Spectrangle in Java

F.P.J. Nijweide N.T. Shenkute

February 4, 2019

1 Introduction

The goal of this report is to describe an implementation of a Board game in Java. The board game discussed in this report is Spectrangle, a triangular tile based abstract strategy game. The game can have between 1 and 4 players and it takes a playing time of about 45 minutes. The triangular tiles, also called trangs, come double sided with a permutation of colors consisting red, yellow, green, cyan, and magenta. In addition, the tiles consist of a Joker with white colors on all its sides. Furthermore, each tile comes with a number that indicates its value with respect to other tiles. The board itself contains marked locations that will grant a player bonus point. The game consists of 36 tiles, a board and a scoring mechanism for each player.

This implementation was made with a server-client design. The communication was handled via BufferedReaders (and writers) over sockets. This allows for multiplayer gameplay across networks, but adds several difficulties such as the need for multithreading, and lots of exception handling. In implementing this we chose to use the Model-View-Controller pattern. We seperated the areas of the game that handle different tasks into seperate packages, which can be seen in the next section. Next to the basic functionality, we also implemented a chat extension.

While this report presents reasoning behind general design decisions, detailed documentation for all the classes, fields and methods can be found in the Javadoc files of our project. We encourage the reader to take a look at it, as methods are described in much more detail there.

2 Model-View-Controller Pattern (MVC)

Next to the MVC pattern, there are two files: "ClientMain.java" and "ServerMain.java" in the root source directory. These classes are used to start the game and do not contain much code themselves.

2.1 Model package

In this section the classes that model the game and the game logic are discussed.

1 `Board.java`

Board.java is the main class that represents the backbone of the Spectrangle game model. It is responsible for representing a Spectrangle board, the rules regarding to the different locations within the board, and the rules regarding to the validity of possible moves and the points attributed to those moves. When initialized, the class requires no fields and it will initialize an instance of a Spectrangle board of type Board containing the following fields:

- `DIM`: a field of type `final int`, which represents the dimension of the board (which in this case is 36
- `BoardLocation[]`: a list of `BoardLocation` instances (see subsubsection 3)

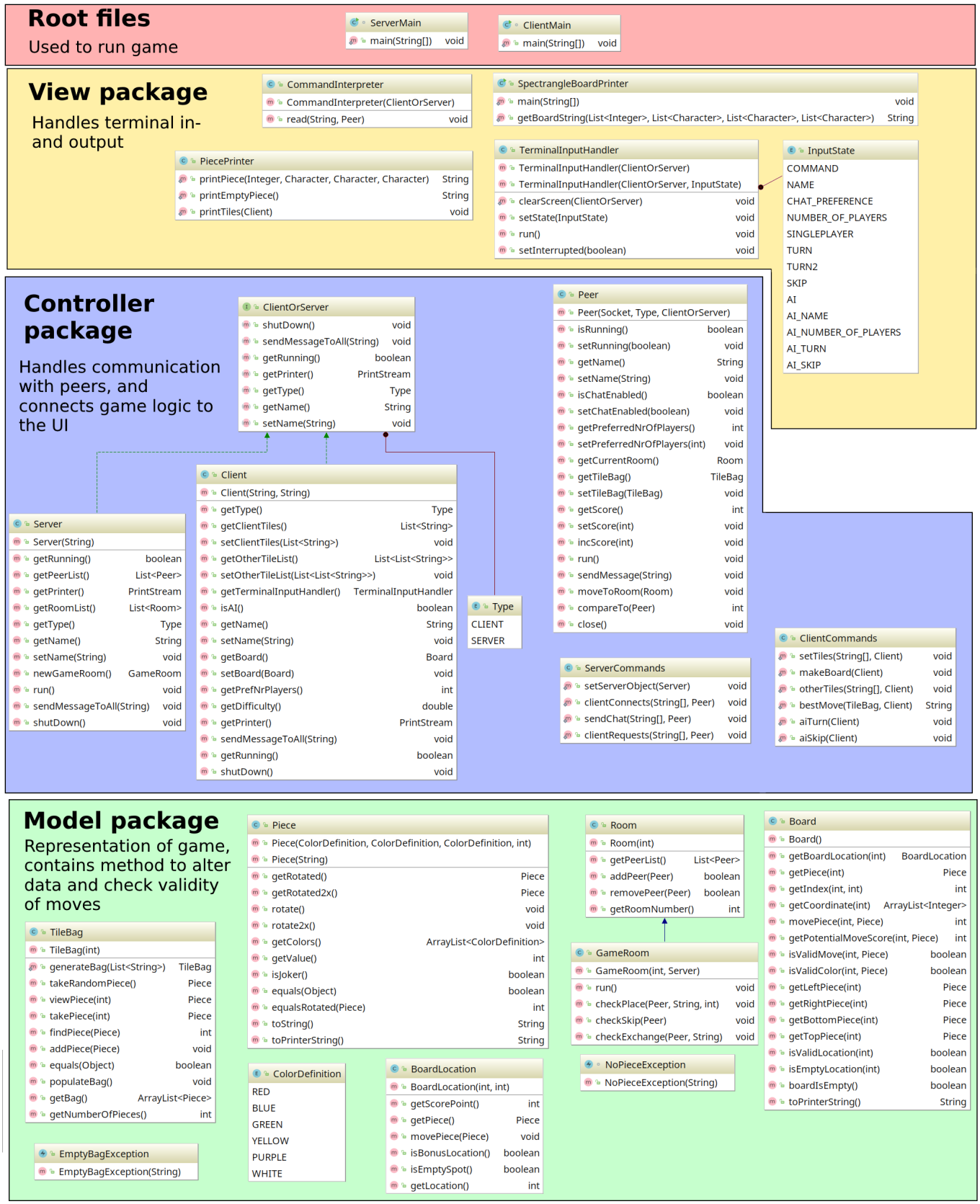


Figure 1: *An outline of the design and all the classes for this project*

This class assigns an appropriate values to each of the `BoardLocation` instances based on the rules of the game, for example, at location 10 a bonus value of 2 is assigned to the `BoardLocation`. This class implements all the game rules of Spectrangle. The most important methods of `Board.java` are summarized in Table 1

Table 1: *Functions within `Board.java`*

Method	Return Type	Usage
`getBoardLocation(int location)`	`BoardLocation`	Get a BoardLocation object
`getPiece(int location)`	`Piece`	Get the piece object that lies on the given location
`getIndex(int r, int c)`	`int`	Return the index of the given location based on the coordinates given
`getCoordinate(int index)`	`ArrayList<Integer>`	Return the coordinates of a given index (conversion)
`isValidMove(int location, Piece piece)`	`boolean`	Checks the validity of the move based on Spectrangle rules
`isValidColor(int location, Piece piece)`	`boolean`	Checks color validity of a given move
`movePiece(int location, Piece piece)`	`int`	Moves the given piece (if possible) and returns the number of points gained
`getLeftPiece(int index)`[1]	`Piece`	Returns the piece object that lies on the left of the given index
`isValidLocation(int index)`	`boolean`	Checks if the given index is valid
`isEmptyLocation(int index)`	`boolean`	Checks if the given index is empty
`boardIsEmpty()`	`boolean`	Checks if this Board is empty
`toPrinterString()`	`String`	Prints the board appropriately

2 `Piece.java` (Tile)

This class represents a `Piece` object that contains information regarding the piece (tile) according to the rules of Spectrangle. The fields of this class are shown below:

- `value`: Value of this `Piece` object of type `int`
- `left`: a `ColorDefinition`[2] object representing the color on the left side of this `Piece` object.

[2] An enum class that defines the colors of a Spectrangle game (red, blue, green, yellow, purple, and white)

This class is integrated with the main class, `Board.java`, to represent the tiles as the game continues. The colors of a tile are defined by using an `enum` class containing all the colors of a Spectrangle game. The main methods of this class are summarized below in Table 2:

Table 2: *Functions within `Piece.java`*

Method	Return Type	Usage
`getRotated()`[3]	`Piece`	Returns a Piece object after rotating this Piece
`rotate()`[4]	`void`	Rotates this piece
`getColors()`	`ArrayList<ColorDefinition>`	Returns the orientation of this Piece, which is stored in a form of an array
`getValue()`	`int`	Returns the value of the piece
`isJoker()`	`boolean`	Checks if this piece is jocker
`equals(Object obj)`[5]	`boolean`	Checks if the given object is equivalent to this piece
`toString()`	`String`	Returns a String representation of the piece
`String toPrinterString()`	`String`	Prints the piece

3 `BoardLocation.java`

The `BoardLocation.java` class represents an instance of an object that contains information about the location, points available on that location and the piece on that location. This class has the following fields:

- `location`: Location of type `int`
- `scorePoint`: Number of points available on this location type `int`
- `piece`: A piece of type `Piece`, if it exists on this location

When initialized this class will contain information about an index location, on a Spectrangle board, the main methods and their use are shown below in Table 3:

Table 3: *Functions within* `BoardLocation.java`

Method	Return Type	Usage
`getScorePoint()`	`int`	Returns the amounts of bonus points this class has
`getPiece()`	`Piece`	Returns the Piece object that lies on this class
`movePiece(Piece p)`	`void`	Moves the given Piece object to this class
`isBonusLocation()`	`boolean`	Validates if this class has bonus points
`isEmptySpot()`	`boolean`	Checks if this class does not contain a Piece object
`getLocation()`	`int`	Returns the location index of this class

4 `TileBag.java`

This class is responsible for generating an instance of a tile bag consisting of 36 tiles according to the rules of Spectrangle. It only contains one field:

- `pieces`: An `ArrayList<Piece>` object consisting of 36 tiles (pieces)

This class populates the `ArrayList<Piece>` object with 36 tiles and their corresponding points. The main functions within the class are summarized below in Table 4:

Table 4: *Functions within* `TileBag.java`

Method	Return Type	Usage
`takeRandomPiece()`	`Piece`	Returns a random piece object out of the tile bag
`takePiece(int i)`[6]	`Piece`	Returns the piece object at the given index and removes it from the tile bag
`findPiece(Piece inputPiece)`	`int`	Returns the index (if found) of the given piece
`addPiece(Piece p)`	`void`	Adds the piece object to the tile bag
`populateBag()`	`void`	Populates the tile bag with 36 pieces
`getBag()`	`ArrayList<Piece>`	Returns the tile bag
`getNumberOfPieces()`	`int`	Returns the size of the tile bag

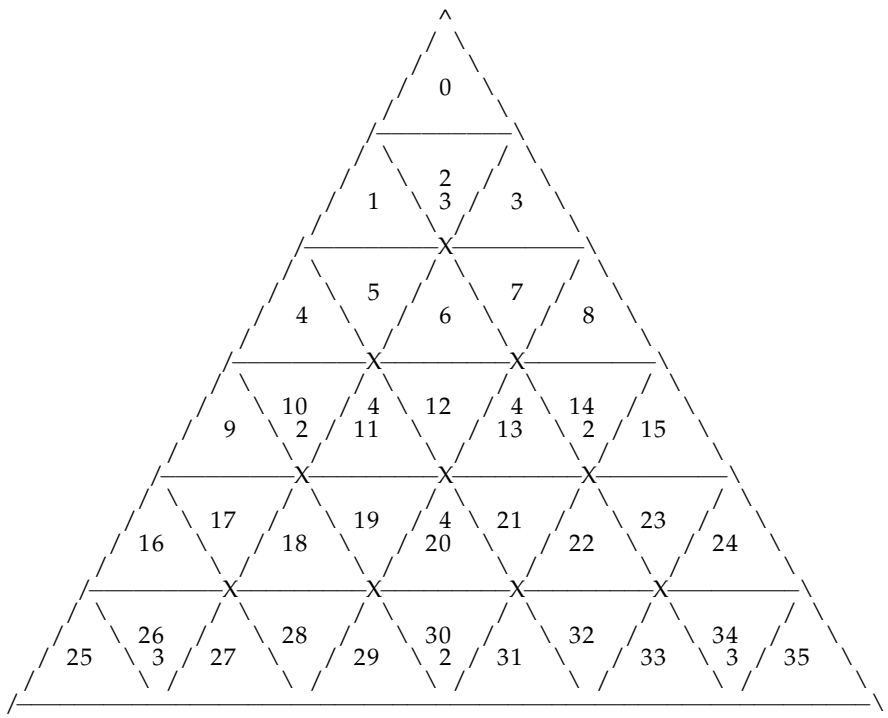
5 `ColorDefinition`

This class simply defines the different colors involved in Spectrangle pieces. The colors defined are as follows:

- `RED, BLUE, GREEN, YELLOW, PURPLE, WHITE`

2.2 View package

The view package consists of all the elements that are necessary in order to print the different aspects of the game to the console. In this project a collection of `ASCII` characters are used to print the board and the pieces to the console, this is shown below:



A class with methods to do this was provided to us by the TA's on Canvas: SpectrangleBoardPrinter. Its functionality has been left unchanged.

1 `PiecePrinter.java`

We created an adapted version of SpectrangleBoardPrinter, PiecePrinter, that prints a single piece. This class also has a method printTiles to print 4 pieces next to each other. It does not contain any fields.

Table 5: *Functions within* `PiecePrinter.java`

Method	Return Type	Usage
`printPiece(Integer value, Character verticalIn, Character leftIn, Character rightIn)`	`String`	Returns ASCII art of piece
`printPiece(String s)`	`String`	Returns ASCII art of piece
`printEmptyPiece())`	`String`	Returns empty lines with sizeof piece ASCII art
`printTiles(Client clientObject))`	`void`	Prints 4 pieces in the "hands" of clientObject to terminal

2 `TerminalInputHandler.java`

The class TerminalInputHandler implements Runnable, thus a Thread can be started on it. In its run method, it checks for input from the terminal. What it does with this input is determined by its field "state", which is of the enum InputState. Its only other important field, a boolean "interrupted" is constantly polled to check if the handler should stop checking for this kind of input and switch to another task.

Table 6: *Functions within* `TerminalInputHandler.java`

Method	Return Type	Usage
`clearScreen(ClientOrServer parent)`	`void`	Clears screen
`readString()`	`String`	Attempts to read a string from the terminal.
`setState(InputState state)`	`void`	Setter for state field
`run()`	`void`	Tries to check for input according to current value of state
`setInterrupted(boolean interrupted)`	`void`	Setter for interrupted field

It also contains the enum InputState which contains the following states:

- `COMMAND, NAME, CHAT_PREFERENCE, NUMBER_OF_PLAYERS, SINGLEPLAYER, TURN, TURN2, SKIP, AI, AI_NAME, AI_NUMBER_OF_PLAYERS, AI_TURN, AI_SKIP`

3 `CommandInterpreter.java`

The class CommandInterpreter is used to start other methods, according to the commands received in networking communication. It uses switches to identify the command, and starts other methods (usually in ClientCommands and ServerCommands). It has two fields:

- `parent`: a ClientOrServer representing the "parent" class
- `parentType` a clientOrServer.Type of the parent

It has only one method:

Table 7: *Functions within `CommandInterpreter.java`*

Method	Return Type	Usage
`read(String inputString, Peer peer)`	`void`	Reads line that came from a certain peer and runs other methods depending on what its command is

2.3 Controller package

The classes contained in this package are less easy to explain with parameters and return types. Most methods here return void, and print lines to the terminal or to connected sockets. The parameters they take are often commands and arguments in the form of: a string, an array of strings, or a list of strings. Often, the ClientOrServer "parent" object is passed along with these arguments.

1 `ClientOrServer.java`

The interface ClientOrServer is used for representing the "parent", running all the communication to other peers, and executing game logic while doing so.

It has several methods:

Table 8: *Functions within `ClientOrServer.java`*

Method	Return Type	Usage
`shutDown()`	`void`	Shut down program, and all communication
`sendMessageToAll(String s)`	`void`	Send a line to all connected peers
`getRunning()`	`boolean`	Check if this is running or in the state of shutting down
`getPrinter()`	`PrintStream`	Returns the PrintStrean used for terminal output of the program
`getType()`	`Type`	Returns the type of this: CLIENT or SERVER.
`getName()`	`String`	Getter for name field
`setName(String s)`	`void`	Setter for name field

It also contains the enum Type which contains the following states:

- `CLIENT, SERVER`

Both controller.server.Server and controller.client.Client implement the aforementioned methods. Client also has a "connect" method which connects to a Socket of an IP, while Server implements Runnable, allowing for a Thread to run that constantly accepts connections to the ServerSocket, creating new Sockets.

2 Peer.java

The Peer class represents the other side of the connection. Thus, each ClientOrServer can have one or more Peers connected to it. Each peer implements Runnable, so a thread can be started on it. Its run method tries to read a line from a BufferedReader of the socket. Because the Peers need to be ordered during gameplay, we implemented Comparable as well.

This class contains many fields for storing preferences (chat yes/no), game logic (peer's score), and general connection properties(socket, name). Each of these fields also has setters and getters where needed. These are all described adequately in the Javadoc. The four interesting (not getter or setter) methods are described below:

Table 9: *Functions within* `Peer.java`

Method	Return Type	Usage
`compareTo(Peer p)`	`int`	Compares one peer's name to another peer's name
`moveToRoom(Room room)`	`void`	If peer is currently in a room, try to remove it from that room's list. Then, place the peer in the new room
`sendMessage(String s)`	`void`	Send a string to this peer over the network
`run()`	`void`	Tries to read a line from BufferedReader of socket
`close()`	`void`	Closes socket, stops any thread running on this peer

3 ServerCommands and ClientCommands

ServerCommands.java and ClientCommands.java contain no fields. They only contain static methods which were removed from CommandInterpreter because the switch statement there was getting too large.

ClientCommands contains:

Table 10: *Functions within* `ClientCommands`

Method	Return Type	Usage
`setTiles(String[] args, CLient clientObject)`	`void`	Takes the arguments from the "tiles" command as specified in the protocol, and sets the tiles field of the Client instance to contain these tiles
`makeBoard(Client clientObject)`	`void`	Set the Client instance's Board instance to a new, empty Board instance.
`otherTiles(String[] args, CLient clientObject)`	`void`	Like setTiles, but for enemies' tiles
`bestMove(TileBag tileBag, Client clientObject)`	`String`	Determines the best move for the client and returns it as a string, i.e. "place RRR6 on 2"
`randomMove(TileBag tileBag, Client clientObject)`	`String`	Determines a random move for the client and returns it as a string, i.e. "place RRR6 on 2"
`aiTurn()`	`void`	Depending on difficulty setting, does a random or best move
`aiSkip()`	`void`	Generates random numbers to determine whether to skip or exchange. Takes amount of tiles left in tile bags into account, and if empty, skips

ServerCommands contains:

Table 11: *Functions within* `ClientCommands`

Method	Return Type	Usage
`clientConnects(String[] args, Peer peer)`	`void`	Takes the arguments from the "connect" command as specified in the protocol, and sets the client's name and chat preference to the specified parameters
`clientRequests(String[] args, Peer peer)`	`void`	Takes the arguments from the "request" command as specified in the protocol, and tries to find a game of specified size for the player
`createGame(List<Peer> peerList)`	`void`	Creates a GameRoom instance for this list of players
`sendChat(String[] args, Peer peer)`	`void`	Sends chat from this peer to all players in the same room who want to receive it
`sendWaiting(Peer peer)`	`void`	Send the "waiting" command as specified in the protocol
`setServerObject(Server server)`	`void`	Setter for the serverObject field

4 `Room.java`

The Room class is nothing but a fancy, encapsulated list. It is used for creating chat rooms, and allows for the use of methods such as sendMessageToRoom (in Server). It has two fields:

- `roomNumber`: an `int`. The number of the room. Is ensured to be unique by always setting it equal to the number of rooms created thus far
- `peerList`: a `List<Peer>`. List of peers in the room

All the methods are nothing more than getters and setters (except for removePeer(Peer p), which calls peerList.remove(p)). These are described in the JavaDoc for this class.

5 `GameRoom.java`

GameRoom extends Room with functionality for playing games. This is where the game logic defined in the model package, and the UI defined in the view package comes together. When enough players are found that are waiting for a game of a certain size, they are moved to this room. This class implement Runnable, so that a thread can be started on it which handles the game logic.

The room is constructed with an instance of Board and TileBag. The TileBag of size 36 is populated with the tiles of the game. Then, the players have tiles distributed, from this bag, to their own (new) tileBags. As specified in the rules of the game, the highest tile drawn (out of all the players' first tiles) is then used to determine the order of turns for the players. Then, until the game is finished, the game keeps starting new turns, going down this list and wrapping back around to the top.

At the start of each turn, all players are sent the "tiles" command as specified in the protocol. The thread then waits until another thread updates the waitingForMove boolean field, meaning a successful move has been executed. In the meantime, the Thread keeps checking if any of the peers disconnect, and if this happens, it shuts down the room and sends the players back to the lobby.

The game ends if no players have viable moves. Whether a move is viable is checked using methods from the model package.

A detailed description of the 15(!) methods of this class, with parameters, return types, and more can be found in the Javadoc.

3 JUNIT Tests

In order to validate the implementation of the Spectrangle game, several `JUNIT` test cases were devised. The tests focused on the model part of the game, meaning that the following classes were tested: Board, Piece, BoardLocation and TileBag. These can easily be tested as most methods have return types which are easy to check. This is in contrast to the server/client connection classes, which are hardly testable. Most methods do not return anything, and only send strings to the terminal, or to the socket. Establishing a working connection with a different computer is not something that can be tested using simple JUNIT tests, and this requires more thorough verification. In the end, through much debugging, we have verified that the program works as intended.

The tests were implemented in such a way that maximized the coverage of the classes in the model package. A coverage of 60% was reached eventually, the individual coverages are shown below in Table 12:

Table 12: *Junit Tests and Coverages*

Test	**Class Tested**	**Coverage (%)**
`BoardTest.java`	`Board.java`	93.5
`PieceTest.java`	`Piece.java`	96.9
`TileBagTest.java`	`TileBag.java`	97.4
`BoardLocationTest.java`	`BoardLocation.java`	100

These tests compare the return values of each of the methods in the Model package (as specified in section 2.1 of this report) to expected values. Writing these tests was quite difficult, as we had to abide by the game rules while also looking for edge cases in our game logic.

The coverages are not all at 100% because it is very difficult to cover certain possibilities that might occur during the game. All tests pass. Thus, our implementation of the game rules seems to be correct.

For these tests, several methods in the classes of the model package had to be made public. These were previously private or package-private. While this leads to less encapsulation, we believe that having these tests run if more important than lowering the visibility of the methods. In the end, we made sure that every field in every class is private, except for static final ones.

Several JML software contracts are defined for the TileBag, Piece and Board methods. These classes were chosen for the same reasons as mentioned above. It is quit easy to check their return values against previous circumstances. To fullfill the preconditions and postconditions of all these contracts, Boards, TileBags and

Pieces must always be generated using their normal constructors. Their properly encapsulated methods should be used to get and set any field values, because these methods ensure game rules are always being followed. Settings values of fields without these methods does not guarantee that the rules are being followed, and might lead to a breach of these conditions.

4 Exceptions

In order to handle erroneous situations, two exception classes were devised to handle problems in the model package. The exception classes and their purpose is summarized below:

- `EmptyBagException`: This exception class handles the situation when the tile bag runs out of tiles
- `NoPieceException`: This exception class handles the situation when no piece is found (in several classes)

5 Conclusion

In the end, our game seems to work almost perfectly. However, there is always room for improvement. In some cases, when the user is asked for input, it is possible to make the game crash by entering it in a completely wrong format (strings where integers are expected, numbers where IP addresses are expected etc.).

In production-level code for companies, all these edge cases would have to be considered, and they would have to be well-guarded to prevent any breaches of data integrity from happening. We are well aware of the dangers involved in not thoroughly checking user input as we both have some experience in C, but it did not seem very relevant to the goal of the project to spend lots of time on this. We tried to prevent most cases of wrong input from breaking the game, but it is still possible to break the game in some instances where user input is asked.

Another feature that we decided to compromise on was the AI scaling. Because the AI calculations are incredibly fast, there is no room for thing such as "stopping the calculation after X seconds". The calculation is always done before the user can even respond. Thus, stopping AI calculations after a certain amount of seconds is not an option right now. We chose to implement an AI "difficulty" where 0 is an AI which chooses random valid moves, and 1 chooses the moves with the highest point rewards.

Ways of improving the AI scaling would be:

- Allowing the AI to pick the second-best, third-best moves, and so on. This would allow for AIs which are smarter than random, but dumber than "best move". Still, this would not allow for long calculations, as this would also finish instantly
- Predicting future moves. We considered this, as we have needed to do this for other game implementations in TCS before, such as tic-tac-toe. However, it would be quite complicated in this situation, as in a game of 4, there are 3 turns between the player's current turn and the next. In those 3 turns, each player can place any of their 4 tiles, in any manner of rotation, on any of the open locations on the board. A quick back of the envelope calculation shows that:

  $$35 \cdot 34 \cdot 33 \cdot 4^3 \cdot 3^3 = 67858560 \text{ possibilities} \quad (1)$$

  Thus, when starting in a game of 4, the player's next turn could be in any of 67+ million board states. Due to the unpredictability of human players in this game, trying to predict the board states with to the assumption that each player does their best move will also not work. There are multiple possible "best moves" for each player, and sometimes they make stupid decisions. Thus, trying to predict the future is very unlikely to work, and the best course of action is probably to pick the move that currently gives the highest amount of points.

Some other points:

- We did not consciously implement the observer pattern, as it seemed to conflict with the way we designed our MVC pattern. However, one could see the communication with the Peer object over the network (through TerminalInputHandlers and CommandInterpreters) as implementing this. The GameRoom is the Observer, and it notifies all peers when anything changes. The Peers can change properties of the GameRoom by sending commands when it is their turn.
- We barely implemented any synchronization. In our project it never happens that two threads access the same method at once, as they all act on completely different objects. There are threads for network communication with peers, threads for game logic handling, threads for terminal input, and threads for accepting new peers. None of these interact.
- Sometimes fields were implemented as volatile to ensure they were not cached, if they were polled often and needed to be accurate (usually, interrupt or isInterrupted booleans). We are not sure if this really changed any behavior, but we implemented this nonetheless.