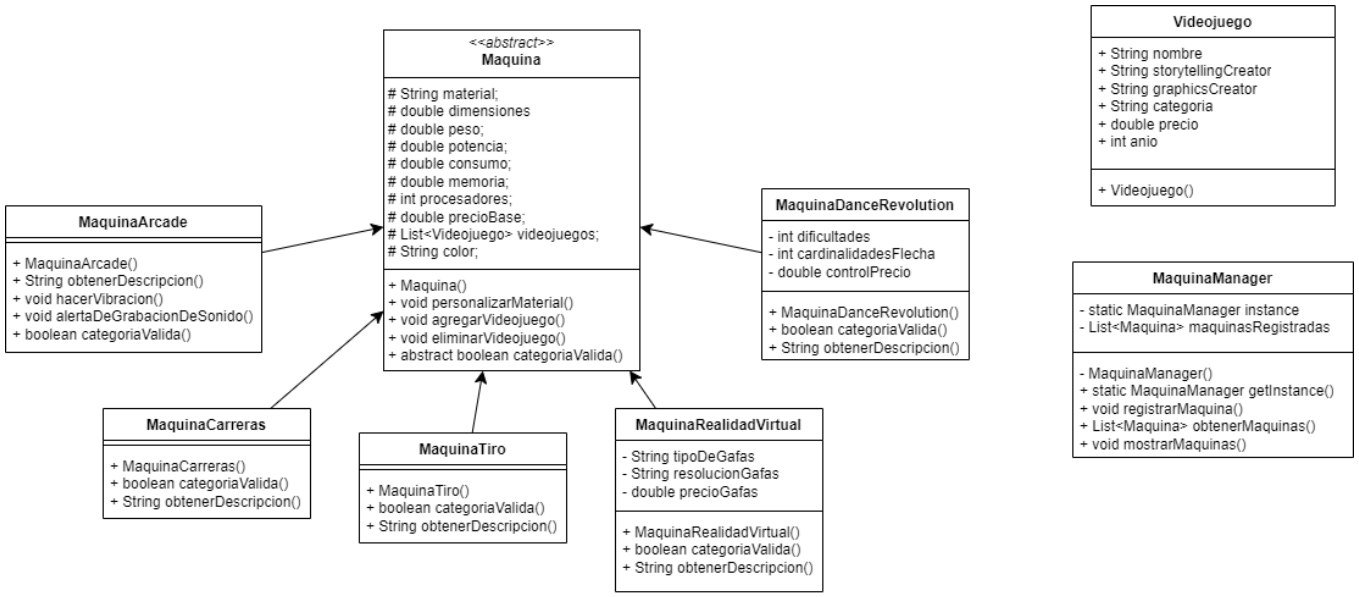
Name: Jaider Santiago Avila Robles - 20231020200

Technical Report for the Arcade Machines System

Introduction

This report details the technical solution for creating a management system for arcade machines and video games. Various design techniques and patterns have been used to ensure a robust, flexible, and easy-to-maintain codebase. The report includes class diagrams and component sub-diagrams that help explain the proposed architecture, along with the technical concerns and decisions that guided the development.

General Class Diagram



Design Patterns Used

Singleton Pattern

The Singleton pattern has been applied to the `MachineManager` class. This pattern ensures that there is a single instance of the class responsible for managing the machines registered in the system. This design is appropriate here because machine management must be centralized to avoid inconsistencies.

Motivation:

- There must be only one instance managing the list of machines throughout the program.

- It ensures that all registered machines are accessible from a single access point.

Technical Details:

The `MachineManager` class is private and does not allow additional instances outside of the class itself. The static method `getInstance()` is implemented to return the single instance of the class.

Liskov Substitution Principle (LSP) and Open/Closed Principle (OCP)

Both the Liskov and OCP principles were followed when creating an abstract `Machine` class, which is extended by specific subclasses like `ArcadeMachine`, `DanceRevolutionMachine`, etc. These subclasses extend the base class's behavior without modifying it. Each subclass can implement its own validation rules and add unique behaviors such as vibration functions or virtual reality headset control.

Motivation:

- Ensure that subclasses can replace the base class without altering the general behavior of the system (LSP).

- Make the code extensible so more types of machines can be added without changing the existing code (OCP).

Technical Details:

Each subclass implements its own logic for category validation (`isValidCategory()`) and additional methods like vibration or headset control:

Single Responsibility Principle (SRP)

Each class in the system has a single responsibility. The `VideoGame` class exclusively represents a video game, while the `Machine` class manages the properties of a machine. The `Main` class only handles user interaction and program flow.

Motivation:

- Ensure that each class has a clear and defined purpose, facilitating code comprehension, maintenance, and extension.

Technical Details:

For example, the `VideoGame` class solely handles storing information related to video games:

Dependency Inversion Principle (DIP)

The `Main` class, which handles the program's flow and user interaction, does not depend on specific machine classes. Instead, it uses the abstract `Machine` class and its subclasses, allowing for high flexibility and extensibility.

Motivation:

- Facilitate easy modification or expansion of the code without having to change the core behavior of the program.

Technical Details:

The method that allows pre-configured machines to be added to the menu does not depend on concrete implementations but on the abstract ‘Machine’ class:

Unused Patterns

In this solution, patterns like Factory or Builder were not deemed necessary, as the creation of pre-configured machines follows a fairly simple process that does not require the additional complexity of these patterns.

Memory Optimization

In this implementation, we have aimed to reduce unnecessary memory usage by:

- Unique references: Avoiding the creation of unnecessary duplicate objects.

- Efficient video game management: Video games are only added to machines if their category is valid, preventing additional processing.

Conclusion

The design and implementation of this system closely follow SOLID principles and use the Singleton pattern to centralize machine management. The architecture is easily extensible, allowing new types of machines or functionalities to be added without modifying the existing code.