SWEN30006 Project 2
Report

Overview of the design

For this Pasur game, we used singleton, strategy, and composite patterns to help us design the scoring logic and the feature of writing game information into a log. Those three design patterns help us achieve polymorphism, high cohesion, and low coupling. It also made future design relatively easier.

Design processing

For this game, we are facing many different scoring rules. We used a strategy pattern to implement different strategies. The strategy pattern is a behavioural pattern that lets us define a family of algorithms and put each of them into a separate class and make their projects interchangeable. To achieve the strategy pattern, we created a strategy package and added an interface ScoringStrategy.

We use the interface instead of an abstract class because the interface is a “can-do” relationship, and the abstract class represents a ‘is’ relationship. The current situation only needs the class to calculate the score, so the interface is better. The subclass can implement multiple interfaces rather than a subclass can only extend one abstract class. So, in the future, if we need to add more rules, we can create more ScoreStrategy to implement the same interface. Moreover, if there is another demand which is different from calculating scores, we can also create another interface and achieve this new feature.

This interface only has one method for the current requirement, *getScore(Hand hand, Hand surs)* return an int value score. This method will be overridden in the different strategies class to calculate scores based on the different strategy rules. In the strategy package, we also implement different strategy classes, which are implemented from the ScoreStrategy interface. For example, Ace.java is used to calculate the corresponding score when a player has an Ace card. The Ace.java class has an override method *getScore(Hand hand, Hand surs)*. In the future, if any new scoring strategies are added, update the current strategy, or delete the current strategy, we only need to add, update or delete strategy in the strategy package.

On the other hand, we also implemented a composite pattern (named ComposteStrategy.java) to apply different scoring strategies better, so that we could add corresponding score strategies to players. However, we found out that all strategies will be covered soon after a few rounds; thus, we decided to add all strategies once in the composite pattern to save time for the system to check if it uses some strategies for each round. In the future, if we need to add more scoreStrategy, we only need to adjust the addStrategy method in CompositeStrategy.java to achieve better performance. Consequently, this leads to high cohesion and low coupling.

For the game’s log design, we created a class named Log.Java to write everything into "pasur.log." To make a better design, we choose to use Singleton Pattern to help implement log features. The reasons are that we only need a single instance (File/Log) in this class, and it will be available to all the clients' classes. Also, the best part for Singleton Pattern is providing a way to access its only objects, which can be accessed directly without the need to instantiate the object of the class.

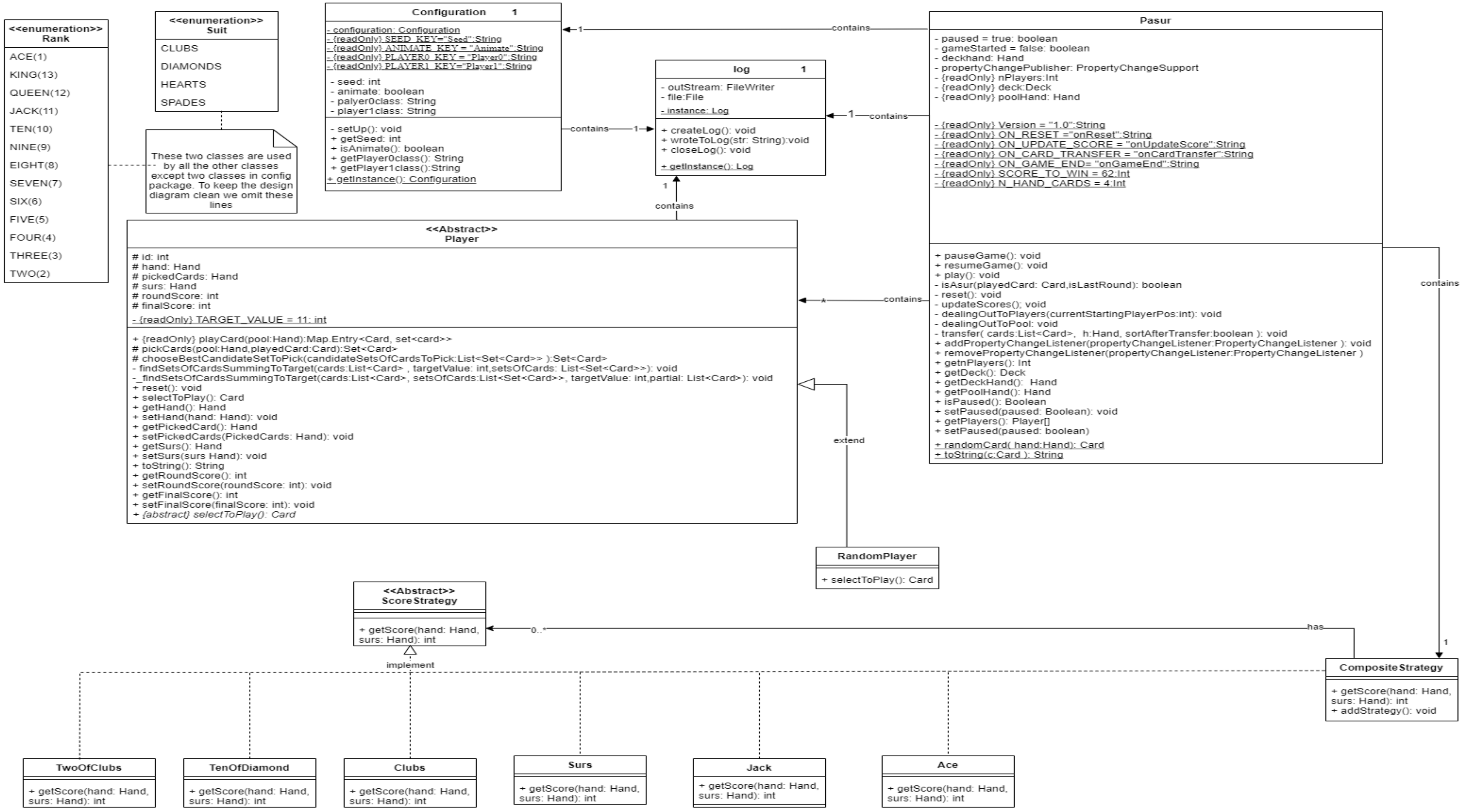
In the Log class, when we first call the *getInstance()* method, it creates an object with a named instance and returns it to the variable. Since an instance is static, it is changed from null to some object. Next time, if we call the *getInstance()* method, since the instance is not null, it returns to the variable instead of instantiating the Log class again.

If any new features are added and need to be written into the log, we need to make sure this class has a *private final Log log = Log.getInstance()*, so we can write to log at any place in this class. If we want to change the log’s name or the log format, we only need to modify those features in the Log.java class, which leads to high cohesion and low coupling.

Conclusion

In conclusion, using the above design patterns (strategy pattern, singleton, strategy, and composite pattern), we implement all the features given by the project specification, and the design is easily visualized and understood. The code is also easy to modify as we can modify each feature in separate classes in the future. This achieves high cohesion and low coupling.

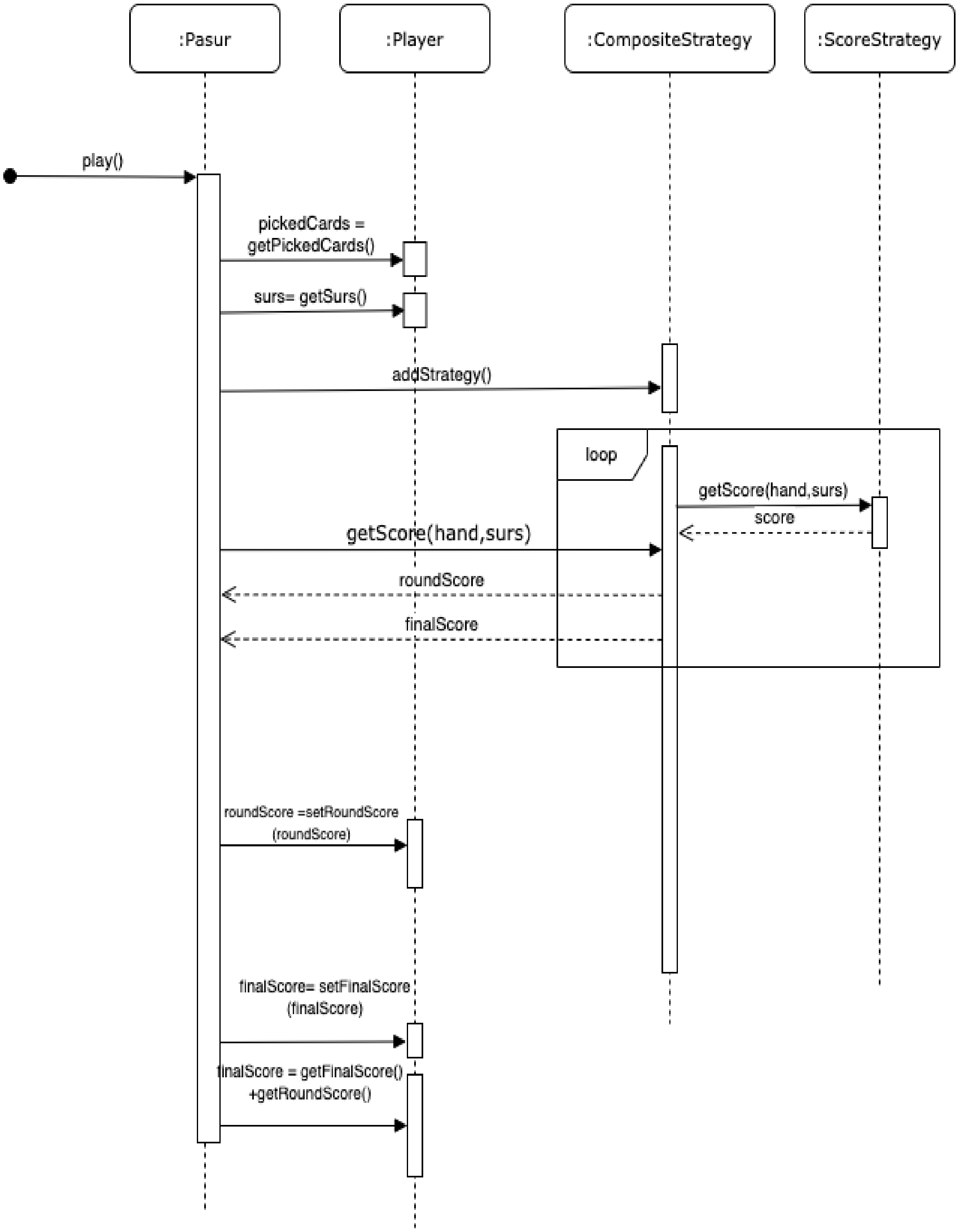
Design Class Diagram



<<enumeration>>
Rank
ACE(1)
KING(13)
QUEEN(12)
JACK(11)
TEN(10)
NINE(9)
EIGHT(8)
SEVEN(7)
SIX(6)
FIVE(5)
FOUR(4)
THREE(3)
TWO(2)
<<enumeration>>
Suit
CLUBS
DIAMONDS
HEARTS
SPADES
These two classes are used by all the other classes except two classes in config package. To keep the design diagram clean we omit these lines
Configuration 1
- configuration: Configuration
- {readOnly} SEED_KEY="Seed":String
- {readOnly} ANIMATE_KEY = "Animate":String
- {readOnly} PLAYER0_KEY = "Player0":String
- {readOnly} PLAYER1_KEY="Player1":String
- seed: int
- animate: boolean
- palyer0class: String
- player1class: String
- setUp(): void
+ getSeed: int
+ isAnimate(): boolean
+ getPlayer0class(): String
+ getPlayer1class():String
+ getInstance(): Configuration
contains
1
log 1
- outStream: FileWriter
- file:File
- instance: Log
+ createLog(): void
+ wroteToLog(str: String):void
+ closeLog(): void
+ getInstance(): Log
1
contains
1
contains
1
contains
Pasur
- paused = true: boolean
- gameStarted = false: boolean
- deckhand: Hand
- propertyChangePublisher: PropertyChangeSupport
- {readOnly} nPlayers:Int
- {readOnly} deck:Deck
- {readOnly} poolHand: Hand
- {readOnly} Version = "1.0":String
- {readOnly} ON_RESET ="onReset":String
- {readOnly} ON_UPDATE_SCORE = "onUpdateScore":String
- {readOnly} ON_CARD_TRANSFER = "onCardTransfer":String
- {readOnly} ON_GAME_END= "onGameEnd":String
- {readOnly} SCORE_TO_WIN = 62:Int
- {readOnly} N_HAND_CARDS = 4:Int
+ pauseGame(): void
+ resumeGame(): void
+ play(): void
- isAsur(playedCard: Card,isLastRound): boolean
- reset(): void
- updateScores(); void
- dealingOutToPlayers(currentStartingPlayerPos:int): void
- dealingOutToPool: void
- transfer(cards:List<Card>, h:Hand, sortAfterTransfer:boolean): void
+ addPropertyChangeListener(propertyChangeListener:PropertyChangeListener): void
+ removePropertyChangeListener(propertyChangeListener:PropertyChangeListener)
+ getnPlayers(): Int
+ getDeck(): Deck
+ getDeckHand(): Hand
+ getPoolHand(): Hand
+ isPaused(): Boolean
+ setPaused(paused: Boolean): void
+ getPlayers(): Player[]
+ setPaused(paused: boolean)
+ randomCard(hand:Hand): Card
+ toString(c:Card): String
contains
*
contains
<<Abstract>>
Player
# id: int
# hand: Hand
# pickedCards: Hand
# surs: Hand
# roundScore: int
# finalScore: int
- {readOnly} TARGET_VALUE = 11: int
+ {readOnly} playCard(pool:Hand):Map.Entry<Card, set<card>>
# pickCards(pool:Hand,playedCard:Card):Set<Card>
# chooseBestCandidateSetToPick(candidateSetsOfCardsToPick:List<Set<Card>>):Set<Card>
- findSetsOfCardsSummingToTarget(cards:List<Card> , targetValue: int,setsOfCards: List<Set<Card>>): void
-_findSetsOfCardsSummingToTarget(cards:List<Card>, setsOfCards:List<Set<Card>>, targetValue: int,partial: List<Card>): void
+ reset(): void
+ selectToPlay(): Card
+ getHand(): Hand
+ setHand(hand: Hand): void
+ getPickedCard(): Hand
+ setPickedCards(PickedCards: Hand): void
+ getSurs(): Hand
+ setSurs(surs Hand): void
+ toString(): String
+ getRoundScore(): int
+ setRoundScore(roundScore: int): void
+ getFinalScore(): int
+ setFinalScore(finalScore: int): void
+ {abstract} selectToPlay(): Card
extend
RandomPlayer
+ selectToPlay(): Card
<<Abstract>>
Score Strategy
+ getScore(hand: Hand, surs: Hand): int
0..*
has
1
implement
CompositeStrategy
+ getScore(hand: Hand, surs: Hand): int
+ addStrategy(): void
TwoOfClubs
+ getScore(hand: Hand, surs: Hand): int
TenOfDiamond
+ getScore(hand: Hand, surs: Hand): int
Clubs
+ getScore(hand: Hand, surs: Hand): int
Surs
+ getScore(hand: Hand, surs: Hand): int
Jack
+ getScore(hand: Hand, surs: Hand): int
Ace
+ getScore(hand: Hand, surs: Hand): int

Design Sequence Diagram

The design sequence diagram below illustrates the flow of calculating each round score and the final score.


:Pasur
:Player
:CompositeStrategy
:ScoreStrategy
play()
pickedCards =
getPickedCards()
surs= getSurs()
addStrategy()
loop
getScore(hand,surs)
score
getScore(hand,surs)
roundScore
finalScore
roundScore =setRoundScore
(roundScore)
finalScore= setFinalScore
(finalScore)
finalScore = getFinalScore()
+getRoundScore()