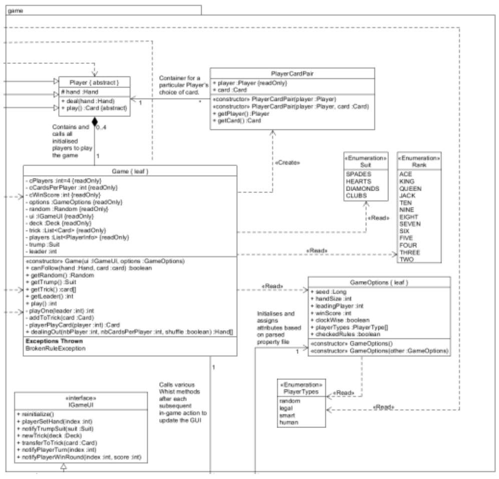
Swen30006 Project 2 - Report

Jiles Denham [914740], Andrew Lee [1083308], Lachlan McKeith [996258]

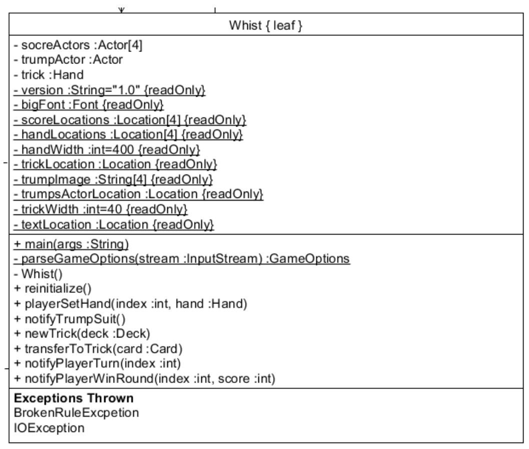
Initial Refactoring

The given program was extensively refactored into a wider set of functions and classes with smaller, more cohesive responsibilities in anticipation of the project requirements. The bulk of the given Whist class implemented three things – The running of the game itself, updating the GUI to reflect the state of the game, and the game's players. In our refactorization the first two responsibilities are delegated to a class each, Game and Whist respectively. In order to support the project requirements of various NPCs, all logic regarding the random and human players was extracted into RandomPlayer and HumanPlayer, both inheriting from an abstract superclass Player. With this class hierarchy, the behaviour of any future NPCs can be polymorphically assigned to a new subclass of Player, allowing Game to only call Player.play() with the subclass managing the specific NPC's playing logic.

The Game class can be seen as the core of the program, as it is responsible for all aspects of driving and maintaining the state of the game. To fulfill this purpose Game must have visibility on all aspects of the game, including all player's hands, the current trick, and the deck. Since it is necessary that Game be the information expert on the overall state of the game, it also makes sense that it be the creator of all objects that are required for the game to be played, such as the deck and various Players. All created objects that are involved in the entire lifetime of the game have attribute visibility to Game and are subsequently called upon in Game.play() and Game.playOne() to move the game forward.


game
Player { abstract }
# hand :Hand
+ deal(hand :Hand)
+ play() :Card (abstract)
Container for a particular Player's choice of card.
PlayerCardPair
+ player :Player (readOnly)
+ card :Card
«constructor» PlayerCardPair(player :Player)
«constructor» PlayerCardPair(player :Player, card :Card)
+ getPlayer() :Player
+ getCard() :Card
Contains and calls all initialised players to play the game
0..4
1
«Create»
«Enumeration» Suit
SPADES
HEARTS
DIAMONDS
CLUBS
«Enumeration» Rank
ACE
KING
QUEEN
JACK
TEN
NINE
EIGHT
SEVEN
SIX
FIVE
FOUR
THREE
TWO
Game (leaf)
- cPlayers :int=4 {readOnly}
- cCardsPerPlayer :int {readOnly}
- cWinScore :int {readOnly}
- options :GameOptions {readOnly}
- random :Random {readOnly}
- ui :IGameUI {readOnly}
- deck :Deck {readOnly}
- trick :List<Card> {readOnly}
- players :List<PlayerInfo> {readOnly}
- trump :Suit
- leader :int
«constructor» Game(ui :IGameUI, options :GameOptions)
+ canFollow(hand :Hand, card :card) :boolean
+ getRandom() :Random
+ getTrump() :Suit
+ getTrick() :card[]
+ getLeader() :int
+ play() :int
- playOne(leader :int) :int
- addToTrick(card :Card)
- playerPlayCard(player :int) :Card
+ dealingOut(nbPlayer :int, nbCardsPerPlayer :int, shuffle :boolean) :Hand[]
Exceptions Thrown
BrokenRuleException
«Read»
GameOptions (leaf)
+ seed :Long
+ handSize :int
+ leadingPlayer :int
+ winScore :int
+ clockWise :boolean
+ playerTypes :PlayerType[]
+ checkedRules :boolean
«constructor» GameOptions()
«constructor» GameOptions(other :GameOptions)
Initialises and assigns attributes based on parsed property file
«interface» IGameUI
+ reinitialize()
+ playerSetHand(index :int)
+ notifyTrumpSuit(suit :Suit)
+ newTrick(deck :Deck)
+ transferToTrick(card :Card)
+ notifyPlayerTurn(index :int)
+ notifyPlayerWinRound(index :int, score :int)
Calls various Whist methods after each subsequent in-game action to update the GUI
«Enumeration» PlayerTypes
random
legal
smart
human

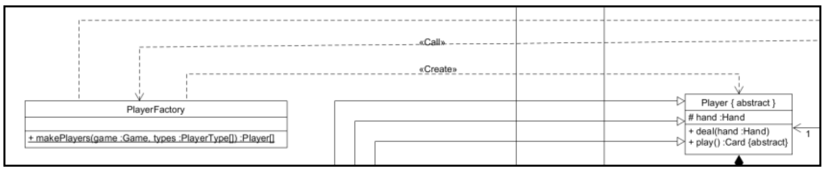
Game, by extension of its overarching responsibility, is also in charge of checking for valid moves during the course of play, containing a canFollow() method for Players to call and throwing BrokenRuleExceptions.


Whist { leaf }
- socreActors :Actor[4]
- trumpActor :Actor
- trick :Hand
- version :String="1.0" {readOnly}
- bigFont :Font {readOnly}
- scoreLocations :Location[4] {readOnly}
- handLocations :Location[4] {readOnly}
- handWidth :int=400 {readOnly}
- trickLocation :Location {readOnly}
- trumpImage :String[4] {readOnly}
- trumpsActorLocation :Location {readOnly}
- trickWidth :int=40 {readOnly}
- textLocation :Location {readOnly}
+ main(args :String)
- parseGameOptions(stream :InputStream) :GameOptions
- Whist()
+ reinitialize()
+ playerSetHand(index :int, hand :Hand)
+ notifyTrumpSuit()
+ newTrick(deck :Deck)
+ transferToTrick(card :Card)
+ notifyPlayerTurn(index :int)
+ notifyPlayerWinRound(index :int, score :int)
Exceptions Thrown
BrokenRuleExcpetion
IOException

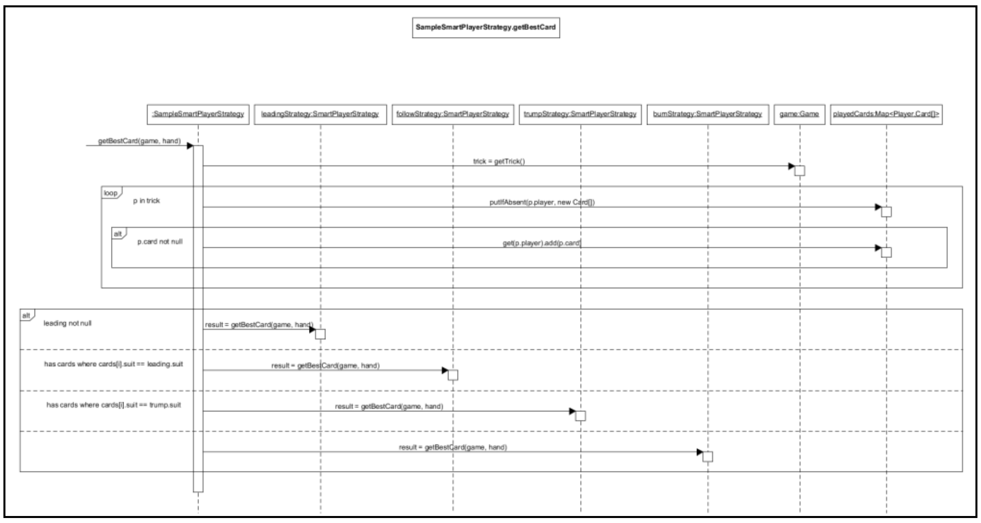
The refactored Whist class is responsible for maintaining the GUI to reflect Game's state, with all logic pertaining to the GUI in the given program refactored into more cohesive functions. Our new Whist is an example of the Observer Pattern, with Game notifying Whist when its internal state changes so that the necessary GUI updates take place. In order to manage and update the GUI, Whist must be the information expert and creator of the various GUI objects, and thus holds constants for all coordinate locations of GUI elements along with initialising/removing the various Actors represented in the GUI. Game has attribute visibility to Whist via the IGameUI interface which Whist implements. This can be seen as a pre-emptive application of the Adapter pattern, with IGameUI as a possible interface that could be implemented by another GUI class that used a different game library. This ensures that as long the GUI class implemented IGameUI, Game could still be used to run and maintain the card game.

Implementing Additional NPCs

Based on the input parameters, the PlayerFactory class constructs each player according to its archetype (Random, Legal, Human or Smart). It is important to note that the PlayerFactory class is a pure fabrication, that is, it doesn't exist within the problem domain. This class was created to boost cohesion through the implementation of the Factory pattern, rather than construct each player in Game, as would be suggested by Information Expert or Creator patterns. If players were constructed in the Game class, that class would have lower cohesion and as such the factory was implemented. The Factory class is also a facade to obscure more complicated logic behind a simpler interface to the Game class.


«Call»
«Create»
PlayerFactory
+ makePlayers(game :Game, types :PlayerType[]) :Player[]
Player { abstract }
# hand :Hand
+ deal(hand :Hand)
+ play() :Card {abstract}
1

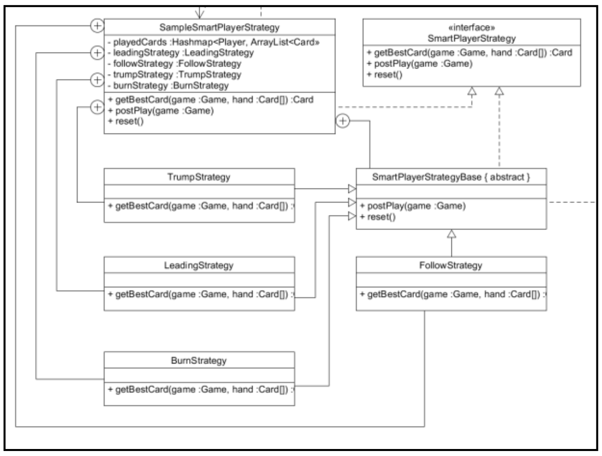
Each player instance is a polymorphic variation based on the player type. The player class inherits from these player type classes, and inherits it's behaviors from them in the process. Each player type is considered an information expert on its strategy so these are broadly included in each specific player type class (except the SmartPlayer which is explained later). HumanPlayer is the most simple of these classes due to needing no algorithm or logic to choose which card to play (since this is picked by a user). Each other class must implement a chosen strategy to pick which card to play based on the outline in the specifications. RandomPlayer simply plays a random card, with no regard for what would be considered a 'legal' move. LegalPlayer inherits this behavior to pick a random card but has an adaptor to ensure the move is legal, or else it will just pick another random card from its hand to play. This relationship between classes is therefore highly coupled, since the behavior of LegalPlayer is based on the behavior of RandomPlayer, but this behavior is acceptable since the LegalPlayer class has the added safeguard of ensuring legal moves before accepting a random card.


SampleSmartPlayerStrategy.getBestCard
:SampleSmartPlayerStrategy
leadingStrategy:SmartPlayerStrategy
followStrategy:SmartPlayerStrategy
trumpStrategy:SmartPlayerStrategy
burnStrategy:SmartPlayerStrategy
game:Game
playedCards:Map<Player,Card[]>
getBestCard(game, hand)
trick = getTrick()
loop
p in trick
putIfAbsent(p.player, new Card[])
alt
p.card not null
get(p.player).add(p.card)
alt
leading not null
result = getBestCard(game, hand)
has cards where cards[i].suit == leading.suit
result = getBestCard(game, hand)
has cards where cards[i].suit == trump.suit
result = getBestCard(game, hand)
result = getBestCard(game, hand)

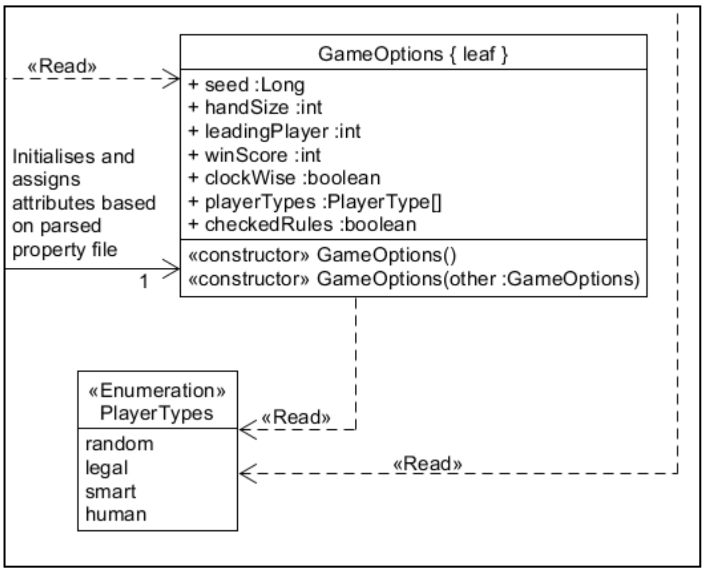
SmartPlayer is the most complex relationship due to the necessity for a SmartPlayerStrategy strategy pattern. The strategy pattern is required due to the dynamic nature of the algorithm and the necessity to meet the facade and decorator pattern requirements. The algorithm in use for this player to find the highest ranking card they currently have and to play that if they have a chance of winning, otherwise 'burn' their lowest ranking card. This algorithm is obscured to the SmartPlayer simply as getBestCard() so follows the strategy pattern as well as the facade pattern. getBestCard is a decorator since it dynamically changes behavior based on the cards that have currently been played. To track which cards have been played, each card publishes to an observer pattern to which each player is subscribed. This made the most sense to implement since the alternative would be to have the player look at the most recent card every time step of the game.

Implementing Configurable Parameters

The path to the desired property file is given as whist.properties, opened, and immediately passed to parseGameOptions() by main. All aspects of the game that we have deemed configurable are contained as final attributes of the GameOptions class. The original Whist's final attributes of winningScore and nbStartCards, and private attribute enforceRules, have been assimilated into GameOptions as configurable parameters of winScore, handSize, and checkedRules respectively. The project requirements of initialising different NPC players and a seed to allow for repeatable runs are also present as attributes of GameOptions, along with the non-essential ability to configure who leads the first round and whether turns go clockwise or anticlockwise. As Game is responsible for all aspects of running the game, it is given attribute visibility to GameOptions so that it may initialise and run the game appropriately.


SampleSmartPlayerStrategy
- playedCards :Hashmap<Player, ArrayList<Card»
- leadingStrategy :LeadingStrategy
- followStrategy :FollowStrategy
- trumpStrategy :TrumpStrategy
- burnStrategy :BurnStrategy
+ getBestCard(game :Game, hand :Card[]) :Card
+ postPlay(game :Game)
+ reset()
«interface»
SmartPlayerStrategy
+ getBestCard(game :Game, hand :Card[]) :Card
+ postPlay(game :Game)
+ reset()
TrumpStrategy
+ getBestCard(game :Game, hand :Card[]) :
SmartPlayerStrategyBase { abstract }
+ postPlay(game :Game)
+ reset()
LeadingStrategy
+ getBestCard(game :Game, hand :Card[]) :
FollowStrategy
+ getBestCard(game :Game, hand :Card[]) :
BurnStrategy
+ getBestCard(game :Game, hand :Card[]) :

In its constructor, Game delegates player initialisation to PlayerFactory.makePlayers(), passing it the configured GameOptions.playerTypes as argument so that the appropriate Player subclasses may be initialised. Here the Factory Pattern is used to hide the creation logic of the appropriate players from game, keeping Game cohesive in its purpose and allowing extensibility in the case of more NPCs and/or an increasingly complex Player initialisation process.


«Read»
Initialises and assigns attributes based on parsed property file
1
GameOptions { leaf }
+ seed :Long
+ handSize :int
+ leadingPlayer :int
+ winScore :int
+ clockWise :boolean
+ playerTypes :PlayerType[]
+ checkedRules :boolean
«constructor» GameOptions()
«constructor» GameOptions(other :GameOptions)
«Enumeration» PlayerTypes
random
legal
smart
human
«Read»
«Read»

To support repeatable runs based on a given seed, Game.random is always initialised with GameOptions.seed if it was given as a property. An alternative dealingOut() method had to also be defined, as the dealingOut() methods defined by ch.aplu.jcardgame.Deck did not take a seed for the shuffling of the deck. Game.dealingOut() uses Game.random to shuffle the deck if needed, ensuring that the same number generator is used for all random aspects of the game.

The GameOptions attributes of handSize, leadingPlayer, winScore, clockwise, and checkedRules all pertain to management of gameplay by Game. GameOptions.handSize, through the alias Game.cCardsPerPlayer, is used by dealingOut() and play() to determine how many cards to deal, and when to deal them. WinScore and leadingPlayer are used solely by play() to determine the first leader and when the game has been won, clockwise used by playOne() to determine the rotation, and checkedRules solely by playerPlayCard() to determine if a BrokenRuleException should be thrown if an illegal move is played.