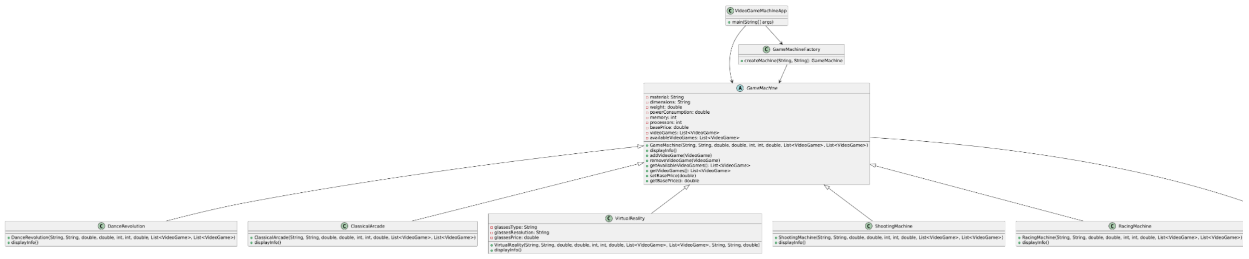
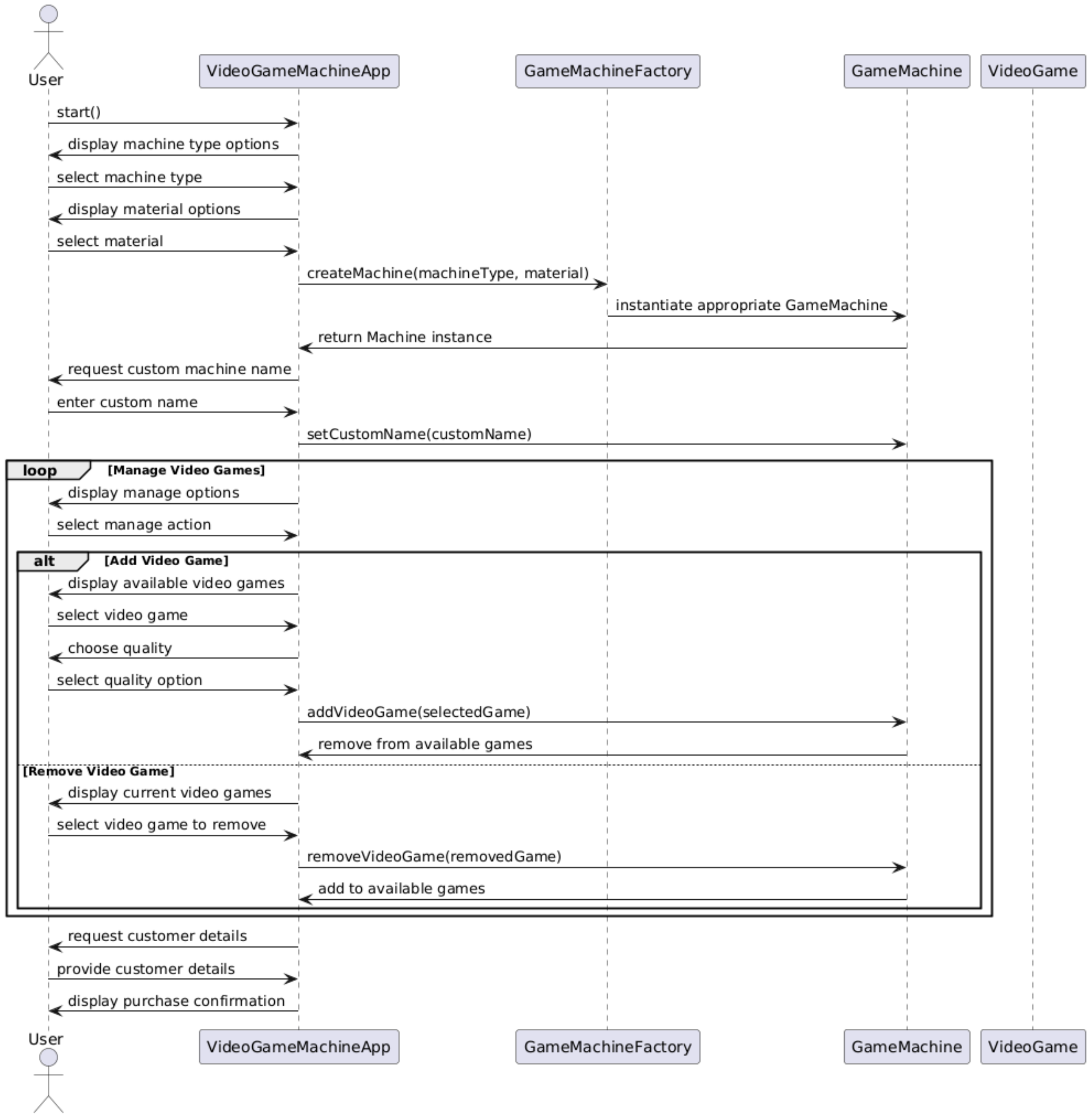
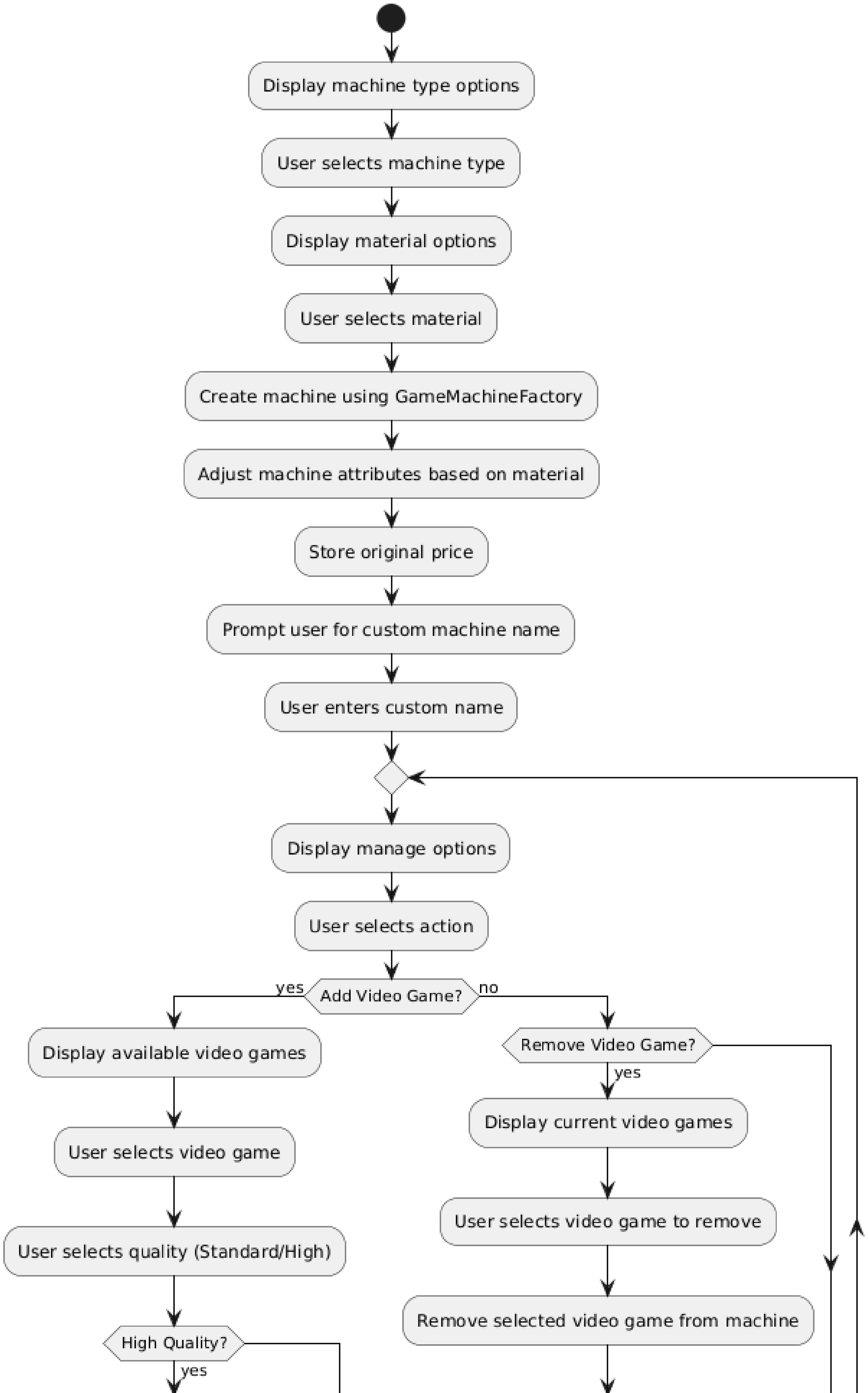
Class diagram:


VideoGameMachineApp
main(String[] args)
GameMachineFactory
createMachine(String, String): GameMachine
GameMachine
material: String
dimensions: String
weight: double
powerConsumption: double
memory: int
processors: int
basePrice: double
videoGames: List<VideoGame>
availableVideoGames: List<VideoGame>
GameMachine(String, String, double, double, int, int, double, List<VideoGame>, List<VideoGame>)
displayInfo()
addVideoGame(VideoGame)
removeVideoGame(VideoGame)
getAvailableVideoGames(): List<VideoGame>
getVideoGames(): List<VideoGame>
setBasePrice(double)
getBasePrice(): double
DanceRevolution
DanceRevolution(String, String, double, double, int, int, double, List<VideoGame>, List<VideoGame>)
displayInfo()
ClassicalArcade
ClassicalArcade(String, String, double, double, int, int, double, List<VideoGame>, List<VideoGame>)
displayInfo()
VirtualReality
glassesType: String
glassesResolution: String
glassesPrice: double
VirtualReality(String, String, double, double, int, int, double, List<VideoGame>, List<VideoGame>, String, String, double)
displayInfo()
ShootingMachine
ShootingMachine(String, String, double, double, int, int, double, List<VideoGame>, List<VideoGame>)
displayInfo()
RacingMachine
RacingMachine(String, String, double, double, int, int, double, List<VideoGame>, List<VideoGame>)
displayInfo()

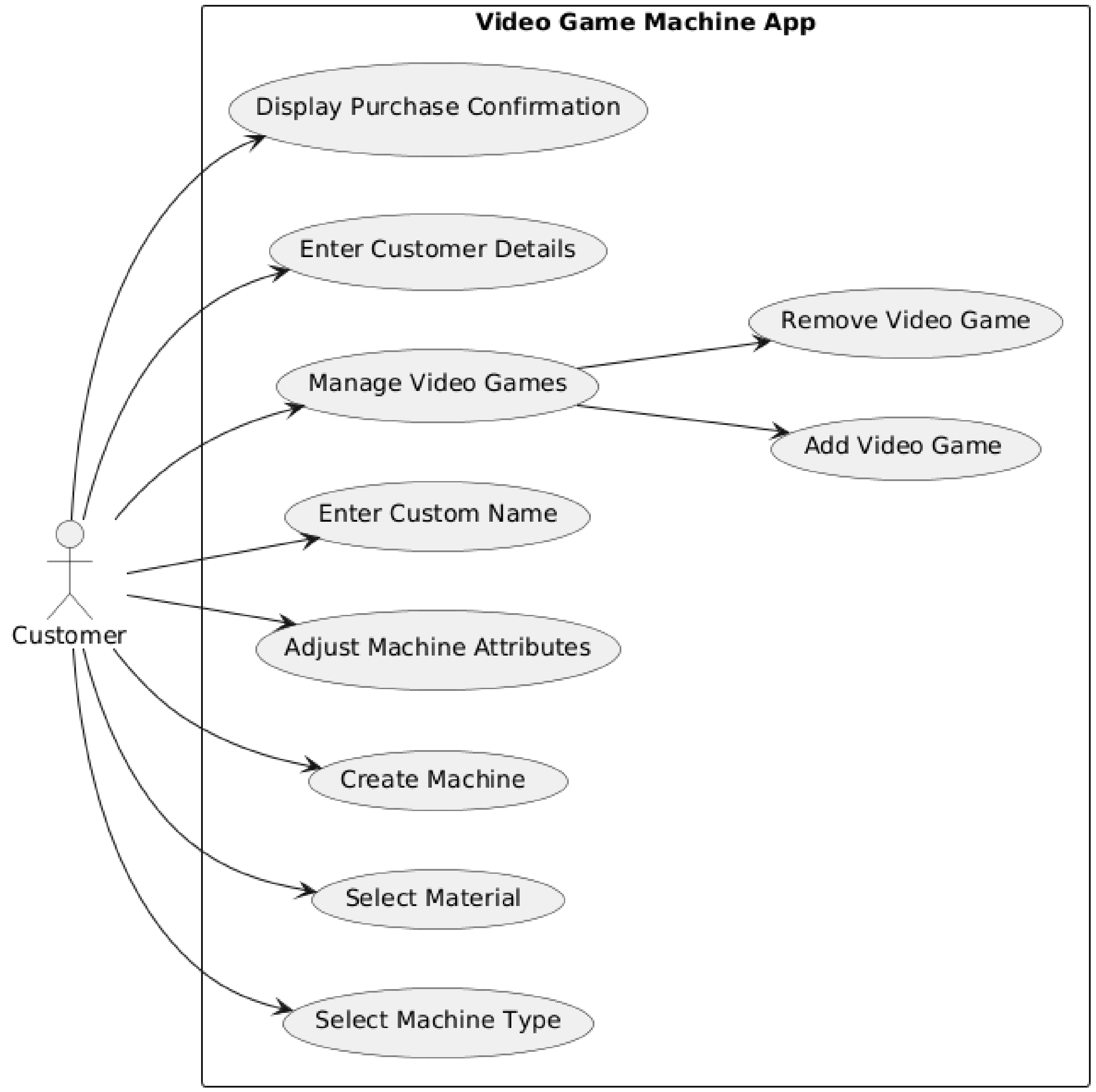
sequence diagram:



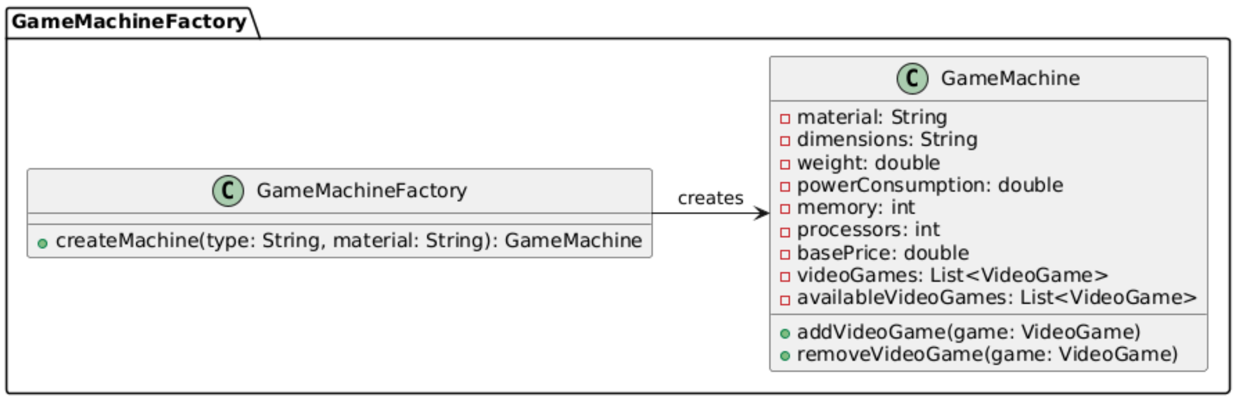
Flow diagram:


Display machine type options
User selects machine type
Display material options
User selects material
Create machine using GameMachineFactory
Adjust machine attributes based on material
Store original price
Prompt user for custom machine name
User enters custom name
Display manage options
User selects action
Add Video Game?
yes
no
Display available video games
User selects video game
User selects quality (Standard/High)
High Quality?
yes
Remove Video Game?
yes
Display current video games
User selects video game to remove
Remove selected video game from machine

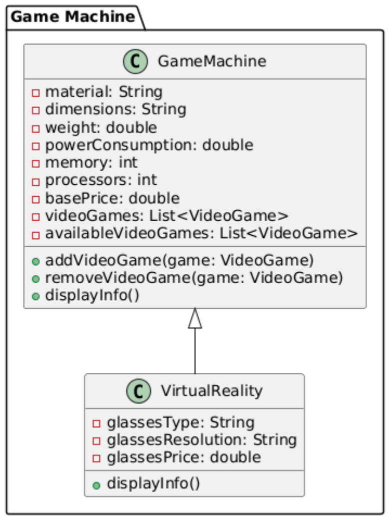
use case diagram:



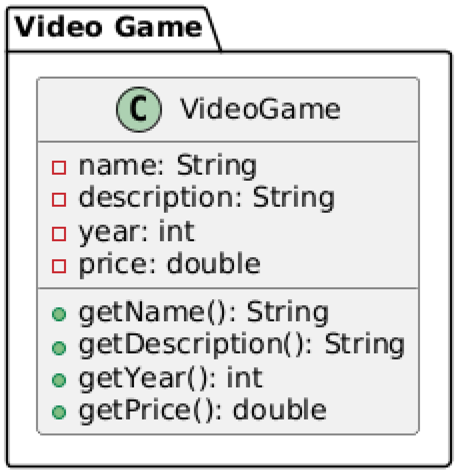
Technical Report:
Components:
GameMachineFactory Component:


GameMachineFactory
GameMachineFactory
+ createMachine(type: String, material: String): GameMachine
creates
GameMachine
- material: String
- dimensions: String
- weight: double
- powerConsumption: double
- memory: int
- processors: int
- basePrice: double
- videoGames: List<VideoGame>
- availableVideoGames: List<VideoGame>
+ addVideoGame(game: VideoGame)
+ removeVideoGame(game: VideoGame)

GameMachine Component:



VideoGame Component:



Factory Design Pattern used

Goal:The Factory design pattern is used to create objects without specifying the exact class of the object that will be created. This promotes loose coupling and enhances flexibility in the code.

Implementation:

In the application, the GameMachineFactory class is responsible for creating instances of various types of game machines (Dance Revolution, Classical Arcade, Virtual Reality, etc.). The factory encapsulates the logic for instantiating these classes, allowing the main application logic to remain clean and focused on higher-level functionality.
By utilizing the Factory Pattern, the object creation logic is encapsulated within the factory. The client (in this case, the VideoGameMachineApp class) does not need to know the details of how each machine type is constructed. This encapsulation allows for changes in the creation logic without affecting the client code, adhering to the Open/Closed Principle.

Not used:
Singleton Pattern
DescriptionThe Singleton Pattern ensures that a class has only one instance and provides a global point of access to it.
The application does not have a requirement for a single instance of any class. Each game machine is created independently, and there is no global state that needs to be shared across the application.

Solid Principles:
1. Single Responsibility Principle (SRP):Each class in the application, such as VideoGame, GameMachine, and its subclasses DanceRevolution, VirtualReality), encapsulates its specific functionality and attributes. For example, VideoGame manages video game properties like name, price, description, and year, while GameMachine handles attributes and methods related to game machines.
2.Open/Closed Principle (OCP):The GameMachine class serves as a base class for various types of machines (like DanceRevolution, VirtualReality, etc.). New machine types can be added by creating new subclasses without modifying existing code, thereby adhering to the OCP. This allows for future extensions without altering the existing functionality.
3.Liskov Substitution Principle (LSP) All subclasses of GameMachine can be used interchangeably wherever a GameMachine type is expected. For instance, whether a DanceRevolution or a VirtualReality machine is instantiated, the application can treat them as GameMachine objects, ensuring that they adhere to the same interface and expected behavior.
4.Dependency Inversion Principle (DIP): The app uses the GameMachineFactory to create instances of game machines, which decouples the instantiation logic from the business logic in the VideoGameMachineApp. This means that the app does not directly depend on concrete implementations of game machines, aligning with the DIP. The factory abstracts the creation process, promoting a more flexible design.