Team 23

COMP 3004 Term Project

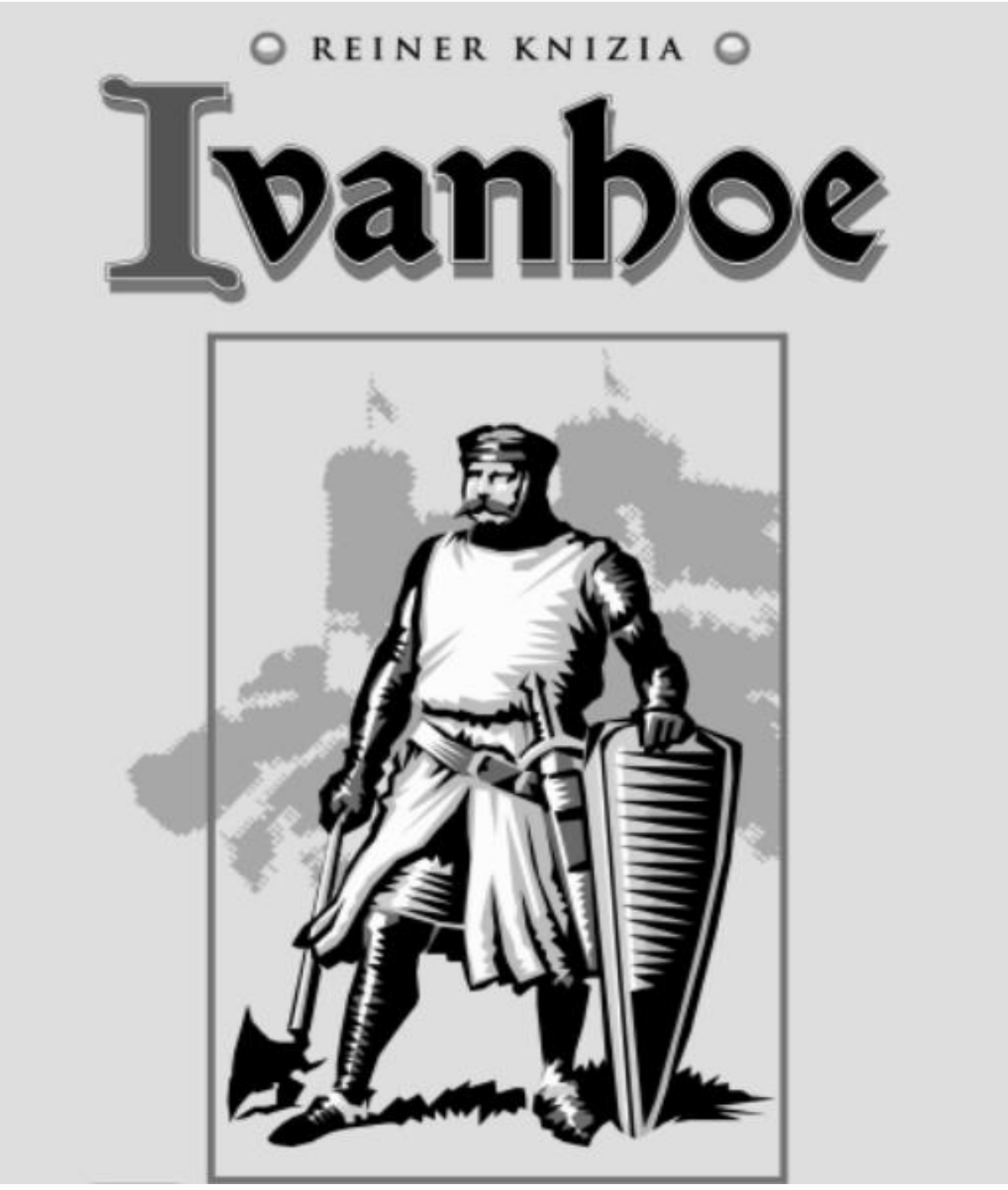


Table Of Contents:

Section 1

Introduction:

The following documentation outlines a networked rendition of Ivanhoe, a card-based board game created by Reiner Knizia. This version of the game was created by Team 23 of Carleton University's COMP3004 course in the winter semester of 2016. The objective of the game is to obtain all five colored tokens by winning tournaments (rounds). Ivanhoe can be played by 2-5 players.

1.1 Overview

The software designed in order to implement this game was written in Java using the Eclipse IDE. The main components of the software can be divided into three major categories; Networking, Logic, and User Interface. The networking utilizes a server-client architecture allowing for information regarding the game to be sent and received by the multiple players. The game Logic was created using object-oriented design strategies with a strong focus on test-driven development. A online BitBucket repository named "TermProject" under the username "kevin manton", was created, and can be accessed (by invite only) in order to view the development of this software from beginning to end. A graphical user interface allows players to visually engage in the game.

1.2 Team 23

Name/ Student Number	**Main Contributions**
Kevin Manton 100853058	Game Logic
Harth Majeed 100896761	User Interface
Ali Hamed 100771513	Networking

Section 2

2.1 Game Rules

(R1) The number of players can range from 2 to 5.

(R2) The first player to obtain 4 or 5 colored tokens wins the game.

-4 tokens needed to win if there are 4 or 5 players.

-5 tokens needed to win if there are 2 or 3 players.

(R3) There are 110 cards in the deck.

-70 color cards.

-14 Purple, 14 Red, 14 Blue, 14 Yellow, and 14 Green

-20 supporter cards.

-16 squires, 4 maidens.

-20 action cards.

(R4) Player who starts tournament chooses tournament color.

-Purple, Red, Blue, Yellow, or Green

(R5) Player who wins tournament receives token of the final tournament color.

-if final tournament color is Purple, player may choose any color token

(R6) When the game starts each player is initially dealt 8 cards from the deck.

(R7) The first player to connect to the server starts the first tournament.

(R8) After the first tournament, the winner of the last tournament starts the next tournament.

(R9) When a player ends their turn, the next players turn is the next player who has not forfeited.

(R10) At the beginning of a players turn, they always draw a card.

(R11) A player may only end their turn by having the highest played value or forfeiting.

(R12) A player may play as many cards as they wants during their turn.

-if a player is stunned they can only add one card to their display.

(R13) A player may only play cards that are valid in the tournament

-action cards are valid in any tournament if they will have an effect

-supporter cards are valid in any tournament.

-each player may only have one maiden in their display at any given time.

-cards of the same color as the current tournament color are valid.

(R14) If the deck becomes empty, the discard pile is shuffled and replaces the deck.

(R15) When choosing a tournament color, the player must have a card of that color in his hand.

-if a player has a supporter card in his hand he may choose any color.

-if the final color of the last tournament was purple, the player must choose a different color than purple for the new tournament.

-if the final color of the last tournament was purple, if a player only has purple or action

cards In his hand, he cannot start a tournament. The turn is passed to the next player.

(R16) A player must play at least one card during his turn to remain in a tournament.

(R17) Color cards and supporter cards are placed face up in the players display when played.

(R18) Action cards are discarded once played

-shield and stunned are not discarded, they are played face up in front of the player that they will affect, separate from the display.

(R19) If a player forfeits from a tournament with a maiden in their display, they lose a token.

(R20) In a green tournament all cards in display are treated as having a value of 1.

(R21) if a player forfeits from a tournament he does not get any more turns in that tournament.

(R22) Ivanhoe the Action card can be played by a player when it is not their turn.

2.1 Action Cards: (directly from manual)

UNHORSE: The tournament color changes from purple to red, blue or yellow, as announced by the player.

CHANGE WEAPON: The tournament color changes from red, blue or yellow to a different one of these colors, as announced by the player.

DROP WEAPON: The tournament color changes from red, blue or yellow to green.

BREAK LANCE: Force one opponent to discard all purple cards from his display.

RIPOSTE: Take the last card played on any one opponent's display and add it to your own display.

DODGE: Discard any one card from any one opponent's display.

RETREAT: Take any one card from your own display back into your hand.

KNOCK DOWN: Draw at random one card from any one opponent's hand and add it to your hand, without revealing the card to other opponents.

The following action cards may affect more than one player.

OUTMANEUVER: All opponents must remove the last card played on their displays.

CHARGE: Identify the lowest value card throughout all displays. All players must discard all cards of this value from their displays.

COUNTERCHARGE: Identify the highest value card throughout all displays. All players must discard all cards of this value from their displays.

DISGRACE: Each player must remove all his supporters from his display.

ADAPT: Each player may only keep one card of each value in his display. All other cards

with the same value are discarded. Each player decides which of the matching-value cards he will discard.

OUTWIT: Place one of your faceup cards in front of an opponent, and take one faceup card from this opponent and place it face up in front of yourself. This may include the SHIELD and STUNNED cards.

SHIELD: A player plays this card face up in front of himself, but separate from his display. As long as a player has the SHIELD card in front of him, all action cards have no effect on his display.

STUNNED: Place this card separately face up in front of any one opponent. As long as a player has the STUNNED card in front of him, he may add only one new card to his display each turn.

IVANHOE: This is the only card a player can play outside of his turn. A player can play it at any time as long as he is still in the tournament. Use this card to cancel all effects of any one action card just played.

2.2 Card Distribution (directly from manual)

CARD DISTRIBUTION

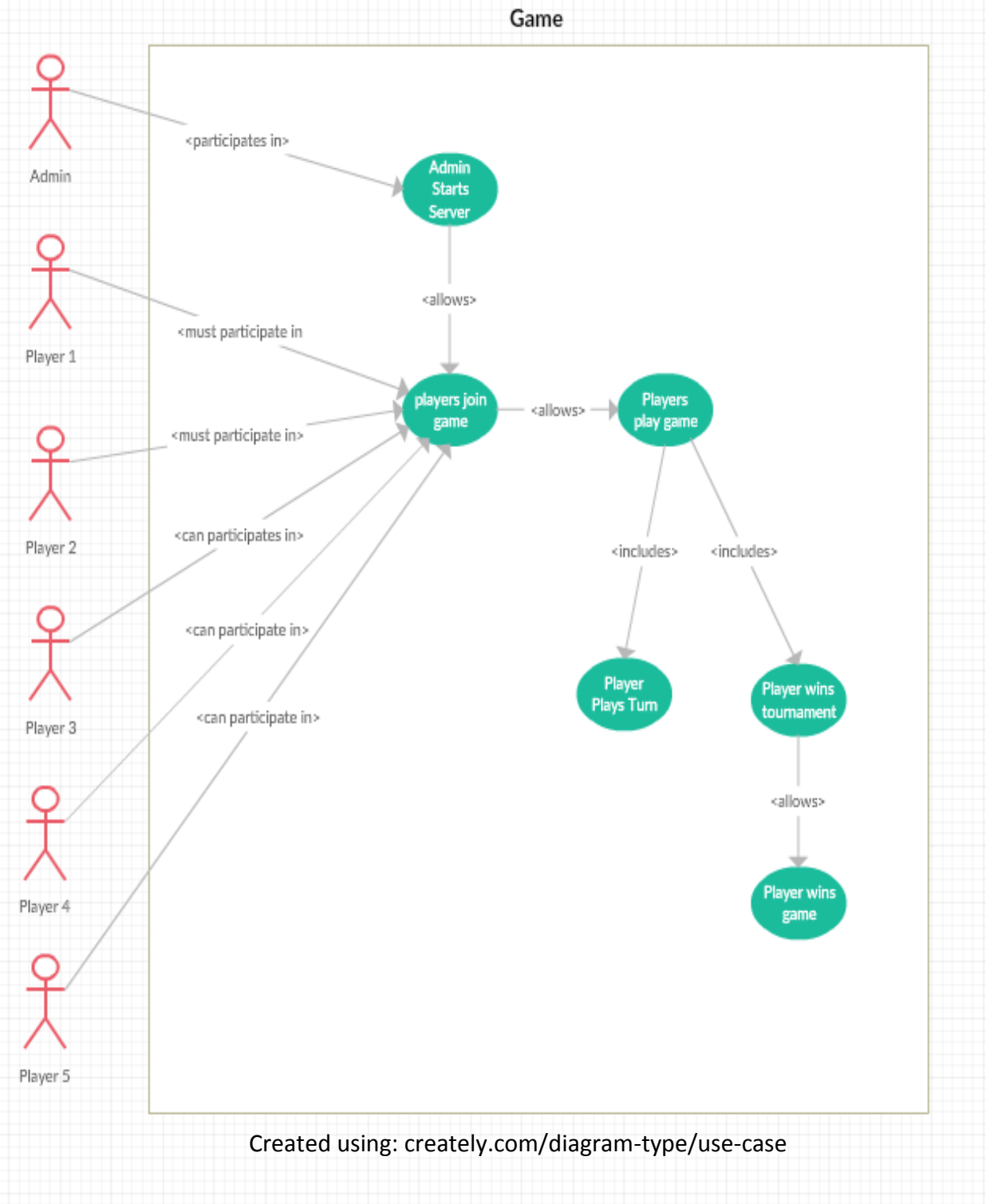
	Values:						
70 Color Cards	**1**	**2**	**3**	**4**	**5**	**6**	**7**
14 Purple (Jousting)	-	-	4	4	4	-	2
14 Red (Sword)	-	-	6	6	2	-	-
14 Blue (Axe)	-	4	4	4	2	-	-
14 Yellow (Morningstar)	-	4	8	2	-	-	-
14 Green (No Weapon)	14	-	-	-	-	-	-
20 Supporters (White)	**1**	**2**	**3**	**4**	**5**	**6**	**7**
16 Squires	-	8	8	-	-	-	-
4 Maidens	-	-	-	-	-	4	-

20 Action Cards

Change Colors	Special	Affect Displays	1 Retreat	1 Countercharge
1 Unhorse	1 Shield	1 Break Lance	2 Knock Down	1 Disgrace
1 Change Weapon	1 Stunned	3 Riposte	1 Outmaneuver	1 Adapt
1 Drop Weapon	1 Ivanhoe	1 Dodge	1 Charge	1 Outwit

Section 3

3.1 Use Case Diagram



Created using: creately.com/diagram-type/use-case

3.2 Use Cases

UC-01	**Admin Starts Server**
Description: This use case describes how an administrator would set up the game server in order to allow for a certain number of players to connect to the server in order to play a game of Ivanhoe.	
External Actors: - Administrator.	
Trigger: Administrator launches server jar file.	
Pre-Condition: none.	
Sequence: **1.)** Administrator launches jar file **2.)** Administrator starts server **3.)** Administrator is prompted for number of players **4.)** Administrator enters number of players (2-5)	
Post Condition: Game server is running.	
Resulting Event: Server waits for specified number of clients to connect.	
Alternatives: none.	
Traceability: R1	

UC-02	**Players Join Game**
Description: This use case describes how a player joins a game of Ivanhoe.	
External Actors: -Player.	
Trigger: Player launches the client jar file.	
Pre-Condition: The game server is running and awaiting player connection.	
Sequence: **1.)** Player launches the jar file. **2.)** GUI pops up.	

3.) Player is prompted to enter name. **4.)** Player enters name.
Post Condition: Player is connected to the game server.
Resulting Event: Player waits for remaining players to connect to server.
Alternatives: -If number of players specified by the server is reached when player connects, the game starts. -If the game is already started, the player connection is refused.
Traceability: R6

UC-03	**Players Play Game**
Description: This use case describes how players play Ivanhoe.	
External Actors: 2, 3, 4, or 5 players	
Trigger: Final player connects to server.	
Pre-Condition: 2, 3, 4, or 5 players are connected to game server.	
Sequence: **1.)** Each player is dealt 8 cards from the deck. **2.)** The first player that connected to the server (player 1) draws one more card. **3.)** Player 1 chooses a tournament color. **4.)** Player 1 plays his turn. **5.)** Other players play their turns in the order that they connected to the server. **6.)** Players continue playing turns until one player wins tournament. **7.)** The last tournament winner chooses the next tournament color. **8.)** Players play their turns in order starting from the player who started the tournament. **9.)** Players continue playing turns until one player wins tournament. **10.)** Repeat: steps 7 through 9 until one player wins game.	

Post Condition: One player is declared winner.
Resulting Event: Game ends.
Alternatives: -If a player disconnects before game is over, game ends, no winner declared.
Traceability: R4, R6, R7, R8, R15, R21.

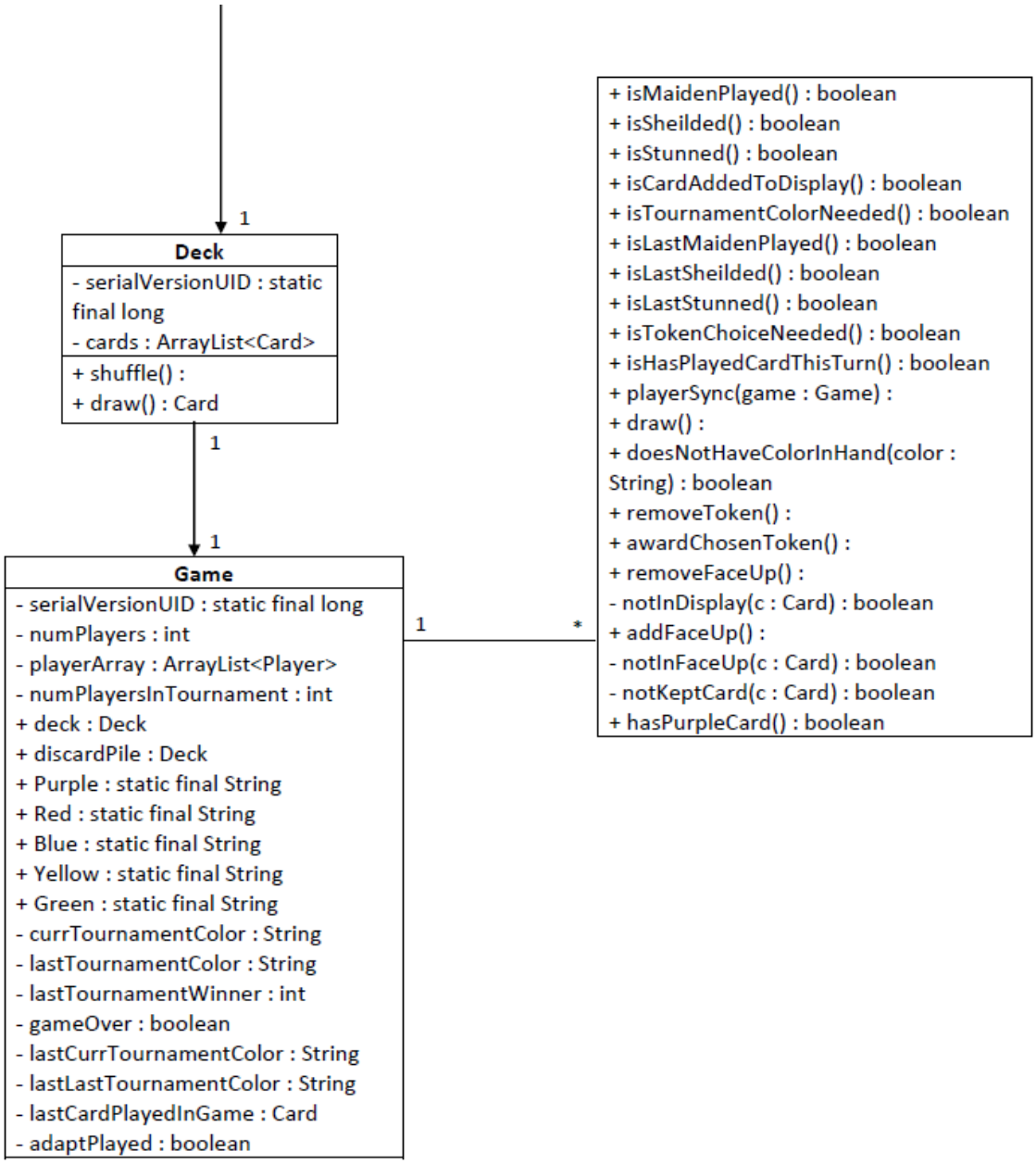
UC-04	**Player Plays Turn**
Description: This use case describes how a player plays a turn in Ivanhoe.	
External Actors: 2, 3, 4, or 5 players.	
Trigger: Player just started a tournament, or previous player in tournament just ended turn, or forfeited.	
Pre-Condition: Player has not yet forfeited from current tournament, and game is not over.	
Sequence: **1.)** Player draws a card from the deck to his hand. **2.)** Player plays cards from his hand. - Color cards add to played value and are placed face up in display. - Supporter cards add to played value and are placed face up in display. - Action cards take effect and are placed in directly in discard pile. - Shield / Stunned are placed face up in front of the affected player separate from display. **3.)** Player ends turn or forfeits from current tournament.	
Post Condition: Player has played highest value or has forfeited from current tournament.	
Resulting Event: Next player in tournament gets to start turn.	
Alternatives: If a player is stunned he may only add one card to his display during his turn.	
Traceability: R3, R9, R10, R11, R12, R13, R16, R17, R18, R19	

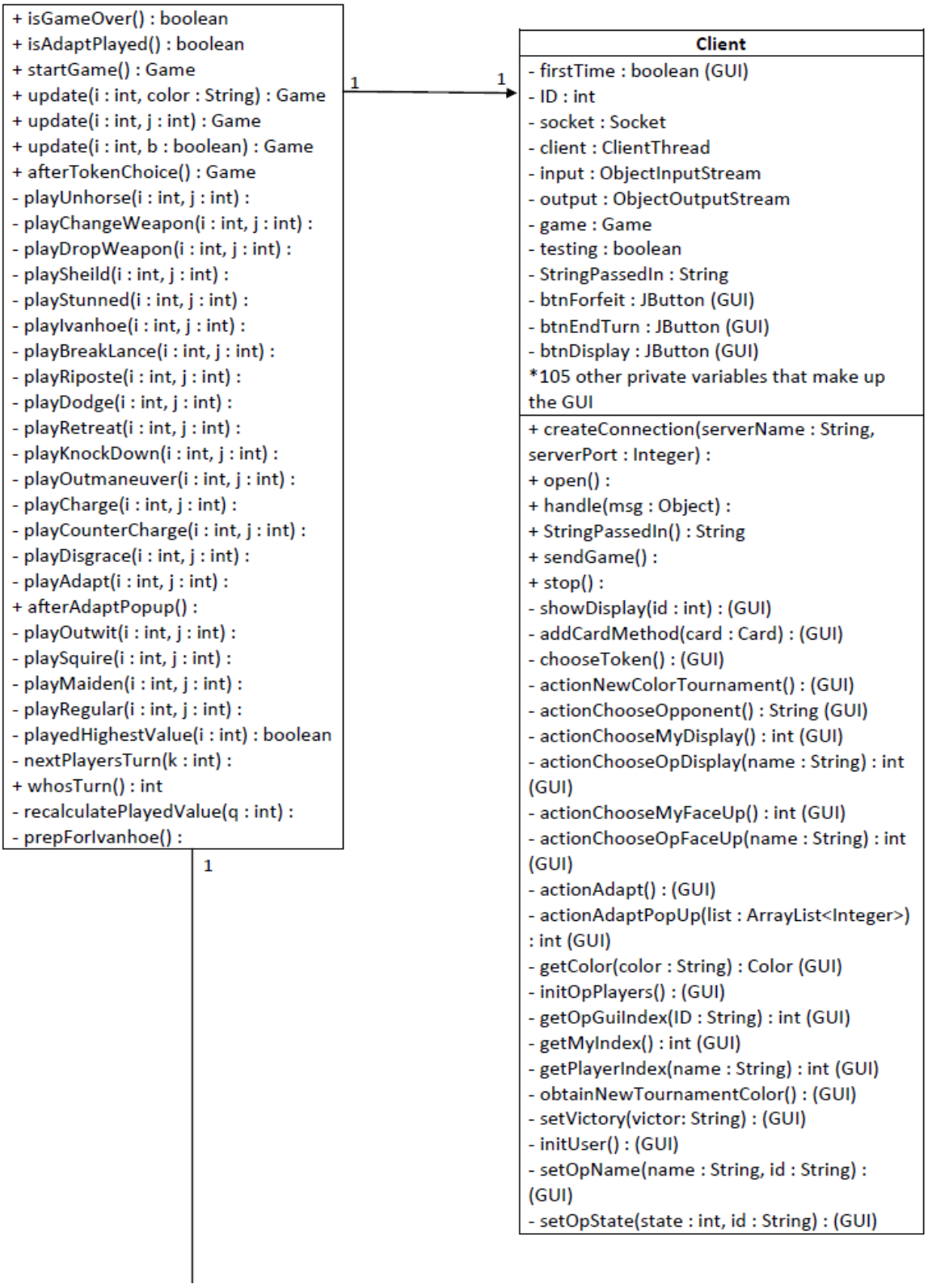
UC-05	Player Wins Tournament
Description: This use case describes how a player wins a tournament in Ivanhoe.	
External Actors: 2, 3, 4, or 5 players.	
Trigger: Second last player forfeits from tournament.	
Pre-Condition: Player has played the highest value in tournament.	
Sequence: **1.)** Second last player in tournament forfeits. **2.)** Player is awarded token.	
Post Condition: Player is awarded token of the current tournament color. If tournament color is purple, player may choose which color token he is awarded.	
Resulting Event: Tournament winner starts next tournament.	
Alternatives: Tournament winner wins game.	
Traceability: R5	

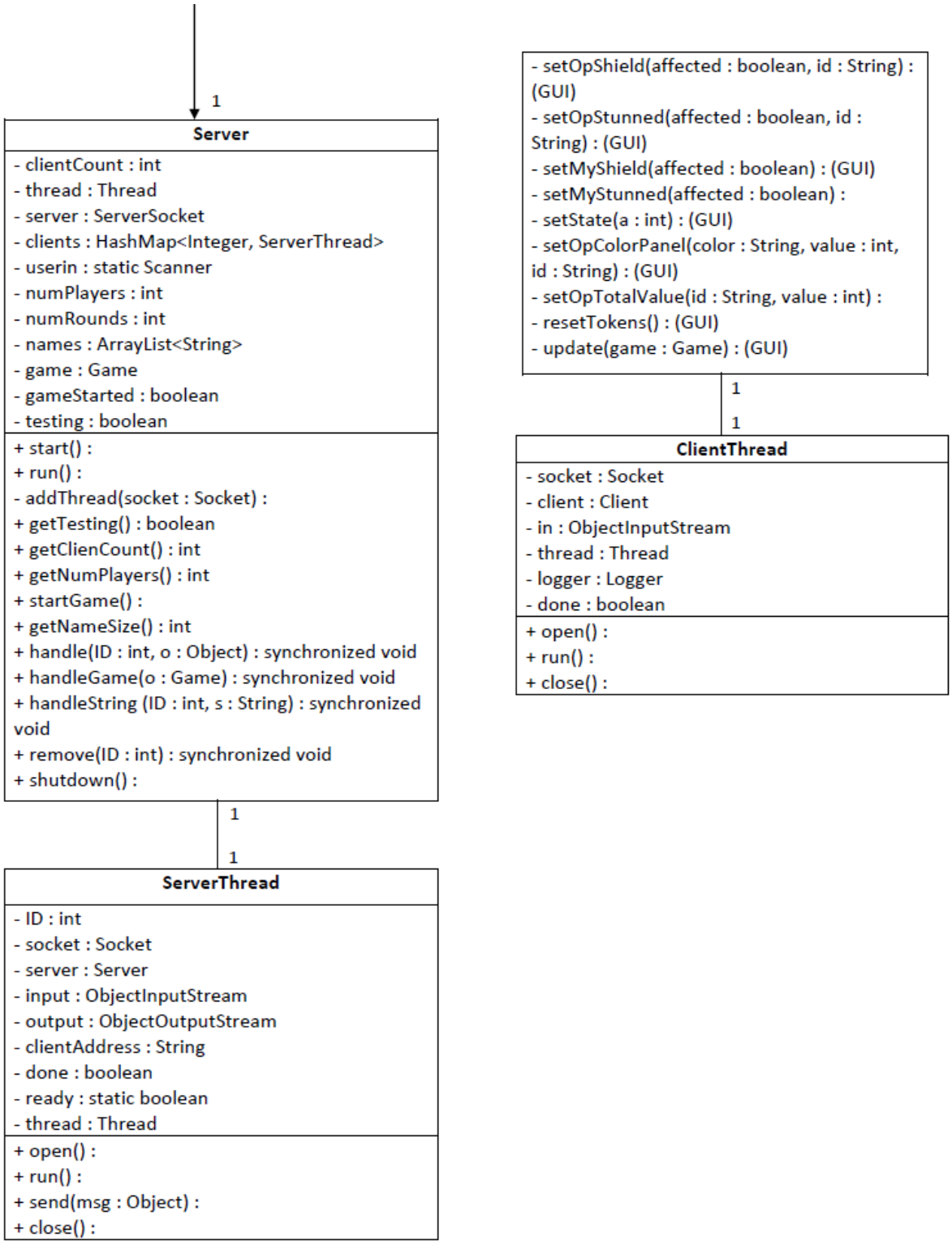
UC-06	Player Wins Game
Description: This use case describes how a player wins a game in Ivanhoe.	
External Actors: 2, 3, 4, or 5 players.	
Trigger: Second last player in tournament forfeits.	
Pre-Condition: **For 2-3 player:** Player has 4 different tokens than the current tournament color and player has played highest value in current tournament, or the player has 4 different color tokens and the current tournament color is purple and player has played highest value in current tournament. **For 4-5 players:** Player has 3 different tokens than the current tournament color and player has played highest value in current tournament, or the player has 3 different color tokens and the current tournament color is purple and player has played highest value in current tournament.	

Sequence: **1.)** Second last player in tournament forfeits. **2.)** Player is awarded the final token needed to win the game.
Post Condition: Player wins game.
Resulting Event: Game ends.
Alternatives: None.
Traceability: R2

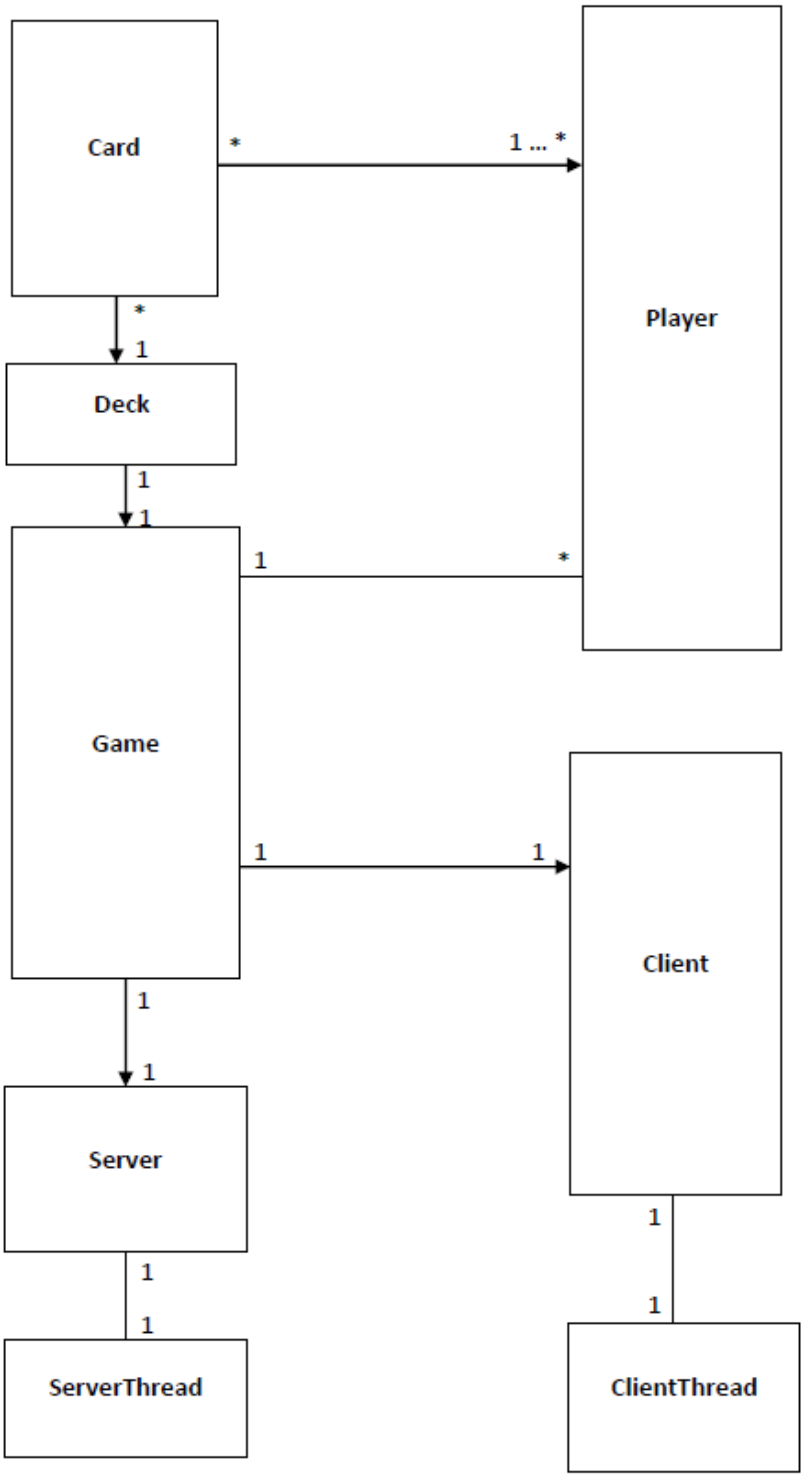
Section 4 – Class Diagrams


1
Deck
- serialVersionUID : static final long
- cards : ArrayList<Card>
+ shuffle() :
+ draw() : Card
1
1
Game
- serialVersionUID : static final long
- numPlayers : int
- playerArray : ArrayList<Player>
- numPlayersInTournament : int
+ deck : Deck
+ discardPile : Deck
+ Purple : static final String
+ Red : static final String
+ Blue : static final String
+ Yellow : static final String
+ Green : static final String
- currTournamentColor : String
- lastTournamentColor : String
- lastTournamentWinner : int
- gameOver : boolean
- lastCurrTournamentColor : String
- lastLastTournamentColor : String
- lastCardPlayedInGame : Card
- adaptPlayed : boolean
1
*
+ isMaidenPlayed() : boolean
+ isSheilded() : boolean
+ isStunned() : boolean
+ isCardAddedToDisplay() : boolean
+ isTournamentColorNeeded() : boolean
+ isLastMaidenPlayed() : boolean
+ isLastSheilded() : boolean
+ isLastStunned() : boolean
+ isTokenChoiceNeeded() : boolean
+ isHasPlayedCardThisTurn() : boolean
+ playerSync(game : Game) :
+ draw() :
+ doesNotHaveColorInHand(color : String) : boolean
+ removeToken() :
+ awardChosenToken() :
+ removeFaceUp() :
- notInDisplay(c : Card) : boolean
+ addFaceUp() :
- notInFaceUp(c : Card) : boolean
- notKeptCard(c : Card) : boolean
+ hasPurpleCard() : boolean


+ isGameOver() : boolean
+ isAdaptPlayed() : boolean
+ startGame() : Game
+ update(i : int, color : String) : Game
+ update(i : int, j : int) : Game
+ update(i : int, b : boolean) : Game
+ afterTokenChoice() : Game
- playUnhorse(i : int, j : int) :
- playChangeWeapon(i : int, j : int) :
- playDropWeapon(i : int, j : int) :
- playSheild(i : int, j : int) :
- playStunned(i : int, j : int) :
- playIvanhoe(i : int, j : int) :
- playBreakLance(i : int, j : int) :
- playRiposte(i : int, j : int) :
- playDodge(i : int, j : int) :
- playRetreat(i : int, j : int) :
- playKnockDown(i : int, j : int) :
- playOutmaneuver(i : int, j : int) :
- playCharge(i : int, j : int) :
- playCounterCharge(i : int, j : int) :
- playDisgrace(i : int, j : int) :
- playAdapt(i : int, j : int) :
+ afterAdaptPopup() :
- playOutwit(i : int, j : int) :
- playSquire(i : int, j : int) :
- playMaiden(i : int, j : int) :
- playRegular(i : int, j : int) :
- playedHighestValue(i : int) : boolean
- nextPlayersTurn(k : int) :
+ whosTurn() : int
- recalculatePlayedValue(q : int) :
- prepForIvanhoe() :
1
1
1
Client
- firstTime : boolean (GUI)
- ID : int
- socket : Socket
- client : ClientThread
- input : ObjectInputStream
- output : ObjectOutputStream
- game : Game
- testing : boolean
- StringPassedIn : String
- btnForfeit : JButton (GUI)
- btnEndTurn : JButton (GUI)
- btnDisplay : JButton (GUI)
*105 other private variables that make up the GUI
+ createConnection(serverName : String, serverPort : Integer) :
+ open() :
+ handle(msg : Object) :
+ StringPassedIn() : String
+ sendGame() :
+ stop() :
- showDisplay(id : int) : (GUI)
- addCardMethod(card : Card) : (GUI)
- chooseToken() : (GUI)
- actionNewColorTournament() : (GUI)
- actionChooseOpponent() : String (GUI)
- actionChooseMyDisplay() : int (GUI)
- actionChooseOpDisplay(name : String) : int (GUI)
- actionChooseMyFaceUp() : int (GUI)
- actionChooseOpFaceUp(name : String) : int (GUI)
- actionAdapt() : (GUI)
- actionAdaptPopUp(list : ArrayList<Integer>) : int (GUI)
- getColor(color : String) : Color (GUI)
- initOpPlayers() : (GUI)
- getOpGuiIndex(ID : String) : int (GUI)
- getMyIndex() : int (GUI)
- getPlayerIndex(name : String) : int (GUI)
- obtainNewTournamentColor() : (GUI)
- setVictory(victor: String) : (GUI)
- initUser() : (GUI)
- setOpName(name : String, id : String) : (GUI)
- setOpState(state : int, id : String) : (GUI)


1
Server
- clientCount : int
- thread : Thread
- server : ServerSocket
- clients : HashMap<Integer, ServerThread>
- userin : static Scanner
- numPlayers : int
- numRounds : int
- names : ArrayList<String>
- game : Game
- gameStarted : boolean
- testing : boolean
+ start() :
+ run() :
- addThread(socket : Socket) :
+ getTesting() : boolean
+ getClienCount() : int
+ getNumPlayers() : int
+ startGame() :
+ getNameSize() : int
+ handle(ID : int, o : Object) : synchronized void
+ handleGame(o : Game) : synchronized void
+ handleString (ID : int, s : String) : synchronized void
+ remove(ID : int) : synchronized void
+ shutdown() :
1
1
ServerThread
- ID : int
- socket : Socket
- server : Server
- input : ObjectInputStream
- output : ObjectOutputStream
- clientAddress : String
- done : boolean
- ready : static boolean
- thread : Thread
+ open() :
+ run() :
+ send(msg : Object) :
+ close() :
- setOpShield(affected : boolean, id : String) : (GUI)
- setOpStunned(affected : boolean, id : String) : (GUI)
- setMyShield(affected : boolean) : (GUI)
- setMyStunned(affected : boolean) :
- setState(a : int) : (GUI)
- setOpColorPanel(color : String, value : int, id : String) : (GUI)
- setOpTotalValue(id : String, value : int) :
- resetTokens() : (GUI)
- update(game : Game) : (GUI)
1
1
ClientThread
- socket : Socket
- client : Client
- in : ObjectInputStream
- thread : Thread
- logger : Logger
- done : boolean
+ open() :
+ run() :
+ close() :

Simpler Class Diagram (Visual Purpose Only)



CRC Cards

Card	
Has a color Has a value Has action card value	

Deck	
Has cards Can shuffle cards Can draw a card	Card

Player	
Has a hand (cards) Has a display (cards) Has a faceUp (cards) Has tokens Has target player Has chosen card Has given card Can draw card	Card Game

Game	
Has all players Has a deck Has a discard pile Has current tournament color Has last tournament color Can start a game Can update game state Can play an action card Can select who is next to play	Deck Player Card

Client	
Creates the GUI Connects to the server Handle game objects received from server Can send game objects to server Can update the GUI	Game ClientThread

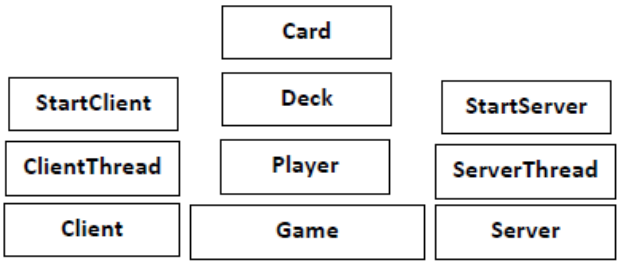
ClientThread	
Can create a thread Can open a socket Can close a socket Listens for incoming objects from server Has a logger	Client

Server	
Has a game Has a list of player names Can start a server Listens for clients to connect with Can start a game Handles game objects from clients Can shutdown	ServerThread Game

ServerThread	
Has a server Can open a socket Listens to incoming objects from clients Can send objects to clients Can close a socket Has a logger	Server

StartServer	
Has a server Has a logger Has a scanner Can initialize and run main program	Server

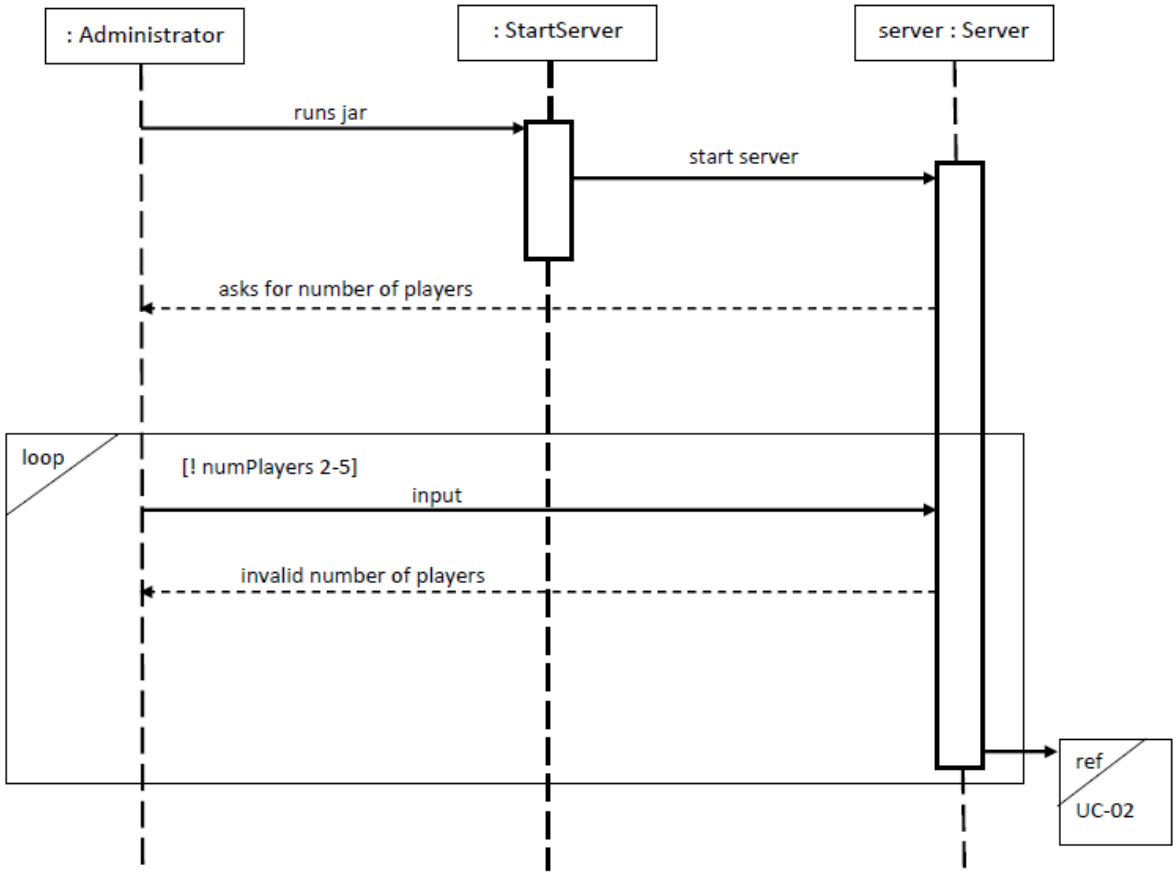
StartClient	
Can initialize and run main program	Client



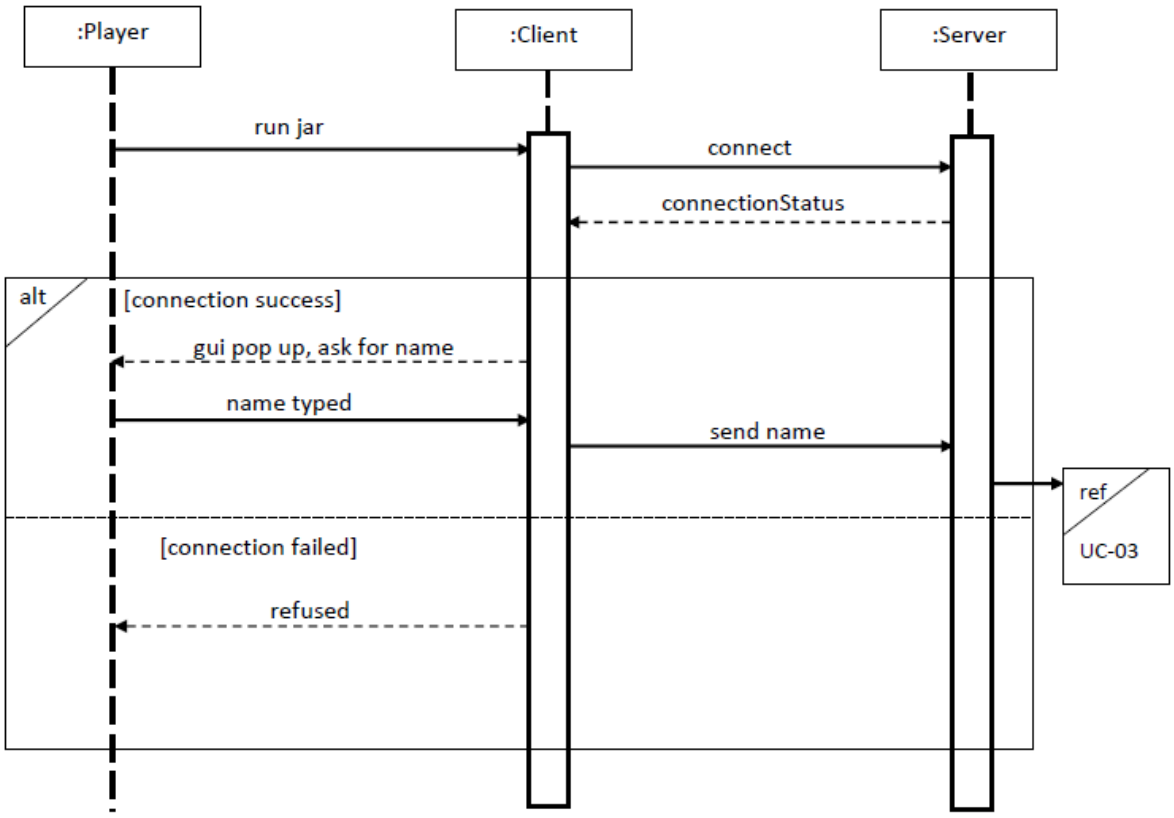
Section 5 – Interaction Diagrams

UML 2.0 Interaction Diagrams corresponding to UC

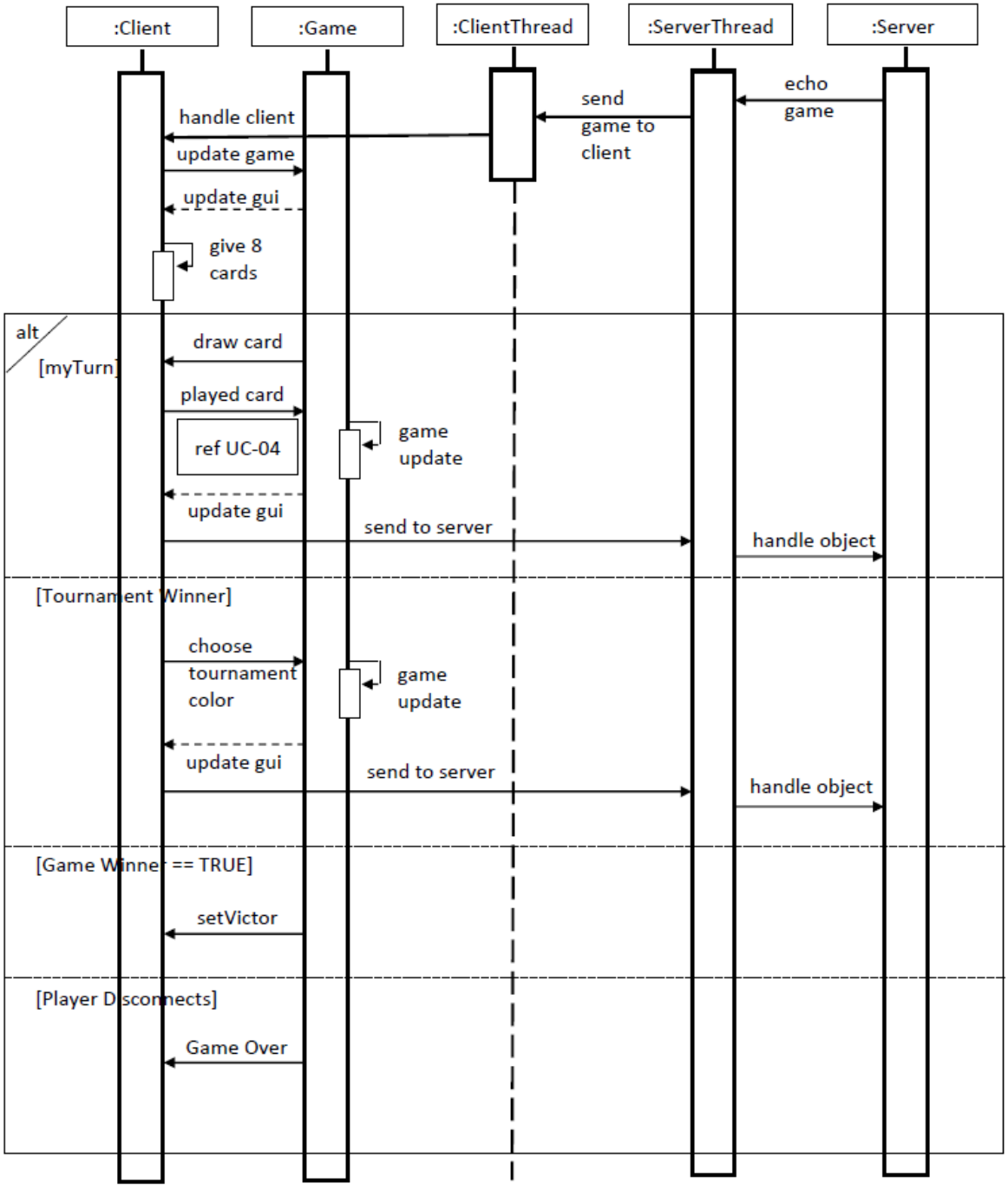
UC-01 : Admin Starts Server



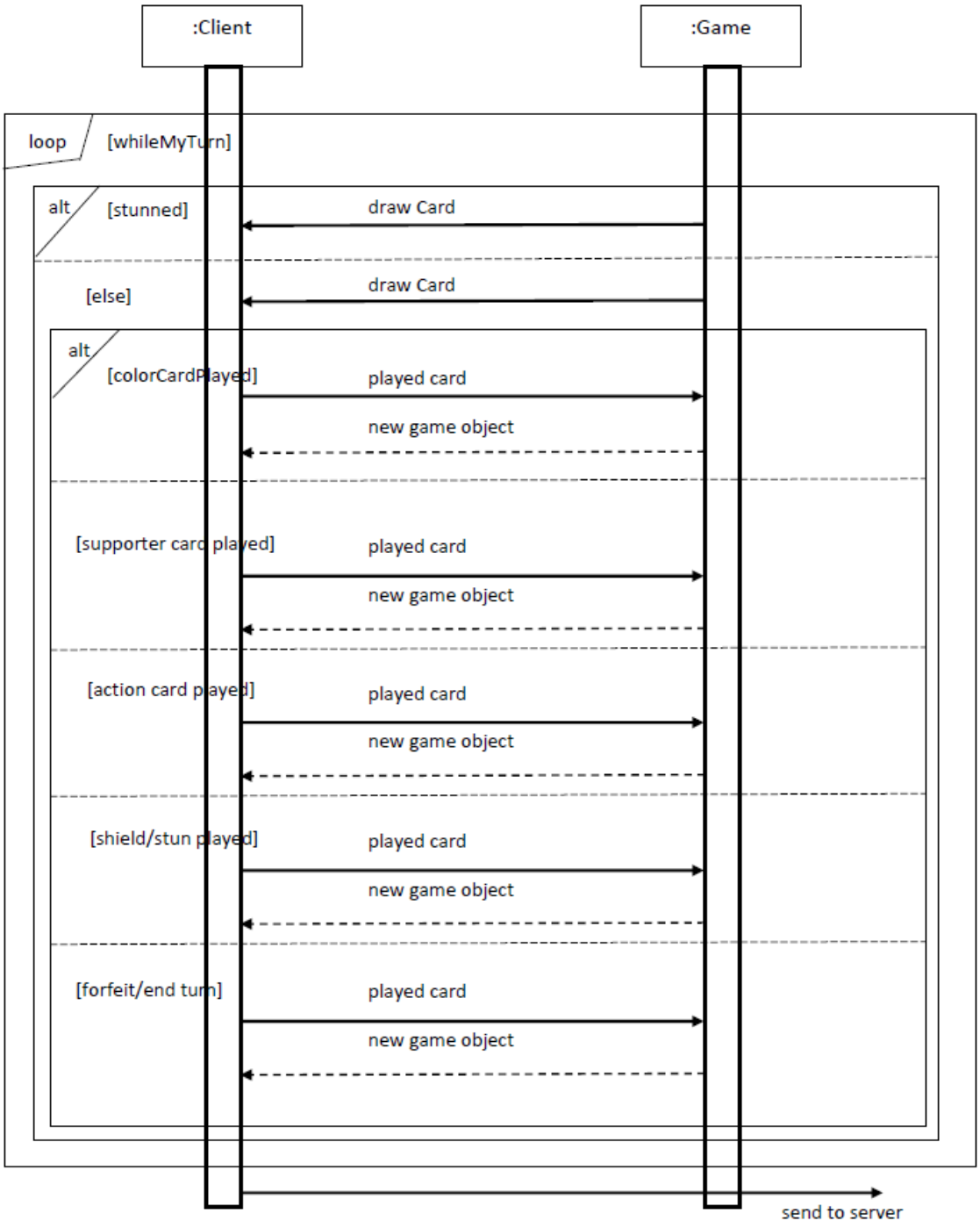
UC-02 : Players Join Game



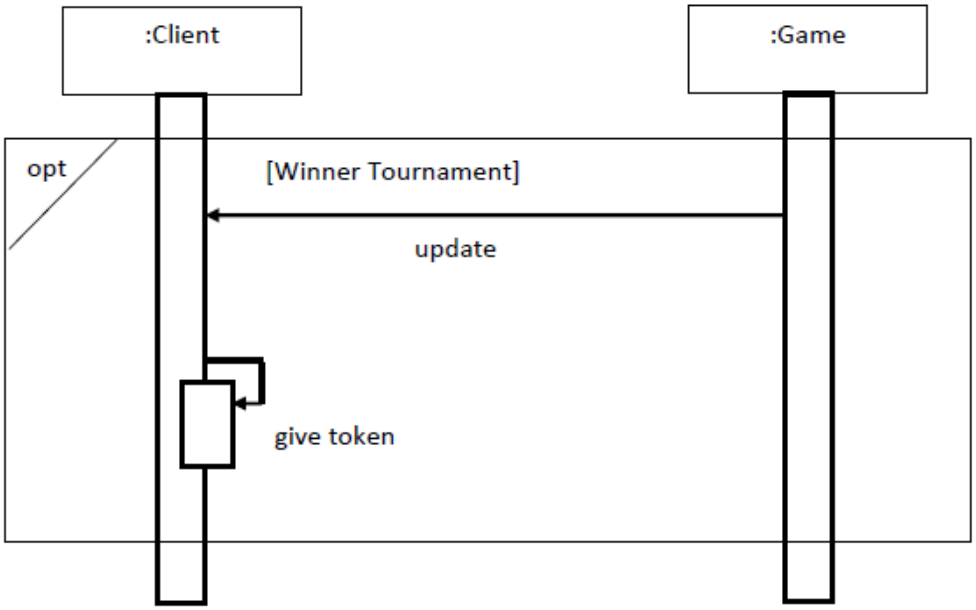
UC-03 : Players Play Game



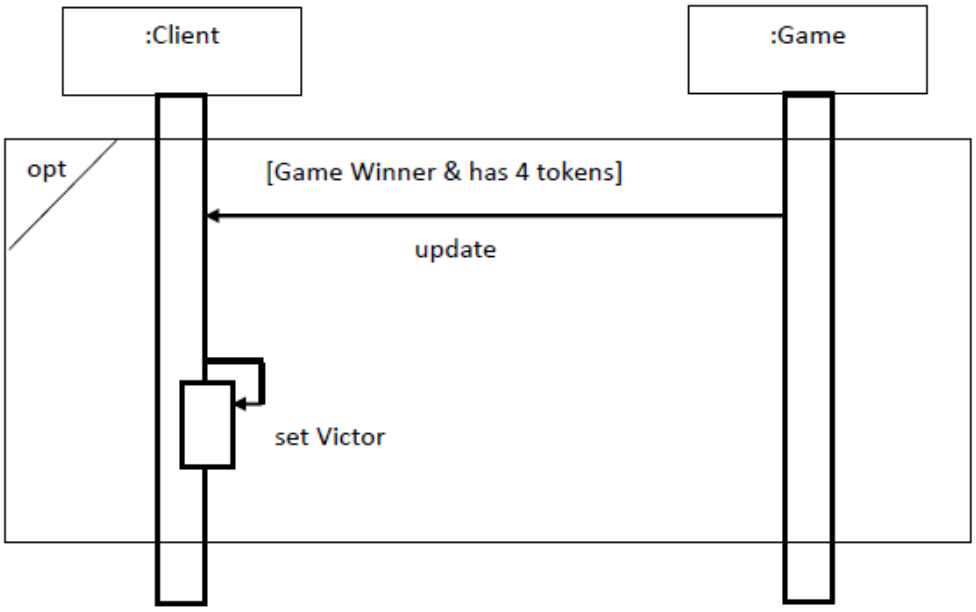
UC-04 : Player Plays Turn

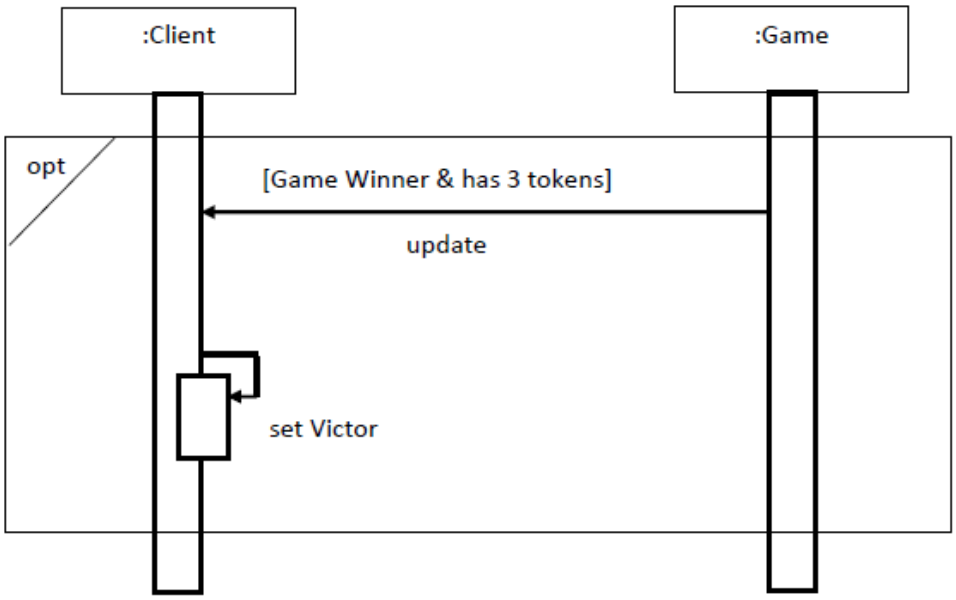


UC-05 : Player Wins Tournament



UC-06 : Player Wins Game




:Client
:Game
opt
[Game Winner & has 3 tokens]
update
set Victor

Section 6

6.1 Networking Model:

The software architecture of choice we chose to go with is the ClientServer Model. Although there exist other design models that are great and robust, sometimes analysis may lead one down a different route so as to meet the requirement that best suits the system at hand. Our analysis shows that in a game environment where each opposing player must wait for their turn to play, it is best to have a model that takes the current state of the Game after player makes a change, and echo that state back to all the players in the game. For example, if it's player 1's turn to play, and the player chooses to draw a certain card, the change in that game state must find a way to show all the other players that this is what the result of player 1's play had achieved. This change is sent as a request to the server through the socket to tell the server please send this back to all the other connected players(including Player1).
As can be seen in Figure 6.2, this strategy is extremely simple and robust, with great encapsulation and control capabilities. With this design strategy, one can always code for efficiency and robustness better than in say a peertopeer(P2P) model where there is no one dedicated server engine. It may make coding a bit more complex, but it pays to have a system follow this model as it is universally accepted as a more robust strategy of networking design(although it's impossible for everyone to agree on one thing). As can be seen in Figure 6.1, a sidebyside comparison of the two main Model's Networkers usually use when designing a network. The first is the one of choice for highlevel businesses, and the later(P2P) is more crude and suceptible to interruptions.

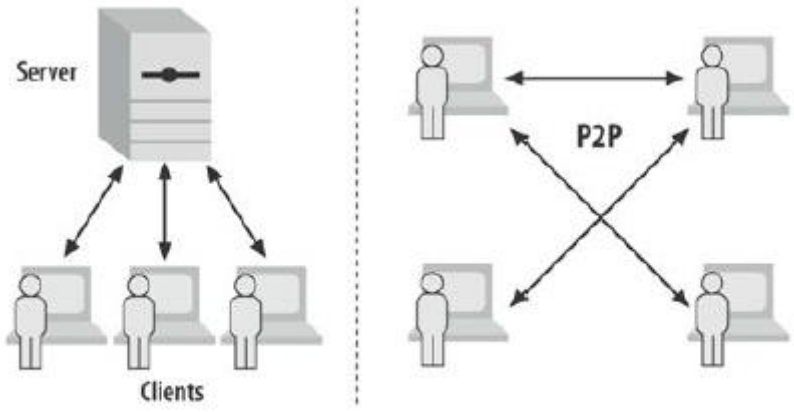


Figure 6.1 (Side-by-side overview of 2 common Network Models)

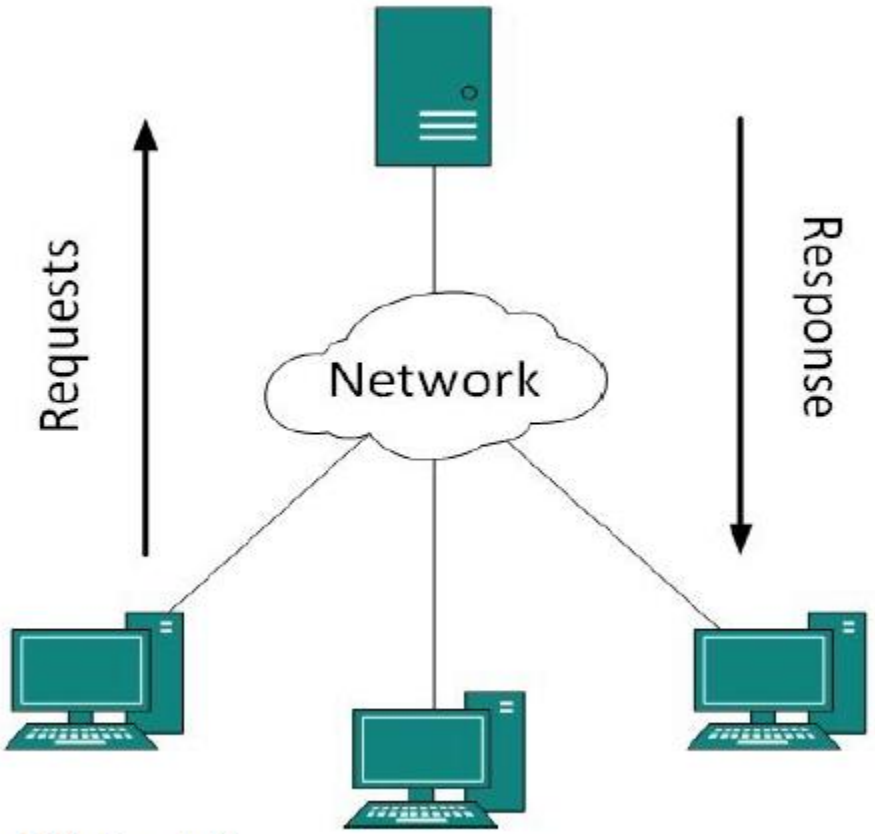


Figure 6.2(Client-Server Model)

Hence, as Figure 6.2 show, the Server sends a game object instance(which is the response part of the figure) to all of the clients, and the clients keep sending requests back. Server echoes a response based on the request from each successive client until a player wins the game and no more requests can be sent to the server anymore.

6.2 Threads

Threads:

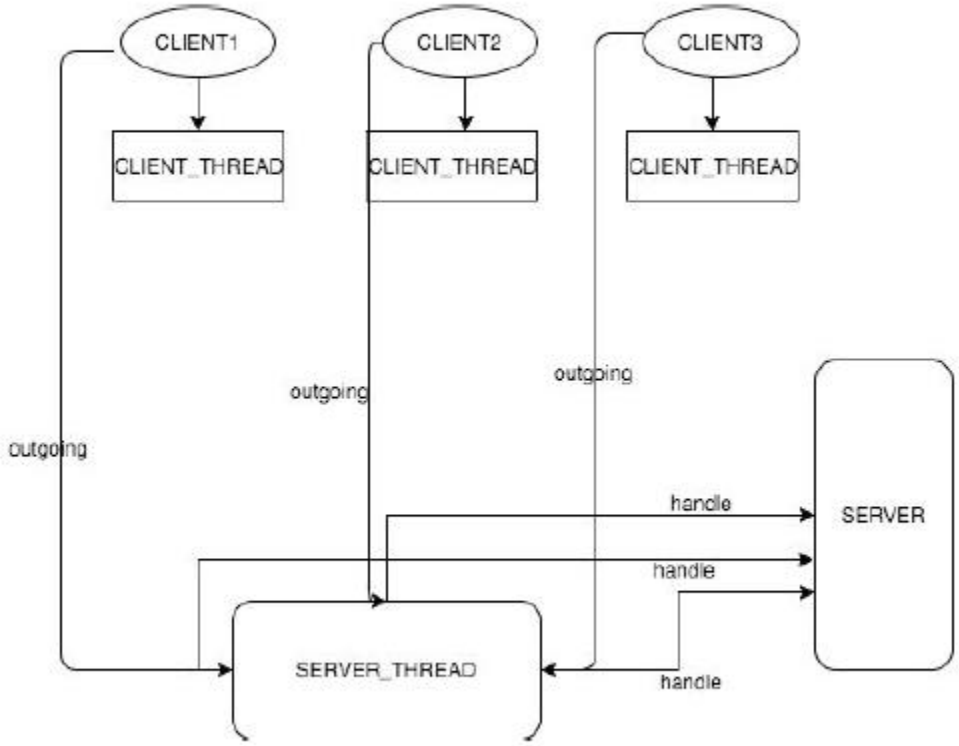


Figure 6.3(Thread Inter-Relation)

What is a thread? How does one create one? After creating it how does it go about being integrated into the logic of the code? Why use the synchronize?
These are all issues that were needed in order to implement this Network. Figure 6.3 depicts how the interactions between the threads and regular serverclient takes
place. It is worth noting that although the threads run independently of one another, given the speed at which this is done the threads can be thought of as coming to life and ending at around the same exact time. Also, in the figure above, the SERVER_THREAD was made large in order to better visualize the intercommunication
that takes place between the classes/threads. It is in fact 3
separate SERVER_THREADS(which is what was meant by the larger size of it).

6.3 Class Interactions

The Network begins it's life in the Server, where the server sets up a port on the specified address and remains listening on that port with the ip address for the life of the system. The Server is run by a proxy class Start_Server which prompts the user a choice of whether to start,shutdown, or go into testing mode(for ease of use for Network_test cases). After choosing to start, the system creates a Server instance and creates a dedicated Server thread in order to better encapsulate the Server and listen for any incoming connection onto the dedicated port it's listening on. The system then prompts the user of the server to choose between 2 and 5 players, remaining active so long as the user goes out of that range. At any one point in time, after choosing player numbers, the server goes into listening mode(among other things). For the purposes of this Game, the GUI has been embedded into the client for ease of use, and functionality. It may appear less encapsulated, yet for the person running and playing the game, it will certainly be more responsive.
When a client connects, the GUI prompts them for their name, and after having entered their name they are automatically placed into the game. After all clients have connected, the Network follows the Transfer Control Protocol(TCP) on how the information is being sent and received all throughout the lifecycle of the program(further on this later).
The diagram above depicts the interactions between the clientserver
and their
respective threads. For each Client there exists a Client_Thread, and a Client_Thread's main purpose is to remain in a while loop 'listening' on the incoming input through its client's Socket Stream(more on this below. Whenever anything gets received it calls the handle of the client class which checks to see what to do with that data. One must remember that the client only has two handling modes, If String then echo it to gui screen(such as name), if Game object link it with GUI for screen update.
On the Server side of things, it follows the same rationale, except that the server does not physically send anything through it's class, but rather delegates the task to the Server_Thread to send to its respective client. In both of the Client and Server classes, both classes have a dedicated handle method to know what type of object is passed through to know how to delegate the tasks correctly. The handle functions are always called from the Threads, which seemed to be the most straightforward way of going about implementing this since that is practically the sole purpose of the Thread.

6.4 Object Streams

There are many different functions that allow one to write to an Object's input and output Stream. The one that was chose for this Game was the Object Stream Readers.
This stream reader was chosen so as to remain compatible with the robustness of the ClientServer
Model. It was found to have the most flexibility given any Object
can be sent through the stream. This is very useful in the long run, should the Client choose to add more bells and whistles, the network stream readers and writer would not have to be changed. All that would be needed would be to handle that Object accordingly to it type.
Figure 6.4 depicts a typical rendering of what a Socket connection between a network appears like with both in and out stream in between the two.
Streaming is the most important factor to take into consideration when dealing with anything that requires an internet connection, and for this reason much attention was put into the integrity of the streams and when to close/open them.

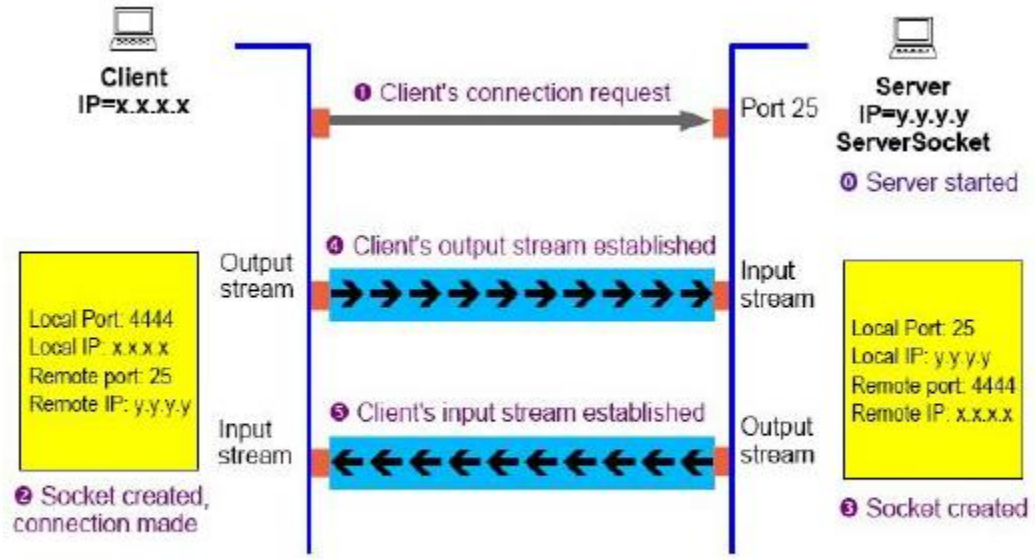


Figure 6.4(Streaming between Client-Server)

6.5 TCP

The TCP protocol is a universally accepted protocol, a standard that defines how to establish and maintain a network conversation via which application programs can exchange data.
TCP works with the IP(Internet Protocol), which is another protocol that defines how computers send packets of data to each other. TCP and IP are the basic rules defining the internet.
The figure below figure further digests the processes that take place when using TCP/IP in order to exchange and receive data.

Application Layer. This layer sends and receives data for particular applications, such as Domain Name System (DNS), Hypertext Transfer Protocol (HTTP), and Simple Mail Transfer Protocol (SMTP). The application layer itself has layers of protocols within it. For example, SMTP encapsulates the Request for Comments (RFC) 2822 message syntax, which encapsulates Multipurpose Internet Mail Extensions (MIME), which can encapsulate other formats such as Hypertext Markup Language (HTML).
Transport Layer. This layer provides connection-oriented or connectionless services for transporting application layer services between networks, and can optionally ensure communications reliability. Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are commonly used transport layer protocols.[2]
IP Layer (also known as the Network Layer). This layer routes packets across networks. Internet Protocol version 4 (IPv4) is the fundamental network layer protocol for TCP/IP. Other commonly used protocols at the network layer are Internet Protocol version 6 (IPv6), ICMP, and Internet Group Management Protocol (IGMP).
Hardware Layer (also known as the Data Link Layer). This layer handles communications on the physical network components. The best known data link layer protocol is Ethernet.

TCP/IP Layers

Source: National Institute of Standards and Technology

http://csrc.nist.gov/publications/nistpubs/800-41-Rev1/sp800-41-rev1.pdf

Section 7

7.1 Overall Architecture

The overall architecture of this software can be seen as three main components; The Game logic, the network, and the graphical user interface. The game logic is divided into four classes. The Card class defines card objects and all the attributes needed to distinguish between the several different types of card in the game. The Deck class is used as a collection class containing an array list of card objects. The deck class is used to define the draw and discard piles within the game and contains both the draw and shuffle methods needed during game play. The Player class defines the individual player objects that are assigned to each person playing the game. The Player class contains all the Card arrays and other attributes such as tokens that pertain to the individual players of the game. The Game class is the control center for the Ivanhoe game. The game object enforces all of the rules of the game and directs the actions of players within the game.

The Network architecture has 5 layers.

- **The hardware** : The hardware is simply the machine running the executable of the code.
- **Software:** The software layer is the Network code, which follows the ClientServer model. This software simply deals with creating 1 main server which deals with all the incoming and outgoing information from and to the client. It links all the parts of the program and offers very good encapsulation and security.
- **Connectivity:** How are the individual components connected? The connectivity is done through the Java compiler and hardware communication through the operating system.
- **Protocols:** The Transmission Control Protocol(TCP) and the Internet Protocol(IP) were used.
- **Mode of Transmission:** The Network can work through either a wired or wireless mode of transmission.

Network architecture refers to the layout of the network, consisting of the hardware, software, connectivity, communication protocols and mode of transmission, such as wired or wireless.
GUI: Container based architecture, fills in display from game object get calls.

7.2 Patterns

Observer Pattern:

In terms of this software the subject would refer to the server's game object. The observers would be the GUI's of each player. As the game is played, any time the server's game object is changed the server must inform and update every player with the new state of the game. This is done by sending game objects to the client containing the new state of the game, and allowing each client's GUI to update the displayed content accordingly. The reason for

implementing the observer pattern within this software is directly linked to the fact that all players must know the state of the game at all times in order to determine appropriate moves and strategies.

Builder Pattern:
Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. The program makes good use of this pattern by use of the Game object. The game object is basically the class that gets created by the Card, Player, and Deck classes. It is inherently a simple class at first, yet when created, becomes a composition of all the other classes as well. The motivation for use of this pattern was obvious; we needed some way of sending only one type of object through the stream. Any more than that, and the code becomes less robust and exponentially more difficult to implement. The game object was able to answer all the possible input a player may instantiate by means of delegating tasks from game to respective classes.

Facade Pattern:
Facade pattern hides the complexities of the system and provides an interface to the client in which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities. It was apparent from the start, the client has no interest in our code. All that the client's need is the ability to access the gui and play the game as per the requirements. The interface can be viewed as the client class under the guise of the gui window. The real logic of all of this is tucked away in the objects package under the Game,Card,Deck, and Player classes.

7.3 Refactoring

Since the submission of iteration 1 many parts of the code base have been refactored. In terms of game logic one of the major changes was made to the play card update method. Originally the update method for playing cards was a giant Swiss army knife function which handled regular cards, supporter cards, and action cards all within the same function. This update function has been refactored into a dispatch method. Depending on which card is played, the update method redirects control to a specific play card method designated to each type of card. This dispatch method redistributed the responsibility of playing cards to smaller specific methods making the code cleaner and more organized. Another major change was the addition of helper methods within the Player, and Game classes allowing for any repetitive code to be encapsulated into a single function which can be called instead of repeating several lines of code in many situations. The addition of helper functions substantially reduced the number of lines of code and increased the readability of many functions.

The Network wasn’t a 100% functional for iteration1. After iteration 1, the Network

became fully functional, keeping with the format of most of iteration 1 Network code, yet adding more robustness so as to have the network properly functioning.
One main change was in getting the game object to properly flow to all clients after all players have connected. This required some changes in the Server_Thread class so as to check certain conditions in the run() method. After all clients have been initialized the server thread automatically calls the server class to echo to all the clients.
Also, for testing purposes, a condition was also put in place when the server first starts up and prompts the user for either to start, shutdown, or testing. Testing option takes the Network into testing mode so as to allow the Network junit test cases to run uninterrupted by the gui and game objects flowing through the stream(shuts them out from working in testing mode). The final change made was the sending of a message through the server thread stream to the client class. A few helper methods in the client were also added in order to allow the access of this message and simplify the testing of whether this message was sent by the server and received by the client. This extra functionality also required a minor change in the handle method of the client in order to deal with strings instead of just game objects. The network test makes use of all these methods in test4, test5, and test6.
These are all extra functionalities that make the Network more testable and functional, and allows the network to work as was initially envisioned.

7.4 Pros/Cons

Pros:

- The logic engine is robust in terms of playing cards such that if a player tries to play a card that is invalid or will have no effect, the game will not allow the card to be played.

- Players cannot play any cards out of turn with exception of Ivanhoe.

- The TDD development of the software ensures that all components are working properly and allows for easy debugging.

- The substantial number of Junit test cases within the logic engine.

- Efficient Use of threads within the network add to the robustness and speed of the software.

- The use of loggers within the network make keeping track of program execution simple

Cons:

- The server and client send large serialized game objects back and forth, which is slow compared to sending messages as strings and requires a large pipeline to transfer data.

- During gameplay the player encounters many popups which can prove annoying and a detriment to the experience.

- The GUI interface is focused more on functionality rather than appearance and is not very aesthetically pleasing.

- The lack of encapsulation between client and GUI code presents some difficulty in identifying the origin of certain bugs.

- No AI strategies have been implemented.

Section 8

This software was developed using test driven development, and the code base contains 42 tests used in the development stage as well as over 100 post completion tests to ensure robustness.

8.1 TDD Tests

All TDD tests can be found in the package: testing

Basic Tests:

-cardConstructorTest.java
-deckConstructorTest.java
-gameConstructorTest.java
-playerConstructorTest.java

Tournament Tests:

-bTournTest.java
-gTournTest.java
-pTournTest.java
-rTournTest.java
-yTournTest.java

Method Tests:

-deckDrawMethodTest.java
-gameStartGameTest.java
-gameUpdateMethodTest.java
-playerDrawMethodTest.java

Situational Tests:

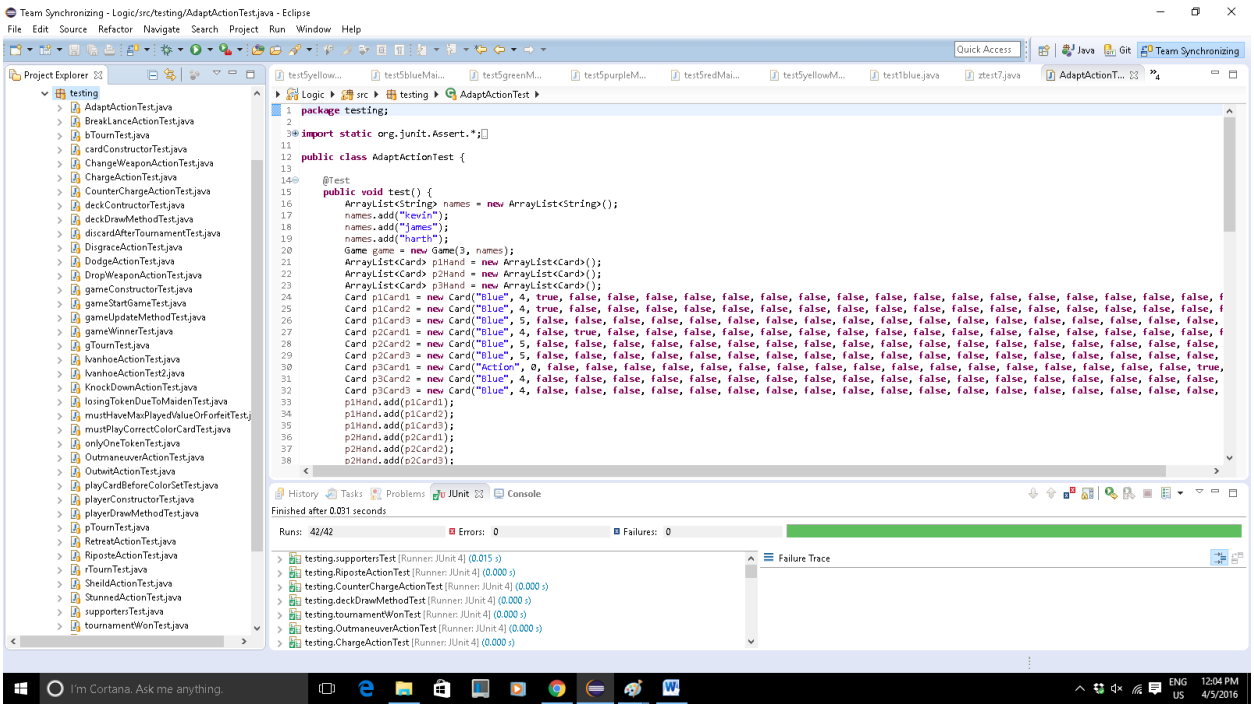
-discardAfterTournamentTest.java
-gameWinnerTest.java
-losingTokenDueToMaidenTest.java
-mustHaveMaxPlayedValueOrForfeitTest.java
-mustPlayCorrectColorCardTest.java
-onlyOneTokenTest.java
-playCardBeforeColorSetTest.java
-supportersTest.java
-tournamentWonTest.java
-unableToStartTournamentTest.java
-winningPurpleTournamentTest.java

Action Cards:

-AdaptActionTest.java

-BreakLanceActionTest.java
-ChangeWeaponActionTest.java
-ChargeActionTest.java
-CounterChargeActionTest.java
-DisgraceActionTest.java
-DodgeActionTest.java
-DropWeaponActionTest.java
-IvanhoeActionTest.java
-IvanhoeActionTest2.java
-KnockDownActionTest.java
-OutmaneuverActionTest.java
-OutwitActionTest.java
-RetreatActionTest.java
-RiposteActionTest.java
-SheildActionTest.java
-StunnedActionTest.java
-UnhorseActionTest.java

Figure 6.1 ALL TDD tests passing



8.2 Post Completion Tests

All post completion tests can be found in the package: Iteration2Tests

Tournament Tests:

a. one player draws/starts, others draw but do not participate (ie withdraw)

Corresponding tests:

- test1blue.java
- test1green.java
- test1purple.java
- test1red.java
- test1yellow.java

b. one player draws/starts, others draw but only one participates by playing

i) a card

Corresponding tests:

- test2blue1.java
- test2green1.java
- test2purple1.java
- test2red1.java
- test2yellow1.java

ii) several cards

Corresponding tests:

- test2blueSeveral.java
- test2greenSeveral.java
- test2purpleSeveral.java
- test2redSeveral.java
- test2yellowSeveral.java

c. one player draws/starts, others draw and some participate by playing

i) a card

Corresponding tests:

- test3blue1.java
- test3green1.java
- test3purple1.java
- test3red1.java
- test3yellow1.java

ii) several cards

Corresponding tests:
- test3blueSeveral.java
- test3greenSeveral.java
- test3purpleSeveral.java
- test3redSeveral.java
- test3yellowSeveral.java

d. one player draws/starts, others draw and all participate by playing
i) a card
Corresponding tests:
- test4blue1.java
- test4green1.java
- test4purple1.java
- test4red1.java
- test4yellow1.java

ii) several cards
Corresponding tests:
- test4blueSeveral.java
- test4greenSeveral.java
- test4purpleSeveral.java
- test4redSeveral.java
- test4yellowSeveral.java

e. starting with
i) a supporter
Corresponding tests:
-- test5blue1.java
- test5green1.java
- test5purple1.java
- test5red1.java
- test5yellow1.java
-test5blueMaiden
-test5greenMaiden
-test5purpleMaiden
-test5redMaiden
-test5yellowMaiden

ii) several supporters

Corresponding tests:

- test5blueSeveral.java
- test5greenSeveral.java
- test5purpleSeveral.java
- test5redSeveral.java
- test5yellowSeveral.java

f. a multiplayer tournament has several rounds where each player plays one and then several supporters in different rounds

Corresponding tests:

- test6blue.java
- test6green.java
- test6purple.java
- test6red.java
- test6yellow.java

g. trying to play cards that do not get the current player to beat the tournament originator (ie not enough to be the leader)

Corresponding tests:

- test7blue.java
- test7green.java
- test7purple.java
- test7red.java
- test7yellow.java

h. restriction to 1 maiden per player per tournament

Corresponding tests:

- test8blue.java
- test8green.java
- test8purple.java
- test8red.java
- test8yellow.java

i. winning and getting token

Corresponding tests:

- test9blue.java
- test9green.java

- test9red.java
- test9yellow.java

j. winning and choosing token when purple tournament

Corresponding tests:

-xtest10purple

k. losing with a maiden and losing a token

Corresponding tests:

- xtest11blue.java
- xtest11green.java
- xtest11purple.java
- xtest11red.java
- xtest11yellow.java

Action Card Tests:

a. playing this card on an unshielded player

Corresponding tests:

- xtest12AdaptEffect.java
- xtest12BreakLanceEffect.java
- xtest12ChangeWeaponEffect.java
- xtest12ChargeEffect.java
- xtest12CounterChargeEffect.java
- xtest12DisgraceEffect.java
- xtest12DodgeEffect.java
- xtest12DropWeaponEffect.java
- xtest12KnockDownEffect.java
- xtest12OutmaneuverEffect.java
- xtest12OutwitEffect.java
- xtest12RetreatEffect.java
- xtest12RiposteEffect.java
- xtest12SheildEffect.java
- xtest12StunnedEffect.java
- xtest12UnhorseEffect.java

b. playing this card on an shielded player

Corresponding tests:
- xtest12AdaptSheilded.java
- xtest12BreakLanceSheilded.java
- xtest12ChangeWeaponNoEffect.java
- xtest12ChargeSheilded.java
- xtest12CounterChargeSheilded.java
- xtest12DisgraceSheilded.java
- xtest12DodgeSheilded.java
- xtest12DropWeaponNoEffect.java
- xtest12KnockDownSheilded.java
- xtest12OutmaneuverSheilded.java
- xtest12OutwitSheilded.java
- xtest12RetreatSheilded.java
- xtest12RiposteSheilded.java
- xtest12SheildNoEffect.java
- xtest12StunnedNoEffect.java
- xtest12UnhorseNoEffect.java

**for Action cards that change the tournament color and shield/stunned, a no Effect test was performed rather than checking effect on shielded player.

c. undoing this card using Ivanhoe

Corresponding tests:
- xtest12AdaptIvanhoed.java
- xtest12BreakLanceIvanhoed.java
- xtest12ChangeWeaponIvanhoed.java
- xtest12ChargeIvanhoed.java
- xtest12CounterChargeIvanhoed.java
- xtest12DisgraceIvanhoed.java
- xtest12DodgeIvanhoed.java
- xtest12DropWeaponIvanhoed.java
- xtest12KnockDownIvanhoed.java
- xtest12OutmaneuverIvanhoed.java
- xtest12OutwitIvanhoed.java
- xtest12RetreatIvanhoed.java
- xtest12RiposteIvanhoed.java
- xtest12SheildIvanhoed.java
- xtest12StunnedIvanhoed.java
- xtest12UnhorseIvanhoed.java

d. checking a used action card is indeed thrown away

Corresponding tests:

- xtest12AdaptThrownAway.java
- xtest12BreakLanceThrownAway.java
- xtest12ChangeWeaponThrownAway.java
- xtest12ChargeThrownAway.java
- xtest12CounterChargeThrownAway.java
- xtest12DisgraceThrownAway.java
- xtest12DodgeThrownAway.java
- xtest12DropWeaponThrownAway.java
- xtest12KnockDownThrownAway.java
- xtest12OutmaneuverThrownAway.java
- xtest12OutwitThrownAway.java
- xtest12RetreatThrownAway.java
- xtest12RiposteThrownAway.java
- xtest12SheildThrownAway.java
- xtest12StunnedThrownAway.java
- xtest12UnhorseThrownAway.java

Scenario Tests:

a. the player who should start cannot start a tournament

Corresponding test:

-ztest1.java

b. last tournament was purple, cannot be purple again

Corresponding test:

-ztest2.java

c. trying to play an insufficient number of cards to become the leader on my turn

Corresponding test:

-ztest3.java

d. trying to play invalid cards (wrong color)

Corresponding test:

-ztest4.java

e. coming to end of deck

Corresponding test:

-ztest5.java

f. using CHARGE in a green tournament with every player with only green 1s: one card must remain

Corresponding test:

-ztest6.java

g. other example of overriding rule: at least one card must remain

Corresponding test:

-ztest7.java

h. winning the game

Corresponding test:

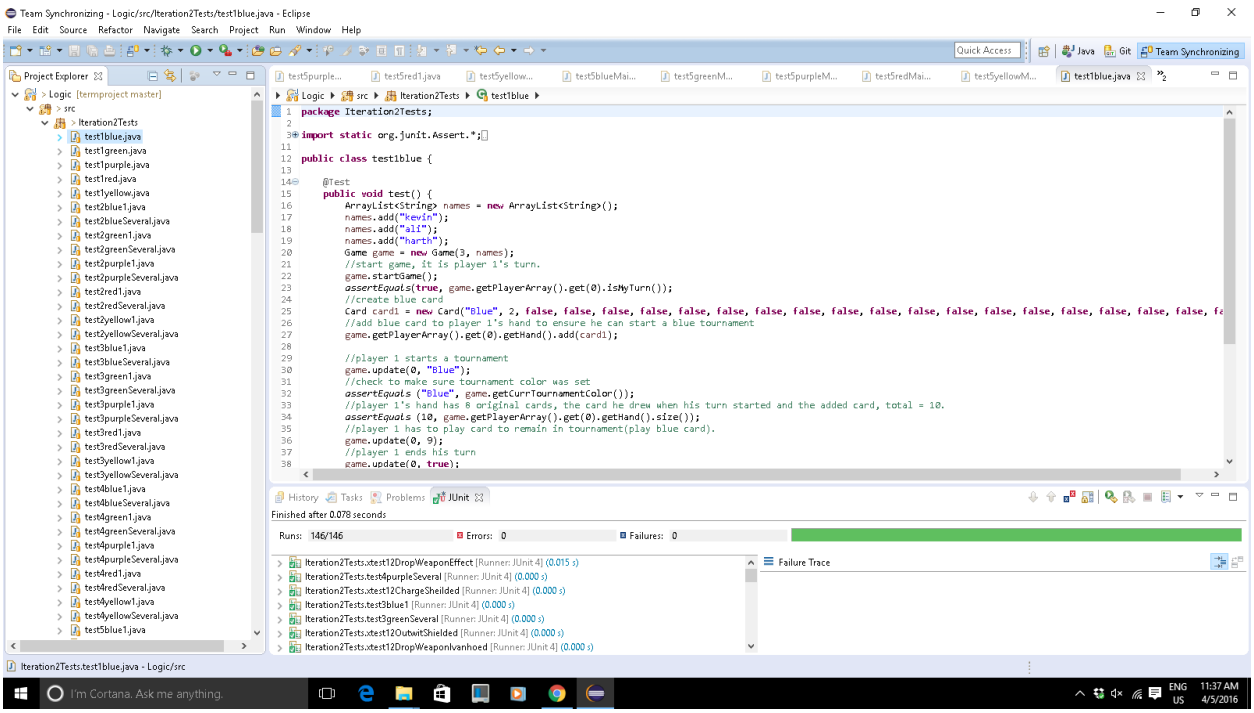
-ztest8.java

i. the deck uses 110 cards

Corresponding test:

-ztest9.java

Figure 6.2 ALL post Completion tests passing.



8.3 Robustness Tests

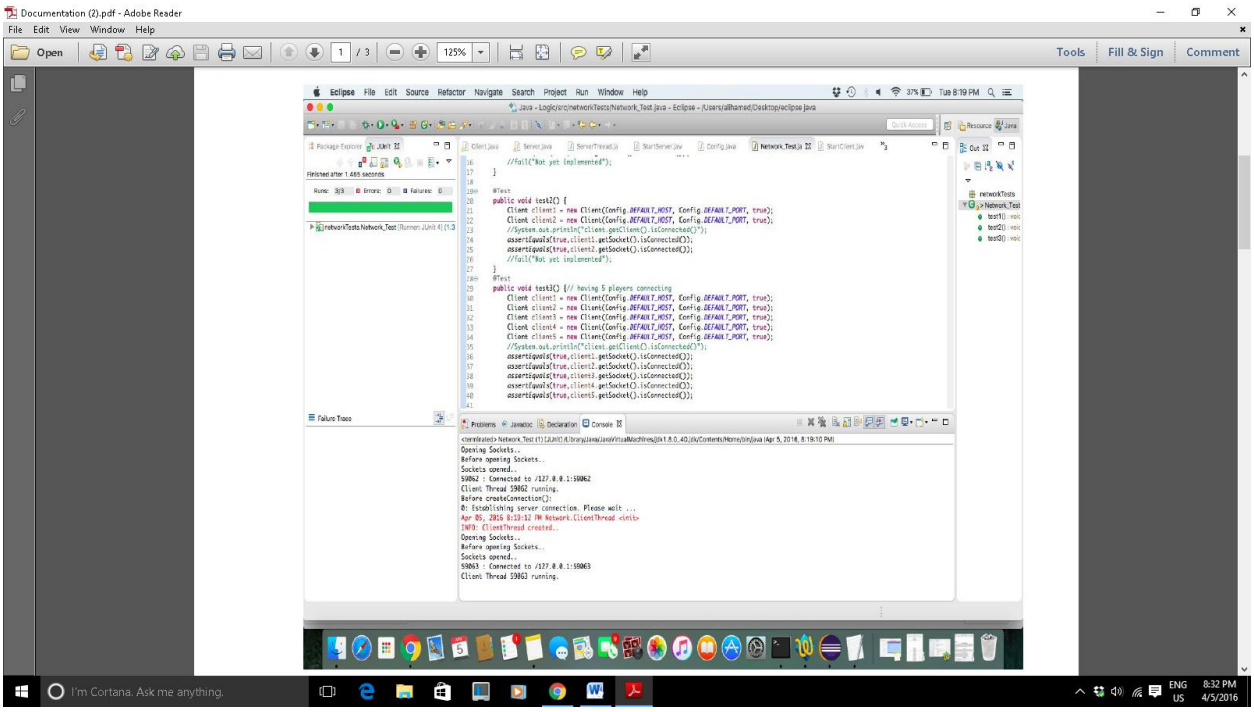
1.) Cannot choice duplicate token after winning purple tournament.
Corresponding test:
zztestRobustDuplicateTokenPurpleTournament.java

2.) Cannot play out of turn.
Corresponding test:
zztestRobustPlayingOutOfTurn.java

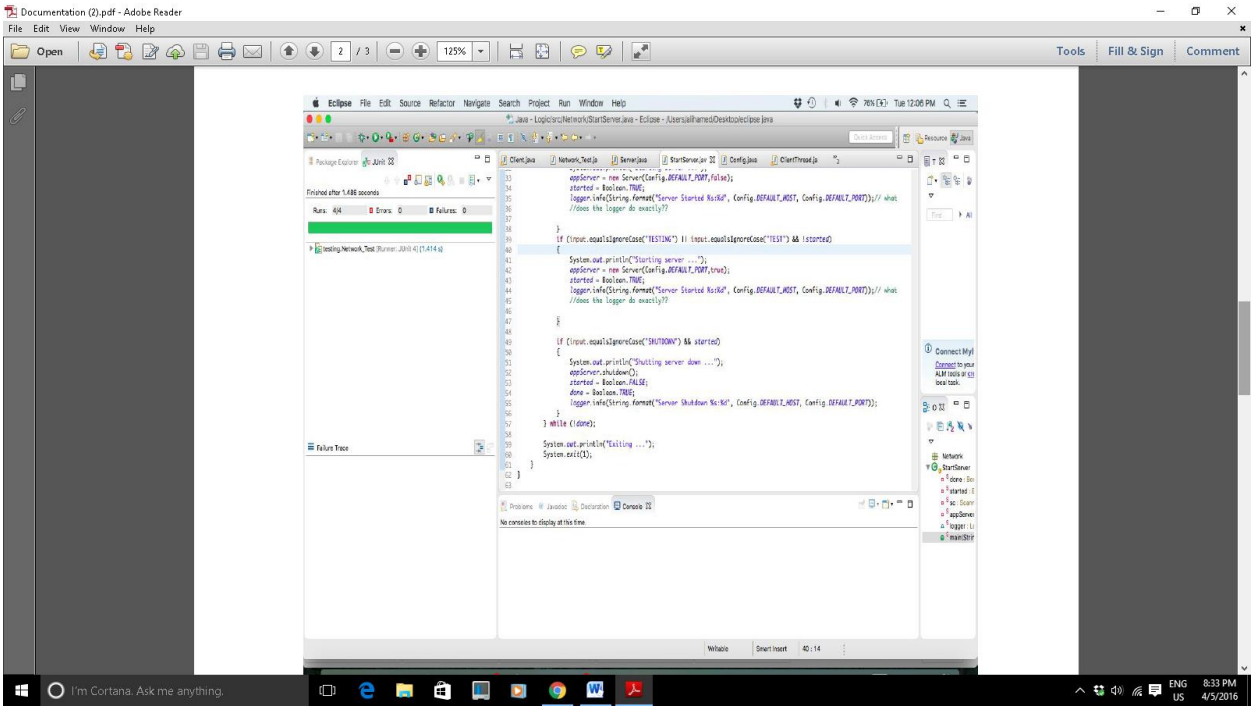
3.) Can play Ivanhoe out of turn.
Corresponding test:
zztestRobustPlayingOutOfTurnIvanhoe.java

8.4 Network Tests

TDD Network:
The first test “test1” checks to see if a client can connect to the server.
The second test “test2” checks to see if 2 clients can connect the server.
The third test “test3” checks to see if 5 players can connect to the server.



The fourth test ”test4” checks to see whether the server successfully sends a message to the client and the client successfully receives it.



The fifth test ”test5” checks to see whether the server successfully sends a message to 2 connected clients and the clients successfully receive them.