Mahjong Solitaire Project Documentation

1. Overview

What the Software Does:
This Mahjong solitaire application provides a playable Mahjong “Turtle” layout, allowing players to select tiles and remove them if they form a matching pair. The game is a single-player puzzle in which tiles must be matched and cleared until no moves remain or the board is empty.

Key functionalities include:

- Displaying a standard Mahjong “Turtle” layout with various tile types (Bamboo, Circle, Pinyin, Seasons, Flowers).
- Allowing the user to select open tiles. If two selected tiles match, they are removed.
- Providing visual cues: open tiles are fully opaque, blocked tiles are partially transparent, and selected tiles are highlighted.
- Offering a “Shuffle” feature to randomize tiles’ positions if the player gets stuck.
- Playing sound effects for clicks, pair removals, and mistakes.

What the Software Does Not Do:

- It does not guarantee that all starting boards are solvable. Some layouts might be unsolvable.
- It does not provide hints or advanced AI assistance to the player.
- It does not support multiplayer, saving/loading states, or advanced features like scoring systems.

The primary goal is to demonstrate a working Qt-based application with a QML frontend and a C++ backend that manages game logic, adhering to an object-oriented design and good software development practices.

2. Software Structure

Overall Architecture:
The application is divided into two main layers:

1. **C++ Backend (Logic Layer):**
    - **Tile**: Represents a single Mahjong tile, with properties like type, value, row, column, layer, open, and selected.
    - **TileModel**: A QAbstractListModel subclass that holds and manages a collection of Tile objects. It exposes these tiles to the QML UI as a model.
    - **Board**: Encapsulates the game logic. It initializes the "Turtle" layout, updates which tiles are open, handles tile selection and matching, and provides the shuffle functionality.

2. **QML Frontend (UI Layer):**
    - A main.qml file that creates a window, lays out the tiles by reading from tileModel, and visually represents each tile with images and colors.
    - Interactivity (clicking on tiles, pressing the Shuffle button) triggers calls to methods in Board through exposed context properties.

Interfaces to External Libraries:

- **QtQuick/QML:** For the declarative UI and layout of graphical elements.
- **QtMultimedia:** For playing sound effects.
- **QtCore, QtGui:** Core application loop, signals/slots, basic data types.
- **C++17 Standard Library:** For random shuffling and basic utilities.

No additional third-party libraries were used.

3. Instructions for Building and Using the Software

Build Requirements:

- Qt 5.15+ or Qt 6 (including QtQuick, QtMultimedia)
- A C++17-compliant compiler (e.g., g++, clang++)
- CMake or qmake (depending on chosen build system)
- Make tool (e.g., make on Linux/Mac or nmake on Windows with MSVC)

Compilation (Using qmake):

1. Clone the repository from Git.

In the project's root directory, run:
bash
Copy code
make clean
qmake project.pro
make
./mahjong

2. This should produce an executable (e.g., ./mahjong on Linux/macOS).

Dependencies: All required Qt modules are assumed to be installed on the system. If Qt is not installed, please download from https://www.qt.io/download.

Running the Program: After successful compilation, run:

bash
Copy code
./mahjong

This will open a window displaying the Mahjong board.

How to Use the Software (User Guide):

1. On launch, you'll see the Mahjong turtle layout with tiles stacked.
2. Click on any open tile (fully visible) to select it. It will highlight.
3. Click on another open tile. If they match, both will be removed with a sound effect. If not, a "mistake" sound will play and selections will reset.
4. Continue removing pairs until no moves remain or the board is empty.
5. If stuck, press the "Shuffle" button at the bottom to randomize tile positions.
6. The game continues until you choose to close the application.

4. Testing

Testing Approach:

- **Unit Tests**: Placed in the tests/ directory.
    - test_tile.cpp, test_board.cpp: Validate tile property changes, board initialization, tile matching logic, and open state calculations.
    - Each test file compiles into a small executable that checks assumptions about the classes.

Testing Steps:

1. Navigate to the tests/ directory.
2. Run make to build the test executables.
3. Execute the test binaries (e.g., ./test_tile, ./test_board) to see if all tests pass.

Copy
cd tests
qmake tests.pro
make clean
make
./tests

Test Coverage:

- **Tile Class Tests:**
    - Confirm that setting type, value, row, column, selected, open, and layer properties works as expected.
    - Check that signal emissions occur on property changes.

- **Board Class Tests:**
    - Confirm that generateTurtleLayout() produces the correct number of tiles and places them at expected positions.
    - Verify that selectTile() logic correctly identifies open tiles and updates state after removal.
    - Check that tilesMatch() accurately identifies pairs (especially Seasons and Flowers).
    - Validate that shuffle() redistributes tiles without losing any.

Testing Outcomes: All basic tests (tile properties, layout generation, matching logic) passed successfully. Minor adjustments were made when initial tests revealed indexing issues. After fixes, tests confirmed stable and expected behavior.

5. Work Log

Project Duration: 4 weeks (4 sprints)

Sprint 1 (Week 1):

- **Planned Tasks:**
    - Set up repository structure, initialize Tile and TileModel classes.
    - Display a placeholder layout in QML.
- **Completed Tasks:**
    - Repository created with directories: src/, plan/, doc/, tests/.
    - Basic Tile and TileModel implemented and integrated with QML as a Repeater.

Sprint 2 -> final

- Continuous work and completion

7. Conclusion

The project achieved its goals:

- A functional Mahjong solitaire game with a QML interface and C++ backend logic.
- Ability to select and remove matching pairs of tiles.
- Additional features like shuffle and sound effects included.
- Proper documentation, testing, and adherence to coding styles.

Though we did not ensure that every layout is solvable, and we did not add advanced features like hints or 3D effects, the core gameplay is implemented and stable.