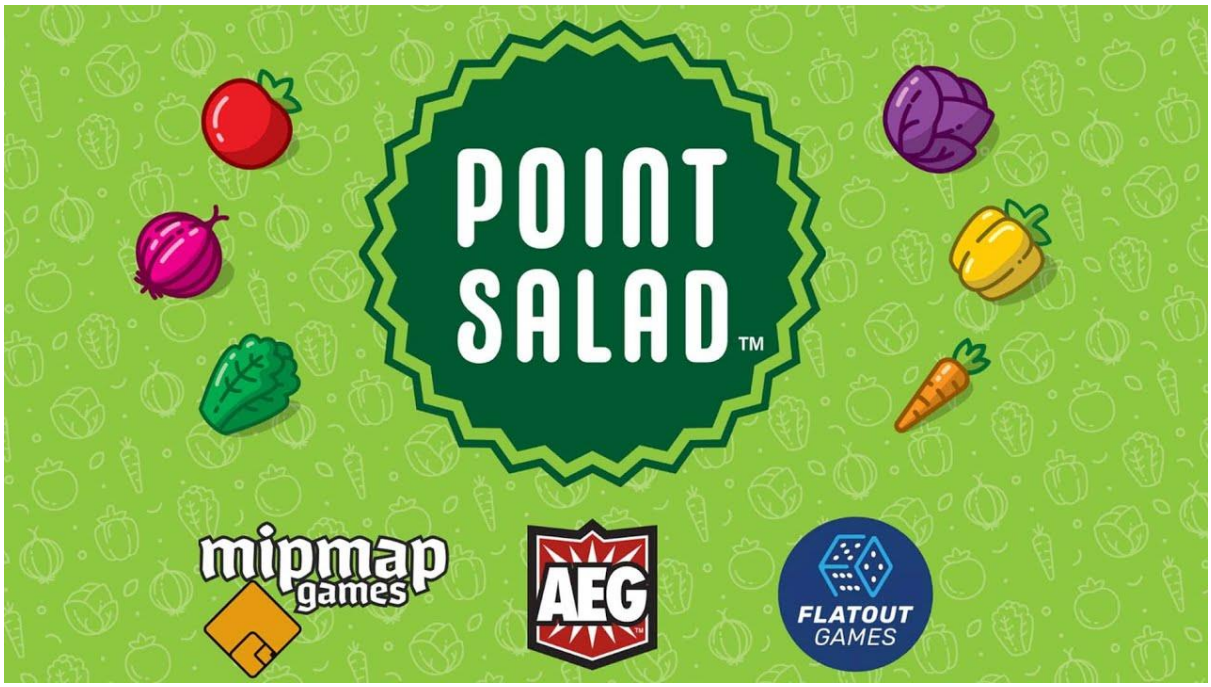
D7032E Home Exam

Point Salad

Cai Hongqi – 22 October 2024

Contents



Unit testing

Unfulfilled Requirements: 1, 6, 11

- Requirement 1: 0 players crash the game, while 7 or more can continue to play.
    a. It is not possible to use Junit assert to test. If the game reads an invalid number of players (e.g., 1 player), it will likely produce its own error before reaching the assertion check. This is because the game logic is executed immediately after the input is read, and any errors that occur during this process will prevent the assertions from being executed.

- Requirement 3: Odd number mismatch
    a. Using division when there is an odd number of players performs a floor division, negating the decimal point and thus allocating the wrong number of cards for 3 and 5 players. It is not possible to use Junit assert as the player's hand is tightly coupled with the game's logic code.

- Requirement 10: The point card that was flipped to form the vegetable market is different than from the previous topmost card.
    a. It is not possible to use Junit assert to test as the game requires a full playthrough before any of the assert statements are executed.

- Requirement 11: I am not allowed to continue drawing from one of the piles if it becomes empty. I should be able to draw from the bottom of the draw pile with the most cards.
    a. It is not possible to use Junit assert to test as the game will never run the assert line when it runs out of cards and will prompt the user that a card can no longer be drawn from the empty pile.

In summary, the game logic is tightly coupled with the initialization process, making it difficult to reach the assert statements in the tests without running and finishing the entire game. By then, all variables would've been freed and thus difficult to pinpoint the exact locations in the code that cause the requirements to fail.

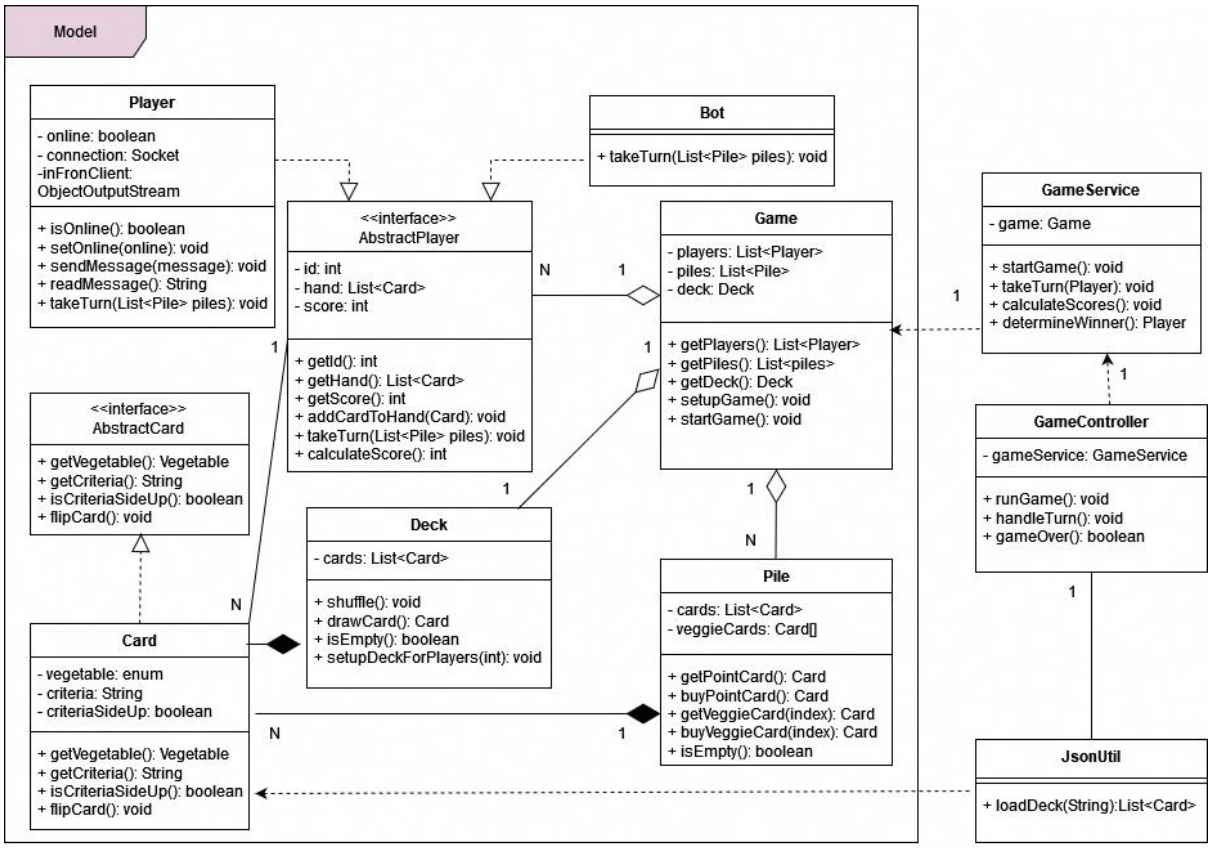
Software Architecture design and refactoring

Modularity and Layering

The system will be divided into 4 main layers:

- Models
- Services
- Controllers
- Utilities

UML Diagram of refactored code:



Models:

The model represents the data logic and data that is stored in the PointSalad application. It contains the classes Player, Game, Card, Deck and Pile.

- Player (interface) class:
    - Represents a player or bot's hand and score, also handles actions to select and manipulate cards (e.g., flipping a point card to the vegetable side).
- Player (concrete) class:
    - Represents player and bot-specific actions such as socket connections and messages.
- Game class:

    - Manages the state and rules of the PointSalad game, overseeing players, piles, and the deck of cards.
- Card (interface) class:
    - A single card object. Each card has two sides (vegetable and point side).
- Card (concrete) class:
    - A card can either be:
        - VegetableCard (for the vegetable side)
        - PointCard (for the point criteria side)
    - Allows easy flipping between the 2 sides.
- Deck class: Encapsulates deck management (shuffling, card distribution, card removal). Also supports modular deck management based on the number of players (e.g., removing cards for fewer players).
- Pile class: Represents a pile of cards in the game. It manages the cards in the pile and the veggie cards laid out in front of the pile.

Game Service:

The GameService class contains the business logic and rules of the game. It manages game operations like shuffling, dealing cards, and calculating scores. It is also contains the method to start the game by setting up the game and initializing the game state.

- `void startGame(int nrPlayers)`: Starts the game by setting up the game and initializing the game state.
- `void takeTurn(Player player)`: Manages the logic for a player's turn.
- `void calculateScores()`: Calculates the scores for all players.

Game Controller:

The GameController class manages user interactions and the game flow. It handles the game loop and player interactions. It manages the current game service instance. In the future if more game sessions are held concurrently, it only needs to call the GameController class.

- `GameController(GameService gameService)`: Constructor to initialize the controller with a game service instance.
- `void runGame(int nrPlayers)`: Runs the game by starting it and managing the game loop.
- `boolean gameOver()`: Checks if the game is over by verifying if all piles are empty.

Utilities:

The JsonUtil class contains shared utility functions for JSON parsing and loading like random number generation, file handling (for manifest reading), and constants (e.g., types of vegetables, point criteria).

- `static List<Card> loadDeck(String filePath)`: Loads the deck of cards from a JSON file and returns it as a list of Card objects.

Addressing Quality Attribute Requirements

Modifiability and Extensibility

- The use of interfaces like AbstractPlayer and AbstractCard supports modifiability and extensibility. For example, if new types of cards (e.g., special vegetable cards or additional criteria for scoring) are needed, new classes can simply implement the AbstractCard interface without modifying existing code.
- Similarly, Bot and Player both implement the AbstractPlayer interface, allowing for easy integration of additional player types in the future, such as different bot strategies or new player behaviors.
- The GameService and GameController classes are separated, allowing game rules or logic to be updated in GameService without affecting the controller. This separation of concerns enhances extensibility when implementing additional game features like the "Point City" variant.
- The Deck and Pile classes are designed to be independent, which allows easy modification or addition of game rules that manipulate cards, decks, or piles.

Testability

- Interfaces such as AbstractPlayer and AbstractCard increase testability by enabling the use of mock objects during unit testing. For example, different card types or player behaviors can be simulated without needing a full game environment.
- The modular design of GameService, GameController, and Deck ensures that each component can be tested individually. For instance, the game logic in GameService can be tested independently of UI or input/output handling in GameController.

Design Patterns

- Strategy Pattern (for Player Types):
    - The Player and Bot classes are concrete implementations of the AbstractPlayer interface. The Strategy design pattern is applied here because different types of players (e.g., human players or AI players) can be injected dynamically based on game rules. This pattern enhances the flexibility of player behaviors without changing core game logic.
- Factory Pattern (for Cards):
    - A factory pattern is used to instantiate cards (CardFactory). This design pattern enables the game to dynamically generate different types of cards, making the system more extendable if new types of cards (with unique behaviors or rules) need to be added in the future.
- Observer Pattern (for Turn Management):

    - The game flow between GameService and GameController shows an Observer pattern. The GameController is observing the GameService for changes such as turns or game state updates, ensuring that the system responds to changes in a controlled and modular way.

- Singleton Pattern (for GameService or Deck):
    - GameService implements a Singleton pattern, ensuring that there is only one instance managing the game's state and rules. This would prevent multiple instances from interfering with the game logic, improving stability and state management. Although this may prevent multiple instances or “lobbies” of the game, I assume that only 1 instance of the game is needed at any given time.