**Deliverable 2**

**Team Members:**

Nupur Dhiman, Sahil Bedi, Labeeque Ahmad, Kashvi Tewari

**Sheridan College**

SYST17796-Fundamentals of Software Design and Development

**Prof. Fagun Vankawala**

July 21th, 2024

**Project Scope: War Card Game**

**Game Overview:** War is a two-player card game where players draw and compare cards from their decks.

**How to Play and Rules:**

**Objective**: Win all 52 cards.

**Players**: 2 to 4.

**Setup**: Shuffle and divide the deck evenly, each player gets 26 cards face down.

**Play**: Both players flip their top card simultaneously. The higher card wins the round and the winner takes both cards.

**Card Rank:** Ace (high), King, Queen, Jack, 10-2 (low). Suits are irrelevant.

**War**: If cards are equal, players place three cards face down and flip a fourth. The higher fourth card wins all 10 cards.

**Game End:** When one player has all the cards.

**Tie**: If players have equal cards when the game ends.

**Console output:** Displays current round, cards played, and winner of each round.

Detailed output for war situations and end game summary.

**Technical Scope:**

* Implement the game using provided Java code.
* Apply OOP principles: Encapsulation, Abstraction, Delegation, Aggregation, Composition, Cohesion, and Flexibility.
* Command-line interface with text-based random inputs for two players, round scores, and final winner declaration.

**Design Choices and OO Principles:**

**Encapsulation:** Protects object state and restricts access to variables.

* **CardEx Class:** Encapsulates card properties (suit, rank, value) with getter and setter methods.
* **GameEx Class:** Manages players, rounds, and game flow.
* **PlayerEx Class:** Manages player names and card deck.
* **GroupOfCards Class:** Manages the deck of cards.
* Benefits: Protects data, controls access/modification, reduces errors.

**Cohesion:** Ensures each module has a clear purpose.

* **CardEx Class:** Manages card properties and operations.
* **GameEx Class:** Handles gameplay mechanics.
* Benefits: Improved code readability, easier maintenance, reduced error risk.

**Coupling:** Reduces dependencies between classes.

* **PlayerEx and GameEx Classes:** Relatively loosely coupled, knowing only interfaces, not internal details.
* Benefits: Easier modifications, enhanced flexibility, simplified testing, reduced bug risk.

**Inheritance:** Allows classes to inherit properties and behaviors.

* **Card and CardEx:** CardEx extends the abstract Card class.
* **Player and PlayerEx:** PlayerEx extends the abstract Player class.
* **Game and GameEx:** GameEx extends the abstract Game class, providing specific implementations.
* Benefits: Reduces code duplication, enhances maintainability and flexibility, supports polymorphism.

**Aggregation:** Represents HAS-A relationship, where classes contain collections of other classes.

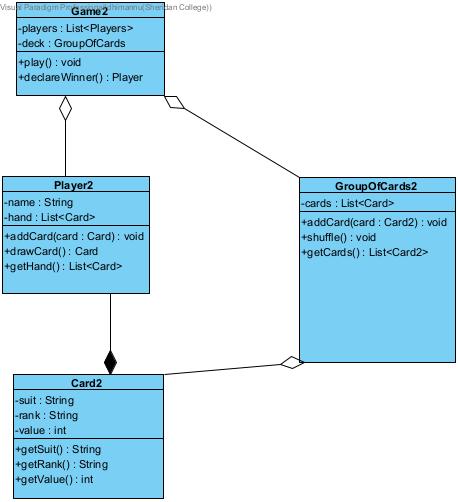
* **GameEx and GroupOfCards:** GameEx contains GroupOfCards.
* **Game and Player:** Game contains a list of Players.
* **PlayerEx and Card:** PlayerEx contains a list of Cards.
* Benefits: Flexible relationships, easier maintenance, reduced coupling.

**Composition:** Stronger form of aggregation with dependent entities.

* **PlayerEx Class:** Contains an ArrayList of Card objects.
* **GroupOfCards Class:** Manages the cards list as part of its state.
* Benefits: Strong ownership, modular and reusable code, easier management of complex objects.

**Flexibility:** Ease of adapting and extending code.

* Uses abstract classes (Card, Game, Player) to define common behaviors.
* Promotes modification of individual components without affecting others.
* Supports polymorphism in Player and Game classes.
* Benefits: Easier maintenance, quick bug fixes, and accommodates changes efficiently.



**Class Diagram:**

* **Game to Player:** Aggregation (Game has a list of Player objects).
* **Game to GroupOfCards:** Aggregation (Game has a GroupOfCards object).
* **Player to Card:** Composition (Player owns a list of Card objects).
* **GroupOfCards to Card:** Aggregation (GroupOfCards has a list of Card objects).

**Explanation:**

* Aggregation indicates that classes can exist independently, promoting flexibility.
* Composition indicates a stronger dependency, ensuring contained objects are integral to the owner's state.