2AA4 A3: Design Patterns and Evolution – Report

Adrian Winter - wintea10

GitHub Repo URL: https://github.com/Adrianw27/a1-mazerunner

1. The unit test suite for the maze runner project was designed to target the critical functionalities of the maze solving logic. In specific, unit tests were written to test the implementation of the RightHandMazeSolver class. The tests covered both positive and negative scenarios to ensure that all cases were considered. For path validation, a sample maze was generated and used for unit tests with hardcoded paths. These test cases included correct path validation, incorrect path handling (invalid moves, invalid characters, extra characters), and edge cases (empty paths). Additionally, tests were written to validate that the solver generates non-empty, executable paths. Finally, one unit test validated the path factorization behaviour of the PathFormatter class. Each test was structured to be independent and minimal, targeting one specific case of a feature. This ensured maintainability while also improving the accuracy of the tests. The test suite was executed after each meaningful code change during development, especially when introducing new logic in the solver or when modifying core behaviours. This feedback loop made it easy to catch bugs early, leading to manageable and simple debugging and enforcing expected behaviour. The benefits of using this test suite were substantial. It enabled confidence in refactoring and adding features to the code, such as the abstract template and decorator classes. It also provided a reliable benchmark for

the correctness of path generation and validation. Frequent testing encouraged consistency and promoted modularity and robustness in the design.

2. The first design pattern I decided to implement was the Template pattern. When I selected this pattern, I was considering the next steps to improving the code, and the main extension I noticed was implementing a more efficient search algorithm. I then selected the Template pattern to enforce a consistent structure across different maze-solving algorithms with this in mind. In the original implementation, the maze solver logic was directly implemented in the RightHandMazeSolver class, leading to tight coupling and poor extensibility. Recognizing that all solvers shared a common high-level structure (navigate maze, track path, stop at exit), this flow was abstracted into the MazeSolverTemplate. I also realized that the validatePath() method was independent of the search algorithm used, so it was entirely abstracted into the template, preventing code duplication. The abstract template class thus defined the structure for findAnyPath() and validatePath() methods. It required implementing classes to define the specific path finding logic in a protected solve() method. This made it easy to extend the class with other solving strategies (such as BFS, Djikstras, etc.) without changing the template or client code. The template was used to handle common concerns like path recording and input validation, while the subclass was left to focus solely on pathfinding logic. Overall, this decision was made because it promoted reusability and extensability

through abstraction, improved testability by separating concerns, prevented code duplication, and aligned with the Open/Closed principle.

The second design pattern that I decided to implement was the Decorator pattern. I selected this pattern when I noticed old solver implementations may require temporary features to aid in debugging, which could not be achieved without modifying the internal code at the time. The pattern was then implemented to add new behaviour, specifically logging, without modifying the internal path validation and generation logic. This was achieved by an abstract class MazeSolverDecorator that implemented the MazeSolverAlgorithm, and delegated calls to an internal wrapped instance. The concrete class I implemented was LoggingMazeSolverDecorator, which added console logging before and after solving or validating a path for testing and debugging concerns. The main benefit of the decorator is that more features can easily be added, combined, and/or removed in the future without changing the existing code, such as a performance metrics decorator. This pattern was mainly chosen to achieve separation of concerns as well as improve extensibility. By wrapping any solver implementation, such as RightHandMazeSolver, I could dynamically add new functionality, without altering the existing code. This is useful in a system where variations of behaviour (debugging, monitoring, performance) may be required conditionally or temporarily. In this case, the decorator kept the solver code clean, but allowed for easy toggling of the logging feature to find points of errors. Both the patterns I implemented completented each other. The template standardized the structure of solver logic,

while the decorator allowed for new runtime behaviour. Together, they made the codebase more flexible, maintainable, and testable.

3. One valuable design pattern that is not yet implemented but would be highly beneficial in the design is the Factory pattern. In the current codebase, the selection and instantiation of maze solvers is hardcoded. This can be seen in the line: MazeSolverAlgorithm solver = new RightHandMazeSolver();. This violates the Open/Closed principle and couples the Main class to a specific solver implementation. In the future, new algorithms may be introduced to maximize path generation efficiency, for example, implementing a breadth first search solver. This would not be possible to implement and test without modifying external code. The Factory pattern would solve this problem by abstracting the solver creation logic into a factory class. For example, a SolverFactory could use a public static method getSolver(String type) that returns a solver instance based on configuration. Internally, it could use a switch-case to instantiate RightHandMazeSolver, or future solvers such as BFSMazeSolver. This would allow the Main class to be decoupled from the solver type. This would also make it easier to extend the system with new algorithms or use different decorators conditionally (for example: use logging solver only in debug mode). This change would introduce one new class to the code (the factory), as well as a small update to the entry point of the program in Main. This change would allow for easy scalability and reduce the risk of runtime errors due to

incorrect instantiation logic. Additionally, the code would become more modular, less coupled, easier to test, and easier to maintain.

4. Working with design patterns on this project offered insight into the value of writing clean structured code with abstraction. Early in the project, the logic for solving and validating mazes was embedded directly in the solver class. As the codebase grew, the negative effects of poor modularity and extensibility became clear. Learning and applying the Template and Decorator patterns helped separate concerns, reduce code duplication, and promote consistent abstraction. The biggest realization I had was how design patterns improved the ease of extending and maintaining the code. Once the Template pattern was in place, implementing a new algorithm for solving the maze became as simple as extending the base class. There was no more code duplication from the validatePath method, and the new class would only have to implement logic for finding a path. Similarly, wrapping solvers with decorators gave me a flexible mechanism for runtime enhancements, allowing me to easily add optional features to existing solvers such as logging, without invasive changes to external client code. Had I known these patterns from the beginning, the system would have been designed with more abstract layers from the start. For instance, the solver instantiation logic and validation method could have been abstracted and reused from the start. This would not only allow for easier testing and debugging, but also future proof the code. Design patterns improved the current code, but their main benefit was making future changes more predictable,

manageable, and easier to implement. In future projects, I would strongly consider applying design patterns early in the design process, especially in systems that are expected to grow or support multiple variants. They offer both good code structure and flexibility, ultimately leading to cleaner and more maintainable code.