Software Modeling I

Season 2024-III

Report

Workshop No. 2

Anderson David Arenas Gutierez – 20231020030

Project Overview

The Arcade Machine Management System is designed to streamline the management of arcade machines and video games in an arcade setting. The system provides functionalities for user management, machine creation, game management, and purchase tracking, enhancing the operational efficiency of arcade businesses. The project aims to build a robust application that caters to different user roles, allowing administrators to manage machines and games while enabling clients to explore and purchase available machines.

Objectives

1. To develop a user-friendly application that facilitates the management of arcade machines and video games.
2. To implement a system that supports multiple user roles (e.g., Client, Manager), each with specific functionalities.
3. To enable the creation of various arcade machines through a factory method, allowing for easy extensibility and customization.
4. To maintain detailed records of video games, including descriptions, categories, and pricing.

Object-Oriented Principles Analysis for Arcade Machine Management System

The architecture of the arcade machine management system leverages several object-oriented principles and design patterns to ensure scalability, maintainability, and adherence to SOLID principles. Below is an analysis of the key technical concerns and design decisions made during the development of this application.

SOLID Principles Implementation

S - Single Responsibility Principle (SRP)

Each class in the system has a single responsibility:

1. **User** class: Manages user details and behaviors (e.g., Client and Manager classes).

2. **VideoGame** class: Encapsulates properties and methods related to video games.
3. **FactoryMachines** and **PredefinedMachines** :Separate responsibilities for creating different types of machines.

O - Open/Closed Principle (OCP)
The design supports extension without modification:

1. New arcade machine types can be added by creating new subclasses of Machine without changing the existing code.
2. The factory method create_machine can be expanded to accommodate additional categories.

L - Liskov Substitution Principle (LSP)
Derived classes can be substituted for their base classes without altering the functionality:

1. The Client and Manager classes inherit from the User abstract class. Any method expecting a User can work with either a Client or Manager.

I - Interface Segregation Principle (ISP)
Interfaces are specific to the needs of clients:

1. The abstract FactoryMachines class defines the interface for creating machines withouten forcing unnecessary methods on implementing classes.

D - Dependency Inversion Principle (DIP)
Higher-level modules do not depend on lower-level modules but on abstractions:

1. The system uses abstract classes (User, FactoryMachines) to decouple the instantiation of objects from the business logic, allowing for easier testing and modifications.

Design Patterns Used

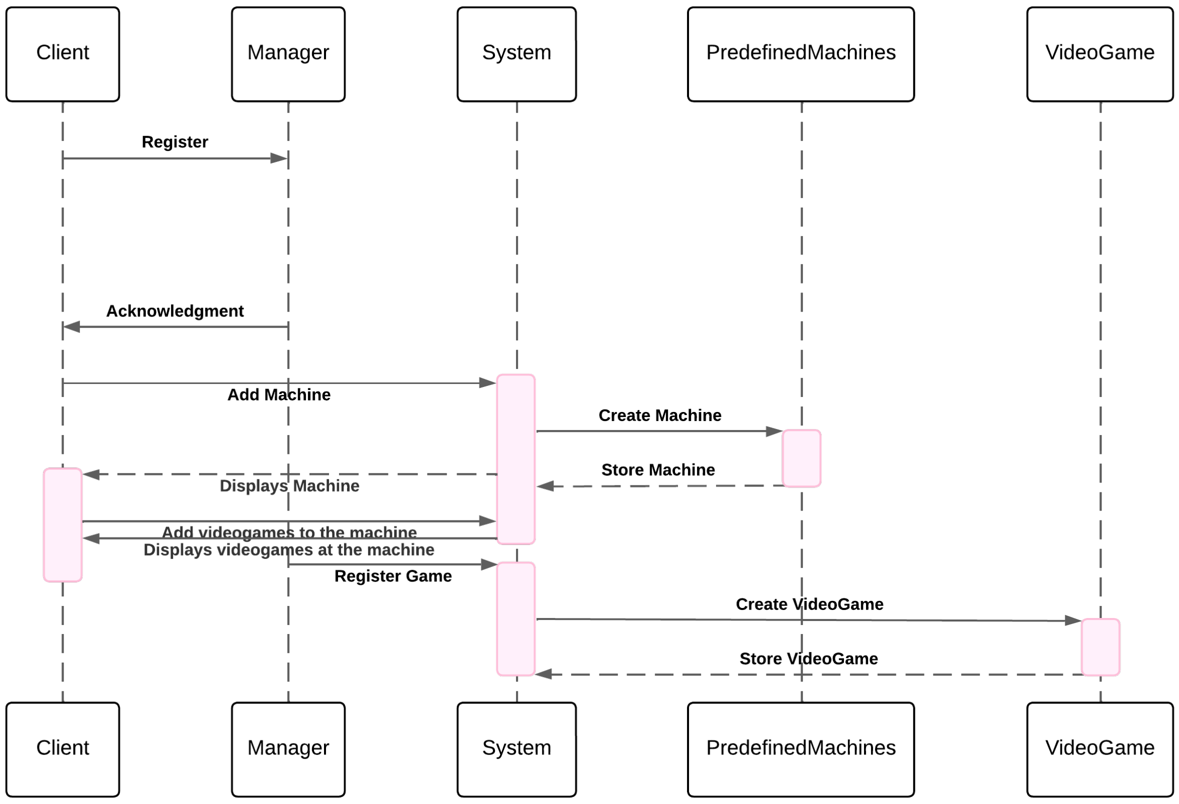
Factory Method Pattern

1. The FactoryMachines abstract class and its PredefinedMachines subclass utilize the Factory Method pattern to create different types of arcade machines. This encapsulates the creation logic, allowing the code to be more flexible and maintainable.

Technical Concerns and Decisions

1. **Encapsulation** : Attributes of classes are kept private with access methods (getters and setters) to maintain control over how the data is accessed and modified. This prevents unauthorized access and keeps the data consistent.
2. **Inheritance and Composition** : The design leverages inheritance (e.g., Client and Manager inheriting from User) and composition (e.g., Client containing a list of Address instances) to create flexible and reusable code structures.
3. **Abstraction** : Abstract classes are utilized to provide a common interface for derived classes. This is evident in the User and FactoryMachines classes, which allow for polymorphic behavior in the system.
4. **Error Handling** : The application anticipates errors, such as invalid categories in machine creation. Custom exceptions can be introduced for better error handling and user feedback.

Sequence diagram



The sequence diagram for the Arcade Machine Management System effectively captures the interactions between different user roles and system components. It illustrates:

1. The flow of requests and responses during user authentication, game viewing, and purchasing processes.
2. How the system processes manager requests for machine creation.
3. The management of purchase history.

This representation helps identify how the application architecture supports user interactions and ensures a seamless experience for both clients and managers. Additionally, the sequence diagram highlights the importance of clear communication between components, adhering to object-oriented principles and promoting maintainability within the system.

Components of the program

User Management Component

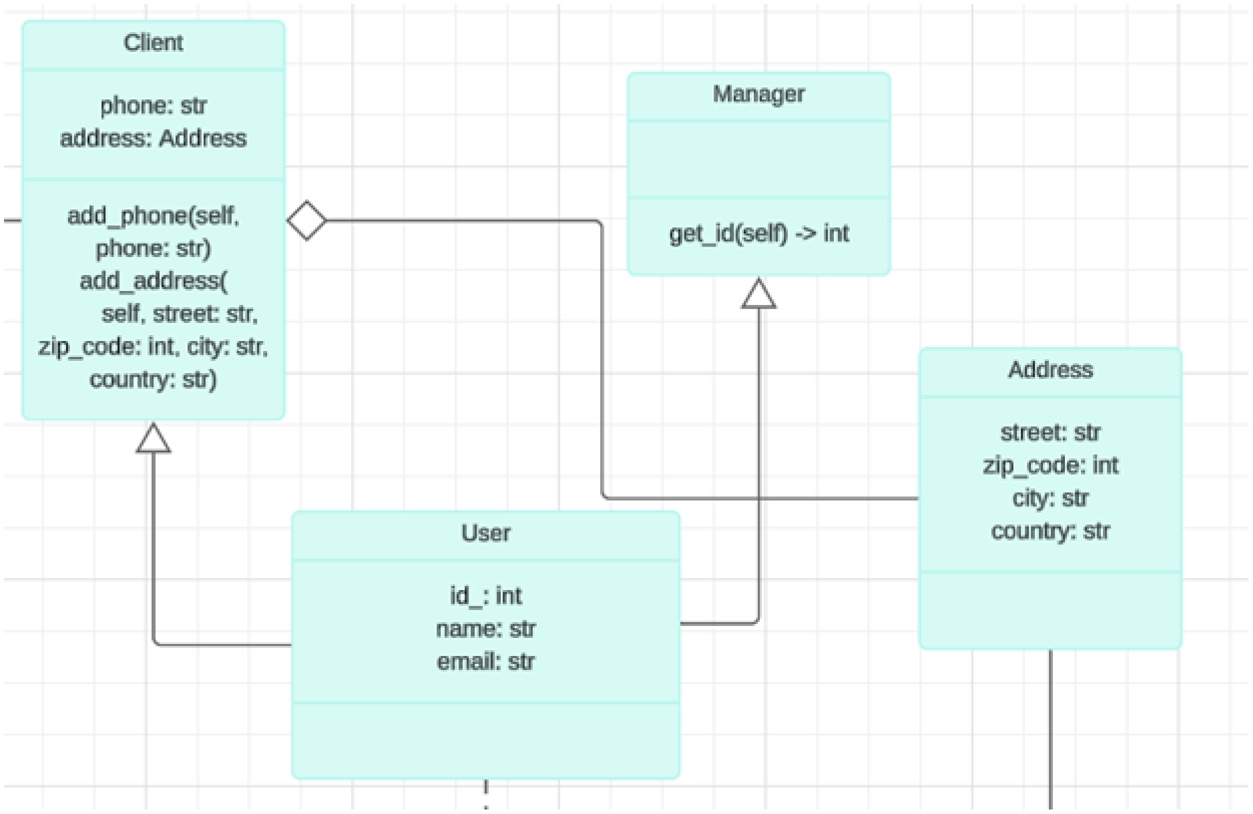
- Classes: User,Client, Manager

- Responsibilities:

1. This component handles user-related functionalities such as registration, authentication, and profile management.

2. The User class serves as an abstract base class that provides common properties (like id, name, and email) for both Client and Manager.

3. The Client class represents the end-user, providing methods for managing personal details (like phone numbers and addresses).

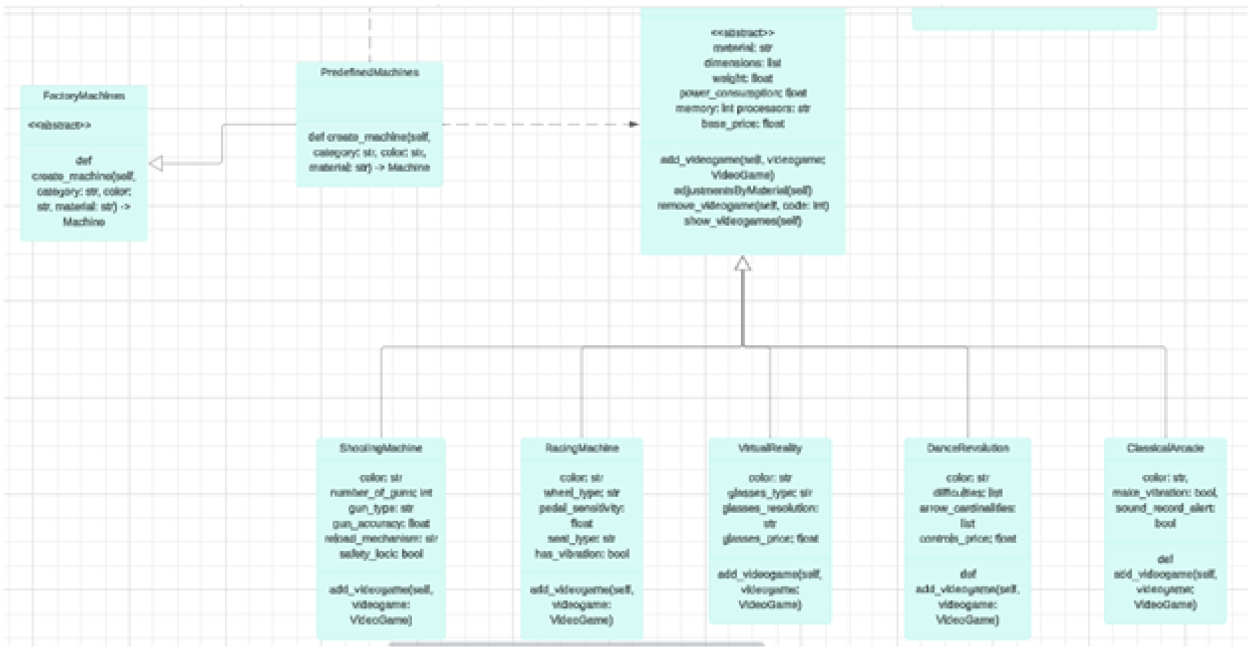


Arca deMachi ne Factory Compo nent

- Classes : FactoryMachines, Predefine dMachine s, Machine

- Responsibi lities:

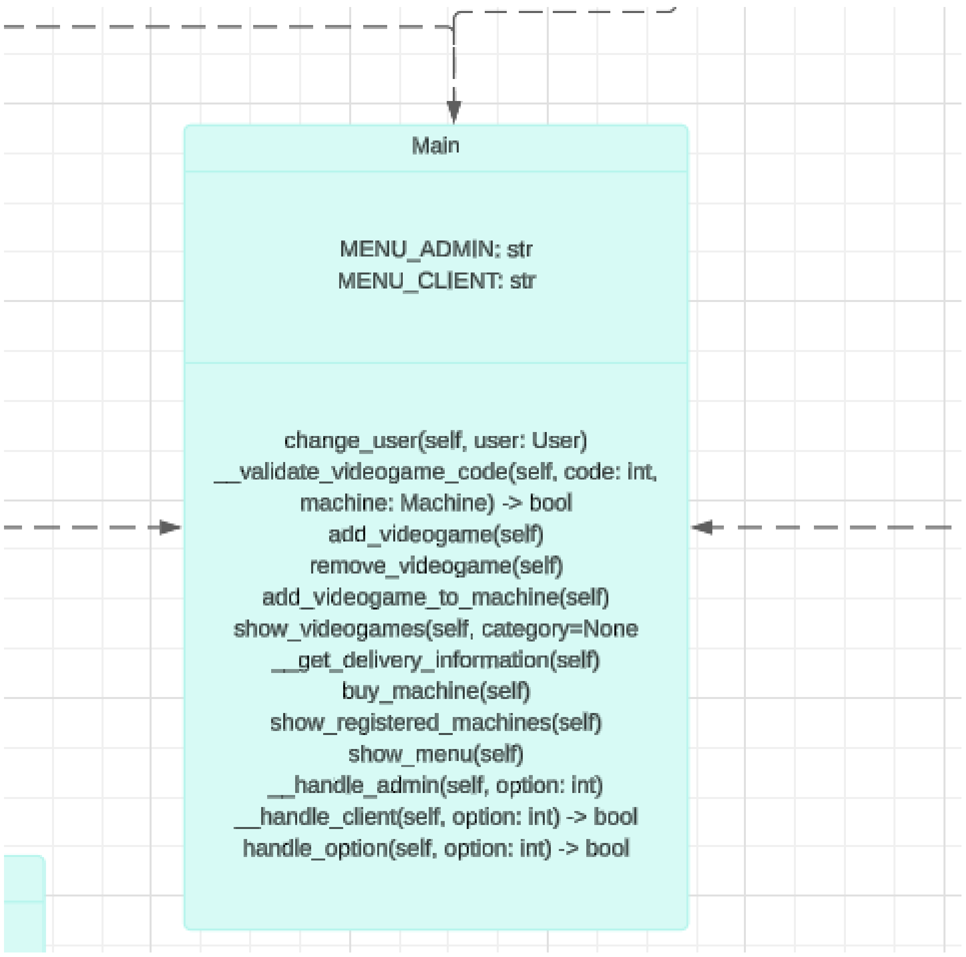
1. This compon ent follows the Factory Desig n Pattern to crea te diff erent types of arcade machines.
2. The FactoryMachin es abstract class defines the interface for mach ine creation ,wh ile the Predefin edMachine s class implemen ts this interface to create instan ces of sp ecific machine types (e.g., Dance Revolution, Shootin gMachine ,Raci ngMachin e, e tc.).
3. This allows for easy expansio n of machine types withou tal te ring the existing code structure.



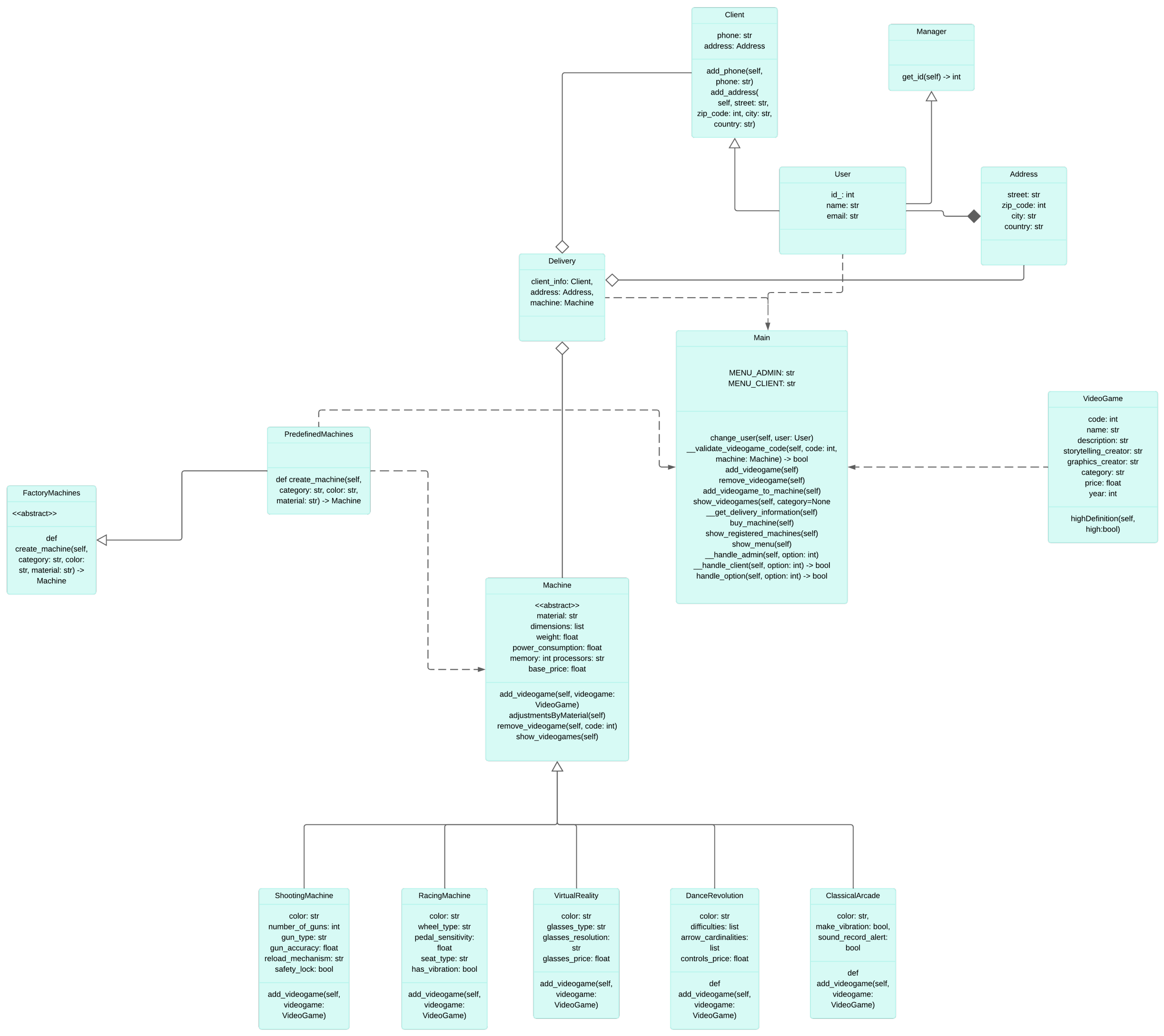
Main Appl ication Component

- Responsibi lities:

1. This compon ent serves as the entry point for users to interact with the system, typically through a command-li ne interface (CLI).
2. It manages the flow of control ,al lowing use rs to log in, view avail able game s, make purch ases, and manage their profiles.
3. This compon ent orchestrates interaction s among the other compon ents based on user input.


Main
MENU_ADMIN: str
MENU_CLIENT: str
change_user(self, user: User)
__validate_videogame_code(self, code: int, machine: Machine) -> bool
add_videogame(self)
remove_videogame(self)
add_videogame_to_machine(self)
show_videogames(self, category=None
__get_delivery_information(self)
buy_machine(self)
show_registered_machines(self)
show_menu(self)
__handle_admin(self, option: int)
__handle_client(self, option: int) -> bool
handle_option(self, option: int) -> bool

Class Diagra m


Client
phone: str
address: Address
add_phone(self, phone: str)
add_address(self, street: str, zip_code: int, city: str, country: str)
Manager
get_id(self) -> int
User
id_: int
name: str
email: str
Address
street: str
zip_code: int
city: str
country: str
Delivery
client_info: Client,
address: Address,
machine: Machine
Main
MENU_ADMIN: str
MENU_CLIENT: str
change_user(self, user: User)
__validate_videogame_code(self, code: int, machine: Machine) -> bool
add_videogame(self)
remove_videogame(self)
add_videogame_to_machine(self)
show_videogames(self, category=None
__get_delivery_information(self)
buy_machine(self)
show_registered_machines(self)
show_menu(self)
__handle_admin(self, option: int)
__handle_client(self, option: int) -> bool
handle_option(self, option: int) -> bool
VideoGame
code: int
name: str
description: str
storytelling_creator: str
graphics_creator: str
category: str
price: float
year: int
highDefinition(self, high:bool)
PredefinedMachines
def create_machine(self, category: str, color: str, material: str) -> Machine
FactoryMachines
<<abstract>>
def create_machine(self, category: str, color: str, material: str) -> Machine
Machine
<<abstract>>
material: str
dimensions: list
weight: float
power_consumption: float
memory: int processors: str
base_price: float
add_videogame(self, videogame: VideoGame)
adjustmentsByMaterial(self)
remove_videogame(self, code: int)
show_videogames(self)
ShootingMachine
color: str
number_of_guns: int
gun_type: str
gun_accuracy: float
reload_mechanism: str
safety_lock: bool
add_videogame(self, videogame: VideoGame)
RacingMachine
color: str
wheel_type: str
pedal_sensitivity: float
seat_type: str
has_vibration: bool
add_videogame(self, videogame: VideoGame)
VirtualReality
color: str
glasses_type: str
glasses_resolution: str
glasses_price: float
add_videogame(self, videogame: VideoGame)
DanceRevolution
color: str
difficulties: list
arrow_cardinalities: list
controls_price: float
def add_videogame(self, videogame: VideoGame)
ClassicalArcade
color: str,
make_vibration: bool,
sound_record_alert: bool
def add_videogame(self, videogame: VideoGame)