Atypon Training
Uno Assignment

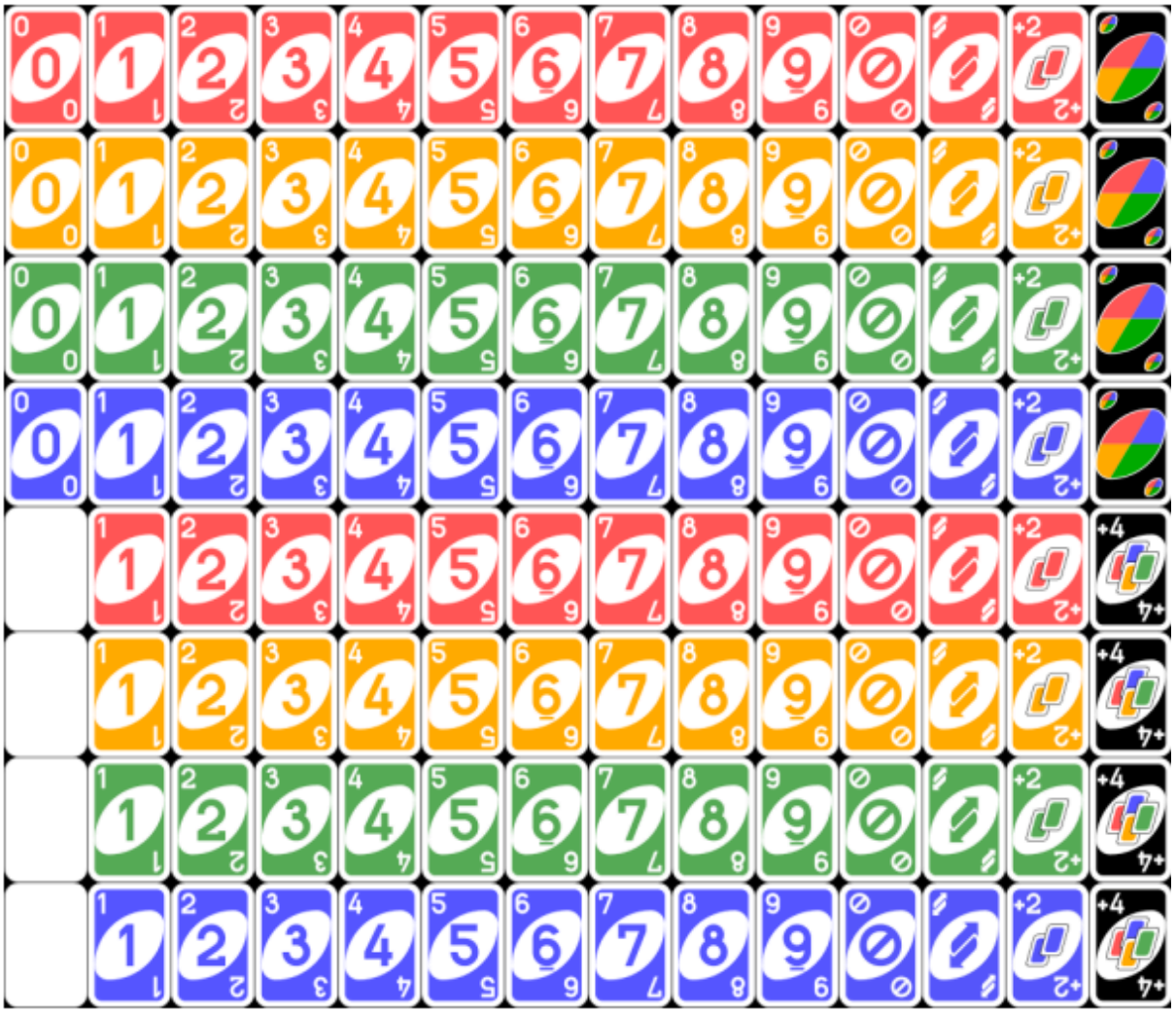
Saif Haitham Ali

Table of Contents

Objective

Build an Uno game engine to be used by other developers.

The problem Domain



- 2-10 players, and contains 108 cards divided as follows:
- Numbered cards (0-9)
- Action cards (Reverse, Skip, Draw Two)
- Wild cards (Wild, Wild Draw Fmy), or any type.
- Create a state model to map the game state of the game.
- Create a model to map the concept of “Cards”
- Actions need to be represented in an extendable way.
- Rules need to be implemented such that new ones can be created easily.

Design Choices

Immutability

In this section, I want to focus on the design choices that prioritize immutability in my UNO game engine implementation. Immutability, the concept of objects whose state cannot be modified after creation, plays a crucial role in simplifying the code, and maintaining code integrity.

1. Immutable Card Objects

One key aspect of my design is to make the Card objects immutable. Once created, a card's color and value should remain unchanged throughout the game. Immutability guarantees consistency and prevents accidental modifications that could result in incorrect gameplay. By treating Card objects as immutable, I can rely on their initial state without fear of unexpected changes.

2. Immutable Player State

Immutability also extends to the player's state in my game engine. Once a player's state, such as their hand of cards, is defined, it should not change during the game. By enforcing immutability, I maintain fairness and avoid inconsistencies or race conditions that could arise when multiple threads access and modify player states simultaneously. Immutable player state ensures that each player's cards remain intact and unaltered.

3. Immutable Game State

The overall game state, encompassing elements like the current player, the discard pile, and the draw pile, should also be designed with immutability in mind. Immutability of the game state ensures that it remains consistent throughout the gameplay, preventing unexpected modifications that could lead to erroneous game logic. Immutable game state simplifies reasoning about the game's behavior and facilitates reliable implementation.

4. Designing Immutable Classes

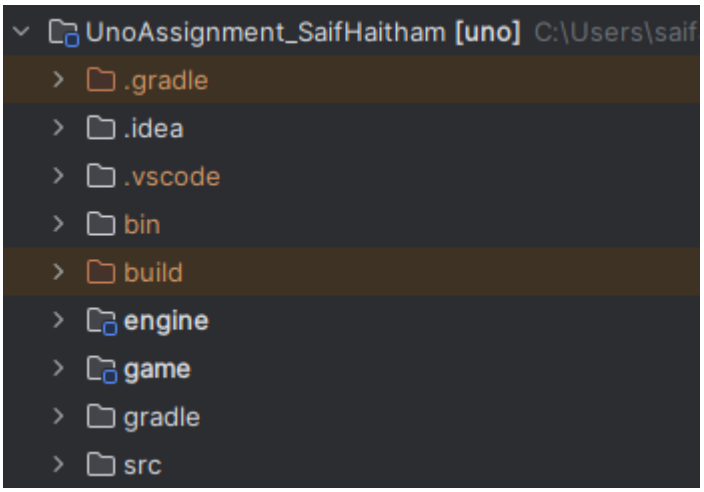
To achieve immutability, I follow specific design guidelines for my classes in Java. I declare classes as final, making them uninheritable, and mark fields as private and final to prevent external modification. I avoid providing setters and instead offer getters only when necessary to retrieve immutable data, also I tend to use records which are immutable data objects in Java. By adhering to these principles, I enforce immutability throughout my UNO game engine implementation.

5. Benefits and Trade-offs

Emphasizing immutability brings several benefits to my UNO game engine implementation. It enhances code reliability by preventing accidental modifications that could introduce bugs.

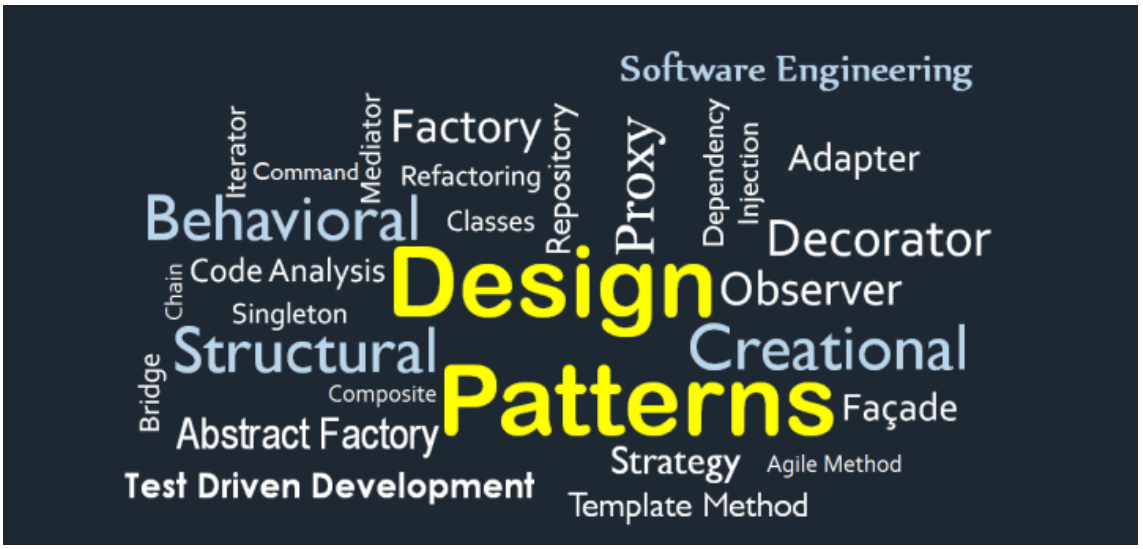
Immutability improves code maintainability, making it easier to reason about the behavior of objects. While immutability requires creating new objects for changes, the benefits it offers outweigh these trade-offs in most cases.

Federation



To ensure that I stay in the mentality of creating a library with the intention of it being used by other developers, I used the “Gradle” build manager to create a project “uno” and two sub-projects “engine” & “game”. Code in the “game” project is dependent on the code in the “engine” project thus making the development experience federated.

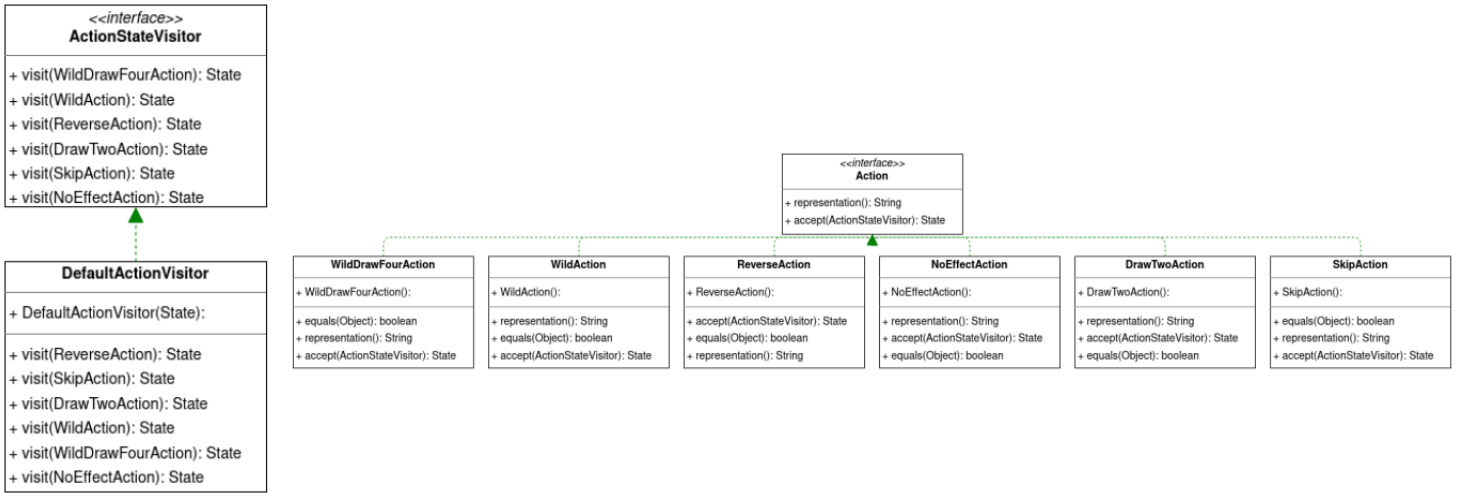
Design Patterns



Design patterns are reusable solutions to common software design problems that have been proven effective over time. They provide a structured approach to designing software systems, enabling developers to create flexible, maintainable, and extensible code. Design patterns encapsulate best practices and architectural principles, guiding the development process and promoting code reusability. By employing design patterns, developers can address recurring design challenges, enhance code organization, and improve system scalability. These patterns serve as a shared language among developers, facilitating communication and collaboration in software development teams.

My approach to design patterns was to code until I needed to abstract, for example, I had an “Action” interface, and classes that implement that interface, since Java doesn’t support multiple dispatch, I had to resort to using the visitor pattern and I ended up with the following hierarchy.

Visitor Design Pattern



Pipeline Design Pattern

Since I'm using immutable state, combining the builder pattern and the decorator design pattern, resulted in something I call the "Pipeline" pattern. Starting from an initial value, we execute steps to create a final value that is of the same type as the initial value.

```
final var bank = new Pipeline<List<Card>, List<Card>>()
        .add(new ActionCardsStep(Color.Red))
        .add(new ActionCardsStep(Color.Green))
        .add(new ActionCardsStep(Color.Blue))
        .add(new ActionCardsStep(Color.Yellow))
        .add(new ValueCardsStep(Color.Red))
        .add(new ValueCardsStep(Color.Green))
        .add(new ValueCardsStep(Color.Blue))
        .add(new ValueCardsStep(Color.Yellow))
        .add(new WildCardsStep())
        .run(new ArrayList<>());
```

With this pattern code reusability is at a maximum, this code excerpt shows the Pipeline pattern in action, "bank" after calling "run" will contain a "List" of the different shapes the user wants, for example the "ActionCardsStep" class looks like the following.

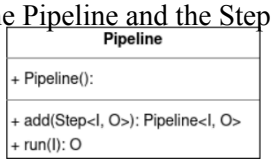
It merely takes the result of the previous step and adds to it, returning a new list after processing.

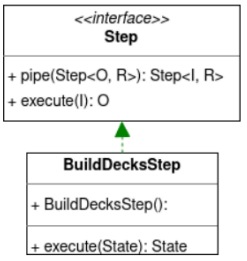
```
public record ActionCardsStep(Color color) implements Step<List<Card>, List<Card>> {  4 usages  saifalghost7

    @Override  3 usages  saifalghost7
    public List<Card> execute(List<Card> input) {

        for (int i = 0; i < 2; i++) {
            input.add(new Card(new ReverseAction(), color, new EmptyCardValue()));
            input.add(new Card(new SkipAction(), color, new EmptyCardValue()));
            input.add(new Card(new DrawTwoAction(), color, new EmptyCardValue()));
        }
        return input;
    }
}
```

```
var initialState = new Pipeline<State, State>()
        .add(new PickPlayerCountStep())
        .add(new BuildBankStep())
        .add(new BuildDecksStep())
        .add(new BuildPlayPileStep())
        .run(new State());
```

Here is the same pattern used to build the initial state of the game, each step modifies the state as necessary and after calling the run method, the initial state is formed.

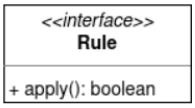
Here are the UML diagrams outlining the Pipeline and the Step interfaces.





Builder + Wrapper Design Patterns

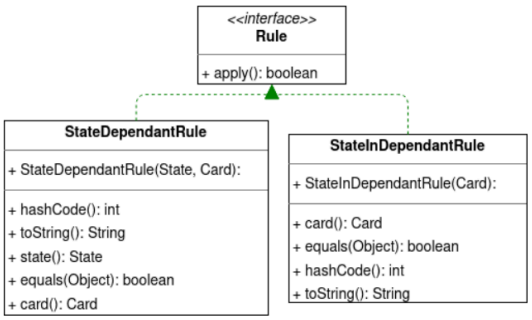
To build complex rules, I needed to combine two powerful design patterns and utilize them together to achieve a greater level of abstraction and code reuse.



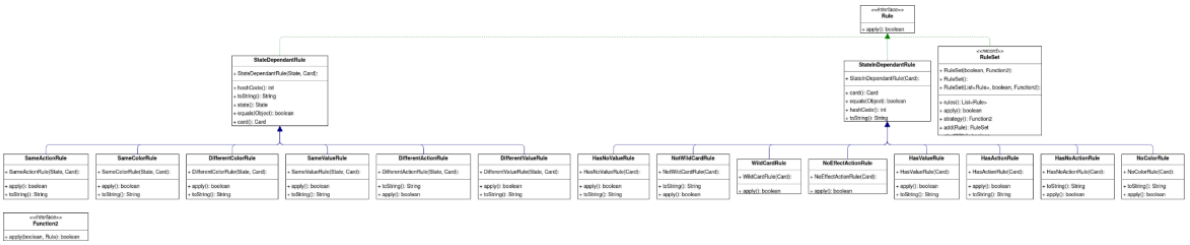
I first created a simple “Rule” interface, then implemented this interface two times, resulting in the “StateDependantRule” and “StateInDependantRule” abstract classes, the latter does not require access to the gamestate to check whether a move is valid or not while the former is passed an instance of the game state to check the the next move against.

For example, a class might implement the StateInDependant abstract interface and look like this.

```java
public final class HasValueRule extends StateInDependantRule {  10 usages  saifalghost7

    public HasValueRule(Card card) {  10 usages  saifalghost7
        super(card);
    }

    @Override  saifalghost7
    public boolean apply() {
        return !(card.value() instanceof EmptyCardValue);
    }

    @Override  saifalghost7
    public String toString() {
        return "HasValueRule";
    }
}
```



In order to augment this powerful pattern even more and have the ability to combine multiple rules, I implemented a third class from rule called “RuleSet”, the “RuleSet” itself is a rule, here is a code excerpt and the UML design diagram to aid in visualising.



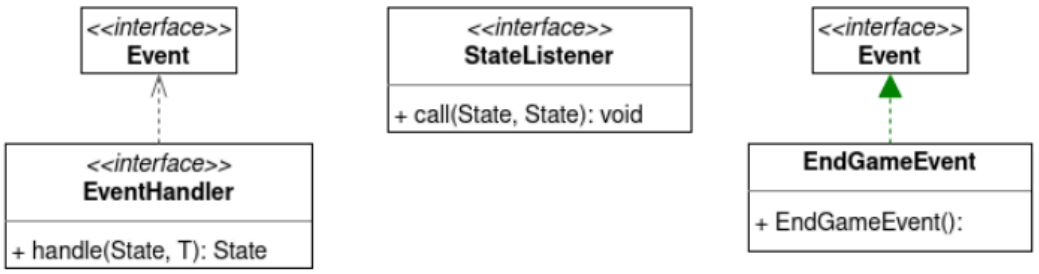
```
var ruleSet = new RuleSet( startWith: false, (p, c) -> c.apply() || p)
        .add(new RuleSet()
                .add(new HasValueRule(event.card()))
                .add(new HasValueRule(state.topOfPlayPile()))
                .add(new SameValueRule(state, event.card()))
                .add(new SameColorRule(state, event.card())))
        .add(new RuleSet()
                .add(new HasValueRule(event.card()))
                .add(new HasValueRule(state.topOfPlayPile()))
                .add(new DifferentValueRule(state, event.card()))
                .add(new SameColorRule(state, event.card())))
        .add(new RuleSet()
                .add(new HasValueRule(event.card()))
                .add(new HasValueRule(state.topOfPlayPile()))
                .add(new SameValueRule(state, event.card()))
                .add(new DifferentColorRule(state, event.card())))
        .add(new RuleSet()
                .add(new HasValueRule(event.card()))
                .add(new HasActionRule(state.topOfPlayPile()))
                .add(new SameColorRule(state, event.card())))
        .add(new RuleSet()
                .add(new HasValueRule(event.card()))
                .add(new WildCardRule(state.topOfPlayPile())))
        .add(new RuleSet()
                .add(new HasActionRule(event.card()))
                .add(new HasValueRule(state.topOfPlayPile()))
                .add(new SameColorRule(state, event.card())))
        .add(new RuleSet()
                .add(new HasActionRule(event.card()))
                .add(new WildCardRule(state.topOfPlayPile())))
        .add(new RuleSet()
                .add(new HasActionRule(event.card()))
                .add(new HasActionRule(state.topOfPlayPile()))
                .add(new SameActionRule(state, event.card()))
                .add(new SameColorRule(state, event.card())))
        .add(new RuleSet()
                .add(new HasActionRule(event.card()))
                .add(new HasActionRule(state.topOfPlayPile()))
                .add(new SameActionRule(state, event.card()))
                .add(new DifferentColorRule(state, event.card())))
        .add(new RuleSet()
                .add(new HasActionRule(event.card()))
                .add(new HasActionRule(state.topOfPlayPile()))
                .add(new DifferentActionRule(state, event.card()))
                .add(new SameColorRule(state, event.card())))
        .add(new RuleSet()
                .add(new WildCardRule(event.card())));
```

Observer Pattern

A powerful pattern that enables multiple places in the codebase to execute in response to an event, I utilized the pattern to do the following:
1. When an EndGameEvent is emitted, the handler that was registered to handle the event gets triggered, displaying a helpful message to the user.
2. When a CardChosenEvent is emitted, the handler that was registered to handle the event gets triggered, then the rules of the game are checked and the state is modified accordingly.
3. When a DrawFromPileEvent is emitted, the current player's deck gets a new card added to it.
4. On any state change the EventStoreListener is triggered and the state is re-rendered through the supplied Render implementation.

```
final EventStore eventStore = new EventStore()
        .registerHandler(DrawFromBankPileEvent.class, new DrawFromBankPileEventHandler())
        .registerHandler(CardChosenEvent.class, new CardChosenEventHandler())
        .registerHandler(EndGameEvent.class, new EndGameEventHandler())
        .registerListener(new EventStoreListener(renderer))
        .setState(initialState);
```

Here are a few UML diagrams to give a clearer view of the EventStore and its related objects and classes.



Clean Code & SOLID principles report

SOLID Principles

Single Responsibility Principle (SRP)

To achieve SRP I had to split the responsibility of each task in my implementation to simpler unit of execution, for example, EventHandler handles events, each event is triggered in a specific circumstance, the handler for each event type is delegated to only care about that specific event, this is only 1 case of many where my design takes SRP into consideration.

Open/Closed Principle (OCP)

I achieved OCP by using abstraction and inheritance to create a flexible and modular architecture. By relying on abstraction and design patterns, I can easily add new functionality without modifying existing code. This approach promotes code reusability, maintainability, and scalability while minimizing the risk of introducing bugs.

Liskov Substitution Principle (LSP)

In my implementation, I have designed the subclasses to honor the contracts and behaviors defined by their base class. Any method or property defined in the base class can be invoked or accessed on the subclass without causing unexpected errors or violating the expected behavior. This enables seamless interchangeability between objects of different subclasses and promotes code reuse.

By following LSP, I have created a hierarchy of classes where each subclass adheres to the behavior and functionality defined by the base class. This ensures that client code can rely on the base class interface and work with any subclass without having to make modifications or adjustments. The LSP contributes to the overall design flexibility and extensibility of the codebase, allowing for easy addition of new subclasses and promoting the principles of abstraction and encapsulation.

Interface Segregation Principle (ISP)

To adhere to the Interface Segregation Principle (ISP), I have designed interfaces that are focused and tailored to the specific requirements of the clients that use them. The principle emphasizes that clients should not be forced to depend on interfaces they do not use or need.

In my implementation, I have created fine-grained interfaces that only expose the methods and properties relevant to the clients that utilize them. By doing so, I avoid bloating interfaces with unnecessary methods that can lead to tight coupling and code dependencies.

Instead of having a single large interface that encompasses all possible functionalities, I have divided the responsibilities into smaller, more specialized interfaces. Each interface represents a

cohesive set of operations related to a specific aspect of the system. This allows clients to depend only on the interfaces that are relevant to their needs, reducing unnecessary coupling and promoting code modularity.

By adhering to ISP, I ensure that changes made to one interface do not affect the clients that depend on other interfaces. This promotes loose coupling and increases the flexibility of the system, allowing for easier maintenance and extension.

Dependency Inversion Principle (DIP)

When working on the RuleSet class, I wanted the comparison strategy to be user defined, to achieve that I allowed the user to pass the method a “lambda” function allowing them to control the behavior as they desire, thus achieving an important goal of the dependency inversion principle.

Clean Code

Readability

Throughout my codebase I tried to maintain a consistent naming scheme, I also tried as much as possible to make variable names meaningful, I also tried to make functions have the least amount of functional and easily read code, I also followed the 80 character limit rule.

Simplicity

My code is straightforward, avoiding unnecessary complexity and duplication. It follows the "Don't Repeat Yourself" (DRY) principle, promoting code reuse through abstraction and modularization. Simple code is easier to maintain, debug, and extend.

Maintainability

Maintainability is a key focus in my implementation. I prioritize writing clean and readable code, using meaningful names and following consistent formatting. I emphasize modularity to isolate and modify specific parts of the system without affecting the whole codebase. Documentation and thorough testing also contribute to maintainability. By focusing on maintainability, I ensure the code is easier to understand, modify, and extend, enabling efficient development and long-term success.

Efficiency

Efficiency is a crucial consideration in my implementation. I strive to optimize performance by minimizing unnecessary computations, utilizing efficient algorithms and data structures, and minimizing resource consumption. This ensures that the software operates smoothly and delivers optimal performance. By writing efficient code, I enhance the overall effectiveness and responsiveness of the system, providing a seamless user experience and maximizing resource utilization.