Home Exam - PointSalad game

Enzo Brossier-Sécher : enzbro-4@student.ltu.se

Part A

I - Unit testing

With the given implementation of the game, every requirement that could be manually tested is correctly implemented.
However, three of them can not be tested without modifying the code. It is the case for requirements 4, 5 and 11.

Requirement 4 cannot be tested as we cannot see the number of remaining cards in each pile, and they are refilled before showing us that they ran out of cards.
Requirement 5 cannot be tested as we have no data printing nor control before the market has been set up.
Requirement 11 cannot be tested as we do not know how many cards there are in each pile. We therefore cannot know from which pile an empty pile is being refilled. In fact, we do not even know when a pile is being refilled.

To create a test on the market setup to verify the number of cards in the three different piles, we need to instantiate a PointSalad object to be able to reach its setPiles method and then test its piles attribute. However, instantiating a PointSalad object automatically launches the game and waits for user inputs.
Therefore, we cannot even test it. We could only launch our test once the game is over, but the piles will then be empty.

Similarly, it is the reason why we cannot either test where the initial vegetable cards of the market come from nor where the refilling vegetable cards come from.

2 - Software Architecture Design and Refactoring

Please find the diagrams in the appendix at the end of this report. You can also find them in the PointSalad/umldoc/ folder to be able to zoom in with better quality.

Addressing the quality attributes of requirement 16
With this design, I am addressing the three required quality attributes of requirement 16.

Indeed, when it comes to **Modifiability**, the code and behavior of the game will be easy to modify thanks to the use of the JSON file to change cards, and the Config file to modify basic parameters of the game.
In order to get deeper modifiability of the code itself, the use of many interfaces and abstract classes will allow the code to be easily modifiable. We will be able to code a new class implementing the correct interface, or extending the correct abstract class, and then substitute the default one with this new class to reach the desired updated behavior.
For instance, if we want to modify the way the Network is handled, we will be able to code a new server class and client connection class. Both of them will implement the appropriate IServer and IClientConnection, and they will then be used in the PointSaladHost and PointSaladClient classes instead of the default classes.
We will also be able to create subclasses of the generic classes, such as State, if we need to make it more detailed. Though the original State class is generic enough to be highly adaptable to many new situations.

These remarks can be extended to **Extensibility** as we will be able to add new functionalities or behaviors for another game in a similar way. We will be able to add new classes that implement the base interfaces and abstract classes to create a whole new game, adding new criteria, new cards, new game phases, and so on.
Moreover, we won't have to code everything from scratch as some classes are already generic. For instance, we will keep the already coded StateManager and Pile classes. Same for the players, but with the need to implement a new BotLogic for this new game. We would still have to create new factories or update the existing ones to make them compatible with the game's new cards and criteria.

Finally, about **Testability**, this implementation of the game is provided with a generic StateManager that allows state injection in order to easily test a specific phase of the game, or create a dummy game state to launch the game from in order to track specific bugs more easily.
Additionally, the use of a State Pattern with phases will allow us to test phases on their own, without having to create a server and clients to be able to test the game.

Now that the code is implemented, this is what I used in the RequirementsTest class to test requirement 12. I was able to create a dummy drafting phase and check if the next phase was the flipping phase for the same player. Then I could check if the next phase after a flipping phase was a drafting phase for the next player if the market is not empty, or a scoring phase else.

Design patterns in this design
This design uses several Design patterns in order to address the quality attributes mentioned above.

It uses two **Factory** patterns (on cards and criteria) to allow the easy loading of these objects from a file. In the end, they address Modifiability.
It also uses two **Singleton** classes (Config and TerminalInput) to ensure the corresponding classes are always accessible and instantiated only once. This allows for higher cohesion in classes since they do not have to manage this part anymore. However, it also leads to higher coupling as these two classes are used in these classes.
In the end, this makes the Config file usable, which leads to higher Modifiability. It also ensures to avoid some bugs, such as the terminal-scanner related ones with a single scanner taking inputs from the terminal, and increases Testability as we would only have to test TerminalInput to ensure that every user input from the terminal is correctly handled.
This design relies on a **State pattern** as mentioned above in order to address **Testability** as well as Extensibility. This is due to the lower coupling permitted by this pattern, with phases that can be tested individually and that possess the whole logic to process themselves from the current state of the game, as well as to get the next phase. It allows for higher cohesion in the game processing logic.
This design also relies on many **Strategy patterns**, such as the game selection in the Main class. Though it will be revealed later on, when extensions will be added. This helps to achieve Extensibility in this case, as we can easily choose which game will be executed.
Additionally, the way criteria are implemented is also based on a **Strategy pattern**, where each criterion (strategy) possesses the logic to compute the score of a given hand. This pattern decreases the coupling among the scoring logic and allows for more primitive methods. Therefore, it also improves Testability, as it is shown in the many testing classes I have implemented to test each criterion’s logic.
This can also be extended to many other objects in the code, such as Network objects as we can modify the default Network behavior by switching the “default” Server and ClientConnection classes to new ones that would implement the correct interfaces. Therefore, this pattern also addresses Modifiability in some cases.

Appendix

You can also find these diagrams in the PointSalad/umldoc/ folder to be able to zoom in with better quality.

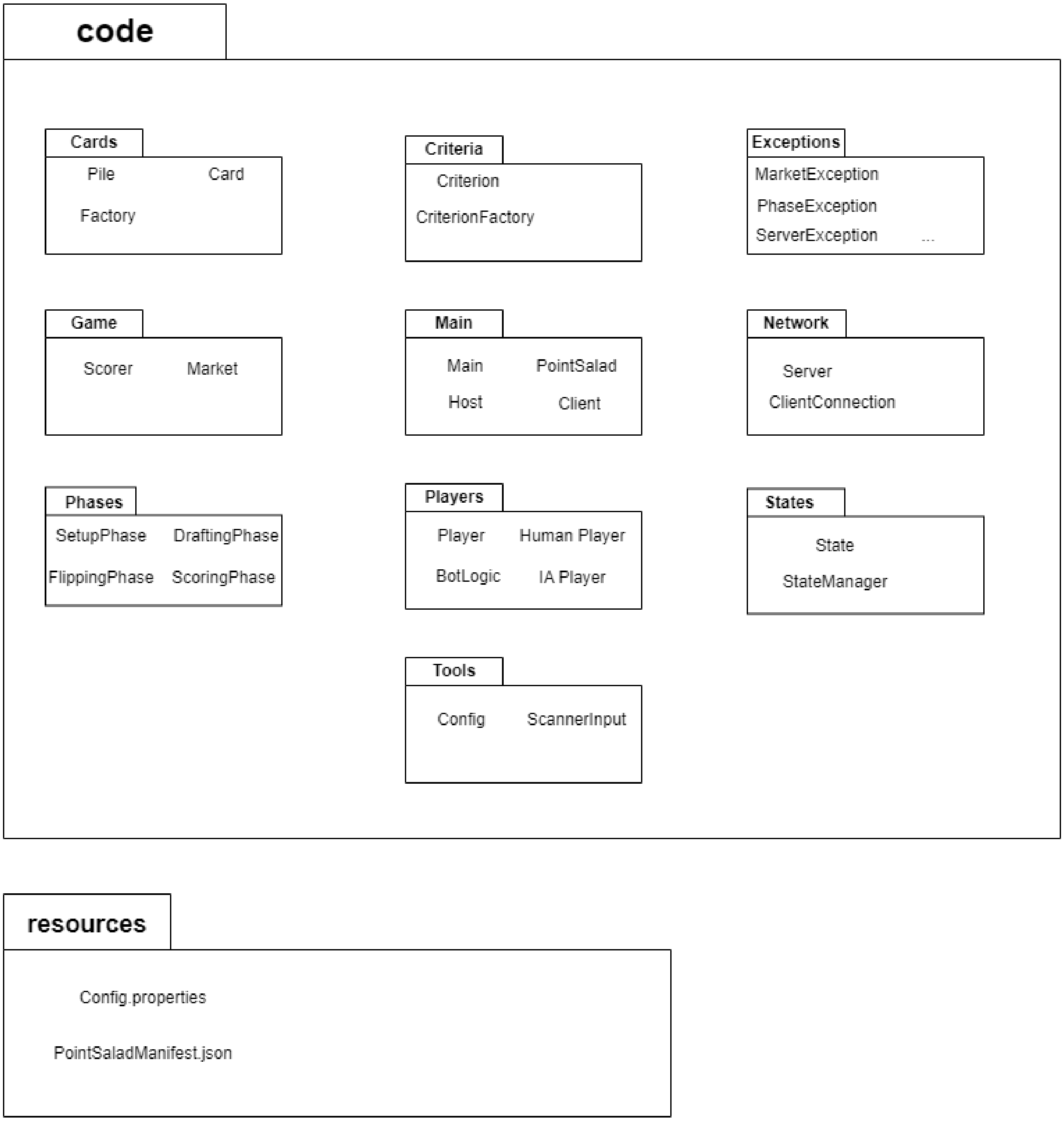
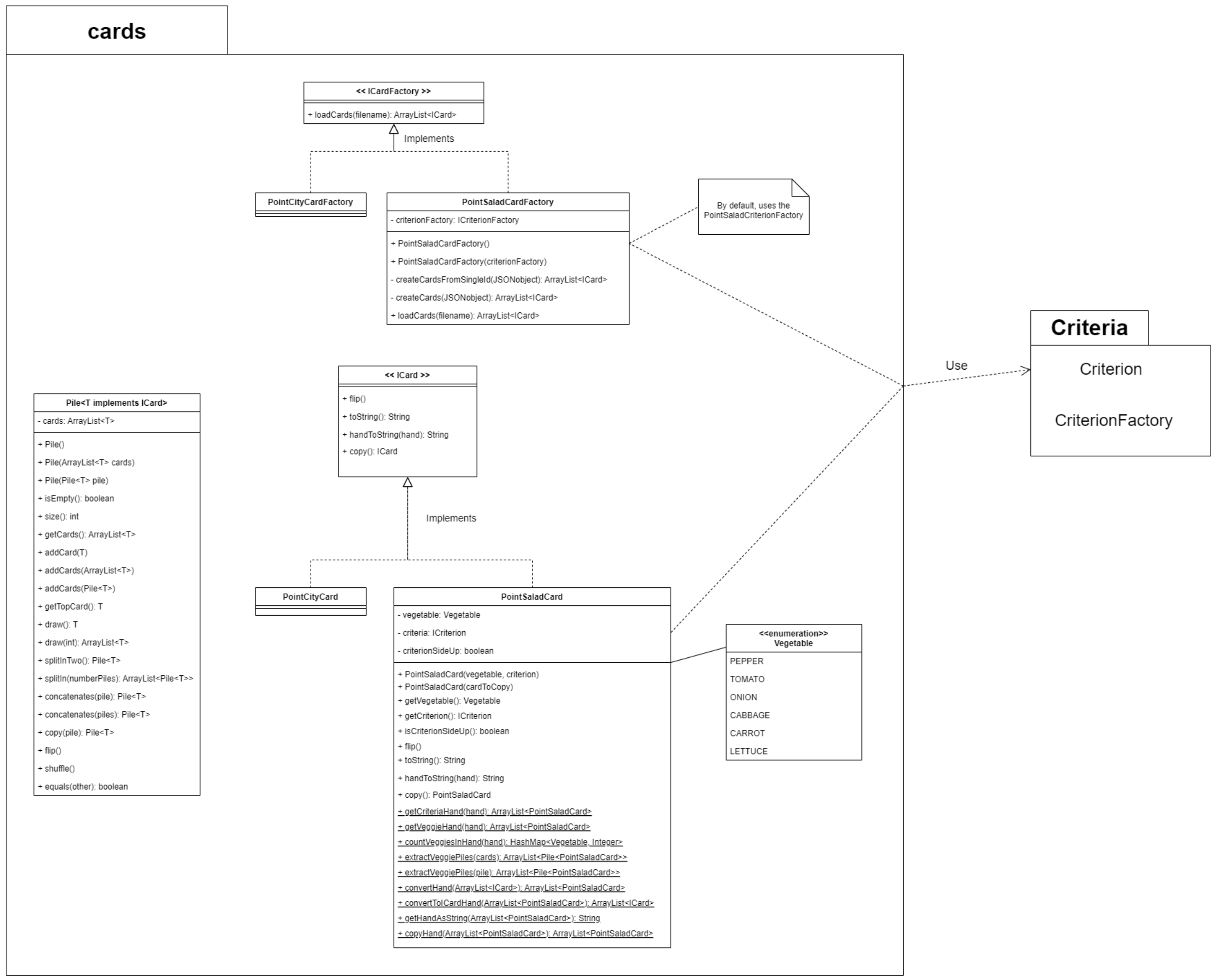
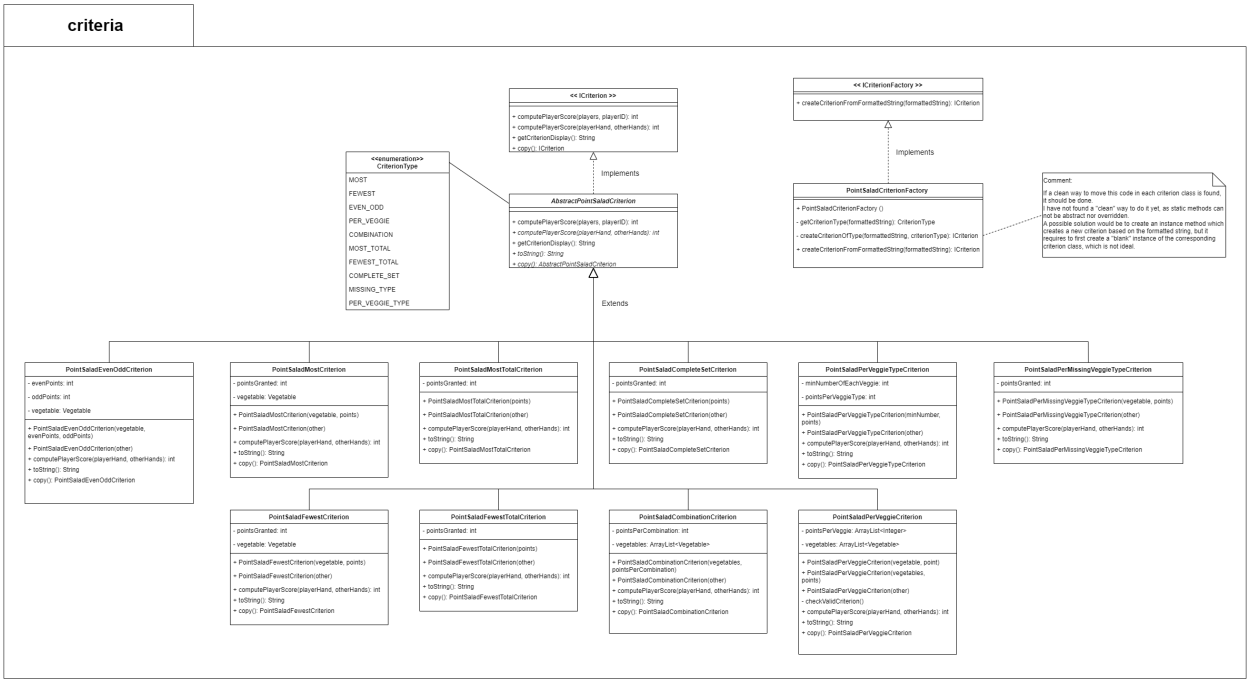


Fig 1: packages architecture


cards
<< ICardFactory >>
+ loadCards(filename): ArrayList<ICard>
Implements
PointCityCardFactory
PointSaladCardFactory
- criterionFactory: ICriterionFactory
+ PointSaladCardFactory()
+ PointSaladCardFactory(criterionFactory)
- createCardsFromSingleId(JSONobject): ArrayList<ICard>
- createCards(JSONobject): ArrayList<ICard>
+ loadCards(filename): ArrayList<ICard>
By default, uses the PointSaladCriterionFactory
Criteria
Criterion
CriterionFactory
Use
Pile<T implements ICard>
- cards: ArrayList<T>
+ Pile()
+ Pile(ArrayList<T> cards)
+ Pile(Pile<T> pile)
+ isEmpty(): boolean
+ size(): int
+ getCards(): ArrayList<T>
+ addCard(T)
+ addCards(ArrayList<T>)
+ addCards(Pile<T>)
+ getTopCard(): T
+ draw(): T
+ draw(int): ArrayList<T>
+ splitInTwo(): Pile<T>
+ splitIn(numberPiles): ArrayList<Pile<T>>
+ concatenates(pile): Pile<T>
+ concatenates(piles): Pile<T>
+ copy(pile): Pile<T>
+ flip()
+ shuffle()
+ equals(other): boolean
<< ICard >>
+ flip()
+ toString(): String
+ handToString(hand): String
+ copy(): ICard
Implements
PointCityCard
PointSaladCard
- vegetable: Vegetable
- criteria: ICriterion
- criterionSideUp: boolean
+ PointSaladCard(vegetable, criterion)
+ PointSaladCard(cardToCopy)
+ getVegetable(): Vegetable
+ getCriterion(): ICriterion
+ isCriterionSideUp(): boolean
+ flip()
+ toString(): String
+ handToString(hand): String
+ copy(): PointSaladCard
+ getCriteriaHand(hand): ArrayList<PointSaladCard>
+ getVeggieHand(hand): ArrayList<PointSaladCard>
+ countVeggiesInHand(hand): HashMap<Vegetable, Integer>
+ extractVeggiePiles(cards): ArrayList<Pile<PointSaladCard>>
+ extractVeggiePiles(pile): ArrayList<Pile<PointSaladCard>>
+ convertHand(ArrayList<ICard>): ArrayList<PointSaladCard>
+ convertToICardHand(ArrayList<PointSaladCard>): ArrayList<ICard>
+ getHandAsString(ArrayList<PointSaladCard>): String
+ copyHand(ArrayList<PointSaladCard>): ArrayList<PointSaladCard>
<<enumeration>>
Vegetable
PEPPER
TOMATO
ONION
CABBAGE
CARROT
LETTUCE

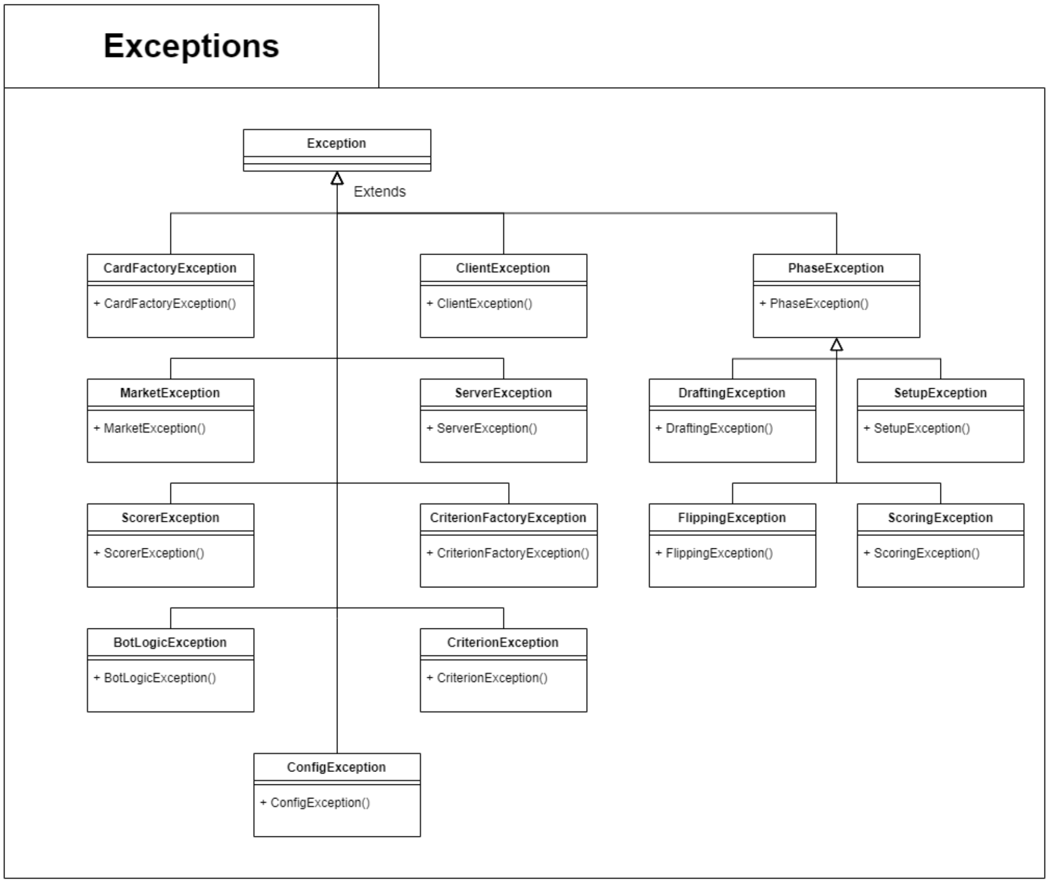
<u>Fig 2:</u> Cards Class diagram


criteria
<< ICriterion >>
+ computePlayerScore(players, playerID): int
+ computePlayerScore(playerHand, otherHands): int
+ getCriterionDisplay(): String
+ copy(): ICriterion
<< ICriterionFactory >>
+ createCriterionFromFormattedString(formattedString): ICriterion
Implements
<<enumeration>>
CriterionType
MOST
FEWEST
EVEN_ODD
PER_VEGGIE
COMBINATION
MOST_TOTAL
FEWEST_TOTAL
COMPLETE_SET
MISSING_TYPE
PER_VEGGIE_TYPE
Implements
AbstractPointSaladCriterion
+ computePlayerScore(players, playerID): int
+ computePlayerScore(playerHand, otherHands): int
+ getCriterionDisplay(): String
+ toString(): String
+ copy(): AbstractPointSaladCriterion
PointSaladCriterionFactory
+ PointSaladCriterionFactory ()
- getCriterionType(formattedString): CriterionType
- createCriterionOfType(formattedString, criterionType): ICriterion
+ createCriterionFromFormattedString(formattedString): ICriterion
Extends
PointSaladEvenOddCriterion
- evenPoints: int
- oddPoints: int
- vegetable: Vegetable
+ PointSaladEvenOddCriterion(vegetable, evenPoints, oddPoints)
+ PointSaladEvenOddCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladEvenOddCriterion
PointSaladMostCriterion
- pointsGranted: int
- vegetable: Vegetable
+ PointSaladMostCriterion(vegetable, points)
+ PointSaladMostCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladMostCriterion
PointSaladMostTotalCriterion
- pointsGranted: int
+ PointSaladMostTotalCriterion(points)
+ PointSaladMostTotalCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladMostTotalCriterion
PointSaladCompleteSetCriterion
- pointsGranted: int
+ PointSaladCompleteSetCriterion(points)
+ PointSaladCompleteSetCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladCompleteSetCriterion
PointSaladPerVeggieTypeCriterion
- minNumberOfEachVeggie: int
- pointsPerVeggieType: int
+ PointSaladPerVeggieTypeCriterion(minNumber, points)
+ PointSaladPerVeggieTypeCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladPerVeggieTypeCriterion
PointSaladPerMissingVeggieTypeCriterion
- pointsGranted: int
+ PointSaladPerMissingVeggieTypeCriterion(vegetable, points)
+ PointSaladPerMissingVeggieTypeCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladPerMissingVeggieTypeCriterion
PointSaladFewestCriterion
- pointsGranted: int
- vegetable: Vegetable
+ PointSaladFewestCriterion(vegetable, points)
+ PointSaladFewestCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladFewestCriterion
PointSaladFewestTotalCriterion
- pointsGranted: int
+ PointSaladFewestTotalCriterion(points)
+ PointSaladFewestTotalCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladFewestTotalCriterion
PointSaladCombinationCriterion
- pointsPerCombination: int
- vegetables: ArrayList<Vegetable>
+ PointSaladCombinationCriterion(vegetables, pointsPerCombination)
+ PointSaladCombinationCriterion(other)
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladCombinationCriterion
PointSaladPerVeggieCriterion
- pointsPerVeggie: ArrayList<Integer>
- vegetables: ArrayList<Vegetable>
+ PointSaladPerVeggieCriterion(vegetable, point)
+ PointSaladPerVeggieCriterion(vegetables, points)
+ PointSaladPerVeggieCriterion(other)
- checkValidCriterion()
+ computePlayerScore(playerHand, otherHands): int
+ toString(): String
+ copy(): PointSaladPerVeggieCriterion

Comment:

If a clean way to move this code in each criterion class is found, it should be done.
I have not found a "clean" way to do it yet, as static methods can not be abstract nor overridden.
A possible solution would be to create an instance method which creates a new criterion based on the formatted string, but it requires to first create a "blank" instance of the corresponding criterion class, which is not ideal.

<u>Fig 3:</u> Criteria class diagram



<u>Fig 4:</u> Exceptions class diagram

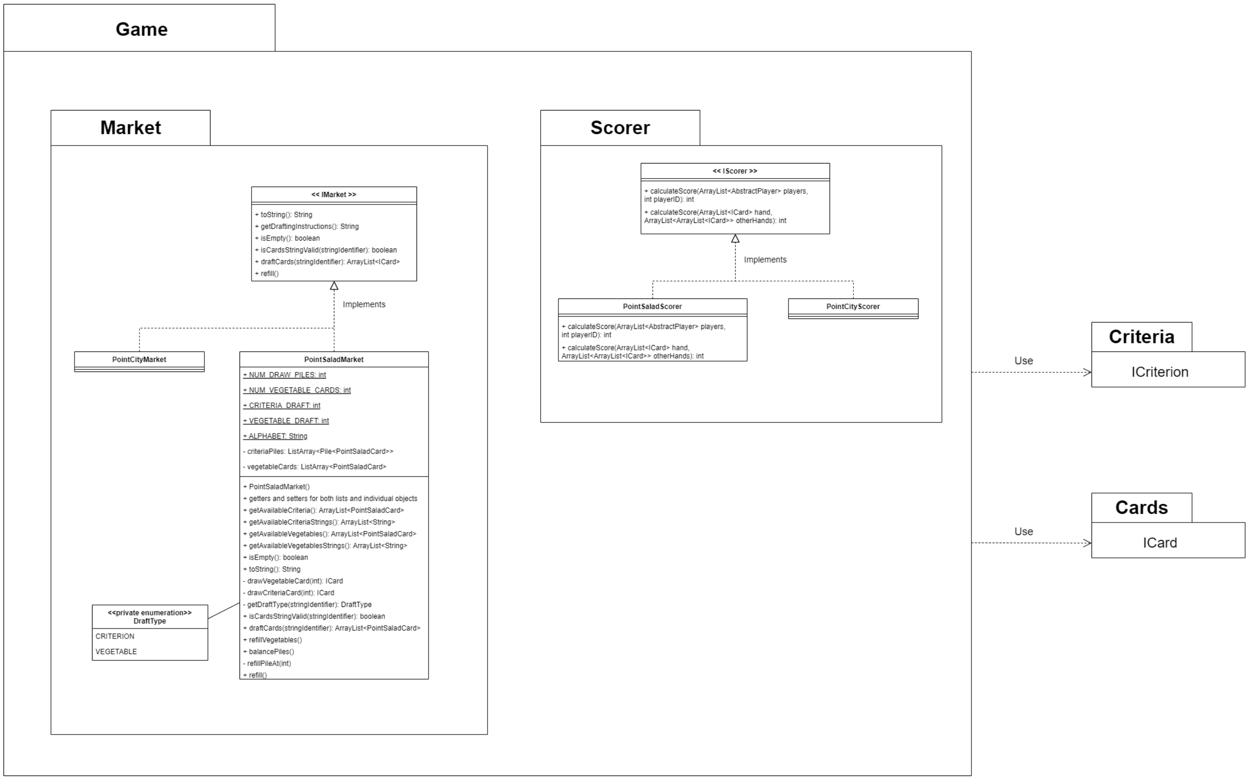

Game
Market
<< IMarket >>
+ toString(): String
+ getDraftingInstructions(): String
+ isEmpty(): boolean
+ isCardsStringValid(stringIdentifier): boolean
+ draftCards(stringIdentifier): ArrayList<ICard>
+ refill()
Implements
PointCityMarket
PointSaladMarket
+ NUM_DRAW_PILES: int
+ NUM_VEGETABLE_CARDS: int
+ CRITERIA_DRAFT: int
+ VEGETABLE_DRAFT: int
+ ALPHABET: String
- criteriaPiles: ListArray<Pile<PointSaladCard>>
- vegetableCards: ListArray<PointSaladCard>
+ PointSaladMarket()
+ getters and setters for both lists and individual objects
+ getAvailableCriteria(): ArrayList<PointSaladCard>
+ getAvailableCriteriaStrings(): ArrayList<String>
+ getAvailableVegetables(): ArrayList<PointSaladCard>
+ getAvailableVegetablesStrings(): ArrayList<String>
+ isEmpty(): boolean
+ toString(): String
- drawVegetableCard(int): ICard
- drawCriteriaCard(int): ICard
- getDraftType(stringIdentifier): DraftType
+ isCardsStringValid(stringIdentifier): boolean
+ draftCards(stringIdentifier): ArrayList<PointSaladCard>
+ refillVegetables()
+ balancePiles()
- refillPileAt(int)
+ refill()
<<private enumeration>>
DraftType
CRITERION
VEGETABLE
Scorer
<< IScorer >>
+ calculateScore(ArrayList<AbstractPlayer> players, int playerID): int
+ calculateScore(ArrayList<ICard> hand, ArrayList<ArrayList<ICard>> otherHands): int
Implements
PointSaladScorer
+ calculateScore(ArrayList<AbstractPlayer> players, int playerID): int
+ calculateScore(ArrayList<ICard> hand, ArrayList<ArrayList<ICard>> otherHands): int
PointCityScorer
Use
Criteria
ICriterion
Use
Cards
ICard

Fig 5: Game class diagram

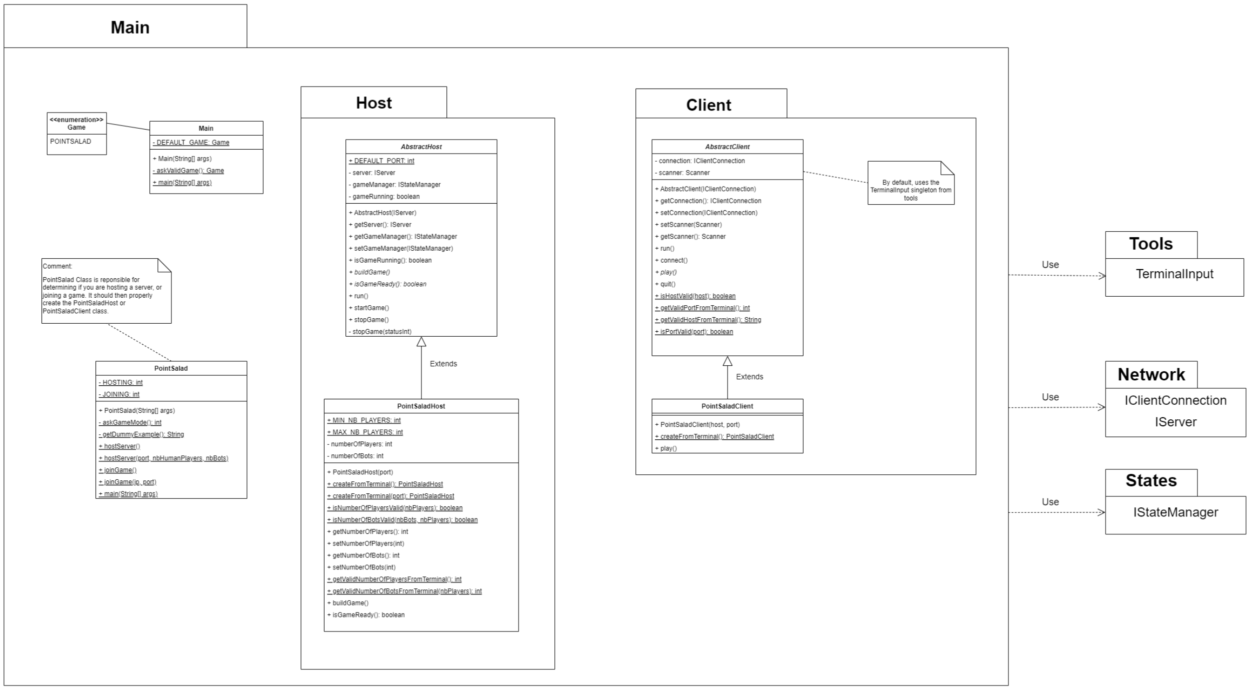

Main
<<enumeration>>
Game
POINTSALAD
Main
- DEFAULT_GAME: Game
+ Main(String[] args)
- askValidGame(): Game
+ main(String[] args)
Comment:
PointSalad Class is reponsible for determining if you are hosting a server, or joining a game. It should then properly create the PointSaladHost or PointSaladClient class.
PointSalad
- HOSTING: int
- JOINING: int
+ PointSalad(String[] args)
- askGameMode(): int
- getDummyExample(): String
+ hostServer()
+ hostServer(port, nbHumanPlayers, nbBots)
+ joinGame()
+ joinGame(ip, port)
+ main(String[] args)
Host
AbstractHost
+ DEFAULT_PORT: int
- server: IServer
- gameManager: IStateManager
- gameRunning: boolean
+ AbstractHost(IServer)
+ getServer(): IServer
+ getGameManager(): IStateManager
+ setGameManager(IStateManager)
+ isGameRunning(): boolean
+ buildGame()
+ isGameReady(): boolean
+ run()
+ startGame()
+ stopGame()
- stopGame(statusInt)
Extends
PointSaladHost
+ MIN_NB_PLAYERS: int
+ MAX_NB_PLAYERS: int
- numberOfPlayers: int
- numberOfBots: int
+ PointSaladHost(port)
+ createFromTerminal(): PointSaladHost
+ createFromTerminal(port): PointSaladHost
+ isNumberOfPlayersValid(nbPlayers): boolean
+ isNumberOfBotsValid(nbBots, nbPlayers): boolean
+ getNumberOfPlayers(): int
+ setNumberOfPlayers(int)
+ getNumberOfBots(): int
+ setNumberOfBots(int)
+ getValidNumberOfPlayersFromTerminal(): int
+ getValidNumberOfBotsFromTerminal(nbPlayers): int
+ buildGame()
+ isGameReady(): boolean
Client
AbstractClient
- connection: IClientConnection
- scanner: Scanner
+ AbstractClient(IClientConnection)
+ getConnection(): IClientConnection
+ setConnection(IClientConnection)
+ setScanner(Scanner)
+ getScanner(): Scanner
+ run()
+ connect()
+ play()
+ quit()
+ isHostValid(host): boolean
+ getValidPortFromTerminal(): int
+ getValidHostFromTerminal(): String
+ isPortValid(port): boolean
By default, uses the TerminalInput singleton from tools
Extends
PointSaladClient
+ PointSaladClient(host, port)
+ createFromTerminal(): PointSaladClient
+ play()
Use
Tools
TerminalInput
Use
Network
IClientConnection
IServer
Use
States
IStateManager

Fig 6: Main class diagram

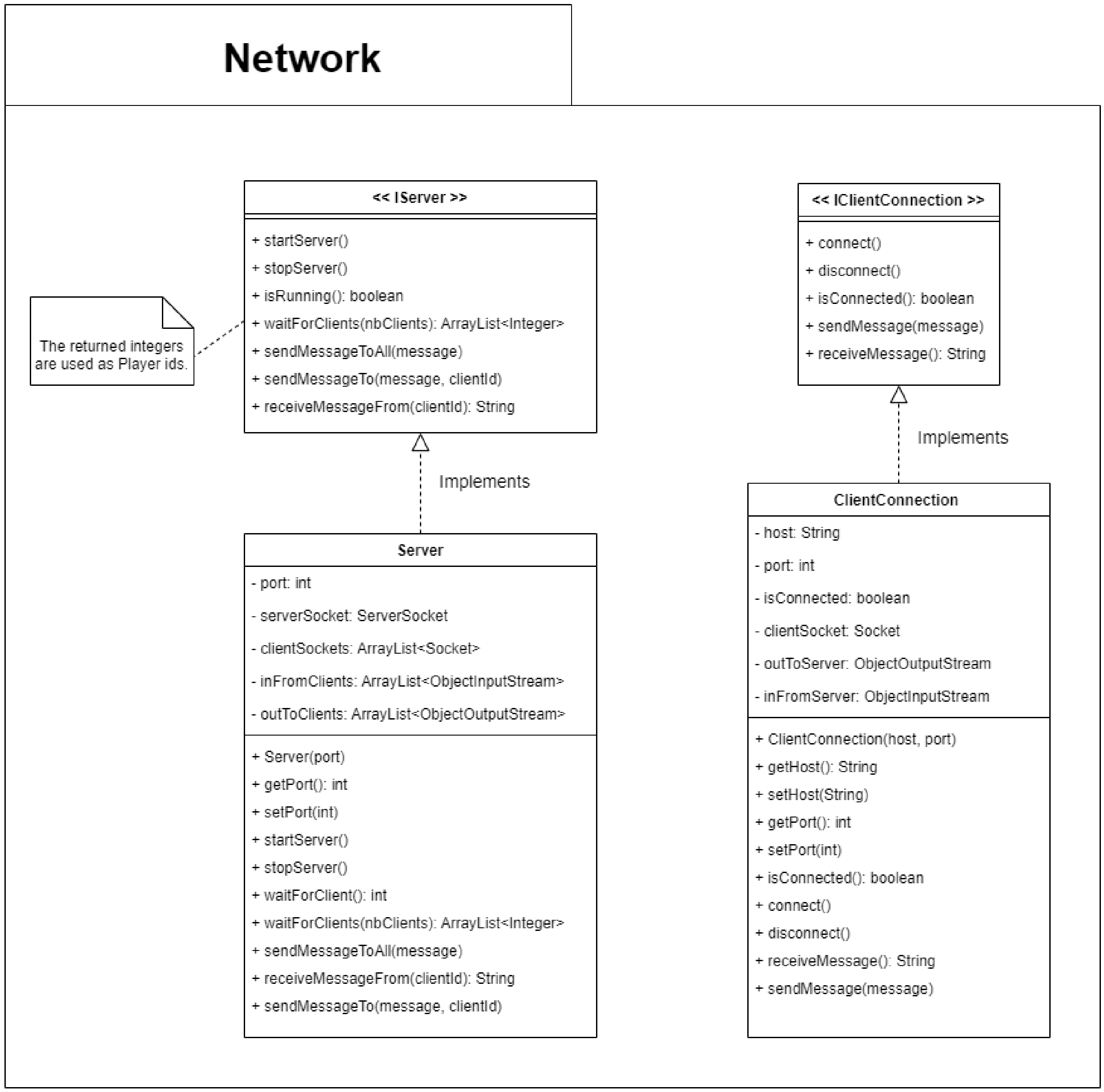

Network
<< IServer >>
+ startServer()
+ stopServer()
+ isRunning(): boolean
+ waitForClients(nbClients): ArrayList<Integer>
+ sendMessageToAll(message)
+ sendMessageTo(message, clientId)
+ receiveMessageFrom(clientId): String
The returned integers are used as Player ids.
Implements
Server
- port: int
- serverSocket: ServerSocket
- clientSockets: ArrayList<Socket>
- inFromClients: ArrayList<ObjectInputStream>
- outToClients: ArrayList<ObjectOutputStream>
+ Server(port)
+ getPort(): int
+ setPort(int)
+ startServer()
+ stopServer()
+ waitForClient(): int
+ waitForClients(nbClients): ArrayList<Integer>
+ sendMessageToAll(message)
+ receiveMessageFrom(clientId): String
+ sendMessageTo(message, clientId)
<< IClientConnection >>
+ connect()
+ disconnect()
+ isConnected(): boolean
+ sendMessage(message)
+ receiveMessage(): String
Implements
ClientConnection
- host: String
- port: int
- isConnected: boolean
- clientSocket: Socket
- outToServer: ObjectOutputStream
- inFromServer: ObjectInputStream
+ ClientConnection(host, port)
+ getHost(): String
+ setHost(String)
+ getPort(): int
+ setPort(int)
+ isConnected(): boolean
+ connect()
+ disconnect()
+ receiveMessage(): String
+ sendMessage(message)

Fig 7: Network class diagram

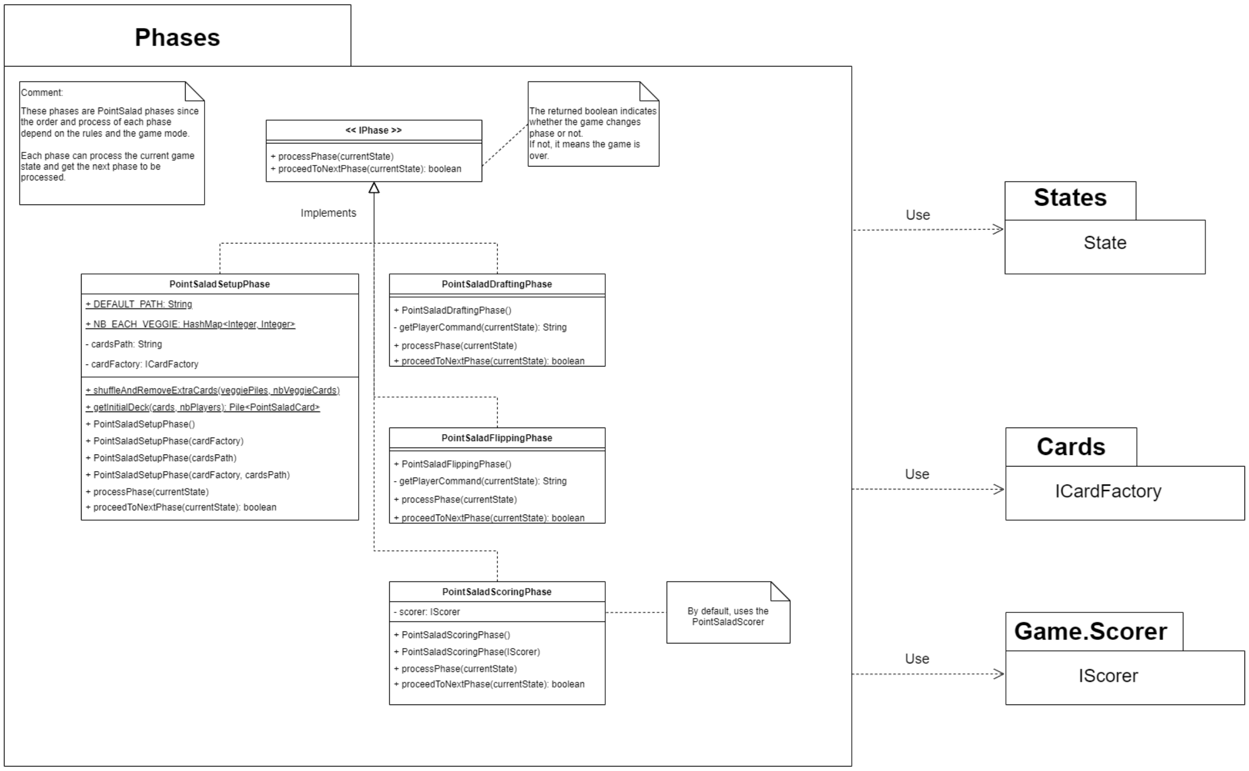

Phases
Comment:
These phases are PointSalad phases since the order and process of each phase depend on the rules and the game mode.
Each phase can process the current game state and get the next phase to be processed.
<< IPhase >>
+ processPhase(currentState)
+ proceedToNextPhase(currentState): boolean
The returned boolean indicates whether the game changes phase or not.
If not, it means the game is over.
Implements
PointSaladSetupPhase
+ DEFAULT_PATH: String
+ NB_EACH_VEGGIE: HashMap<Integer, Integer>
- cardsPath: String
- cardFactory: ICardFactory
+ shuffleAndRemoveExtraCards(veggiePiles, nbVeggieCards)
+ getInitialDeck(cards, nbPlayers): Pile<PointSaladCard>
+ PointSaladSetupPhase()
+ PointSaladSetupPhase(cardFactory)
+ PointSaladSetupPhase(cardsPath)
+ PointSaladSetupPhase(cardFactory, cardsPath)
+ processPhase(currentState)
+ proceedToNextPhase(currentState): boolean
PointSaladDraftingPhase
+ PointSaladDraftingPhase()
- getPlayerCommand(currentState): String
+ processPhase(currentState)
+ proceedToNextPhase(currentState): boolean
PointSaladFlippingPhase
+ PointSaladFlippingPhase()
- getPlayerCommand(currentState): String
+ processPhase(currentState)
+ proceedToNextPhase(currentState): boolean
PointSaladScoringPhase
- scorer: IScorer
+ PointSaladScoringPhase()
+ PointSaladScoringPhase(IScorer)
+ processPhase(currentState)
+ proceedToNextPhase(currentState): boolean
By default, uses the PointSaladScorer
Use
States
State
Use
Cards
ICardFactory
Use
Game.Scorer
IScorer

Fig 8: Phases class diagram

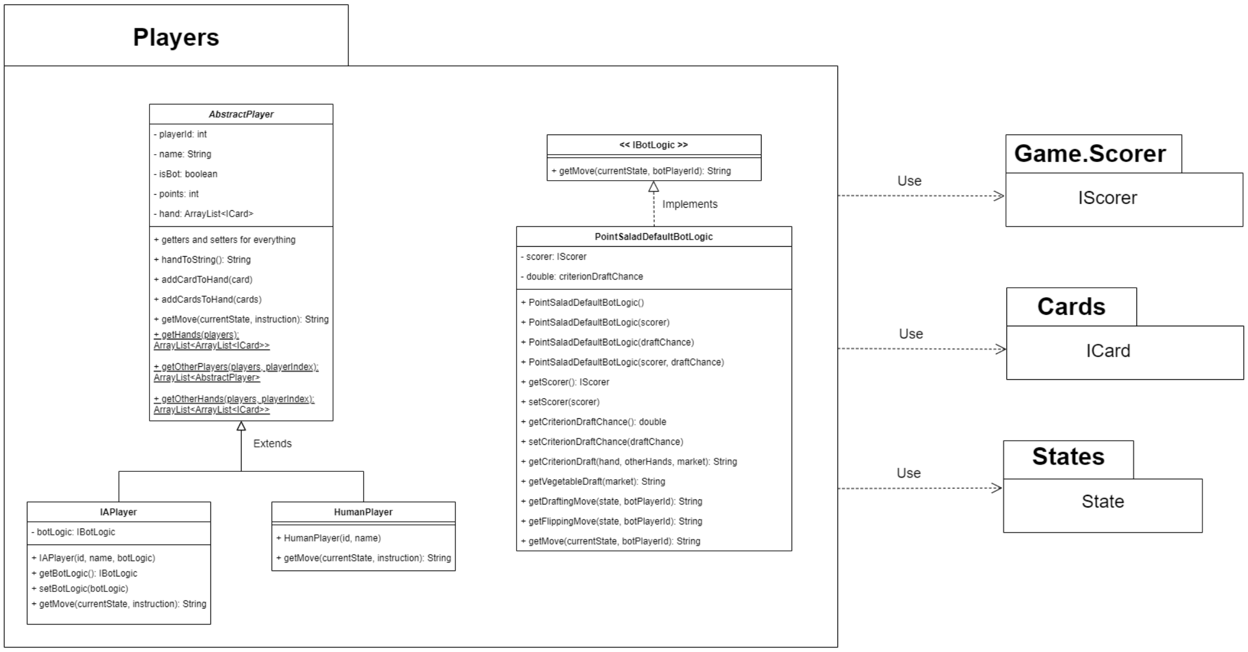

Players
AbstractPlayer
- playerId: int
- name: String
- isBot: boolean
- points: int
- hand: ArrayList<ICard>
+ getters and setters for everything
+ handToString(): String
+ addCardToHand(card)
+ addCardsToHand(cards)
+ getMove(currentState, instruction): String
+ getHands(players): ArrayList<ArrayList<ICard>>
+ getOtherPlayers(players, playerIndex): ArrayList<AbstractPlayer>
+ getOtherHands(players, playerIndex): ArrayList<ArrayList<ICard>>
Extends
IAPlayer
- botLogic: IBotLogic
+ IAPlayer(id, name, botLogic)
+ getBotLogic(): IBotLogic
+ setBotLogic(botLogic)
+ getMove(currentState, instruction): String
HumanPlayer
+ HumanPlayer(id, name)
+ getMove(currentState, instruction): String
<< IBotLogic >>
+ getMove(currentState, botPlayerId): String
Implements
PointSaladDefaultBotLogic
- scorer: IScorer
- double: criterionDraftChance
+ PointSaladDefaultBotLogic()
+ PointSaladDefaultBotLogic(scorer)
+ PointSaladDefaultBotLogic(draftChance)
+ PointSaladDefaultBotLogic(scorer, draftChance)
+ getScorer(): IScorer
+ setScorer(scorer)
+ getCriterionDraftChance(): double
+ setCriterionDraftChance(draftChance)
+ getCriterionDraft(hand, otherHands, market): String
+ getVegetableDraft(market): String
+ getDraftingMove(state, botPlayerId): String
+ getFlippingMove(state, botPlayerId): String
+ getMove(currentState, botPlayerId): String
Use
Game.Scorer
IScorer
Use
Cards
ICard
Use
States
State

Fig 9: Players class diagram

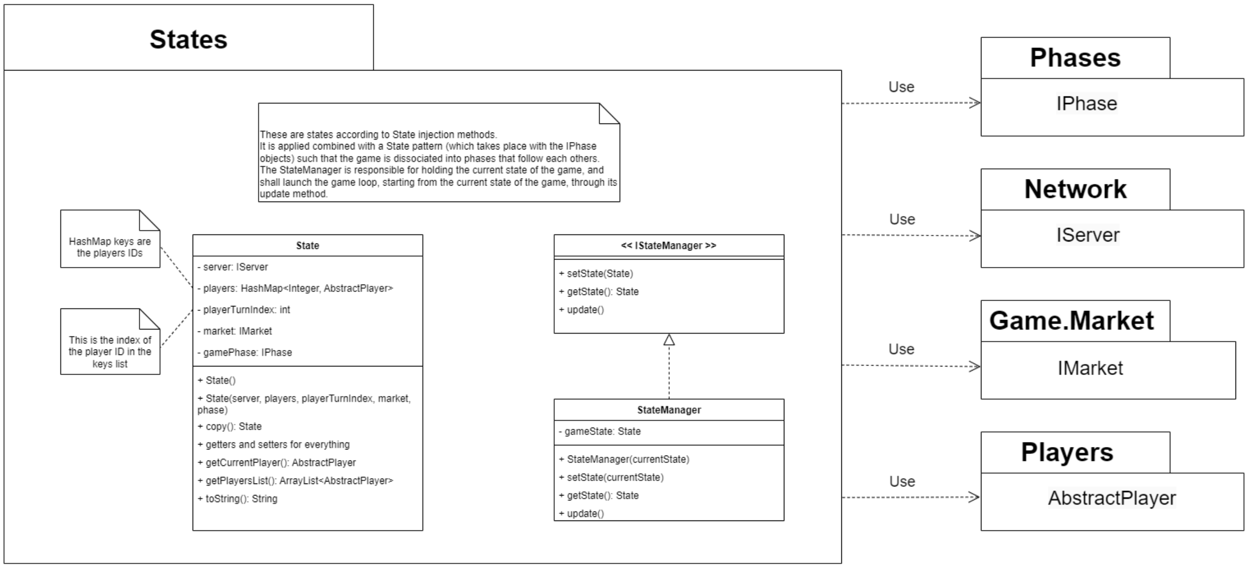

States
These are states according to State injection methods.
It is applied combined with a State pattern (which takes place with the IPhase objects) such that the game is dissociated into phases that follow each others.
The StateManager is responsible for holding the current state of the game, and shall launch the game loop, starting from the current state of the game, through its update method.
HashMap keys are the players IDs
This is the index of the player ID in the keys list
State
- server: IServer
- players: HashMap<Integer, AbstractPlayer>
- playerTurnIndex: int
- market: IMarket
- gamePhase: IPhase
+ State()
+ State(server, players, playerTurnIndex, market, phase)
+ copy(): State
+ getters and setters for everything
+ getCurrentPlayer(): AbstractPlayer
+ getPlayersList(): ArrayList<AbstractPlayer>
+ toString(): String
<< IStateManager >>
+ setState(State)
+ getState(): State
+ update()
StateManager
- gameState: State
+ StateManager(currentState)
+ setState(currentState)
+ getState(): State
+ update()
Use
Phases
IPhase
Use
Network
IServer
Use
Game.Market
IMarket
Use
Players
AbstractPlayer

Fig 10: States class diagram

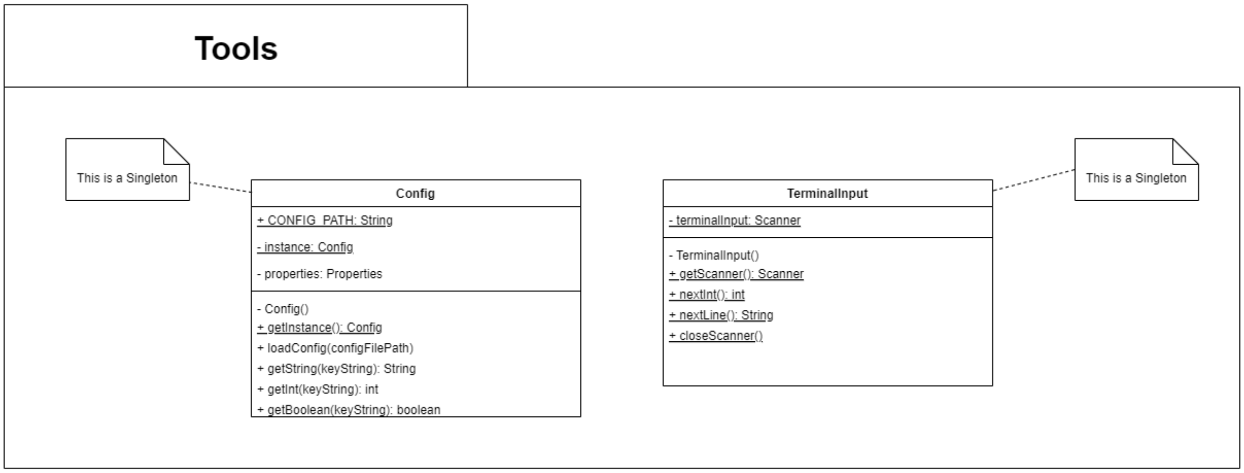


Fig 11: Tools class diagram