Workshop 2

Joan Sebastian Duran Pradilla
Maiker Alejandro Hernandez Archila
University Distrital
Email: jsduranp@udistrital.edu.co mahernandeza@udistrital.edu.co

***Abstract*—This report details the operation of the program, principles and improvements that have been implemented to date.**

I. System Functionality

The system simulates the purchase of an arcade machine. It is structured into several classes that represent different components of the purchase process. Each component is responsible for managing specific aspects, allowing for improved modularity and maintainability of the code.

II. Class Diagram

The class diagram consists of the following main classes:

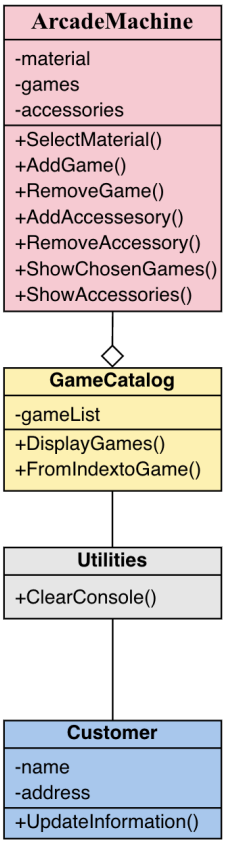

Fig. 1. Class Diagram

A. ArcadeMachine

- **Responsibilities:**
  - Manage the material of the arcade machine.
  - Handle the list of selected games.
  - Manage the list of additional accessories.
- **Methods:**
  - `SelectMaterial()`, `AddGame()`, `RemoveGame()`, `AddAccessory()`, `RemoveAccessory()`, `ShowChosenGames()`, `ShowAccessories()`

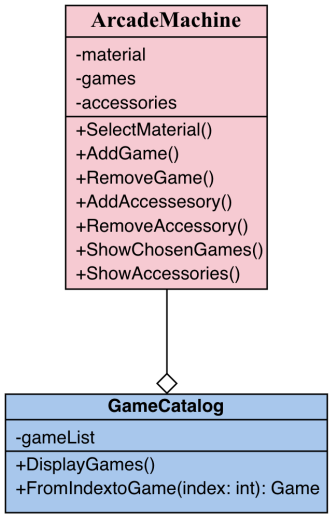

Fig. 2. ArcadeMachine

B. GameCatalog

- **Responsibilities:**
  - Store and manage the list of available games.
  - Provide methods to display games and retrieve a game based on its index.
- **Methods:**
  - `DisplayGames()`, `FromIndextoGame()`

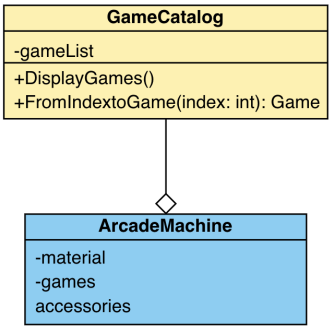

Fig. 3. GameCatalog

C. Utilities

- **Responsibilities:**
  - Provide utility methods, such as clearing the console.
- **Methods:**
  - `ClearConsole()`

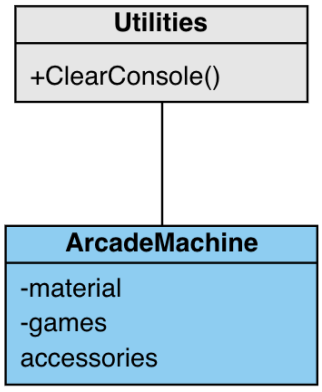

Fig. 4. Utilities

D. Customer

- **Responsibilities:**
  - Manage customer information, such as name and address.
- **Methods:**
  - `UpdateInformation()`

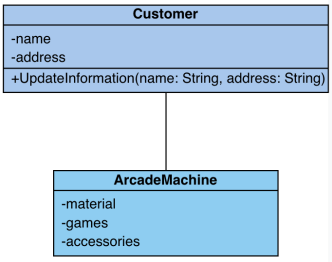

Fig. 5. Customer

III. Component Details and Relationships

A. ArcadeMachine

- **Attributes:**
  - `material`: Type of material for the machine.
  - `games`: List of selected games.
  - `accessories`: List of additional accessories.
- **Relationships:**
  - Aggregation with `GameCatalog` (can contain games).
  - Association with `Utilities` and `Customer`.

B. GameCatalog

- **Attributes:**
  - `gamesList`: List of available games.
- **Methods:**
  - Allows `ArcadeMachine` to access available games through display and selection.

C. Utilities

- **Methods:**
  - Provides auxiliary functions that can be utilized by several classes, promoting code reuse.

D. Customer

- **Attributes:**
  - `name`: Customer's name.
  - `address`: Customer's address.
- **Relationships:**
  - Association with `ArcadeMachine` (provides contact information for the purchase).

IV. Design Patterns

- **Singleton Pattern:**
  - In the `Utilities` class only one utility instance is needed throughout the system. This avoids creating multiple instances.
- **Composition Pattern:**
  - The `ArcadeMachine` class uses aggregation by containing instances of games and accessories. This allows each arcade machine to have its own configuration of games and accessories without modifying the `GameCatalog` class.

V. SOLID Principles

- **S - Single Responsibility Principle:**
  - Each class has a single responsibility:
    - `ArcadeMachine` manages the machine.
    - `GameCatalog` manages the games.
    - `Utilities` provides utility functions.
    - `Customer` manages customer information.
- **O - Open/Closed Principle:**
  - The classes are designed to be extensible (new games can be added to `GameCatalog` or new accessories

to `ArcadeMachine`) without modifying existing code.

- **L - Liskov Substitution Principle:**
  - This principle is applied implicitly since classes derived from a base class could be replaced without altering the functionality of the system.
- **I - Interface Segregation Principle:**
  - Although there are no explicit interfaces in the current design, the separation of responsibilities implies that each class can be seen as an ”interface” that provides its own specific methods, falling into this principle.
- **D - Dependency Inversion Principle:**
  - This method is not applied yet.

VI. Conclusion

The improvements made to date reflect a focus on system extensibility, making it easier to add new functionality in the future. As we continue to develop and refine this system, it will be crucial to continue applying good programming practices to ensure its robustness and efficiency in program management.