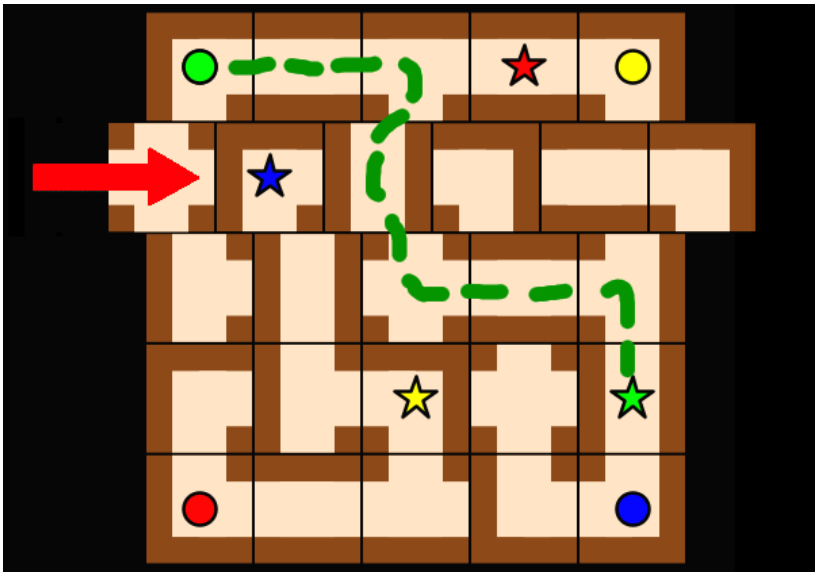
CS144 Project Specification Document (PSD)

Version 1.0

20 July 2022

Stellenbosch University

Contents

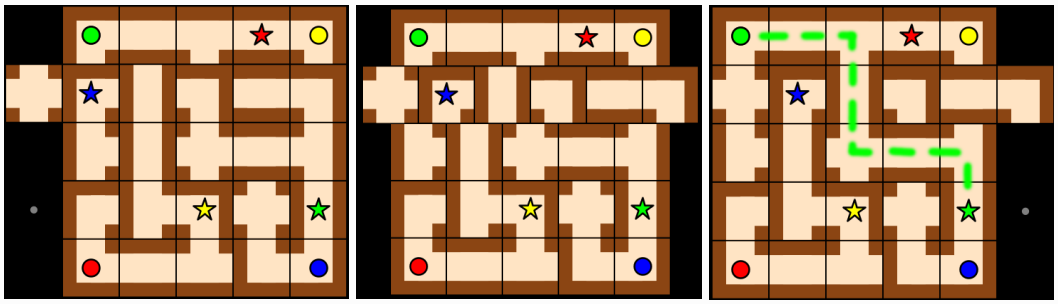


1 Introduction to Moving Maze

In the game of **Moving Maze**, players must traverse an ever-changing maze and race other players to collect ancient valuable **relics**. The aim of Moving Maze is to be the first player to collect all their relics and make it back out.

The **maze** is a complete grid of **tiles**, and each has multiple entry points. These hallways, bends, and junctions on the tiles chain them together to form the pathways of the maze. One additional tile (called the **floating tile**) sits outside the maze. As part of a player's turn, they slide the floating tile into a row or column of the maze. This offsets the entire line of tiles, connecting/disconnecting tiles' entry points and destroying/creating pathways in the process. The tile that gets pushed out becomes the new floating tile, and the cycle continues. This process is exemplified in Figure 1.

Your project brief is to code a game client for Moving Maze, which will facilitate the game rules as described in this document.



(a) Before the tile is slid in. **(b)** During the insertion. **(c)** After the tile is slid in.

Figure 1: A demonstration of the tile-sliding principle that dominates Moving Maze. At the start of the green player's turn, the board is in the state of 1a. Green wants their adventurer (the circle) to reach the relic (the star), but there is no path on the board between their current location and the relic. To achieve their goal, Green slides the floating tile (on the left in 1a) into the second row of the board, pushing all the tiles in that row to the right (pictured in motion in 1b). No other tiles were moved in the process. The tile that was in the second position of that row has now been moved to the third position, creating a path from the green adventurer to the green relic (as seen in 1c). Green can now reach their relic.

2 How Moving Maze works

The game's rules are explained below. The overall game concept and game loop is laid out in Section 2.1. Further definitions, clarifications, and finer details are given in Section 2.2.

2.1 Game overview

Moving Maze is a four-player game, with each player taking the mantle of a color from among green, yellow, red and blue. A grid of tiles represents the maze. Relics (items which the players must collect) are scattered across the maze. Each player has a representative adventurer (colored according to their players' identities) showing the player's position in the maze. Play proceeds in turns.

Game board setup. Before play begins, the game board (the **tile maze** and the **floating tile**) must be set up. First, place the **tiles** in a grid formation to create the **tile maze**. Then, place the **adventurers** on the four corners of the tile maze:

- Green starts in the top-left;
- Yellow starts in the top-right;
- Red starts in the bottom-left; and
- Blue starts in the bottom-right.

Player turns. The game's turns begin here. Players take their turns in the order listed above (green, yellow, red, blue). A turn has two phases: sliding and moving.

1. **Sliding phase.** Take the floating tile, rotate it however you like, and push it into the maze from one edge. This moves the entire line of tiles (including the adventurers and relics on those tiles) and pushes out a tile on the other side; this tile becomes the new floating tile. Sliding is mandatory.
2. **Moving phase.** Move your adventurer across the paths that exist in the tile maze. If you collect a relic, the moving phase ends immediately. Otherwise, you may end the moving phase whenever you want. Moving is therefore technically optional.

Win condition. Play proceeds in turns until a player's adventurer has collected all their relics and has returned to their starting position in the tile maze.

2.2 Definitions

2.2.1 Tiles

A tile has four sides, and each side may be open or closed (although a minimum of two must be open). All the tile's open sides are connected to each other and form one pathway / junction. A tile may have a **relic** on it, and that relic stays on that tile (regardless of how the tile moves). Tiles cannot have multiple relics — either one relic, or no relic at all.

2.2.2 The tile maze

The tile maze is a complete grid of tiles. The grid can be rectangular and not just square, but both its width W and its height H (in tiles) will always be odd numbers and be constrained to $3 \leq W \leq 9$ and $3 \leq H \leq 9$. Rows and columns are numbered from 1 onward (not 0). The odd-numbered rows and columns are fixed in place and cannot be **slid**.

2.2.3 The floating tile

The floating tile is separate from the tile maze and does not contribute to any pathways in the maze while it is floating. As part of the sliding phase of a turn, the active player can change the floating tile's orientation as they like before sliding it into the **tile maze**.

2.2.4 The game board

The **tile maze** and **floating tile** are collectively referred to as the game board.

2.2.5 Relics

Relics are items that players must reach with their adventurers to collect. Each player has the same number of relics scattered across the tile maze. This number (which is denoted as K) is known at the start of the game. Therefore, there is $4K$ relics in total at the start of the game. Furthermore, $K < 10$, always.

Relics have a color indicating which player must collect them. A player cannot collect a relic that doesn't match their color. In addition, relics must be collected in a certain order, and players cannot collect relics outside of this order. The next relic that a player must collect is called that player's **active relic**. Collecting relics in order is enforced by hiding the relics that aren't next in line to be collected. Therefore, only one relic of each color (four in total) is visible and the rest are hidden at any point in time — unless one or more players have already collected all relics and are heading back to their starting corner, in which case there would be less than four relics visible.

Relics are tied to tiles. They are completely attached to their host tiles, and they always move together (even if a host tile becomes the **floating tile** by being **slid** out of the maze).

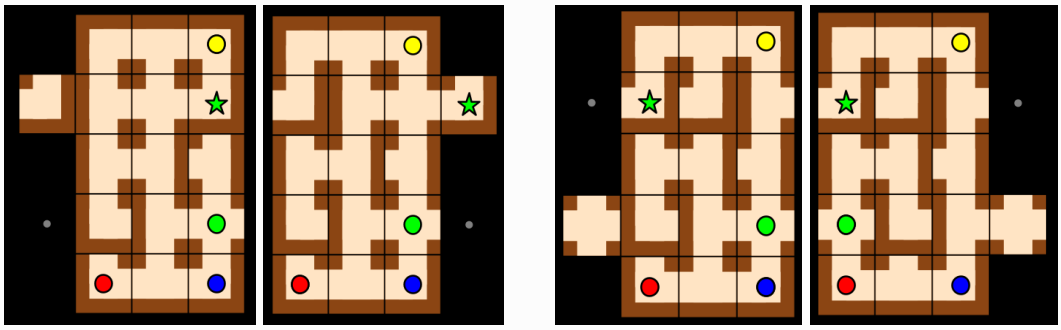
2.2.6 Adventurers

Adventurers are the player's representatives in the maze. Adventurers are colored according to their players' identities (green, yellow, red, blue). At any given point in the game, an adventurer is on one specific **tile** — adventurers cannot occupy multiple tiles or edges between tiles. Multiple adventurers can occupy the same tile. Adventurers can pass through tiles that other players already occupy.

2.2.7 Sliding the floating tile into the tile maze

As stated in the high-level game rules, the first phase of a player's turn involves sliding the **floating tile** in any orientation into the **tile maze**. **Adventurers** and **relics** on tiles that are moved during this slide, are moved together with their host tiles. See Figure 2 for a visual explanation of these rules.

- If an **adventurer** is moved off the board due to a slide, the adventurer is moved to the opposite side of the maze, onto the newly placed tile. This can trigger a relic collection (Section 2.2.9) if the newly inserted tile hosts the displaced player's active relic.
- **Relics** always stay on their host tiles, including on tiles that become the floating tile. A relic on the floating tile will return to play when the tile is slid back into the maze.



(a) Before slide #1 **(b)** After slide #1 **(c)** Before slide #2 **(d)** After slide #2

Figure 2: How relics and adventurers are treated when slid off the board. In 2a, the floating tile is about to be slid into the second row of the tile maze. 2b shows the game board directly after that slide. The green relic (the green star)'s tile has now become the floating tile, taking the green tile with it. Some turns later, in 2c, the tile hosting the green relic has been slid back into the board, and the floating tile is about to be pushed into the fourth row, which would push out Green's adventurer (the green circle). In 2d, the slide of 2c has been executed and the green adventurer has immediately been moved to the opposite side of the maze.

The floating tile can only be slid into even-numbered rows and columns, thereby fixing some tiles in place and making them safe spots to stand on. The floating tile cannot be slid into the last position where it exited, i.e. a slide cannot undo the slide that was performed immediately before it.

2.2.8 Moving adventurers across the tile maze

In the moving phase, the current player may move their adventurer through the maze's pathways. For an adventurer to move between two tiles, they must be directly adjacent to each other, and both tiles' sides on which they touch must be open. See Figure 3 for examples of how tiles can be traversed based on these rules.

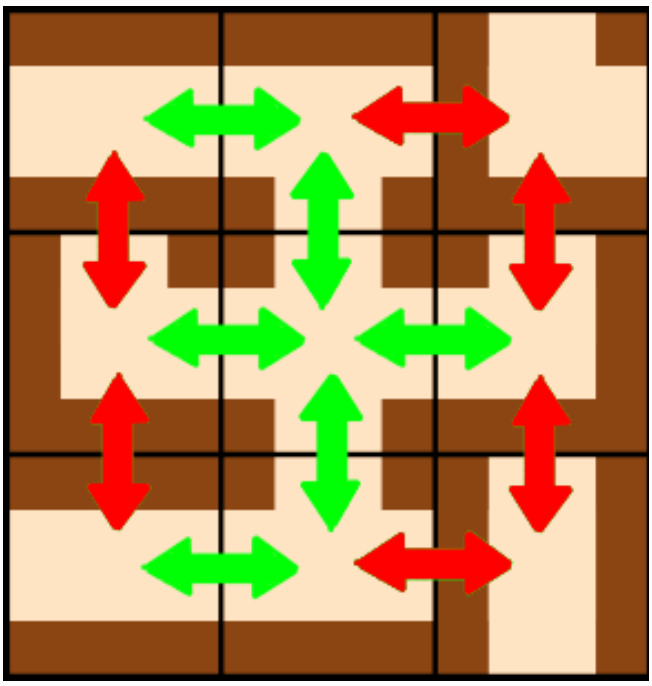


Figure 3: A small game board that shows which other tiles can be reached from each tile.

You can opt not to move your adventurer at all. As stated in Section 2.2.6 on adventurers, you can move onto or through a tile on which another adventurer is standing. You must end your move phase solely on one tile — no standing on edges.

2.2.9 Collecting relics

If an adventurer ever stops on a tile that hosts the active **relic** of their color, the relic is collected immediately and the player's move phase ends. The next relic in that player's collection order becomes the active relic and its location is revealed.

3 Broad program description

Your program is a game client that facilitates the game of Moving Maze according to the game rules laid out in Section 2. There are two visual modes: text mode (defined in Section 4) and GUI mode (defined in Section 5). There are some commonalities between the two modes that are defined in this section.

3.1 Execution

Your program's main file (`MovingMaze.java` — see Section 7.3) will be executed as follows:

```
$ java MovingMaze <filename> <visualmode>
```

3.2 Arguments

Certain parameters of the game must be defined at the start of the game — this is done with program arguments. The arguments your program receives must be validated — see each argument's description for more information.

The game board file name. The first argument of the program is the name of a game board file that contains a game board encoded as plain text. The argument is a string with the exact name of a game board file on the system's hard drive. This game board file must be read and the game must be set up according to the contents of this file. See Section 3.3 to learn how this works.

The game board file name must be validated as follows:

- The file must exist. If it does not, the program must report so with the message `The game board file does not exist.` and immediately exit.
- (Third hand-in only) The contents of the game board file must be validated to ensure internal consistency — see Section 3.3.3.

The visual mode. The second argument of the program determines whether the game launches in text mode or in GUI mode, and can be one of two options: `text` or `gui`. If `text` is passed to this argument, the game launches in text mode (Section 4); if `gui` is passed to this argument, the game launches in GUI mode (Section 5). If an unknown option is given to this argument, the program must report so with the message `Unknown visual mode.` and immediately exit.

3.3 Loading a game board from a text file

To set up a game of Moving Maze, pre-configured game boards (which are the tile maze and the floating tile) are loaded into the program with **game board files** (GBFs). The first argument used when executing the program will be the name of a GBF. How a game board is encoded in such a GBF, is explained below.

3.3.1 GBF encoding scheme

A GBF is a plain-text file with N lines.

- Line 1 indicates the width and height (measured in number of tiles) of the **tile maze** as two integers with a single space between them.
- Line 2 is a single integer K that indicates how many **relics** of each color there are in total across the maze.
- Line 3 describes the **floating tile** (see Section 3.3.2 on how tiles are encoded).
- Lines 4 to N describe the rows of the **tile maze**. One line represents one row. Each line describes the tiles in that row with its six characters each (see Section 3.3.2). These strings are separated by single spaces.

Figure 5 exemplifies the contents of a GBF and the corresponding game board.

3.3.2 Tile encoding

When loading the game board from a text file, tiles are encoded as a string of six characters. The first four characters are four bits (`1`s and `0`s) indicating whether the tile is open on its four sides (`1` is open and `0` is closed), starting at north and proceeding clockwise. The fifth and sixth character represent the relic on the tile. The fifth character indicates the color of the player that must collect it, and the sixth character indicates the position of the relic in the collection order, starting at 1 and incrementing by 1. If there is no relic on a tile, the fifth and sixth characters will both be `x`s.

Figure 4 exemplifies some tiles and their six-character encoded strings.

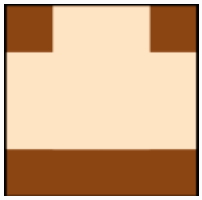
3.3.3 Validation

For the first and second hand-ins, no validation of game board files are necessary. You can assume that the lines of any game board file given to your program will always follow Section 3.3.1's scheme and that all its tiles will be encoded as per Section 3.3.2. You can assume that the dimensions of the board are correct. You can further assume that there are no relics with duplicate numbers, that there are exactly K relics of each color present, and that the relic numbers begin at 1, increment by 1, and end at K.

For the third hand-in, you must perform validation of the above-mentioned criteria, and ensure that the game board file is internally consistent. If anything is incorrect, the program must state `The game board file is inconsistent.` and immediately exit. The exact reason for invalidity does not need to be listed.

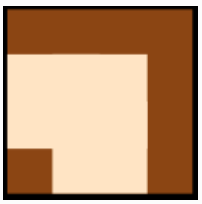
3.3.4 Examples

Examples of tiles and how they are encoded are given in Figure 4. An example of a GBF and the resulting game board is shown in Figure 5.



(a) `1101xx`

(b) `1010g1`

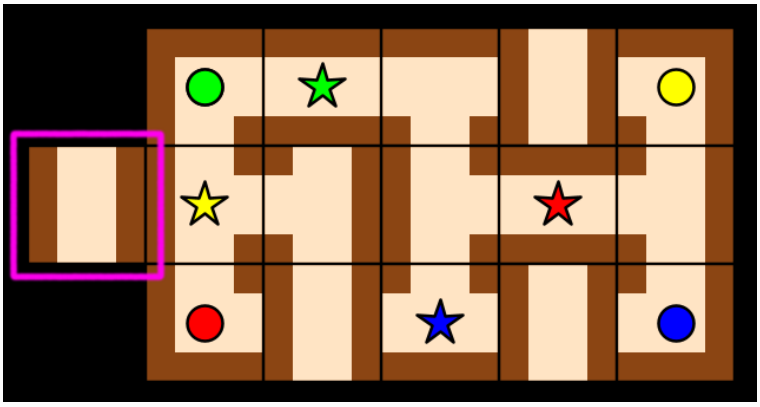


(c) `0011r2`

Figure 4: Three examples of tile encoding. Note that the visual representations assume that the game has just started; therefore, 4c has no relic drawn. If the red player had already collected red relic #1, then 4c would have the relic drawn as a red star.

```
5 3
2
1010xx
0110xx 0101g1 0111b2 1010y2 0011xx
1110y1 1011r2 1110xx 0101r1 1011xx
1100xx 1010xx 1101b1 1010g2 1001xx
```

(a) The contents of an example game board file.



(b) The resulting game board. The floating tile is highlighted in purple.

Figure 5: The contents of an example game board file (GBF) (5a), and the actual game board that it represents (5b). Note:
a) The correspondence between the dimensions of the tile maze and the first line of the GBF;
b) How each tile corresponds to its six-character string encoding; and
c) How only the relics numbered `1` (which would be active at the start of the game) are shown (those marked `2` are hidden, as they are inactive).
The player locations are not a result of the GBF, as their starting positions are always the same.

4 Text mode

Text mode is played in the console. User inputs (such as for rotating/inserting the floating tile and moving adventurers) are given to the program via strings to the standard input stream. Program outputs (the board and messages) are printed to the console. This section defines the text mode's requirements and its flow of inputs and outputs.

4.1 Game start

At the start of the program, the game start banner must be printed. It consists of four lines, which are described below. An example of a game start banner for a game with two relics per player is shown in Figure 6. After the game start banner is printed, the game board must be printed (Section 4.2).

- A line of 50 dashes, `-- ... --`;
- The text `Moving Maze`;
- The text `Relic goal: K`, where K is the number of relics of each color scattered around the tile maze; and
- Another line of 50 dashes.

```
--------------------------------------------------
Moving Maze
Relic goal: 2
--------------------------------------------------
```

Figure 6: The game start banner that is printed at the beginning of the program.

4.2 The game board

The game board is printed by combining certain characters to represent the tile faces, tile borders, adventurers, and relics. The game board is printed as shown in Figure 7. An example of a tile is shown in Figure 8.

The floating tile is printed below the tile maze, with a single line-break separating them and an additional empty line above the maze and below the floating tile. Both constructs hug the left side of the terminal, but the floating tile has no row numbering, so it looks more left-aligned. The remainder of this section clarifies how the text representation is constructed.

Special characters

The special characters that are used to display the game board do not appear on a standard keyboard. They are provided in a plain-text file on SUNLearn for convenience.

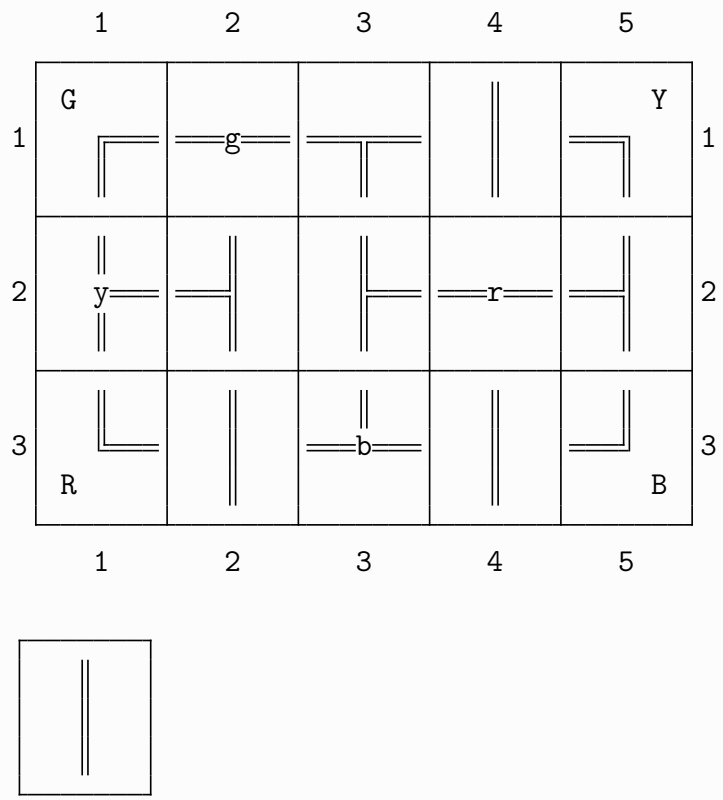


Figure 7: An example of what a typical game board would look like in text mode at the start of the game. The **tile maze** is the structure at the top, and the **floating tile** is the single tile at the bottom. A single line break separates the two constructs. The **adventurers** (represented by uppercase letters) are at their starting corners of the maze. The active **relics** (indicated in lowercase) are shown, while the inactive relics are hidden (as per Section 2.2.5). The row and column numbers must be printed with the board — there is no horizontal or vertical space between the numbers and the borders.

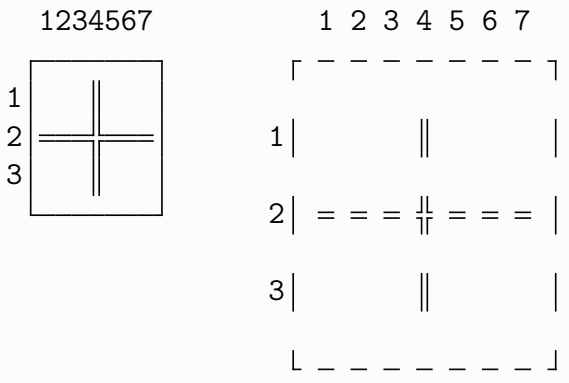


Figure 8: A tile that is open to all directions. On the left is its normal text representation. On the right, the representation is exploded to visualize the component characters better. The numbers are to aid in visual understanding and must not be printed.

4.2.1 Tile faces

In text mode, a **tile**'s face is represented with three lines of seven characters each. An additional two rows and two columns (for each side of the tile) make up the boundary of/between tiles (more on that in Section 4.2.2). See Figure 9 for a diagram of how a tile face is constructed. There are three aspects of tile faces to consider: **pathways**, **relics**, and **adventurers**.

> **❓ Why are tile representations so wide?**
>
> Remember that in the majority of consoles, the characters are much taller than they are wide. Even though the tile faces are much wider in characters, they are visually close to square.

Pathways. The central (fourth) character of the face's top and bottom rows, as well as the entirety of the middle row, visualize the pathways that exist on the tile according to which of the four sides are open. The special characters used in the pathways are shown in Figure 10. An example of a tile's pathways using these characters is shown in Figure 8. Figure 11 demonstrates how each of the 11 tiles that can be found in Moving Maze are represented using these characters. Note that these tiles contain no relics and there are no adventurers on them.

Relics. If a tile hosts an active relic, the central character of the middle row (which usually visualizes the junction) is overwritten by a lowercase initial of the color of that relic. See Figure 12. Remember that active relics are the ones next in their player's collection order — if a relic is not active, it must not be drawn.

Adventurers. The second-from-the-outside characters of the top and bottom rows are uppercase initials of any adventurers present on the board. Because of the limitations of text mode, adventurers are not printed to look as if they are *in* the pathways of the tile. Therefore, if an adventurer is printed on a tile, they are considered to be on the pathways of that tile. Multiple adventurers can appear on one tile — the initial of each adventurer is printed in the corners of the tiles that correspond to the starting locations of the adventurers in the maze (i.e. green in the top-left corner, yellow in the top-right corner, etc.). See Figure 13.

Note that an active relic is immediately collected when their correspondingly colored player is on the relic's host tile (as per Section 2.2.9). Therefore, an adventurer and a relic of the same color will never be printed together on the same tile.

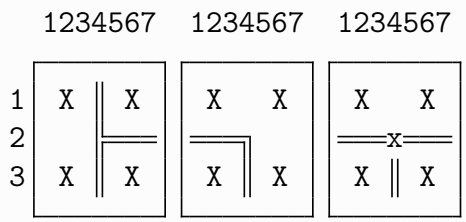


Figure 9: A diagram of how the tile face text representation is constructed in text mode. A tile face is made up of three rows of seven characters. The pathways and junction of the tile (indicating which sides of the tile are open) are drawn with special characters. In the center of the tile, a junction character must be printed, as exemplified in the first and second tiles. If the tile has a relic, the lowercase initial of the color of the relic must be printed instead of the junction, as exemplified in the third tile. In the second and sixth columns of the first and third rows, the uppercase initials of the adventurers standing on the tile must be printed. There is an order involved in the adventurers — see Figure 13.

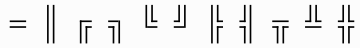


Figure 10: The characters with which the pathways on tiles must be printed.

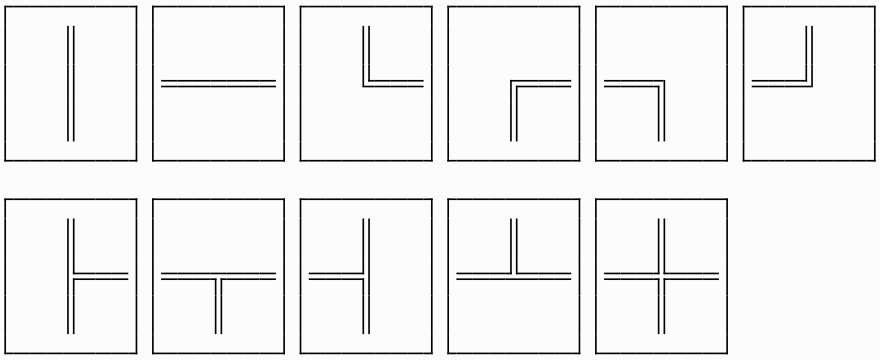


Figure 11: The eleven possible tile configurations that can be found in Moving Maze in their text representations. These do not include relics or adventurers.

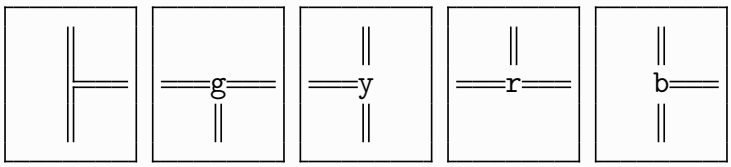


Figure 12: In the leftmost tile, there is either no relic at all or it is hidden. In the remaining four tiles, there is an active relic on each of them — green, yellow, red, and blue, respectively.

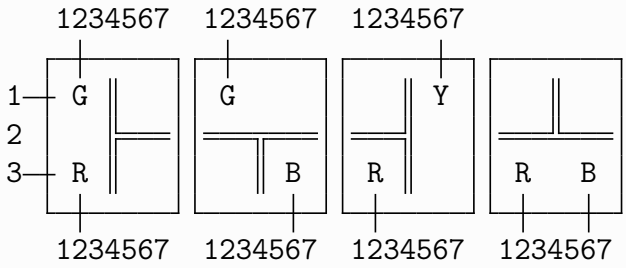


Figure 13: Examples of adventurers printed on a tile face. Note that they are printed in columns 2 and 6 (not 1 and 7). Also note that their locations on the tile are always the same and correspond to where they would start on the game board. Green and red occupies the first tile. Green and blue occupies the second tile. Yellow and red occupies the third tile. Red and blue occupies the fourth tile. The numbers and the lines pointing to them are meant to aid in visual understanding — they must not be printed.

4.2.2 Tile borders

Whether printed as a single tile (such as with the **floating tile**) or as part of the **tile maze**, tiles have borders around them. The border is represented by the set of characters displayed in Figure 14 (which are also provided on SUNLearn).

Figure 14: The characters with which tile borders must be printed.

When placed adjacent to each other, tile borders are merged, requiring the last five characters in Figure 14. In addition, the tile maze's rows and columns must be indicated with numbers that are printed directly against the tile maze's borders on all sides. See an example in Figure 7 to compare a single tile to a tile grid, and to see the row and column numbers.

4.3 The scoreboard

The scoreboard reports how many relics have been collected by each player. The scoreboard is five lines long and adheres to the format explained below.

In the first line of the scoreboard, the text `Relics collected /K:` (where K is the number of relics per player) is printed. The second to fifth lines each name a player and the number of relics that the player has collected so far. The list is in order of the turns of play, and not in the order of score. The lists uses `-` as bullets, with a single space between the bullets and the player names. Furthermore, the scoreboard is spaced so that the scores are vertically aligned with each other, and so that yellow (the player with the longest name) has a single space between their name and their score. Figure 15 shows an example of a scoreboard.

The scoreboard is printed at the end of each player's turn (see Section 4.4.3) and when the game is quit (see Section 4.4.4).

```
Relics collected /4:
- Green  3
- Yellow 2
- Red    4
- Blue   1
```

Figure 15: An example of a scoreboard. There are four relics of each color scattered around the maze, as reported at the top of the scoreboard. Green has collected three of their relics, yellow has collected two, red has all four, and blue only one.

4.4 User inputs and program messages

4.4.1 General output rules

There will never be any empty lines between any lines of the program's outputs, with the exception of the empty line above the game board, the empty line below the game board, and the single empty line that separates the tile maze from the floating tile.

4.4.2 Prompts to the user

When a user input is prompted for any input, the following format must be adhered to. First, the active player's color (who must respond to the prompt) is printed in square brackets, with the first letter in uppercase and the rest in lowercase. The prompt message is printed after, and it always ends with a colon. In the next line, a greater-than sign is printed, with a single space after it. After that space, the player can enter their move to standard input (which can be accessed however is preferred). There are no empty lines between repeated prompts. See Figure 16 for a visualization of the format and an example of an actual user prompt.

```
[Player] Prompt:
> // Input goes here
```

```
[Green] Move your adventurer:
>
```

Figure 16: The format of a text mode prompt (left), and an example thereof (right).

4.4.3 Turn structure and resolving actions

Gameplay follows turns in the order of green, yellow, red, and blue. Each turn is split into two phases — the sliding phase and the moving phase. After any (legal) input is given, the action it represents must be reported as successful, the action must be resolved, and the game board must be reprinted.

The sliding phase. In this phase, the active player is allowed to rotate the floating tile and insert it once they are ready. The prompt message for this phase is `Rotate and slide the floating tile:`. Valid responses are:

- A single `r` — To rotate the floating tile once to the right (clockwise). Report the rotating action as successful with `Rotating right.`.
- A single `l` — To rotate the floating tile once to the left (counterclockwise). Report the rotating action as successful with `Rotating left.`.
- A "sliding indicator" string that describes where the floating tile must be inserted. Examples of sliding indicator strings include `n4`, `s6`, and `e2`. This string consists of:
    - A single character representing the side into which floating tile must be inserted — the options are `n`, `e`, `s`, and `w` (indicating the northern, eastern, southern, and western sides, respectively).

- A single integer representing the row number (if the character is `e` or `w`) or column number (if the character is `n` or `s`) into which the floating tile must be inserted. This indexing corresponds with the numbering on the game board's edges, i.e. it starts at 1 for the first row/column.

Report the action as successful with `Inserting at X`, where `X` is the sliding indicator string.

The moving phase. In this phase, adventurers are moved in single steps with single-character inputs that indicate the four compass directions. The prompt message for this phase is `Move your adventurer:`. The options are `n`, `e`, `s`, and `w`, indicating movement in the northern, eastern, southern, and western directions, respectively.

Each movement step that is input in this way traverses a one-tile-long step directly in the indicated direction. The game must not attempt to find any other path through any tiles other than the current tile and the destination tile. If there is no direct connection via the connected tile sides that are both open (purely between the current tile and the destination tile), the step cannot be taken. See Figure 17 for an example of such an illegal move.

The current player can voluntarily end this phase with the `done` command. This phase also ends when an adventurer reaches its player's active relic, at which point that relic is also collected.

When a move of an adventurer was successful, report the action as successful with `Moving X.`, where `X` is the full direction in lowercase (e.g. `west`).

(Third hand-in only) Pathfinding moving method. The current player can also move their adventurer to *any* tile that is connected via pathways to the player's tile. This functionality is triggered when the player enters a pair of integers separated by a single comma (e.g. `3,1`) which represents the X- and Y- coordinates (column and row numbers, starting at 1) of the desired new location to the movement prompt. Report the action as successful with `Moving to X,Y`, where `X` and `Y` are the tile coordinates as they were entered. Taking a path *past* a relic (i.e. not stopping exactly on the relic's tile) will not trigger that relic's collection. Using this movement method does not end the move phase — multiple coordinates (and/or direction characters) may be given before the turn is ended with `done`. See Figure 17.

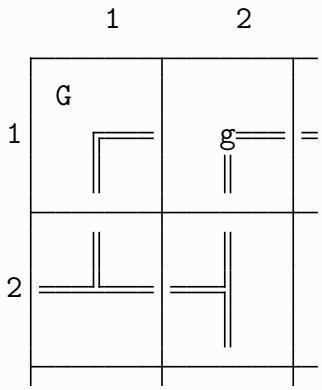


Figure 17: Green wants to reach the green relic on the tile to the east. They enter the move `e`, but the move is illegal because the tiles are not immediately connected via pathways to Green's tile's east, even though there is a detour to the south. To get to the tile to the east of the current tile, Green will have to take the long path with manual steps, i.e. `s`, `e`, `n`. Keep in mind that the player *can* move to the relic-hosting tile with the third hand-in's pathfinding functionality, by entering `2,1` to the moving prompt.

Relic collection. When a relic is collected (either because an adventurer was moved by its player or because an adventurer was displaced by a slide), the message `X has collected a relic.` must be printed. If the last of the player's relics was just collected, the message `X has all their relics.` must be printed as well. In both messages, `X` is the color of the player who collected the relic and is in title case, e.g. `Blue`. Immediately after that, the scoreboard must be printed.

The end of a turn. Regardless of whether a turn is ended by a relic collection or by the active player's own volition, an end-of-turn message must be printed. If a relic is collected, the end-of-turn message comes after the relic collection message. The end-of-turn message is simply `End of X's turn.`, where `X` is the active player whose turn just ended and is in title case (e.g. `Blue`). The scoreboard must be printed directly afterwards.

Upon a player winning. A player has won if they have collected all their relics and have navigated their adventurer back to their starting corner. If this is achieved, the winner should be announced with the message `X has won.`, where `X` is the color of the winning player and is in title case (e.g. `Blue`). After announcing the winner, the scoreboard should be printed and the program should terminate immediately after.

Move legality. For the first and second hand-ins, it can be assumed that any inputs given to the program will always conform to the schemes described above. However, the legality of inputs must still be assessed. If any illegal input is given, the appropriate message must be printed and the prompt repeated without reprinting the game board. The illegalities and accompanying messages are listed in Table 1.

(Third hand-in only) Input validation. For the third hand-in, inputs must be validated to ensure that they conform to the formatting schemes described above (which was as-

sumed to be true in hand-ins 1 and 2.). If any invalid input is given, it must be reported as invalid with the message `Invalid input.`, and the prompt must be repeated without reprinting the game board.

Table 1: Illegal actions in text mode and the error messages that should be printed when they are attempted. Illegal actions should of course not be resolved.

Illegal action	**Error message**
Sliding the floating tile into an odd-numbered row/column	`Cannot slide into odd positions.`
Sliding the tile into the position where the floating tile just exited	`Cannot slide into last exit point.`
Trying to move off the board through an open side in the current tile	`Cannot move X: off the board.`, where `X` is the full direction in lowercase (e.g. `west`)
Trying to move to a tile to which there is no connection (as per Section 2.2.8), or trying to move in a direction in which the current tile is not open	`Cannot move X: no path.`, where `X` is the full direction in lowercase
(Third hand-in only) Trying to use pathfinding to a tile that is not accessible	`Cannot move to X,Y: no path`, where `X` and `Y` is the desired position to pathfind to (in the same order as they were given).

4.4.4 Quitting the game

At any point in the game, any player can enter the command `quit` to whatever prompt is currently open. If this occurs, the message `Game has been quit.` must be printed, the scoreboard must be printed directly after, and the program must then terminate. See Figure 18 for an example.

```
[Green] Move your adventurer:
> quit
Game has been quit.
Relics collected /2:
- Green  0
- Yellow 0
- Red    0
- Blue   0
```

Figure 18: The behavior to be followed when the game is quit with the `quit` command.

5 GUI mode

The GUI mode's requirements are lax to grant more creative freedom to the student. Note that the GUI is only required for the third hand-in (see Section 6).

5.1 Visual elements

The visual elements of the GUI must be clear, unambiguous, and concise.

- All physical elements of the game (which are the tiles in the tile maze, the floating tile, the adventurers and the relics) must be drawn at all times.
- The tiles must be squares and the pathways on them must be clear.
- The tiles must have some space between them (to show where one tile begins and another ends).
- Where adventurers occupy the same tile, the active adventurer must be visible. This can be achieved by drawing it on top of other adventurers (if they overlap) or by drawing adventurers in slightly different locations on tiles (to prevent overlap).
- The players' scores must be displayed accurately at all times.
- The current player's color and the current turn's phase (slide vs move) must be indicated at all times.

5.2 Player actions

The active player must be able to perform any action that is allowable at the current point of the game. These actions are:

- Rotating the floating tile (clockwise or counterclockwise) during the sliding phase;
- Sliding the floating tile into the tile maze (at any admissible location of the tile maze) during the sliding phase;
- Moving their adventurer to connected tiles during the move phase (note that the third hand-in's pathfinding functionality must be enabled).
- Quitting the game.

5.3 Inputs

Inputs can be given with the keyboard or mouse. An exact specification of which user inputs (keystrokes and mouse presses) cause which player action is not given. Marks will be given for sensibly using the mouse and keyboard to allow appropriate player actions.

5.4 GUI example

Figure 19 exemplifies a GUI that meets all the above requirements. The colors, iconography and overall style is not mandatory, and a completely different approach can be followed as long as the requirements listed above are followed.

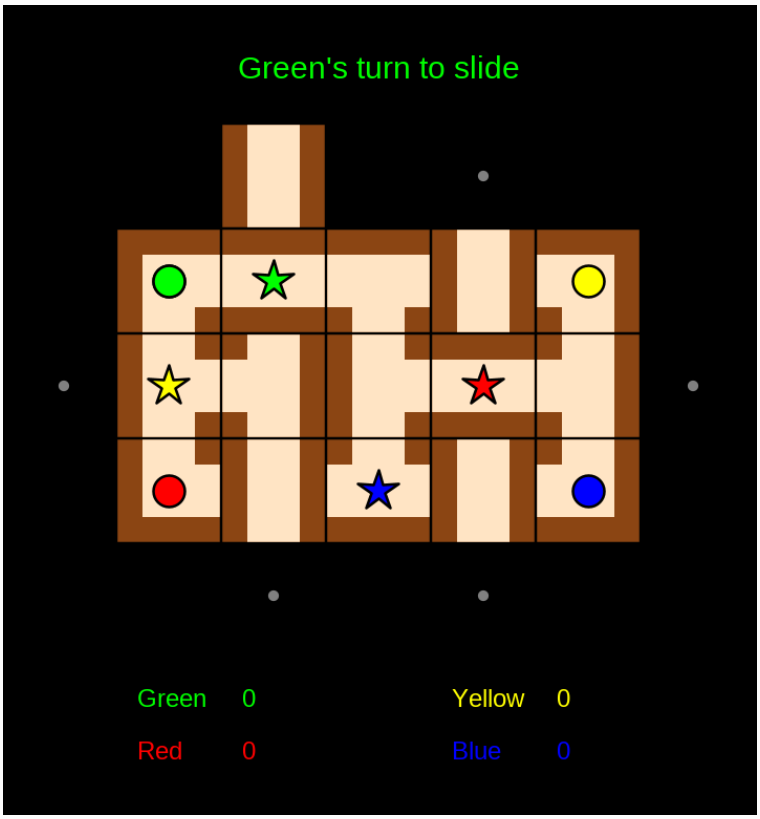


Figure 19: An example of a GUI. The tiles are drawn as squares, with the pathways drawn in a lighter color on their faces. The adventurers are drawn as circles (colored as per their players) and the relics are drawn as stars (also colored correspondingly). The locations where the floating tile can be inserted are shown as gray dots. The players' scores are displayed at the bottom. The active player's turn phase is indicated at the top.

6 Functionality demarcation per hand-in

The project is split into three hand-ins. Their deadlines are indicated on SUNLearn. Each hand-in builds on the work of the previous, meaning that the functionality of earlier hand-ins must be present in subsequent hand-ins. The third and final hand-in will involve the full implementation of the game client with the full game rules, while the first and second hand-ins will only require a subset of the functionalities that have been specified. These are listed in this section.

The following sections explain what functionalities must be implemented and which functionalities need not be implemented in the corresponding hand-ins. If a functionality is not listed as unnecessary for a hand-in, assume it must be implemented.

6.1 Hand-in 1

Text mode only (no GUI). Your program must work in text mode — there is no need to include the GUI mode.

No relics. No relic-related functionality (printing, collecting, scoring) will be assessed in the first hand-in. You can assume that during the assessment of your first hand-in submission, the GBFs your program will be given will contain no relics. All other aspects of the game must still be present. Notably, the scoreboard must still be printed, even though no player's score will ever surpass 0. The game cannot be won, and the game can only be terminated with the `quit` command.

6.2 Hand-in 2

Text mode only (no GUI). Your program must work in text mode — there is no need to include the GUI mode.

Relics. Unlike the first hand-in, your program must include all relic-related functionality for the second hand-in.

6.3 Hand-in 3

Both text and GUI. Your program must work in text mode and in GUI mode.

Pathfinding. As described in Section 4.4.3, the active player must be able to enter two integers (which represent tile coordinates) and have their adventurer move to that tile if a (possibly multi-tile) pathway exists.

Validation of game board files. As per Section3.3.3, GBFs fed to the program must be validated for internal consistency.

Validation of user inputs. As per Section 4.4.3, all user inputs throughout the entirety of the text mode's operation must be validated.

7 Technical specifications

7.1 Java code

Data types. There are no bans on the use of any specific data types. Creating your own data types where appropriate is highly encouraged and will earn marks (see Section 7.2).

Libraries. There are no bans on the use of any libraries (whether built-in or external). Note that we reserve the right to deduct marks should any external library or codebase be used to trivialize large sections of the project. If in doubt, ask the project coordinators to make a judgment on a library — just to be safe.

Code readability. Your code must be easy to read and interpret. Use comments, use descriptive variable names, and apply proper indentation. Doing so will earn you marks.

7.2 Design patterns

Object-oriented programming. A key outcome of Computer Science 144 is the use of OOP. As such, capability with OOP must be demonstrated throughout this project. Marks will be awarded for sensible use of OOP, such as classes, inheritance, interfaces, etc.

The game of Moving Maze was chosen as the project due to its potential in learning and using OOP. However, if you're unfamiliar with OOP, it can be hard to find its use cases. Refer to the game rules and consider: What concepts within the game can be encapsulated as objects? If you had a physical game of Moving Maze in front of you, what physical objects would there be? What behaviors and attributes can be combined and which should be separate, and how can they interact to facilitate the game while constituting a clean codebase?

Here's one possible way to do it...

- A tile is a physical object, and the game features multiple of them. Why not make a `Tile` class?
- A relic has a color and a number. You could store these values as primitives part of the `Tile` class, but there are unique behaviors that relic'd tiles must follow. Make a `Relic` class to encapsulate this and take it off the shoulders of the `Tile` class. Then, you can give your `Tile` class a `Relic` field (which may be `null` if there is no relic on that tile).
- The tile maze may be a 2D array of `Tiles`, but it also has unique behaviors (such as the tile sliding mechanism). A `TileMaze` class (whose primary field is a 2D `Tile` array and which features methods that modifies the array) can be helpful.
- Lastly, a game of Moving Maze has an internal state that e.g. keeps track of whose turn it is, and what tile is currently the floating tile. Why not make a `GameState` class that handles these for you?

7.3 Main file

Your submission must include a file named `MovingMaze.java`, which will henceforth be called your submission's main file. Your main file must begin with a comment listing your full name and your student number. Your main file must include a main method which executes the game as explained in Section 3.1. In addition, your main file must include the testing API functions (as described in Section 8.2) that are appropriate to the current hand-in. A main file template that meets the requirements above is available on SUNLearn.

7.4 Submission format

Your submission must be a single `.zip` file containing your full project submission (all your code, any external libraries, and any media files). It must be named `SUxxxxxxxx.zip`, where `xxxxxxxx` is your student number. You may include multiple `.java` files. No Java `.class` files may be included — these will be removed before your submission is assessed.

Faulty submissions will not be granted an opportunity to resubmit. Here's how to ensure your submission is in the correct format:

1. Place your submission (as the `.zip` file that you are planning to submit) in an empty new folder.
2. Extract its contents (i.e. unzip it).
3. If there are any `.class` files, delete them from your folder (and any subfolders).
4. *Use the terminal* to compile *only* your main file with the `javac` command: `javac MovingMaze.java`. This will implicitly find other Java files that your main file needs and compile them too. Do not use an IDE (VS Code, Eclipse, etc.) for this step!
5. Run your program with `java MovingMaze <board> <text/gui>`. You can use one of the test files' boards, or even create your own.
6. If no error has occurred up to here, you are good to go!

Use the terminal!

For this project, it is vital that your project can compile with the Ubuntu terminal, as the terminal will be used to launch your project when it's assessed. Get comfortable with using the terminal — there are many helpful resources for this online. Basic Linux proficiency is an increasingly important skill for everyone, and is basically mandatory for computer scientists.

8 Automated testing

This section gives *technical details* on how your project will be tested for the first and second hand-ins (see Section 9 for the *administrative details*, such as mark allocations). Two approaches will be followed: Input-output testing (Section 8.1) and the implementation of a test API (Section 8.2).

❗ Operating system support for the testing infrastructure

The test API works on all operating systems, as it is pure Java code. However, the input-output testing scripts are only guaranteed to work on Linux. There are many ways to get access to Linux, such as using a virtual machine (VM), using Windows Subsystem for Linux (WSL), dual-booting Linux alongside Windows, wiping your hard-drive and installing Linux (recommended if your major is Computer Science), or using the standardized Ubuntu system in NARGA. No tech support will be given to those trying to make things work on their personal computers. Learning how to fix computer problems yourself is a valuable skill ☺

8.1 Input-output testing

In this testing approach, your program's text mode will be run with known movesets that must result in known outputs. The input-output tests will be facilitated with bash scripts — these will execute your program with the `java` shell command, feed it a series of moves, compare the output to an output file, and report whether it matches. Your program's output must match the expected output exactly, or it will not earn marks.

8.1.1 Test cases

A test case simulates the user using the program to play the game once. Three test files make up each test case:

- `boardX.txt`, a game board file (explained in Section 3.3);
- `movesX.txt`, a file containing a set of moves (explained in Section 4.4; and
- `outputX.txt`, the expected text output of the program should `boardX.txt`'s game board be loaded and `movesX.txt`'s moves be inputted. Note that in these output files, the move inputs do not appear at the prompts due to the fact that redirecting to standard input does not print the information to standard output.

There are multiple test cases, and their test files are provided in `student-test-files.zip` on SUNLearn. A separate set of secret test cases will be used to assess your project.

8.1.2 Test scripts

Three bash scripts are provided, which input the GBF and the user's moves as explained in Section 8.1.1

- `cs144-run.sh`, which presents the output of your program when it simulates a single test case. It takes one argument, which is the ID number of the test case. Use this

to visually inspect your outputs and find obvious visual bugs. **Tip:** pipe the output to the `less` command to get a scrollable interface.

- `cs144-compare.sh`, which reports whether the output of your program matches the expected output of a set of moves (and reports where the outputs differ if it doesn't match). It takes one argument, which is the ID number of the test case. Use this to summarize whether your program works correctly for a singular test case.
- `cs144-testall.sh`, which reports whether your program works correctly for each of the provided test cases. Use this if you are more or less sure your program is fully functional — if the script says it's not, use the other two scripts to inspect test cases one at a time.

Make the scripts executable and run them with

```
$ chmod +x cs144-*.sh
$ ./cs144-run.sh 1
$ ./cs144-compare.sh 1
$ ./cs144-testall.sh
```

8.1.3 Folder structure

To operate correctly, the testing script and its supporting folders must be in the directory structure as shown in Figure 20.

```
<your project folder>
├── boards
│   ├── board1.txt
│   ├── ...
│   └── boardN.txt
├── moves
│   ├── moves1.txt
│   ├── ...
│   └── movesN.txt
├── outputs
│   ├── output1.txt
│   ├── ...
│   └── outputN.txt
├── <all your .java files>
├── MovingMaze.java    // with a main function to start the game
├── cs144-compare.sh   // will call $ java MovingMaze
├── cs144-run.sh       // will call $ java MovingMaze
└── cs144-testall.sh   // will call $ java MovingMaze
```

Figure 20: The correct folder structure for the testing scripts to work.

8.2 Test API

The test API allows your program's inner workings and intermediate steps to be tested. It consists of a set of predetermined functions (which you must write yourself) and which our testing suite will call so that the correctness of the individual functionalities involved can be assessed. The test API is more lenient than an input-output testing approach — with the test API, the functionalities of your program can be tested in isolation from other functionalities.

> **What's an API?**
>
> An application programming interface (API) is a method by which two computer programs can communicate with one another. It usually features a predefined set of operations that one program can execute at the request of the other program. In this case, the two programs are your game client (which will contain the functions described in this section) and our test suite (which will call these functions with known inputs and compare it to known outputs).

The functions described below must be included in your main file, `MovingMaze.java` (see Section 7.3). They must be static functions.

8.2.1 Test API function descriptions

See Appendix A for full specifications of the test API's functions.

8.2.2 Test suites

During assessment, a test suite program will call the test API functions in your main file with known input parameters and evaluate the outputs. Two practice suites are provided on SUNLearn; these are called `StudentTestSuiteX.java`, where `X` is the number of the appropriate hand-in (1 or 2).

8.2.3 Folder structure

To ensure the test suite can communicate with your test API functions, ensure the folder structure in Figure 21 is met.

```
<your project folder>
├── <all your .java files>
├── MovingMaze.java          // write the API functions in here
├── StudentTestSuiteBaseClass.java
├── StudentTestSuite1.java   // for the first hand-in
└── StudentTestSuite2.java   // for the second hand-in
```

Figure 21: The correct folder structure for the testing scripts to work.

8.2.4 How to write functions for the test API

The goal of the API functions are to demonstrate that subsections of your code works. You should make use of the functions and classes from the rest of your codebase wherever possible. The API functions are not meant to implement standalone solutions — they should contain as little code as possible, and the little code they do contain should be high-level code that only references the rest of your codebase.

For the reasons above, it is highly recommended that you first create a semi-functional program (as per the specs of the current hand-in). Then, you can implement the test API functions by using classes and functions that you've defined when writing the game. If done right, none of the test API functions' bodies will need more than five lines of Java code, as the majority of the logic should be done by the rest of your codebase.

Marks will be deducted for poor-quality implementations of the API functions.

Example. Consider a hypothetical API function that should return `true` if a six-character string-encoded tile (e.g. `1111g1`) contains a relic, and return `false` if the tile has no relic. Figure 22 shows the incorrect and correct ways of writing such an API function based on the guidelines above.

```
// A bad API function
boolean tileHasRelic(String encoding) {
    // check if the fifth character is not an 'x'.
    char color = encoding.charAt(4);
    // if it is not 'x', there is a relic.
    return color != 'x';
}
```

(a) What a test API function *should not* look like. It doesn't touch any of the other code in the submission's codebase, and it does low-level operations (checking for a character's presence in a string).

```
// A good API function
boolean tileHasRelic(String encoding) {
    // make a Tile object (the Tile class is defined in Tile.java)
    Tile tile = Tile(encoding);
    // use its hasRelic method (defined in the Tile class)
    return tile.hasRelic();
}
```

(b) What a test API function *should* look like. It interfaces with the codebase (namely the `Tile` class defined in `Tile.java`) and uses high-level code (using a method of `Tile` defined which does all the logic).

Figure 22: How a test API function should be constructed — correct vs incorrect.

9 Assessment

9.1 Project mark and ratio of hand-ins

Your project mark counts 50% of your course total. The three hand-ins will count 15%, 15%, and 20% respectively towards your course total. The remaining 50% of your mark consists of tests and tutorials — consult the module framework for more information.

9.2 Assessment method

Your submission for all hand-ins will be assessed with automated testing (see Section 8). Adherence to the project specifications will be enforced strictly in the automated tests. Ensure that all your text outputs and your API function definitions match the specifications — use the public testing infrastructure provided to do so.

Programs that fail to compile will be awarded 0 marks for the hand-in. For all three hand-ins, your projects will be assessed on a NARGA computer running Ubuntu. It is your responsibility to ensure that your submission works in this environment. Any problems your submission encounters (that were within your hands to fix) may not be solved by the assessor before evaluating your submission.

Your submission for the third hand-in will be assessed during an in-person demonstration with a learning assistant. Attendance of this assessment is compulsory — absence will result in an automatic 0 for the hand-in.

9.3 Assessment rubrics

9.3.1 First hand-in

Mark for first hand-in m_1 has 20 marks available and weighs 15% towards your course mark.

- 10 marks — Testing API (see Appendix A.1 for these functions' descriptions)
  - `isTileOpenToSide` (2)
  - `rotateTileClockwise` (1)
  - `rotateTileCounterclockwise` (1)
  - `slideTileIntoMaze1` (2)
  - `canMoveInDirection` (2)
  - `canMoveAlongPath` (2)
- 10 marks — Input-output testing
  - 10 games in total, 1 mark each
  - 6 of the 10 games are correct (error-free) and are played until `quit` is entered. They are of increasing length:
    - 1st and 2nd games — only input will be `quit`.
    - 3rd and 4th games — rotate the tile, slide it in, quit.
    - 5th game — rotate+slide, move, end turn, quit.
    - 6th game — four full turns, then quit. Will include an adventurer being slid off the board.

- 4 of the 10 games will test incorrect move responses
  - 1st game — sliding into odd positions
  - 2nd game — sliding into last exit point
  - 3rd game — moving in inaccessible directions
  - 4th game — moving off the board through an open side

9.3.2 Second hand-in

Mark for second hand-in m_2 has 10 marks available and weighs 15% towards your course mark.

- 5 marks — Testing API (see Appendix A.2 for these functions' descriptions)
  - `tileHasRelic` (1)
  - `slideTileIntoMaze2` (4)
- 5 marks — Input-output testing. 5 fully played games, 1 mark each. Increasing complexity.

9.3.3 Third hand-in

Mark for third hand-in m_3 has 50 marks available and weighs 20% towards your course mark.

- 20 marks — Demo
  - 10 marks — questions on your code, with a subminimum of 5 marks. If less than 5 marks are earned here, the demo will be repeated with a lecturer or project coordinator.
  - 10 marks — GUI
- 20 marks — Code review
  - 10 marks — style (comments, indentation, general good coding practices)
  - 10 marks — use of OOP
- 10 marks — Automated tests (input-output only), which will primarily test pathfinding.

A Test API function specifications

The functions that the testing API must implement, and descriptions of how each must work, is given below. Match the functions' parameter types and output types carefully.

A.1 First hand-in

`public static boolean isTileOpenToSide(String tileEncoding, char dir)`
Returns `true` if the tile encoded as `tileEncoding` is open in the direction indicated with `dir` and `false` if not. As per the project specification, the `tileEncoding` string will be six characters long. The direction `dir` will be one of `n`, `e`, `s`, and `w`, respectively indicating north, east, south and west.

`public static boolean[] rotateTileClockwise(String tileEncoding)`
Take the tile encoded with `tileEncoding` and rotate it once clockwise. Return a boolean array with length 4 – each element in this boolean array must indicate whether the **rotated** tile is open to a specific side. The order of sides is north, east, south, west.

`public static boolean[] rotateTileCounterclockwise(String tileEncoding)`
Same as `rotateTileClockwise`, but rotate the tile once counterclockwise instead of once clockwise.

`public static boolean[] slideTileIntoMaze1(String[][] mazeTileEncodings, String floatingTileEncoding, String slidingIndicator)`
Take the floating tile encoded with `floatingTileEncoding`, insert it at the sliding position `slidingIndicator` into the maze encoded with the 2D array of strings `mazeTileEncodings`, and return a boolean array of length 4. Each element in this boolean array must indicate whether the **new** floating tile (**after the slide is performed**) is open to a specific side. The order of sides is north, east, south, west.

`public static boolean canMoveInDirection(String curTileEncoding, String newTileEncoding, char dir)`
Returns `true` if the tile encoded as `newTileEncoding` can be stepped to from the tile encoded as `curTileEncoding` if `newTileEncoding` is in direction `dir` from `curTileEncoding`. Returns `false` if not. As per the project specification, the two tile encoding strings will each be six characters long. The direction `dir` will be one of `n`, `e`, `s`, and `w`, respectively indicating north, east, south and west.

`public static boolean canMoveAlongPath(String[][] mazeTileEncodings, char[] steps)`
Returns `true` if an adventurer starting in the top-left corner can successfully complete the steps in the directions contained in `steps`. The char array `steps` is populated from among the options `n`/`e`/`s`/`w`, which indicate the four compass directions. The maze's tiles are encoded in the 2D array `mazeTileEncodings`. The function must return `false` if any step in the path cannot be taken for any reason. No other adventurers are present in the maze.

A.2 Second hand-in

`public static boolean tileHasRelic(String tileEncoding)`
Returns `true` if the tile encoded with `tileEncoding` has a relic on it, and returns `false` if not.

`public static char slideTileIntoMaze2(String[][] mazeTileEncodings, String floatingTileEncoding, String slidingIndicator)`
Take the floating tile encoded with `floatingTileEncoding`, insert it at the sliding position `slidingIndicator` into the maze encoded with the 2D array of strings `mazeTileEncodings`, and return a Java character. This character represents the relic on the **new** floating tile — if it has a relic (regardless of its collection order number), return its lowercase initial; if it has no relic, return `'x'`.