Atypon training
Java and DevOps (May 2023)
Section 2

Uno Game

Assignment 3

Name: Mohammad Al-Qaisy

Date: 20th Jun 2023

Table of Contents

Introduction

The Uno Game Engine is a software component designed to provide a foundation for building Uno card game applications. Uno is a popular card game played by 2-10 players, involving a deck of 108 cards. The game consists of numbered cards (0-9), action cards (Reverse, Skip, Draw Two), and wild cards (Wild, Wild Draw Four). This game engine is intended to be used by developers to create Uno game applications with various variations and customizable rules.

Uno games can have multiple variations beyond the basic version, allowing for additional action or wild cards with unique rules. The game engine is flexible and adaptable, enabling developers to incorporate these variations and customize the game according to their specific requirements. Whether it's introducing new action cards, modifying the initial card dealing process, or implementing additional penalty rules, the Uno Game Engine offers the foundation to accommodate these changes.

The primary goal of the Uno Game Engine is to provide a robust and reliable framework for developers to create Uno game applications. It handles the core mechanics of the game, including card management, turn order, rule enforcement, and game state tracking. By utilizing the engine, developers can focus on designing the user interface, incorporating additional features, and enhancing the player experience.

Object-Oriented Design

The central class in the Uno game engine is the **Game** class, which serves as the main class for managing the game. It composes the UnoCards, Player, and GameState classes and implements the UnoGame and rulesOfUnoGame interfaces. The Game class is responsible for coordinating the gameplay, handling interactions between players, and enforcing the rules defined by the rulesOfUnoGame interface.

The **UnoCards** class represents the deck of Uno cards in the game. It is composed of individual **Card** objects, each representing a specific Uno card with its own properties and behavior. UnoCards manages card-related operations, such as shuffling the deck, dealing cards to players, and maintaining the draw and discard cards.

The **GameState** class implements the **GameAlertState** interface, providing functionality to notify players about the current state of the game. It tracks important game information, such as waiting players, skip turn, reversed turns,and other relevant game-specific details. GameState ensures that players receive appropriate alerts and updates regarding the progress of the game.

The **Player** class implements the **Observer** interface and represents a player in the Uno game. It has methods and attributes to handle player-specific actions, such as playing cards, drawing cards, and throwing cards. The Player class also includes an update method, allowing it to send notifications to other players.

The **MyGame** class inherits from the Game class and represents a customized version of the Uno game with specific rules and variations. It extends the base functionality provided by the Game class and implements additional game-specific features according to your defined rules.

Together, these classes and interfaces form the core components of the Uno game engine. The Game class acts as the orchestrator, utilizing the UnoCards, Player, and GameState classes to manage the game flow and enforce the rules. Players interact with the game through the Player class, which receives updates from the GameState and communicates with other players. The UnoCards class handles the deck and card-related operations, ensuring the availability and proper management of Uno cards throughout the game. The MyGame class extends the Game class to introduce custom rules and variations specific to your version of the Uno game.

Class / Interface	Description	Relationships
UnoGame	Interface that defines the contract for the Uno game engine.	

rulesOfUnoGame	Interface that defines the rules and variations of the Uno game.	
Game	Base class for the Uno game engine, providing common functionality and structure for the game.	Composes UnoCards, Player, GameState Implements UnoGame, rulesOfUnoGame
GameAlertState	Interface for the state pattern, provide the state of the game.	
GameState	Represents the state of the game, including the waiting players, skip turn, play card, draw card and reversed turns.	Implements GameAlertState
Card	Represents an individual card in the Uno game.	
UnoCards	Represents the deck of Uno cards and manages the card-related operations.	
Observer	Interface for the Observer pattern, defining the update method for receiving game state updates.	
Player	Represents a player in the Uno game.	Implements Observer
MyGame	Extends the base Game class and contains the game logic for the Uno game.	Inherit Game

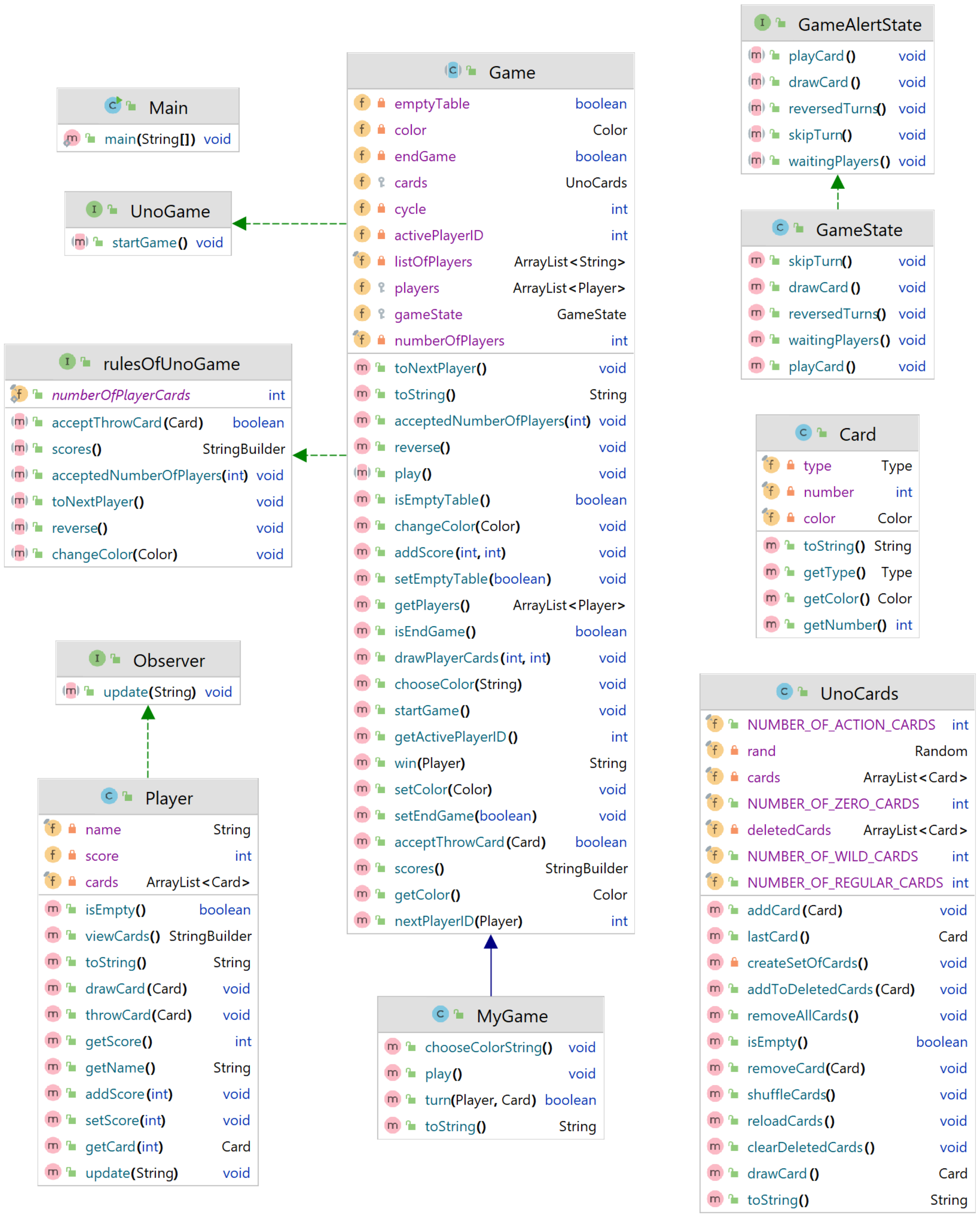

Figure: Class Diagram from IntelliJ

Design Patterns

In the Uno game engine, I have employed two design patterns to enhance the structure and functionality of the game: the State design pattern and the Observer design pattern.

State Design Pattern:

The State design pattern is used to manage different game states within the Uno game engine. The GameState class implements the GameAlertState interface, which defines the behavior related to game states. By utilizing this pattern, the game engine can easily handle various states such as waiting for players, skipping turns, playing cards, drawing cards, and reversing turns. The State pattern promotes clean code organization and simplifies the implementation of game logic.

Observer Design Pattern:

The Observer design pattern is employed to enable communication and synchronization between the different components of the Uno game engine. The Player class implements the Observer interface, which defines the update method for receiving game state updates. This pattern allows players to register as observers and be notified of relevant changes in the game state. The GameState class, acting as the observable object, can then notify the registered players about player updates or other significant events. facilitates real-time updates to players, keeping them informed about the latest game state changes.

The State design pattern is used in the Uno game engine to handle different game states, determining what the game is currently waiting for. On the other hand, the Observer pattern is employed to provide information on player actions and updates within the game.

Clean Code Principles

Naming

The naming conventions in the Uno game engine are clear, descriptive, and meaningful. They help in easily understanding the purpose of variables, functions, and classes. The names are chosen to be unambiguous, allowing for easy searchability and readability of the code. While some function names may be longer, such as "acceptedNumberOfPlayers" and "rulesOfUnoGame," they accurately represent their intended functionality. Additionally, there are no abbreviations or typos, ensuring clarity and reducing the risk of confusion. Overall, the names used in the code align with their purpose, promoting maintainability and comprehension of the codebase.

Functions

The functions in the Uno game engine have a clear and focused purpose, often related to a specific user perspective. For example, the "play" function inside the MyGame class serves as the main function for that class. As the central function developed by a developer using the game engine, it may perform multiple tasks, but its main objective is to manage the game by connecting players and their inputs with game components.

Regarding function arguments, most functions require fewer than three arguments, except for the card constructor. This adherence to simplicity helps maintain code clarity and avoids unnecessary complexity.

Overall, the functions in the Uno game engine follow the principle of doing one thing and are designed with a user perspective in mind. They aim to facilitate game management by connecting players, their inputs, and game components efficiently.

Classes

Most classes in the Uno game engine are small, except for the Game and MyGame classes. The UnoCards class, although slightly longer, remains simple in its implementation. The Game class, despite its length, includes essential get and set methods for private fields and serves as the core of the game engine, integrating all components together. On the other hand, the MyGame class contains two crucial methods: play and turn. The play method acts as the central function, managing the game by connecting players, their inputs, and game components.

Comments

The Uno game engine emphasizes clean code practices and minimal comments. While there were some initial comments during the implementation phase, they have been removed from the final code. Most of the functions are self-descriptive and adhere to clean code techniques, making the need for comments unnecessary. Only two comments remain in the MyGame class. Although comments were helpful during the implementation process, they were deemed unnecessary in the final code as they no longer served a purpose. The focus is on writing code that is self-explanatory and easily understandable without relying heavily on comments.

Dead code

The Uno game engine prioritizes the main assignment requirement and provides developers with the necessary code to customize game rules, cards, and gameplay. While efforts are made to avoid dead code, there are certain functions that may not be currently utilized but are included as features within the game engine. These functions are intended to offer developers flexibility and enable them to implement various functionalities as per their requirements.

Duplication

The code in the Uno game engine minimizes duplication, ensuring that there are very few instances of repeated code. Each method is designed to perform its specific task without relying on other methods to do its work. I have made a conscious effort to adhere to the DRY (Don't Repeat Yourself) principle, ensuring clean and concise code throughout the implementation.

Negative Conditions

there is generally a consistent pattern where most conditions in the code are positive, except for one in MyGame class negative condition (!isEndGame). This approach ensures clarity and readability, making it easier to understand the flow of the game logic. By primarily utilizing positive conditions, the code maintains a more straightforward and intuitive structure.

Effective Java Items

I have applied several effective Java items to improve the codebase:

Item 2: Consider a builder when faced with many constructor parameters.

In the Card class, I encountered the need for two constructors, which could result in a proliferation of parameters. To address this, I implemented a builder pattern to provide a more readable and flexible approach to constructing Card objects.

Item 6: Avoid creating unnecessary objects.

In the Player class, I utilized the StringBuilder class instead of repeatedly creating new String objects. This approach minimizes unnecessary object creation, leading to improved performance and reduced memory usage.

Item 7: Eliminate obsolete object references.

Throughout the code, I ensured that I carefully manage object references and avoid holding onto objects that are no longer needed. This helps to prevent memory leaks and improves overall efficiency.

Item 8: Avoid finalizers and cleaners.

In adherence to best practices, I did not utilize any finalizers or cleaners in the code. These mechanisms are generally considered error-prone and are not necessary in the context of the Uno game engine.

Item 10: Obey the general contract when overriding equals.

Given the need to compare Card objects, I overrode the equals method in the Card class to ensure it follows the general contract specified in Java. This ensures proper equality comparisons and consistency when comparing Card objects.

Item 11: Always override hashCode when you override equals.

In conjunction with overriding the equals method, I also appropriately overrode the hashCode method in the Card class. This ensures consistency between hashCode and equals and is crucial when using Card objects in data structures such as hash-based collections.

Item 12: Always override toString.

In all classes within the game engine, I implemented the toString method to provide a meaningful and concise representation of the object's state. This aids in debugging and logging purposes.

Item 13: Override clone judiciously.

To avoid unnecessary complexity and potential issues, I consciously decided not to use or override the clone method in the codebase. This ensures simplicity and reduces the risk of unintended consequences related to cloning objects.

By adhering to these effective Java items, the Uno game engine benefits from improved code organization, performance optimizations, and adherence to established best practices.

SOLID Principles

Single Responsibility Principle

In the Uno game engine, each class has a single responsibility. For instance, the Card class is responsible for individual cards, while the UnoCards class handles the collection or set of Uno cards. This adherence to the SRP ensures that each class has a clear and focused purpose, making the codebase more maintainable and easier to understand.

However, you correctly pointed out that the Game class in the view has multiple responsibilities. While this may deviate from the SRP, it's important to consider the context of the Game class as the central component of the game engine. The Game class is responsible for integrating and coordinating various game components, which may involve handling game logic, player actions, and state management. Although it may encompass multiple responsibilities, this can be justified as part of its role as the orchestrator of the game engine.

In summary, while most classes in the Uno game engine adhere to the Single Responsibility Principle by having a single, well-defined responsibility, the Game class may have multiple responsibilities due to its central role in integrating game components.

Open Closed Principle

The principle is followed in many places through the use of interfaces and abstract classes. By employing these abstractions, you create a framework that allows for extensions and modifications through the implementation of new classes that adhere to the existing interfaces or inherit from abstract classes. This promotes the idea of "open for extension" while being "closed for modification."

However, you correctly pointed out that the Card class may not directly adhere to the OCP as it lacks an abstract or interface implementation. In the context of the Uno game, where the Card class represents a specific set of defined cards, it may not require modifications or extensions. Since the Card class is already well-defined and does not change at the game level, it doesn't violate the OCP. The principle is more relevant to components that are subject to changes or extensions in the future.

In summary, while the Uno game engine incorporates the Open-Closed Principle in many places through the use of interfaces and abstract classes, the Card class itself may not require such abstractions as it represents a fixed set of cards that are not intended to be modified or extended within the game.

Liskuv Substitution Principle

The principle is adhered to as there is no change in behavior when inheriting the superclass. The Game class serves as the superclass, while the MyGame class acts as the subclass. By following LSP, the MyGame class can be used as a substitute for the Game class without altering or modifying the expected behavior defined by the superclass.

This adherence to LSP ensures that objects of the subclass (MyGame) can seamlessly replace objects of the superclass (Game) without causing any unexpected or incorrect behavior. It promotes code reusability and maintainability by allowing the subclass to inherit and extend the functionality of the superclass without breaking the existing codebase.

In summary, the design of the Uno game engine aligns with the Liskov Substitution Principle, as the MyGame class maintains the same behavior as the superclass (Game) and can be used interchangeably without any adverse effects.

Interface Segregation Principle

The principle is followed as all the methods used in the interfaces or superclasses are essential for the correctness of the game logic. This adherence to ISP ensures that interfaces are focused and cohesive, containing only the methods that are relevant and necessary for specific roles or responsibilities.

By adhering to ISP, I avoid creating overly large and bloated interfaces that impose unnecessary dependencies on implementing classes. Instead, I design interfaces that are tailored to the specific needs of the classes that implement them, promoting better modularity and flexibility in the codebase.

Dependency Inversion Principle

The principle is adhered to as the classes depend on abstractions or interfaces rather than concrete implementations. This promotes loose coupling between classes and allows for flexibility and extensibility in the system.

For example, the Game class, State class, and Player class all depend on abstractions or interfaces instead of directly relying on low-level concrete classes. This allows for easy substitution of different implementations without impacting the functionality or behavior of the dependent classes.

By depending on abstractions, you can decouple the high-level modules from the low-level modules, resulting in a more modular and maintainable codebase. It also facilitates easier testing and promotes better separation of concerns.