**Deliverable 2**

Jashanpreet Kaur Chahal, Romy Romy, Saurav Bedi and Arjun Singh Dhillon

Sheridan College

SYST17796 Fundamentals of Software Design and Development

Fagun Vankawala

July 21st, 2024

**Scope of the project (War Card Game)**

**Overview of the game:**

The War card game is a two-player game where players compete by drawing cards from their decks and comparing their values.

* **How to play the game and rules?**
* The goal is to be the first player to win all the 52 cards of deck. It is generally played between 2 players, but up to 4 people can play.
* After shuffling the deck, it is divided evenly among players, each player receiving 26 cards. Each player places their stack of cards face down, in front of them.
* Each player flips the top card of the stack at the same time.
* The player with the higher card wins the round and collects both cards to add to their hand. The rank of cards from high to low is - Ace (highest), King, Queen, Jack, 10, 9, 8, 7, 6, 5, 4, 3, 2 (lowest). (Only the value of the card matters, not the suit).
* If the flipped cards are same, then, it is a ‘War’. Each player must place three more cards face-down on the table and flip over a fourth card. Whoever has the higher fourth card is the player who takes all 10 cards from the round. If a player doesn't have enough cards to play the war, the player must turn his/her last card face-up.
* The game ends when one player won all the cards.
* If both players have an equal number of cards when the game ends, the game is declared a tie.

 The game is having console output to display the current round, the cards played by each player, and the winner of each round.

 Detailed output for "war" situations to show the additional cards drawn and the resolution of the tie.

 A summary at the end of the game declaring the overall winner or if the game is a tie.

**Technical Scope of the project:**

The project's technical scope includes creating a card game with the Java basic code that has been provided. Object-Oriented Programming (OOP) principles such as Encapsulation, Abstraction, Delegation, Aggregation, Composition, Cohesion and Flexibility will be applied in this project. The interface of this project is Command-line having text-based random inputs for two players playing the game and giving score at each round and at last, declaring the winner.

**Design Choices and OO Principles:**

1. **Encapsulation:** Encapsulation is a technique used to prevent users from having direct access to specific parts of an object, preventing them from knowing the state values of all the variables in that object.

* **CardEx Class:** This class encapsulates the properties of the cards, i.e. suit, rank and value. It has getter and setter methods to access and modify these properties.
* **GameEx Class:** This class encapsulates the implementation of how to play the game by managing players, rounds and flow of the game.
* **PlayerEx Class:** It encapsulates the player names and card deck. It has methods for adding, drawing and accessing the cards.
* **GroupOfCards Class:** It encapsulates the deck of cards by providing methods to add, access and shuffle the cards.

So, the code is hiding the internal details of classes and providing only controlled access to its properties.

**Why to use it:**

* It helped to protect data from unauthorized access and modification.
* It allowed us to control how the data is accessed and modified.
* Without encapsulation, there would be no control over how the data is accessed or modified, increasing the risk of introducing errors.

1. **Cohesion**: Cohesion is the degree to which the elements inside a module belong together.

High cohesion means that the component does only one thing well and has a clear purpose and scope.

* **CardEx class**: This class is highly cohesive because it is only responsible for representing a card and its related operations (e.g., getting and setting its suit, rank, and value, and providing a string representation).
* **GameEx class**: This class focuses on the gameplay mechanics, such as initializing the deck, playing the rounds, and declaring the winner which is also representing high cohesion.

**Why to use it:**

* High cohesion improved understandability and readability of the code.
* High cohesion reduced the likelihood that a change in one part of code will affect other parts.
* Low cohesion leads to complex code with tightly coupled responsibilities, making it harder to debug and modify.

1. **Coupling:** Coupling is basically dependency of one class on another. Loose coupling refers to a scenario where two classes, modules, or components have minimal dependencies on each other. It signifies that these classes are independent, with one class knowing only what the other exposes through its interfaces.

* **PlayerEx and GameEx classes:** These classes are relatively loosely coupled. For example, PlayerEx depends on the Card class but does not need to know the specific details of how a card is implemented. Similarly, GameEx uses PlayerEx and GroupOfCards without needing to know their internal workings.

**Why to use it:**

* It is simpler to alter individual components without having an impact on the system when there is low coupling between modules.
* Loose coupling also enhanced flexibility and testing of the code.
* Changes in one class would necessitate changes in other classes, increasing the risk of introducing bugs, if there is High coupling between the classes.

1. **Inheritance**: Inheritance represents IS-A relationship. Inheritance allows classes to inherit the properties and behaviors of another class. An inherited class is called a subclass or child class of the class it inherits from. And the class being inherited is called either a parent class, superclass, or base class.

* **Card and CardEx:** CardEx extends the abstract Card class, which allows it to provide specific implementations of the card properties and methods.
* **Player and PlayerEx:** PlayerEx extends the abstract Player class, inheriting its properties and methods and allowing for specific implementations related to the game logic.
* **Game and GameEx:** Game is an abstract base class that defines the general structure and behavior of a game. It includes abstract methods play() and declareWinner() which must be implemented by any subclass. GameEx is a subclass that extends Game and provides specific implementations for the play() and declareWinner() methods. This class adds additional properties and methods specific to the "War" card game, such as handling the deck of cards and managing the game rounds.

**Why to use it:**

* Inheritance saved code duplication by letting derived classes reuse properties and methods defined in the parent class.
* It helped to make the code easy to maintain and flexible.
* It supported polymorphic behavior, which allowed objects from various classes to be regarded as if they were from the same superclass.
* Without inheritance, common functionality would need to be implemented in multiple classes, resulting in code redundancy.

1. **Aggregation:** Aggregation represents HAS-A relationship. Aggregation is a form of association where one class contains a collection of another class. It is a one-way relationship where both entries can survive individually which means ending one will not affect the other.

* **GameEx and GroupOfCards:** In the GameEx class, there is a reference to a GroupOfCards object, representing the deck of cards. The GroupOfCards instance can exist independently of the GameEx instance.
* **Game and Player:** In the Game class, there is a reference to a list of Player objects. The Player instances can exist independently of the Game instance.
* **Player and Card:** In the Player class, there is a reference to a list of Card objects. The Card instances can exist independently of the Player instance.

**Why to use it:**

* It enabled a more adaptable relationship between the classes as it was less complex and easy to analyze.
* Aggregation reduces coupling between classes, making the system easier to maintain and modify.
* Since the second class can exist independently of the first class, aggregation provided flexibility in the design of the system.
* Without Aggregation, the relationships between objects would be inadequately specified, resulting in a lack of understanding about how various areas of the code interact.

1. **Composition:** Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.  When there is a composition between two entities, the composed object cannot exist without the other entity.

* **PlayerEx class**: It contains an ArrayList of Card objects representing the player's deck. This is composition because the cards list is part of the player's state and should be destroyed if the player is destroyed.
* **GroupOfCards class:** The cards are created and managed by GroupOfCards. The cards list is a part of the GroupOfCards class, and it doesn't make sense for the cards list to exist without the GroupOfCards object.

**Why to use it:**

* Itensured strong ownership and control over the contained object.
* It helped to build more modular and reusable code that can adapt to different requirements without the need for extensive modification.
* It provided code-reusability also.
* Without Composition, it would be more difficult to manage complex objects and their interactions.

1. **Flexibility**: It is the ease with which a program can be changed. It is easy to adapt to the changes and extend functionality of the code.

* The code uses abstract classes: Card, Game and Player, defining the common behavior that can be used by other classes extending them. New games can be developed with minimal changes to existing code.
* Each class is responsible for different aspects of the game, which is promoting flexibility as we need to modify individual components for any changes without affecting others.
* Polymorphism used in Player and Game classes allows to add new players and games without affecting existing code.

**Why to use it:**

* It helped to maintain the code and fix the bugs easily.
* It can accommodate changes or features very easily without rewriting much of our code and save the time.
* Without flexibility, the code would be hard and difficult to change, making it impossible to meet changing requirements or addition of new features.

Class Diagram:

A diagram of a computer

Description automatically generated

This diagram shows Associations (Aggregation and Composition) between classes.

* **Player to Card:** Aggregation, with a multiplicity of one to many (Player can have many Card objects and the Player class has a collection of Card objects, but the Card objects can exist independently of the Player).
* **Game to Player:** Aggregation, with a multiplicity of one to one (Game class is responsible for holding and managing a collection of Player objects, but the Player objects can exist independently of the Game).
* **PlayerEx to Card:** Aggregation, with a multiplicity of one to many (One PlayerEx can have many card objects and the PlayerEx class manages a collection of Card objects, and these Card objects can exist independently of the PlayerEx instance).
* **GroupOfCards to Card:** Composition with a multiplicity of 0 to many (GroupOfCards can have zero or more cards. Card instances is tied to the GroupOfCards instance. If a GroupOfCards instance is destroyed, all the Card instances it contains are also destroyed).
* **GroupOfCards to CardEx:** Composition with a multiplicity of 1 to many (One GroupOfCards can have many CardEx objects. GroupOfCards has a strong ownership relationship with CardEx. If a GroupOfCards object is destroyed, its CardEx objects are also destroyed).
* **GameEx to GroupOfCards:** Aggregation, with a multiplicity of one to one (GameEx "has-a" GroupOfCards, but they are not tightly coupled. The GroupOfCards instance can exist independently of the GameEx instance).