VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF SCIENCE — VNUHCM
Faculty of Information Technology

PROJECT REPORT — THE GAME OF GO

(final version)

Subject: Introduction to Computer Science

Student:	Vo Ba Thong
Student ID:	25125036
Submission Date:	December 14, 2025

Ho Chi Minh City, 2025

Contents

1 Chapter 1: Project Overview

1.1 Introduction

Go (Weiqi/Baduk) is one of the oldest and most complex strategy board games in the world, originating in China over 2,500 years ago. Unlike Chess, which relies on hierarchical pieces, Go uses simple identical stones to create deep strategic complexity through territory control.

This project is a comprehensive digital implementation of Go, developed for the *Introduction to Computer Science* course. It is built using the **C++** programming language and the **SFML** (Simple and Fast Multimedia Library) to provide a high-performance, hardware-accelerated graphical interface.

1.2 Objectives and Goals

The project aims to achieve the following core objectives:

- **Standardized Rules:** Implement a fully functional game engine adhering to international Go rules (Capture, Ko, Suicide prevention, Territory scoring).
- **OOP Mastery:** Demonstrate proficiency in Object-Oriented Programming (OOP) through clean architecture, resource management, and event handling.
- **AI Integration:** Develop a basic internal bot and integrate a powerful open-source engine (Pachi) for advanced difficulty levels.
- **Premium UX/UI:** Deliver a polished user experience with smooth animations, dynamic soundscapes, and responsive controls.
- **System Completeness:** Provide robust Save/Load functionality and configurable Settings, meeting real-world application standards.

2 Chapter 2: User-Visible Features

2.1 Game Modes

The application supports two primary game modes:

1. **PvP (Player vs Player):** Local multiplayer on a single device. The game automatically manages turns (Black/White), preventing out-of-turn moves and displaying the current player's status clearly.
2. **PvAI (Player vs AI):** Single-player mode against a computer opponent. It features three distinct difficulty tiers:
   - *Novice:* Uses a custom Minimax algorithm with shallow depth. Ideal for beginners learning the basics.
   - *Adept / Master:* Integrated with the **Pachi Engine**. Difficulty is adjusted dynamically by limiting the engine's search time and simulation depth per move, providing a challenge for experienced players.

2.2 Core Game Mechanics

Rules Logic

- **Capture Rule:** Stones or groups of stones are automatically removed from the board when their last liberty (adjacent empty point) is occupied by the opponent.
- **Ko Rule:** Prevents infinite loops by forbidding moves that recreate the immediate previous board state.
- **Suicide Rule:** Moves into positions with no liberties are blocked, unless that move captures opponent stones.
- **Pass and Scoring:** When both players pass consecutively, the game enters the Scoring Phase. The system calculates the score based on Chinese Rules (Area Scoring: Living Stones + Territory).
- **Dead Stone Detection:** The system automatically identifies dead stones during the scoring phase.

2.3 Utility Features

- **Undo / Redo:** An infinite history stack allows players to traverse backward and forward through the game state to correct mistakes or analyze variations.
- **Resign:** A dedicated button allows a player to surrender the match at any time.
- **Hint System:** Each player is granted one AI-suggested move per match to assist in difficult situations.
- **Time Control:** Configurable time limits (similar to Chess clocks) or an "Unlimited Time" option for casual play.

2.4 User Interface & Experience (UI/UX)

The interface follows a minimalist design philosophy with rich audiovisual feedback:

- **Interactive Elements:**
  - Buttons feature hover effects (color change/highlight) and click animations ("Bounce" scale effect).
  - Selection screens (Board Size/Difficulty) use "Scale-up & Highlight" effects for the active item, dimming others.
- **Visual Feedback:**
  - **Stone Placement:** Stones scale down ($1.5x \rightarrow 1.0x$) upon placement to simulate physical impact.
  - **Last Move Marker:** A distinct red dot highlights the most recently placed stone.
  - **Timeline Bar:** A visual bar displays thinking time for the last 20 moves, helping players track game pacing.

  - **History List:** A scrollable sidebar logs moves (e.g., "D4", "Q16") for review.
  - **Real-time Status:** A compact panel shows captured stones count and remaining time.
  - **Smart Logs:** In-game notifications (Info, Error, Success) appear with color-coding and auto-fade after a few seconds.

- **Theming:**
  - Hot-swap themes for Board and Stones during gameplay without restarting.
  - Distinct sound packs for each stone theme (e.g., Glass tap vs. Wood clack).

2.5 End Game Effects

Visualizing the Result

- **Dead Stones Animation:** Dead stones fade out (alpha transparency) and shrink to visually distinguish them from living stones.
- **Territory Wave:** A gradient "wave" animation propagates through empty territory regions, visualizing the flood-fill algorithm's progress.
- **Screen Transitions:** Smooth Slide and Scale effects are used when switching between menu and game screens.
- **Scoreboard:**
  - *Resign/Timeout:* A notification banner drops down with a "Winner Stamp" effect.
  - *Scoring:* A detailed scoreboard drops down. Points (Stones + Territory + Komi) count up from 0 with sound effects, followed by a final "Stamp" animation declaring the winner.

2.6 System Utilities

- **Save/Load System:**
  - Unlimited save slots using text-based serialization.
  - **Visual Preview:** Generates a thumbnail image of the board state for each save slot using `sf::RenderTexture`.
  - Displays rich metadata: Board size, Game Mode, Timestamp, Difficulty.
  - Safety features: Confirmation dialog before deleting save files.
- **Settings:**
  - Controls: Steppers (Time Limit, Komi), Sliders (Volume), Radio Buttons (Themes).
  - Accessible globally from Main Menu or In-game Pause Menu.
  - Auto-save preferences.

3 Chapter 3: Design Document

3.1 Overall Architecture

The project follows a modular architecture to separate concerns, ensuring maintainability and scalability:

- **GameCore:** Manages the core business logic (Rules, Board State, Bot communication, Resource loading).
- **UI Module:** Handles rendering, animations, user input processing, and audio output.

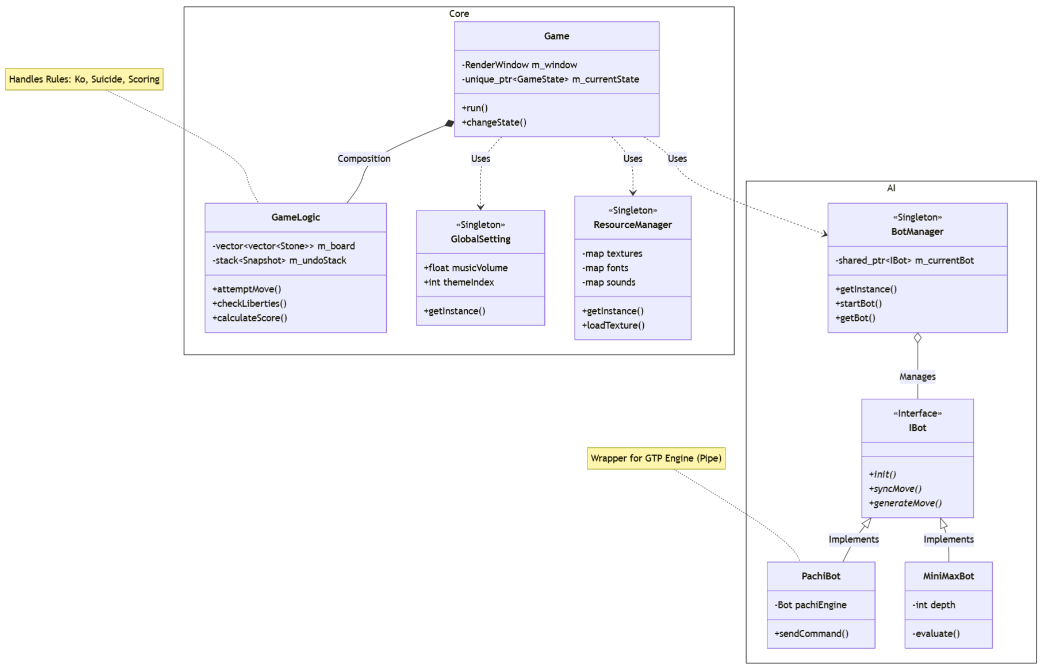

Figure 1: Backend Architecture: Game Logic and AI Subsystems

3.2 State Management (State Pattern)

The game utilizes the **State Design Pattern** to manage screen transitions effectively.

- **Context:** The `Game` class holds the current state.
- **Base State:** An abstract `GameState` class defines the interface.
- **Concrete States:** `MainMenu`, `NewGame`, `SavedGame`, `Setting`, `Gameplay`, `SizeSelect`, `PauseMenu`, `About`.
- **Signals:** Special signals like `ResetGame` or `SaveGameRequest` facilitate communication between states.

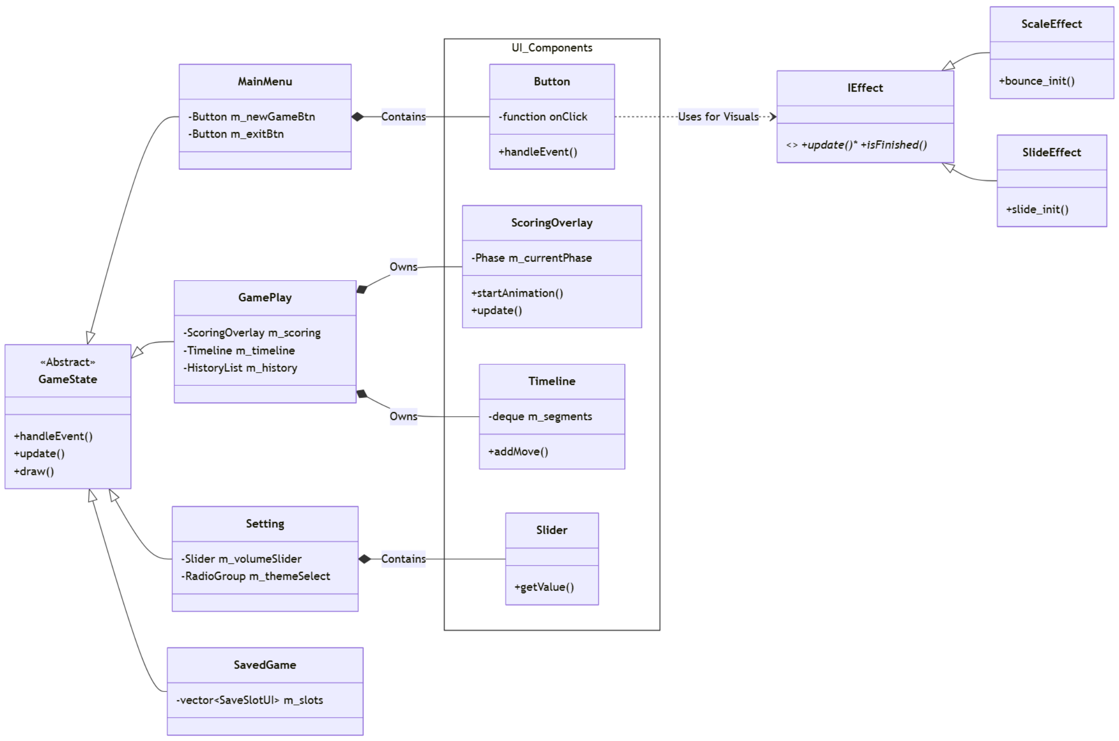

Figure 2: Frontend Architecture: State Machine and UI Components

3.3 Resource Manager (Singleton)

Implemented as a **Singleton**, the `ResourceManager` class centralizes asset management. It pre-loads all textures, sounds, and fonts into `std::map` containers at startup. This prevents gameplay lag caused by on-the-fly loading.

3.4 Bot Integration

- `IBot`: An abstract interface defining common bot behaviors (e.g., `generateMove`).
- `MinimaxBot` and `PachiBot`: Concrete implementations. `PachiBot` wraps the external process communication.
- **Multi-threading:** AI calculations run on a separate worker thread. This ensures the main UI thread remains responsive (preventing the "Not Responding" window state) while the bot "thinks".

4 Chapter 4: Game Development Environment

This chapter provides a comprehensive, step-by-step tutorial on setting up the development environment used for this project.

4.1 Environment Specifications

- **IDE:** Code::Blocks 20.03
- **Compiler:** MinGW-w64 GCC 13.1.0 (64-bit)

- **Library:** SFML 2.6.1 (GCC 13.1.0 MinGW (SEH) - 64-bit)

4.2 Detailed Setup Guide

4.2.1 Step 1: Download and Extract SFML

1. Visit the official SFML download page: `https://www.sfml-dev.org/download/sfml/2.6.1/`
2. Locate the version matching your compiler. For this project, download GCC 13.1.0 MinGW (SEH) -
3. Extract the downloaded archive to a permanent location (avoid spaces in the path).
4. *Example Path:* `C:/Libraries/SFML-2.6.1`

4.2.2 Step 2: Create a Project in Code::Blocks

1. Open Code::Blocks. Go to **File** → **New** → **Project...**
2. Select **Console Application** and click **Go**.
3. Choose **C++** as the language.
4. Name your project (e.g., "GoGame") and select a folder to save it. Click **Next**.
5. Ensure the Compiler is set to **GNU GCC Compiler**. Click Finish.

4.2.3 Step 3: Configure Compiler and Linker

This step links your project to the SFML headers and library files.

- Right-click your project name in the workspace → Select **Build options**

A. Search Directories (Tab: Search directories)

1. Select the **Compiler** sub-tab:
   - Click **Add**. Browse to the `include` folder inside your SFML directory.
   - *Path:* `C:/Libraries/SFML-2.6.1/include`
2. Select the **Linker** sub-tab:
   - Click **Add**. Browse to the `lib` folder inside your SFML directory.
   - *Path:* `C:/Libraries/SFML-2.6.1/lib`

B. Linker Settings (Tab: Linker settings) *You must configure Debug and Release targets separately.*

- **For Debug Target:**
  1. Select **Debug** in the left panel.
  2. Under **Link libraries**, click **Add** and type the following libraries exactly (order matters, note the `-d` suffix for Debug):

  - `sfml-graphics-d`
  - `sfml-window-d`
  - `sfml-audio-d`
  - `sfml-system-d`
  - `sfml-network-d`

- **For Release Target:**
  1. Select Release in the left panel.
  2. Under Link libraries, add the same libraries but **WITHOUT** the `-d` suffix:
     - `sfml-graphics`
     - `sfml-window`
     - `sfml-audio`
     - `sfml-system`
     - `sfml-network`

4.2.4 Step 4: Manage DLL Files

The executable needs dynamic link libraries (DLLs) to run.

1. Navigate to the `bin` folder in your SFML directory (`C:/Libraries/SFML-2.6.1/bin`).
2. Copy all `.dll` files (e.g., `sfml-graphics-2.dll`, `openal32.dll`).
3. Paste them into the folder where your project's `.exe` is generated (usually `bin/Debug` and `bin/Release` inside your project folder).

4.3 Sample Starter Code

Use the following code in `main.cpp` to verify your installation:

```cpp
#include <SFML/Graphics.hpp>

int main() {
    // Create the main window
    sf::RenderWindow window(sf::VideoMode(800, 600), "Go Game Setup Test");
    window.setFramerateLimit(60);

    // Create a graphical shape
    sf::CircleShape shape(50.f);
    shape.setFillColor(sf::Color::Green);
    shape.setPosition(375.f, 275.f);

    // Main game loop
    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        // Render cycle
        window.clear();
        window.draw(shape);
        window.display();
    }
    return 0;
}
```

Listing 1: Basic SFML Test Application

5 Chapter 5: Implementation Details

5.1 Project Structure

The source code is organized as follows:

- `include/`: Header files defining classes and data structures.
- `src/`: Implementation files (.cpp) for Logic, UI, and Bot.
- `resources/`: Static assets (Images, Sounds, Fonts).
- `assets/`: Runtime generated data (Settings, Saved Games).

5.2 Libraries and Tools Used

- **SFML Modules:**
  - `sfml-graphics`: Rendering sprites, shapes, and text.
  - `sfml-window`: Managing the OS window and input events.
  - `sfml-audio`: Playing sound effects and background music.
- **Standard C++ Libraries:**
  - `vector, queue`: Managing dynamic lists (e.g., board history, BFS queues).
  - `cmath`: Mathematical calculations for animations and geometry.
  - `filesystem`: Managing save directories and file paths.
  - `fstream, sstream`: File I/O for saving/loading and parsing configurations.
  - `thread, mutex`: Handling concurrency for AI processing.
  - `functional, memory`: Managing callbacks and smart pointers.
  - `algorithm, iomanip`: Data processing and formatting.

5.3 Major Components

Key Technical Implementations

- **Multi-threading AI:** Decoupling AI logic from the main thread prevents UI freezing.
- **Territory Scoring (BFS):** A Breadth-First Search algorithm determines connected regions of empty space to calculate territory ownership.

- **Rule Enforcement:** Robust logic checks for Ko, Suicide, and Capture rules.
- **Custom UI Framework:** A bespoke UI system (Buttons, Sliders, Scrollbars) built from primitives, independent of external GUI libraries.
- **Animation State Machines:** Complex animations (like the Scoring Overlay) are managed via state machines (Phases: DeadStone → Territory → Scoreboard → Stamp).
- **Save/Load System:** Text-based serialization combined with `sf::RenderTexture` for generating visual thumbnails.

5.4 System Workflow

The following sequence diagram illustrates the runtime interaction between the Game Loop, Logic Core, and Rendering Engine during critical phases (Startup/Scoring).

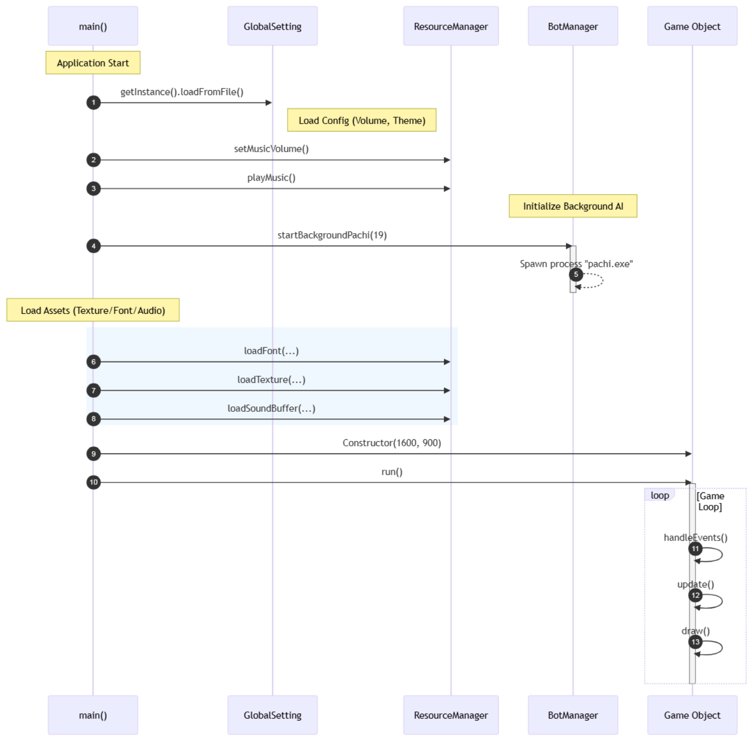

Figure 3: System Sequence Diagram: Interaction Flow

6 Chapter 6: Testing

Rigorous testing was conducted to ensure stability and correctness:

- **Rule Verification:** Verified Ko and Suicide rules by attempting invalid moves. Checked Capture logic with various shapes (snapback, ladder).
- **AI Performance:** Tested Novice (Minimax) vs Master (Pachi) to confirm difficulty scaling. Verified "Thinking" UI feedback.

- **System Utilities:** Validated Save/Load data integrity (Timer, Board State). Verified correct audio playback for different themes.
- **Functionality:** Tested Resign, Hint, and End Game scoring sequences.

7 Chapter 7: Future Improvements

1. **Game Review Mode:** Allow players to step through completed games move-by-move. Implement "What-if" branching to explore alternative strategies.
2. **Win-Rate Graph:** Display a real-time evaluation graph showing the win probability shift after each move (based on AI analysis).
3. **Bot Analytics:** Display detailed statistics on Bot performance (nodes visited, depth reached).

8 Chapter 8: Conclusion

The Go Game project has successfully met all core requirements of the *Introduction to Computer Science* course while incorporating advanced features in graphics, audio, and AI.

Through this project, the team has effectively applied:

- Object-Oriented Programming (OOP) principles and Design Patterns (State, Singleton, Factory).
- Advanced C++ techniques (Memory management, Concurrency).
- Practical integration of external libraries (SFML, Pachi Engine) into a cohesive software product.

The final product is a complete, stable, and practical game application.

References

[1] Laurent Gomila et al., *Simple and Fast Multimedia Library (SFML) Documentation*, `https://www.sfml-dev.org/`.

[2] Petr Baudis, *Pachi: A simple modular Go program*, `https://github.com/pasky/pachi`.

[3] Sensei's Library, *Chinese Rules of Go*, `https://senseis.xmp.net/?ChineseRules`.

[4] C++ Reference, *Standard C++ Library reference*, `https://en.cppreference.com/`.