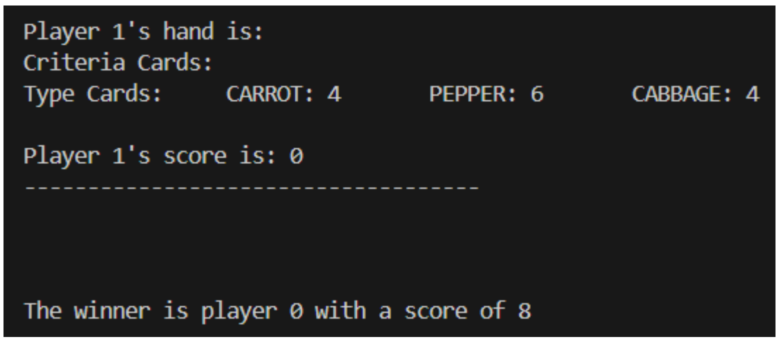
D7032E
Home Exam



```
Player 1's hand is:
Criteria Cards:
Type Cards:     CARROT: 4       PEPPER: 6       CABBAGE: 4

Player 1's score is: 0
------------------------------------



The winner is player 0 with a score of 8
```

Anton Follinger
antfol-1@student.ltu.se
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

October 22, 2024

Contents

1 Unit testing

Rule ID	Passing	Testability
1	×	somewhat
2	✓	yes
3	×	somewhat
4	✓	yes
5	✓	yes
6	✓	yes
7	✓	yes
8	✓	no
9	✓	no
10	✓	yes
11	✓	yes
12	✓	no
13	✓	yes
14	✓	yes

- When it comes to rule number 1 if the amount of player is less than 2 there will be a *IndexOutOfBoundsException* (if and only if the combination of bots and players is less than 2). The problem with testing the code is the lack of error handling which means it work with player larger than 6 (example one player and 6 bots).
- For rule number 3 the amount of cards is correct and can also be tested using a simple *assertequal* function comparing the piles to a predetermined value calculated manually. When it comes to picking random cards for removal/dividing it is not fulfilled, and the testability of randomness is not possible either. The reason that randomness can not be tested depends on that it is random, you could argue that with statistic we can be almost certain that a large value of input and outputs is random but never 100 percent.
- For rule number 8 even if the rule is working when using the correct input it still has a bug when writing the incorrect input.

2 Software Architecture design, Design Patterns and refactoring

The main objective for creating the new architecture is priority on **Modifiability**, **Extensibility** (requirement 15), and **Testability**. In order to acigheve this their are several different methods but firstly I will focus on fulfilling the *SOLID principles* and *Booch metrics.*

2.1 SOLID principles and Booch metrics

When designing the new software architecture our main starting points correlate closely to the two first points of *SOLID principles* and *Booch metrics.*

1. We want to prioritize *single responsibility principle*, we can do this by dividing the main code into different modules and class with their own responsibility while keep full functionality by acting together, this result in a code where each part is easily identified and there fore also easy to modify and test.
2. We want to prioritize *open/closed principle*, we can do this by implementing *abstract classes* and there for subclasses. This allow us to easily extend our code without changing the core which lead us to implement additional games such as **Point City**.
    - During this point it is also very important to apply the the two principle of *Liskov's Substitution* and *dependency inversion* as they are a foundation for creating working superclasses and subclasses for our priorities of **modifiability**, **extensibility**.
3. We want to prioritize *high cohesion* but the first choice is not directly creating this as *Logical Cohesion* will be used for the superclasses. In order to create *high cohesion* the use of *Functional Cohesion* is applied inside these *Logical Cohesion* often described as *Layered architecture.* This directly related to, for example, testability of the code as we can pinpoint the *Logical Cohesion* parts and test them individually. This *Logical Cohesion* also result in "parts" that could be changed to preform another task.
4. We want to prioritize *low coupling* this is done by using a combination of *data coupling* and *stamp coupling* which means modules interact with each other as as independently as possible.
    - An important part of applying these keeping the *Sufficiency*, *Completeness* and *Primitiveness.* What this refers to is the importance of creating code that is fully functional without overengineering and keeping the code primitive.

2.2 Design patterns

2.2.1 Mediator Pattern

I believe that the use of *Mediator Pattern* will be suitable for this projects as it main characteristics of having a central class coordinates interactions between different classes where the Mediator class controls the flow and delegates the responsibilities to others. This will be good for the refactoring as we can create a new Mediator class when implementing new features (as long as it follow the superclass) with new implemented classes with different responsibilities. This Pattern is therefore a superb option to reach **modifiability**, **extensibility** and **Testability**.

2.2.2 Facade Pattern

The *Facade Pattern* will also be used as it creates a simple interface (the main class) to a complex system of classes. The main class would instantiate and pass objects to these lower-level classes (which are the subsystems) to perform their own operations on the object. This addition would not necessary help with **modifiability**, **extensibility** and **testability** but is important for user usage. We could also use Subsystem Facades if any subclass will use user inputs.

2.3 Software Architecture design

The combination of these *design patterns* would create something resembling a *Layered architecture.* This could be described using diagram 1.

2.3.1 Layered architecture with mediated communication

The Layered architecture would combine the two design patterns mentioned before, this would result in 3 distinct layers. The first layer is the Main *Facade* layer which will take inputs in order to start the *Mediator* class/layer (maybe a game Eninge or game initialiser) which will distribute initialised object using *stamp coupling* to other Subsystem (each could have their own facades layer if neascery) that would preform their *Single Responsibility.* This approach would lead to high **modifiability**, **extensibility** and **testability** as each part could be changed/added with new subsystems or layers but still keep the core functionality (most classes will probably have abstract classes with specific rules). There would also be possibility for testing each subsystem individually as well as change them for other subsystems. This approach would also fullfill several of the *SOLID principles and Booch metrics* mentioned before.

3 Conclusion

In conclusion, the redesigned architecture prioritizes **modifiability**, **extensibility**, and **testability** by adhering to SOLID principles, leveraging the Mediator and Facade patterns, and utilizing a layered approach with low coupling and high cohesion. These design choices will be applied to the the code and refactoring process, ensuring the system remains flexible, maintainable, and scalable while simplifying interactions and improving modularity.

3.1 Diagrams

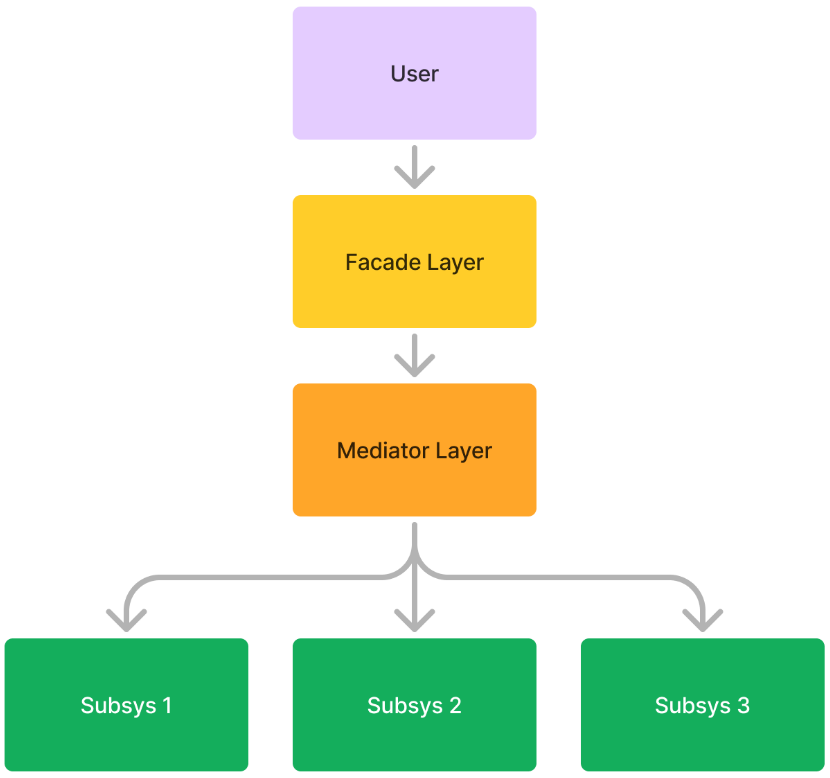


Figure 1: Layered

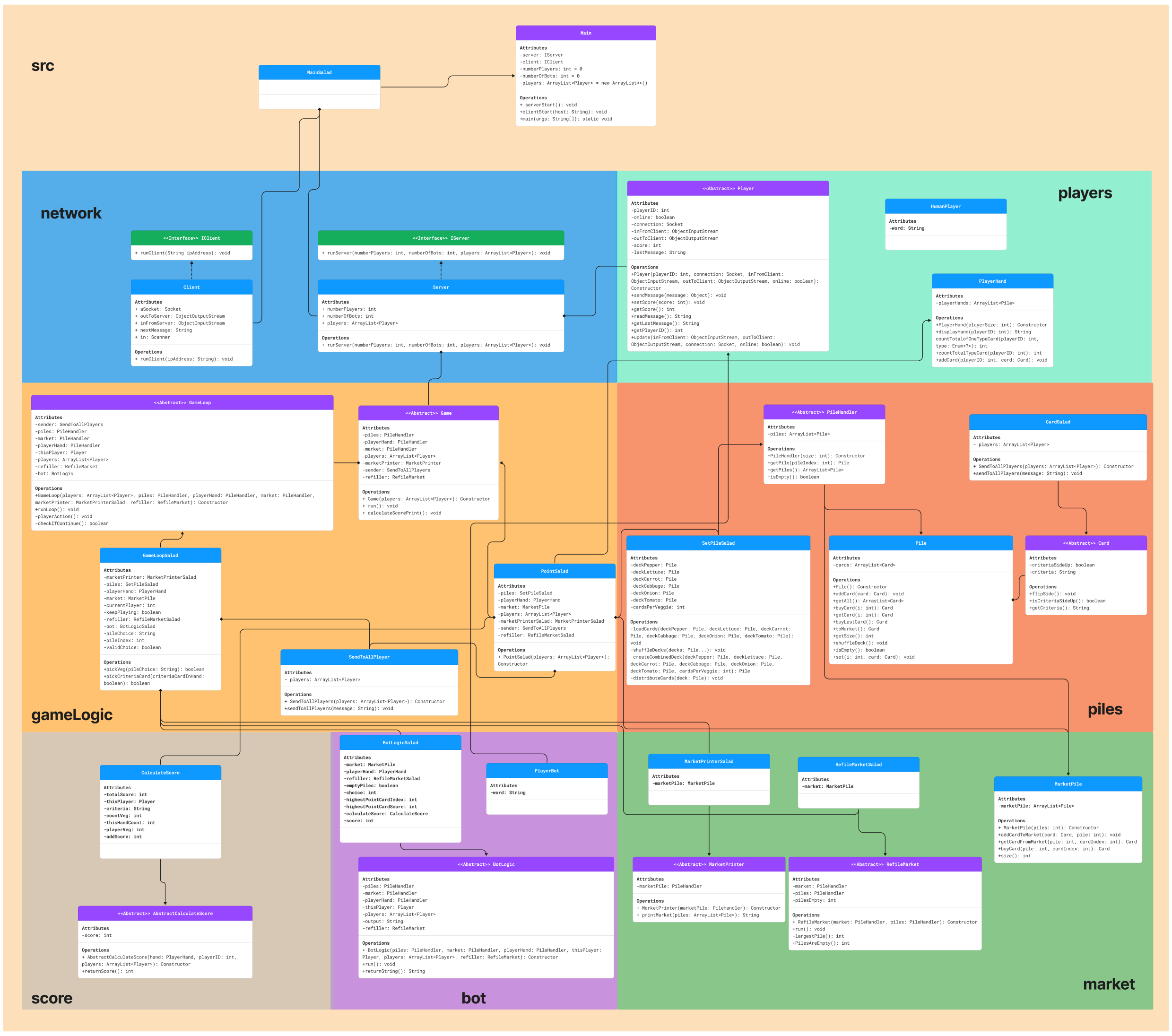

src
MainSalad
Main
Attributes
-server: IServer
-client: IClient
-numberPlayers: int = 0
-numberOfBots: int = 0
-players: ArrayList<Player> = new ArrayList<>()
Operations
+ serverStart(): void
+clientStart(host: String): void
+main(args: String[]): static void
network
<<Interface>> IClient
+ runClient(String ipAddress): void
Client
Attributes
+ aSocket: Socket
+ outToServer: ObjectOutputStream
+ inFromServer: ObjectInputStream
+ nextMessage: String
+ in: Scanner
Operations
+ runClient(ipAddress: String): void
<<Interface>> IServer
+ runServer(numberPlayers: int, numberOfBots: int, players: ArrayList<Player>): void
Server
Attributes
+ numberPlayers: int
+ numberOfBots: int
+ players: ArrayList<Player>
Operations
+ runServer(numberPlayers: int, numberOfBots: int, players: ArrayList<Player>): void
players
<<Abstract>> Player
Attributes
-playerID: int
-online: boolean
-connection: Socket
-inFromClient: ObjectInputStream
-outToClient: ObjectOutputStream
-score: int
-lastMessage: String
Operations
+Player(playerID: int, connection: Socket, inFromClient: ObjectInputStream, outToClient: ObjectOutputStream, online: boolean): Constructor
+sendMessage(message: Object): void
+setScore(score: int): void
+getScore(): int
+readMessage(): String
+getLastMessage(): String
+getPlayerID(): int
+update(inFromClient: ObjectInputStream, outToClient: ObjectOutputStream, connection: Socket, online: boolean): void
HumanPlayer
Attributes
-word: String
PlayerHand
Attributes
-playerHands: ArrayList<Pile>
Operations
+PlayerHand(playerSize: int): Constructor
+displayHand(playerID: int): String
countTotalofOneTypeCard(playerID: int, type: Enum<?>): int
+countTotalTypeCard(playerID: int): int
+addCard(playerID: int, card: Card): void
gameLogic
<<Abstract>> GameLoop
Attributes
-sender: SendToAllPlayers
-piles: PileHandler
-market: PileHandler
-playerHand: PileHandler
-thisPlayer: Player
-players: ArrayList<Player>
-refiller: RefileMarket
-bot: BotLogic
Operations
+GameLoop(players: ArrayList<Player>, piles: PileHandler, playerHand: PileHandler, market: PileHandler, marketPrinter: MarketPrinterSalad, refiller: RefileMarket): Constructor
+runLoop(): void
-playerAction(): void
-checkIfContinue(): boolean
<<Abstract>> Game
Attributes
-piles: PileHandler
-playerHand: PileHandler
-market: PileHandler
-players: ArrayList<Player>
-marketPrinter: MarketPrinter
-sender: SendToAllPlayers
-refiller: RefileMarket
Operations
+ Game(players: ArrayList<Player>): Constructor
+ run(): void
+ calculateScorePrint(): void
GameLoopSalad
Attributes
-marketPrinter: MarketPrinterSalad
-piles: SetPileSalad
-playerHand: PlayerHand
-market: MarketPile
-currentPlayer: int
-keepPlaying: boolean
-refiller: RefileMarketSalad
-bot: BotLogicSalad
-pileChoice: String
-pileIndex: int
-validChoice: boolean
Operations
+pickVeg(pileChoice: String): boolean
+pickCriteriaCard(criteriaCardInHand: boolean): boolean
PointSalad
Attributes
-piles: SetPileSalad
-playerHand: PlayerHand
-market: MarketPile
-players: ArrayList<Player>
-marketPrinterSalad: MarketPrinterSalad
-sender: SendToAllPlayers
-refiller: RefileMarketSalad
Operations
+ PointSalad(players: ArrayList<Player>): Constructor
SendToAllPlayer
Attributes
- players: ArrayList<Player>
Operations
+ SendToAllPlayers(players: ArrayList<Player>): Constructor
+sendToAllPlayers(message: String): void
piles
<<Abstract>> PileHandler
Attributes
-piles: ArrayList<Pile>
Operations
+PileHandler(size: int): Constructor
+getPile(pileIndex: int): Pile
+getPiles(): ArrayList<Pile>
+isEmpty(): boolean
CardSalad
Attributes
- players: ArrayList<Player>
Operations
+ SendToAllPlayers(players: ArrayList<Player>): Constructor
+sendToAllPlayers(message: String): void
SetPileSalad
Attributes
-deckPepper: Pile
-deckLettuce: Pile
-deckCarrot: Pile
-deckCabbage: Pile
-deckOnion: Pile
-deckTomato: Pile
-cardsPerVeggie: int
Operations
-loadCards(deckPepper: Pile, deckLettuce: Pile, deckCarrot: Pile, deckCabbage: Pile, deckOnion: Pile, deckTomato: Pile): void
-shuffleDecks(decks: Pile...): void
-createCombinedDeck(deckPepper: Pile, deckLettuce: Pile, deckCarrot: Pile, deckCabbage: Pile, deckOnion: Pile, deckTomato: Pile, cardsPerVeggie: int): Pile
-distributeCards(deck: Pile): void
Pile
Attributes
-cards: ArrayList<Card>
Operations
+Pile(): Constructor
+addCard(card: Card): void
+getAll(): ArrayList<Card>
+buyCard(i: int): Card
+getCard(i: int): Card
+buyLastCard(): Card
+toMarket(): Card
+getSize(): int
+shuffleDeck(): void
+isEmpty(): boolean
+set(i: int, card: Card): void
<<Abstract>> Card
Attributes
-criteriaSideUp: boolean
-criteria: String
Operations
+flipSide(): void
+isCriteriaSideUp(): boolean
+getCriteria(): String
score
CalculateScore
Attributes
-totalScore: int
-thisPlayer: Player
-criteria: String
-countVeg: int
-thisHandCount: int
-playerVeg: int
-addScore: int
<<Abstract>> AbstractCalculateScore
Attributes
-score: int
Operations
+ AbstractCalculateScore(hand: PlayerHand, playerID: int, players: ArrayList<Player>): Constructor
+returnScore(): int
bot
BotLogicSalad
Attributes
-market: MarketPile
-playerHand: PlayerHand
-refiller: RefileMarketSalad
-emptyPiles: boolean
-choice: int
-highestPointCardIndex: int
-highestPointCardScore: int
-calculateScore: CalculateScore
-score: int
PlayerBot
Attributes
-word: String
<<Abstract>> BotLogic
Attributes
-piles: PileHandler
-market: PileHandler
-playerHand: PileHandler
-thisPlayer: Player
-players: ArrayList<Player>
-output: String
-refiller: RefileMarket
Operations
+ BotLogic(piles: PileHandler, market: PileHandler, playerHand: PileHandler, thisPlayer: Player, players: ArrayList<Player>, refiller: RefileMarket): Constructor
+run(): void
+returnString(): String
market
MarketPrinterSalad
Attributes
-marketPile: MarketPile
RefileMarketSalad
Attributes
-market: MarketPile
MarketPile
Attributes
-marketPile: ArrayList<Pile>
Operations
+ MarketPile(piles: int): Constructor
+addCardToMarket(card: Card, pile: int): void
+getCardFromMarket(pile: int, cardIndex: int): Card
+buyCard(pile: int, cardIndex: int): Card
+size(): int
<<Abstract>> MarketPrinter
Attributes
-marketPile: PileHandler
Operations
+ MarketPrinter(marketPile: PileHandler): Constructor
+ printMarket(piles: ArrayList<Pile>): String
<<Abstract>> RefileMarket
Attributes
-market: PileHandler
-piles: PileHandler
-pilesEmpty: int
Operations
+ RefileMarket(market: PileHandler, piles: PileHandler): Constructor
+run(): void
-largestPile(): int
+PilesAreEmpty(): int

Figure 2: UML