UNIVERSITY OF HERTFORDSHIRE

Faculty of Science, Technology and the Creative Arts

Modular BSc in Computer Science

6COM0282 – Computer Science Project

Final Report

April 2012

Kingdom of Warcraft

Xu Han

Supervised by: Ian Brandford

Abstract

Playing board game with friends during a party is a fantastic method to have fun. In this kind of game, strategy and chance both play important roles. Different games will focus on a different aspect, or a mixture of the two. The most traditional and popular kind of board game can be represented by a battle between two armies. To win a game, a player needs to beat all hostile groups, as well as defend the leader of his or her own team.

A good board game should be interesting, intelligence-needed and easy to be extended. It is very easy for a person to be addicted to such a game. However, there is a situation when someone wants to play a board game but there is no friend nearby. Therefore if there is a platform online, or just over a local network, which can be used by different people to play board game together, people will get much more fun.

In this project I have set out my idea to build such a platform for people to play a board game called “Kingdom of Warcraft”. It is a cross-platform application written in Java. After one player sets up a server over a local network, other players in the same net could join in and play together.

Content

CONTENTS PAGE

Chapter 1 Introduction

1.1 Project motivation

As a board game fan and a student whose major is computer science, I have always been interested in writing a board game application for people in different ages from all walks of life. It is a good way to make more people know what board game is, what they need to do and what they can experience in such a game. This project gives me a great opportunity to achieve my dream and implement this idea into real, as well as consolidates my skills in Java.

Before I started doing this project, the first thing came into my mind was the name and the content of this game. To choose a suitable board game which will be served by this application, I have undertaken a plenty of research on different board games. After playing most of them, I decide not to use any of existing ones. The reason is that some games are two hard to be specified clearly since they are born with a lot of FAQs, other games which do not have this problem are all in Chinese. Instead I came up with a great idea, which was making up a new game named “Kingdom of Warcraft”, referring to another popular board game called “The Killers of Three Kingdoms” in China and some features of an American game --“BANG”.

My previous experience in doing projects was only playing a role as a programmer in a team, and the knowledge I used in a project was always relating to Java EE, which relied on my ability of doing Servlet and JSP. However, this project is the first complete Java application done all by myself. Before doing this project, I have studied Java for about 3 years. Therefore, I am adept at using most of java knowledge like socket programming, object serialization, graphic user interface, and frameworks like Spring into my project. All the other aspects which I am not skilled with were easily searched and answered by surfing internet and my tutors respectively.

Since board game is an area I am interested in, as well as the fact I am proficient in Java, doing this project is a joyful and fabulous experience. After this development I have a much clearer thought on how to make a Java application, why design plays an important role in software development, and how a good software development model will benefit the programmer.

1.2 Aims and objectives

The key objectives which build the foundation of the project are:

1. Allow one user in a network to build a server with a specific port number

chosen by this player.

2. Allow other users (at most 7) in the same network to join in the game with the IP address and the port number of the existing server.

3. Allow each user to specify a unique name and choose a favourite show which will be displayed when they are wandering the waiting hall.

4. Allow the user who builds the server to specify the rule in the game
   *Rule means which kind of mode the game will use, whether some extension cards will be used, whether players can chat during the game, the time limit for a player to take some action, and so on.

5. The person who builds the server should be informed to be okay to start the game once all the other players are at ready status.

6. The whole game flow should be controlled by this application, including each player plays his or her own phase in turns, the event triggered when a player uses a card, informing the player to take the corresponding action when he or she turns to be the target of an event, when the game is finished, who are the winners, and so on.

7. The corresponding sound should be played when an event is triggered

8. Allow the users to set their own configurations of the application.
   * Configurations include the control of volume, turn on or turn off the sound and animations, and so on.

In addition, the advanced aims which are desirable if time permitting are:

1. Allow a player to choose a general to act as at the very beginning of the game. Different generals will have different skills, which makes the game more interesting.

2. Appropriate method should be taken by the server if some players turn to be off-line during the game so this application will not crash.

3. Allow a player joins the game as an “observer”.
   *An observer cannot play the game at this moment. What he or she can do is just seeing what happens during the game.

4. The corresponding animation should be shown when an event is triggered.

5. Allow a player view the information about all the cards and generals before

they connect to or build a server.

These aims also become the functional requirements for this project. All these requirements are set and collected all by myself. This part can be treated as the first stage – requirements gathering in a project development cycle.

1.3 Report Structure

Chapter 2 illustrates the basic rules of the game. A more detailed game specification is attached in Appendix 1. The initial architecture designed is specified in Chapter 3.

Chapter 4 talks about the software design stage with a use case diagram and a class diagram. The core bit, software implementation is specified in details in Chapter 5. This project starts with dealing with graphic user interface; this aspect will be discussed in Chapter 5.1. In Chapter 5.2, I mainly talks about how to deal with the communication between client and server using Java sockets. A hard bit of this application, round event, is elaborated in Chapter 5.3. Chapter 5.4 mainly talks about how a software designed pattern – the observer pattern is used in this project. Some useful tool classes are introduced in Chapter 5.5. A common part of all the online games, the waiting hall, Chapter 5.6 illustrates the implementation of it. Chapter 5.7 shows the second hard bit of this application, also where this project ends: the method to deal with skills. The method to improve the performance is talked in Chapter 5.8.

Chapter 6 specifies the testing in this project. An evaluation of my success in writing this software is contained in Chapter 7, as are future improvements which could be made.

Appendix Ⅰ is a detailed game specification.

Chapter 2 Description of the game

"Kingdom of Warcraft", which is offered by this application, is a board game which combines strategy, competition, reasoning with camouflage together. To win a game, all players must think carefully before they take any actions. It is one player, one action, or maybe mere one simple card that can twist the present situation of one game.

There are two basic modes in this game. The first one is called "Survival Mode" which requires a player to kill all the other opponents then to be the

only one alive at last. The other one is “Role Mode”, which is also the key mode of this game, is a little complex compared with the former one. There are four types of roles in a game: Lord, Loyalist, Rebel and Spy. At the beginning of the game, each player will be allocated a role. The task of Lord and Loyalist is to kill all the Rebels and Spy, Rebels’ aim is to overthrow the Lord. In contrast, the mission of Spy is the heaviest. Spy is required to kill all the other players except the Lord at first, then duel with Lord, beat him or her to be the final winner.

After assigning the roles, players will be given some generals (normally 3) to choose from. Different generals have different skills. Skill is designed according to the ability and the character of the general. When all the players have finished their choice, Survival Mode game will start with a random player’s round, while Role Mode game will start at the Lord’s round first.

During the game, each player will do his or her own round in turns. A simple round can be divided into six different phase: round-start, judging, drawing, action, discarding, and round-finish. If a player is in an abnormal state, judging phase is needed to decide what kind of action will be taken on the player such as jump the action phase, drawing phase… Generally, player can draw two cards in their drawing phase. In the action phase player can do whatever he or she can do allowed by the game rule such as attack other player, recover HP and so on. Player is required to discard some cards in the discarding phase. The amount of the cards to be discarded is decided by player’s current HP. For example, the current HP of a player is 2 and he or she has 5 cards in hand, he or she must discard 3 cards, which is the difference between the total amounts of cards in hand and player’s current HP.

All the game cards can be divided into three kinds: basic, equipment and skill cards. There are three kinds of cards which are belonging to the basic card: [Attack], [Miss] and [Peach]. [Attack] is the basic way to attack other players. However the target can avoid this attack using a card [Miss]. If the target cannot play such a card, he or she will receive 1-point damage from the attack source. [Peach] can be used to recover one HP after the player has been hurt in the action phase. When a player’s HP reaches 0, he or she will ask all the other players whether they can give him or her some [Peach]. If no one responses to that, this player will die and be removed from the game.

Equipment card can also be divided into four sorts: weapon, armour, “+1 horse” and “-1 horse”. One player can only have one weapon, one armour, one “+1 horse” and one “-1 horse” at the same time. A weapon can both makes a player’s attack range further and has a special effect when some event is triggered. Armours can protect a player against the attack to some degree. “+1 horse” makes a player “far away” from other players so that other

players will find it is more difficult to attack him or her, while “-1 horse” makes a player’s attack range larger. In other words, “+1 horse” helps about defence and “-1 horse” helps about attack.

Skill card is a kind of card which can cause some special event like discard other player’s card; get a card from other player’s hand; ask all the players to play an [Attack] card in turns; all players recover one HP and so on. Each effect triggered by a skill card can be avoided by another special skill card called [Flawless Defence], the effect of [Flawless Defence] can also be avoid by another [Flawless Defence], too.

All in all, “Kingdom of Warcraft” is a board game which is suitable for all kinds of people above 12. One round of game will take about 10 to 30 minutes. The minimum player amount is 2, and the maximum number will be 10 or 11. Playing Kingdom of Warcraft is a good way to relax yourself in your spare time.

Chapter 3 Initial application architecture

Before doing this project, I decided this application to be a Java program because Java is cross-platform and easy to handle. Netbeans 6.7.1 helps to generate some UML diagrams to document the system in the design stage. The IDE used to develop this application is Eclipse 3.7, which is an open source software supported by the eclipse foundation. Since it is a multiple-players online game, this application is consist of both client-side and server-side programming. To communicate between the client and server, I prefer to use Java socket and object serialization rather than the complex Java RMI. The object serialized and transferred between client and server is called “Command”. A third-party jar lib file called mp3spy is used to play sounds. The server will be built when a player creates a game waiting hall, so there is no need for a user to run another server application before the game starts.

Chapter 4: Software Design

Before implementing this application with Java code, it is necessary and reasonable to do some design ahead. The UML, which is widely adopted as a modelling language works here to help to do the software design.

To document the requirements listed above and descript the behaviour that is required of the system, the following Use Case diagram will be used:

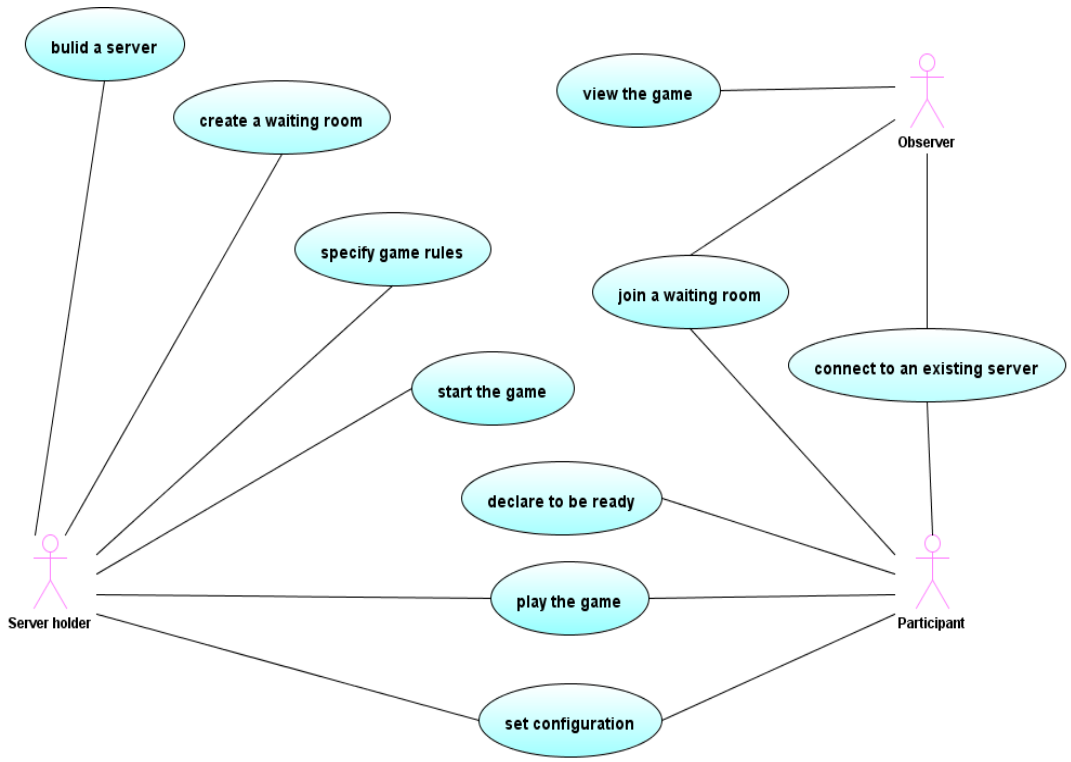

Fig 1.1: Use case diagram
(Generated by NetBeans 6.7.1)

There are three kinds of actors take part in this application: server holder, participant and observer. The server holder is the player who builds up a server. This player obtains the power to set up a specific game rule. The participant needs to connect to an existing sever by knowing its IP address and port number. This player can indicate whether he or she is ready to begin the game then the server holder can start the game after being told that everyone is ready. What an observer could do is just connecting to the server and watching the game, no else functionalities are implemented for them.

Each use case listed in the diagram above is corresponded to a requirement. “Connect to an existing server” and “Join a waiting room” are two use cases shared by a participant and observer, similarly both the server holder and participant can set their own configurations for this application.

The following UML class diagram is used to model the relationships between classes in this application at the design stage:

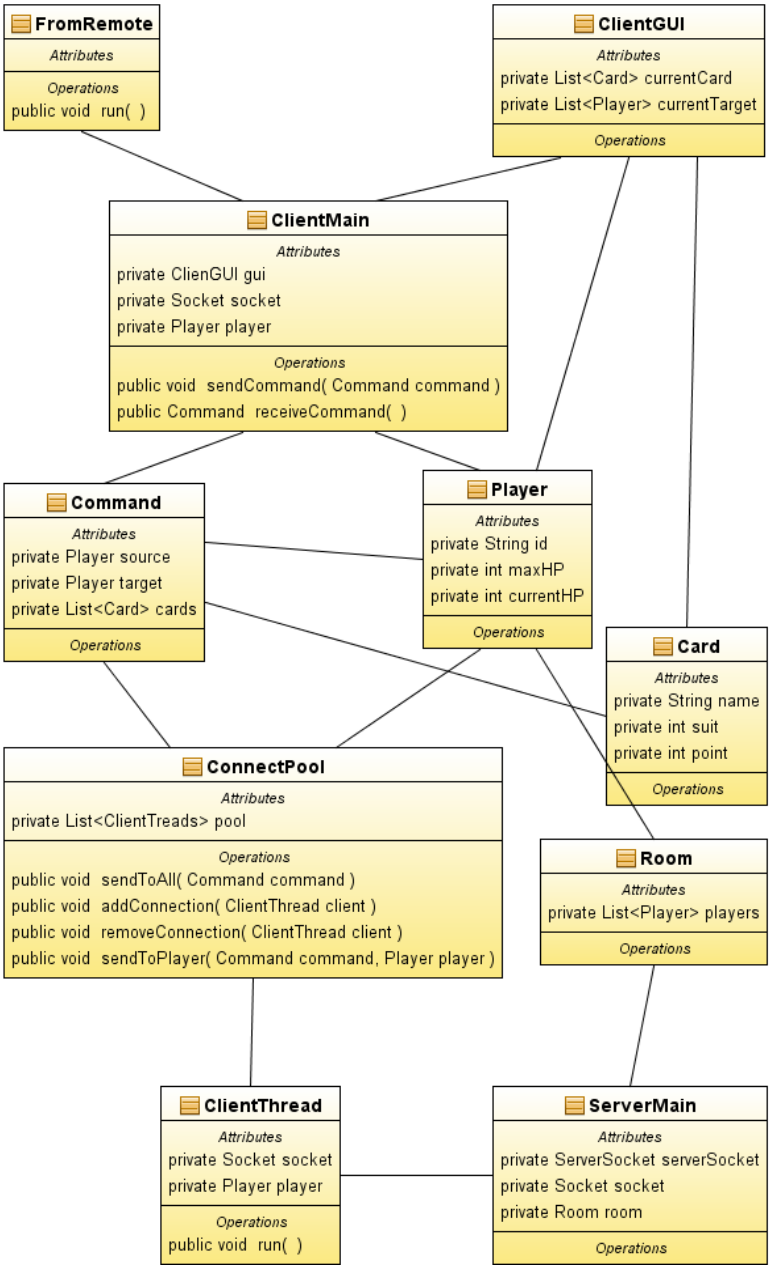

Fig 1.2: Class diagram

(Generated by NetBeans 6.7.1)

CRC cards below specify the responsibilities and collaborators for each class appears in the diagram above:

Class **ClientMain**	
Responsibility	Collaborators
The entrance of the application on the client side Keep a socket connection to the server	FromRemote ClientGUI Command Player

Class **FromRemote**	
Responsibility	Collaborators
Extends the Java Thread class Keep running continuously to receive commands from the server until the application terminates	ClientMain

Class **ClientGUI**	
Responsibility	Collaborators
The graphic user interface presented before a player Keep a copy of the current card and target player a client has chosen	ClientMain Player Card

Class **Command**	
Responsibility	Collaborators
An entity class implements the java.io.Serializable Transmitted between the client and server representing the event happens	ClientMain Player Card ConnectPool

Class **Player**	
Responsibility	Collaborators
An entity class implements the java.io.Serializable Transmitted between the client and server representing the source and target of an event happening Store the max and current HP	ClientMain Command ClientGUI ConnectPool Room

Class **Card**

Responsibility	Collaborators
An entity class implements the java.io.Serializable Transmitted between the client and server representing a card with a specified name, suit and point	Command ClientGUI

Class **ServerMain**	
Responsibility	Collaborators
The entrance of the application on the server side Build a ServerSocket first then give each client a different thread after a successful connection then accept for another	Room ClientThread

Class **ClientThread**	
Responsibility	Collaborators
A thread which is given by the server to a client after a successful connection Keep receiving commands from the client until the server is shut down	ServerMain ConnectPool

Class **ConnectPool**	
Responsibility	Collaborators
Hold all the ClientThread running at the server side Enable the server to broadcast some command to all the players or a specified player	ClientThread Command Player

Class **Room**	
Responsibility	Collaborators
Keep a copy of all the player objects in a list at the server side	Player ServerMain

Fig 1.3: CRC cards

These classes construct the skeleton of this application. More and more classes will be added when doing the implementation, all based on these.

Chapter 5 Software implementation

5.1 Start point: Graphic user interface

After the design process, I start to do the implementation part from the user interface coding. Graphic user interface plays a very important role in all kinds of game applications so developers should do their best make it as much beautiful as possible in order to catch players' eyes.

At the very beginning, I just put all the GUI components, all code into only one class, including the action listener for the buttons. This immature idea caused the code length reached more than 1,000 lines. Such a big class is a significant sign of bad programming and also makes the further development more difficult. Code refactoring needed to be done urgently before taking any further steps. To divide this huge class into separate parts, it is a good way to divide the GUI panel first then the whole game panel is separated. Each of different parts can be extracted into a separate class; the action listener for a corresponding component can be also token out from the huge class. There is also some information needs to be stored in the game GUI, such as the current cards a player has chosen to play, the current targets a player has specified so two array lists are used to maintain them.

There are also some components need to be handled specially. CardButton is a GUI component used to show a card. Its appearance will change according to different event triggered by the mouse cursor so the CardButton class needs to implement the interface MouseListener. When initialising a CardButton, buffered image and graphics 2D works together. Graphics 2D draws the image of the card first; then draws the suit and point separately so that I will not need that many card images (otherwise 160 different images are needed, causing the program much bigger and slower). When a mouse cursor enters a CardButton, the card is required to become lighter than normal in this application then the filter method of a RescaleOp object is called to make this card outstanding. After some testing I found it was not a good idea because a lot of time and CPU resource were cost to do this filter .Every time a cursor enters the CardButton, filter method is invoked once. Instead, I did the image filter just once in the constructor then stored this image in a field. Now the situation is each time a curser enters, the system will just show a different image. This idea performs much satisfactory. The

location of a CardButton is required to change when a player clicks on it. This card will go up or down according to its current position. An up or down method is called to set the card's location to a higher or a lower place then a player can know which card he or she decides to play and whether he or she has changed the mind to use another card.

A RollBlank represents a player in the current game and RollSelf represents the player self. They both have some same fields and methods so a super abstract class called RollComponent comes out. Both RollBlank and RollSelf inherit from this class. Some common fields are placed in the super class to avoid code duplication. Firstly, there must be a player object stored in the class to identify which player this component represents. Some other properties such as whether this player is the source or a target of an event, whether this roll component can be clicked to be a target of some events are also stored in the super class. From the GUI perspective, they both need to contain an image which represents the general a player has chosen, the current HP in a graphic way. To do this, a class called RollHeadComponent is created. Similar with how to deal with the CardButton, the image of the general and HP are drawn separately. A coloured frame is added to the image of the general with the help of a ColorCovertOp object when some event happens. A red frame means this player is the target, a blue frame means this player is the source, and an orange blank means the player is doing his or her round at the moment. Also, there is a very important component needs to be specified separately: the time bar.

Before the game starts, the player who builds up the server needs to specify a time within which a player must take some action. A time bar needs to be shown when a player is asked to do something. For example, when a player is doing his round, he or she must play the card within this time limit. Each time bar contains an over-time handler in it. The over-time handler works when time is up. Imagine a player is being asked whether to use a [Miss] card after he or she becomes the target of an [Attack] card, but he or she has already leaves the sit so no response can be given. When the time is up, the over-time handler will tell the server this player decides not to use such a card and then be hurt automatically. A time bar should be a thread running independently from the main program. Since a time bar component has inherited from JProgressBar already, so it must implements the Runnable interface, which is another way of implementing Thread that Java gives us. What does in the run method is keeping counting the seconds since a time bar component is shown to a player. If the time limit reaches then the handle method of the over-time handler object will be called.

At this point, this project comes out with such a simple user interface. However, what a player does with this interface needs to be transmitted to the

server. Nothing happens when a user clicks on the OK and Cancel buttons at this moment.

5.2 Make this application work: Client – Server communication

Server plays an important role in all kinds of online games. The efficiency of the server usually determines the quality of the whole game. As I have already mentioned, in this game a server is built up when a player creates a waiting hall. Java socket is used to deal with the communication between client and server in this project so a server socket object is created when initialising a server. Then the server will loop forever to accept the connection from other clients until the server is shut down. Each connection is given an own thread. A class called ConnectPool helps to keep and maintain all these threads. What this thread does is also looping forever to receive the command transmitted from the client side, deal with each command then continue to read next one.

On the client side, for a participant, after the user types the IP address and port number of the existing server, the underlying Java socket facilities will connect to such a server program then a thread called FromRemote is told to start. Similar with the thread in the server, this thread keeps receiving command coming from the server, display this command in a graphic way to the player.

There is some information of the current game needs to be stored on the server side, which is all included in the Room class. This class keeps a copy of all the player instances in the current game in an array list. Player object is used for representing a real player in this game and will be transmitted between client and server to identify the source and targets of an action. The player object here must be consistent with the one on the client side. The next player to do its round after one player finishes can also be gained from this class. Other information such as the last command received by the server, the configuration of the current game is also kept by this Room class.

After a successful connection, what to be considered next is how to deal with the command transmitted between the client and server. At the very beginning, since the project is just built and rather small, I choose to use a String defined in a specific format to represent a command. For example, “player1#attack#player2#using#heart@3@attack” means player 1 uses an [Attack] card whose suit is heart and point is 3 to attack player 2. At first, this String command performs not bad. However, as command gets more and more complex, it becomes harder and harder to parse a command since the command is also getting longer and longer. A more efficient way needs to be used to replace this complex String command; then the idea of using object

serialization comes into my mind.

It is much better to represent a command using object rather than a long String. The reason why I choose String at first is that String can provide a mechanism of encapsulation. However, object in an object-oriented language is born with the feature of encapsulation. Take the String command above as example, instead of making such a long String; I just create a class called “AttackCommand”, which contains three fields: the source, the target and the card. Client will send such a command after a player has chosen an [Attack] card; a target player then clicks the “OK” button using the object output stream contained in the FromRemote class. After the server receives this command, it will add the [Attack] card into the waste cards pile; resend this command back to all the players. What a client needs to do after getting this command is to show this [Attack] card, show the source and target all in a graphic way and then asks the target player whether he or she wants use a [Miss] card. This flow is very efficient and clear and easy to be understood compared to the String one. All the actions such as a player players a card, enters into another round phase, uses a skill and so on, will be transmitted to the server and back to all the clients to display. There are over one hundred different commands in this application. At the starting point, when there are not so many commands, both client and server need to judge the command using the Java keyword “instanceof” and a lot of “if” statements after they receive a command. This method turns out to be a bad design idea after the amount of commands increasing significantly. Instead, another feature of object-oriented programming gives me a way to handle this -- polymorphism. An abstract super class of all the command class is extracted, named AbstractCommand. Two methods are defined in this class, handleForClient and handleForServer. Since all the command classes extend this super class, they all must give a specific way to implement these two methods. handleForClient deals with what the client need to do after it receives such a command, and handleForServer is invoked when a command is received by a server. So hundreds of if statements can be replaced by just one line of code: command.handleForServer() or command.handleForClient().

At this point, this application can works with some basic command, such as the use of card [Attack], [Miss], and [Peach] to recover 1 HP for the player itself.

5.3 Difficulty: Round Event

The logic of the game is far more complex than the basic actions listed above. There is a typical kind of game flow called round event needs to be handled differently. For example, a player can use a card called [Arrow Rain] which requires all other players to play a [Miss] card; otherwise the target player will

receive 1-point damage. When a player's HP reaches 0, he or she must ask all other players whether to use some [Peach] cards to him or her then the system will judge whether this player will die or not. All these situations refer to a flow like "ask all the other players to take some action then do something". At first, all these events are handled separately. However, after copy some same piece of code again and again, it makes more sense to find a good algorithm to deal with this situation as a whole.

The effect of one round event may be influenced by another. Imagine a player uses a card [Arrow Rain], one target player has no card in hand and his HP is 1. Since he cannot play a [Miss] card, no doubt he will receive 1-point damage and dying. At this point, the round event [Arrow Rain] is interrupted, another round event, which is asking for [Peach] from other players must be done first. The round event [Arrow Rain] will continue after this asking round event finished. Obviously, there must be a container to hold all round events taking place at a moment. Compared to other Java collections, stack is a very suitable data structure to do this. Later events will be handled and removed first. So, on the server side, in the Room class, a stack is build to hold all the round events taking place at one time. What stored in this stack is an abstract model of the round event. A RoundEvent object simply contains which command causes this event, and the players involved in this event. The command contained here should be a special kind of command, which needs to specify three kinds of actions: what to do next after one player responses to this event, what action the server should take if one player cannot response to this event, and what to do next after this event finishes. The best way to guarantee all these class to implement these three methods is to make a Java interface; all command which will cause a round event must implement it. MultipleCommand is such an interface.

Also take the situation above as an example, the new handling method is: when a player has chosen to use a card [Arrow Rain] and clicks the OK button, an AskArrowRainCommand will be transmitted to the server. Just as I have already mentioned in part 5.2, server will invoke the handleForServer() method of AskArrowRainCommand class. What server will do is add this [Arrow Rain] card into the waste card pile, get all the other players indexed from the source player as the targets of this event from the Room class on the server side. Then the server will make a round event, put this command and the target players into this round event, push this event into the Stack in Room Class, send this command to all the clients to display, then starts to handle this round event by calling the continueCommand() method of this command class. The continueCommand() method will check whether this round event has finished yet first. If so, the finishCommand() method is invoked to finish this round event, else it will get the next player who needs to response to this round event and send an ArrowRainCommand to him or her. Mind the

difference between the AskArrowRainCommand and ArrowRainCommand. AskArrowRainCommand is the command which is transmitted from one client to the server and then sent back from the server to all the clients. This command will cause a round event. However, ArrowRainCommand is sent by the server to the client who is being asked to response to the round event: play a [Miss] card in this example. Imagine this target player cannot play a [Miss] card, an object of NoActionCommand class will be transmitted to the server representing that this player cannot response to this event. The server will get the command which is contained in the RoundEvent stored at the top of the stack, invoke the handleNoAction() method of this command. In this situation, this command is AskArrowRainCommand obviously since we just push only this event into the stack. What the handleNoAction() method does in AskArrowRainCommand is subtract 1 from the target player's current HP. If this HP reaches 0, according to the game rule, the application should deal with this dying event before continuing the Arrow Rain event. A new round event object containing a DyingCommand and players to ask for [Peach] is created and pushed onto the top of the stack, then the server calls the continueCommand() method to handle this new event. If some player response to this command, after the receiving this message, the server will add 1 to the dying player's HP, pop the top event from the stack and then continue to do the round event Arrow Rain in this example. If nobody responses, this player will die and be removed from the current game. After that the server will do the same thing, which is popping the current event and continuing to handle the next target player.

After a lot of testing, this flow turns out to serve all events with more than one player as target very well. As I have mentioned in the game description, there is a special card called [Flawless Defence], which is used to avoid the effect of a skill card. So there is one more flow which should be added to the example above: before the sending an ArrowRainCommand to a client, the server should ask all the players whether to use a [Flawless Defence] card at first. Since [Flawless Defence] can be avoided by another [Flawless Defence], the server needs to keep sending AskFlawlessDefenceCommand to all the players until no player decides to response to it. Then the server will count the amount of the responses from the players. If this number is odd, it means the effect of this skill card is not avoided so the target player will still need to take some action. Otherwise this target player does not to do anything. The server will just continue to deal with the next target player.

The handling method of round event is one of the hard bits of this application. The following work to do is to write command for each card a player can play. There are more than 40 kinds of card with extension in this game, some commands are not that easy to handle and imports a lot of new concepts. Generally speaking, writing these commands is the main job of doing this

project and takes up the most developing time, nearly one and a half month.

5.4 Advanced technique: The observer pattern

After finishing all the commands, the main functionalities of this game is implemented and it works very well. The next task needs to be done is code refactoring. There are a lot of implementing method can be simplified by using some more advanced Java technique. For example, after a player is hurt, he or she will lose one HP and the graphical user interface should present this change by repaint the corresponding roll blank. At this point, after a client receives the command indicating that one player is hurt, the client will find this player's roll blank, subtract one from this player's HP and repaint this component. It seems that there is some relationship existing between the HP of player and the graphic presentation of this amount in the GUI. They should keep consistent with each other at all times. This situation is a typical example when the observer pattern should be used. The observer pattern is a kind of design patterns used when a change to one object affects other objects. In this example, the Player class is observable and the RollBlank class is an observer. After a player changes its HP, it should tell the corresponding observer RollBlank to repaint. Java has a facility to deal with the observer pattern very well -- a class called PropertyChangeSupport in package java.beans. To use this framework, the observable class, Player in this example, will contains an object of PropertyChangeSupport as a field. In the setter method for the HP, except for just change the HP amount, this method also needs to "tell" the observers this change has happened. It is done by called the firePropertyChange method of the PropertyChangeSupport object. Which change is happened, the old value and the new value is specified in the parameter list. On the other side, the observer, RollBlank class here, must implement an interface called PropertyChangeListener. Since the corresponding Player object is stored in this class, the method addPropertyChangeListener method is called on the PropertyChangeSupport object held in the Player object in RollBlank class's constructor to build such an observable – observer relationship. This RollBlank class should implement a method called propertyChange specified in the interface PropertyChangeListener. What goes in this method is the action to be taken after this observer has received the message about a change has happened. In this example this method will check the PropertyChangeEvent name first, which is the first argument of the firePropertyChange method. If this name shows this change is a HP change, the RollBlank class will repaint. Different changes can be also handled with this kind of pattern.

5.5 Utility tools

In this part, I will discuss about three tool classes used in this application:

Strategy, CardFilter and OfflineHandler.

According to the game rule specification, a player can only attack another only in his or her attack range. So after a player clicks an [Attack] card, the application should enable all the RoleBlanks whose player is in this player's attack range and keep others disabled. After the player chooses a target, the client should keeps this player in somewhere and transmit it to the server when this player clicks the OK button. All these functionality is implemented in the handler class of CardButton. However, there must be a lot of if statements to judge which kind of card the player clicks since different handling method will be applied to different card. This makes the code in the handler class so long and hard to be maintained. An alternative way of handling this comes into my mind after my consideration. In this method, writing a strategy for each kind of card. The word strategy here has nothing relationship with the famous Strategy pattern. All these strategies must implement two methods: what the GUI needs to do after a user clicks a card and then a RollBlank. Inheritance is used here obviously. All these strategy classes extend a super class called AbstractStrategy which specifies these two methods as abstract methods named prepare() and handleClick(). For example, if a player clicks an [Attack] card, a corresponding AttackStrategy is loaded onto the game panel and the method prepare is called. The GUI will enable all the RollBlanks which are in this play's attack range. Method handleClick() is invoked after a player clicks a RollBlank as target. The client will store this specified player in the array list stored in game panel waiting for being transmitted to the server. This method helps to get rid of all if statements in the CardButtonHandler class and makes the code more clearly.

A player can just use some specified card at a time. For example, a player cannot play a [Peach] card during his playing phase when his or her HP is full. A player can only play a [Miss] card when he or she becomes the target of [Attack] and is asked for response normally. It is necessary for this application to prevent a player to play an unreasonable card when having this game. Just as the strategy mentioned above, another tool class called CardFilter helps to deal with this problem. The GUI will hold a corresponding CardFilter at different time point. When a player is not doing his or her round, an object of OutOfRoundFilter will work to this application to disable all the card buttons so the player cannot click any card buttons. When he or she is asked to play a [Miss] card, an object of MissCardFilter will work to enable all [Miss] cards and keeps all the other cards disabled. Sometimes, if a player has already chosen a card then clicks another, different actions will be taken under different circumstance. If this player is doing a playing phase the former should go down which means this player does not want to use this card any longer. However, if this player is discarding excess cards in the discarding phase, the client should keep both these two cards up then discards them after the player

clicks the OK button. To conclude, as a CardFilter, it must have two methods. One is to filter out the specific kind of card, another is the action taken by the application after a player clicks a card button. To do this, all CardFilter classes inherit from an abstract super class, AbstractFilter, which specifies these two abstract methods for subclasses to implement.

A player may turn to be off-line during a game due to something wrong with the network connection. When an off-line player is asked to take some action, since he or she cannot send any command back to so the server will wait forever then causing this application crashed. To be a robust application, there must be some handling method for this situation. Another tool class OfflineHandler is imported to deal with this problem. Each time a player is asked for response, an object of OfflineHandler will be created then the handle method is called with this player object as parameter. If this player is not off-line, this handler will do nothing, otherwise this handler will take some action according to the event happens. In further development, the off-line player will get some penalty if he or she just closes the game window and leave other team members ignored. However, it is still necessary for this application to be robust.

All the game logic and code refactoring is done at this point. However, the game will start at the same time when a player opens this application. It is unreasonable obviously. The next part needs to be done is to build a waiting hall then start this game after all the players have joined this game.

5.6 Before game starts: Waiting Hall

This part has nothing relationship with this game. The waiting hall may be a universal part of all the card and chess online game. Everyone who has experience about these online games knows that if he or she wants to play a round of game, the first thing to do is either to create a room and waiting for other players to come or just search for an existing room and join in. A participant player will have a button called “Ready”. The room holder has a button called “Start” which will be enabled after all the other players have clicked their “Ready” button. In this application it is almost the same. The only one difference may be there is only one room can be created by the server since this game is small in scale at this point. Instead of starting the game immediately, a welcome page will be displayed after a client runs this application. Player can choose whether to build a new server, connect to an existing server, do some configuration or view the details about all cards and generals. The first and second core functionalities will be discussed in this part and the others will be developed in future.

After a player clicks the “Create” button, a dialog box will come out asking this

player to enter a name and the port number used for network connection. The name player enters should be unique since this application will use this name to identify a player. This game also allows the player to choose a favourite show which will be displayed when wandering in the waiting hall. A game server will be built after clicking the OK button and this player is in a waiting room already since I have mentioned that only one room will be served by a server. Since this player is the room holder, he or she will have the power to specify the rule of this game by using the Setting button. Rule here means the time limit to take an action, which game rule (Survival or Role) is going to be used, whether players can chat during this game and so on. The original maximum player amount for doing one round of game is 8. The room holder can reduce this amount by clicks the “Close” button in each empty space. If this room is full, other players cannot join any more.

To join a game by connecting to a server, The “Join” button should be clicked. User will be asked to enter a unique name, the IP and port number of the existing server. The participant player cannot change the rule settings in this room. All what he or she could do is to click the “Ready” saying that this player is ready to begin a game. A ReadyCommand will be sent to the server then back to all the clients using the command structure elaborated in part 5.2. Other participants will sign this player with a mark indicating he or she is ready. The room holder needs do more jobs. It should check whether all the participants have indicated to be ready. If so, the “Start” button in room holder’s GUI will be enabled to start a round of game.

Besides the room holder and participant, there is another kind of player called observer. A player can turn to be an observer by choose the “observer” radio button in the joining room dialog box. An observer has no button to click. The game does not need to ensure the observer is ready before starts. An observer can just see the cards player plays, the event happens, nothing else special he or she can do. It is a good way to give a tutorial for some new players to learn how to play this game by watching other players’ competition.

To terminate this game and tell the winners and losers, some judgements should be made after a player kills another. Since different game rule will have a different way to judge, so at the beginning of the game, a corresponding Rule object is loaded into this round of game according to the game configuration set by the room holder. This Rule class must implements three core methods. The first one is invoked after the game starts. For example, in the Role rule, each player will be allocated a role to play. This is done in this method called start. The event may cause this game finish is someone is killed. At this time, the judge method is called to check whether this player’s death can lead this game to terminate. If so this game will finish at once, the server will tell the winners and losers apart then send the result to all the client,

the client will show this result on a panel, otherwise just continue as normal. The killer may get some award or punishment from this action so the kill method decides whether this killer can get or lose some cards.

If one round of the game finishes, no one will prefer to close this application and build the server and join in one more time. A reasonable handing method should let all these players go back to the waiting hall. Since they may return at different time interval, the client needs to get the information about all these players from the server. It is possible that some players are ready; someone has already leaves the waiting hall. A hand-shake communication with server is done to get the right state of each player and then show it properly.

Since the server is build up by the game holder, if this player turns to be off-line, the server will stop working then causes this application crashed. So the room holder is required to have a stable network connection to confirm the running of this application.

The main body of this project is finished after all these functionalities is implemented. However, to be a good application it should be easy to maintain and extend. The rest task is to add some extension to this game, including some new cards, the effect of a weapon, and finally the generals.

5.7 Finishing point: Extensions

The first part of extension is about adding some new cards. This is easy to be implemented under the existing game structure since they are just some new cards as the previous ones. The functionality of each card is achieved by some specific commands transmitted between client and server, the same as the flow talked above.

One feature which makes this game more interesting and challenging is that a player can pick up a general to act as at the beginning of the game. Different generals will have different skills which makes the battle more exciting. As I have mentioned in Chapter 2, each weapon also has its own skills. The skill of a weapon and a general may seems totally different in a player's view, however, they are the same thing in programming to some degree. The player will get a skill after he or she has equipped weapon, lose it when this weapon disappears. The skill of a general is just a special skill which will be held by the player permanently – that is the only difference. So, the final task for this project, also the second hard point is to implement the handling method for skills.

Imagine that a player gets armour "BG" which has the following function: "each time when you are asked to play a [Miss] card, you can do a judge. If

the suit of judge card is heart or diamond, it is regarded as you have played a [Miss] card“. Two classes will help a player to get this skill: BG and BGSkill. An object of BGSkill class is stored in when a BG object is created. After the player equips such armour, a BG object is stored in the Player class and the BGSkill object is taken out then adds it into the skill list in the Player class. The skill list will keep all the skills a player gets at the moment.

Skills are triggered by different events. In this example, the events which can trigger this skill is that this player becomes the target of [Attack], [Thunder Attack], [Fire Attack] or [Arrow Rain] ([Thunder Attack] and [Fire Attack] are two kinds of extension cards). Class BGSkill must implement two methods: one will return a result telling whether this skill is triggered by a specified event, the other is the handling method for this skill. For example, a player equipped with BG is under attack, the server needs to handle skill first before telling this target player to play a [Miss] card as the specification. The server will get the skill list of this player out, iterate through each skill object to check whether a skill is triggered by calling the isTriggered() method of each skill object. There will be a circumstance when two different skills will both be triggered so the first skill needed to be handled should be given high priority according to the game rule. If no skill is triggered the handling method will be the same as normal, otherwise the handle method in the skill object will be invoked to handle this skill. A Boolean result will be returned by this method. True stands for the server should stop to handle this method and not executing the code below any longer, false means the not. Since skills can be triggered by a lot of different events, this flow is done at every point an event happens.

The skill talked above is called “passive skills”, which is triggered by some event. There is also another kind of skill called “subjective skill” which the player can use. A typical example for subjective skill is that “You can use two cards as an [Attack] to play”. To implement this kind of skill, a good method should be to add a button for each skill which players can click, indicating he or she is using a skill, then a specified strategy and card filter is loaded to filter all the card buttons and roll blanks to help the user take a right action. Corresponding command is also transmitted between the server and clients to implement this skill.

Although skills make the game rich and colourful, it gives programmers huge tasks to do to handle them properly. Owning to the time limitation, only two generals, six skills are implemented. More and more generals, skills can be added to the current game since the software architecture is already mature enough.

5.8 Improve the performance

After the implementation of the functionalities, it is the time to improve the performance of this application. This program will take more than 100M memory after running in general. About 10 seconds are needed before the starting to load all the resources including the pictures and sounds. To improve the performance, it is reasonable to move all these loading work to a separate thread and starts this application more quickly. However, a fatal may happen which is the game frame is initialised before the loading of the resources causes a blank frame with nothing appears to a user.

To deal with this bug, also in order to improve the performance, instead of putting all the resource images under the project root, I choose to zip them into a .dat file. A class called ImageContant works for loading all these images into memory. This loading is done by another thread called initThread which starts after the running of ClientMain. This thread will load all the images, wait for the game frame to be initialised. The frame is set to be visible to a user after both of these jobs are accomplished to avoid the bug specified above. There are also another two threads running in this program: one for display an animation and another for playing sounds. A third-party jar tool mp3spi1.9.5 is used to play sound after some events.

The development of this project terminates here. Generally speaking, nearly all the initial aims have been achieved. All the implementation method is under deeply consideration and improved again and again. Perhaps there will be better handling methods, these updates can be done in the further development.

The use of thread makes the performance much better since the main program does not need to wait before moving the next task. However, thread also imports some problems such as safety and deadlock. What I have done is try my best to avoid these problems and use it carefully.

After the coding part, as a normal flow in a software development model, the next task to be talked about will be testing.

Chapter 6 Software testing

Testing is a vital stage in development of all kinds of projects. The aim for testing is to make sure all the code achieves the objectives and to find potential bugs in the application. Three levels of testing are used in the development of this project: unit testing, integration testing and system testing.

The unit testing is done after a method, or a class is finished. A typical

example for this should be the testing for the game GUI. A temporary test class is created to run this GUI class instead of running the class ClientMain which needs to connect the server. The height, width and layout of the GUI are adjusted under a lot of testing.

Integration testing is done after each milestone marked as the subtitle of Chapter 5. From integration testing, a plenty of bugs is found in the game structure. The handling method of round event talked in Chapter 5.3 is changed and fixed by a large number of integration testing.

The system testing helps to get a whole feel of this application after developed. Unlike some business and web application, there is not a detailed testing data file. The best method to test a game is to play it again and again, try to use a different thinking method, try my best to cause errors to check whether this application can deal with it properly.

Testing should be a separate stage in the plan-driven software development model. However, since this is a game application, I do not leave a stage which is wholly about testing. Testing is done interleaved with the development.

Chapter 7 Discuss and Evaluation

7.1 Evaluation of each chapter

This project has turned out to be challenging in many ways. Each stage has presented its own problem to overcome.

When choosing a suitable game for this application, I have tried more than 10 different board games. However, most of these games are all Chinese which causes a problem of translation. It is not hard to translate them into English literally, but the background, essence of the game will lose to some degree. I also tried an American board game called “BANG” - obviously it is in English, however the specification of this game is not clear enough for me to do programming since I do not know which will come first when two events happens at the same time. I have no choice but to translate a familiar Chinese game into English. Although I do my best for the translation, it is hard for me to guarantee this is the best expression for English.

As advised by the supervisor, I made all the requirements by myself rather than gathering from other people. May be players want a different functionality but I have not provided. All these requirements are the core for this application. I have also included some advanced requirements if time permits; these are turned out to be implemented at last.

The design stage does not take me too much time and quite straightforward. As it is an online game application, it is reasonable to be divided into a client side and server side. The use case diagram is just a way of documenting the requirements, and class diagram shows the basic underlying structure of this project. The workflows are all in my mind because I have played such a game for many times so I do not choose to draw the activity diagram. It is unnecessary and all the diagrams will get very long if I do that.

Before doing the graphic user interface, I came up with a nightmare – where do the resources come from? Since it is an individual project, I did all the photoshop work although I am not familiar with PS. Some of the images come from the Internet some are drawn by me in this project. This may cause this application looks not perfect enough. GUI programming is not hard for me, so everything goes smoothly. A refactoring is done after finish making the code more clearly.

The handling method for command is improved from using Strings to object just as I have mentioned above. The new idea comes into my mind quite straightforward. I am familiar with Java so I know that Java has the facility to deal with this problem. Writing the first command is a little hard; it needs a lot of testing to ensure it works properly. The rest work becomes much easier after the first command is done. Although there are nearly 100 different commands, they all share a same structure.

The first hard point in this project is the round event. I cannot find an existing Java structure which handles this very well. Before coming up with the final idea, I tried some other structures but none of them performs well enough. It takes me about a whole week to get the final algorithm. This new architecture may be the most valuable part of this application and also gives a new way of thinking for other programmers.

The topic of the observer pattern is covered in module 6COM0277, Further Object-Oriented Development A of the University of Herfordshire which I have enrolled in Semester A. I find it is a very useful pattern and decide to use it in my project. I choose to use the PropertyChangeSupport rather than the basic Observable and Observer. This is because PropertyChangeSupport is also provided by Java and much easier to handle. I have tried some tutorials online for this before applying it into this project. It gives me a further understanding of the observer pattern.

The tool classes do not exist at first. It is born since this application needs an encapsulate method for doing same kind of works. The construction of the waiting hall is also very easy and straightforward. The other hard bit in this

project is to handle the skills. To tell the truth, I am still not quite satisfied with this structure used at this moment. It is rather complex and hard to be understood. Bugs can happen at everywhere causing the program hard to be maintained. A better handling method should be used for skills in the further development.

7.2 Evaluation of Project Management

The detailed plan of this project is set in January 2012. The Gantt Chart in Appendix is the initial plan. However, this plan does not seem to be a good one since the design phase does not take that long time in fact. Most of the time is spent on the software implementation and testing. The actual work done in each week is contained both in Appendix and weekly report.

Finally, nearly all the initial objectives are reached by the deadline. The most obvious weak point of this application now is that the number of generals is too little: just 2 in total. Normally, according to the game rule, there will not be two same generals appear at the same time in the game; however, this rule cannot be implemented at this moment.

7.3 Further Work

The first thing needs to be done is to add more generals to this application. Just two generals cannot support the whole game. As I have mentioned above, the handling method for skills should be reconsidered. Owing to the feature of this game, there is no limit to extend this game with new kinds of cards, new generals, even new rules. These extensions can be made in the future.

To improve the look and feel, some images with high quality should be used in this game GUI. Instead of publishing a server in this application; the further idea is allow a carrier to build the server, all the players need to connect the central server to have the game. This idea adds commercial value into this game.

Bibliography

David J. Barnes & Michael Kolling, “Objects First with Java - A Practical Introduction using BlueJ”.

Lecture slides from module 5COM0087, Programming And Program Design 2, University of Hertfordshire.

Lecture slides from module 6COM0277, Further Object-Oriented Development A, University of Hertfordshire.

Wikipedia, Board game.

“The killers of the Three Kingdoms – The most popular board game in China“, http://www.sanguosha.com

Baidu Encyclopedia, “BANG!”

PropertyChangeSupport API document: http://docs.oracle.com/javase/6/docs/api/java/beans/PropertyChangeSupport.html

Java™ Platform, Standard Edition 6 API Specification: http://docs.oracle.com/javase/6/docs/api/

Eclipse-indigo 3.7: http://www.eclipse.org/