COMP3004-Ivanhoe

group project for COMP3004 (Object-Oriented Software Engineering)

Table of Contents:

Authors:

Team #13: - Matthew Pepers - Khalil van Alphen

About:

Ivanhoe is a playing card game where players take the role of knights battling in tournaments. Players play numbered weapons and supporter cards to combat each other, while using special action cards to swing the tournament in their favor. The game supports up to five players. As in any turn based card game, players must consider the unknowns of their opponent's hand, the cards in the deck, and think many steps ahead to become victorious. The rules and faq for the game can be found at /doc/rules/.

From the Rules Description:

Take on the role of a knight and join the prestigious tournaments at the king's court. Use your cards to win the jousting competitions, or fight with your sword, axe or morningstar. Rally your squires, gain the support of a maiden, and surprise your opponents. The first player to win four or five different tournaments becomes the overall victor.

This Java project attempts to faithfully recreate the original game as a digital computer game, in which you can connect to other players over a network.

Major Releases:

- Iteration 1
- Final Iteration

Installation / Setup:

- have the project open in Eclipse
- make sure the project is using Java 8 (add it to the build path in preferences)
- navigate to /src/test/resources/
- right-click on the following files, go to 'Build Path' -> 'Add to Build Path':
  - hamcrest-core-1.3.jar
  - junit-4.12.jar
  - log4j-1.2.17.jar
  - miglayout-4.0.jar
  - abbot.jar

Running:

- open a command prompt/terminal
- navigate to Client.jar and Server.jar files
  - for Iteration 1: /doc/iteration1/
  - for Final Iteration: TBA
- run the Server
  - normal game
  - `java -jar Server.jar`
  - real-time Ivanhoe
  - `java -jar Server.jar -r`
- run the Client(s)
  - in GUI Mode
  - `java -jar Client.jar`
  - or Command Line Mode
  - `java -jar Client.jar -c`

Design:

Networking:

Our strategy for networking was to pass lots of serializable objects between the Clients and Server. When the Client is started, a new thread is created for reading user input, another thread for receiving objects from the Server over the socket. When the Server is started, a thread is created for reading user input from the console, another thread is created just to wait for new Clients that are attempting to connect, and a final thread to handle the Server's responsibilities to the game. On both the Client and Server, everything that is receieved is a serializable object, including all Client Actions, Chat messages, and Player and GameState objects, etc.

Overall Architecture:

The game has the following basic components: * Cards * Display cards of a colour and a value * Supporter cards of a value * Action cards with special effects * A main player area, where players create their 'display' or stack of cards * A hand of cards for each player * Single deck of cards that player's draw from * Some tokens to mark when a player wins a tournament

The game is implemented using a Server/Client socket system. Its primary actors are the following entities.

Server:This entity is the core of the game. It holds information on its connected players, the state of the game, and regulates network communication while enforcing game rules.
Client:This entity relays commands from the human player to the server over the network, while simultaneously interpreting information received back from the server into graphical or textual information.
Player:Ivanhoe requires players to operate. These are entities that make decision on the game to effect it in various ways. They can be human or computer.

Patterns:

1. Chain-of-Responsibility:
   - ValidCommand class (ValidCommand UML Class Diagram):
     - the Client and Server each have a list of commands that the user can type in, in Command Line (CLI) Mode
     - each command starts with `/` and a list of commands and their syntax can be viewed by typing `/help`
     - the commands are required to play through the game, with such favourites as `/play [card]` and `/withdraw`
     - it was quickly noticed that when checking for the validity of typed commands (such as their arguments, and whether it was an appropriate time to use that command or not), there were a lot of repetition in what we were checking for, and often many checks would need to happen in a row
     - the Chain-of-Responsibility design pattern allowed us to chain multiple validity checks together, reuse those checks for multiple commands, and return quickly if any one check failed
2. Command:
   - PromptCommand concrete class, CommandInterface, and CommandInvoker class (

ExecuteActionCards UML Class Diagram):

- the Server has a prompt(String,Player,ArrayList) method that prompts a Client, corresponding to a Player, for a choice from the ArrayList
- this is used to get the player's choice for things such as which token they want after winning a purple tournament, but also for Action card choices when they attempt to play an Action card
- we realized that it would be a good design choice to encapsulate the rules of the game involving Action cards within the GameState class, but the GameState does not have knowledge of the Server or the Client (which we believe it shouldn't)
- but we still needed to prompt the Client for choices involving Action cards while they're rules were being executed in the GameState
- the Command design pattern allowed us to invoke a concrete PromptCommand class to run the prompt method on the Server, from within the execute method in the GameState

◦ ClientAI (corresponding UML Class Diagram):

- the ClientAI class runs a Client through AI decisions
- multiple AI can play against each other or against human players
- ClientAI makes use of the Command design pattern by executing the concrete classes: StartTournament, PlayCard, EndTurn, and Withdraw
- each time the AI invokes the concrete classes's methods, it makes a decision to take action or not (based on the AI's skill levels)
- this was an excellent design decision and allowed us to encapsulate all actions and information needed for the AI to make each of it's four major decisions

3. Marker Interface:
   ◦ Action interface, and Actions (corresponding UML Class Diagram):
   - the Marker Interface Pattern allowed us to group all Client actions under one interface which doesn't need to specify much common behaviour
   - the interface enforces a getAction() method to get the name of the Action, but more importantly in enforces all actions extending Serializable
   - allows for the Client and Server to create and send serializable Action objects, which made sending data back and forth extremely easy
4. Proxy Pattern
   ◦ Loading images:
   - Images are loaded into a storage structure when they are first needed. Subsequent requests for the image go through the images proxy first before attempting to load a new picture.

Refactoring:

Since iteration 1, the following has been refactored: - Colour class added: - many classes had a colour, such as DisplayCards, Tournament, Token, etc. - that duplicate code was encapsulated in the new Colour class - ValidCommand class added, and Chain-of-Responsibility pattern: - the list of commands that could be used on the Client and Server grew quite a bit - but the Client and Server verified the commands slightly differently - ValidCommand was added to standardize the checks for valid commands across Client and Server, and removed some duplicate code - Executing Action Cards: - rules for Action card execution were checked on Client and Server and GameState at first -

most of the execution rules were then brought to the GameState class, so that the rules would be separate, and would only need to be checked in one place - GUI and AI were introduced - Prompt objects: - were added to send the Client their choices - removed a lot of code were the Client originally made decisions that were better left to the GameState or Server

Pros/Cons:

- Pros
  1.The general design chosen was done so to maximize outward growth. That is, its underlying mechanism are malleable and expandable. For example, any number of additional actions can be added to the server and client with little to no refactoring.
  2.The serialization of objects was a deliberate choice as it decreased the chances of error given two versions, and it allowed for a simple translation of Java objects over a TCP connection. Alternative solutions would likely not have been easier for a project of this scale.
  3.The game's inner workings are abstracted away from the player, allowing us have a secure experience while maintaining a record of responsibility for changes that occur over the course of a game.
  4.Our input/output system allowed us to give options to clients for a GUI mode or a TEXT mode. Since we allowed for varied input and prepared for varied output, we were able to implement this high-impact enhancement with almost zero refactoring.
- Cons
  1.The implementation of action cards required them to be hard-coded in on a per card basis. This obviously means that new action cards would require moderate refactoring and heavy testing before they could be considered implemented. However, given that the number of cards was finite, and there were no plans to add custom cards, this was deemed acceptable, and for the most part, less work overall.
  2.The delegation of the game rules to the GameState class bloated it far more that our team felt was necessary. In retrospect, a more detailed examination of the exact responsibilities of this class perhaps would have organized the code and split its functions, members into subclasses and sibling classes that would have made sense.

Additional Features:

- **AI** - Server can start AI Clients that will connect to the Server and play on their own
  - AI have four skills (which are recorded as numbers from -1 to 1):
  - StartTournament skill
  - PlayCard skill
  - EndTurn skill
  - Withdraw skill
  - Server starts AI Client with:
  - `/ai [StartTournament skill] [PlayCard skill] [EndTurn skill] [Withdraw skill]`
    - eg: `/ai 1 0.85 0 -0.2`

    - OR `/ai [AI Profile Name]`
        - eg: `/ai Max Power`
        - AI Profile Names are the name corresponding to a profile in `aiprofiles.txt` which is loaded when the Server starts
        - to see which profiles are loaded on Server, type `/listai`
        - each line in the `aiprofiles.txt` file is a new profile, and new profiles can be manually added with a text editor
            - syntax is: `[AI Profile Name] [StartTournament skill] [PlayCard skill] [EndTurn skill] [Withdraw skill]`
            - eg: `Max Power 1 1 1 1`
- **Real-Time Ivanhoe**
    - start Server with `java -jar Server.jar -r` (has `-r` argument)
    - there is no turn order, all Players play at once
    - when any Player ends their turn, all Players draw a card from the deck instead
- **Ban / Unban Clients**
    - from Server:
    - `/ban [client ip address]` - adds ip address to `banList.txt`, and prevents that Client from connecting
    - `/pardon [client ip address]` - removes ip address from `banList.txt`
- **Global Chat** - Clients and Server can all talk to each other through messages
    - **Translating** - translate chat to another language (word replacement)
    - Server:
        - `/translate oldEnglish` to translate chat to Old English
        - `/translate none` to stop translating chat
    - Client:
        - CLI Mode:
        - `/translate oldEnglish` to translate chat to Old English
        - `/translate none` to stop translating chat
        - GUI Mode:
        - click the `Translate` button on the Menu Scroll and select option from the dialog box
    - **Censoring** - censors profanity in the chat
    - Server:
        - `/censor` to toggle censoring on/off
    - Client:
        - CLI Mode:
        - `/censor` to toggle censoring on/off
        - GUI Mode:
        - click the `Censor` button on the Menu Scroll to toggle censoring on/off