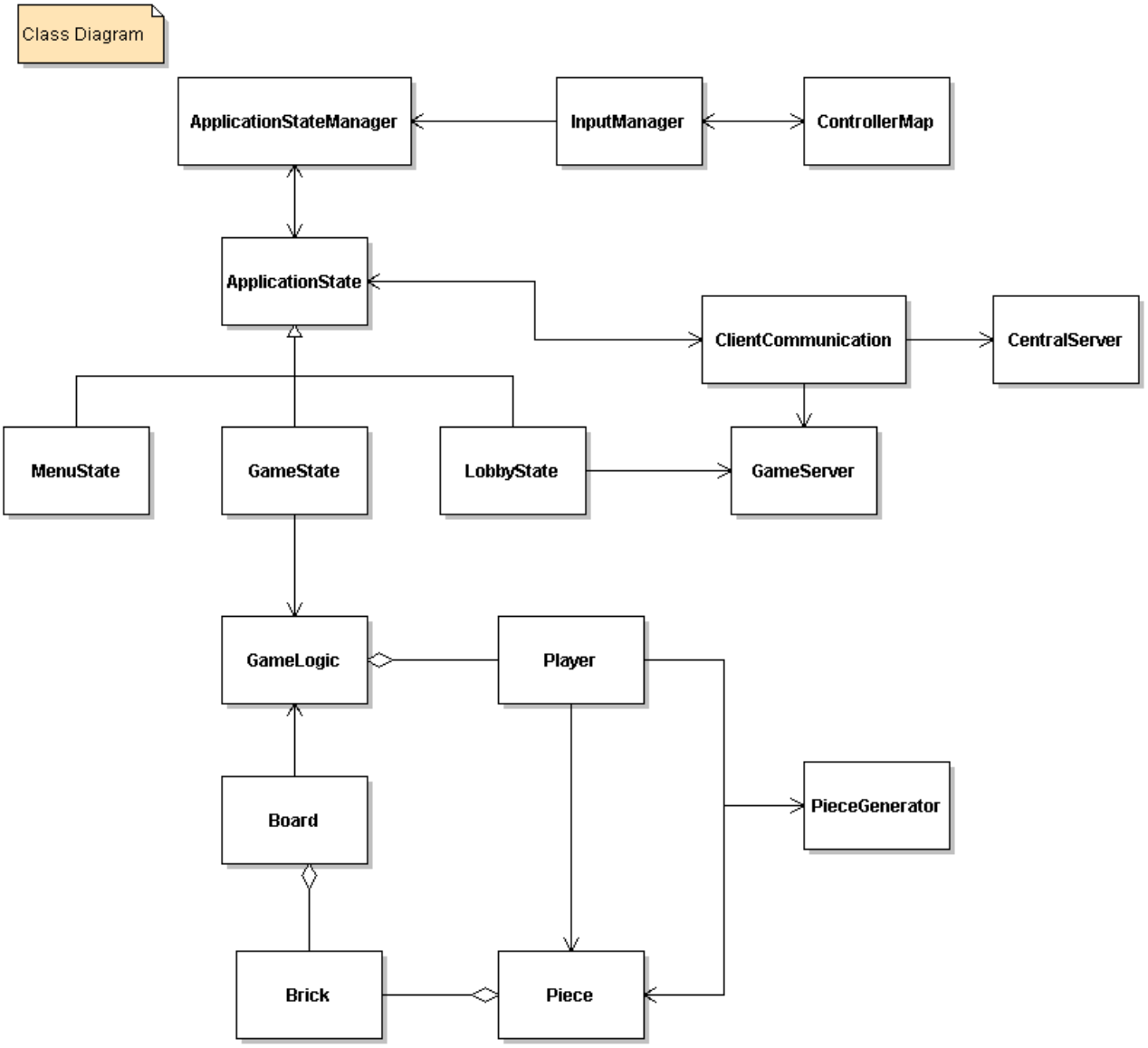
Class ClientCommunication	
Responsibilities - Handles network communication.	**Collaborators** GameServer CentralServer ApplicationState

Class ControllerMap	
Responsibilities - Translates player input into commands.	**Collaborators** InputManager

Class GameServer	
Responsibilities - Relays player actions to all users.	**Collaborators** ClientCommunication

Class CentralServer	
Responsibilities - Keeps a list of all open GameServers.	**Collaborators** ClientCommunication

5.2 Class Diagram



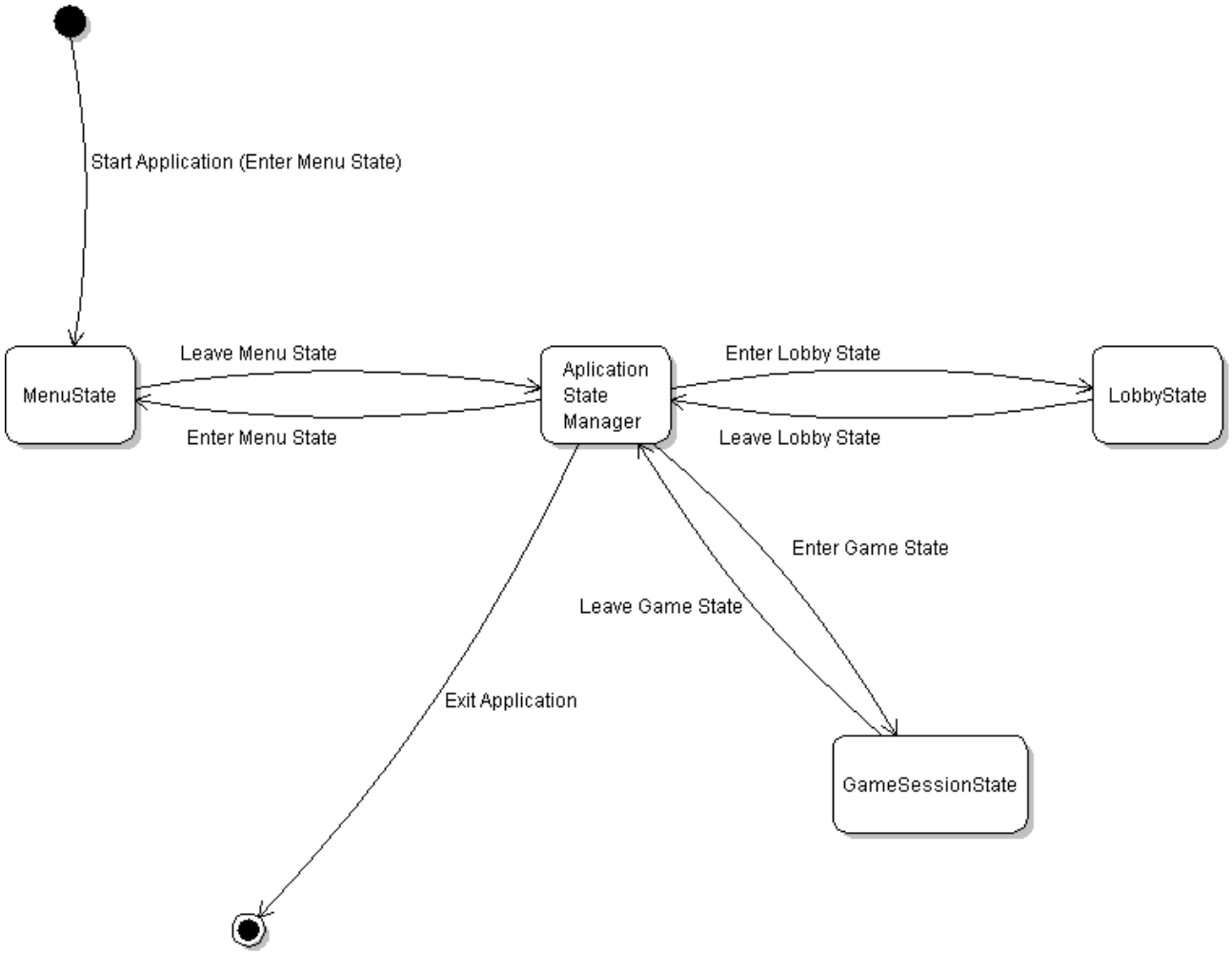
Legend:

Aggregation. For example, Board has several Bricks.

Association. The Board and GameLogic classes are associated.

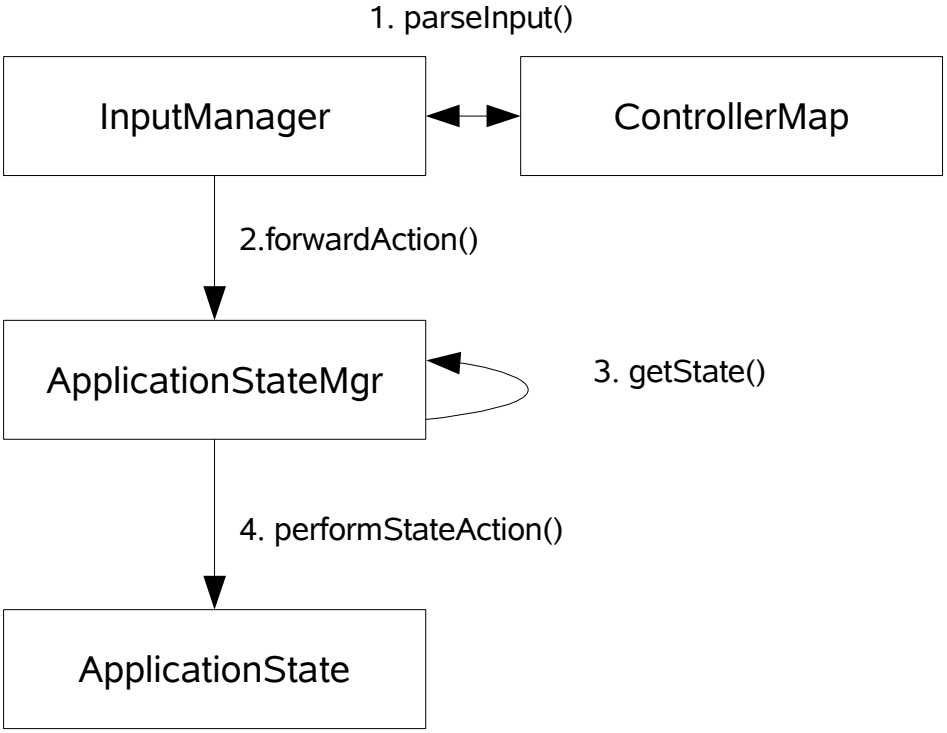
Inheritance. The different states inherit the ApplicationState.

5.3 State Charts

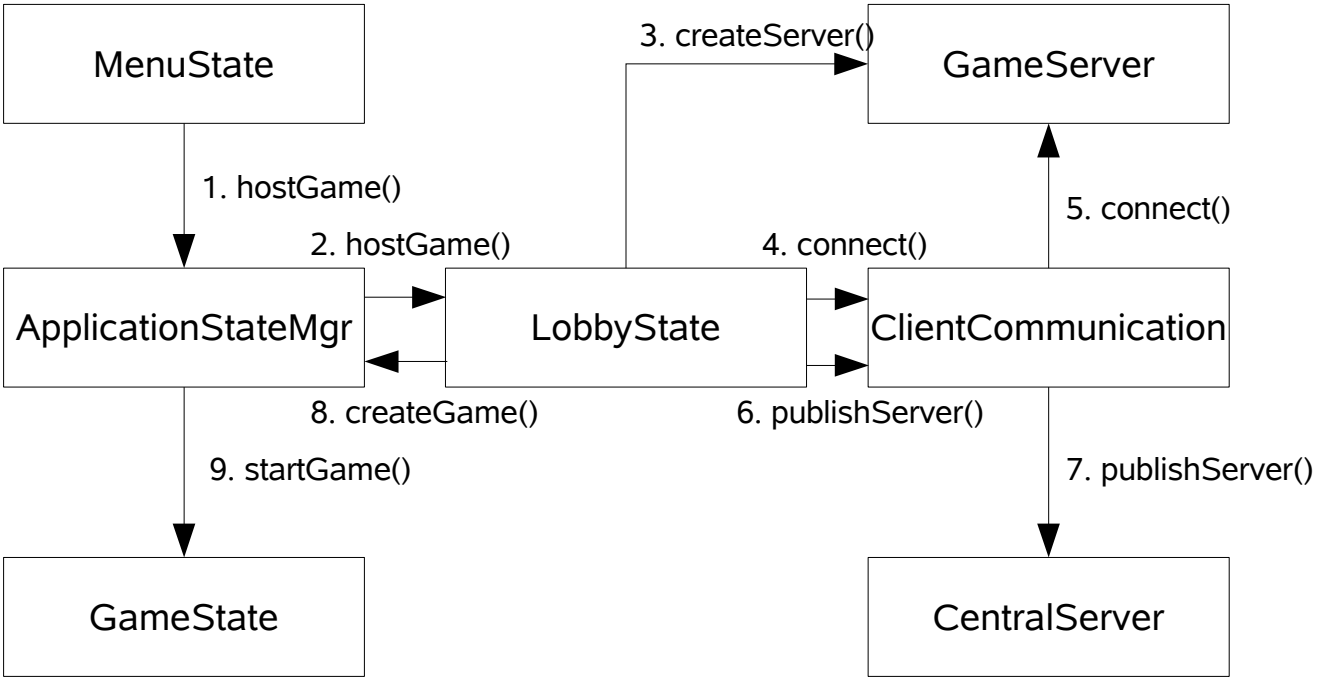


5.4 Interaction Diagrams

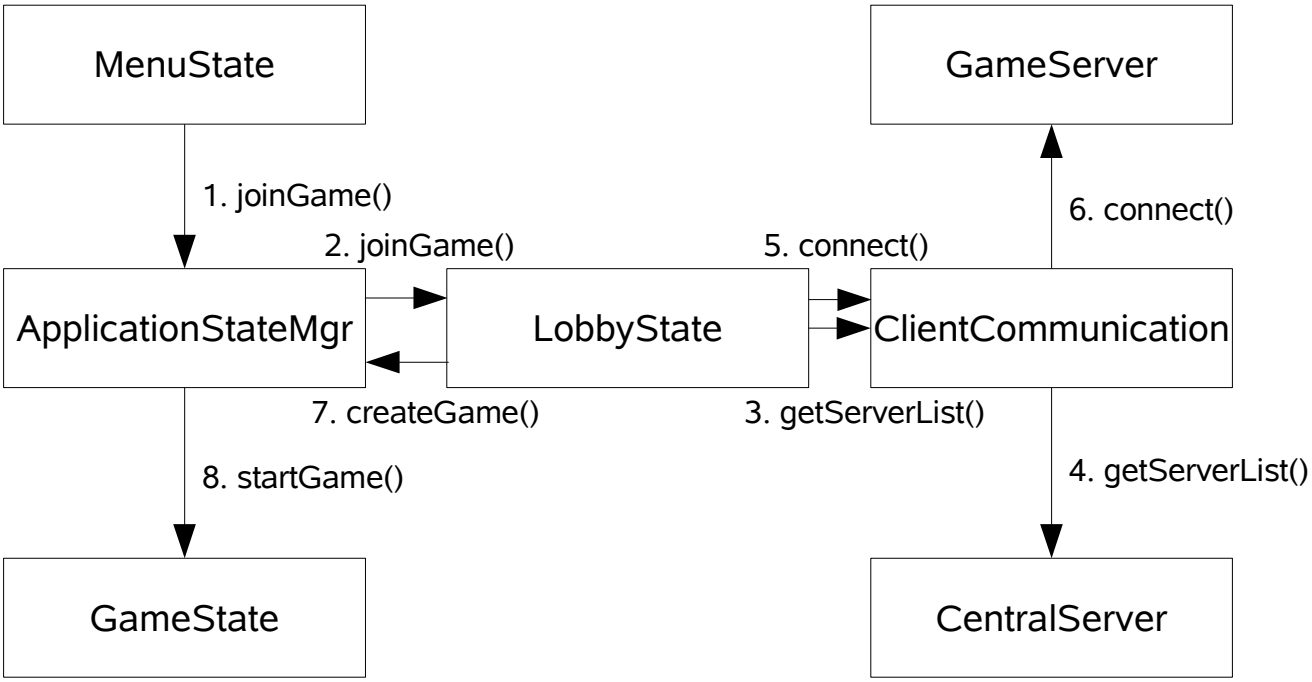
5.4.1 General Input Handling



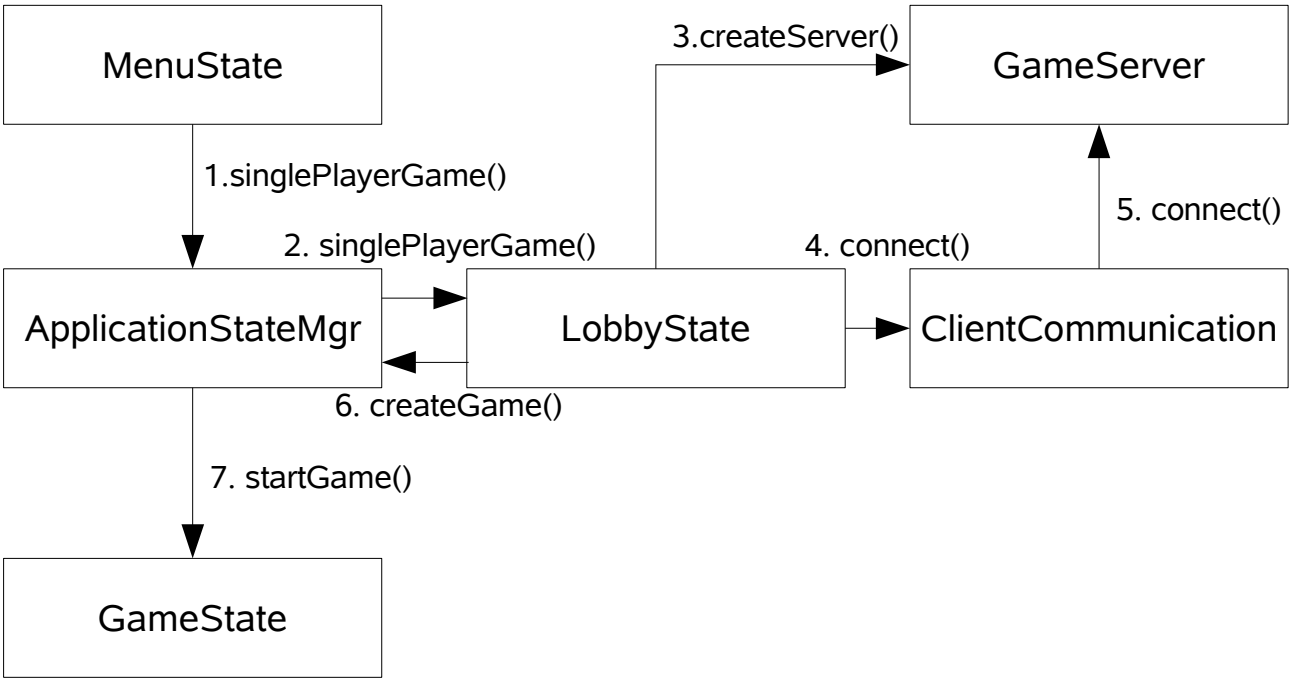
5.4.2 Host Game



5.4.3 Join Game



5.4.4 Singleplayer game



5.5. Detailed Design

5.5.1. Javadoc

Interface ApplicationState

All Known Implementing Classes:
GameSessionState, LobbyState, MenuState

```
public interface ApplicationState
```

This is the interface for the different application states.

Method Summary	
void	**doInputAction**(java.lang.String action) Updates the state according to user action.
void	**render**(Graphics g) Renders the current state.

Method Detail

render

```
void render(Graphicsg)
```

Renders the current state.

Parameters:
g - The graphics context upon which to render the state.

---

doInputAction

```
void doInputAction(java.lang.Stringaction)
```

Updates the state according to user action.

Parameters:
`action` - A String representing an action.

Class ApplicationStateManager

```
java.lang.Object
  └ ApplicationStateManager
```

```
public class ApplicationStateManagerextends java.lang.Object
```

Keeps track of the current states and switches between them. Input provided by the player is routed to the state with the lowest int index. Network actions are routed to the GAME if possible, otherwise to the LOBBY state. This class will extend the BasicGame class and thus have an update loop. This loop will be responsible for gathering network commands, and will call forwardNetworkActions.

Field Summary

private boolean[]	**activeStates** The states currently active.
static int	**GAME** Gamestate.
static int	**LOBBY** Gamestate.
static int	**MENU** Gamestate.
static ClientCommunication	**network** For server communication.
private ApplicationState[]	**states** Contains the actual states.

Constructor Summary

ApplicationStateManager()	

Method Summary

void	**ApplicationStateManager**() Constructor for the ApplicationStateManager class
void	**forwardInput**(java.lang.String action) Forwards actions from InputManager to the current state.
void	**forwardNetworkActions**(java.lang.String[] actions) Forwards server actions from the ClientCommunication class to current state.
static void	**main**(java.lang.String[] args) This is the main method called on application startup.
void	**setStateActive**(int state, boolean value) Sets a state as active.
void	**switchState**(int state) Change the current state.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

MENU

```
public static final int MENU
```

Gamestate.

See Also:
Constant Field Values

LOBBY

```
public static final int LOBBY
```

Gamestate.

See Also:
Constant Field Values

GAME

```
public static final int GAME
```

Gamestate.

See Also:
Constant Field Values

---

network

```
public static ClientCommunication network
```

For server communication.

---

states

```
private ApplicationState[] states
```

Contains the actual states.

---

activeStates

```
private boolean[] activeStates
```

The states currently active.

Constructor Detail

ApplicationStateManager

```
public ApplicationStateManager()
```

Method Detail

ApplicationStateManager

```
public void ApplicationStateManager()
```

Constructor for the ApplicationStateManager class

---

switchState

```
public void switchState(intstate)
```

Change the current state.

Parameters:
`state` - One of the three predefined states MENU, LOBBY and GAME.

---

forwardInput

```
public void forwardInput(java.lang.Stringaction)
```

Forwards actions from InputManager to the current state.

Parameters:
`action` - A String representing an action.

---

forwardNetworkActions

```
public void forwardNetworkActions(java.lang.String[]actions)
```

Forwards server actions from the ClientCommunication class to current state. This method is called by the program's update loop.

Parameters:
`actions` - An array of Strings representing actions to perform.

---

setStateActive

```
public void setStateActive(intstate,
                           booleanvalue)
```

Sets a state as active.

Parameters:

`state` - The state to be set.
`value` - true for active.

---

main

```
public static void main(java.lang.String[]args)
```

This is the main method called on application startup.

Class Board

```
java.lang.Object
  └ Board
```

```
public class Boardextends java.lang.Object
```

A class representing a Board.

Field Summary	
Brick [][]	**board** A matrix containing the Bricks in this Board.

Constructor Summary	
Board(int width, int height) Constructor for Board.	

Method Summary	
void	**fixPiece**(Piece piece) Fixates a Piece on the Board by splitting it into Bricks and moving them to this Board.
Brick	**getBrick**(int x, int y)

	Gets the Brick in a specific position.
Brick	**removeBrick**(int x, int y) Removes the Brick at the specified position.
int[]	**removeRow**(int y) Removes all the Bricks in the specified row.
Brick	**setBrick**(int x, int y, Brick brick) Adds a Brick to the specified position.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

board

```
public Brick[][] board
```

A matrix containing the Bricks in this Board.

Constructor Detail

Board

```
public Board(intwidth,
             intheight)
```

Constructor for Board.

Parameters:

`width` - The width of the Board.
`height` - The height of the Board.

Method Detail

removeRow

```
public int[] removeRow(inty)
```

Removes all the Bricks in the specified row. All the bricks above the specified row that has been fixated will move down one step. If any of the removed bricks contains a powerup it will be extracted and returned. This method is called by doActions in the GameLogic class.

Parameters:
`y` - The index of the row to be removed.
Returns:
A list containing the extracted powerups.
See Also:
GameLogic

---

fixPiece

```
public void fixPiece(Piecepiece)
```

Fixates a Piece on the Board by splitting it into Bricks and moving them to this Board. This method is called by doActions in the GameLogic class.

Parameters:
`piece` - The Piece to fixate.
See Also:
GameLogic

---

removeBrick

```
public Brick removeBrick(intx,
                         inty)
```

Removes the Brick at the specified position. This method is called by doActions in the GameLogic class.

Parameters:
`x` - The x-coordinate of the Brick to be removed.
`y` - The y-coordinate of the Brick to be removed.
Returns:
The Brick that was removed.
See Also:
GameLogic

---

setBrick

```
public Brick setBrick(intx,
                      inty,
                      Brickbrick)
```

Adds a Brick to the specified position. If the specified position already contains a brick it will be returned. This method is called by doActions in the GameLogic class.

Parameters:

`x` - The x-coordinate of the brick to be added to the Board.

`y` - The y-coordinate of the brick to be added to the Board.

`brick` - The Brick to be added to the Board.

Returns:

The previous Brick the specified position, or null if the specified position was empty.

See Also:

GameLogic

---

getBrick

```
public Brick getBrick(intx,
                      inty)
```

Gets the Brick in a specific position. This method is called by doActions in the GameLogic class.

Parameters:

`x` - The x-coordinate.

`y` - The y-coordinate.

Returns:

The brick at the specified position, or null if the specified position was empty.

See Also:

GameLogic

Class Brick

```
java.lang.Object
  └ Brick
```

```
public class Brickextends java.lang.Object
```

Class representing a Brick.

Field Summary

private int	**edges** An int representing where the Brick is connected to other Bricks.
private Player	**owner** The player that is or was in control of this Brick.
private int	**powerupType** The type of the powerup contained, 0 if none.

Constructor Summary

Brick(Player player, int edges) Constructor for Brick.	

Method Summary

int	**getEdges**() Gets the neighbors of this Brick.
Player	**getPlayer**() Retrieves the owner of this Brick.
int	**getPowerup**() Retrieves the powerup contained in this brick.
void	**removeEdges**(int edges) Removes edges from this Brick where a bit is set to 1.
void	**setPowerup**(int powerup) Sets the powerup contained in this Brick.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

owner

```
private Player owner
```

The player that is or was in control of this Brick.

edges

```
private int edges
```

An int representing where the Brick is connected to other Bricks. Each bit is 1 if the edge exists or 0 otherwise. The four least significant bits represent, from most significant to least significant: Top, right, down, left.

---

powerupType

```
private int powerupType
```

The type of the powerup contained, 0 if none.

Constructor Detail

Brick

```
public Brick(Playerplayer,
             intedges)
```

Constructor for Brick.

Parameters:

`player` - The player controlling the Brick.
`edges` - The neighbors of the brick.

Method Detail

setPowerup

```
public void setPowerup(intpowerup)
```

Sets the powerup contained in this Brick.

Parameters:

`powerup` - The powerup type, 0 if none.

---

getPowerup

```
public int getPowerup()
```

Retrieves the powerup contained in this brick.

Returns:
The type of the powerup contained, 0 if none.

---

getPlayer

```
public Player getPlayer()
```

Retrieves the owner of this Brick.

Returns:
Returns the player that owns the brick.

---

getEdges

```
public int getEdges()
```

Gets the neighbors of this Brick.

Returns:
Returns the number of edges with neighbours.

---

removeEdges

```
public void removeEdges(intedges)
```

Removes edges from this Brick where a bit is set to 1.

Parameters:
`edges` - The edges to remove.

Class CentralServer

```
java.lang.Object
```

└ **CentralServer**

---

public class **CentralServer**extends java.lang.Object

Class representing the central server.

Field Summary

private	**gameServers** Array of connection sockets, one per game server.
private	**serverList** List containing information about any game servers.
private ServerSocket	**serverSocket** A ServerSocket for listening to incoming connections.

Constructor Summary

CentralServer() Constructor for CentralServer.	

Method Summary

void	**addGameServer**(java.lang.String[] serverInfo) Adds a server to the game server list.
	getGameServers() Gets the list of game servers.
java.lang.String[]	**getServer**(java.lang.String name) Gets the game server with the given name.
void	**mainLoop**() The main loop, which is responsible for checking server availability (removes game servers that cannot be contacted.
void	**removeGameServer**(java.lang.String address) Removes a server from the game server list.
void	**resetTimeout**(java.lang.String address) Resets the timeout of the given server.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

serverSocket

```
private ServerSocket serverSocket
```

A ServerSocket for listening to incoming connections.

---

gameServers

```
private  gameServers
```

Array of connection sockets, one per game server. These connections are closed as soon as the server has completed sending its information.

---

serverList

```
private  serverList
```

List containing information about any game servers. Contains each server's name, address, the maximum number of players, information about whether the server is public or not (true or false), and the server's timeout. The format is as specified: [name address nrOfPlayers privateGame timeout]

Constructor Detail

CentralServer

```
public CentralServer()
```

Constructor for CentralServer.

Method Detail

mainLoop

```
public void mainLoop()
```

The main loop, which is responsible for checking server availability (removes game servers that cannot be contacted. The timeout is incremented for each game server in regular intervals, and any servers with too great a timeout are removed. It is each game server's responsiblity to contact this central server and reset the timeout.

---

resetTimeout

```
public void resetTimeout(java.lang.Stringaddress)
```

Resets the timeout of the given server.

Parameters:
`address` - The address of the game server whose timeout to reset.

---

getServer

```
public java.lang.String[] getServer(java.lang.Stringname)
```

Gets the game server with the given name.

Parameters:
`name` - The name of the game server to retrieve.
Returns:
The information of the game server with the given name, or null if it does not exist.

---

addGameServer

```
public void addGameServer(java.lang.String[]serverInfo)
```

Adds a server to the game server list.

Parameters:
`serverInfo` - A list of strings containing information about the server.

---

removeGameServer

```
public void removeGameServer(java.lang.Stringaddress)
```

Removes a server from the game server list.

Parameters:
`address` - The unique address of the server to be removed.

---

getGameServers

```
public  getGameServers()
```

Gets the list of game servers.

Returns:
A list of strings containing information about the server.

Class ClientCommunication

```
java.lang.Object
  └ ClientCommunication
```

```
public class ClientCommunicationextends java.lang.Object
```

A class for handling communication between the client and the GameServer or CentralServer.

Field Summary	
static int	**CENTRAL_SERVER** An int representing the CentralServer.
private static java.lang.String	**CENTRAL_SERVER_ADDRESS** The address of the currently active CentralServer.
private Socket	**centralServerSocket** A socket connected to the current CentralServer.
static int	**GAME_SERVER** An int representing the GameServer.
private java.lang.String	**gameServereAddress** The address of the currently active GameServer.

private Socket	**gameServerSocket** A socket connected to the current GameServer.

Constructor Summary

ClientCommunication(java.lang.String centralServerAddress) Constructor for ClientCommunication.	

Method Summary

java.lang.String[]	**getActions**(int server) Gets any new actions sent by a given server.
void	**sendAction**(java.lang.String action, int server) Sends an action to the current GameServer.
void	**setGameServer**(java.lang.String gameServerAddress) Sets the current GameServer.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

GAME_SERVER

```
public static final int GAME_SERVER
```

An int representing the GameServer.

CENTRAL_SERVER

```
public static final int CENTRAL_SERVER
```

An int representing the CentralServer.

CENTRAL_SERVER_ADDRESS

```
private static final java.lang.String CENTRAL_SERVER_ADDRESS
```

The address of the currently active CentralServer.

---

gameServereAddress

```
private java.lang.String gameServereAddress
```

The address of the currently active GameServer.

---

gameServerSocket

```
private Socket gameServerSocket
```

A socket connected to the current GameServer.

---

centralServerSocket

```
private Socket centralServerSocket
```

A socket connected to the current CentralServer.

Constructor Detail

ClientCommunication

```
public ClientCommunication(java.lang.StringcentralServerAddress)
```

Constructor for ClientCommunication.

Parameters:

`centralServerAddress` - The address of the current CentralServer.

Method Detail

setGameServer

```
public void setGameServer(java.lang.StringgameServerAddress)
```

Sets the current GameServer.

Parameters:
`gameServerAddress` - The address of the current GameServer.

---

sendAction

```
public void sendAction(java.lang.Stringaction,
                       intserver)
```

Sends an action to the current GameServer. This method is called by the doInputAction methods of the GameSessionState and LobbyState classes.

Parameters:
`action` - The action to send.
`server` - The server to send the action to. Should be either GAME_SERVER or CENTRAL_SERVER.
See Also:
GameSessionState, LobbyState

---

getActions

```
public java.lang.String[] getActions(intserver)
```

Gets any new actions sent by a given server. This method is called by the update loop of the ApplicationStateManager.

Parameters:
`server` - The server whose actions to get. Should be either GAME_SERVER or CENTRAL_SERVER.
See Also:
ApplicationStateManager

Class ControllerMap

```
java.lang.Object
  └ ControllerMap
```

```
public class ControllerMapextends java.lang.Object
```

A class for mapping user input to in-game commands.

Constructor Summary

ControllerMap() Constructor for the ControllerMap.	

Method Summary

java.lang.String	**parseInput**(java.lang.String input) Parses an input string and converts it to an in-game action.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ControllerMap

```
public ControllerMap()
```

Constructor for the ControllerMap.

Method Detail

parseInput

```
public java.lang.String parseInput(java.lang.Stringinput)
```

Parses an input string and converts it to an in-game action. This method is called by getAction in the InputManager class.

Parameters:

`input` - The input to parse.

Returns:
An action depending on the input.
See Also:
InputManager

Class GameLogic

```
java.lang.Object
  └ GameLogic
```

```
public class GameLogicextends java.lang.Object
```

The GameLogic class, responsible for the game logic. For example, user commands are handled by the game logic, which then determines whether they are valid or not.

Field Summary

private Board	**board** The game Board used in the current game session.
private	**players** The list of players participating in the game.

Constructor Summary

GameLogic(players, int width, int height) The constructor for GameLogic.	

Method Summary

void	**doActions**(java.lang.String[] actions) Performs a list of given actions in order from first to last.
private void	**dropPiece**(Player player) Moves the given player's piece downward as far as possible, and then fixates it.
Board	**getBoard**() Gets the game Board used in the current game session.

	getPlayers() Gets a list of the players participating in the game session.
private boolean	**movePiece**(Player player, int direction) Moves a player's piece in the specified direction.
private void	**rotatePiece**(Player player, int clockwise) Rotates the player's piece a given number of steps.
private void	**usePowerup**(Player player, int slot) Uses the powerup in the given slot, held by the given player.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

players

```
private  players
```

The list of players participating in the game.

---

board

```
private Board board
```

The game Board used in the current game session.

Constructor Detail

GameLogic

```
public GameLogic(players,
                 intwidth,
                 intheight)
```

The constructor for GameLogic.

Parameters:

`players` - The players participating in the game session.
`width` - The width of the Board.
`height` - The height of the Board.

Method Detail

getBoard

```
public Board getBoard()
```

Gets the game Board used in the current game session. This method is called by the render method of the GameSessionState class.

Returns:
the board
See Also:
`GameSessionState`

---

getPlayers

```
public  getPlayers()
```

Gets a list of the players participating in the game session.

Returns:
A list containing the participating players.

---

doActions

```
public void doActions(java.lang.String[]actions)
```

Performs a list of given actions in order from first to last. Each action is represented as a string. For example, rotating "player three"'s current piece counter-clockwise one step could be similar to "p3 ccw1". This method is called by doNetworkActions in GameSessionState. This method calls usePowerup, movePiece, rotatePiece, and dropPiece.

Parameters:
`actions` - A list of strings representing the actions to perform.
See Also:
`GameSessionState`

---

usePowerup

```
private void usePowerup(Playerplayer,
                        intslot)
```

Uses the powerup in the given slot, held by the given player. This method is called by doActions.

Parameters:

`player` - The player that is holding the powerup to be used.
`slot` - The slot containing the powerup.

---

movePiece

```
private boolean movePiece(Playerplayer,
                          intdirection)
```

Moves a player's piece in the specified direction. The piece is only moved if the requested move is valid. This method is called by doActions.

Parameters:

`player` - The player whose piece to move.
`direction` - The direction in which to move the piece.

Returns:

True if the piece was successfully moved, false otherwise.

---

rotatePiece

```
private void rotatePiece(Playerplayer,
                         intclockwise)
```

Rotates the player's piece a given number of steps. This method is called by doActions, and in turn calls the rotate method of the Piece class.

Parameters:

`player` - The player whose piece to rotate.
`clockwise` - The number of steps to rotate in a clockwise direction. Negative values will rotate the piece counter-clockwise.

See Also:

`Piece`

---

dropPiece

```
private void dropPiece(Playerplayer)
```

Moves the given player's piece downward as far as possible, and then fixates it. This method is called by doActions.

Parameters:

`player` - The player whose piece to drop.

Class GameServer

```
java.lang.Object
  └ GameServer
```

```
public class GameServerextends java.lang.Object
```

Class representing a game server. The class uses sockets to collect input from the clients, which are then sent with timestamps to each client. Each client is then responsible for determining whether a command is valid or not.

Field Summary	
private	**actions** Array of actions to perform.
private boolean	**gameStarted** The gameStarted property.
private int	**numberOfPlayers** The number of players in the game.
private	**players** Array of connection sockets, one per player.
private ServerSocket	**serverSocket** A ServerSocket for listening to incoming connections.
private int	**timeStamp** The timeStamp property holds the current time stamp of the server.

Constructor Summary

GameServer(int numberOfPlayers) Constructor for GameServer.	

Method Summary

void	**mainLoop**() The main loop, responsible for collecting commands and activating sendPulse at given intervals.
void	**sendPulse**() Synchronizes the clients by sending collected commands, bundled with timestamps.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

serverSocket

```
private ServerSocket serverSocket
```

A ServerSocket for listening to incoming connections.

---

players

```
private  players
```

Array of connection sockets, one per player.

---

actions

```
private  actions
```

Array of actions to perform.

---

gameStarted

```
private boolean gameStarted
```

The gameStarted property. True when the game session is running, false otherwise.

---

numberOfPlayers

```
private int numberOfPlayers
```

The number of players in the game.

---

timeStamp

```
private int timeStamp
```

The timeStamp property holds the current time stamp of the server. It is incremented once each time a pulse is sent.

Constructor Detail

GameServer

```
public GameServer(intnumberOfPlayers)
```

Constructor for GameServer.

Parameters:

`numberOfPlayers` - The number of players to participate in the game.

Method Detail

mainLoop

```
public void mainLoop()
```

The main loop, responsible for collecting commands and activating sendPulse at given intervals.

sendPulse

```
public void sendPulse()
```

Synchronizes the clients by sending collected commands, bundled with timestamps. This method is called by mainLoop.

Class GameSessionState

```
java.lang.Object
  └ GameSessionState
```

All Implemented Interfaces:
ApplicationState

```
public class GameSessionStateextends java.lang.Objectimplements ApplicationState
```

A class representing the game session state.

Field Summary

private java.lang.StringBuffer	**chatMessage** The chatMessage being written.
private boolean	**isChatting** The isChatting property, indicating whether the user is currently entering a chat message or not.
private GameLogic	**logic** The GameLogic instance used in this game session.
private ApplicationStateManager	**manager** The ApplicationStateManager controlling this state.

Constructor Summary

GameSessionState(ApplicationStateManager manager, players, int width, int height) The constructor for GameSessionState.

Method Summary

void	**doInputAction**(java.lang.String action) Updates the state according to user action.
void	**doNetworkActions**(java.lang.String[] actions) Updates the state according to the given server actions.
private void	**leaveState**() Leaves the GameSessionState and enters the MenuState.
void	**render**(Graphics g) Renders the current state.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

manager

```
private ApplicationStateManager manager
```

The ApplicationStateManager controlling this state.

---

logic

```
private GameLogic logic
```

The GameLogic instance used in this game session.

---

isChatting

```
private boolean isChatting
```

The isChatting property, indicating whether the user is currently entering a chat message or not.

---

chatMessage

```
private java.lang.StringBuffer chatMessage
```

The chatMessage being written.

Constructor Detail

GameSessionState

```
public GameSessionState(ApplicationStateManagermanager,
                        players,
                        intwidth,
                        intheight)
```

The constructor for GameSessionState.

Parameters:

`manager` - The manager of this state.
`players` - A list containing the different players to participate in the game session.
`width` - The width of the Board.
`height` - The height of the Board.

Method Detail

render

```
public void render(Graphicsg)
```

Renders the current state.

Specified by:

`render` in interface `ApplicationState`

Parameters:

`g` - The graphics context upon which to render the state.

---

doInputAction

```
public void doInputAction(java.lang.Stringaction)
```

Updates the state according to user action. This method calls sendAction in ClientCommunication unless isChatting is true, in which case chatMessage is updated.

Specified by:
doInputAction in interface ApplicationState
Parameters:
action - A String representing an action.

---

doNetworkActions

```
public void doNetworkActions(java.lang.String[]actions)
```

Updates the state according to the given server actions. This method calls the doActions method of the GameLogic class.

Parameters:
actions - An array of Strings representing actions to perform.
See Also:
GameLogic

---

leaveState

```
private void leaveState()
```

Leaves the GameSessionState and enters the MenuState.

Class InputManager

```
java.lang.Object
  └ InputManager
```

```
public class InputManagerextends java.lang.Object
```

A class that gathers user input, passes it through a ControllerMap, and then sends it to an ApplicationStateManager.

Field Summary

private int	**inputDeviceId** An integer representing the current input device.
private ApplicationStateManager	**manager** The ApplicationStateManager to recieve any input.
private ControllerMap	**map** Maps player input to in-game commands, for easy setup of multiple input configurations.

Constructor Summary

InputManager(ControllerMap cmap, int inputDeviceID) Constructor for the InputManager class.	

Method Summary

java.lang.String	**getAction**(java.lang.String input) Get the in-game action associated with the the given input.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

inputDeviceId

```
private int inputDeviceId
```

An integer representing the current input device.

---

manager

```
private ApplicationStateManager manager
```

The ApplicationStateManager to recieve any input.

---

map

```
private ControllerMap map
```

Maps player input to in-game commands, for easy setup of multiple input configurations.

Constructor Detail

InputManager

```
public InputManager(ControllerMapcmap,
                    intinputDeviceID)
```

Constructor for the InputManager class.

Parameters:

`cmap` - The ControllerMap to use in this InputManager.
`inputDeviceID` - The ID of the input device to use.

Method Detail

getAction

```
public java.lang.String getAction(java.lang.Stringinput)
```

Get the in-game action associated with the the given input. This method is called when input is gathered, and in turn calls the parseInput method of the ControllerMap class.

Parameters:

`input` - A string containing the user input.

Returns:

A string representing the associated action, as given by the ControllerMap.

See Also:

`ControllerMap`

Class LobbyState

```
java.lang.Object
  └ LobbyState
```

All Implemented Interfaces:
ApplicationState

```
public class LobbyStateextends java.lang.Objectimplements ApplicationState
```

Class representing a lobby state.Implements the ApplicationState interface. The lobby state contains the host and join multiplayer submenus together with a chat area.

Field Summary

private int	**activeWindow** The submenu currently beeing navigated.
private static int	**HOST_WINDOW** One possible submenu.
private boolean	**isHost** True if a game session is being hosted.
private static int	**JOIN_WINDOW** One possible submenu.
private static int	**LIST_WINDOW** One possible submenu.
private static int	**LOBBY_WINDOW** One possible submenu.
private ApplicationStateManager	**manager** The ApplicationStateManager controlling this state.
private int	**numberOfPlayers** The number of players set in a hosted game.

Constructor Summary

LobbyState(ApplicationStateManager manager) Constructor for LobbyState.	

Method Summary

private void	**cancel**() Goes one step back in the menu hierarchy.
private void	**createGame**(java.lang.String nickname, int numberOfPlayers,

	boolean privateGame) Creates a new game session.
void	**doInputAction**(java.lang.String action) Updates the state according to user action.
void	**doNetworkActions**(java.lang.String[] actions) Updates the state according to the given server actions.
void	**hostGame**() Sets up a hosted game.
private void	**joinGame**(java.lang.String address) Joins a hosted multiplayer game session.
private void	**launchGame**() Starts a multiplayer game session.
private void	**leaveState**(int newState) Leaves the LobbyState and enters the MenuState or GameSessionState.
void	**listGames**() Gets available games from CentralServer.
void	**render**(Graphics g) Renders the current state.
private void	**sendMessage**(java.lang.String message) Constructs a chat message to be sent to the server.
private void	**showMessage**(java.lang.String message) Displays a received chat message.
private void	**startMultiplayer**() Starts a multiplayer game session.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

HOST_WINDOW

```
private static final int HOST_WINDOW
```

One possible submenu.

See Also:
Constant Field Values

JOIN_WINDOW

```
private static final int JOIN_WINDOW
```

One possible submenu.

See Also:
Constant Field Values

LIST_WINDOW

```
private static final int LIST_WINDOW
```

One possible submenu.

See Also:
Constant Field Values

LOBBY_WINDOW

```
private static final int LOBBY_WINDOW
```

One possible submenu.

See Also:
Constant Field Values

---

manager

```
private ApplicationStateManager manager
```

The ApplicationStateManager controlling this state.

---

activeWindow

```
private int activeWindow
```

The submenu currently beeing navigated.

isHost

```
private boolean isHost
```

True if a game session is being hosted.

---

numberOfPlayers

```
private int numberOfPlayers
```

The number of players set in a hosted game.

Constructor Detail

LobbyState

```
public LobbyState(ApplicationStateManagermanager)
```

Constructor for LobbyState.

Parameters:
`manager` - The manager of this state.

Method Detail

render

```
public void render(Graphicsg)
```

Renders the current state.

Specified by:
`render` in interface `ApplicationState`
Parameters:
`g` - The graphics context upon which to render the state.

---

doInputAction

```
public void doInputAction(java.lang.Stringaction)
```

Updates the state according to user action.

Specified by:
doInputAction in interface ApplicationState
Parameters:
action - A String representing an action.

---

doNetworkActions

```
public void doNetworkActions(java.lang.String[]actions)
```

Updates the state according to the given server actions.

Parameters:
actions - An array of Strings representing actions to perform.

---

listGames

```
public void listGames()
```

Gets available games from CentralServer.

---

hostGame

```
public void hostGame()
```

Sets up a hosted game.

---

cancel

```
private void cancel()
```

Goes one step back in the menu hierarchy.

---

leaveState

```
private void leaveState(intnewState)
```

Leaves the LobbyState and enters the MenuState or GameSessionState.

Parameters:

`newState` - The state to enter.

---

showMessage

```
private void showMessage(java.lang.Stringmessage)
```

Displays a received chat message.

Parameters:

`message` - A String representing a chat message.

---

sendMessage

```
private void sendMessage(java.lang.Stringmessage)
```

Constructs a chat message to be sent to the server.

Parameters:

`message` - A String representing the message to be sent.

---

startMultiplayer

```
private void startMultiplayer()
```

Starts a multiplayer game session.

---

createGame

```
private void createGame(java.lang.Stringnickname,
                        intnumberOfPlayers,
                        booleanprivateGame)
```

Creates a new game session.

Parameters:

`nickname` - The nickname of the hosting player.
`numberOfPlayers` - The number of players allowed.
`privateGame` - True if this game session is to be hidden.

---

launchGame

```
private void launchGame()
```

Starts a multiplayer game session.

---

joinGame

```
private void joinGame(java.lang.Stringaddress)
```

Joins a hosted multiplayer game session.

Parameters:
`address` - The address to the host.

Class MenuState

```
java.lang.Object
  └MenuState
```

All Implemented Interfaces:
ApplicationState

```
public class MenuStateextends java.lang.Objectimplements ApplicationState
```

A class representing the menu state of the game.

Field Summary	
private ApplicationStateManager	**manager** The ApplicationStateManager controlling this state.

Constructor Summary	
MenuState(ApplicationStateManager manager)	

	The constructor for MenuState.

Method Summary

void	**doInputAction**(java.lang.String action) Performs a menu action depending on the action string provided.
private void	**exit**() Exits the game.
void	**render**(Graphics g) Renders the state.
private void	**showHelp**() Displays the help documention for the game.
private void	**showLobby**(boolean host) Switches to the lobby state.
private void	**startSingleplayer**() Starts a singleplayer game session.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

manager

```
private ApplicationStateManager manager
```

The ApplicationStateManager controlling this state.

Constructor Detail

MenuState

```
public MenuState(ApplicationStateManagermanager)
```

The constructor for MenuState.

Parameters:

manager - The ApplicationStateManager to control this state.

Method Detail

render

```
public void render(Graphicsg)
```

Renders the state.

Specified by:
render in interface ApplicationState
Parameters:
g - The graphics context upon which to render the state.

---

doInputAction

```
public void doInputAction(java.lang.Stringaction)
```

Performs a menu action depending on the action string provided.

Specified by:
doInputAction in interface ApplicationState
Parameters:
action - A string representing the action the user wants to perform.

---

showHelp

```
private void showHelp()
```

Displays the help documention for the game.

---

showLobby

```
private void showLobby(booleanhost)
```

Switches to the lobby state.

Parameters:
host - True if the user has requested to host a game, false if he wants to see a list of games.

---

exit

```
private void exit()
```

Exits the game.

---

startSingleplayer

```
private void startSingleplayer()
```

Starts a singleplayer game session.

Class Piece

```
java.lang.Object
  └ Piece
```

```
public class Pieceextends java.lang.Object
```

The Piece class, representing all the different piece types.

Field Summary	
private Brick[][]	**matrix** An array containing the Bricks held in the current piece.
private char	**name** The name of the piece type.
private Player	**owner** The owner of this Piece.
private int	**rotationState** The current rotation state of the piece.

Constructor Summary

Piece(Player player, char pieceType) Constructor for Piece.	

Method Summary

boolean	**rotate**(int clockwise) Rotates the piece clockwise a given number of steps.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

matrix

```
private Brick[][] matrix
```

An array containing the Bricks held in the current piece.

---

name

```
private char name
```

The name of the piece type. Must be either I, J, L, O, S, T or Z.

---

rotationState

```
private int rotationState
```

The current rotation state of the piece.

---

owner

```
private Player owner
```

The owner of this Piece.

Constructor Detail

Piece

```
public Piece(Playerplayer,
             charpieceType)
```

Constructor for Piece.

Parameters:

`player` - The Player who controls this piece.
`pieceType` - The type of piece.

Method Detail

rotate

```
public boolean rotate(intclockwise)
```

Rotates the piece clockwise a given number of steps. This method is called by the rotatePiece method in the GameLogic class.

Parameters:

`clockwise` - The number of steps to rotate clockwise. Negative values rotates the piece counter-clockwise.

See Also:

`GameLogic`

Class PieceGenerator

```
java.lang.Object
  └ PieceGenerator
```

---

public class **PieceGenerator**extends java.lang.Object

A class representing a piece generator. When instanced, the piece generator will create two bags that each hold one of each piece in random order. When a bag is empty the other bag will take over while the first one is refilled.

Field Summary

private Player	**myPlayer** The Player object controlling the PieceGenerator.
static java.util.ArrayList<java.util.ArrayList<Brick[][]>>	**pieceStates** A list representing the different possible pieces.
private Random	**rand** Used to generate random numbers internally.
private ShuffleBag[]	**s** A list of "bags" that hold randomly generated pieces.

Constructor Summary

PieceGenerator(Player player, int seed) Constructor for PieceGenerator.	

Method Summary

Piece	**getNext**() Get the next randomly generated piece.
Piece	**peekNext**(int offset) Looks at a future piece held in one of the ShuffleBags.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

pieceStates

```
public static java.util.ArrayList<java.util.ArrayList<Brick[][]>> pieceStates
```

A list representing the different possible pieces. The outer list holds different piece types, the inner list holds all possible rotations.

---

rand

```
private Random rand
```

Used to generate random numbers internally.

---

myPlayer

```
private Player myPlayer
```

The Player object controlling the PieceGenerator.

---

s

```
private ShuffleBag[] s
```

A list of "bags" that hold randomly generated pieces.

Constructor Detail

PieceGenerator

```
public PieceGenerator(Playerplayer,
                      intseed)
```

Constructor for PieceGenerator.

Parameters:

`player` - The player that this PieceGenerator belongs to.
`seed` - A random seed for this PieceGenerator.

Method Detail

getNext

```
public Piece getNext()
```

Get the next randomly generated piece.

Returns:
A random Piece.

---

peekNext

```
public Piece peekNext(intoffset)
```

Looks at a future piece held in one of the ShuffleBags.

Parameters:
`offset` - The offset index of the piece. The next piece is at index 0.
Returns:
The piece with the given offset in the list of upcoming pieces.

Class Player

```
java.lang.Object
  └ Player
```

```
public class Playerextends java.lang.Object
```

A class representing a player.

Field Summary	
private Piece	**currentPiece** The piece that the player is currently controlling.
private int	**id** The player's id (to avoid name collisions).

private java.lang.String	**name** The name of the player.
private PieceGenerator	**pieceGenerator** A generator that generates random pieces for the player .
private int[]	**powerups** A list containing the player's powerups.

Constructor Summary

Player(java.lang.String name, int id, int slots, int seed) Constructor for Player.	

Method Summary

void	**addPowerup**(int powerup) Add a powerup to the player's list of powerups.
Piece	**getCurrentPiece**() Returns player's current piece.
int[]	**getPowerUps**() Get a list of the player's powerups.
void	**newPiece**() Generates a new piece and gives the player control of it.
int	**removePowerup**(int index) Remove a given powerup from the player's powerup list.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

currentPiece

```
private Piece currentPiece
```

The piece that the player is currently controlling.

---

powerups

```
private int[] powerups
```

A list containing the player's powerups. Each powerup is represented by an int, defining the type.

---

name

```
private java.lang.String name
```

The name of the player.

---

id

```
private int id
```

The player's id (to avoid name collisions).

---

pieceGenerator

```
private PieceGenerator pieceGenerator
```

A generator that generates random pieces for the player .

Constructor Detail

Player

```
public Player(java.lang.Stringname,
              intid,
              intslots,
              intseed)
```

Constructor for Player.

Parameters:

`name` - the name of the player.
`id` - the id of the player.

`slots` - the maximum number of powerups that can be held by this player.
`seed` - a random seed for the PieceGenerator.

Method Detail

getCurrentPiece

```
public Piece getCurrentPiece()
```

Returns player's current piece.

Returns:
The player's current piece.

---

getPowerUps

```
public int[] getPowerUps()
```

Get a list of the player's powerups.

Returns:
An int array containing the player's puwerups.

---

removePowerup

```
public int removePowerup(intindex)
```

Remove a given powerup from the player's powerup list.

Parameters:
`index` - The index of the powerup to be removed.
Returns:
An integer representing the removed powerup. If the supplied index is invalid -1 is returned.

---

addPowerup

```
public void addPowerup(intpowerup)
```

Add a powerup to the player's list of powerups.

Parameters:
`powerup` - An integer representing the type of powerup to add.

---

newPiece

```
public void newPiece()
```

Generates a new piece and gives the player control of it. This method is called from a GameLogic doActions method when needed.

See Also:
GameLogic

5.5.2. Requirement references

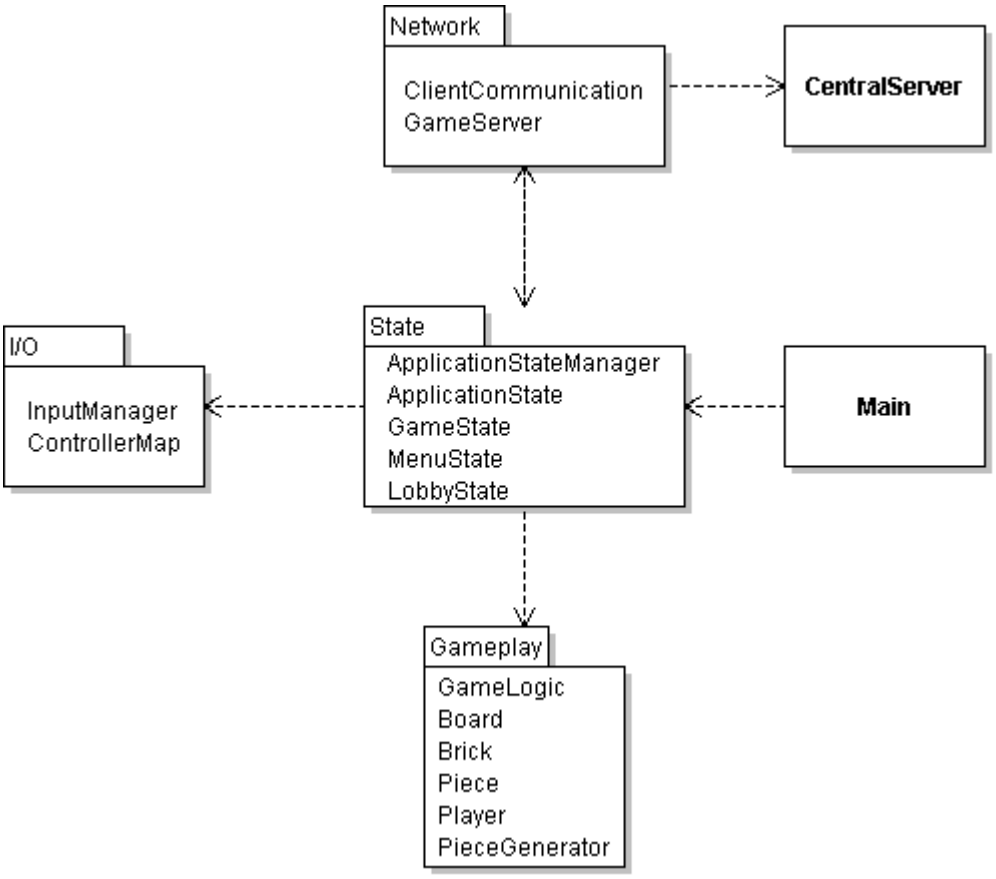
The table below links each requirement in the RD to the design in the DD. Each requirement is numbered according to the corresponding requirement of the RD.

Requirements Document	Design Document implementation
#1 Game pieces	The Piece and Brick classes.
#2 Boundaries	The Board and GameLogic classes (specifically, the movePiece method in the GameLogic class).
#3 Falling pieces	The Piece, GameLogic and GameServer classes (the GameLogic doNetworkActions recieves pulses from the GameServer when it is to move a Piece downward).
#4 Score	The GameLogic handles this in the various methods that move and rotate Pieces and Bricks.
#5 Get help	The MenuState class's showHelp method.
#6 Menu system	The MenuState class's render method shows the options, while input is handled by doActions.
#7 Start singleplayer game session	The MenuState class's startSingleplayer method.
#8 Host multiplayer game	The MenuState and LobbyState classes handle this. In MenuState the showLobby method is used, and in the LobbyState createGame and launchGame are used.
#9 Host private game	Same as above, with a String to indicate that the game is private sent to the CentralServer from the createGame method.
#10 Join multiplayer game	The MenuState and LobbyState classes handle this. In the MenuState the showLobby method is used, and in the LobbyState listGames is used.

#11 Chat with other users	The LobbyState and GameSessionState respectively handles this with doInputActions and doNetworkActions.

#12 Move piece vertically	The GameLogic class's dropPiece handles this.
#13 Move piece horizontally	The GameLogic class's movePiece handles this.
#14 Rotate piece	The GameLogic class's rotatePiece handles this.
#15 Collide with other player-controlled bricks	The GameLogic class's movePiece and dropPiece handle this.
#16 Piece control	The GameLogic class's doActions and the Player class's newPiece handle this.
#17 Fixate piece	The GameLogic class's movePiece and dropPiece and the Board class's fixPiece handle this.
#18 Remove row	The GameLogic class's movePiece and dropPiece and the Board class's removeRow handle this.
#19 Finishing a game session	The GameLogic class's createPiece handles this.
#20 Obtain powerup	The GameLogic class's doActions, the Board class's removeRow and the Player class's addPowerup handle this.
#21 Create powerup	The GameLogic class's doActions, the Board class's removeRow and getBrick, and the Brick class's setPowerup handle this (That is, when a row is removed, doActions will call getBrick and change that Brick to a powerup Brick using setPowerup).
#22 Use powerup	The GameLogic class's doActions and the Player class's removePowerup handle this.

5.6. Package Diagram



6. Functional test cases

Below are test cases ordered by the functionality requirement covered. Each test case corresponds to the requirement with the same number in the Requirements Document.

6.1. Test Descriptions

6.1.1. Game properties

#1 Game pieces
Requirement: There shall be seven different pieces which can be placed. Each piece consists of four parts called bricks.
Input: Controller input
Output: Visible effect; seven different game pieces shown
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Move the first piece that falls down to the leftmost part of the board.

3. For the next six pieces, verify that each piece is visibly different from the all previous pieces.

#2 Boundaries
Requirement: There shall be boundaries that restricts piece movement. Specifically, there is the top, lower and side boundaries. No piece may pass any boundary other than the top one.
Input: Controller input
Output: Visible effect; the piece cannot be moved outside the boundaries
Step-by-step procedure:
1. Select "Sinleplayer" from the main menu.
2. Move the first piece that falls down to the leftmost part of the board and verify that it can go no further.
3. Move the second piece that falls down to the rightmost part of the board and verify that it can go no further.
4. Let the third piece fall to the ground, and verify that it gets fixed to the bottom of the board.

#3 Falling pieces
Requirement: Any piece controlled by a player shall move downward with a given speed, that is increased when a piece gets fixated.
Input: Controller input
Output: Visible effect; a piece will have a downward speed that increases as the game progresses
Step-by-step procedure:
1. Select "Singleplayer" from the main menu.
2. Verify that a piece is shown at the top of the board, and that it has a downwards speed.
3. Fixate the first piece to the board.
4. Verify that the speed increases (there may not be noticeable effect after only one piece was fixated, so fixate 10 pieces before measuring the speed again).

#4 Score
Requirement: The system shall keep and display a total score that is altered when pieces are fixated or pieces collide.
Input: Controller input
Output: Visible effect; the score meter will increase when a row is removed
Step-by-step procedure:
1. Select "Singleplayer" from the main menu.
2. Fixate pieces so that a complete row is formed.
3. Verify that the score is increased.
4. Enter the main menu, and select "Host game".
5. Start the game and have somebody join it.
6. Launch the game session.
7. Move the two player-controlled pieces together.
8. Keep trying to move the pieces into each other for one second.
9. Verify that the pieces are destroyed and the score decreased.

#5 Get help
Requirement: The system shall provide a manual for the user. The manual shall describe how to start, stop and play a game session, as well as how to navigate pieces.
Input: Controller input
Output: The game displays the help documentation
Step-by-step procedure:
1. Select "Help" from the main menu.
2. Verify that said points (start, stop, playing a game session, navigating pieces) are included in the help

documentation.

#6 Menu system
Requirement: The system shall allow the user to choose from one of the following options through a menu system: Singleplayer, Host game, Join game, Help, Exit game.
Input: Controller input
Output: Visible effects; different screens are shown depending on user input.
Step-by-step procedure:

1. Select “Host game” from the main menu.
2. Verify that the host game screen is shown.
3. Go back to the main menu.
4. Select “Join game” from the main menu.
5. Verify that the join game screen is shown.
6. Go back to the main menu.
7. Select “Singleplayer” from the main menu.
8. Verify that a game session is started.
9. Go back to the main menu.
10. Select “Help” from the main menu.
11. Verify that the help screen is shown.
12. Go back to the main menu.
13. Select “Exit” from the main menu.
14. Verify that the application was shut down.

6.1.2. Game sessions

#7 Start singleplayer game session
Requirement: The player shall be able to start a singleplayer game session.
Input: Controller input
Output: A game session is started
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Verify that a game session starts (the board is shown).

#8 Host multiplayer game
Requirement: The system shall allow a user to start a game that other users are able to join.
Input: Controller input.
Output: A multiplayer game session is started
Step-by-step procedure:

1. Select “Host game” from the main menu.
2. Enter name and number of players as appropriate. Do not check the “Private game” checkbox.
3. Start the game.
4. Have somebody launch the client, select “Join game” from the main menu, find the game in the game list, and enter it.
5. Verify that the player is shown in the player list.
6. Launch the multiplayer game session.
7. Verify that a multiplayer game session was started.

#9 Host private game
Requirement: The user shall be able to host a private network game. This should be the same as hosting a standard

network game, except that the game will not show up in the game list for other users. However, the central server should still keep track of the game to allow the desired users to join it by entering its name.
Input: Controller input
Output: A multiplayer game session is started.
Step-by-step procedure:

1. Select “Host game” from the main menu.
2. Enter name and number of players as appropriate. Check the “Private game” checkbox.
3. Start the game.
4. Have somebody launch the client, select “Join game” from the main menu, and input the game name.
5. Verify that the player is shown in the player list.
6. Launch the multiplayer game session.
7. Verify that a multiplayer game session was started.

#10 Join multiplayer game

Requirement: The system shall allow users to join previously hosted games.
Input: Controller input
Output: A multiplayer game was joined.
Step-by-step procedure:

1. Have somebody launch the client and host a game with a given name.
2. Select “Join game” from the main menu.
3. Enter the game name.
4. Verify that the player lobby is shown.

Test: Letting another user start a network game session and joining it.

#11 Chat with other users

Requirement: The user shall be able to chat with other users when in a game. At any time during a game, a user should be able to send a message that will be displayed to all participating users.
Input: Controller input
Output: A message is received.
Step-by-step procedure:

1. Select “Host game” from the main menu.
2. Enter name and number of players as appropriate. Do not check the “Private game” checkbox.
3. Start the game.
4. Have somebody launch the client, select “Join game” from the main menu, find the game in the game list, and enter it.
5. Launch the multiplayer game session.
6. Press the “Chat” controller key, enter a chat message, and then press the “Chat” controller key again.
7. Verify that the other user recieves the message.
8. Have the other user write a message and send it.
9. Verify that you receive a message.

6.1.3. Piece movement

#12 Move piece vertically

Requirement: The user shall be able to increase the downward speed of the current piece by a constant factor.

Test: Starting a game session and making sure the piece can be accelerated downwards.
Input: Controller input
Output: Visible effect; the piece is accelerated downward
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Press the “Down” controller key to accelerate the piece.
3. Verify that the piece's downward speed increases.

#13 Move piece horizontally
Requirement: The user shall be able to move the piece horizontally with constant steps.
Input: Controller input
Output: Visible effect; the piece is moved sideways
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Press the “Left” conroller key to move the piece to the left.
3. Verify that the piece is moved to the left.
4. Press the “Right” controller key to move the piece to the right.
5. Verify that the piece is moved to the right.

#14 Rotate piece
Requirement: The user shall be able to rotate the piece he controls, either clockwise or counter-clockwise.
Input: Controller key
Output: Visible effect; the piece is rotated
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Press the “Rotate clockwise” controller key to rotate the piece clockwise.
3. Verify that the piece is rotated clockwise.
4. Press the “Rotate counter-clockwise” controller key to rotate the piece counter-clockwise.
5. Verify that the piece is rotated counter-clockwise.

#15 Collide with other player-controlled bricks
Requirement: Two non-fixed bricks that collide shall either be removed or moved apart. If they are removed, each player shall gain a new piece and a fixed score shall be subtracted from the total score.
Input: Controller input
Output: Visible effect; the pieces are removed or moved apart
Step-by-step procedure:

1. Select “Host game” from the main menu.
2. Enter name and number of players as appropriate. Do not check the “Private game” checkbox.
3. Start the game.
4. Have somebody launch the client, select “Join game” from the main menu, find the game in the game list, and enter it.
5. Launch the multiplayer game session.
6. Move the two player-controlled pieces together.
7. Keep trying to move the pieces into each other for one second.
8. Verify that the pieces are removed.
9. Verify that the score decreases.
10. Verify that each player recieves a new piece.
11. Move the new player-controlled pieces together.
12. Move the pieces apart before one second has passed.
13. Verify that the pieces are moved apart.

6.1.4. Brick placement

#16 Piece control
Requirement: When game session starts the user shall receive a brick to control. After a brick has been fixated the user shall receive a new brick.
Input: Controller input
Output: Visible effect; a piece is shown at the top of the board
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Verify that a piece is shown.
3. Press the “Drop” controller key to fixate the piece.
4. Verify that the piece was fixated.
5. Verify that a new piece is shown.

#17 Fixate piece
Requirement: When the vertical movement is obstructed by a fixated piece or the lower boundary, the piece shall stop moving and be taken out of player control.
Input: Controller input
Output: A piece stops moving and is no longer controlled by the player
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Press the “Drop” controller key to fixate the piece.
3. Verify that the piece is no longer moving.
4. Press the “Left” controller key to try moving the fixated piece left.
5. Verify that the piece was not moved.
6. Verify that the new piece was moved instead.

#18 Remove row
Requirement: Complete brick rows reaching between the two side boundaries shall be removed and points shall be added to the total score.
Input: Controller input
Output: Visible effect; the completed remove is removed and all bricks above the removed row are moved downwards.
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Move pieces so that a complete row is formed.
3. Verify that the row is removed.
4. Verify that the score is increased.
5. Verify that all bricks above the removed row are moved downwards.

#19 Finishing a game session
Requirement: When a piece gets fixated above the top boundary, the game session shall be terminated and the total score presented to the user. In a multiplayer game session, all participating players shall be presented with the total score.
Input: Controller input
Output: Visible effect; the game over screen is shown
Step-by-step procedure:

1. Select “Host game” from the main menu.
2. Enter name and number of players as appropriate. Do not check the “Private game” checkbox.
3. Start the game.
4. Have somebody launch the client, select “Join game” from the main menu, find the game in the game list, and enter it.
5. Launch the multiplayer game session.
6. Press the “Drop” controller key to fixate each piece shown until a piece gets fixated above the top boundary.
7. Verify that the game session is terminated.
8. Verify that the total score is shown to both participating players.

6.1.5. Powerups

#20 Obtain powerup
Requirement: The user shall be able to obtain powerups contained in completed rows.
Input: Controller input
Output: Visible effect; the obtained powerup is shown in the powerup list
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Move pieces so that a complete row is formed, with at least one brick still on the board. One brick will then be changed into a powerup brick.
3. Move pieces so that the row containing the powerup is completed.
4. Verify that the powerup is shown in the powerup list.

#21 Create powerup
Requirement: When a row is removed, a random brick on the game board shall be replaced with a powerup brick. If there are no bricks on the game board, nothing happens.
Input: Controller input
Output: Visible effect; one brick is changed into a powerup.
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Move pieces so that a complete row is formed, with at least one brick still on the board.
3. Verify that one brick is changed into a powerup brick.

#22 Use powerup
Requirement: At any time during a game session, the player shall be able to activate a powerup that he has obtained. This will affect gameplay or the game board. A list of powerups with corresponding effects is given in Table 1.
Input: Controller input
Output: Visible effect, as described in table 1 below.
Step-by-step procedure:

1. Select “Singleplayer” from the main menu.
2. Move pieces so that a complete row is formed, with at least one brick still on the board. One brick will then be changed into a powerup brick.
3. Move pieces so that the row containing the powerup is completed. This powerup will be shown in the powerup list.
4. Press the corresponding “Use powerup” controller key to use the powerup.
5. Verify that the powerup affects the game according to Table 1.

Table 1.

Powerup	**Effect**
Clear row	Removes all bricks in the bottom row from the game board.
Clear partial row	Removes all bricks in a given part of the bottom row from the game board.
Clear field	Removes all bricks on the game board.
Mirror flip	Allows the player to flip pieces horizontally.
Gravity	Pulls all bricks on the board as far down as possible, which removes any gaps.
No gravity	Removes gravity for the current piece, removing its falling speed.
Foresight	Shows upcoming pieces to the player.

7. References

(1) Tetris – http://en.wikipedia.org/wiki/Tetris
(2) The Java Runtime Environment, JRE – http://java.sun.com/
(3) OpenGL – http://www.opengl.org/
(4) Lightweight Java Game Library – http://www.lwjgl.org/ (a Java library for making games)