Pathfinding and Maze Traversal Engine: An Object-Oriented Design

OOPS Compliance Report by
CS24I1029 YASHVANTH S
CS24I1035 JEEVANANDHAM T
CS24I1039 P Y NITHILAKRISHI
CS24B2054 SRI HARI S

November 17, 2025

1 Overview

This document summarizes how the Maze Engine project satisfies the required Object-Oriented Programming (OOP) concepts: encapsulation, abstraction, operator overloading, inheritance, polymorphism, exception handling, and templates. Each subsection highlights representative classes and methods within the repository (commit state dated November 17, 2025).

2 Encapsulation

- `Maze` (`include/Maze.h`) stores its grid, dimensions, and endpoints as private members while exposing safe getters and setters.
- `Path` and `Point` encapsulate their coordinate and cost data, offering controlled mutation via methods like `addPoint` and `setCost`.
- `Renderer` keeps rendering buffers and palette data private, exposing only high-level rendering APIs.

3 Abstraction

- `Maze::getNeighbors` abstracts grid traversal details for solver algorithms.
- `PathAnalyzer` (`src/PathAnalyzer.cpp`) delegates calculation details to helpers such as `calculateTurns` and `countNarrowPassages`, offering a simple `analyze` interface.
- `MazeGenerator` exposes preset generation methods (`generateEasy`, `generateHard`) that hide recursion, randomness, and terrain placement implementation details.

4 Operator Overloading

- `Point` overloads arithmetic (`operator+`, `operator-`) and comparison operators to ease geometric manipulation.
- `Path` overloads `operator[]` for bounds-checked access and `operator+` to concatenate paths.
- `PathMetrics` overloads comparison operators and stream insertion to simplify reporting (`src/PathAnalyzer.cpp`).

5 Inheritance and Polymorphism

- `MazeSolverStrategy` (`include/MazeSolverStrategy.h`) defines the abstract contract for maze-solving algorithms.
- `BFSSolver` and `DijkstraSolver` now inherit from `MazeSolverStrategy`, overriding `solve`, `getNodesExplored`, and `name`.
- `MazeSolverApp::handleQuickSolve` (`src/main.cpp`) uses a `std::vector<std::unique extunderscore ptr<MazeSolverStrategy>>` to iterate polymorphically over solvers, invoking the virtual interface without knowing concrete types.

6 Exception Handling

- `Exceptions.h` introduces `MazeException` and `AnalysisException` derived from `std::runtime extunderscore error`.
- `MazeSolverApp::handleLoadMaze` wraps file loading in a `try-catch` block, throwing `MazeException` on failure to provide meaningful feedback.
- `PathAnalyzer::analyze` employs a `try-catch` block when computing averages, gracefully handling empty sample sets via `AnalysisException`.

7 Templates

- `StatsAggregator<T>` (`include/StatsAggregator.h`) is a templated utility constrained to arithmetic types, providing `min`, `max`, and `average` calculations.
- `PathAnalyzer::analyze` instantiates `StatsAggregator<double>` to compute per-step traversal averages, illustrating template usage in production code.

8 Summary Table

Concept	Implementations
Encapsulation	`Maze`, `Path`, `Renderer` keep state private and expose public APIs.
Abstraction	`Maze::getNeighbors`, `PathAnalyzer::analyze`, `MazeGenerator` presets.
Operator Overloading	`Point` arithmetic/comparison, `Path::operator+`, `PathMetrics` comparisons.
Inheritance	`BFSSolver`, `DijkstraSolver` derive from `MazeSolverStrategy`.
Polymorphism	`MazeSolverApp::handleQuickSolve` executes solvers via virtual interface.
Exception Handling	`MazeSolverApp::handleLoadMaze`, `PathAnalyzer::analyze`, custom exceptions in `Exceptions.h`.
Templates	`StatsAggregator<T>` reused in `PathAnalyzer::analyze`.