МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
„КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. Сікорського”

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

КУРСОВА РОБОТА

з дисципліни «Програмування»
на тему
Шаблони проектування в ООП. Консольний додаток

Виконала студентка

ІІ курсу групи КП-11
Кирильчук Олександра Артурівна

Керівник роботи

доцент, к.т.н. Заболотня Т.М.

Оцінка

(дата, підпис)

КИЇВ 2023

Зміст

ВСТУП

Об'єктно-орієнтоване програмування (ООП) є однією з найпоширеніших парадигм програмування, що дозволяє розробляти складні програмні системи за допомогою об'єктів, які взаємодіють один з одним. ООП надає багато інструментів і прийомів для ефективної організації коду, забезпечуючи легку розширюваність, підтримку інкапсуляції і перевикористання коду.

Одним з найважливіших аспектів ООП є використання шаблонів проектування. Шаблони проектування є загальними рішеннями для типових проблем, що виникають під час розробки програмного забезпечення. Вони надають стандартизований підхід до розробки, спрощуючи процес проектування та підвищуючи якість коду. Використання шаблонів проектування сприяє створенню гнучких, розширюваних і легко зрозумілих систем.

Мета – дослідження і практичне використання шаблонів проектування в контексті об'єктно-орієнтованого програмування. Основною метою є вивчення різних типів шаблонів проектування і їх застосування для розробки консольної програми, яка реалізує гру в карти. Метою цієї роботи є розробка програмного забезпечення, яке дозволяє користувачам грати в різноманітні ігри в карти на консольному інтерфейсі. Програма повинна надавати можливість створення гри з заданими правилами, роздавати карти гравцям, виконувати правила гри та визначати переможця.

Об'єкт – процес гри в карти. Ігри в карти є популярними і розважальними заняттями, які вимагають стратегічного мислення, вміння приймати рішення та співпрацювати з іншими гравцями.

Для досягнення цієї мети робота включає такі завдання:

1. Аналіз різних шаблонів проектування і їхніх особливостей.
2. Розробка архітектури програми: створення класів для представлення карт, колоди, гравців та гри.

3. Реалізація логіки гри: визначення правил гри, роздача карт, виконання ходів гравців та визначення переможця.
4. Реалізація консольного інтерфейсу: створення меню для створення гри, введення правил та взаємодії з гравцями.
5. Тестування та оцінка ефективності використання шаблонів проектування в розробці банкомату

В роботі буде використано мову програмування C#, оскільки вона має зручний синтаксис, широкі можливості для обробки об'єктів та велику кількість наявних бібліотек.

Ця програма може бути корисною для людей, які цікавляться настільними іграми та хочуть грати в гру в карти на комп'ютері. Вона надає зручний спосіб для гри в карти без потреби мати фізичну колоду та інших матеріалів.

У подальшому розвитку цієї програми можна розглядати можливість додаткових функцій, таких як розширені правила гри, підтримка різних типів ігор в карти, гра з комп'ютером та мережева гра з іншими користувачами.
Реалізовані шаблони програмування: Фабричний метод, Спостерігач, Команда, Шаблонний метод, Фасад, Стратегія, Міст, Декоратор.

До функціональних можливостей входять: створення гравців, колоди, гри; можливість ходити, відбивати карти та підкидати їх.

В подальшій частині роботи будуть описані деталі архітектури програми, проектування та реалізація логіки гри, а також деталі консольного інтерфейсу та можливості програми.

Для функціонування розробленої програми необхідно забезпечити наявність на комп’ютері 2 Мб вільного дискового простору.

1. ОПИС СТРУКТУРНО-ЛОГІЧНОЇ СХЕМИ ПРОГРАМИ

1.1 Опис структурно-логічної схеми програми

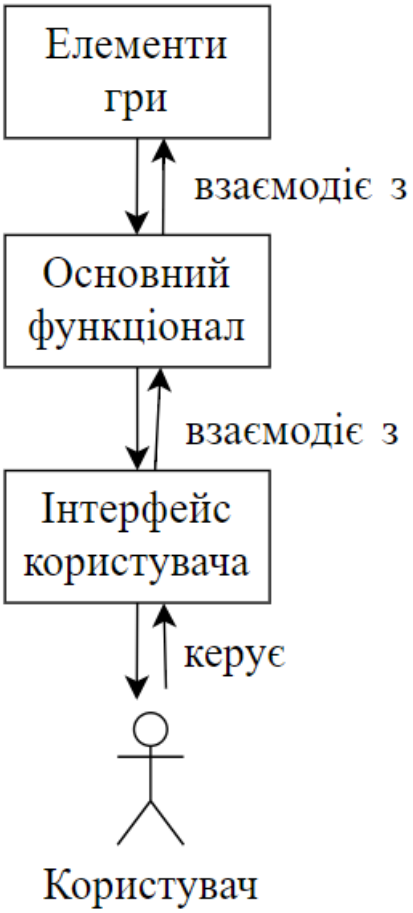

Рис.1.1.1-Приклад структурної схеми програмного забезпечення

У цій структурно-логічній схемі користувач взаємодіє з програмою через інтерфейс користувача. Інтерфейс користувача передає команди та інструкції основному функціоналу програми. Основний функціонал виконує потрібні дії, такі як початок гри, перемішування колоди, роздача карт гравцям тощо. Основний функціонал також взаємодіє з класом Game, який управляє самою грою. Клас Game в свою чергу викликає методи класу Player, який представляє окремого гравця в грі. Крім того, клас Player може використовувати клас SpecialCard, якщо в грі використовуються спеціальні карти.

Ця структурно-логічна схема демонструє загальний потік взаємодії між різними компонентами програми "CardGame". Зверніть увагу, що це лише загальна схема і може бути більше деталей та компонентів, які не відображені у цій схемі.

Опишемо кожен логічний елемент зазначеної структурно-логічної схеми програми "CardGame2":

1. Користувач: Це особа, яка взаємодіє з програмою. Користувач вводить команди та надає інструкції через інтерфейс користувача.
2. Інтерфейс користувача: Це компонент, який дозволяє користувачеві взаємодіяти з програмою. Він може бути реалізований у вигляді графічного інтерфейсу (GUI) або командного рядка (CLI). Інтерфейс користувача отримує вхідні дані від користувача та передає їх основному функціоналу програми.
3. Основний функціонал: Це центральна частина програми, яка виконує основні дії та операції, пов'язані з грою. Основний функціонал може включати функції, такі як початок гри, перемішування колоди, роздача карт гравцям, обробка ходів гравців, визначення переможця тощо. Він отримує команди та інструкції від інтерфейсу користувача та взаємодіє з іншими класами для виконання необхідних операцій.
4. Елементи гри:
    1) Game: Це клас, який управляє самою грою. Він отримує команди від основного функціоналу та виконує необхідні дії для програвання гри, такі як встановлення правил гри, контроль ходів гравців, розрахунок рахунку тощо. Клас Game може взаємодіяти з класом Player для обробки дій кожного гравця в грі.
    2) Player: Це клас, який представляє окремого гравця в грі. Він отримує команди та інструкції від класу Game та виконує власні ходи та дії відповідно до правил гри. Наприклад, клас Player може визначати, яку карту грати або як взаємодіяти з іншими гравцями.

Ці логічні елементи взаємодіють між собою, утворюючи основу програми "CardGame" і забезпечуючи користувачу можливість грати в карткову гру та взаємодіяти з іншими гравцями чи спеціальними картами у відповідності до правил гри.

1.2 Функціональні характеристики

Основні функціональні характеристики програми включають:

1. Factory Method: Програма має реалізацію шаблону проектування "Factory Method". У ній визначені два типи карток - NumberCard і SpecialCard, а також дві фабрики - NumberCardFactory і SpecialCardFactory, які створюють відповідні об'єкти карток.
2. Observer: Програма має реалізацію шаблону проектування "Observer". Інтерфейси IObserver і IObservable визначають спостерігачів та спостережуваний об'єкт відповідно. Клас Game є спостережуваним об'єктом, а клас Player реалізує інтерфейс IObserver. Коли гра закінчується, всі спостерігачі (гравці) повідомляються про це.
3. Command: Програма має реалізацію шаблону проектування "Command". Інтерфейс ICommand визначає методи Execute() і Undo(), а класи StartGameCommand, ShuffleDeckCommand і DealCardsCommand реалізують цей інтерфейс. Команди виконуються для початку гри, перемішування колоди карток і роздачі карток гравцям. Існує також можливість відміни виконаної команди.
4. Класи Card, NumberCard і SpecialCard: Вони визначають основні властивості та методи карток. Клас NumberCard представляє картки з числовими значеннями, а клас SpecialCard - картки з особливими властивостями. Обидва ці класи успадковуються від абстрактного класу Card.
5. Класи CardFactory, NumberCardFactory і SpecialCardFactory: Вони визначають фабрики для створення відповідних карток.

NumberCardFactory створює об'єкти типу NumberCard, а SpecialCardFactory створює об'єкти типу SpecialCard.

6. Клас Deck: Представляє колоду карток. Він містить список карток і методи для перемішування колоди та роздачі карток гравцям.
7. Клас Game: Це спостережуваний об'єкт, який керує грою. Він містить колоду карток, список гравців і методи для початку гри, роздачі карток, виконання ходу та визначення переможця. Клас Game використовує команди (Command) для виконання дій.
8. Клас Player: Представляє гравця гри. Він має список карток, методи для отримання та відтворення карток, а також методи для гри.

Ця програма демонструє використання декількох шаблонів проектування для покращення структури і розширення функціональності. Вона дозволяє створювати карткові ігри з різними типами карток та взаємодіяти з гравцями.

1.3 Опис реалізованих класів

`Card`

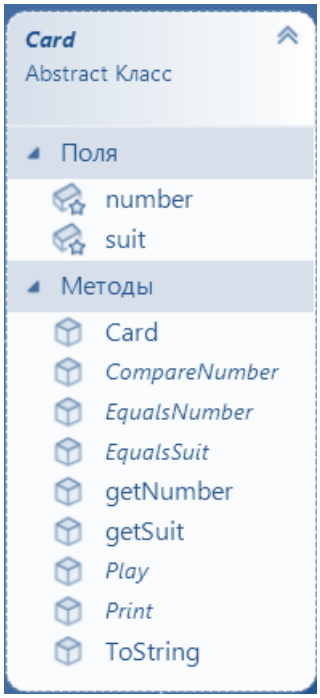

Рис.1.3.1 Діаграма класу `Card`

Клас `Card` є абстрактним базовим класом для представлення карт у грі. У ньому оголошено конструктор, який приймає номер карти (перераховування `CardNumber`) та масть карти (перераховування `CardSuit`). Клас також містить методи `Play()`, `Print()`, `CompareNumber(Card card)`, `EqualsSuit(Card card)`, `EqualsNumber(Card card)`, а також властивості `number` та `suit`.

Класи **CardNumber** і **CardSuit** представляють перерахування (enum) для представлення значень номерів карт і мастей в гральній колоді. Ці перерахування використовуються для ідентифікації і представлення номерів і мастей карт у гральній колоді. Вони дозволяють легко визначити значення карт у грі і виконувати різні операції з ними, такі як порівняння, сортування тощо.

`NumberCard`

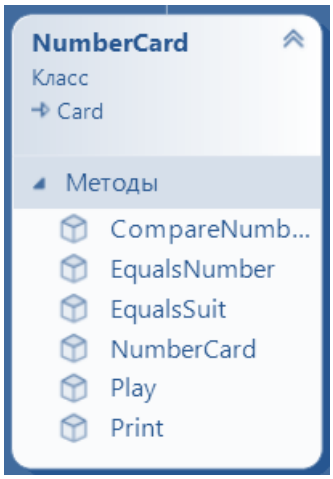

Рис.1.3.2 Діаграма класу `NumberCard`

Клас `NumberCard` успадковує клас `Card` і представляє карту з числовим значенням. У ньому визначені методи `CompareNumber(Card card)`, `EqualsNumber(Card card)`, `EqualsSuit(Card card)`, `Play()`, `Print()`, а також конструктор, який приймає номер карти та масть карти.

`SpecialCard`

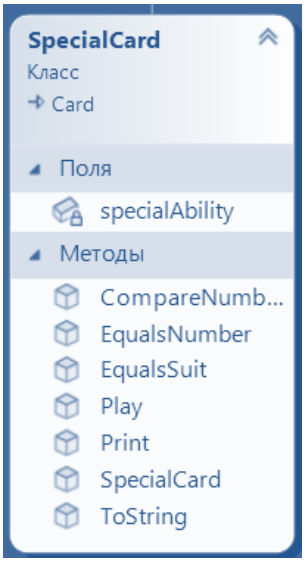

Рис.1.3.3 Діаграма класу `SpecialCard`

Клас `SpecialCard` також успадковує клас `Card` і представляє спеціальну карту з додатковою властивістю `specialAbility`. У ньому визначені методи `CompareNumber(Card card)`, `EqualsNumber(Card card)`, `EqualsSuit(Card card)`, `Play()`, `Print()`, а також конструктор, який приймає назву спеціальної властивості карти, номер карти та масть карти.

`CardFactory`

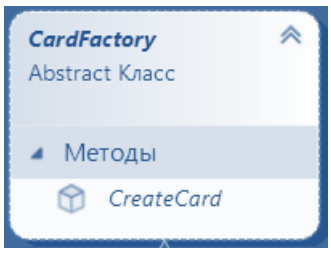

Рис.1.3.4 Діаграма класу `CardFactory`

`SpecialCardFactory`

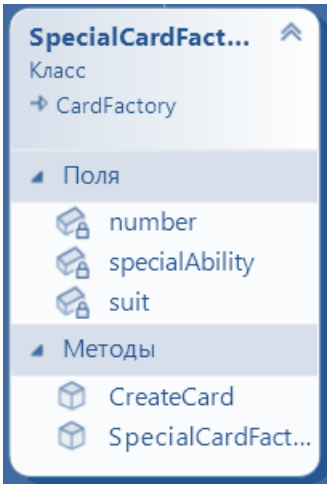

Рис.1.3.5 Діаграма класу

`NumberCardFactory`

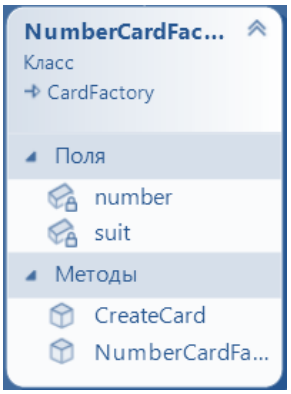

Рис.1.3.5 Діаграма класу `NumberCardFactory`

Клас `CardFactory` є абстрактним базовим класом для фабрик карт. В ньому визначений абстрактний метод `CreateCard()`, який повертає об'єкт типу `Card`. Класи `NumberCardFactory` та `SpecialCardFactory` успадковують клас `CardFactory` і реалізують метод `CreateCard()` для створення відповідно числових карт і спеціальних карт.

`CardDecorator`

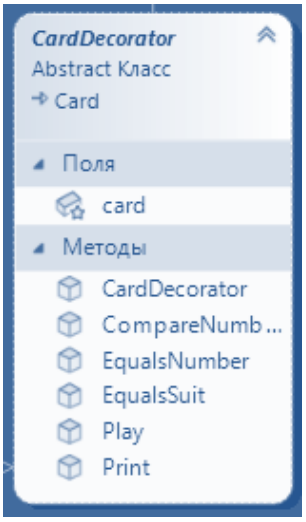

Рис.1.3.6 Діаграма класу `CardDecorator`

Клас `CardDecorator` є абстрактним класом, який розширює клас `Card` і використовується для декорування (обгортання) об'єктів типу `Card`. Він має такі методи:

`CardDecorator(Card card)`: Конструктор класу, приймає об'єкт типу `Card` як параметр і передає його базовому конструктору класу `Card`. Це дозволяє зберегти посилання на оригінальний об'єкт `card`.

`Play()`: Перевизначений метод з класу `Card`, який викликає метод `Play()` оригінального об'єкта `card`. Таким чином, коли викликається `Play()` на об'єкті `CardDecorator`, викликається метод `Play()` оригінального об'єкта `card`.

`Print()`: Перевизначений метод з класу `Card`, який викликає метод `Print()` оригінального об'єкта `card`.

`CompareNumber(Card card)`: Перевизначений метод з класу `Card`, який викликає метод `CompareNumber(Card card)` оригінального об'єкта `card`.

`EqualsNumber(Card card)`: Перевизначений метод з класу `Card`, який викликає метод `EqualsNumber(Card card)` оригінального об'єкта `card`.

`EqualsSuit(Card card)`: Перевизначений метод з класу `Card`, який викликає метод `EqualsSuit(Card card)` оригінального об'єкта `card`.

`SpecialAbilityCardDecorator`

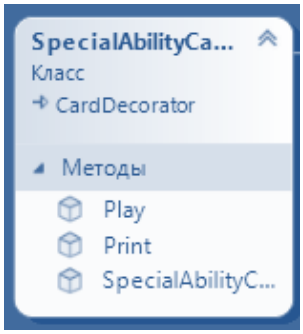

Рис.1.3.7 Діаграма класу `SpecialAbilityCardDecorator`

Клас `SpecialAbilityCardDecorator` є підкласом класу `CardDecorator` і використовується для декорування спеціальних карт з особливими властивостями. Він має такі методи:

`SpecialAbilityCardDecorator(Card card)`: Конструктор класу, приймає об'єкт типу `Card` як параметр і передає його базовому конструктору класу `CardDecorator`.

`Play()`: Перевизначений метод з класу `CardDecorator`, який викликає метод `Play()` оригінального об'єкта `card`, але перед цим встановлює колір фону консолі на синій, колір шрифту на білий, а потім скидає налаштування кольору. Таким чином, коли викликається `Play()` на об'єкті `SpecialAbilityCardDecorator`, спочатку змінюється колір виводу в консоль, потім викликається метод `Play()` оригінального об'єкта `card`, і потім кольори скидаються.

`Print()`: Перевизначений метод з класу `CardDecorator`, який викликає метод `Print()` оригінального об'єкта `card`, але перед цим встановлює колір фону консолі на синій, колір шрифту на білий, а потім скидає налаштування кольору

`Game`

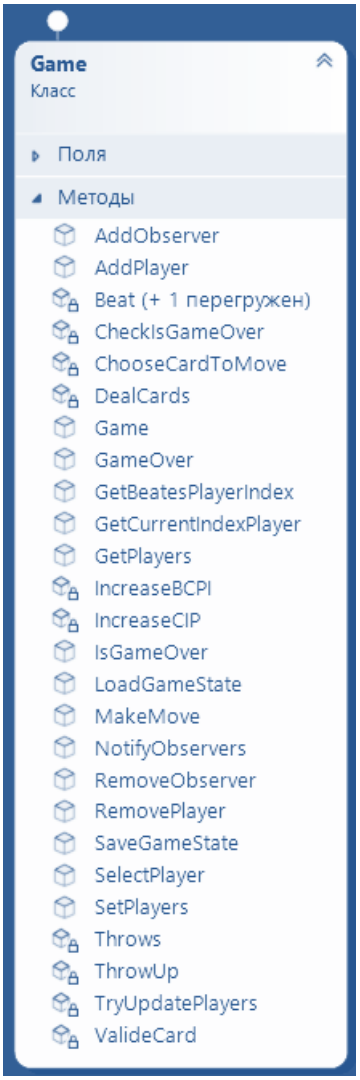

Рис.1.3.8 Діаграма класу `Game`

Клас `Game` представляє гру і реалізує інтерфейс `IObservable`, що дозволяє спостерігати за змінами стану гри. У ньому визначені поля для зберігання спостерігачів, гравців, ознаки завершення гри, індексів поточного гравця, гравця, що виставив карту, гравця, чию карту побили. В класі реалізовані методи для додавання, видалення та сповіщення спостерігачів, додавання та видалення гравців, завершення гри, перевірки, чи гра завершена, отримання списку гравців тощо.

Опис кожного методу класу `Game`:

`AddObserver(IObserver observer)`: Додає спостерігача до списку спостерігачів гри. Метод приймає об'єкт, який реалізує інтерфейс `IObserver` і додає його до списку `observers`.

`RemoveObserver(IObserver observer)`: Видаляє спостерігача зі списку спостерігачів гри. Метод приймає об'єкт, який реалізує інтерфейс `IObserver` і видаляє його зі списку `observers`.

`NotifyObservers()`: Повідомляє всіх спостерігачів про зміну стану гри. Метод перебирає всіх спостерігачів у списку `observers` і викликає для кожного з них метод `Update()`.

`AddPlayer(Player player)`: Додає гравця до списку гравців гри. Метод приймає об'єкт `Player` і додає його до списку `players`.

`RemovePlayer(Player player)`: Видаляє гравця зі списку гравців гри. Метод приймає об'єкт `Player` і видаляє його зі списку `players`.

`GameOver()`: Викликається, коли гра завершилась. Встановлює значення `gameOver` на `true` і повідомляє всіх спостерігачів про зміну стану гри за допомогою методу `NotifyObservers()`.

`IsGameOver()`: Перевіряє, чи гра завершилась. Повертає значення змінної `gameOver`, яка вказує на стан гри (завершена або ні).

`GetPlayers()`: Повертає список гравців гри. Повертає значення списку `players`.

`SetPlayers(List<Player> players)`: Встановлює список гравців гри. Метод приймає новий список гравців і присвоює його значення змінній `players`.

`GetCurrentIndexPlayer()`: Повертає поточний індекс гравця. Повертає значення змінної `currentPlayerIndex`, яка вказує на індекс поточного гравця в списку `players`.

`SelectPlayer()`: Визначає гравця, який має право першим кинути карту. Метод перебирає всіх гравців у списку `players` і їх картки, шукає найменшу карту серед спеціальних карт або карт з особливими властивостями і повертає гравця, який має цю карту.

`IncreaseBCPI(int i)`: Збільшує значення змінної `beatesCardPlayerIndex` на 1. Якщо `beatesCardPlayerIndex` стає більшим або рівним довжині списку `players`, то встановлює його значення на 0.

`IncreaseCIP()`: Збільшує значення змінної `currentPlayerIndex` на 1. Якщо `currentPlayerIndex` стає більшим або рівним довжині списку `players`, то встановлює його значення на 0. Якщо `beatesCardPlayerIndex` дорівнює 0, то `currentPlayerIndex` встановлюється на 1.

`Beat(Player player, Card move)`: Метод для биття карт. Гравець (`player`) обирає карту (`move`), якою він б'є. Метод викликається рекурсивно, доки гравець не обере карту, яка б'є попередню карту. Якщо гравець не може побити карту, то він забирає всі використані картки і картки, які мав бити, відновлюється його рука.

`Beat(Player player, Card move, Card beat)`: Метод, що перевіряє чи вибрана карта (`beat`) може побити попередню карту (`move`). Якщо так, то картка додається до списку використаних карт гравця (`player`), і вона видаляється з руки гравця та списку карт, які потрібно побити.

`ChooseCardToMove(PlayerCardSelectionMechanism playerCardSelectionMechanism)`: Метод, який дозволяє гравцеві вибрати карту для ходу. Метод викликає метод `Play()` об'єкту `playerCardSelectionMechanism` і повертає обрану карту.

`ThrowUp(Player player)`: Метод, який дозволяє гравцеві відкинути карту. Гравець (`player`) обирає карту для відкидання зі своєї руки. Якщо карта є валідною (тобто можна відкинути карту, яку потрібно побити), вона видаляється з руки гравця і повертається. Якщо карта не є валідною або гравець не хоче відкидати карту, повертається `null`.

`ValideCard(Card card, List<Card> cards)`: Метод, який перевіряє, чи карта (`card`) є валідною для биття серед списку карт (`cards`). Метод перебирає всі карти в списку `cards` і порівнює їх з картою `card` за допомогою методів

порівняння `EqualsNumber()` та `EqualsSuit()`. Якщо хоча б одна карта валідна, метод повертає `true`, інакше повертає `false`.

`Throws()`: Метод, який виконує відкидання карт гравцями. Для кожного гравця в списку `players`, крім гравця, який має картки для побиття (`players[beatesCardPlayerIndex]`), виконується метод `ThrowUp()`. Якщо гравець відкидає карту, викликається метод `Beat()` для побиття карти.

`DealCards(Deck deck)`: Роздача карт гравцям. Метод проходить по кожному гравцеві в списку `players` і додає карту з колоди (`deck`) до руки гравця, доки у нього не буде 6 карт або колода не закінчиться.

`TryUpdatePlayers()`: Перевіряє, чи у гравців закінчились карти. Якщо у гравця відсутні картки, він видаляється зі списку `players` і викликається метод `Update()` для нього.

`CheckIsGameOver()`: Перевіряє, чи гра завершилась. Повертає `true`, якщо кількість гравців у списку `players` менше або рівна 1, інакше повертає `false`.

`SaveGameState ()`: Зберігає поточний стан гри.

`LoadGameState()`: Завантажує збережений раніше стан гри.

`Player`

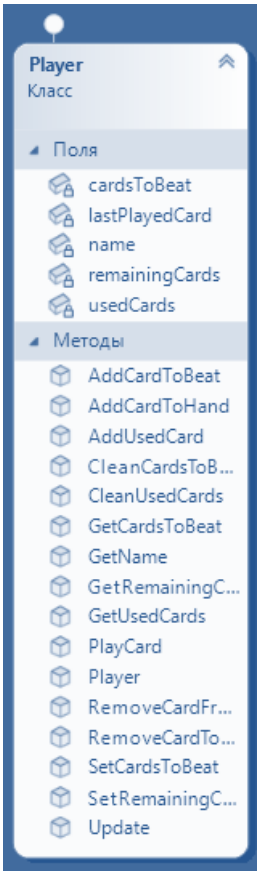

Рис.1.3.9 Діаграма класу `Player`

Клас **Player** представляє гравця в грі. Він реалізує інтерфейс **IObserver** для сприймання оновлень від гри. Клас містить інформацію про гравця, таку як його ім'я, список залишених карт, список карт, що можуть бути перебиті, та список використаних карт. Він також має методи для здійснення ходу гравця та взаємодії з картами.`Update(Game game)`: Метод отримує оновлення від гри та виконує відповідні дії гравця.

`Update()`: Метод, який викликається при сповіщенні гравця про його вибування з гри. Виводить повідомлення про те, що гравець з ім'ям `name` був елімінований з гри.

`PlayCard(Card card)`: Метод, який дозволяє гравцеві зіграти карту `card`. Встановлює `card` як останню зіграну карту (`lastPlayedCard`), виводить повідомлення про те, що гравець з ім'ям `name` зіграв карту з номером та мастю, і видаляє цю карту зі списку `remainingCards` (карти в руці гравця).

`GetName()`: Метод, який повертає ім'я гравця (`name`).

`AddUsedCard(Card card)`: Додає карту `card` до списку використаних карт (`usedCards`) гравця.

`GetUsedCards()`: Повертає список використаних карт (`usedCards`) гравця.

`CleanUsedCards()`: Очищує список використаних карт (`usedCards`) гравця.

`AddCardToHand(Card card)`: Додає карту `card` до списку карт в руці (`remainingCards`) гравця.

`RemoveCardFromHand(Card card)`: Видаляє карту `card` зі списку карт в руці (`remainingCards`) гравця.

`SetRemainingCards(List<Card> remainingCards)`: Встановлює список карт в руці (`remainingCards`) гравця на заданий список `remainingCards`.

`GetRemainingCards()`: Повертає список карт в руці (`remainingCards`) гравця.

`AddCardToBeat(Card card)`: Додає карту `card` до списку карт для побиття (`cardsToBeat`) гравця.

`RemoveCardToBeat(Card card)`: Видаляє карту `card` зі списку карт для побиття (`cardsToBeat`) гравця.

`SetCardsToBeat(List<Card> cardsToBeat)`: Встановлює список карт для побиття (`cardsToBeat`) гравця на заданий список `cardsToBeat`.

`CleanCardsToBeat()`: Очищує список карт для побиття (`cardsToBeat`) гравця.

`GetCardsToBeat()`: Повертає список карт для побиття (`cardsToBeat`) гравця.

`Deck`

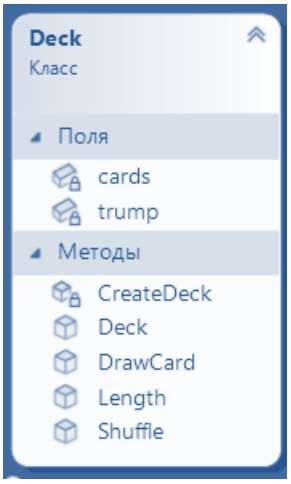

Рис.1.3.10 Діаграма класу `Deck`

Опис методів класу `Deck`:

`Deck()`: Конструктор класу. Він ініціалізує об'єкт типу `Random` для випадкового вибору козирної масті, отримує всі значення мастей в енумі `CardSuit`, вибирає випадкову козирну масті і зберігає її в полі `trump`. Далі він викликає метод `CreateDeck()` для створення колоди карт.

`Length()`: Повертає кількість карт у колоді.

`CreateDeck()`: Приватний метод, який створює колоду карт. Він створює порожній список `deck`, а потім додає до нього спеціальні карти з використанням фабричного методу `SpecialCardFactory` для кожного значення `CardNumber`. Потім він створює список `cardSuits`, який містить всі масті, крім козирної масті, і додає до колоди всі інші карти з використанням фабричного методу `NumberCardFactory`. На виході метод повертає створену колоду.

`Shuffle()`: Метод перемішування колоди. Він виводить повідомлення про перемішування в консоль, створює об'єкт типу `Random`, і для кожної карти у колоді виконує наступне: вибирає випадкову позицію `j` в межах від 0 до `i` (де `i` - індекс поточної карти), обмінює поточну карту з картою, що знаходиться на позиції `j`. Це забезпечує перемішування карт у колоді.

`DrawCard()`: Метод витягування карти з колоди. Якщо кількість карт у колоді більше 0, він видаляє першу карту зі списку `cards` і повертає її. Якщо кількість карт дорівнює 0, метод повертає `null`.

Цей клас відповідає за створення, перемішування та витягування карт з колоди.

`GameFacade`

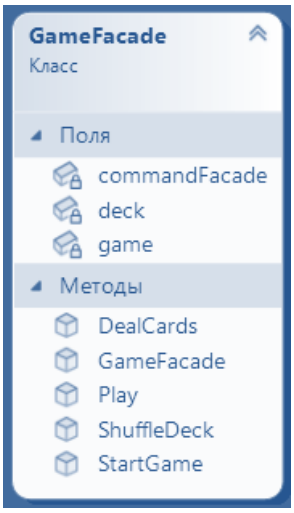

Рис.1.3.11 Діаграма класу `GameFacade`

Опис методів класу `GameFacade`:

`GameFacade()`: Конструктор класу. Він створює об'єкти класів `Game`, `Deck` і `GameCommandFacade`. Клас `Game` відповідає за логіку гри, клас `Deck` - за колоду карт, а `GameCommandFacade` - за набір команд для взаємодії з грою і колодою.

`StartGame()`: Метод для початку гри. Викликає відповідну команду в об'єкті `commandFacade`, яка ініціалізує гру.

`ShuffleDeck()`: Метод для перемішування колоди. Викликає відповідну команду в об'єкті `commandFacade`, яка перемішує колоду карт.

`DealCards()`: Метод для роздачі карт гравцям. Викликає відповідну команду в об'єкті `commandFacade`, яка роздає карти гравцям з колоди.

`Play()`: Метод для початку гри. Викликає відповідну команду в об'єкті `commandFacade`, яка починає гру.

`GameFacade` служить як інтерфейс для зручного використання гри. Він упрощує взаємодію з грою і колодою, надаючи лише обмежений набір методів для початку гри, перемішування колоди, роздачі карт та початку самої гри.

`ICardSelectionStrategy` (інтерфейс):

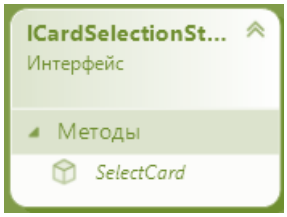

Рис.1.3.12 Діаграма інтерфейсу `ICardSelectionStrategy`

`SelectCard(Player player)`: Метод, який обирає карту для гравця. Приймає об'єкт гравця і повертає обраний ним карту.

`RandomCardSelectionStrategy`

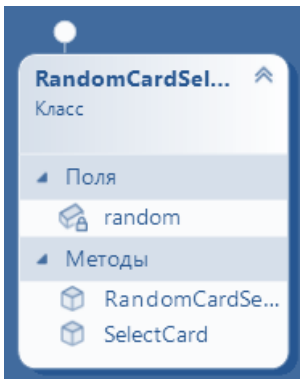

Рис.1.3.13 Діаграма класу **`RandomCardSelectionStrategy`**

`RandomCardSelectionStrategy()`: Конструктор класу. Створює новий об'єкт `Random` для випадкового вибору карт.

`SelectCard(Player player)`: Метод, який вибирає карту для гравця випадковим чином. Бере список доступних карт гравця, генерує випадковий індекс та повертає відповідну карту. Вибрана карта також видаляється з руки гравця.

`LowestCardSelectionStrategy`

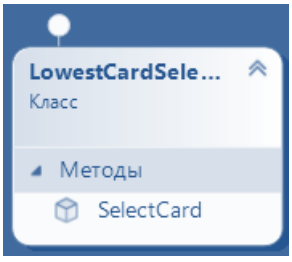

Рис.1.3.14 Діаграма класу `LowestCardSelectionStrategy`

`SelectCard(Player player)`: Метод, який вибирає найменшу карту для гравця. Бере список доступних карт гравця, перевіряє кожну карту і порівнює її номер з найнижчою відомою картою. Повертає найменшу карту і видаляє її з руки гравця.

`ByNumberCardSelectionStrategy`

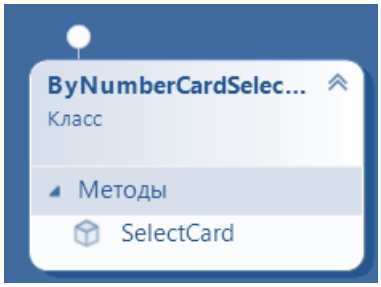

Рис.1.3.15 Діаграма класу `ByNumberCardSelectionStrategy`

`SelectCard(Player player)`: Метод, який дозволяє гравцю вибрати карту з введенням номера. Виводить доступні карти гравця, запитує номер картки від користувача, видаляє вибрану карту з руки гравця і повертає її.

Ці класи реалізують інтерфейс `ICardSelectionStrategy` і надають різні стратегії вибору карт для гравця. `RandomCardSelectionStrategy` обирає випадкову карту, `LowestCardSelectionStrategy` обирає найменшу карту за номером, а `ByNumberCardSelectionStrategy` дозволяє гравцю вибрати карту за номером, введеним користувачем.

`ICommand` (інтерфейс):

Рис.1.3.16 Діаграма інтерфейсу `ICommand`

`Execute()`: Метод, який виконує команду.

`Undo()`: Метод, який відміняє виконану команду.

`StartGameCommand`

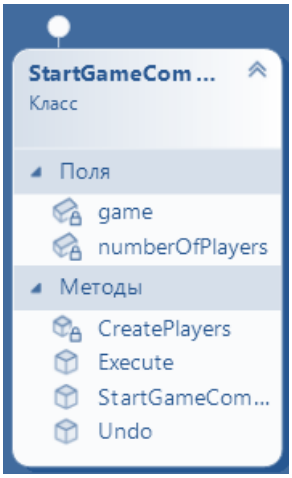

Рис.1.3.17 Діаграма класу `StartGameCommand`

`StartGameCommand(Game game)`: Конструктор класу. Приймає об'єкт гри.

`CreatePlayers()`: Приватний метод, який створює гравців гри на основі введених даних.

`Execute()`: Метод, який розпочинає гру. Запитує кількість гравців, створює їх і додає до гри.

`Undo()`: Метод, який відміняє команду старту гри. Встановлює гравців гри на `null`.

`ShuffleDeckCommand`

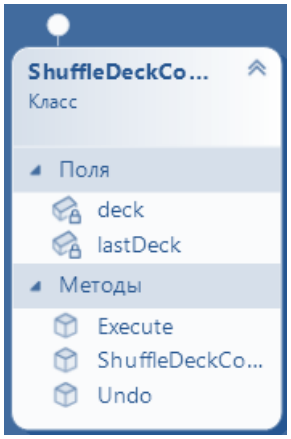

Рис.1.3.17 Діаграма класу `ShuffleDeckCommand`

`ShuffleDeckCommand(Deck deck)`: Конструктор класу. Приймає об'єкт колоди.

`Execute()`: Метод, який перетасовує колоду.

`Undo()`: Метод, який відміняє команду перетасовки колоди. Відновлює попередній стан колоди.

`DealCardsCommand`

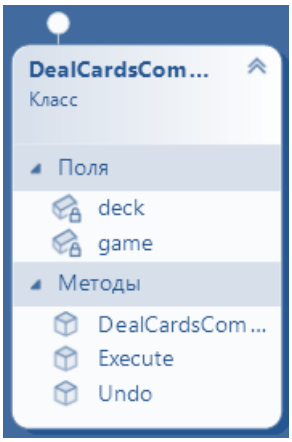

Рис.1.3.18 Діаграма класу `DealCardsCommand`

`DealCardsCommand(Game game, Deck deck)`: Конструктор класу. Приймає об'єкти гри і колоди.

`Execute()`: Метод, який роздає карти гравцям. Для кожного гравця бере шість карт з колоди і додає їх до руки гравця.

`Undo()`: Метод, який відміняє команду роздачі карт. Скидає руки гравців.

`PlayGameCommand`

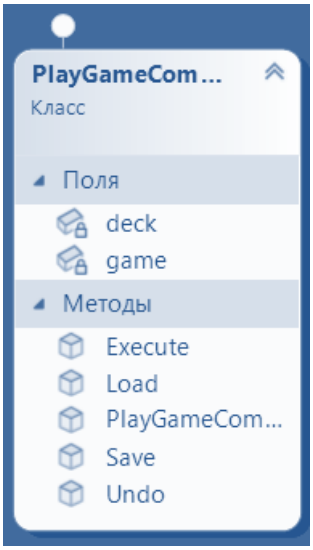

Рис.1.3.19 Діграма класу `PlayGameCommand`

`PlayGameCommand(Game game, Deck deck)`: Конструктор класу. Приймає об'єкти гри і колоди.

`Execute()`: Метод, який розпочинає гру і виконує ходи гравців до завершення гри. Запитує гравця, який може зробити хід, і вибирає стратегію вибору карт для гравця в залежності від вибору користувача. Виконує хід гравця, оновлює поточного гравця і повторює цей процес до завершення гри.

`Undo()`: Метод, який відміняє команду гри. Завершує гру і виводить повідомлення про зупинку гри.

`Save()`: Метод, який викликає зберігання поточного стану гри.

`Load()`: Метод, який завантажує збережений стан гри.

Ці класи реалізують інтерфейс `ICommand` і представляють різні команди, які можуть бути виконані в грі. `StartGameCommand` виконує команду початку гри, `ShuffleDeckCommand` виконує команду перетасовки колоди, `DealCardsCommand` виконує команду роздачі карт гравцям, а `PlayGameCommand` виконує команду гри та ходів гравців. Кожна команда має також метод `Undo()`, який відміняє виконану команду і повертає систему до попереднього стану.

`GameCommandFacade`

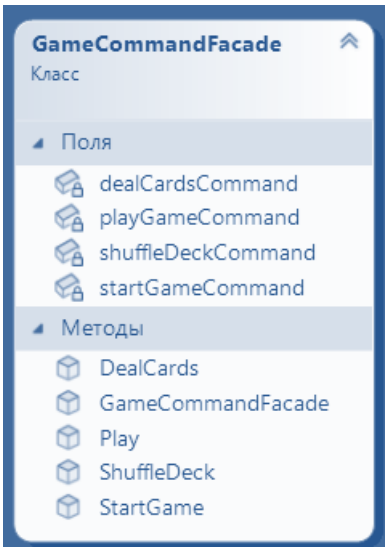

Рис.1.3.20 Діаграма класу `GameCommandFacade`

Клас `GameCommandFacade` є фасадом команд гри і забезпечує простий інтерфейс для виконання різних команд гри. Ось опис методів цього класу:

`GameCommandFacade(Game game, Deck deck)`: Конструктор класу. Приймає об'єкти гри і колоди та створює екземпляри відповідних команд з переданими об'єктами гри і колоди.

`StartGame()`: Метод, який викликає команду початку гри (`startGameCommand.Execute()`). Цей метод розпочинає гру, запитуючи кількість гравців і створюючи їх.

`ShuffleDeck()`: Метод, який викликає команду перетасовки колоди (`shuffleDeckCommand.Execute()`). Цей метод перетасовує колоду карт.

`DealCards()`: Метод, який викликає команду роздачі карт (`dealCardsCommand.Execute()`). Цей метод роздає карти гравцям.

`Play()`: Метод, який викликає команду гри (`playGameCommand.Execute()`). Цей метод починає гру і виконує ходи гравців до завершення гри.

Клас `GameCommandFacade` забезпечує простий спосіб взаємодії з різними командами гри, приховуючи деталі їх реалізації і надаючи зручний інтерфейс для керування грою.

`GameMechanism`

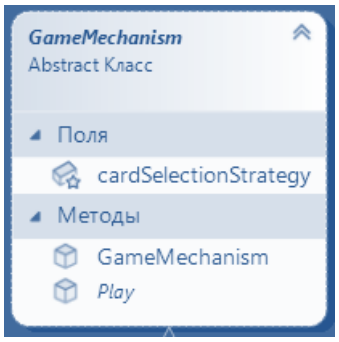

Рис.1.3.21 Діаграма класу `GameMechanism`

Клас `GameMechanism` є абстрактним класом, який визначає механізм гри. Основна його відповідність полягає у виконанні ходів у грі з використанням певної стратегії вибору карт.

`GameMechanism(ICardSelectionStrategy cardSelectionStrategy)`: Конструктор класу. Приймає стратегію вибору карт і зберігає її в полі `cardSelectionStrategy`.

`Play()`: Абстрактний метод, який представляє виконання ходу у грі. Конкретна реалізація цього методу буде залежати від підкласу, який розширює `GameMechanism`.

`PlayerCardSelectionMechanism`

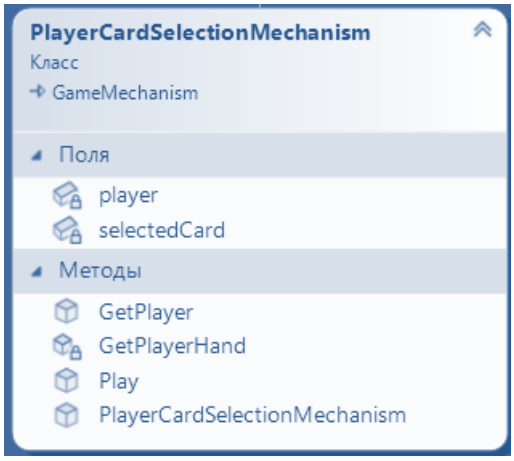

Рис.1.3.22 Діаграма класу `PlayerCardSelectionMechanism`

Клас `PlayerCardSelectionMechanism` є конкретною реалізацією `GameMechanism` для вибору карти гравцем. Він зберігає посилання на гравця, який здійснює хід, і обраний ним карту.

`PlayerCardSelectionMechanism(Player player, ICardSelectionStrategy cardSelectionStrategy)`: Конструктор класу. Приймає об'єкт гравця і стратегію вибору карт. Викликає конструктор батьківського класу `GameMechanism` і передає йому стратегію.

`Play()`: Реалізація методу `Play()`, у якому гравець вибирає карту зі своєї руки за допомогою стратегії `cardSelectionStrategy.SelectCard(player)`. Обрана карта зберігається у полі `selectedCard`, а також виводиться повідомлення про обрану карту.

`GetPlayer()`: Метод, який повертає об'єкт гравця, пов'язаний з цим механізмом гри.

`GetPlayerHand()`: Приватний метод, який повертає руку гравця (список карт), які залишилися у нього.

Клас `PlayerCardSelectionMechanism` дозволяє гравцю здійснювати хід у грі за допомогою конкретної стратегії вибору карт.

`GameTemplate`

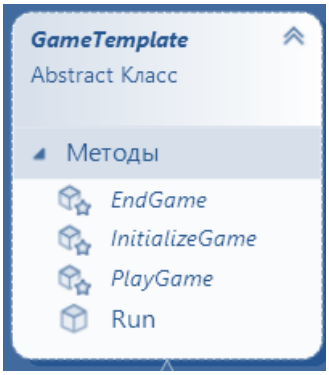

Рис.1.3.23 Діаграма класу `GameTemplate`

Клас `GameTemplate` є абстрактним класом, який визначає шаблон для гри. Він використовує шаблонний метод, який складається з трьох основних кроків гри: ініціалізація, гра і завершення гри.

Опис методів цього класу:

`InitializeGame()`: Абстрактний метод, який визначає логіку ініціалізації гри. Конкретна реалізація цього методу буде залежати від підкласу, який розширює `GameTemplate`.

`PlayGame()`: Абстрактний метод, який визначає логіку гри. Конкретна реалізація цього методу буде залежати від підкласу, який розширює `GameTemplate`.

`EndGame()`: Абстрактний метод, який визначає логіку завершення гри. Конкретна реалізація цього методу буде залежати від підкласу, який розширює `GameTemplate`.

`Run()`: Метод, який виконує шаблонний метод гри. Викликає послідовно методи `InitializeGame()`, `PlayGame()` і `EndGame()`.

`CardGame`

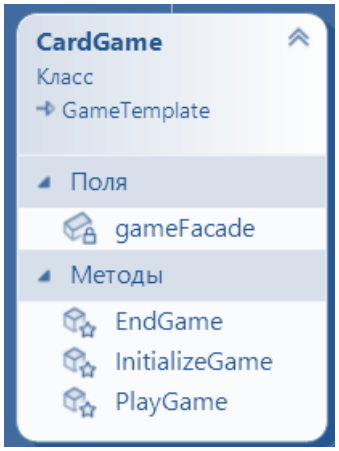

Рис.1.3.24 Діаграма класу `CardGame`

Клас `CardGame` є конкретною реалізацією `GameTemplate` для гри з картами. Він розширює базовий клас і надає конкретну реалізацію для кожного з трьох методів.

`InitializeGame()`: Реалізація методу `InitializeGame()`, в якому виконується ініціалізація гри з картами. Створюється об'єкт `GameFacade`, який використовується для реалізації логіки гри.

`PlayGame()`: Реалізація методу `PlayGame()`, в якому виконується гра з картами. Викликається метод `Play()` об'єкта `GameFacade`, який відповідає за реалізацію ходів у грі.

`EndGame()`: Реалізація методу `EndGame()`, в якому виконується завершення гри з картами.

Клас `CardGame` використовує шаблонний метод `Run()`, щоб виконати послідовність кроків гри: ініціалізацію, гру і завершення гри.

2. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ

2.1 Обґрунтування вибору та опис шаблонів проектування для програмної реалізації консольного-додатку.

Шаблони проєктування (патерн, pattern) – це ефективні засоби вирішення характерних задач проєктування, зокрема проєктування комп'ютерних програм.

Патерн не є закінченим зразком проєкту, який може бути перетворений на код. Це, скоріше, опис або зразок того, як вирішити задачу таким чином, щоб це можна було використати в різних ситуаціях.

Об'єктноорієнтовані шаблони показують відношення та взаємодію між класами та об'єктами без визначення того, які кінцеві класи чи об'єкти програми будуть використовуватися.

1) **Factory method**

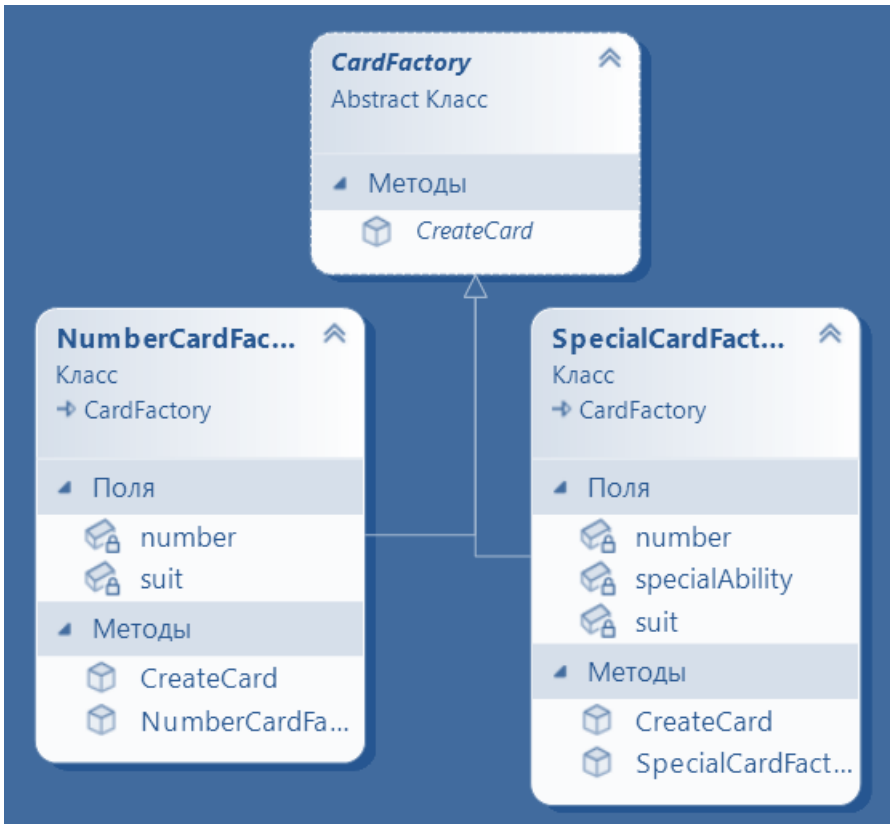

Рис.2.1.1 Діаграма шаблону Factory method

Фабричний метод (Factory method) – шаблон, який визначає інтерфейс для створення об’єкта, але рішення про те, який саме об’єкт створювати, залишає за підкласами. Таким чином, «*Фабричний метод*» дозволяє класу делегувати дію інстанціювання підкласам. Шаблон проектування Factory Method використовується для створення об'єктів, не вказуючи конкретний клас цих об'єктів. Він визначає інтерфейс, який дозволяє створювати об'єкти, але конкретну реалізацію цього створення визначають підкласи.

Використання шаблону Factory Method має такі переваги:

1. Розділення створення об'єктів від їх використання: Шаблон Factory Method дозволяє зосередитися на створенні об'єктів у відповідних фабриках, розділяючи це від логіки використання цих об'єктів. Це сприяє зменшенню залежностей і полегшує модифікацію коду.
2. Підтримка принципу "відкрито для розширення, закрито для змінення" (Open-Closed Principle): Шаблон Factory Method дозволяє додавати нові типи об'єктів без необхідності змінювати існуючий код. Це дозволяє розширювати функціональність системи, не порушуючи її цілісності.
3. Забезпечення гнучкості інстанціювання об'єктів: Завдяки шаблону Factory Method можна змінювати спосіб створення об'єктів, використовуючи різні фабрики, що реалізують специфічні логіки створення. Це дозволяє легко змінювати інстанціювання об'єктів в залежності від потреб програми.
4. Забезпечення однорідного інтерфейсу для створення об'єктів: Класи-фабрики реалізують спільний інтерфейс `CardFactory`, що дозволяє однорідно створювати об'єкти. Це полегшує розуміння коду та спрощує його управління.

Учасники шаблону Factory Method:

1. `Card`: Абстрактний клас, який представляє загальні властивості та методи для всіх видів карт.

2. `NumberCard`: Конкретний клас, який реалізує числові карти.
3. `SpecialCard`: Конкретний клас, який реалізує спеціальні карти з особливими властивостями.
4. `CardFactory`: Абстрактний клас фабрики, який визначає спільний інтерфейс для створення карт.
5. `NumberCardFactory` і `SpecialCardFactory`: Конкретні класи фабрик, які реалізують створення числових карт і спеціальних карт відповідно.

Використання шаблону Factory Method дозволяє звести до мінімуму залежності від конкретних класів карт, а також спрощує створення та розширення різних типів карт. Код стає більш гнучким, розширюваним і підтримує принципи доброго програмування.

2) **Decorator**

Рис.2.1.2 Діаграма шаблону Decorator

Decorator (Декоратор) - використовується для реалізації патерну "декоратор" для розширення функціональності карт. Динамічне додавання нової функціональності до існуючих об’єктів чи її видалення (*Decorator*).

Клас `CardDecorator` є базовим класом для декораторів карт. Клас `SpecialCardDecorator` є конкретними декораторами, які додають додаткову функціональність до базової карти. Кожен декоратор має посилання на об'єкт базової карти і додаткові методи або властивості, які розширюють поведінку базової карти.

Шаблон проектування Decorator використовується для динамічного додавання нової функціональності до об'єктів без зміни їх базової структури. Він дозволяє розширювати функціональність об'єкта, обгортаючи його в декоратори, які додають нові можливості.

В контексті CardGame використання шаблону Decorator має такі переваги:

1. Гнучкість розширення функціональності: Шаблон Decorator дозволяє додавати нові функції до об'єктів на льоту, без необхідності змінювати базовий клас. Це дозволяє додавати, комбінувати та видаляти функції динамічно під час виконання програми.
2. Збереження принципу "відкрито для розширення, закрито для змінення" (Open-Closed Principle): Декоратори дозволяють розширювати функціональність об'єкта, не змінюючи його базової реалізації. Це сприяє збереженню цілісності коду і полегшує його розширення без необхідності модифікувати вже наявний код.
3. Розділення відповідальностей: Кожен декоратор відповідає за свою конкретну функціональність. Це спрощує структуру коду і забезпечує виокремлення логіки розширення в окремі класи.
4. Можливість створення комплексних функцій: Завдяки можливості комбінувати декоратори між собою, можна створювати складні функції з

різними рівнями декорування. Це дозволяє створювати різноманітні комбінації функцій з одним базовим об'єктом.

Учасники шаблону Decorator:

1. `Card`: Базовий клас, який представляє об'єкт, до якого будуть додаватись декоратори.
2. `CardDecorator`: Абстрактний клас декоратора, який розширює функціональність базового класу `Card`. Він зберігає посилання на об'єкт базового класу і передає виклики до нього.
3. `SpecialAbilityCardDecorator`: Конкретний клас декоратора, який додає спеціальну функціональність до об'єкту `Card`. У цьому конкретному випадку, декоратор змінює візуальне представлення та поведінку об'єкту.

Використання шаблону Decorator дозволяє гнучко розширювати функціональність об'єктів без втручання у їх код, що сприяє полегшенню розширення і підтримки принципів доброго програмування.

3) Observer

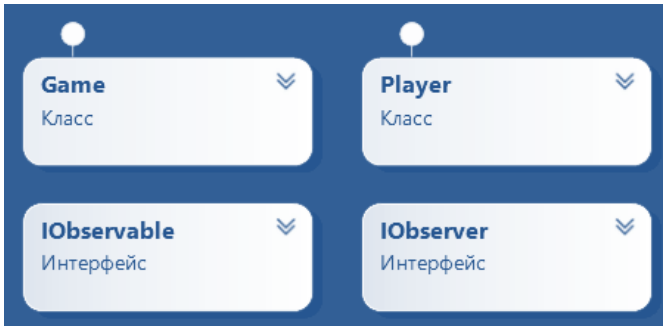

Рис.2.1.3 Діаграма шаблону Observer

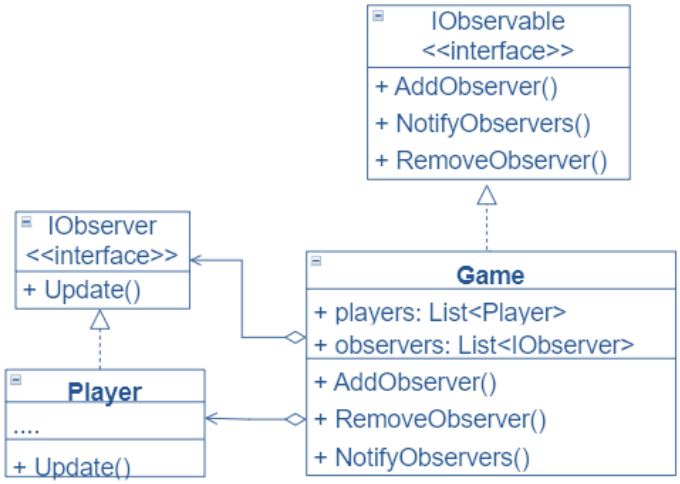

Рис.2.1.4 Діаграма шаблону Observer

Спостерігач — це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах.

Шаблон програмування Observer використовується для забезпечення взаємодії між об'єктами, де зміна стану одного об'єкта викликає автоматичну оновлення всіх залежних від нього об'єктів, які його спостерігають.

Інтерфейс `IObserver` оголошує метод `Update()`, який викликається при сповіщенні спостережуваного об'єкту. Клас `Game` реалізує інтерфейс `IObservable` і містить список спостерігачів (`observers`). Він також містить методи для додавання/видалення спостерігачів та сповіщення їх про зміни. Клас `Player` реалізує інтерфейс `IObserver` і містить метод `Update()`, який викликається при сповіщенні.

Основні компоненти шаблону Observer:

1. `IObserver`: Інтерфейс, який описує метод `Update()`, який буде викликаний спостережувачами при зміні стану.
2. `IObservable`: Інтерфейс, який описує методи для додавання, видалення та сповіщення спостережувачів.
3. `Game`: Клас, який виступає в ролі спостережуваного об'єкта. Він має список спостерігачів (`observers`), методи для додавання, видалення та

сповіщення спостережувачів. Клас також має методи для здійснення дій у грі та перевірки стану гри.

4. `Player`: Клас, який виступає в ролі спостерігача. Він реалізує метод `Update()`, який викликається при оновленні стану гри.

Використання шаблону Observer має такі переваги:

1. Розділення обов'язків: Шаблон дозволяє розділити логіку спостереження від логіки спостережування. Спостережувані об'єкти не залежать від конкретних класів спостерігачів і можуть легко сповіщати багато спостерігачів без необхідності знати про них деталі.
2. Розширюваність: За допомогою шаблону Observer можна додавати нові спостерігачі і спостережувані об'єкти без зміни коду спостерігання. Це спрощує розширення функціональності і забезпечує гнучкість системи.
3. Модульність: Компоненти системи, такі як спостерігачі і спостережувані об'єкти, можуть бути легко перевикористані і тестовані незалежно від інших частин системи. Це полегшує модульне тестування і підтримку коду.
4. Зменшення зв'язності: Використання шаблону Observer дозволяє зменшити залежність між класами, оскільки спостерігачі не пов'язані напряму зі спостережуваними об'єктами. Це полегшує зміни і рефакторинг коду.

Учасники шаблону Observer:

1. Спостерігач (`Observer`): Клас, який спостерігає за спостережуваним об'єктом і реалізує метод `Update()`, який викликається при зміні стану спостережуваного об'єкта.
2. Спостережуваний (`Observable`): Клас, який має можливість додавати, видаляти і сповіщати спостерігачів. Він також зберігає стан, за яким спостерігають спостерігачі.
3. Клієнт: Клас, який використовує спостерігачів і спостережувані об'єкти для взаємодії та обміну інформацією.

Загальною метою шаблону Observer є визначення залежностей "один до багатьох" між об'єктами, забезпечення гнучкості та спрощення розширення функціональності.

4) **Command**

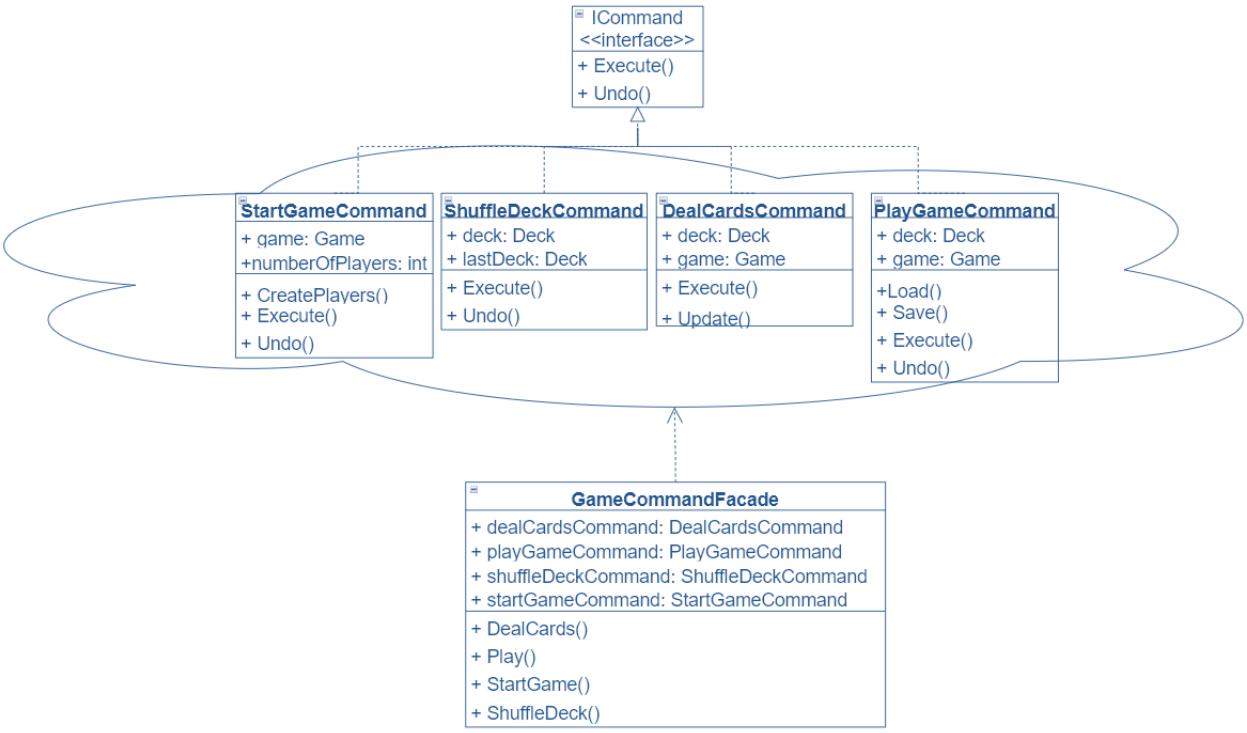

Рис.2.1.5 Діаграма шаблону Command

Command (Команда) - використовується для реалізації патерну "команда". **Команда (Command)** – поведінковий шаблон, який інкапсулює різні алгоритми в єдину сутність (об’єкт), завдяки чому можна параметризувати клієнтів різними запитами, вести історію виконаних операцій та підтримувати скасування операцій.

Шаблон програмування Command використовується для імплементації механізму виконання операцій як об'єкти. Він дозволяє упакувати виклики методів в окремі об'єкти, що виконуються в пізніший час або з можливістю відміни. Основна ідея шаблону полягає у винесенні операцій до окремих класів команд, які реалізують інтерфейс `ICommand`.

Плюси використання шаблону Command:

1. Розділення класів, які ініціюють операцію (відправників) від класів, які знають, як її виконати (отримувачів). Це полегшує підтримку та розширення системи.
2. Можливість реалізації черги команд, логування, відміни та повторення операцій.
3. Забезпечення гнучкості, оскільки нові команди можуть бути додані без зміни вже існуючого коду.
4. Дозволяє розширювати логіку виконання команд без модифікації класів команд.

Учасники шаблону:

1. `ICommand`: Це інтерфейс, який описує методи `Execute()` та `Undo()`. `Execute()` виконує дії, пов'язані з командою, а `Undo()` відміняє виконану команду.
2. Конкретні команди (`StartGameCommand`, `ShuffleDeckCommand`, `DealCardsCommand`, `PlayGameCommand`): Ці класи реалізують інтерфейс `ICommand` та містять логіку виконання конкретних операцій. Кожен клас команди може мати свої поля, що зберігають стан, необхідний для виконання та відміни команди.
3. `GameCommandFacade`: Цей клас є фасадом, який надає простий інтерфейс для виконання послідовності команд гри. Він упаковує команди створення гри (`StartGameCommand`), тасування колоди (`ShuffleDeckCommand`), роздачі карт (`DealCardsCommand`) та гри (`PlayGameCommand`) в один об'єкт, що дозволяє легко керувати ходом гри.

У даному коді шаблон Command використовується для реалізації механізму гри. Кожна команда відповідає певній операції, такі як старт гри, тасування колоди, роздача карт та гра. Клас `GameCommandFacade` дозволяє послідовно викликати ці команди і керувати ходом гри. Кожна команда має методи

`Execute()`, який виконує операцію, та `Undo()`, який відміняє виконану команду.

Цей шаблон дозволяє легко додавати нові операції до гри, зберігаючи їх в окремих класах команд. Також він надає можливість відмінити виконану команду та повторити її за потреби. Шаблон Command допомагає зробити код більш модульним, легким для розширення та підтримки.

5) Template Method

Рис.2.1.6 Діаграма шаблону Template Method

Шаблонний метод (template method) – поведінковий шаблон, який визначає функціональність конкретних методів в рамках лише абстрактних сутностей.

Шаблон програмування Template Method використовується для визначення скелету алгоритму у базовому класі, залишаючи деякі кроки реалізації підкласам. Цей шаблон дозволяє підкласам змінювати певні кроки алгоритму, не змінюючи загальної структури.

Плюси використання шаблону Template Method:

1. Дозволяє встановити загальну структуру алгоритму в базовому класі і дозволяє підкласам замінювати або розширювати окремі кроки алгоритму.
2. Забезпечує гнучкість та розширюваність системи, оскільки можна додавати нові підкласи, які реалізують власні варіації алгоритму, без зміни коду базового класу.
3. Спрощує повторне використання коду, оскільки загальна логіка алгоритму знаходиться в базовому класі, і його можна використовувати в різних підкласах з різними реалізаціями кроків.
4. Дозволяє контролювати потік виконання алгоритму та послідовність його кроків.

Учасники шаблону:

1. `GameTemplate`: Це абстрактний клас, який визначає скелет алгоритму з декількома абстрактними методами, які підкласи повинні реалізувати. Метод `Run()` викликає ці абстрактні методи у визначеній послідовності, встановлюючи загальну структуру алгоритму.
2. `CardGame`: Цей клас є конкретним підкласом `GameTemplate` і реалізує конкретну гру з картами. Він перевизначає абстрактні методи `InitializeGame()`, `PlayGame()` та `EndGame()`, щоб визначити конкретні кроки гри з картами.

У даному коді шаблон Template Method використовується для створення загальної структури гри з картами. Базовий клас `GameTemplate` визначає скелет алгоритму гри, а підклас `CardGame` надає конкретні реалізації окремих кроків гри.

Це дозволяє легко створювати нові види ігор, створюючи нові підкласи `GameTemplate` та перевизначаючи відповідні методи. Всі додаткові функціональні можливості, пов'язані зі структурою гри, залишаються в

базовому класі, тоді як специфічні деталі і реалізації перекладаються на підкласи.

Це дозволяє забезпечити високий рівень гнучкості та розширюваності системи і дозволяє швидко впроваджувати нові види ігор, змінювати порядок виконання кроків та логіку гри без необхідності змінювати код в багатьох місцях.

6) Facade

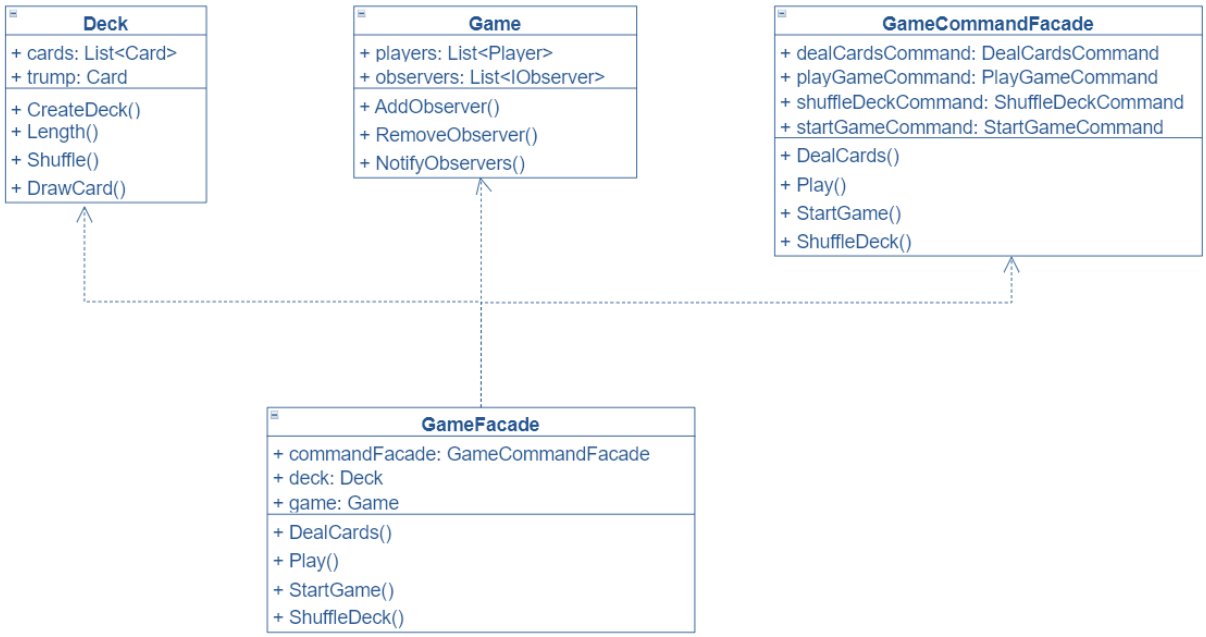

Рис.2.1.7 Діаграма шаблону Facade

Шаблон програмування Facade (Фасад) використовується для створення простого інтерфейсу, який приховує складну логіку або підсистему від клієнтського коду. Він надає високорівневий інтерфейс для спрощення взаємодії з підсистемою. Реорганізація системи, що складається з багатьох підсистем, у слої з єдиною точкою входу (*Façade*).

Плюси використання шаблону Facade:

1. Спрощує використання складної підсистеми, надаючи простий інтерфейс з обмеженим набором операцій.
2. Забезпечує високий рівень абстракції та розбиття системи на окремі компоненти, що полегшує розвиток, підтримку та розширення системи.

3. Зменшує залежність від складної підсистеми, оскільки клієнтський код взаємодіє тільки з фасадом, а не з безпосередньою підсистемою.
4. Підвищує розуміння коду, оскільки фасад надає явний інтерфейс, що дозволяє зрозуміти, як взаємодіяти з підсистемою.

Учасники шаблону:

1. `Deck`: Цей клас представляє підсистему, що відповідає за роботу з колодою карт. Він має методи для створення колоди, перемішування та видачі карт.
2. `GameFacade`: Цей клас виступає як фасад і надає простий інтерфейс для взаємодії з підсистемою гри. Він містить посилання на об'єкти підсистеми (гру та колоду) і надає методи для запуску гри, перемішування колоди та роздачі карт.
3. `GameCommandFacade`: Цей клас використовується в `GameFacade` для реалізації методів фасаду. Він виконує внутрішні команди та взаємодіє з підсистемою (грою та колодою) для виконання відповідних операцій.

Фасадний шаблон використовується тоді, коли необхідно спростити складний інтерфейс підсистеми, забезпечити простоту використання та зменшити залежність від деталей реалізації. В даному випадку, використання фасаду дозволяє клієнтському коду легко взаємодіяти з грою та колодою карт, не звертаючись безпосередньо до їх складної логіки.

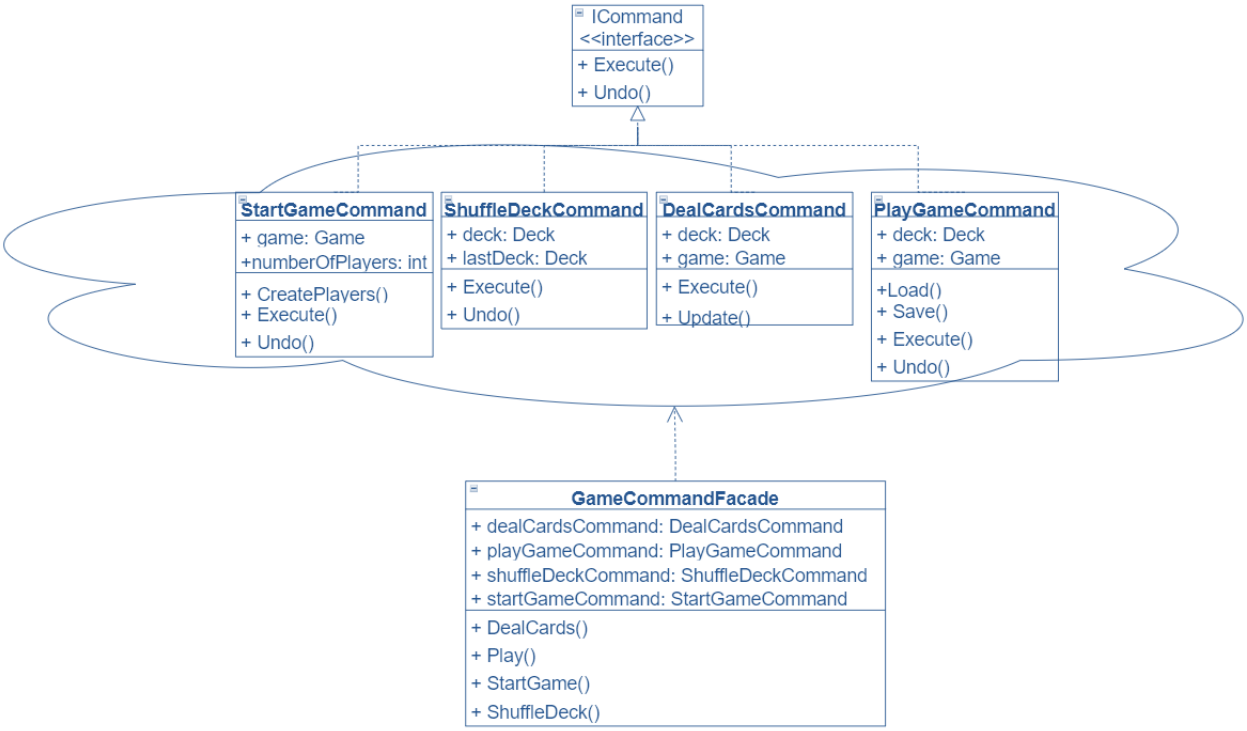

Рис.2.1.10 Діаграма шаблону Фасад

У даному коді ми маємо набір класів, які реалізують інтерфейс ICommand. Цей інтерфейс визначає два методи: Execute() (Виконати) і Undo() (Скасувати).

1. Клас `StartGameCommand` представляє команду "Почати гру". У методі Execute() він запитує користувача про кількість гравців та їх імена, створює об'єкти гравців і додає їх до гри. Метод Undo() відновлює початковий стан гри.
2. Клас `ShuffleDeckCommand` представляє команду "Перемішати колоду". У методі Execute() він зберігає поточний стан колоди, перемішує її і метод Undo() відновлює колоду до попереднього стану.
3. Клас `DealCardsCommand` представляє команду "Роздати карти". У методі Execute() він роздає кожному гравцю по 6 карт з колоди. Метод Undo() скидає всі картки гравців.
4. Клас `PlayGameCommand` представляє команду "Грати гру". У методі Execute() він ініціалізує поточного гравця, запитує користувача про стратегію вибору картки, виконує хід гравця і викликає метод Undo() для

скасування ходу. Метод Undo() перевіряє, чи користувач не вказав "STOP" для зупинки гри, якщо так, то він зберігає поточний стан гри у файлі "gamestate.dat" і завершує гру. Якщо користувач не вказав "STOP", він пропонує зберегти гру і завантажити збережений стан гри.

Клас `GameCommandFacade` виступає як фасад, який надає простий інтерфейс для взаємодії з грою. Він містить посилання на об'єкти різних команд і надає публічні методи для виклику цих команд. Цей фасад дозволяє спростити використання команд і приховати складність викликів окремих методів команд зовнішньому користувачу. Замість безпосереднього створення об'єктів команд і виклику їх методів, користувач може викликати методи фасаду для виконання певних дій з грою. Фасад вирішує проблему складності використання окремих команд і дозволяє зосередитися на вищорівневих операціях з грою.

Отже, використання фасаду `GameCommandFacade` у даному коді дозволяє спростити використання окремих команд і надає зручний інтерфейс для взаємодії з грою. Користувач може викликати методи фасаду для початку гри, перемішування колоди, роздачі карт і гри взагалі, не заморочуючись деталями реалізації окремих команд. Фасад виконує всі необхідні дії за користувача, спрощуючи процес взаємодії з грою.

7) Strategy

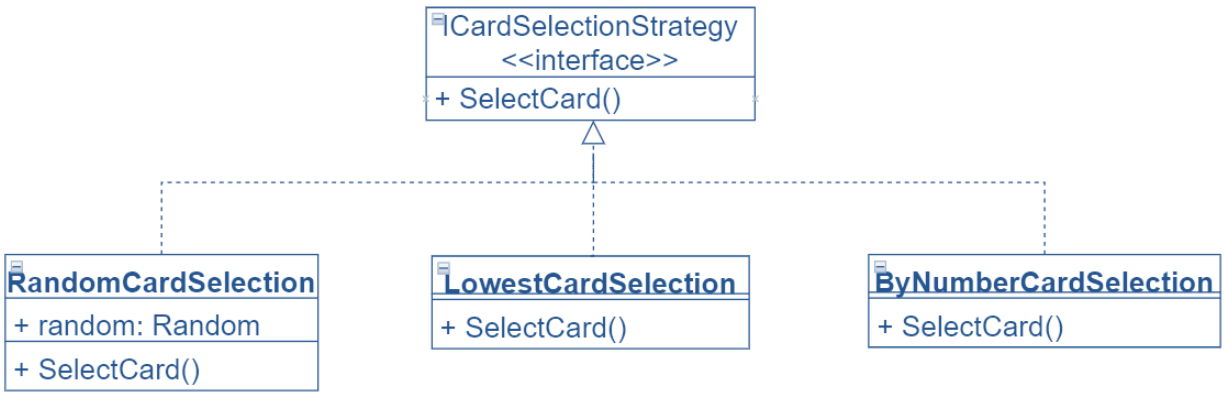

Рис.2.1.9 Діаграма шаблону Strategy

Strategy (Стратегія) - використовується для реалізації патерну "стратегія" для логіки гри. **Стратегія (strategy)** – поведінковий шаблон, який визначає

сімейство алгоритмів, інкапсулює кожен з них та робить їх взаємозамінними. Дозволяє змінювати алгоритми незалежно від коду клієнтів.

Шаблон програмування Strategy використовується для визначення сімейства алгоритмів, інкапсуляції кожного алгоритму та забезпечення можливості обміну їх між собою. Основна ідея шаблону полягає у визначенні окремих класів, що реалізують спільний інтерфейс, що дозволяє замінювати один алгоритм на інший без впливу на клієнтський код.

Плюси використання шаблону Strategy:

1. Відокремлення алгоритмів від клієнтського коду, що полегшує розширення, підтримку та тестування системи.
2. Можливість динамічно обирати алгоритм під час виконання програми.
3. Забезпечення принципу "відкрито для розширення, закрито для модифікації".
4. Підвищення повторного використання коду, оскільки різні алгоритми можуть використовуватись у різних частинах програми.

Учасники шаблону:

1. `ICardSelectionStrategy`: Це інтерфейс, який описує метод `SelectCard()`, що представляє алгоритм вибору карти гравцем. Класи, що реалізують цей інтерфейс, визначають різні стратегії вибору карти.
2. Конкретні стратегії (`RandomCardSelectionStrategy`, `LowestCardSelectionStrategy`, `ByNumberCardSelectionStrategy`): Ці класи реалізують інтерфейс `ICardSelectionStrategy` та визначають різні алгоритми вибору карти. Наприклад, `RandomCardSelectionStrategy` вибирає випадкову карту з решти карт гравця, `LowestCardSelectionStrategy` вибирає найменшу карту, а `ByNumberCardSelectionStrategy` дозволяє гравцеві вибрати карту зі списку доступних карт.

3. Клас `Player`: Цей клас має методи, що використовують стратегію (`ICardSelectionStrategy`) для вибору карти гравцем та видалення її з його руки.

У даному коді шаблон Strategy використовується для визначення різних стратегій вибору карти гравцем. Класи `RandomCardSelectionStrategy`, `LowestCardSelectionStrategy` та `ByNumberCardSelectionStrategy` реалізують спільний інтерфейс `ICardSelectionStrategy` і надають різні алгоритми вибору карти. Клас `Player` використовує вибрану стратегію для вибору та видалення карти з руки гравця.

Цей шаблон дозволяє легко додавати нові стратегії вибору карти без зміни існуючого коду. Він також спрощує тестування різних алгоритмів, оскільки кожен алгоритм може бути перевірений окремо. Шаблон Strategy полегшує розширення та модифікацію системи, забезпечуючи гнучкість та розділення відповідальностей між класами.

8) **Bridge**

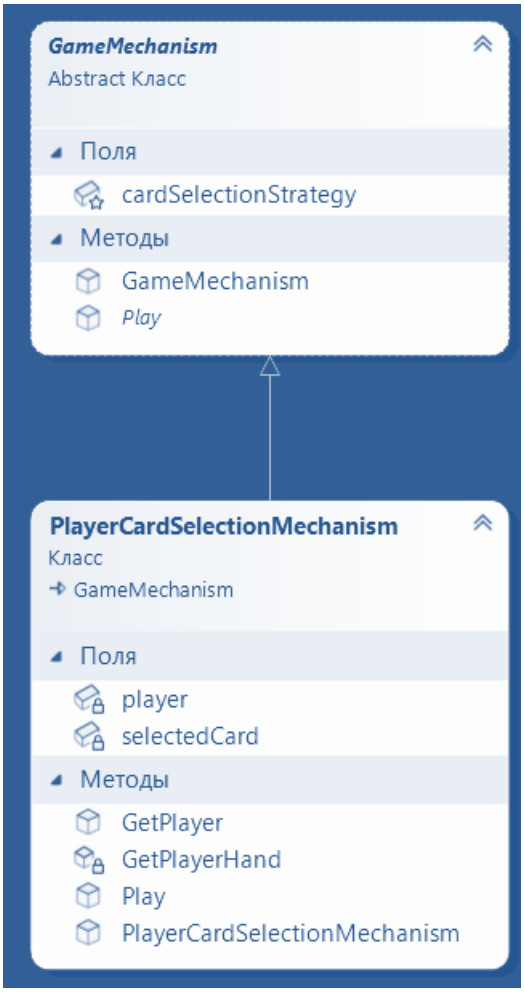

Рис.2.1.10 Діаграма шаблону Bridge

Шаблон програмування Bridge використовується для розділення абстракції від її реалізації, дозволяючи їм змінюватись незалежно одне від одного. Він використовує принцип композиції замість успадкування для розділення функціональності між двома наборами класів.

Плюси використання шаблону Bridge:

1. Розділення абстракції від реалізації дозволяє їм змінюватись незалежно одне від одного. Зміна в одному з них не впливає на інший.
2. Забезпечує гнучкість та розширюваність системи, оскільки можна додавати нові підкласи абстракції та реалізації без зміни існуючого коду.
3. Дозволяє розширювати функціональність шляхом додавання нових класів абстракції та реалізації та комбінування їх між собою.

4. Спрощує тестування, оскільки абстракція та реалізація можуть бути тестовані окремо.

Учасники шаблону:

1. `GameMechanism`: Це абстрактний клас, який представляє абстракцію, тобто високорівневий модуль. Він містить посилання на об'єкт інтерфейсу `ICardSelectionStrategy` та оголошує метод `Play()`, який використовує об'єкт `cardSelectionStrategy` для виконання гри.
2. `PlayerCardSelectionMechanism`: Цей клас є конкретною реалізацією абстракції `GameMechanism`. Він має посилання на об'єкт `Player` та використовує об'єкт `cardSelectionStrategy` для вибору карти гравцем. Метод `Play()` викликає метод `SelectCard()` залежно від обраної стратегії.

У даному коді шаблон Bridge використовується для розділення вибору карти гравцем (`PlayerCardSelectionMechanism`) від конкретної стратегії вибору карти (`ICardSelectionStrategy`). Це дозволяє змінювати та комбінувати різні стратегії вибору карти без зміни коду гравця або самої гри. Це полегшує розширення та модифікацію системи, а також спрощує тестування окремої функціональності.

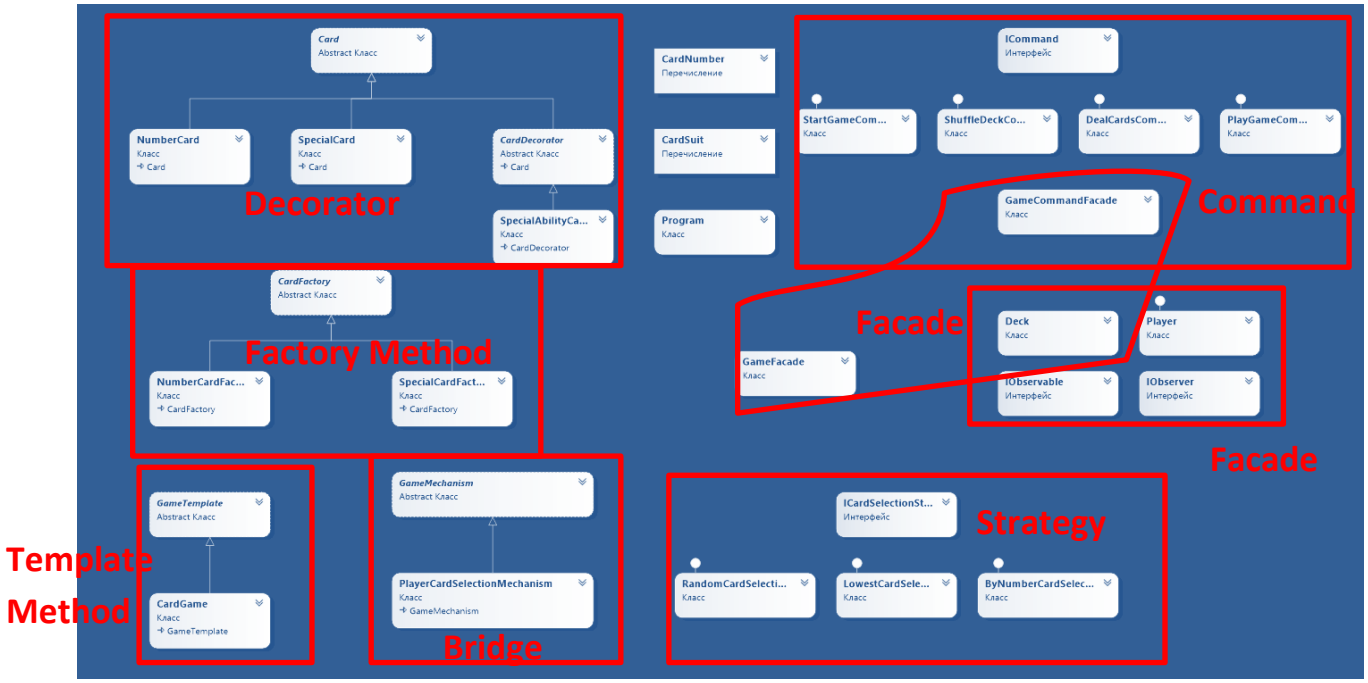

Рис.2.1.11 Діаграма класів з виділеними шаблонами

2.3 Опис результатів роботи програми

Для демонстрації результатів роботи розроблено консольний застосунок гру в карти.

```
Initializing the card game...
Starting the game...
Enter the number of players
4
Enter name of 0 player
p1
Enter name of 1 player
p2
Enter name of 2 player
p3
Enter name of 3 player
p4
Shuffling the deck...
Dealing cards to players...
Playing the card game...
```

Рис.2.3.1 Ілюстрація запуску програми

Бачимо, що при запуску відбувається:

1. Ініціалізація гри:
   - Виводиться повідомлення "Initializing the card game..." у консоль.
2. Початок гри:
   - Виводиться повідомлення "Starting the game..." у консоль.
3. Введення кількості гравців:
   - Виводиться повідомлення "Enter the number of players" у консоль.
   - Гравець вводить кількість гравців (у даному випадку 4).
   - Кількість гравців зберігається для подальшого використання.
4. Введення імен гравців:
   - Виводиться повідомлення "Enter name of [номер гравця] player" у консоль для кожного гравця.
   - Гравець вводить ім'я гравця (у даному випадку "p1" та "p2" …).
   - Імена гравців зберігаються для подальшого використання.
5. Перетасування колоди:

- Виводиться повідомлення "Shuffling the deck..." у консоль.

6. Роздача карт гравцям:

- Виводиться повідомлення "Dealing cards to players..." у консоль.

- Кожному гравцю роздається 6 карт з колоди.

- Кожна карта з колоди передається гравцеві.

7. Проведення гри в картки:

- Виводиться повідомлення "Playing the card game..." у консоль.

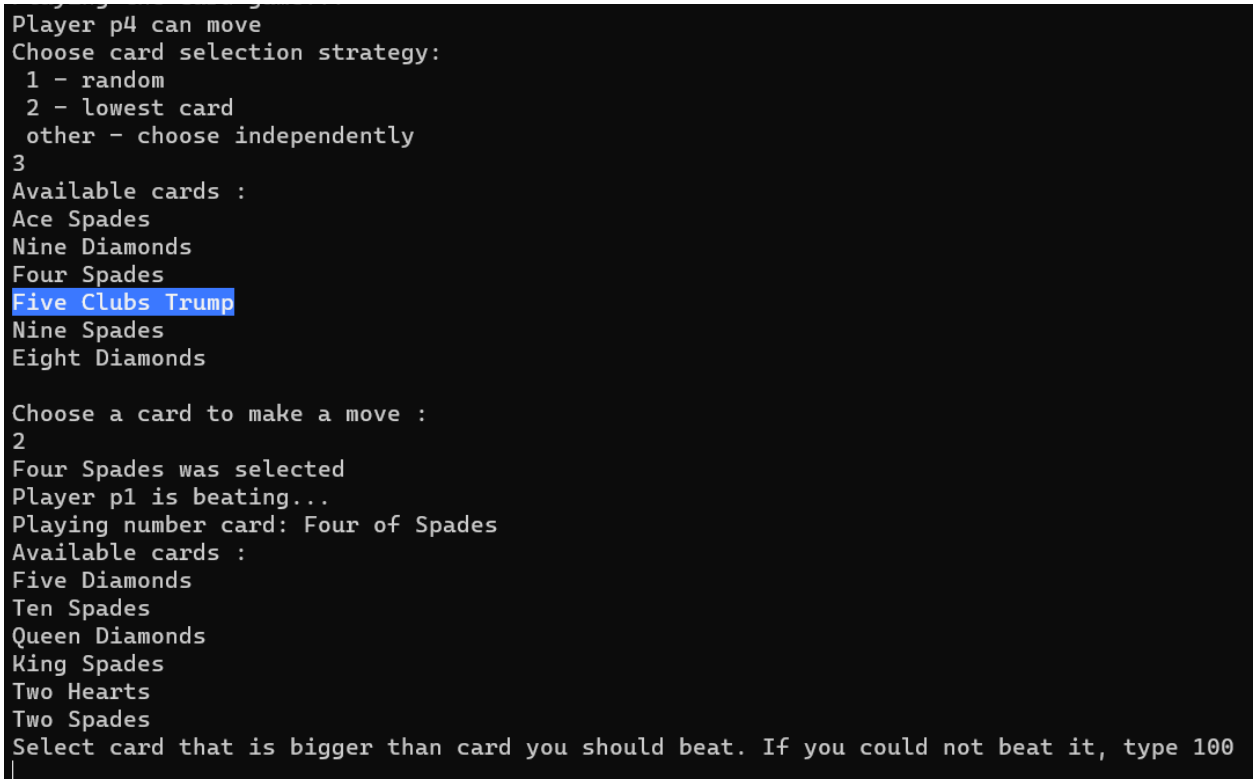

Рис.2.3.2 Ілюстрація продовження роботи програми

8. Хід гравця p4:

- Виводиться повідомлення "Player p4 can move" у консоль.

- Виводиться повідомлення "Choose card selection strategy:" у консоль.

- Гравець вводить стратегію вибору карт(у даному випадку other – choose independently).

- Гравець обирає карту, якою ходить.

- Виводиться повідомлення з ім’ям гравця, який відбивається, та його доступні карти.

```
Select card that is bigger than card you should beat. If you could not beat it, type 100
1
Playing number card: Ten of Spades
Card is beaten
Available cards :
Queen Clubs Trump
King Hearts
Jack Spades
Six Hearts
Ace Diamonds
Three Hearts
Used cards :
Four Spades
Ten Spades
Do you(player p2) want to throw up? Enter Yes/No
```

Рис.2.3.3 Ілюстрація продовження роботи програми

-Гравець обирає карту, якою буде відбиватися.

-Карта відбита і кожному гравцю пропонується підкинути.

Далі гра так може продовжуватись, доки один з гравців не витратить всі карти або хтось її не зупинить.

```
Do you(player p4) want to throw up? Enter Yes/No
no
Player p2 can move
Choose card selection strategy:
 1 - random
 2 - lowest card
 other - choose independently
2
Three Hearts was selected
Player p3 is beating...
```

Рис.2.3.4 Приклад використання стратегії вибору найменшої карти

```
Player p3 can move
Choose card selection strategy:
 1 - random
 2 - lowest card
 other - choose independently
1
Two Diamonds was selected
Player p4 is beating...
```

Рис.2.3.5 Приклад використання шаблону довільної карти

```
p1 has been eliminated from the game!
p2 has been eliminated from the game!
Game Stop
Ending the card game...
```

Рис.2.3.6 Визначення переможців(хто останній вибув – програв)

```
Select card that is bigger than card you should beat. If you could not beat it, type 100
5
Playing number card: Four of Diamonds
Card is beaten
Available cards :
Five Diamonds
Queen Diamonds
King Spades
Two Hearts
Two Spades
Five Spades
Used cards :
Two Diamonds
Four Diamonds
Do you(player p1) want to throw up? Enter Yes/No
Yes
Choose a card to throw up :
3
Player p4 is beating...
Playing number card: Two of Spades
Available cards :
Ace Spades
Nine Diamonds
Five Clubs Trump
Nine Spades
Eight Diamonds
Select card that is bigger than card you should beat. If you could not beat it, type 100
```

Рис.2.3.7 Приклад підкидання карти

```
Four Clubs was selected
Player P2 is beating...
Playing number card: Four of Clubs
Available cards :
Ace Diamonds
Four Spades Trump
Four Hearts
Nine Hearts
Five Diamonds
Eight Spades Trump
Select card that is bigger than card you should beat. If you could not beat it, type -1
-1
```

Рис.2.3.8 Приклад ходу «беру»

```
Playing special card with ability: Trump
Card is beaten
If you want to stop game print STOP
STOP
p1 has been eliminated from the game!
p2 has been eliminated from the game!
Game Stop
Ending the card game...
```

Рис.2.3.9 Приклад передчасного завершення гри.

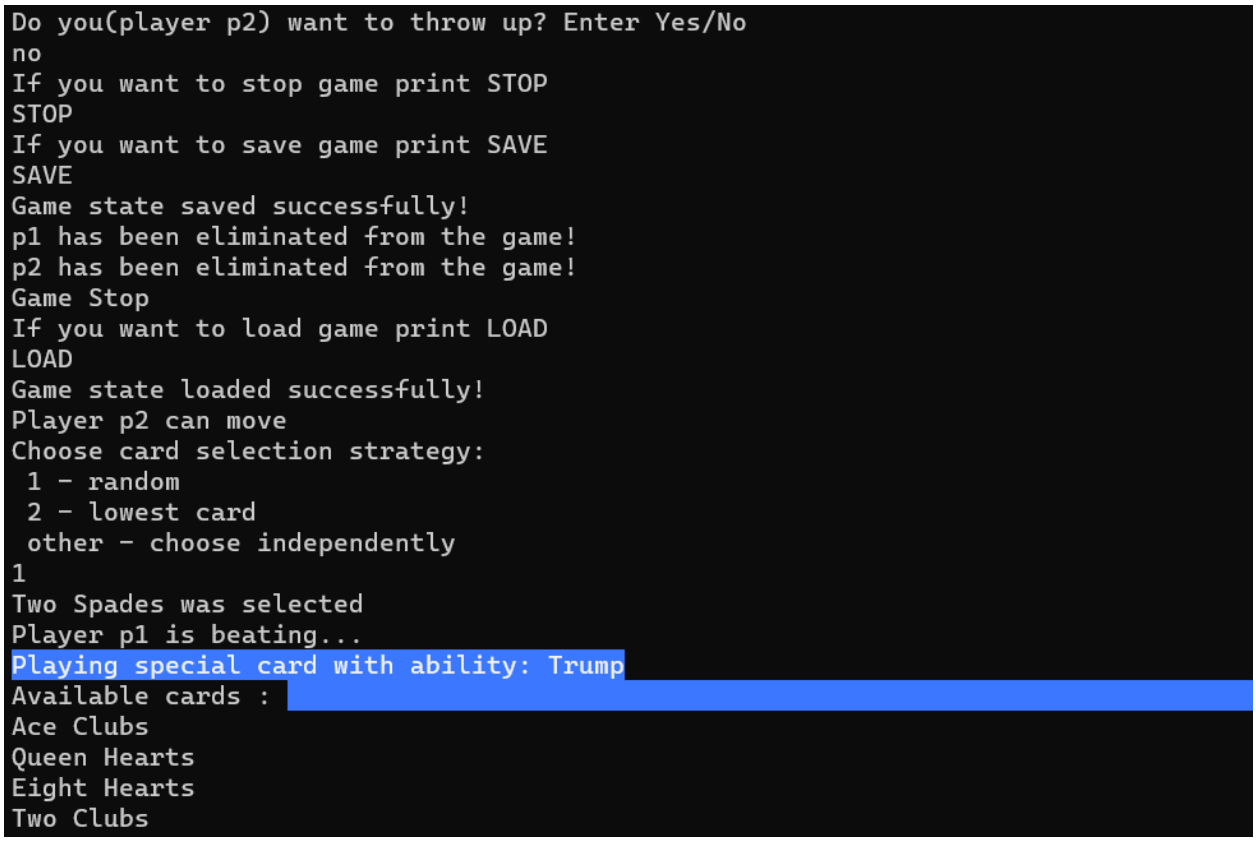

Рис.2.3.10 Приклад зберігання гри

Висновки

У даній курсовій роботі була розглянута тема використання шаблонів проектування в об’єктно-орієнтованому програмуванні, з фокусом на застосування їх у розробці гри в карти. Було вивчено основні шаблони проектування в ООП і їх практичне використання для створення функціоналу гри в карти.

В результаті виконання даної роботи були отримані наступні результати:

1. Був проведений аналіз різних шаблонів проектування і їхніх особливостей.
2. Була розроблена архітектура програми, включаючи створення класів для представлення карт, колоди, гравців та гри.
3. Була реалізована логіка гри, включаючи визначення правил гри, роздачу карт, виконання ходів гравців та визначення переможця.
4. Був реалізований консольний інтерфейс, який надає можливість створення гри, введення правил та взаємодії з гравцями.
5. Було протестовано та оцінено ефективність використання шаблонів проектування в розробці програми.

Досягнуті здобутки:

1. Використання шаблонів проектування, зокрема Фабричного методу, Спостерігача, Команди, Шаблонного методу, Фасада, Стратегії, Мосту та Декоратора. Це дозволило забезпечити більшу гнучкість, розширюваність та повторне використання коду.
2. Розроблено функціональні можливості, такі як створення гравців, колоди та гри, можливість ходити, відбивати карти та підкидати їх.
3. Забезпечено взаємодію різних логічних елементів програми, що дозволяє користувачу грати в карткову гру та взаємодіяти з іншими гравцями відповідно до правил гри.

Основні рекомендації щодо подальшого поліпшення програми і впровадження результатів роботи у виробничу сферу:

1. Додати можливість грати в різні види ігор в карти. Розширити логіку гри, щоб підтримувати різні правила, такі як Покер, Блекджек, Козирна п'ятниця і т.д. Це дозволить користувачам вибирати свої улюблені ігри і налаштовувати правила гри за своїм бажанням.
2. Реалізувати комп'ютерних гравців. Додати інтелект штучного інтелекту (ШІ), який зможе грати проти користувача або інших гравців. Це зробить програму більш цікавою і дозволить користувачам грати, коли немає реальних супротивників.
3. Додати можливість мережевої гри. Реалізувати можливість грати в гру з іншими користувачами через Інтернет. Це дозволить гравцям взаємодіяти та грати один з одним незалежно від їхнього фізичного розташування.
4. Вдосконалити графічний інтерфейс. Замість консольного інтерфейсу можна розробити графічний інтерфейс користувача (GUI), що забезпечить більш зручну та привабливу взаємодію з програмою. Додати анімацію, звукові ефекти та візуальні ефекти для покращення враження від гри.
5. Збільшити безпеку та стійкість програми. Перевірити програму на наявність можливих уразливостей та помилок. Забезпечити обробку помилок і винятків для запобігання некоректної роботи програми. Застосувати криптографічні методи для забезпечення безпеки мережевої гри та захисту конфіденційної інформації користувачів.
6. Розширити функціональні можливості програми. Додати можливості збереження прогресу гри, статистики та досягнень гравців. Реалізувати систему нагород та викликів, що надихатимуть гравців до подальшого розвитку.
7. Провести дослідження та оптимізацію алгоритмів штучного інтелекту для поліпшення рівня гри комп'ютерних противників. Використовувати

методи машинного навчання і глибокого навчання для тренування моделей, які здатні приймати кращі рішення в грі.

8. Збирати відгуки користувачів і проводити оновлення програми на основі отриманих даних. Враховувати пропозиції та побажання користувачів щодо покращення функцій і інтерфейсу програми.
9. Розглянути можливості комерціалізації програми. Дослідити можливості випуску програми у вигляді платного продукту або включення в існуючі гральні платформи. Розробити бізнес-стратегію для популяризації та розповсюдження програми серед широкої аудиторії.

Отже, ці рекомендації допоможуть розширити функціональність і вдосконалити якість програми, зробити її цікавішою та привабливішою для користувачів.

Список використаної літератури

1. Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software." - Рік випуску: 1994. Кількість сторінок: 395.
2. Freeman, Eric, Elisabeth Robson, Bert Bates, Kathy Sierra. "Head First Design Patterns." - Рік випуску: 2004. Кількість сторінок: 694.
3. Shalloway, Alan, James R. Trott. "Design Patterns Explained: A New Perspective on Object-Oriented Design." - Рік випуску: 2004. Кількість сторінок: 480.
4. Fowler, Martin. "Refactoring: Improving the Design of Existing Code." - Рік випуску: 1999. Кількість сторінок: 431.
5. Fowler, Martin. "Patterns of Enterprise Application Architecture." - Рік випуску: 2002. Кількість сторінок: 560