2AA4 Assignment3 Report

1. Elaborate on the tests: design, execution, targeted functionality, etc. What were the benefits of using a test suite? How frequently did you run your tests? About 1 page.

To ensure the correctness of my MazeRunner project, I designed a suite of unit tests using the JUnit 5 framework. The goal of these tests was to validate the system's core functionality by covering typical execution paths, edge cases, and potential failure scenarios. In total, I wrote 13 unit tests that covered both the fundamental maze logic and the newly added design patterns. Initially, I focused on the core classes like Maze, Position, and Path. Each test was written to validate one specific behavior, and I stuck to a “one test per feature” mindset to make debugging easier and results more readable. For example, I had testGetStart() and testGetEnd() to confirm that the maze correctly detects the entrance and exit points. I also included checks like testIsWall() and testIsPassage() to ensure that cell values were correctly interpreted. Another important one was testValidatingPath(), which tested if a given string of movement instructions actually led the explorer from the start to the end. That one proved especially useful, because at one point it failed when parsing commands like 14F. I realized the path validation code only handled single-digit numbers, so I rewrote that part to build multi-digit integers with a StringBuilder. That fix made the parser much more robust and let me handle complex paths properly.

Once I introduced the Command and Observer patterns, I expanded my test suite to make sure those pieces were working as expected. For this, I created a new test file that focused entirely on testing the new design logic without relying on the maze-solving algorithm. I wrote a set of tests like testMoveCommand(), which checks if executing a move command updates the x or y coordinate correctly depending on the direction. Similarly, testTurnCommand() validates whether a left or right turn actually changes the explorer’s direction. I also added a testMultipleCommands() test that runs a sequence of moves and turns to check if they all execute in the correct order and produce the expected final position and orientation. Alongside those, I created a TestObserver class that acts like a fake observer to track command executions. This observer didn’t print anything or affect the output, but instead recorded what commands were run and what the position was at each step. This helped me confirm that the observer pattern was wired up correctly and that it responded to every single command execution. It was a clean way to validate the pattern without cluttering the program's runtime output.

I ran the full test suite regularly—pretty much after every major code change or whenever I introduced something new. This gave me constant feedback and helped catch issues early. It also gave me the confidence to refactor and improve things, since I knew any bugs would immediately show up in the test results. Especially after introducing design patterns, having these tests made it easier to isolate logic, debug problems, and make sure the patterns weren’t just added for structure, but were actually functioning the way they were supposed to.
In the end, the testing phase not only helped verify that the program was working, but also made development faster and more efficient. It changed how I thought about writing code—more modular, more testable, and way easier to extend.

2. Document how you selected the two implemented design patterns. Argue and justify your points properly. 2-3 pages max.

Taking another look at Assignment 1, I noticed that public methods in the Position class—such as move() and turn()—were being invoked directly and frequently to perform specific actions. These method calls acted as clear "commands," each intended to manipulate the internal state of the explorer. This revealed a perfect opportunity to adopt the Command pattern.

The Command pattern encapsulates requests (actions) as objects. This decouples the invoker of the command from the object that actually performs it, enabling better modularity, extensibility, and history tracking. In this case, rather than calling position.move() directly, I could encapsulate this logic within a MoveCommand class that implements a generic Command interface. Similarly, TurnCommand would handle direction changes.
To implement this pattern, I started by creating a Command interface with a standard execute() method. Then I wrote two concrete command classes—MoveCommand and TurnCommand—each holding a reference to a Position object and encapsulating a single unit of movement or rotation.

c > main > java > ca > mcmaster > se2aa4 > mazerunner > J Command.java > Language Support for Java(TM

```java
package ca.mcmaster.se2aa4.mazerunner;

public interface Command {
  void execute(); // This method returns the Position that is affected by the command
Position getTargetPosition();
}

```

I also added a CommandInvoker class to handle execution and record commands, which enabled step-by-step processing and history tracking.

```java
package ca.mcmaster.se2aa4.mazerunner;

import java.util.ArrayList;
import java.util.List;

public class CommandInvoker {
  private List<Command> commandHistory = new ArrayList<>();

  private List<CommandObserver> observers = new ArrayList<>();
public void execute(Command command) {
  command.execute();
  commandHistory.add(command);
  notifyObservers(command);
}
```

This redesign made a noticeable difference. It created better separation of concerns—logic for movement and rotation was now modular and self-contained. It also made each action easier to test on its own, and the encapsulation meant I didn't have to expose or tightly couple other parts of the system just to move or turn. It became a lot easier to maintain, since I could

add new behaviors without touching the core logic. And maybe the biggest benefit long-term is how this pattern opens the door to experimenting with new solving algorithms. Right now, the maze is solved using the right-hand rule, but if I wanted to try a more efficient algorithm that needed to backtrack or re-order steps, the Command pattern would make that straightforward. I could undo commands, rearrange them, or insert new ones without changing the main logic. Command also sets the project up nicely for future features like undo/redo, animation playback, or even saving and loading step histories. Separating the creation of commands from their execution just made the whole structure more flexible and ready to grow.

After implementing the Command pattern, I started thinking about how I could decouple output or monitoring behavior from the core maze logic. I didn't want logging or external effects to be hardcoded into the movement logic itself. I realized that every time a command like move or turn gets executed, it could be considered an event—something that other parts of the system might want to listen to. This led me to bring in the Observer pattern.

The idea behind the Observer pattern is pretty straightforward: whenever something important happens (in this case, a command is executed), we can notify a list of listeners or "observers" without the command needing to know who those listeners are or what they do with the information. This fits perfectly with how the Command pattern already works. Once a command is executed, we can notify any registered observers that the command was completed and give them details like the new position and direction.

To get this working, I started by creating a CommandObserver interface. It had just one method—onCommandExecuted()—which takes in the executed command and the current position.

src > main > java > ca > mcmaster > se2aa4 > mazerunner > CommandObserver.java > ...

```java
package ca.mcmaster.se2aa4.mazerunner;

public interface CommandObserver {

  void onCommandExecuted(Command command, Position position);

}
```

Then, inside my CommandInvoker class (which already handles executing the commands), I added a list of observers and made sure that every time a command was run, those observers would get notified. To keep the implementation flexible and clean, I also created a SilentCommandObserverclass, which implements the observer interface but doesn't actually do anything. This was useful because the assignment asked for no extra output in the terminal, but I still wanted to show that the observer system was in place and functional.

src > main > java > ca > mcmaster > se2aa4 > mazerunner > SilentCommandObserver.java > Java Language Support > SilentCommandObserver

```java
package ca.mcmaster.se2aa4.mazerunner;

public class SilentCommandObserver implements CommandObserver {
  @Override public void onCommandExecuted(Command command, Position position) { // This observer intentionally does nothing.
}
}
```

Adding this pattern gave me a lot of freedom. It meant I could completely separate command logic from any side effects like logging, metrics, or future visualization. If I wanted to later track all moves in a file, update a GUI in real-time, or even count steps for analytics, I wouldn't have to touch a single line of movement code—I'd just add another observer. It also helped keep things more consistent and maintainable. By centralizing where notifications happened, I reduced the chance of forgetting to add logging or checks in certain places. During testing, I wrote a special TestObserver that collected all command execution events. This let me verify not just that a move or turn changed the position as expected, but also that the observer system was working correctly and got notified every single time. It gave me confidence that the observer integration was reliable and ready for future extensions.

In the end, implementing the Observer pattern alongside Command felt like a natural fit. It reinforced the separation of concerns in the code and made the system easier to understand, modify, and grow. With both patterns working together, the codebase feels way more modular and ready for future upgrades like undo functionality, logging support, or even animated replay of the solution steps.

3. Elaborate on the third design pattern. Why and how would you use it in your code? Explain the changes the pattern would introduce in your code base. About 1 page.

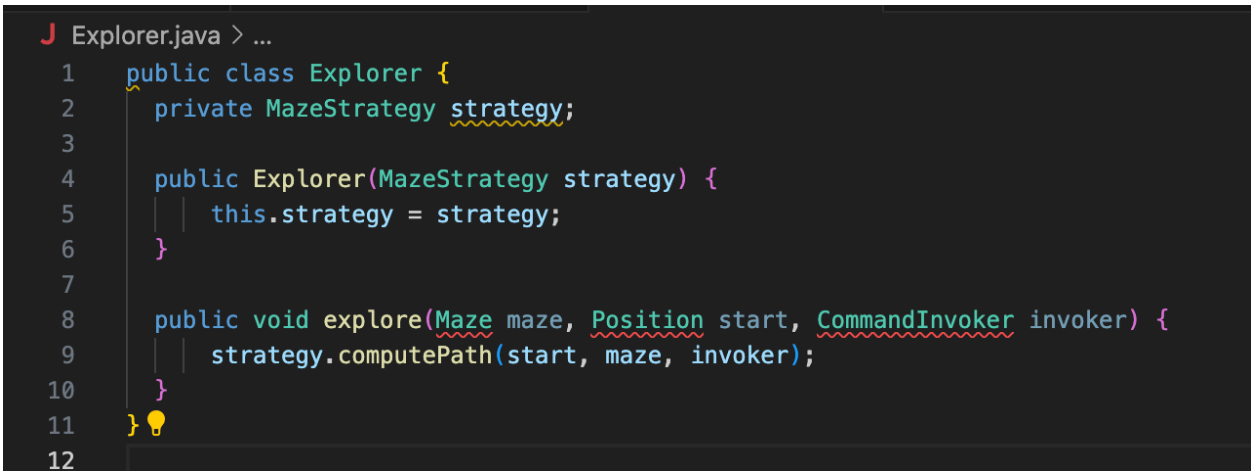
In the future, it would be a good idea to introduce the Strategy design pattern into my codebase. This pattern would make it possible to support multiple maze-solving algorithms, not just the right-hand rule I have hardcoded right now. The strategy pattern is all about defining a family of algorithms, encapsulating each one in its own class, and allowing the algorithm to vary independently from the rest of the code that uses it. For this project, it fits perfectly—because the maze-solving logic can easily be abstracted and swapped out as needed.

Currently, the Explorer class has a method called rightHandAlgorithm(), and that's the only maze-solving logic available. But in the future, if I wanted to try something like a left-hand algorithm, BFS, DFS, or even A*, I would have to write those directly in the Explorer class or build a bunch of if/else logic, which would get messy fast. By introducing the Strategy pattern, I could avoid all that. Instead of locking my code into one specific algorithm, I would define a MazeStrategy interface with a method like computePath() (see figure below).

```
1
2   public interface MazeStrategy {
3       void computePath(Position position, Maze maze, CommandInvoker invoker);
4   }
5
```

Then, each solving algorithm would go into its own class like RightHandStrategy, LeftHandStrategy, or AStarStrategy—and each of those would implement the MazeStrategy interface. This fully follows one of SOLID principles(interface segregation principle), as the higher level code depends on the interface instead of concrete implementation.The Explorer class would no longer care which strategy is used; it would simply hold a reference to a MazeStrategy object and call computePath() on it. This change makes the

explorer way more flexible and removes the need to ever touch its internal code just to switch solving algorithms (see figure below).



That one small change would allow me to easily support different algorithms just by passing in a different strategy object. I wouldn't have to rewrite anything in Explorer, and the different algorithms would be cleanly separated into their own files.
Adding the Strategy pattern would also make testing easier. Since each strategy is its own class, I could write unit tests for each one in isolation without having to touch or even initialize the full Explorer class. It would also make benchmarking different algorithms more practical, since I could run them side-by-side and compare their outputs and performance.

4. Reflection. What did you learn about design patterns? Would you use them in the future? Would you have designed your system differently had you known the design patterns earlier? About 1 page.

Working on Assignment 3, I learned a lot about the actual implementation of the design patterns we talked about in lecture. Seeing them applied in real code helped me understand not just what they are, but *why* they matter and when they actually make a difference. Refining Assignment 1 gave me the chance to look back at the code I had previously written with fresh eyes. Since I already had a solid understanding of how the original code worked, it was easier to spot where things could be improved, and where a design pattern could clean up structure or reduce complexity. With the knowledge I've gained now, I can quickly identify potential design patterns I should apply to my code, and use them to make the structure more maintainable, extendable, and cleaner overall.

Design patterns, in a way, act like templates or blueprints for solving common problems in software design. They give you battle-tested solutions for situations that pop up all the time—like needing to decouple components, or support different behaviors without rewriting everything. When applied properly, they help reduce potential code smells, make logic easier to test, and allow the system to evolve without major rewrites. In the case of this maze runner project, there's a clear need for a chain of actions like move() and turn() that the explorer must perform. This kind of problem fits really well with the Command pattern. By first creating

a Command interface and then building separate classes for different actions, the maze runner becomes much easier to follow, test, and potentially undo or replay steps—something that would've been a nightmare with just raw method calls and tightly coupled logic.

In the future, I know I'll be using more design patterns in my projects. Now that I understand how they work and why they're useful, I see so many situations where they could come in handy. Whether it's designing more flexible systems, making testing easier, or just keeping code organized when projects start to grow—design patterns help manage complexity in a smart way. They give structure to your code and let you solve new problems without constantly reinventing the wheel. Even if a pattern isn't needed right away, just knowing it exists helps me plan better and write more thoughtful designs from the start.

Looking back, I would've definitely designed my system differently if I had known about these patterns earlier. For Assignment 1, I spent about a day or so planning before jumping into the code, but the design was still really basic. I made a few rough UML diagrams, but I didn't think much about long-term structure. Most of the work was done in an "XP" (extreme programming) kind of way—just coding to pass test cases and make things run. As a result, a lot of the code had limitations. It wasn't very flexible, and adding new features felt harder than it should've been.

If I had learned about design patterns back then, I probably would've introduced them in the design phase itself. For example, I would have created a Strategy interface from the beginning instead of throwing all the algorithm logic into the Explorer class. That change alone would've made it so much easier to support multiple solving algorithms later on. It would also follow the interface segregation principle by ensuring that higher-level logic depends on abstraction instead of concrete implementations. I would have also brought in the Command and Observer patterns earlier, so they would feel like a natural part of the system instead of add-ons. Observer, in particular, could have been a helpful debugging tool during development, since it would let me track movements and state changes without cluttering up the core logic or output.

Overall, having the opportunity to refine my code from Assignment 1 gave me a much deeper understanding of how and when to use design patterns. It's one thing to learn about these patterns in theory, but actually applying them to real code shows their true value. Now that I've seen what they can do, I'll be a lot more intentional about using them in future projects—from the very start.