Documentation and plan of attack for the project Straights

Introduction

Straights is a project that implements an executable, with which we can perform a card game. According to the guideline of the project, the card game is a four-player card game where the players play one card one-by-one each time and end a round when all cards on the players' hands are played, and the whole game ends when there is one player gets more than 80 points, then the player with the fewest points will win, otherwise another round begins. Through this documentation and plan of attack, the classes that are planned to be implemented, the role each of the classes will play, the design patterns that will be utilized, the relationships between classes, and the necessary detailed methods of implementing the classes' functionality including the most important fields and member functions within the classes will be clearly clarified and demonstrated.

Overview

The program basically includes 5 classes: A Card class that represents cards in the game; An Execution class to specify how a player will behave during its turn depending on what type of the player is; A Player class represents players in the game, either a human player or a computer player. It owns Card objects that are distributed to it, and an Execution object depending on what type of the player is; A Gameplay class gathers all the players, a deck of cards that are played with, then shuffles/reshuffles the deck of cards and distributes them to the players, and runs the game under a specific rule, then print out the information timely; A Functionality class processes a Gameplay class to let it implement different levels of resilience to change the game mechanism that the users request.

Design

`The program is designed under four design patterns for different stages: 1. A template pattern to decide how the game is run (e.g. what cards are legal, who should play first, when does the game end, how is the game printed, etc.), which is implemented by the Gameplay class and its derived classes for different game mechanisms; 2. A strategy pattern to decide how to play a turn for different types of players (i.e. human players and different level of computer players) and enable dynamic changes of behaviour; 3. An observer pattern to update information about the played cards to players and the game runner, which is implemented by two abstract classes: Subject and Observer, and its derived classes: Card, Player, and Gameplay; 4. A decorator design pattern to dynamically decide what changes of the game are expected to be implemented, decided by the user, so that users are able to DIY their own house rules without having to specify every detail, which is implemented by Functionality class and its derived classes. In addition, more detailed implementation of very important classes and modules involved in the program is discussed below:

Class Card (concrete class derived from Subject)

Fields: 1. "observers" is a vector of Observer; 2. "suit" (const string) and "rank" (const int) are constant fields that represent what the card is; 3. "status" is either "in hand", "played", or "discarded" in string; 4. "islegal" (bool) is a either true or false.

Methods: 1. notifyObservers() is inherited from Subject to call notify() on all elements in "Observer"; 2. beplayed() to set its status to "played" and set its "isLegal" to false and then call notifyObservers(*this) to let observers know which card is currently played; 3. bediscarded() to set its status to "discarded" and set the "isLegal" to false.

Class Player (concrete class derived from Observer)

Fields: 1. "currenPlayed" is pair<string, int> to record the current played card; 2. "deck" is a vector of string which stores the deck of cards we are playing with; 3. "isHuman" is either true or false to represent whether the player is a human or a computer player; 4. “score” records the score the player has got in the current round; 5. “baseScore” records the total score the player has without the score in this round; 6. “no” records an integer which is the order number of the player; 7. “exePolicy” stores a pointer to an Execution, based on which the player behaves; 8. “cards” is a vector of Card objects that are the player’s distributed cards. The field is accessible to Gameplay and Execution.

Methods: 1. setDeck(const std::vector<std::string> & theDeck) copies a given deck of cards to the field “deck”; 2. printDeck() prints the deck the players are playing with; 3. notify(Subject & whoNotified) is derived from Observer and implemented to update “currentPlayed” based on the parameter; 4. returnCard() clears “cards”; 5. play(Card & playedcard) calls Card::beplayed() on playedcard; 6. discard(Card & discardedcard) calls Card::bediscarded() on discardedcard and add the rank of the card to “score”; 7. execute() builds an Execution object (either Human or Computer based on the type of the player) and updates “exePolicy” to the address of the Execution object. Then it calls Execution::exeAlg(this) on “exePolicy” to let “exePolicy” make the player behave correctly. If Execution::exeAlg(this) returns 1, updates “isHuman” to false and rerun execute(). The method is the client in the strategy pattern; 8. printCards() prints the player’s all cards in hand and legal plays; 9. combineScore() adds “score” to baseScore and set “score” to be 0; 10. clearDeck() clears “deck”.

Class Human (concrete class derived from Execution)

Method: exeAlg(Player *p) overrides the pure virtual method from Execution, which communicates with the user to input commands. If the user inputs “quit”, then it terminates the program, if the user inputs “play”, then it iterates p->cards to check whether the card exists, is legal, and is in-hand, then either call p->play(theCard) or print error message and try to get another command. If the user input “discard”, then it iterates p->cards to check whether there are still legal in-hand cards and the existence of the card, then either calls p->discard(theCard) or print error message and try to get another command. If the user inputs “deck”, it calls p->printDeck(). If the user inputs “ragequit”, it returns 1. Returns 0 when finishing playing or discarding.

Class Computer (concrete class derived from Execution)

Method: exeAlg(Player *p) overrides the pure virtual method from Execution, which plays the first legal and in-hand card by calling p->play() or discard the first card by calling p->discard() if there is no legal in-hand cards.

Class Gameplay (abstract class derived from Observer)

Fields: 1. “dre” to store a default_random_engine; 2. “deck” is a vector of string that is the deck will be played with, which is from AC to KS by default; 3. “players” is a vector of Player that involves all players in the game; 4. “clubs”, “diamonds”, “hearts”, and “spades” stores all played cards in the specific suits; 5. “winPlayers” is a vector of integer that stores order number of all players who win.

Methods: 1. recalculate() is a pure virtual private method. It traverses call cards of all players and updates “isLegal” field of the cards according to specific criteria; 2. printGame(Player & thePlayer) is a pure virtual private method that prints “clubs”, “diamonds”, “hearts”, and “spades” and calls thePlayer.printCard(); 3. ifEnd() is a pure virtual private method that returns true if the game reaches the condition to end the game, returns false otherwise; 4. printScoring() is a pure virtual private method that prints the discarded cards and scores for each player; 5. findfirst() is a pure virtual private method that returns an iterator of “players” that points to the player who should play first in this game; 6. gotoNext(std::vector<Player>::iterator & current) is a pure virtual private method that move current to the player who should play in the next turn; 7. proceed() is a public concrete method that calls findfirst() to get the iterator to to the first player, then calls Player::execute() on the current player who should play in this turn, then calls recalculate(), and then calls gotoNext() to iterate to the next player and repeat the process, until ifEnd() returns true. It finally calls printScoring() to show the final result of the round of game. The method, together with findfirst(), ifEnd(), printGame(), recalculate(), gotoNext(), and printScoring(), forms the template pattern utilized to enable resilience to changes in the game rule; 8. notify(Subject & whoNotified) overrides the pure virtual method from Observer to get the information about which card is currently played and updates either “clubs”, “diamonds”, “hearts”, or “spades” accordingly; 9. addPlayer(bool isH, int no) emplaces back a Player object to “players”; 10. reshuffle() calls shuffle() on “deck” to shuffle the cards; 11. getCard() makes Card objects according to the information from “deck”, then calls Card::attach() on each Card objects to add the players and itself to as the card’s observers, then distribute the Card objects evenly to each player’s “cards” field; 12. gameOver() is a pure virtual method that returns true if the game reaches the condition to end, returns false otherwise;

Class Originrule (concrete class derived from Gameplay)

Methods: 1. recalculate() implements Gameplay::recalculate() to satisfy the basic version. It traverses “cards” for each Player objects in “players”. If a card has rank 7, then set it to be legal, if a card is in the same suit as “currentPlayed”, and has a rank 1 smaller or bigger than “currenPlayed”, then set it to be legal. The mutation is done by a mutator; 2. printGame(Player & thePlayer) implements Game::printGame(Player & thePlayer). It prints all played cards first using “clubs”, “diamonds”, “hearts”, and “spades”, then it calls thePlayer.printCard(); 3. findfirst() implements Gameplay::findfirst(). It traverses “cards” of each Player object in “player” until it finds 7S, then it sets 7S to be legal and returns the iterator pointing to the player; 4. gotoNext(std::vector<Player>::iterator & current) implements Gameplay::gotoNext(std::vector<Player>::iterator & current). It simply increments current and if it reaches players.end(), it is reset to be players.begin(); 5. ifEnd() implements Gameplay::ifEnd(). It traverses “cards” of each player in “players”, if it meets a card whose status is “in hand”, it returns false. If all cards are traversed, and no card’s status is “in hand”, then it returns true; 6. prinScoring() traverses “players” and then traverses “cards” of the player to print out the cards in status “discarded”, then it informs the player’s “baseScore” and “score” in the required format, and calls Player::combineScore() on the player and then informs its “baseScore” again in the required format; 7. gameOver() implements Gameplay::gameOver(). It traverses “players” and if it meets a player whose “baseScore” is above 80, it returns true. If it finishes the traversing and no such player is found, then it returns false.

Class Bonusrule (concrete class derived from Originrule)

Fields: 1. “endScore” represents until which score a player reaches will end the game. It is default to be 80; 2. “firstplayRank” is the rank of the card that must be played first and will always legal whatever the suit is, which is default to be 7; 3. “firstplaySuit” is the suit of the card that must be played first, which is default to be “S”; 4. “adjacentInv” stores what difference in rank that two cards in the same suit are regarded as adjacent cards, which is 1 by default.

Methods: 1. recalculate() overrides Originrule::recalculate(). It traverses “cards” for each Player objects in “players”. If a card has rank that equals “firstplayedRank”, then set it to be legal, if a card is in the same suit as “currentPlayed”, and has a rank “adjacentInv” smaller or bigger than “currenPlayed”, then set it to be legal; 2. findfirst() overrides Originrule::findfirst(). It traverses “cards” of each Player object in “player” until it finds a card whose rank is equal to “firstplayedRank” and the suit is equal to “firstplayedSuit”, then it sets the card to be legal and returns the iterator pointing to the player; 3. printHouserule() prints details about the house rule in the game: the first card must be played, the rank that must be legal, the definition of adjacent cards, the players in the game and their types, and the deck of cards that will be played (this is done by calling printDeck()); 4. printDeck() prints the deck of cards that will be played with.

Class Functionality (abstract class)

Method: makeGamplay() is a pure virtual method. It is designed to process “g” and return a shared pointer to it.

Class Default (concrete class)

Fields: 1. Public field “g” a shared pointer to a Bonusrule object, which is the Bonusrule object that we want to process using a decorator design pattern; 2. Private field “seed” is the seed to randomize “g”.

Method: makeGameplay() implements Functionality::makeGameplay(). It randomize “g” by calling g->setRandom(seed), and returns “g”.

Class Decorator (abstract class)

Field: “base” is a shared pointer to a Functionality object.

Class Addplayer (concrete class)

Method: makeGamplay() implements Functionality::makeGameplay(). It communicates with the user and tries to change the number of players that will be included in the game, then it emplaces back Player objects to “g->players” as required by the user by calling g->addPlayer. Finally it returns “g”.

Class Changecard (concrete class)

Method: makeGamplay() implements Functionality::makeGameplay(). It communicates with the user and tries to change g->deck. The user should input cards that are expected to be played in the game respectively, which even allows occurrences of several same cards. However, it will not end until the number of cards reaches the number of players in the game to ensure that at least one card can be distributed to each player. Finally it returns “g”.

Class Changeendscore (concrete class)

Method: makeGamplay() implements Functionality::makeGameplay(). It communicates with the user and tries to change g->endScore. Finally it returns “g”.

Class Houserule (concrete class)

Method: makeGamplay() implements Functionality::makeGameplay(). It communicates with the user and tries to change other house rules of the game (e.g. g->firstplayedRank, g->firstplayedSuit, and g->adjacentInv). Finally it returns “g”.

Class Testmode (concrete class)

Fields: 1. “cmd” is a queue storing commands either from std::cin or ifstream; 2. “histo” is a queue storing commands that are used; 3. “iftest” stores whether the the game is in test mode or in regular mode.

Methods: 1. makeCmd() queues a command from ifstream in “cmd” if “iftest” is true, otherwise it queues in a command string from std::cin to “cmd”; 2. getCmd() pops out the first element stored in “cmd”, and queues the command string in “hist”. It finally returns the popped-out element.

Main.cc

In main.cc, we implement a main() function to control everything. First, main.cc receives arguments and get the random seed, then it decides to run the base version or the bonus version, and check whether the game is run under test mode. If the bonus version is run, call function bonusMain(argc, argv), where we will implement the bonus version. In here, a shared pointer pointed Testmode object is allocated and set its field “istest” accordingly. A pointer to the object will be added to all classes within which communications will happen, so we will omit discussing the steps. Then, a shared pointer pointed Gameplay object is built. Then, the stage where the decorator design pattern works: First we allocate a shared pointer pointed Bonusrule object, and use the seed and the Bonusrule pointer to initialize a shared pointer pointed Default object to let the shared pointer to the Bonusrule object and the seed in its field, and copy assign itself to a shared pointer of Functionality to implement polymorphism. Then, we communicate with the user to decide which house rules are expected to be changed, whereas others will remain default. Note that the changes should be requested in some specific orders, which is demonstrated in the demo. Then, once a change is requested, we build a new shared pointer pointed Decorator object by initializing the "base" field to be the first allocated Functionality shared pointer, which is specified to be either Addplayer, Changecard, Changeendscore, or Houserule, and copy assign that pointer to the first allocated Functionality shared pointer. Finally, we call makeGameplay() on the last Functionality shared pointer pointed Decorator object and we get the Gameplay we want. We then print the final house rule to the user and let him/her either reset the rule or proceed. The decorator pattern stage thus ends. Then, we run a loop, where we first shuffle the deck of cards by calling g->reshuffle(), then we create and distribute g->deck to players and attach the players and g as each Card objects' observers by calling g->getCard(), then we run g->proceed() to play a round of game, then we call returnCard() to clear up players' cards and "clubs", "diamonds", "hearts", and "spades" fields in g. The loop keeps running until g->ifEnd() returns true, when the game meets the condition to end. And in ifEnd(), the final winners' names will be attached to g->winPlayers field, and informed to the user. Finally, we ask the user whether to play again and then either call bonusMain() to rerun another game or quits. However, if main() does not receive an argument "bonus", it will return main0(argc, argv), and we will not run the decorator system and just leave everything as default.

Generally speaking, the Card, Execution, Player, Gameplay, and Functionality classes cooperates to implement one goal: Receiving and processing users' input and finishing running one game, which shows a high cohesion. Then, to have a low coupling, there is no public fields among classes and only one friend field "cards" in Player for Execution and Gameplay, and information is passed by calling functions and necessary parameters.

Resilience to change

Just as what has been discussed in Design section, I implement a template pattern to ensure that the public concrete method Gameplay::proceed() can be implemented in a different way, by optionally overriding the primitive methods ifEnd(), findfirst(), recalculate(), gotoNext(), printGame(), and printScoring() in a new subclass derived from Gameplay. To be more specific, changing ifEnd() changes the condition in which a round of game ends; Changing findfirst() changes the criteria of how to find the player who should play first; Changing recalculate() changes the criteria of legality; Changing gotoNext() changes how/in what order we turn to the next player; Changing printGame() changes the way how to output the current game situation; Changing prinScoring() changes the way how to output each player's score. Therefore, by implementing the template pattern, the way how we run a round of game can be simply changed. In addition, we can also override other methods in Gameplay to DIY any changes in the whole game. Also, a strategy pattern is implemented to enable dynamic changes to player behaviour, and when we want to have more computer player strategies, we simply define a new subclass derived from Execution and rearrange the algorithm. Then, the implementation of decorator design pattern makes changing user interface at run-time possible. For example, users may feel tired of having to specify each resilience to change in a game, especially when more resilience of change is enabled. Therefore, the decorator pattern lets them select which rules they want to change, and leave others as default to make the user interface cleaner and efficiency. Finally, the observer pattern can be employed to add a graphics UI easily, because we may just let the Gameplay inherit Subject, and let the graphics UI class inherit Observer, then any game behaviour under the monitor of Gameplay may be notified to the graphics UI to demonstrate.

Answers to questions

1. What sort of class design or 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.

I implement a template design pattern to make my structure more flexible. First, no matter how the rule changes, running a round of game has basically four steps: 1. Deciding which player should play first; 2. Deciding which cards a player has are legal plays; 3. Outputting the table, cards in the hand, and legal plays; 4. Check if the operation is legal and process the illegal or legal operations made by the player. Therefore, it is reasonable to build an abstract class with one public method to "run a round of game", and declare the four important steps as four pure virtual private methods that are expected to be implemented in the derived classes. Thus, the rule of a game can be easily changed by introducing a new subclass of the abstract class that implements the four steps in a new way. This is considered to be a template design pattern.

2. **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?**

To allow a switch from human player to computer player, and different strategies used by computer players, I would implement a strategy design pattern. For a human player, we print the game situation and wait for any input from the user, and check if the operation done by the player is legal, and give proper reactions to the input from the user to run a turn; For a computer player, it decides legal plays and selects a legal play by a strategy to run a turn; For some other players, it may have smarter strategies to play the turn. Therefore, we may design an Execution class that declares a pure virtual public method, based on which its derived classes can implement different algorithms of what to do to run a turn in different contexts (i.e. who is the player that is playing the game) within that method. Then, we can initialize a class that encapsulates the algorithm (strategy) we want based on the game context, so that we can use the algorithm to run turns. Also, we may simply abandon the present strategy object we are using and change it to another one immediately to dynamically change the strategies we are using during the play of the game.

3. 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?

The answer is the same as the answer to question 2: We implement a strategy pattern, where there is an abstract class whose derived class implements either an algorithm for playing a human player's turn or an algorithm for playing a computer player's turn. If the player quits, then the algorithm should return an error code to let us (the client) know, so that the client can abandon the currently using algorithm for human players and change it into a computer player algorithm immediately, and replay the turn.

Extra credit features

The details of how to implement the credit features are discussed in section Design. In summary:

1. Every data combination utilizes vector, and every non-stack-allocated data employs shared pointer, so there is no manual memory management, thus no deletion.
2. Enable changes to the number of players;
3. Enable changes to the deck of cards;
4. Enable changes to the house rules, including the score until which the game is over, the criteria of cards' legality, and the criteria of selecting the player who should play first;
5. Enable dynamic changeable user interface based on what resilience to changes are requested by the user and should be specified.
6. Addition of test mode and the feature to save game states. To add a test mode, I implement a class Testmode to handle commands. If an argument "test" is detected, then Testmode::makeCmd() will get a file name from cin and then read in all input from the file to a Queue of string field "cmd". And every time a std::cin >> cmd happens, cmd = getCmd() will be called to pop out and return the first element. Once the size of "cmd" reaches 0 (commands stored in "cmd" are used up), Testmode will quit the test mode. When the test mode is not activated, cmd = getCmd() will only do std::cin >> cmd[0]. Note that the robustness of the class and the methods is handled, and the description is not perfectly accurate, but aim at describing the underlying principle.

Final questions

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?

I worked alone. First, I learned that before beginning coding, we need to first think about the whole structure of the program, and which components are responsible for which parts of functionality that we want to implement. Also, what key functions should be predicted so that we can make sure that the design works. Doing so, we do not usually lose track of what we have done and what we should do during the long program development time, and it reminds us encapsulation. Also, we should consider the possibility of adding extra features or changing existed features when designing the basic structure, so that we may make the maintenance and the reusability of the code more efficient. Also, the most essential lesson is that implementing design pattern properly can make solving a difficult problem more efficient and further improve the code reusability. About memory management, using shared pointer and vectors can reduce lots of memory problems, but understanding the working methods behind the interface is extremely necessary, so that we can avoid problems such as cyclic references or trying to get the address of an element in a resized vector (because it changes after the vector is resized). Finally, exceptions handle keeps track of many bugs, which significantly reduce the time we need to locate a bug.

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

First, I would use more exceptions to handle every violation of invariant, because I wasted lots of time on locating where bugs occurred. Then, I would check whether there will be any possibility of occurrence of cyclic references before using a shred pointer, because in my program, Player contains Cards, and Cards, as Subject, contains Player as Observer, which caused a cyclic references. I had to spend lots of time changing all std::shared_ptr<Player> to stack-allocated Player. Then, I would avoid using the address of an element in a resized vector if possible, because when the vector resizes, the addresses of its elements change and the problem was so implicit that I spent much time in solving it. Finally, I may try to implement the MVC design pattern to better reuse code and improve the cohesion.

Changes from the original design

1. All heap allocated objects are changed to shared pointer pointed objects.

2. “players” field in Gameplay is not implemented in vector<Player*> but vector<Player>, because there is no need to use pointers due to the few times Player objects are passed.
3. Gameplay inherits features from Observer to observe Card objects, so that it know which card is played and it can update the information immediately.
4. A decorator pattern is implemented to enable run-time changes to the user interface, because too much resilience to changes have been enabled, users will feel tired of having to specify every rule.