HYDRA PROJECT

Design

By: Alyssa Gao (20823661)

Table of Contents

1. Introduction

Hydra is a two or more players card game. Each player has two piles (draw pile and discard piles) of cards. On the table, there are one or more piles of card, called heads. Players take turns to play. In each turn, the player takes cards from their draw pile and place them on the heads. If there are two or more heads present, the player can also put one card into reserve and swap it back later when they decide to play it. The goal of each player is to place all their drawn cards on the heads. The first player who has used all their cards wins.

This document describes the design regarding the implementation of the Hydra game.

2. Overview

The project implementation of the Hydra game has employed object-oriented design technique and used a few design patterns. The Single Responsibility Principle (SRP) has been applied in the design to keep low coupling and high cohesion in the UML model. This is described in section 3.

The implementation also allows for future changes and flexibility for new game rules. This is described in section 4.

In section 5, it describes the design and implementation of bonus features.

The document ends with answering questions given in the project specification in section 6.

The project is programmed using the C++ language, and is able to compile and run on Waterloo Linux environment and supports both text-based Linux terminal console and X11 Window. The source code consists of interfaces (.h) files and implementation (.cc) files, along with a Makefile for building the executable, hydra. Two additional libraries are used for building the executable: X11 and Xpm (to support the GUI display feature).

The runtime environment consists of the executable, **hydra**, and an images folder that contains 55 image (.xpm) files.

3. Design

3.1 Object-Oriented Design

The design has employed Object-Oriented design techniques as well as a few design patterns and has kept high cohesion for each class and low coupling between classes. In brief, classes are grouped under five categories:

- **Card category**: consists of classes that represent all types of cards, including regular cards (2-10), Ace, and Joker. It also includes Suit classes as decorators
- **Head category**: consists of a single class, Head, which encapsulates all attributes and operations of a head.
- **Player category**: consists of a single class, Player, which encapsulates all attributes and functionalities of a Player, for both human player and computer player.

- **Display category**: consists of classes that encapsulate functionalities for input and outputs. It has an abstract Display class and two sub-classes: TextDisplay and GUIDisplay
- **Game category**: consists of three sub-groups of classes. The game classes encapsulate the state of the game and implement the logic of the game rules. Additionally, it also includes game creator classes (applying factory pattern) and strategy classes for implementing computer player algorithm.

3.1.1 Card

In this category, a **Card** class represents a regular poker card, Ace, 2 – 10, J – K. It has only one attribute, faceValue.

JokerCard is a subclass of the Card class and represents the Joker card due to its special behaviour (i.e., it doesn't have a face value until announced before placing on a head, or has face value of 2 if it's the first card on a head). In order to handle this special behaviour, visitor classes, **CardProcessor** and its subclass **CardValueProcessor**, are introduced to have overloaded methods to obtain the face value for both the regular cards (Card class) and the Joker card (JokerCard class).

The base Card class does not declare a suit attribute. Suit value is implemented using Decorator pattern through the abstract Suit class and its four subclasses: **Heart**, **Spade**, **Diamond**, **Club**, as decorators.

3.1.2 Head

In this category, there is only one **Head** class. It represents a single head when it's created during the game and is placed on the table. The Head class aggregates Card class, which represents the cards placed on the head. It also encapsulates all head related operations: placeCard(), getTopCard(), getSize(), removeCardFromHead().

3.1.3 Player

In this category, there is only one **Player** class defined. It encapsulates all following required attributes and public methods for a single player (for simplicity, accessors and mutators are not listed):

- Attributes:
    - int playerId;
    - std::vector<std::shared_ptr<Card>> drawPile;
    - std::vector<std::shared_ptr<Card>> discardPile;
    - std::shared_ptr<Card> reserveCard = nullptr;
    - std::shared_ptr<Card> cardInHand;
    - bool inTurn = false;
    - int numOfRemainingCards = 0;
    - PlayerStatus status = PlayerStatus::HumanPlayer;
- Methods:
    - bool hasWon() const;
    - std::shared_ptr<Card> takeCard();
    - int getDrawPileSize() const noexcept;
    - int getDiscardPileSize() const noexcept;

- void swapCardReserveCard(std::shared_ptr<Card> & card);
- void placeCardBackToDrawPile(std::shared_ptr<Card> card);
- void addToDiscardPile(std::shared_ptr<Card> card);
- void endTurn();

Instead of implementing an additional class for computer player, the Player class defines a status field to indicate whether it's a human or computer player. Computer player strategies are implemented by **GameStrategy** class and its subclasses and is used by the **Game** class (see section 3.1.5 Game Classes)

3.1.4 Display

The display classes are responsible for displaying outputs (e.g., headers and players' information) and taking inputs from users.

Having applied the Observer design pattern, the abstract **GameDisplay** class acts as an observer and implements the Observer::notify() method which will be called by Game class each time when there are changes in the Game object. Upon the notification, the GameDisplay class then invokes Game::getGameInfo() method to "pull" the game's state. The GameDisplay class aggregates the **GameInfo** struct, to store game state for display purposes.

The GameDisplay class has declared the following input and output methods as pure virtual methods. They're implemented by its subclasses, TextDisplay and GUIDisplay, to display outputs and take inputs in text mode and GUI mode. They're also responsible for basic validation, such as a valid integer. The validation related to game rules are implemented in the Game class.

- Input methods:
  - virtual int inputNumPlayers() = 0;
  - virtual void inputToAcceptATurn(int, bool) = 0;
  - virtual int inputHeadId(int, std::string, bool, int) = 0;
  - virtual int inputJokerValue(int) = 0;
  - virtual std::string promptForCardValue() = 0;
  - virtual void acknowledge (const std::string) const = 0;
- Output methods:
  - virtual void displayHeads() const noexcept = 0;
  - virtual void displayPlayers() const noexcept = 0;
  - virtual void displayWinner() const noexcept = 0;
  - virtual void displayMessage(const std::string) const = 0;

The **TextDisplay** class reads input from std:cin and output to std:cout and std::cerr.

As part of the bonus feature, the **GUIDisplay** class implements GUI features using X11 Library. The fundamental methods are implemented in the **X11Window** class, which has declared and implemented methods for displaying window, drawing/filling rectangles, drawing strings, managing colours, displaying images. The GUIDisplay inherits both X11Window class and GameDisplay class in order to implement game input/output functions using X11 Library functions. Additionally, if images are unable to load properly, it uses pixels to draw cards. These pixels are defined in the PokerCardPixelLibrary (contains all pixels and lookup methods) and PokerCardPixel (represents a single pixel, i.e., width, height, colour) classes. By using these two classes, the GUIDisplay class is able to draw a card. For example, for 2H, the

GUIDisplay::drawImageIfNotExist() method will retrieve the pixels for “2” and “Heart”, and draw those pixels (retrieved from the PokerCardPixelLibrary) on the screen.

3.1.5 Game

This category consists of three parts:

- **Game** and **HydraGame** classes
- **GameCreator** and **HydraGameCreator** classes
- **GameStrategy**, **BeginnerComputerPlayingStrategy** and **AdvancedComputerPlayingStrategy** classes

The **Game** class encapsulates all game state and operations for managing the game process. It has composition relationships with the following classes:

- Card class (and its subclasses): Game class generates cards required for a single game run (54 cards x number of players). When allocating cards to players and placing cards on heads, pointers to Card objects are passed. Using smart pointers and STL containers simplifies the memory allocation and release.
- Head class: Game class manages creation and removal of heads during the game.
- Player class: Game class creates Player objects based on user input and manages them as game runs.

The Game class aggregates the **GameStrategy** class to engage computer player strategy implemented using Strategy design pattern. An instance of GameStrategy class (or its subclass) can be passed via an optional parameter to Game class’s constructor (from user input through command-line argument). When a player quits, if a GameStrategy object is set, the game will call the GameStrategy object to obtain inputs generated by the strategy class.

Testing mode is implemented in Game::inputCardValueForTesting() method by defining an attribute in the **GameOptions** struct (GameOptions::testingMode) and passed to Game class through its constructor. When the testing mode is enabled, each time when game takes a card from the player’s draw pile, the user is asked to input the card and suit value and replaces the card value taken from player’s draw pile. (Testing mode also works in GUI display mode)

Additionally, Observer design pattern is applied in design. The Game class, which is the subject, inherits the **GameSubject** abstract class. Therefore, it aggregates GameDisplay class. GameDisplay class and its subclasses, TextDisplay and GUIDisplay, are observers and inherit from the abstract Observer class. When its state changes, the Game class notifies its observers.

The abstract Game class defines a public startGame() method for managing the game run process. By applying Template Method design pattern, it declares pure virtual methods for each step of the game run. These methods are called inside Game::startGame() method and implemented by its subclass, HydraGame.

In order to hide the complexity of the creation of the Game object (e.g., whether testing mode is enabled, computer playing strategy is required, Text or GUI display is used), the Factory Method design pattern is used. The abstract **GameCreator** class and its subclass **HydraGameCreator** class are implemented for this purpose. The concrete HydraGameCreator class takes input from its client (main.cc) and a HydraGame object with corresponding options (via the GameOptions struct).

3.2 Design Patterns

3.2.1 Decorator Pattern

The Card class and its subclass, JokerCard class, only have a faceValue attribute for a card. The suit is implemented using Decorator design pattern to decorate base Card and Joker class with suit name, i.e., Heart (H), Spade (S), Diamond (D) and Club (C). This is for outputting the card's face value and name.

As shown in the UML, the abstract Suit class and its concrete subclasses, Heart, Spade, Diamond and Club classes, are decorators. Card and JokerCard classes are base classes.

3.2.2 Visitor Pattern

To handle the special behaviour of the Joker card, Visitor design pattern is used to handle the differences between Joker card and regular cards (i.e., doesn't have face value until announced before placing on a head).

The Card class has declared an accept() method that takes a CardProcessor object (the visitor) as the parameter. The accept() method is overridden by its subclass JokerCard. The abstract CardProcessor class and its concrete CardValueProcessor subclass are the visitor classes, which have overloaded methods to obtain the face value for both regular cards (Card class) and Joker card (JokerCard class). If it's a JokerCard, the CardValueProcessor will throw a JokerCardValueRequiredException, so the game will know and can then ask the player to announce the value for the JokerCard.

3.2.3 Observer Pattern

Observer design pattern is used in the design for Game class to notify display classes for outputting game states (i.e., heads and players information). The Game class and its subclass HydraGame (which inherits the abstract GameSubject class) are subject classes. The abstract GameDisplay class inherits the abstract Observer class, and hence, its subclasses (TextDisplay and GUIDisplay classes) are observers.

Whenever the state of Game class changes (e.g., information of heads, players, current player, etc), GameDisplay class will be notified through GameDisplay::notify() method. Within the method, the GameDisplay class "pulls" the game state through Game::getGameInfo() method to obtain (and store) up-to-date game information for later display.

3.2.4 Factory Method Pattern

In order to hide the complexity of the creation of the Game object, the Factory method design pattern is used in the design for the creation of the Game object. It encapsulates the details of the game creation and hides game configuration away from the client (i.e., main.cc). The GameCreator class is the abstract factory class. HydraGameCreator is the concrete factory class that creates HydraGame object per requirements passed from the client (main.cc). The main program takes input options (such as enable testing mode, enable enhancement, use GUI or Text display, computer player level, etc) from the user (via command-line arguments) and passes it to HydraGame::newGame() method to create an HydraGame object. The main program knows little about the creation of an HydraGame object. HydraGameCreator class takes input options from main program and prepares all dependencies, such as the Display object or GameStrategy object, and creates a HydraGame object for the main program to start the game.

3.2.5 Template Method Pattern

Template method design pattern is used in the design in the implementation of managing and controlling the game rules and logic. The Game class is an abstract class, which defines a startGame() method to manage the overall game running process. Several of the pure virtual methods that are declared in the Game class represent each key step in the overall game cycle/process. They're called inside the Game::startGame() method at certain points within the process. These pure virtual methods are implemented by its subclass HydraGame.

3.2.6 Strategy Pattern

In order to introduce the computer player strategies and algorithm with a certain flexibility and extensibility, the strategy design pattern is used. The Game class declares a variable with the GameStrategy type. An object of one of GameStrategy's subclasses (BeginnerComputerPlayingStrategy or AdvancedComputerPlayingStrategy class) is passed via an optional parameter to the constructor of the Game class when it is created by one of GameCreator's subclass (i.e., HydraGame class in this project) per user input through command-line arguments. When a player quits, if a GameStrategy object is present, the game will call the GameStrategy object to obtain inputs calculated/generated by the strategy class, and use these values to replace inputs that were expected form a human player.

Using Strategy design pattern allows the user to choose from different computer player strategies by specifying the difficulty level through the GameCreator factory.

In this project, two concrete subclasses (BeginnerComputerPlayingStrategy and AdvancedComputerPlayingStrategy classes) of GameStrategy class are implemented to represent two different levels of computer playing strategies. In the future, if more strategies are to be implemented, there will be no impact to the Game class, and very little impact to the HydraGameCreator class.

4. Resilience to Change

Benefiting from applying design patterns and high cohesion/low coupling principles, the design demonstrates resilience to the following possible changes to the program specification:

- Additional card behaviours: the inheritance and visitor design pattern can be extended to accommodate these changes. Additional subclasses of Card class can be created along with different concrete visitor subclasses of CardProcessor class. These changes will be contained inside the card classes and will have very little impact to other parts of the design.
- Additional card features: by adding additional decorator classes can easily implement additional card features, such as the colour of the card. These changes will be contained inside these decorator classes and will have very little impact to other parts of the design.
- Game rule changes: when there are changes to hydra game rule, it can be done by modifying corresponding template methods implemented in the HydraGame class. If the game needs to support different versions of rules, additional subclasses of Game class can be created and let the GameCreator class choose the subclass to create the game per user's options.
- Additional computer player strategies: These can be easily added by implementing additional subclasses of the GameStrategy class and let the GameCreator class be responsible for choosing the concrete strategy subclass per user's options.

- Additional display mode: This can be added by extending the GameDisplay class. As a benefit from the usage of observer pattern and declaration interface (as pure virtual methods) in the abstract GameDisplay class, this will not impact to other parts of the design.

5. Answers to Questions

5.1 Question #1

What sort of design pattern should you use to structure your game classes so that changing the interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.

Answer: I used Template Method design pattern. In the design, I have created an abstract Game class, which implements the high-level logic that controls the overall game process in its startGame() method. The Game class has declared several pure virtual methods representing each key step of the overall game cycle/process. These pure virtual methods are expected to be implemented by its concrete subclasses. In this project, I have created one subclass, HydraGame, which implements these pure virtual methods per the project specification.

In the future, if there is a need to change some rules, it can be done by either modifying a specific method in the HydraGame class or creating another subclass of the Game class. This will not impact the Game class at all. This allows the flexibility of changing of game rules to a great extent.

Additionally, I have used the Factory Method design pattern to hide the creation of the concrete game objects, so the client (main.cc) will only need to specify the type of rules that they like to use and be free of worry of which concrete subclass and configuration that will be used for the game creation.

5.2 Question #2

Jokers have a different behaviour from any other card. How should you structure your card type to support this without special-casing jokers everywhere they're used?

Answer: I used inheritance to implement the card type. I have designed a Card class to represent regular cards (Ace to King) and a subclass JokerCard (inherits from Card). To handle the different behaviour, I use the Visitor pattern and have implemented the CardValueProcessor class as the visitor.

Using the Visitor design pattern, the Game::startGame() method does not have to check the actual type of the card. Instead, it relies on the Visitor (CardValueProcessor class) to figure it out by using double-dispatch technique through its overloaded methods.

5.3 Question #3

If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

Answer: I used the Strategy design pattern to implement different computer playing strategies. I have designed an abstract GameStrategy class, which consists of all pure virtual methods as the interface for required strategies, such as choosing heads, deciding Joker value etc. Different playing strategies will be

implemented by its subclasses, such as BeginnerComputerPlayingStrategy and AdvancedComputerPlayingStrategy which represent two different levels of computer playing strategies. An object of these concreate subclasses is created by HydraGameCreator, the factory class, and passed to Game class during the game creation. During the game process, when a player quits, a computer player takes over. The game will call the GameStrategy object to obtain inputs calculated/generated by the strategy class (through the declared interface in the GameStrategy class).

Additionally, client (main.cc) can specify level of computer player and let HydraGameCreator class (the factory class) to create concrete strategy. If more strategies are implemented, there will be no impact to the Game class, and very little impact to the HydraGameCreator class.

5.4 Question #4

If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

Answer: I declared a status field inside Player class to indicate if a Player is a HumanPlayer (the default) or a ComputerPlayer. During a game, whenever a player decides to quit, this status will be set to ComputerPlayer. As answered in the last question, the computer player's strategies will be implemented in the subclasses of the GameStrategy class. The Game class, which controls the overall game process, will call the strategies at points when it expects an input from a human player, such as choosing a head or announcing the face value of a Joker card.

6. Extra Credit Features

6.1 Smart Pointer and STL containers

All code in the project doesn't explicitly manage its own memory. By using smart pointers and STL containers, no "delete" statement is used and no memory leak is detected during testing.

6.2 Grammar

The project has implemented an algorithm to replace the indefinite article "a" with "an" in front of a word that begins with a vowel sound. It scans a sentence and locates each usage of indefinite article "a" and replace it with "an" if the word follows it begins a vowel sound. It also determines if a word begins with a vowel sound. The main method is grammarCorrection() which is contained in the util.cc.

6.3 Allows Human Players to Quit

Human players can quit during the game. When player is prompted to accept their turn, they're able to input "Q" to quit the game. If computer player is not enabled, the player will be removed from the game entirely. If there is only one player remaining, the game terminates.

Additionally, in the Text display mode, if Ctrl+D is detected during the game, the game will be terminated, and no memory leak will occur.

6.4 Computer Player

The project implements the computer player feature. When a user quits the game, a computer player can take over (if specified through a command-line argument). Two computer playing strategies have been implemented in this project:

- Beginner Level Strategy (in BeginnerComputerPlayingStrategy class): Calculates based on all current heads and chooses head with largest top card value but smaller than the current player's in-hand card. If unavailable, it then chooses a head that has the same top card value as the current player's in-hand card. If still unavailable, it then chooses the first head to trigger a cut-off head. For Joker card, it always announces as Ace. It never uses reserve.
- Advanced Level Strategy (in AdvancedComputerPlayingStrategy class): It plays aggressively or defensively depending on the following situation.
    - Play aggressively (i.e., placing in-hand card on heads with highest top card value) if
        - most of heads have top cards with low face values, and player has more than half cards remaining in a turn, or
        - most of heads have top cards with high face values, and player has more than half cards remaining in a turn
    - Play defensively (placing in-hand card on heads with lowest possible top card value, reserve is used more often) if
        - most of heads have top cards with high face values, and player has less than half cards remaining in a turn, or
        - most of heads have top cards with low face values, but some high value top card (J,Q,K,A) are present on some heads, and player has very few cards (i.e., <= 2 cards) remaining in a turn
    - For announcing Joker card value, if the player has 2 remaining cards in a turn, then announced value is 2, Otherwise, it's Ace.

As a fun bonus, the users can let all players quit after the game starts and let computer players take over. Then watch all computer players to play until the game is over and a winner is generated.

6.5 GUI Display

The GUI display is implemented using X11 Library for displaying shapes, text and images on the window.

The GUI layout can display 8 players and 9x4 heads on the screen. If more players specified at the beginning of the game, and/or more heads created during the game, additional logics are implemented to page up and down other players/heads (in the GUIDisplay:displayHeads() and GUIDisplay::displayPlayers() methods).

The program originally draws pixels for cards (face value and suit), and later switches to using X11 Lib displaying image function (using XPutImage API). Card images are .png files that are converted to .xpm files (required by the X11) using a free online tool (https://anyconv.com/png-to-xpm-converter/). The drawing pixel feature is then used as a backup feature in case the images are not able to load (e.g., the image files are missing). The input functions are implemented by using XSelectInput() function from X11 Library and capturing the KeyRelease event.

7. Final Questions

7.1 Final Question #1

What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Answer: It was a great experience writing large programs like this project. It did not only make me fully understand what has been taught through this course, but also gave me the chance to apply concepts and choose techniques in the design and implementation of the project. I understood the Object-Oriented design and programming and usage of design patterns much better. I also learned that having a good solid design upfront is vital. I spent a lot of effort at the very beginning of the design, such as how to structure the classes to keep a high cohesion and low coupling and let each class do its job not others (Single Responsibility Principle). This saved my effort in coding time and questions like "what fields/methods should I defined in the class", "what design patterns can be applied and how" have been thoroughly thought about in the design stage. That's why my UML model did not have fundamental changes between DD1 and DD2. Lastly, thanks for the A4 bonus question, I've spent some effort to explore the X11 Library for GUI programming before the project starts. This effort saved me a lot of time in the implementation of GUI display as the bonus feature.

7.2 Final Question #2

What would you have done differently if you had the chance to start over?

Answer: Overall, for my project, I'm quite pleased with the design. If I had the chance to start over, I would like to try for a team project. This would let me learn a different experience in the team working, communication and collaboration. If time allows, I wish I could explore more on the GUI programming, such as making the display more smoothly, adding mouse input feature, etc.

8. Conclusion

It was a great experience to work on this project. It helped me to understand all concepts and techniques learned in this course, as well as writing a large program like this project. It also helped me to learn the importance and benefits of having a good design in the early stages.