L3

EECS-3311 – Lab3 – Maze

Table of Contents

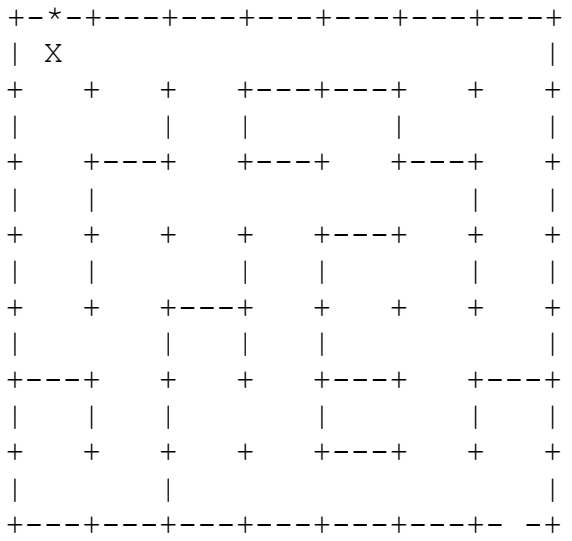
1 Introduction and goals

```
require
     across 0 |..| 2 is i all lab_completed(i) end
     read accompanying document: Eiffel101[1]
     attend all lectures/labs and compile/run/understand lecture code
ensure
     submitted on time
     no submission errors
     lab submission is reliable (correct and robust) for any input
     correct means it matches the output of the oracle character for character
rescue
     ask for help during scheduled labs
     attend office hours for TAs Kevin and Connor
```

- Lab1: Adjacency List Graph (use of `ARRAY[G]` and `LINKED_LIST[G]` from Eiffel base)
- Lab2: Abstract Graph (use of `COMPARABLE_GRAPH[G]` from Mathmodels)
- Lab3/**ETF**: ETF Maze **Design** and implementation. Use a graph as a model of a maze.

[1] See: https://www.eecs.yorku.ca/~eiffel/eiffel101/Eiffel101.pdf

The goal of this Lab is to **design and implement a simple maze game.** An example of a maze is shown below. The player (X) starts in the top left coordinate (1,1) as shown below in a 7x7 coordinate game.



A user plays the game through a series of commands that they enter through a command-line interface. A user will be able to start new games at various difficulties (corresponding to different sizes of mazes), as well as navigate the player through the maze. Each successful maze they complete will earn them points. If they get stuck, they will have access to a "solve" command which will guide them to the exit using the shortest path; however, they won't earn any points for that round if they use the solution.

Part of a game at the command line is shown in Table 1. A new game is started and the player is issued a command to move one coordinate at a time. The goal is to reach coordinate (7,7) at the bottom right in order to score. If a player attempts to move to a coordinate blocked by a wall, a status message will be displayed indicating that the move is illegal.

Table 1 Part of Maze game

```
./maze -i
  State: 0 -> ok
->new_game(easy)
  State: 1 -> ok
  +-*-+---+---+---+---+---+---+
  | X                         |
  +   +   +   +---+---+   +   +
  |       |   |       |       |
  +   +---+   +---+   +---+   +
  |   |                   |   |
  +   +   +   +   +---+   +   +
  |   |       |   |       |   |
  +   +   +---+   +   +   +   +
  |       |   |   |           |
  +---+   +   +   +---+   +---+
  |   |   |       |       |   |
  +   +   +   +   +---+   +   +
  |       |                   |
  +---+---+---+---+---+---+- -+
  Game Number: 1
  Score: 0 / 1
->move(S)
  State: 2 -> ok
  +-*-+---+---+---+---+---+---+
  | *                         |
  + * +   +   +---+---+   +   +
  | X     |   |       |       |
  +   +---+   +---+   +---+   +
  |   |                   |   |
  +   +   +   +   +---+   +   +
  |   |       |   |       |   |
  +   +   +---+   +   +   +   +
  |       |   |   |           |
  +---+   +   +   +---+   +---+
  |   |   |       |       |   |
  +   +   +   +   +---+   +   +
  |       |                   |
  +---+---+---+---+---+---+- -+
  Game Number: 1
  Score: 0 / 1
->move(E)
  State: 3 -> ok
  +-*-+---+---+---+---+---+---+
  | *                         |
  + * +   +   +---+---+   +   +
  | * * X |   |       |       |
  +   +---+   +---+   +---+   +
  |   |                   |   |
  +   +   +   +   +---+   +   +
  |   |       |   |       |   |
  +   +   +---+   +   +   +   +
  |       |   |   |           |
  +---+   +   +   +---+   +---+
  |   |   |       |       |   |
  +   +   +   +   +---+   +   +
  |       |                   |
  +---+---+---+---+---+---+- -+
  Game Number: 1
  Score: 0 / 1
```

One way to **model** a maze is as a graph where the edges of the graph represent possible moves. For example, one of the user commands is "solve", which must display the shortest path to the exit if it exists. In a subsequent section, there is more information on the graphical representation of mazes and its display in an ASCII format.

To check the correctness of your design we will use the **Eiffel Testing Framework** (ETF), which provides an abstract user interface though which we can run acceptance tests to check your game logic. You create the maze game using ETF to setup the user interface; you design the rest of the game. This lab (Lab 3) is a simple introduction for you to develop proficiency using ETF, as well as a way to apply the work you have been doing with graphs to an interactive example. In the coming weeks, Lab 4 and the Project will both make use of ETF, so it essential that you do succeed at Lab3.

1.1 Design

A major goal of a clean design is to facilitate subsequent bug fixes and changes to the requirements. This is where information hiding, modularity, abstraction and a separation of concerns become helpful. The design of the main business logic for the game must demonstrate good modularity, clean specified APIs, suitable information hiding, separation of concerns and helpful abstractions.

In our solution, we have about 10 classes in the business logic. In this Lab only, we provide you with some starter code to help in the representation of mazes. If you have not designed modular systems before (with modules being classes or clusters of classes), this is your opportunity to achieve the skills of a designer. You may have more or fewer classes, but they must demonstrate elements of good design. Classes must be **designed** so that it pertains to a single well-defined abstraction (see OOSC2 chapters 22 and 23):

- A class should be known by its interface, which specifies the services offered independently of their implementation.
- Class designers should strive for simple, coherent interfaces.
- One of the key issues in designing modules is which features should be exported, and which should remain secret.
- When considering the addition of a new exported feature to a class, the feature must be relevant to the data abstraction represented by the class; it must be compatible with the other features of the class; it must not address the same goal as another feature of the class; and must maintain the invariant of the class.
- The design of reusable modules is not necessarily right the first time, but the interface should stabilize after some use. If not, there is a flaw in the way the interface was designed.
- Proper use of assertions (preconditions, postconditions, invariants) is essential for documenting interface **specifications**.
- **Documentation principle**: Try to write the software so that it includes all the elements needed for its documentation, recognizable by the tools that are available to extract

documentation elements automatically at various levels of abstraction (e.g. contract view vs. text view vs. BON/UML class diagram).
- Follow good style (see OOSC chapter 26).
- Do regular rigorous regression testing as you develop the design and implementation using unit tests and acceptance tests.

1.2 Design Decisions

You will be making many design decisions as you construct a solution as mentioned above. In addition to deciding on how to decompose your solution into classes, their architecture and their specifications, you will also have to decide how to ensure that the player does not bump into walls, how the player backtracks, how to ensure that scores are correct, how to find the shortest path to the exit, or what to do when there is no path at all.

This is an open-ended problem with many solutions, some better than others. Recall that **information hiding** requires that you decouple and hide the design decisions in such a way that we can revisit and change those decisions at a later time if needed without it impacting on the rest of the design.

Your **design** must be feasible and **correct** which is why we insist that you actually implement the design and why we will rigorously check your design with acceptance tests.

1.3 Eiffel Testing Framework for Design

You use the ETF framework to separate out the user interface (view/controller) from the business logic (the model). To read more about the Eiffel Testing Framework (ETF), see:
- http://seldoc.eecs.yorku.ca/doku.php/eiffel/etf/start
- Videos: https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#eiffel_testing_framework
- Conference paper: https://www.eecs.yorku.ca/~jonathan/publications/2018/MoDRE18.pdf

The course wiki will have precise details of information we provide you with as well as how to submit your Lab.

2 Roadmap for this Lab

1. Read the background section and watch the tutorials on how to build an ETF project. See Section 1.3.
2. Retrieve the starter folder (“maze-starter”). Compile and run the starter project (“initial-maze”); it will compile but none of the tests will suuceed because all graph routines are no implemented. Transfer your version of LIST_GRAPH implementations to the starter code, compile and run it. Now all the tests should succeed. You will be

able to use the "graph" cluster in this starter code in your ETF project. The "graph" cluster makes it easy for your project to generate and draw a variety of mazes. See Section 4.
3. Create a new ETF Project, transfer over the files you need to their respective model and graph clusters (as explained in section 5.2).
4. Implement your design, writing acceptance tests along the way.

3 Background

This section will provide you with the background knowledge necessary to complete this lab. Please study it carefully.

3.1 Prerequisites

Before continuing reading this document, ensure you have completed the following:

1. **Lab 1 & Lab 2**, with complete implementations for all classes.
2. **Reviewed the introduction to ETF** provided online (below)

You use the ETF framework to separate out the user interface (view/controller) from the business logic (the model). To read more about the Eiffel Testing Framework (ETF), see:

- http://seldoc.eecs.yorku.ca/doku.php/eiffel/etf/start
- Videos: https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#eiffel_testing_framework

3.2 Graphical Maze Representation

One possible way to represent a Maze is to use a graph.

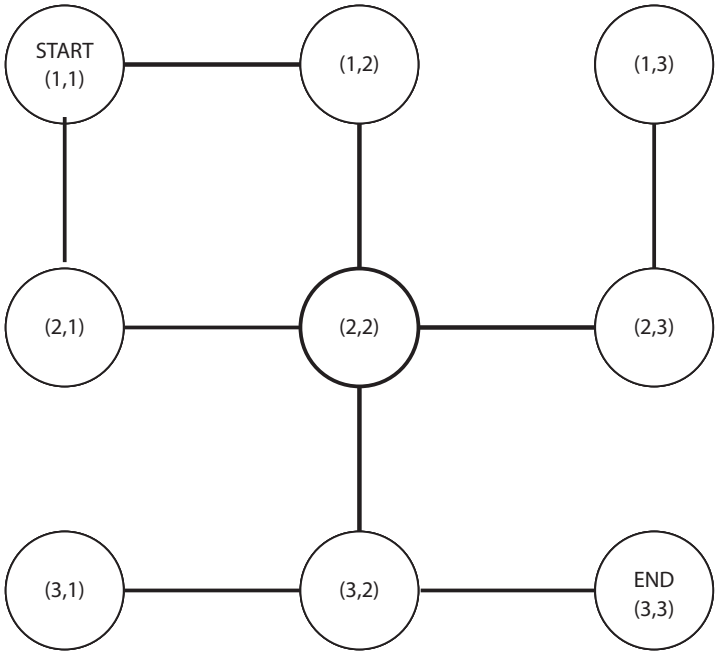


Figure 1 - Graph-like visualization of a maze. Values inside vertices represent the coordinate of that vertex in the maze.

In this graph representation, the **vertices represent valid locations** the player may travel to, and the **edges represent valid paths**. Each vertex corresponds to a particular coordinate, and the edges are the paths between two coordinates.

Here is how the above graph would look converted into an ASCII representation:

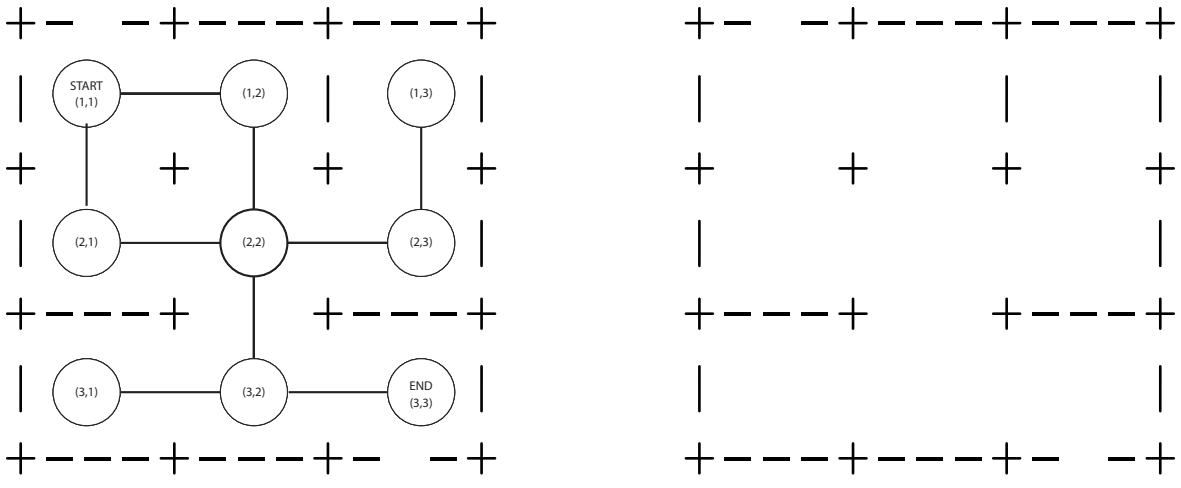


Figure 2 – Maze graph representation converted to ASCII

In labs 1 and 2, we looked at how to implement an implementation for an adjacency list graph structure. We also implemented several graph algorithms for traversing and analyzing the graphs. Now in lab 3, we will be using the LIST_GRAPH to represent maze structures, and it will be your job to complete an implementation for a command-line maze game.

It is important to notice that these maze graph representations are a subset of all possible graphs. In our game, **graph-mazes will be constructed for you**, and you will be able to make the following assumptions about those graphs:

1. All graphs are generated with exactly *n* x *n* vertices, containing a corresponding COORDINATE inside those vertices, where *n* is the number of vertices in each row or column.

2. Paths between valid locations in the maze are two-way. In order to accommodate this in the graph, if an edge from (1,2)->(2,2) exists in the LIST_GRAPH, then (2,2)->(1,2) will as well.

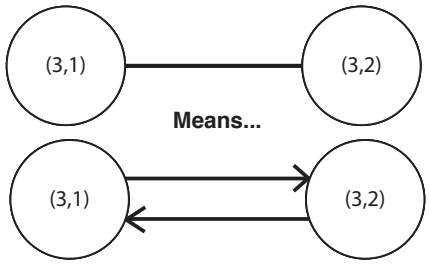


Figure 3 - Edges between vertices in a maze will always have the reverse edge as well.

3. Edges will only be generated between nodes that are **single coordinates apart in *cardinal* directions**. That means a vertex can only have a maximum of 4 possible outgoing edges:
    - North (visually upwards)
    - South (visually downwards)
    - East (visually right)
    - West (visually left)

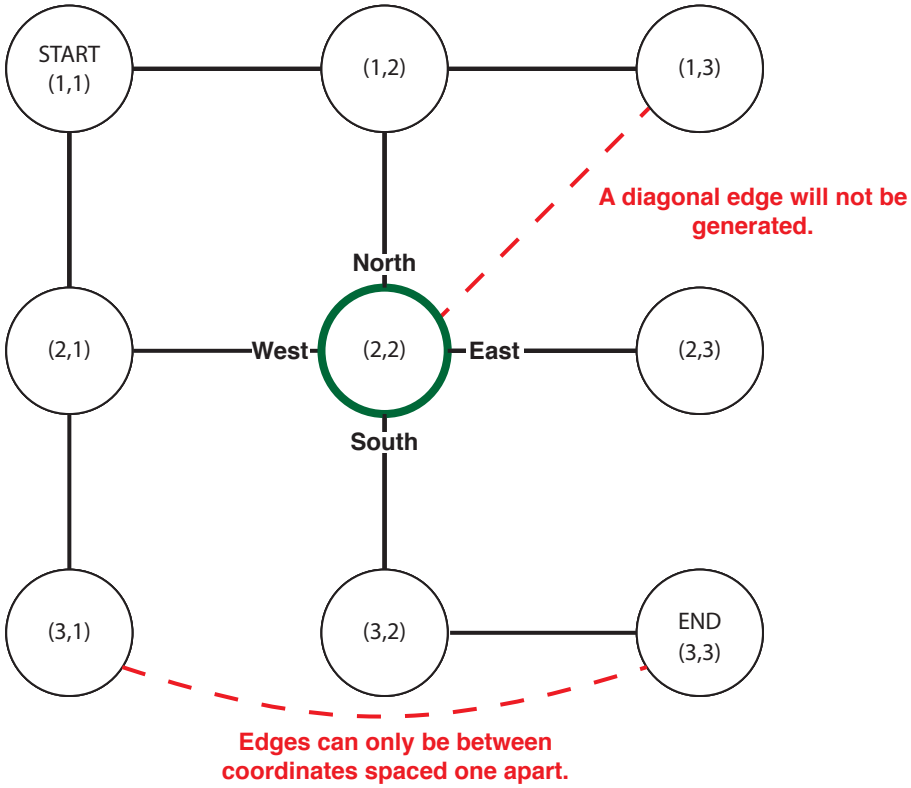


Figure 4 - Illustration of cardinal directions from (2,2), as well as illegal edges marked in red.

3.3 Game Description

The game will include four different commands for the player to enter, as defined by the grammar (shown in section 4).

1. new_game
2. move
3. abort
4. solve

3.3.1 Starting a Game

Players begin by entering *new_game*(*level)* and then they are shown a maze that they must navigate through. The player always starts at the coordinate (1,1) in the top left corner. An easy game (level 1) will be a 7x7 maze, a medium (level 2) will be an 11x11 maze, and a hard game (level 3) will be a 15x15 maze.

An error will be shown if the player attempts to start a new game while in the middle of a game.

3.3.2 Movement

Players can use the *move(direction)* command to navigate their player through the maze, which will move their player character ('X') in the specified direction and leave a trace of the previous positions they have travelled to thus far (as '*'s).

An error will be shown if the player attempts to move while not in a game, or if they attempt to move in a direction that is blocked by a wall in the maze.

3.3.3 Scoring, Winning, and Losing

Throughout the duration of the playing session, a total score is tracked across all the games a player plays. Each easy game adds 1 to the total score, medium adds 2, and hard adds 3. The score is displayed as (points earned) / (total score).

If the player arrives at the bottom right coordinate in the maze (where the exit is located), they win the game and are awarded full points for their win. This means if they were playing easy, they get 1, medium = 2, and hard = 3.

If the player uses the *abort* command at any point along the game, the current game is over and they are awarded 0 points.

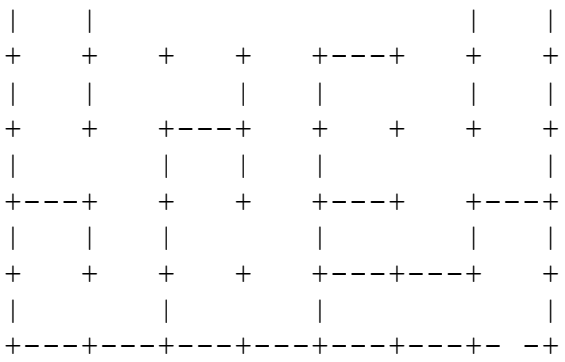
If the player uses the *solve* command at any point in the game, a shortest path is calculated from their current position and a trace is marked out from that position to the exit. The trace will not overwrite any previous player path characters ('*'s) if it would overlap with them. The game continues, allowing the player to complete the maze, but they are not awarded any points for a game where the *solve* command was used.

An error will be shown if players attempt to use the solve command or the abort command while not in a game.

3.3.4 Unsolvable Mazes

Occasionally, the generator will produce an unsolvable maze. An example is shown below:

```
+-*-+---+---+---+---+---+---+
| X                           |
+   +   +   +---+---+   +   +
|       |   |       |       |
+   +---+   +---+   +---+   +
```



If such a maze is produced, then the game shall allow the player to move around as normal. If the player aborts the game, the point total will revert back to what it was previously, since it was impossible for the player to gain points that round. If the solve command was used, the user should be presented with a message informing them that the maze was unsolvable, and that they can abort with no penalty.

4 Starter Code

To retrieve the starter code, please use the following command:

```
~/sel/retrieve/3311/lab3
```

The starter folder has the following directory structure:

```
maze-starter
├── errors.txt
├── initial-maze
│   ├── graph
│   │   ├── coordinate.e
│   │   ├── direction_utility.e
│   │   ├── edge.e
│   │   ├── edge_comparator.e
│   │   ├── list_graph.e
│   │   ├── maze_drawer.e
│   │   ├── maze_generator.e
│   │   └── vertex.e
│   ├── maze_starter.ecf
│   ├── root
│   │   └── root.e
│   └── tests
│       └── starter_tests.e
├── maze_grammar.txt
├── oracle.exe
├── regression-testing
│   ├── ETF_Test.py
│   └── ETF_Test_Parameters.py
└── tests
    └── acceptance
        └── instructor
            ├── at01.expected.txt
            ├── at01.txt
            ├── at02.expected.txt
            └── at02.txt
```

The **maze_starter** folder contains the starter code for the lab. Refer to 2.4.1 to see how to run this, and 2.4.2 for explanations on the classes.

Test Run:02/04/2020 6:53:53.318 PM

ROOT

Note: * indicates a violation test case

FAILED (5 failed & 0 passed out of 5)

Case Type	Passed	Total
Violation	0	0
Boolean	0	5
All Cases	0	5

State	Contract Violation	Test Name
Test1	STARTER_TESTS	
FAILED	Check assertion violated.	t1: Test creation of easy difficulty maze
FAILED	Check assertion violated.	t2: Test creation of medium difficulty maze
FAILED	Check assertion violated.	t3: Test creation of hard difficulty maze
FAILED	Check assertion violated.	t4: Test player movement in easy maze
FAILED	Check assertion violated.	t5: Test player winning in easy maze

The **regression_testing** folder contains the scripts for running the regression tests, as well as an oracle which you can interact with to see how the game should respond to different scenarios.

The **maze_grammer.txt** contains the grammar for creating a new ETF project for this lab.
The **errors.txt** file contains a list of errors that should be displayed when the user enters an invalid command.

The starter code project directory structure looks as follows:

There are 4 different clusters: **graph, model, root, and tests.**

You are provided with this starter code to provide you with a jumping off point for your development. First and foremost, you are given a way to generate these maze-graph structures in a deterministic way, which can be found in **MAZE_GENERATOR**. Additionally, drawing functionality is also provided to you to help save some time in your implementation, which can be found in **MAZE_DRAWER.** MAZE_DRAWER also contains a very naiive implementation for movement, which must be expanded upon and linked to ETF as a part of your project.

You *must* use the MAZE_GENERATOR as the source of the maze structures that you build, but after that are welcome to use as little or as much of the other starting code as you'd like.

However, as it will be covered later, your code will be evaluated **character for character** against the oracle program that is also provided to you. You will probably want to take advantage of the drawing features provided in MAZE_DRAWER as well to save you implementation and debugging time.

4.1.1 Running the Starter Code

In order to run the starter code, **first copy your implementations of the features for LIST_GRAPH into the empty features in the starter code**. The starter code should now compile, and you should be able to run the workbench system. The starter code contains several tests to ensure your implementation carried over well, and if you observe your console, you should see several maze printouts there.

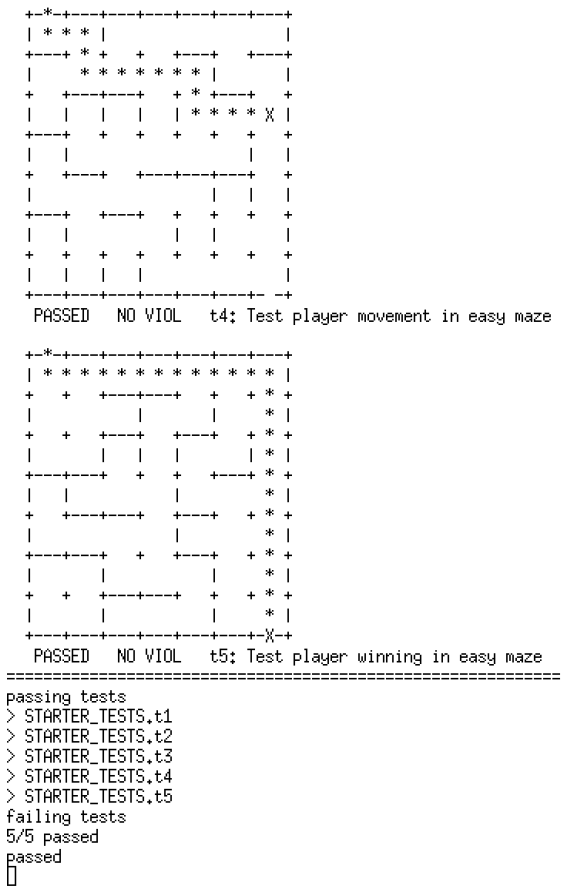


Figure 5 - Output of starter code being run.

4.1.2 Other Classes

The starter code has many comments to help guide you through it. Please refer to the classes for more details on the inner workings. A summary of each class and most notable features are provided below:

MAZE_GENERATOR

- Exports a single method, `generate_new_maze(level: INTEGER)` which must be used to create the graph-maze structures. This accepts an integer value of 1, 2, or 3, based on the difficulty desired.
- Level 1 will generate a maze of 7x7 coordinates, level 2: 11x11, and level 3: 15x15.

MAZE_DRAWER

- Provides an implementation for drawing the maze from the graph (`draw_graph` and `print_graph`), as well as a naiive method for moving the player (`move_player`).

COORDINATE

- The chosen item to store in each VERTEX of the graph. As can be seen in MAZE_GENERATOR, generate_new_maze produces a LIST_GRAPH[COORDINATE]. This is a very simple class that store row and column information for that location.

DIRECTION_UTILITY

- A utility class used by the generator and drawer to abstract the cardinal directions into "tuple modifiers". Think of them as unit vectors in their respective directions.
  - As seen in this class, **south** (visually downwards) is represented by a tuple (1,0). This is used in the starter code by **move_player** to find the coordinate in the southern direction from the player. For example, if the player is in coordinate (1,1), and they move south, then the implementation finds the resulting coordinate by adding (1,1) + (1,0) = (2, 1). So the player should be placed in coordinate (2, 1).
- You do not need to use this class if you don't want to, but you must include it in your ETF project for the generator.

ROOT

- This contains the code for running the starter code. If **test_mode** is set to `true` then it will run the tests. If it is `false`, it will run the tests and just print the mazes and player movement to the console. Use this as a way to learn how to interact with the starter code.

5 Getting Started

We assume you have already studied the information in previous section. This section will help guide you with some tips on how to complete the lab.

5.1 Suggested Steps

1. Plan and draw out your design for the system.
    - Which classes will you need to create? What will control errors? Where will you implement your shortest path algorithm?
2. Generate a new ETF project using the provided grammar.
3. Copy over the files you need from the starter code into your ETF project.
    - See section 5.2 for how to disable contract checking for your graph cluster.
4. Create some very simple regression tests (e.g. just running one **new_game** command). Ensure your regression testing script runs them correctly (see section 5.3.1 for information on running the regression tests).
5. Start working on getting the graph printout to work so you have some good visual feedback on how your implementation is coming along.
6. Complete your implementation, creating more complex regression tests and ensuring that your implementation creates an identical output to the oracle's output.

5.2 Turning Off Contract Checking for the Graph Cluster

Since we have rigorously tested our LIST_GRAPH classes in labs 1 and 2, we wish to disable the contracts from running so that we can avoid the performance hit that they will cause our game to have. If you do not disable contracts, you will notice the game will be quite slow to run.

You may wish to write some contracts for your own code, so we only wish to turn off contracts for the classes that would be using the GRAPH contracts. Here's how you can isolate a specific cluster to disable contracts for the classes inside it.

1. Create new folder in your files directory called **graph**. Do this through the file explorer. Add the files from the **graph** directory in the starter code to this directory. Put it at the same level as maze.ecf, so it's parent directory is **src.**

2. Navigate to the **Project Settings,** expand **Groups -> Clusters,** then select the **Add Cluster** icon and direct it to your graph directory inside this project.

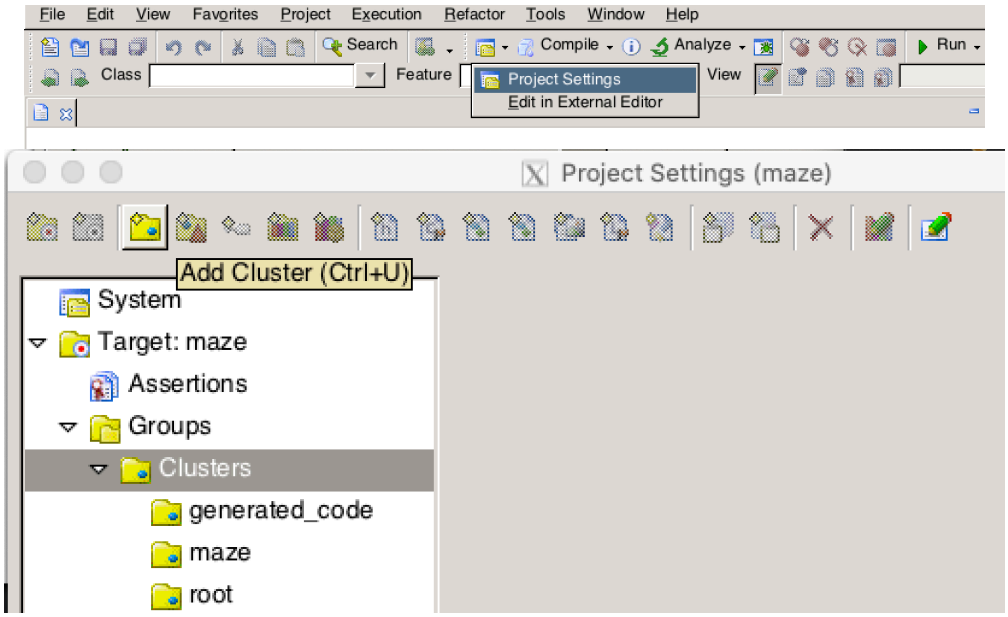


Figure 6 - Adding a new cluster to the project configuration.

3. You should now see graph added to the list of clusters.

4. Change the directory from an absolute path to a relative one. You should just need to put in *./graph/* as the path.

5. Change all of the Assertions from **True** to **False.** Press OK and that should be it!

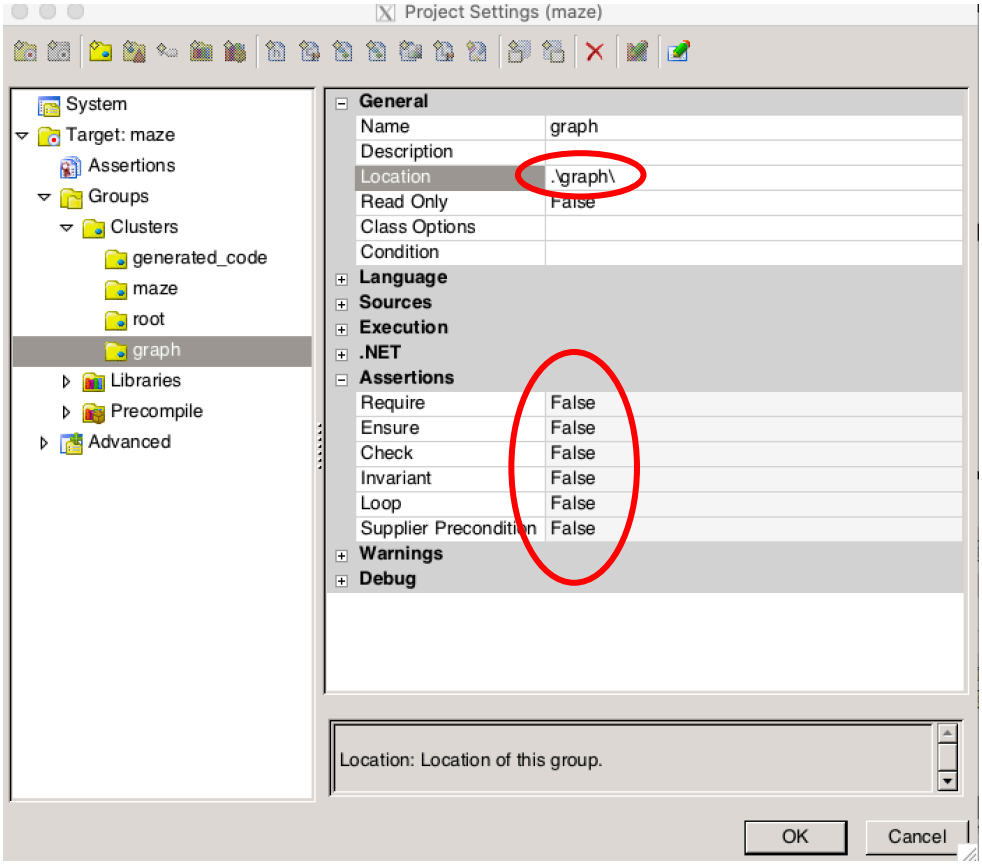


Figure 7 - How your graph configuration should look like at the end. Red circles highlight which parts you need to change.

5.3 Acceptance Tests & Regression Testing

In section 3, we have described a rough specification of how the game should play. While this covers how the player should interact with the game and whatnot, it is not a complete specification.

Instead we provide you with an executable **Oracle** (*oracle.exe),* which acts in place of a complete specification of the game. Thus, you use the Oracle to answer all questions of required behavior that is not described in this document:

Major Requirement: Your design/submission must match the Oracle output character for character for any input specified by the user interface grammar. The output of your submission must also match -- character for character -- any status/error messages generated by the Oracle. In addition to being **correct**, the output of your submission must be **robust**, i.e. your submission must not ever crash with an exception, and it should never enter a non-terminating loop. Failure in any respect cannot be guaranteed a passing grade. A reliable software product must be correct and robust.

With the Oracle you can construct your own acceptance tests. When we test the correctness of your design we have over 30 acceptance tests (some long some short) to check whether you have developed a reliable product.

Requirement
Develop your own suite of acceptance tests using the Oracle, and do rigorous **regression testing** against the Oracle to check the correctness of your design and development.

We will test your submission only with acceptance tests that are syntactically correct, i.e. satisfy the user interface grammar.

Your submission must be robust, i.e. it must be able to operate in the presence of erroneous input from players. We cannot control the user (the Player) and thus we must signal when the Player provides a problematic move.

5.3.1 Running the Regression Tests

1. Configure the ETF_Test_Parameters.py file and ensure the **root** variable holds the correct path information to your tests.
2. Set the value of **is_expected** to **false.** This means it will use the oracle to test your output against.
3. Run the command **python ETF_Test.py.** You will see it generate some output as it runs the regression tests.
4. Failed tests will provide you with a **meld** command that you can copy and paste to display the differences.

6 Design documentation: BON/UML Diagrams

In this course, we use the following diagrams: BON and UML class diagrams, UML sequence diagrams, and possibly UML state-charts. For the UML diagrams, study the following slides:
https://wiki.eecs.yorku.ca/project/eiffel/_media/bon:uml.pdf (Prism login required).
In the project, you will have to provide a Design Document. For the Lab, we will require you to submit two BON class diagrams. Details will be provided on the course wiki.

Requirement: BON diagrams
Please see Lab1 & Lab2 for an example of a BON class diagram generated using the IDE and a BON diagram using the *draw.io* template.

7 Generating a Starter ETF Project

See the course wiki for details on generating a starter ETF project on a Prism machine.

8 Submission

See the course wiki for submission details.

Appendix

Here you may find some useful notes to help motivate your design choices and testing strategies for your implementation.

Design

A major goal of a clean design is to facilitate subsequent bug fixes and changes to the requirements. This is where information hiding, modularity, abstraction and a separation of concerns become helpful. The design of the main business logic for the game must demonstrate good modularity, clean specified APIs, suitable information hiding, separation of concerns and helpful abstractions.

In our solution, we have about 10 classes in the business logic. If you have not designed modular systems before (with modules being classes or clusters of classes), this is your opportunity to achieve the skills of a designer. You may have more or fewer classes, but they must demonstrate elements of good design. Classes must be **designed** so that it pertains to a single well-defined abstraction (see OOSC2 chapters 22 and 23):

- A class should be known by its interface, which specifies the services offered independently of their implementation.
- Class designers should strive for simple, coherent interfaces.
- One of the key issues in designing modules is which features should be exported, and which should remain secret.
- When considering the addition of a new exported feature to a class, the feature must be relevant to the data abstraction represented by the class; it must be compatible with the other features of the class; it must not address the same goal as another feature of the class; and must maintain the invariant of the class.
- The design of reusable modules is not necessarily right the first time, but the interface should stabilize after some use. If not, there is a flaw in the way the interface was designed.
- Proper use of assertions (preconditions, postconditions, invariants) is essential for documenting interface **specifications**.
- **Documentation principle**: Try to write the software so that it includes all the elements needed for its documentation, recognizable by the tools that are available to extract documentation elements automatically at various levels of abstraction (e.g. contract view vs. text view vs. BON/UML class diagram).
- Follow good style (see OOSC chapter 26).
- Do regular rigorous regression testing as you develop the design and implementation using unit tests and acceptance tests.

Practicing creating good design in this lab will help prepare you for Lab 4 and the Project.

9 Validation and Verification: acceptance tests

Figure 8 shows how acceptance tests fit into the Validation and Verification cycle. An acceptance test is a description of the behavior of a software product from the point of view of the user's requirements, generally expressed as an example or a usage scenario. In many cases the aim is to automate the execution of such tests. The Eiffel Testing Framework (ETF) provides a means of doing this. Regression testing at the acceptance testing level is an important method for developing reliable software.

10 Regression Testing

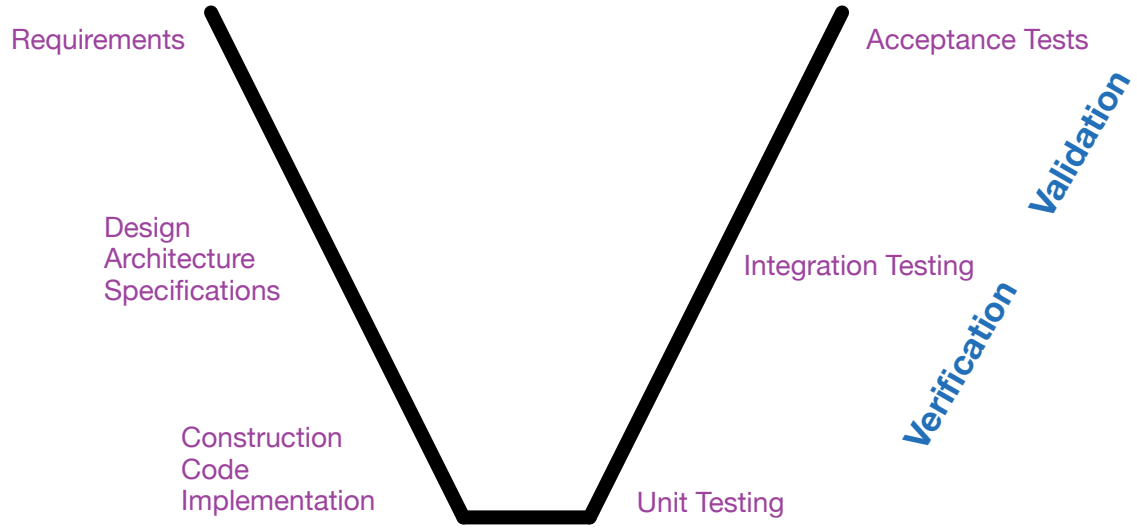
We provide you with Python3 scripts to do regression testing as you design/develop the software. Ensure that you use it and that you run the regression tests every time you make a change.

Regression testing is the process of testing changes to software to ensure that the product still works with the new changes. Regression testing is a normal part of the program development process. The old test cases are run against the changed product to ensure that the old features still work and that new bugs have not been introduced by the changes. Changing or adding new code to a program can easily introduce unintended behaviors and errors.

ETF adds the ability to introduce user requirements into a seamless process of software construction at a point before concrete user interfaces can be specified. ETF allows software developers to produce use cases that can be turned into acceptance tests, and that then free the developer to develop the business logic (the *model*}) while not losing sight of the user requirements. This allows requirements to become part of a seamless development from requirements to implemented code, and allowing change even at the level of requirements. Bertrand Meyer writes:

> The worldview underlying the Eiffel method ... [treats] the whole process of software development as a continuum; unifying the concepts behind activities such as requirements, specification, design, implementation, verification, maintenance and evolution; and working to resolve the remaining differences, rather than magnifying them.

Figure 8 Acceptance Tests in V&V



The terms **Verification** and **Validation** are commonly used in software engineering to mean two different types of analysis. The usual definitions are:

Validation: Are we building the right system?

Verification: Are we building the system right?

Validation is concerned with checking that the system will meet the customer's actual needs (the requirements), while **Verification** is concerned with whether the system is well-engineered, safe, and error-free.

Verification will help to determine whether the software is of high quality, but it will not ensure that the system is fit for use and useful to the customer.

Anyone who has worked in both specification and programming knows how similar the issues are. Formal specification languages look remarkably like programming languages; to be usable for significant applications they must meet the same challenges: defining a coherent type system, supporting abstraction, providing good syntax (clear to human readers and parsable by tools), specifying the semantics, offering modular structures, allowing evolution while ensuring compatibility.

The same kinds of ideas, such as an object-oriented structure, help on both sides. Eiffel as a language is the notation that attempts to support this seamless, continuous process, providing tools to express both abstract specifications and detailed implementations. One of the principal arguments for this approach is that it supports change and reuse. If everything could be fixed from the start, maybe it could be acceptable to switch notations

between specification and implementation. But in practice specifications change and programs change, and a seamless process relying on a single notation makes it possible to go back and forth between levels of abstraction without having to perform repeated translations between levels.