Vietnam National University

University of Science

Faculty Information Technology

Knowledge Engineering Department

Project Report

Topic: SUPER MARIO GAME

Subject: CSC10003 - Object Oriented Programming

Student Group:

23127165 - Nguyen Hai Dang - Leader

23127330 - Ngo An Binh

23127417 - Dao Hoang Duc Manh

21127358 - Nguyen Quy Nhat Minh

Instructors:

Mr. Nguyen Le Hoang Dung

Mr. Pham Ba Thai

January 1, 2025

Contents

1 Information

1.1 Group information

- Group ID: Team 02

- Group Name: Super Mario Game

1.2 Member information

ID	Student ID	Full Name	Task(%)
1	23127165	Nguyen Hai Dang	33%
2	23127330	Ngo An Binh	33%
3	23127417	Dao Hoang Duc Manh	33%
4	21127358	Nguyen Quy Nhat Minh	01%

Table 1: Group members' information and task contribution percentages.

1.3 Contribution

Member	Commits	Contribution
Nguyen Hai Dang	86	- Implement the **GameEngine** class as the core of the game, managing gameplay mechanics, UI updates, player statistics, and score persistence to ensure a smooth and engaging experience, with a focus on scoring, level progression, and game state management. - Deploy the **Menu** class along with its subclasses such as TutorialMenu, OptionsMenu, MainMenu, CreditsMenu, etc. - Develop classes and functions related to **Items**, such as Blocks, Enemy, Character, etc.., including animations, movement, collisions, state transitions, power-ups, and more. - Create a **LevelList**, design the maps, and deploy them for all 3 levels. - Implement functions to handle tasks related to player scores upon completing a level. - Develop **AI** for Enemy characters, managing states (IDLE, MOVING, IN_SHELL, ...) and switching between them based on proximity to the player and other conditions. The AI will control when enemies should attack, move, or change behavior (e.g., when killed or inside a shell). - Implement the **Singleton Pattern** for MainMenu to ensure only one instance exists, managing menu behavior such as starting the game and exiting. - Implement the **Strategy Pattern** for encapsulating different map-building strategies for each game level. This approach ensures ease of maintenance, allows scalability for difficulty adjustments, and optimizes memory usage by enabling map deletion after use. - Source images and music for the menus, and apply sound effects for animations such as block-breaking, jump, eat, die, killed,... - Write Report
Ngo An Binh	46	- **Fix logic** for classes: Highscore, PlayerOptionMenu, PlayerNameMenu,... - Implement the **GameOver** and **Winner** screens for game. - Implement **Command Pattern** for Character class - Write Report
Dao Hoang Duc Manh	94	- Implement the **Character** class using inheritance, and polymorphism. The Character class can serve as a base class for different types of characters (e.g., Mario, Luigi,...) - Research Design Pattern and implement **Factory Pattern** and **State Pattern** for Characters - Design diagram

Table 2: Tasks assigned of each member

2 Expected Point: 10/10

Features	Max Point	Completed
Player Input, Movement, and Collision	15	15
Enemy Behaviors	10	10
Power-Ups Items	10	10
3 Level Completion	15	15
Save/Load	5	5
Sounds	10	10
Object Oriented Design	10	10
Design Pattern	25	25
AI, Multiple Characters	5	5
3D Game	5	0
Total Grade	110	105

3 Completed Requirements

Completed Requirements	Explaination
Player Inputs, Movement and Collision	- Implement user input to capture the player's name and store it for later use. - Use keyboard keys to control characters' movement. - Implement collision detection for Mario to interact with the environment.
Enemy Behavior	- Enemies follow predefined movement patterns, adding consistency and challenge to their behavior. - They can detect characters' proximity and dynamically react, such as chasing him, altering their movement,...
Power-Ups and Items	- Power-ups are strategically placed throughout the game, granting Mario special abilities like temporary invincibility or the ability to collect extra coins.
Three Levels Completion	- The game features three levels, each progressively more challenging. - Each level is populated with iconic map and items. - Once a player successfully completes a level, it becomes unlocked for replay at any time.
Graphics & Sounds	- The game's graphics are rendered using SFML, complemented by immersive background music and dynamic sound effects for actions such as jumping, collecting items, and defeating enemies.
Object-Oriented Design	- All character-related classes, including Mario and Luigi, inherit from a shared base class, Character, promoting consistency and reusability in their behavior and attributes. - Polymorphism is utilized to manage various menu character, with a vector<Characters*>dynamically storing and handling all characters
Game Design Patterns	- Singleton Design Pattern: Applied to the **Main Menu** to guarantee a single instance responsible for managing its state and behavior. - State Design Pattern: Applied to **Characters** to manage all of states. - Other Design Patterns: enhance functionality, streamline workflows, and improve code modularity and maintainability.
Development AI for game	- Develop AI for enemy characters to manage states and switch between them based on player proximity and other conditions, controlling when to attack, move, or change behavior (e.g., when killed or inside a shell).
Game State Management	- Manages various game states, including PLAYING, GAME_OVER, WINNER, and more. - Tracks the player's score and stores remaining lives throughout the game for ranking later.
Multiple Player Characters	- Players can select from 2 characters: Mario or Luigi at the game's main menu. - Each character has unique abilities and stats, offering distinct gameplay experiences.
Environment	- Game Engine: C++ and SFML

Table 3: All of completed requirements of team

4 Project Links

- Link Youtube : Playing Super Mario Game

- Link GitHub: SUPER_MARIO_GAME

5 Source Code

- Link Drive: Super Mario Game - Group 02 - Class 23CLC03

6 Specific Techniques

6.1 Design Patterns

Code Organization: Design patterns facilitate the organization of code into modular and maintainable components, which is critical for managing complex games with diverse features such as levels, enemies, and power-ups.

Reusability: Patterns such as Factory, Singleton, and State encourage the reuse of code across different game components, thereby minimizing redundancy and promoting efficiency.

Consistency: Design patterns provide a standardized approach to solving recurring challenges, such as managing characters, handling game state transitions, or managing maps for each level, ensuring uniformity across the codebase.

Flexibility: Patterns like State and Strategy enable dynamic alterations to game behavior, such as adapting game maps, player abilities, or game states (e.g., pause, game over), enhancing gameplay adaptability.

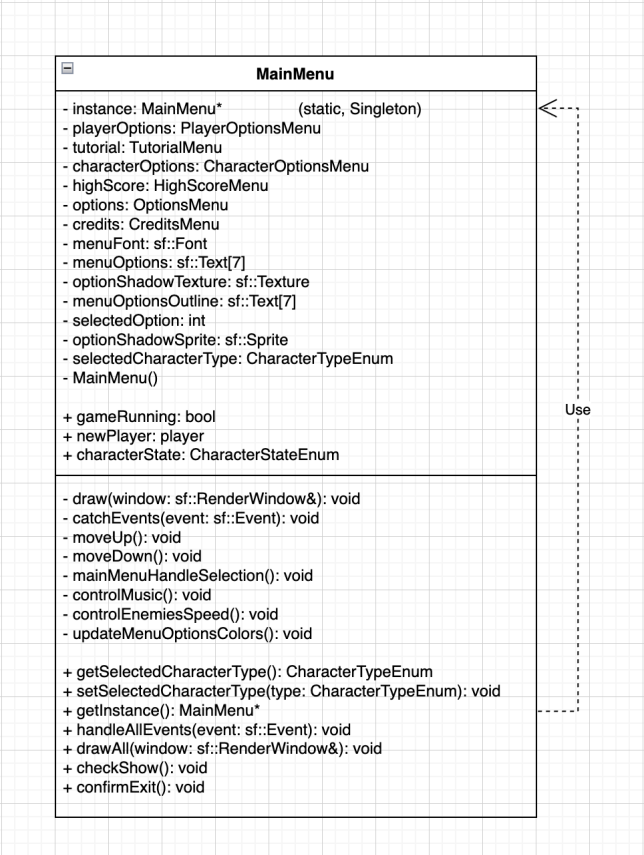
Maintenance: By incorporating design patterns, the codebase becomes more robust and easier to maintain, reducing the likelihood of introducing bugs during updates or when adding new features.

6.1.1 Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

This pattern is applied to the MainMenu class to ensure the creation and management of a

single, globally accessible instance. This guarantees centralized control over its state and behavior, preventing multiple instantiations and ensuring consistency throughout the application's lifecycle.



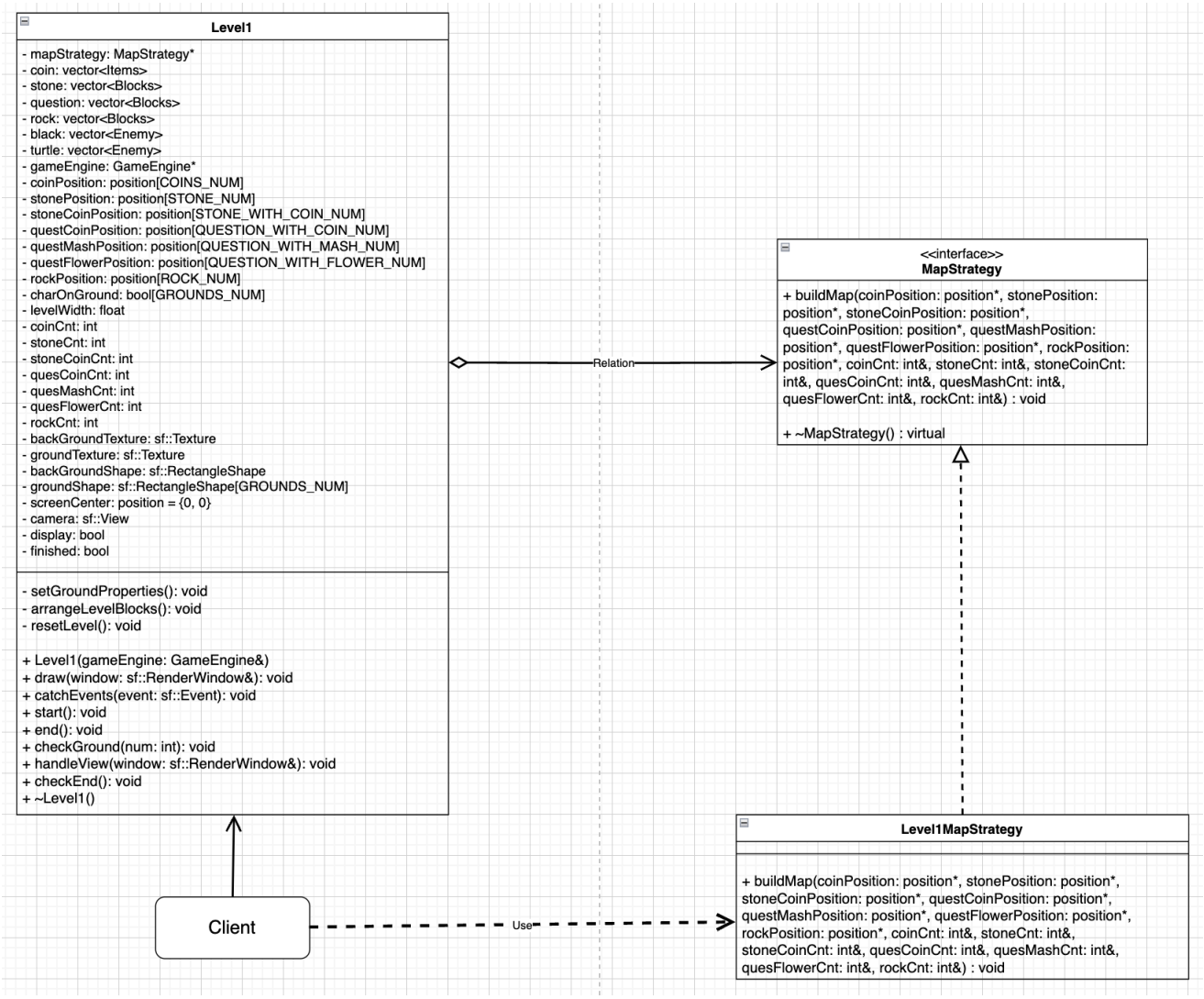
6.1.2 Strategy Pattern

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Implement **Strategy Pattern** to encapsulate different map-building strategies for each game level. This approach enhances maintainability, supports scalability for difficulty adjustments, and optimizes memory usage by allowing maps to be deleted after use. Each level can now have a dedicated map-building strategy, while still adhering to a unified interface.

```
// Base strategy
class MapStrategy {
public:
    virtual void buildMap(position* coinPosition, position* stonePosition,
        position* stoneCoinPosition, position* questCoinPosition,
        position* questMashPosition, position* questFlowerPosition,
        position* rockPosition,
        int& coinCnt, int& stoneCnt, int& stoneCoinCnt,
        int& quesCoinCnt, int& quesMashCnt, int& quesFlowerCnt,
        int& rockCnt) = 0;
    virtual ~MapStrategy() = default;
};
```

Here's how the **Strategy Pattern** is applied in our code to build the map for each level, specifically in Level 1:



6.1.3 State Pattern

The **State Pattern** facilitates seamless transitions between different states of an object, with each state encapsulating its own independent logic. Adding new states is straightforward—simply create a new class without impacting existing state classes—ensuring scalability and maintainability.

In the implementation, **State** classes such as SmallState, BigState, and SuperState are used to manage the various states of a character. Methods like setState, getState, changeToBig, and changeToSuper within the Character class handle state management and transitions efficiently.

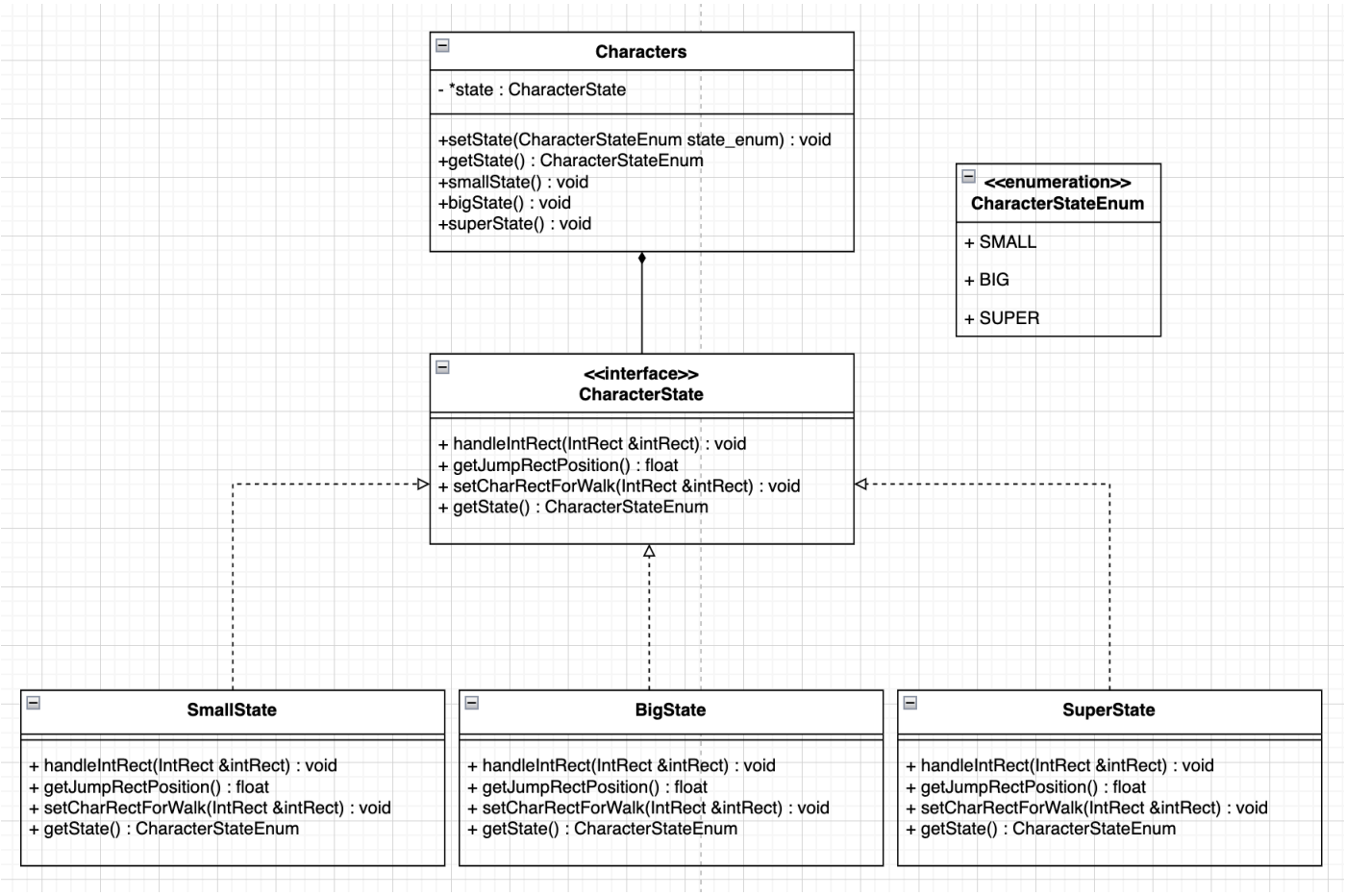
```cpp
void Characters::setState(CharacterStateEnum state_enum) {

    if (state != nullptr) {
        delete state;
    }

    CharacterState* newState = nullptr;

    switch (state_enum) {
    case CharacterStateEnum::SMALL:
        newState = new SmallState();
        break;
    case CharacterStateEnum::BIG:
        newState = new BigState();
        break;
    case CharacterStateEnum::SUPER:
        newState = new SuperState();
        break;
    default:
        break;
    }
    state = newState;
}
```

Here's how the **State Pattern** is applied in our code to manage all of characters' state:



Concrete state classes such as SmallState, BigState, and SuperState are responsible for encapsulating the logic specific to each state. These classes implement methods to handle state-specific behavior, ensuring clear separation of concerns and enhancing code maintainability.

```
class SmallState : public CharacterState
{
public:
    void handleIntRect(IntRect &intRect) override;
    float getJumpRectPosition() override;
    void setCharRectForWalk(IntRect &intRect) override;
    CharacterStateEnum getState() override;
};
```

6.1.4 Command Pattern

Command: Defines a standard interface for executing specific actions. Each action is implemented as a distinct subclass of Command, encapsulating the logic for that action.

Receiver: The entity responsible for executing the command's logic. In this context, the Character class acts as the receiver.

Invoker: The InputHandler class serves as the invoker, delegating user input to the appropriate command for execution.

Specific actions such as jumping, moving left, and moving right are implemented as separate **Command** classes. The InputHandler class maps user input to these commands and invokes the corresponding actions, ensuring a clear separation of input handling and execution logic.

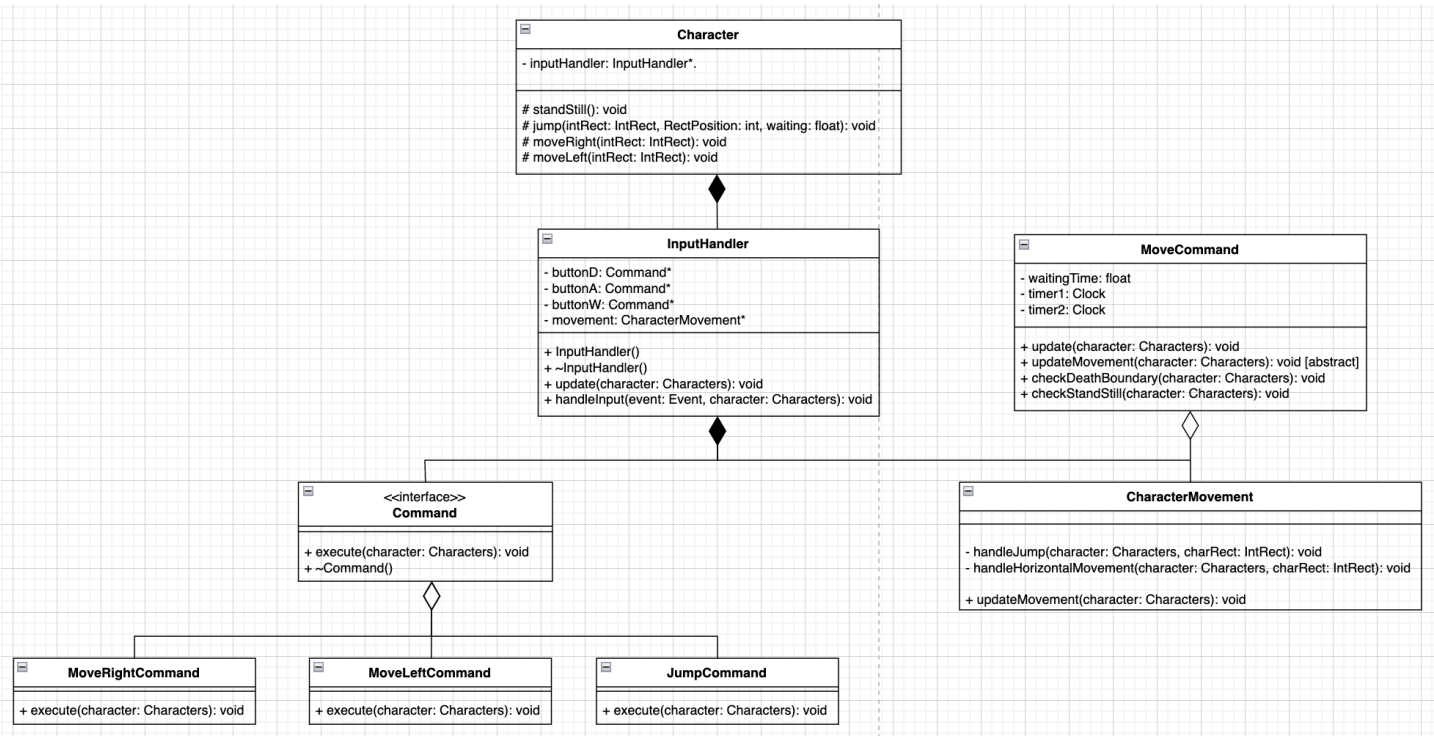
```
void MoveRightCommand::execute(Characters& character) {
    if (!character.dying && (!character.stuck || character.facingDirection == 0)) {
        character.goRight = true;
    }
}

void MoveLeftCommand::execute(Characters& character) {
    if (!character.dying && (!character.stuck || character.facingDirection == 1)) {
        character.goLeft = true;
    }
}

void JumpCommand::execute(Characters& character) {
    if (!character.dying) {
        character.goUp = true;
    }
}

InputHandler::InputHandler() {
    buttonD = new MoveRightCommand();
    buttonA = new MoveLeftCommand();
    buttonW = new JumpCommand();
    movement = new CharacterMovement();
}
```

Here's how the **Command Pattern** is applied in our code to manage all commands from player:



6.1.5 Factory Pattern

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

In the Super Mario game, the **Factory Pattern** is utilized to create character objects such as Mario and Luigi. This design pattern separates the instantiation logic from the code that uses the objects, ensuring better maintainability and scalability of the codebase.

The class **CharacterFactory** is responsible for creating character objects. It includes a static method, createCharacter, which generates character instances based on the specified character type provided as input.

```cpp
#pragma once
#include <memory>
#include "Characters.h"
#include "Mario.h"
#include "Luigi.h"

using namespace std;

class CharacterFactory
{
public:
    static shared_ptr<Characters> createCharacter(CharacterTypeEnum type, float x = 500, float y = 200);
};
```

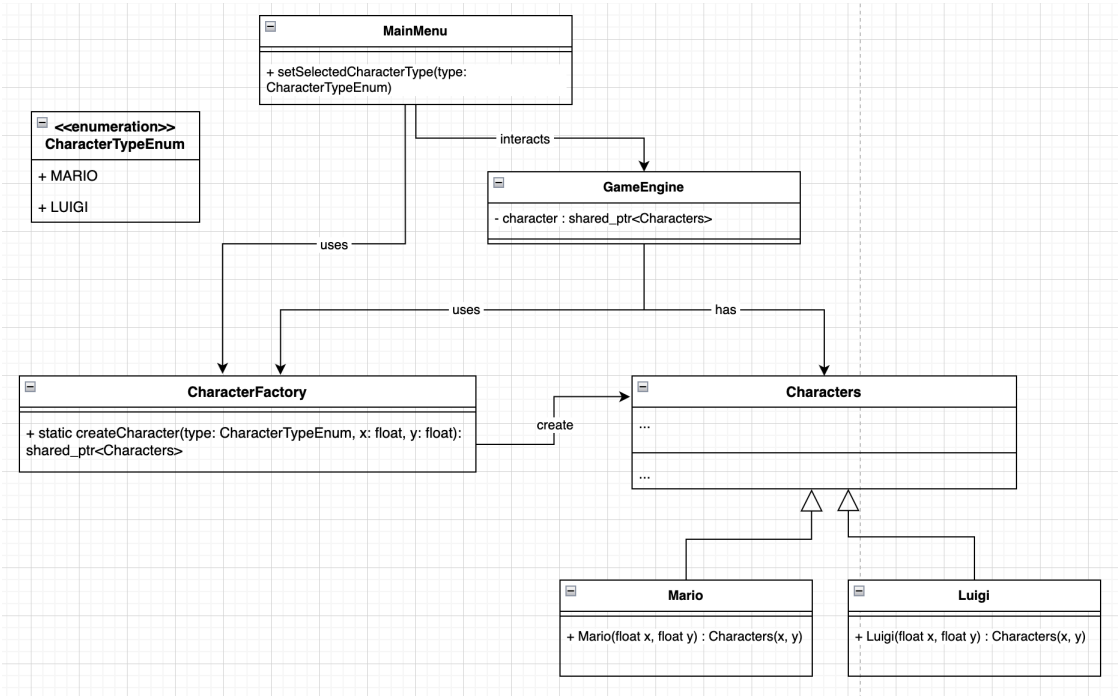
The **createCharacter** method accepts a parameter of type CharacterTypeEnum to determine the type of character to create (e.g., Mario or Luigi). Based on the value of this parameter, the method returns the corresponding character object.

```cpp
shared_ptr<Characters> CharacterFactory::createCharacter(CharacterTypeEnum type, float x, float y)
{
    switch (type)
    {
    case CharacterTypeEnum::MARIO:
        return make_shared<Mario>(x, y);
    case CharacterTypeEnum::LUIGI:
        return make_shared<Luigi>(x, y);
    default:
        return nullptr;
    }
}
```

When the player selects a character from the menu, the **MainMenu** class leverages the CharacterFactory to create the corresponding character object. This character object is then passed to and updated in the GameEngine for gameplay.

```cpp
void MainMenu::setSelectedCharacterType(CharacterTypeEnum type)
{
    selectedCharacterType = type;
    GameEngine::getInstance().character = CharacterFactory::createCharacter(type);
    GameEngine::getInstance().updateCharLifeSprite();
}
```

Here's how the **Factory Pattern** is applied in our code to create objects which is related to characters:



6.2 Object Oriented Programming

Encapsulation: Game entities like *Menu*, *Enemy*, *Item*, and *LevelList* can be modeled as classes, encapsulating their properties (e.g., position, velocity) and behaviors (e.g., movement, collision detection). This keeps the code modular and organized, making it easier to manage and scale.

Inheritance: Shared menu like *MainMenu*, *PlayerNameMenu*, *OptionsMenu*,... can be abstracted into a base class *Menu*. These specific classes can inherit from this base class and extend or override functionality as needed.

Polymorphism: By leveraging polymorphism, the game can handle different characters using a unified interface. For example, all game characters might share a vector<Characters*>dynamically storing and handling

Reusability: OOP facilitates code reuse, such as using the same *Enemy* class for different enemy types by adjusting attributes or behaviors. This minimizes duplication and reduces the chance of errors.

Collision and Interaction Management: Each object class can include methods for collision detection, ensuring that interactions like Mario jumping on enemies or collecting coins are easy to implement and modify.

6.3 SFML - Simple and Fast Multimedia Library

Ease of Use: SFML provides a straightforward and easy-to-understand API for handling common game development tasks such as graphics rendering, window management, event handling, and sound playback. This makes it ideal for projects like Super Mario, where rapid development is essential.

Cross-Platform Compatibility: SFML supports multiple platforms (Windows, macOS, Linux), enabling your Super Mario game to be easily ported across different operating systems with minimal changes to the code.

2D Graphics Support: SFML is optimized for 2D game development, offering simple but powerful tools for drawing shapes, loading textures, and managing sprites—key aspects of the Super Mario game. It also supports animation, which is critical for Mario's movement and interactions.

Audio Handling: SFML includes built-in support for sound and music, which is crucial for your game's audio needs (e.g., background music, sound effects for jumps, power-ups, and enemy interactions). This simplifies the process of implementing rich, immersive soundscapes in your game.

Extensibility: SFML is flexible and can be extended with additional libraries or custom functionality as your Super Mario project evolves. It integrates well with other C++ libraries, allowing for features such as AI, physics, and networking to be added later on.

Active Community and Documentation: SFML has an active and supportive community, as well as extensive documentation and tutorials, making it easier to find help when encountering challenges during development.

6.4 C++ Programming Language

Performance and Efficiency: C++ excels in high-performance computing and low-level memory management, making it suitable for demanding tasks like games, simulations, and embedded systems.

Object-Oriented Programming (OOP): Offers robust support for OOP principles—encapsulation, inheritance, and polymorphism—facilitating code reusability and maintainability.

Extensive Libraries and Frameworks: A vast ecosystem including libraries like Boost and STL provides tools for diverse tasks, including graphics rendering and data processing.

Cross-Platform Development: Ensures code portability across various platforms, enhanc-

ing versatility in deployment.

Game Development Capabilities: Dominates game development with engines like Unreal Engine, leveraging its real-time rendering and complex mechanics capabilities.

Rich Community and Industry Use: Decades of widespread use across industries (gaming, finance, software development) ensure rich community support and abundant resources.

Interoperability: Seamlessly integrates with other technologies, making it ideal for hybrid systems or extending applications.

7 References

[1] Super Mario Bros. (1985) (Wiki)

[2] New Super Mario Bros (Wiki)

[3] Super Mario Game - Game.com

[4] Simple Super Mario game

[5] Research for Design Pattern

[6] Resource for Mario game

[7] Object-Oriented Design (OOD) – System Design

[8] Create a UML class diagram