**ComPuzzle Design Documentation**

**CMSC-4900-001**

**Dr. Chen**

**Team:**

**Abigail Dehart**

**Brianna Dulik**

**Brandon Mastin**

**Cameron McGill**

**Instructor Comments/Evaluation**

**Table of Contents**

Abstract\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5

Description of Document\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_6

Purpose and Use\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_6

Ties to Specification Document\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_6

Intended Audience\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_7

Project Block Diagram\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_8

Design Details\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_9

System Modules and Responsibilities\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_9

Architectural Diagram\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_9

Module Cohesion\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_11

Module Coupling\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_11

Design Analysis\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_12

Data Flow or Transaction Analysis \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_12

Design Organization \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_14

Detailed Tabular Description of Classes/Objects\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_14

Functional Description\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_15

Input/Output/Return Parameters/Types\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_19

Modules Used\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_30

Files Accessed\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_31

Real-Time Requirements\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_31

Messages\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_31

Narrative/PDL\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_31

Decision: Programming Language/Reuse/Portability \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_32

Implementation Timeline\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_32

Design Testing\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_33

Resources\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_34

Appendix: Team Details\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_34

Appendix: Workflow Authentication \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_35

Appendix: Writing Center Report\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_36

**Abstract**

The purpose of this document is to give an overview of how we will design our project, ComPuzzle. Our project is meant to be a fun, 2D video game including a character in a maze and blocks of pseudocode. These blocks of pseudocode must be arranged in feasible orders by the user to manipulate the movement of the game character. The arrangement of these blocks of pseudocode is the only way for the character to be moved and make progress through the maze. This method of solely moving the character with pseudocode blocks forces the user to learn basic problem-solving skills. The user must understand how different syntax-free algorithms will output for their character to progress in the game. Our game is focused on problem solving and having fun. This will provide us with an opportunity to understand in greater detail how to implement these ideas.

**Description of Document**

Purpose and Use

The purpose of this document is to prepare and agree upon our design plans prior to execution. This will not only be a show that we understand how to tackle our project, but it will also give us a layout of how to implement some of the code. This will include various classes we will need and their required functions, along with descriptions of their purpose. Anyone else who reads this document could theoretically begin a project similar after reading. This paper is a good way to get our design necessities neatly organized to have an idea of how we can tackle this project when we begin. The proposals we explain in this paper will prove helpful to remember and look back on when necessary to ensure our work is structured and follows according to the plans.

Ties to the Specification Document

There are many connections between this document and our previous one, the specification document. Many similar topics are covered in both papers. The specification document held a focus in specifying exactly what our game will encompass in its finality, including its quality, standards, hardware and software materials, and various cost and time constraints. In general, the specification document’s purpose was to state the costs and requirements of our project. On the contrary, the design document’s focus will be aimed more towards how we will create our game. Frame working our project is what design is all about. This includes an emphasis on the code: some different classes we will use, data flow, cohesion and coupling within our modules, and many other aspects of programming structures.

Though the focuses of both documentations are different, there are similarities. Our functional description of the classes and functions we plan on using is very similar, with minimal changes to ensure they are in-line with our initial proposals. Some graphs and diagrams are reused, as well, as the prompt was identical in what has been asked. We must carry over some work from the specification document as it is true as presented, including plans we intend to follow through with. Changing too much or creating anew for some of these ideas that were carried over may impact the integrity of our proposals. There are many crossovers between these two documents, but with different important concentrations for both.

Intended Audience

The intended audience for this document is the design and implementation team. As this is meant to be a blueprint for the steps and methods we will take to create our program, it will prove useful to our team. This will be our design specifics-- our agreed upon approach to tackling ComPuzzle. These specifics are an important part of not only identifying the work we need to do but planning ahead and showing that we have an understanding of how to approach this work. The presented ideas in this paper have been talked about and agreed upon by the group and will prove helpful to look back on if we need a refresher on our planned approach.

**Project Block Diagram**

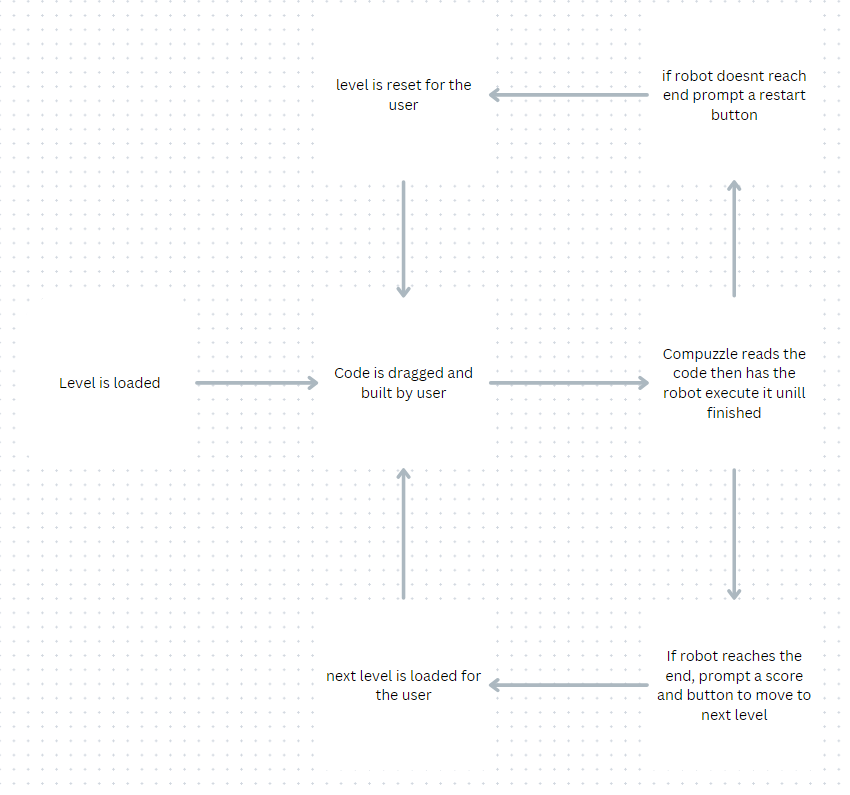


Figure 1: Project Block Diagram on Level Mechanics

The ComPuzzle project will consist of a premade level that is loaded for the user. Then the code and the robot and maze are visualized to the user. The user will then take the code and build the instructions for the robot. ComPuzzle will have the robot execute the code that the user created. Once the robot finishes the code, if it does not reach the goal then the game will prompt a reset button and give the user a chance to try again from the start. If the robot does reach the end, a score will be given based on the number of code blocks used. After, it will prompt the user to move to the next level which will be loaded.

**Design Details**

System Modules and responsibilities

Architectural Diagram

A diagram of a computer system

Description automatically generated

Figure 2: Architectural Diagram on Data Handling

The Diagram, Figure 2, is a simplified show of the architecture of our processing system. The user’s input will be handled by the input handler. This input could be something as simple as hitting “Play” or selecting a level, or it could be the user inputting their pseudocode arrangement to be run. This handler will analyze the user’s input and send it where it needs to go to be processed. The data will go to both the state manager and the game’s logic engine. The state manager will update the state of the objects, actions, or placement in the game. Things such as visual updates and sound effects/game actions are stored and updated in the state manager. This manager will update the GUI and audio for the game with the new state updates after obtaining user requests. These updates are sent back through the input handler to the user, so the user has updated graphics and sound.

The game’s logic engine is the brain of the platform. It will execute the code that creates our game. This code includes the algorithms to deal with player’s pseudocode arrangements. The logic to decipher these processes between the user’s submitted pseudocode and the actions that the in-game character needs to follow are calculated here. The game’s state is retrieved from the state manager, such as player position, and will be updated once the game logic is processed and sent back. Collision detection questions such as “Did the character run into a wall?”, “Was the goal reached?” will need to be asked here. This includes deciphering what the user’s pseudocode arrangement will do after the run button is pressed. Is the pseudocode functional, but the output is not beneficial/correct? Is it functional, correct, but not super-efficient? Is it functional and the best possible input? Is it simply dysfunctional and cannot complete anything? Once these questions are asked, and answers are found, outputs must respond accordingly. The state is gathered from the state manager, questions are asked, and the logic engine can process a response and send it right back to the state engine, sending feedback to the visual/sound systems.

Module Cohesion

Cohesion is the amount a module of a system works with other modules for a main goal (GeeksforGeeks, 2024). Having all of the elements in the above modules of Figure 2 work together cohesively is important for the user’s enjoyment. Cohesion affects the game’s flow and thus the gameplay quite a lot. Our code needs to be organized and focused on a main purpose. If different processes did not work together to create an efficient and detailed output, then our game would not be efficient and smooth. The different modules in Figure 2 have different levels of cohesion.

Input handler – High cohesion, focused primarily on interpreting user inputs while also connected to every other module.

GUI/Audio – High cohesion, focused on rendering the graphics and playing feedback sounds/effects for the user.

State Manager – High cohesion, focused on managing state updates, does not execute code or render graphics.

Game Logic Engine – High cohesion, focused on processing user input logic and communicating with the state manager, does not directly deal with graphics or sound.

As you can see, each module has a main focus while all working together for the main goal of creating a working video game.

Module Coupling

Coupling is how much a module is affected by the changes undergone in a related module (GeeksforGeeks, 2024). Low coupling is usually desired for video games. Our code needs to be able to undergo edits where necessary so that we are not held back by the high interconnection of our classes. If that were the case, code edits would not be as easy, and a lot of code would need edited/replaced. Code scalability as we grow our project over the semester is important, the ability to add on and edit code smoothly. So, lower coupling is the way to go.

Input handler – Moderate coupling, manages input and outputs but requires communication/sharing with the other modules. Without other modules, the input handler would not have much of a purpose.

GUI/Audio – Moderate coupling, depends on the state manager to update what should be output to the screen.

State Manager – Moderate coupling, depends on the GUI/Audio module to take in its response to output to the screen. Also depends on the game logic engine to calculate the next state actions.

Game Logic Engine – Moderate coupling, depends on the state manager to process the next game state.

Our coupling is not as low as it could go, but we believe it is at an okay standpoint for our video game. Our game is simple. These modules work together while also having independently assigned tasks.

Design Analysis

Data Flow/Transaction Analysis Diagram

A diagram of a process

Description automatically generated

Figure 3: Data Flow Diagram

This figure shows the basic surface level flow of transactions between the user and the game. It depicts the possible outcomes. The user starts at the menu and can hit the quit button which will simply exit the application, or can alternatively hit the settings button which will go through the user’s previously saved UI customization options. Changes can be made and will be saved when the back button is hit, bringing the user back to the menu but through the data storing system first. When the user clicks play, the level selection will be brought up. Before this, the user’s previous progress and currently unlocked levels must be loaded from where it is stored to display the possible level options. You may also go back from the play button to the main menu, which will also go through the data system on the chance that changes were made and should be saved. If a valid/ unlocked level is selected, it is loaded from the system. The user can arrange the pseudocode blocks as much as they wish, the output will only be processed once the run/test button is pressed. Once the user presses this button, the arrangement will be processed with the game’s logic engine. If the player’s arrangement is incorrect, but not fatal, the user has more tries at arranging the pseudocode and can try to resubmit something. If the player’s arrangement is incorrect and leads to fatality and the level restarted. This will bring you back to loading the level. If the level is passed, this progress is stored and updated and unlocks the next level.

Design Organization

Detailed tabular description of Classes/Object

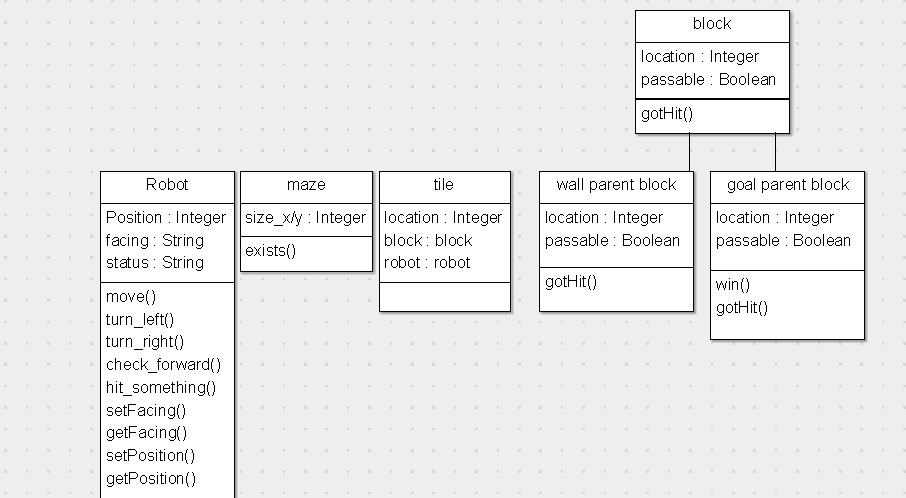


Figure 4: Class Diagram

Class Name:

Robot

Class Description:

The Robot is the primary class that the user will interact with. The robot will be the object that the user sees and attempts to guide through the maze with their blocks of code.

Class Data Members:

Position X: int

Position Y: int

Facing: string

Status: int

Class Member Functions:

setPosition()

getPosition()

setFacing()

getFacing()

move()

turnLeft()

turnRight()

checkForward()

hitSomething()

Class Name:

Block

Class Description:

The block class is the parent class for the wall and the goal class. The block itself will be non-interactable. We still include it because it is more convenient to have the parent class included in the tile class and use inheritance than it would be to make the tile class be able to hold several different classes. The downside of doing it this way is that there can only be one block or child class in a tile, so no doubling up.

Class Data Members:

Location X: int

Location Y: int

Passable: boolean

Class Member Functions:

gotHit()

Class Name:

Wall

Class Description:

The wall class is a child class of Block. The wall class differs from the Block because the wall is visible and impassable. It will not allow the robot to occupy the same tile, nor will the wall allow the robot to pass.

Class Data Members:

Location X: int

Location Y: int

Passable: boolean

Class Member Functions:

gotHit()

Class Name:

Goal

Class Description:

The goal is the objective of each maze. The robot’s objective is to reach the goal tile.

Class Data Members:

Location X: int

Location Y: int

Passable: Boolean

Class Member Functions:

gotHit()

win()

Class Name:

Tile

Class Description:

The tile is the class that will hold the robot and a block class/child class. The tile will act as an in-between for the robot and a block.

Class Data Members:

Robot: Robot

Block: Block

Location X: int

Location Y: int

Class Member Functions:

Update()

Class Name:

Maze

Class Description:

The maze will be the physical map that the robot can traverse and that the user can see.

Class Data Members:

Width: int

Height: int

mapNumber: int

Class Member Functions:

Exists()

Robot Class

setPosition():

INPUT:

The input will be two integers, x and y.

OUTPUT:

This function will update the x and y coordinate of the robot.

RETURN PARAMETERS:

There is no return for this function.

TYPES:

Integers will be the only type used.

getPosition():

INPUT:

There will be no input for the function.

OUTPUT:

There will not be an output for this function.

RETURN PARAMETERS:

The return parameters are an x and y integer for the location of the robot.

TYPES:

Integers will be the type used.

move():

INPUT:

There will be no input for the function.

OUTPUT:

This function will call getFacing() and getPosition(). It will use the direction of facing to determine whether x or y gets updated, positive or negative, for the direction the bot is facing.

RETURN PARAMETERS:

There will not be a return parameter.

TYPES:

Integers and strings will be used.

setFacing():

INPUT:

This function will accept a string.

OUTPUT:

This function will update the facing variable of the robot to, North, South, East or West.

RETURN PARAMETERS:

There will not be a return for this function.

TYPES:

This function will use strings.

getFacing():

INPUT:

There will be no input for the function.

OUTPUT:

There will not be an output for this function.

RETURN PARAMETERS:

The getFacing function will return a string, either North, South, East or West.

TYPES:

This function will use strings.

turnLeft():

INPUT:

There will be no input for the function.

OUTPUT:

The turn left will call getFacing(), determine what new direction it should face, (ex from south to east) then update the robot with setFacing().The robot should appear to turn left on the screen.

RETURN PARAMETERS:

There will not be a return value.

TYPES:

This function will use strings.

turnRight():

INPUT:

There will be no input for the function.

OUTPUT:

The turn left will call getFacing(), determine what new direction it should face, (ex from south to west) then update the robot with setFacing(). The robot should appear to turn right on the screen.

RETURN PARAMETERS:

There will not be a return value.

TYPES:

This function will use strings.

checkForward():

INPUT:

There will be no input for the function.

OUTPUT:

This function will not directly affect the robot.

RETURN PARAMETERS:

This function will call getPosition and getFacing. It will look one tile ahead of its location and direction facing, and see if that spot is occupied. It will return a string with the item name or ‘nothing.’

TYPES:

This function will use strings and integers.

hitSomething():

INPUT:

There will be no input for the function.

OUTPUT:

This function will not directly affect the robot.

RETURN PARAMETERS:

This function will call getPosition, it will see if it occupies the same location as an object. It will then call the interaction function of the object it hit. For us, we will do tile[x][y].block.gotHit();.

TYPES:

This function will use strings and integers.

Block Class

gotHit():

INPUT:

There will be no input for this function.

OUTPUT:

This function is a parent class, there will be no output for this class, but there will be for the child classes.

RETURN PARAMETERS:

There is no return for this function.

TYPES:

The types will depend on the child classes.

wall Class

gotHit():

INPUT:

There will be no input for this function.

OUTPUT:

When the wall is hit, it will get the robot’s direction facing and position, then update it to its previous location by increasing or decreasing its x or y value accordingly.

RETURN PARAMETERS:

There is no return for this function.

TYPES:

This will use integers and strings.

wall Class

gotHit():

INPUT:

There will be no input for this function.

OUTPUT:

When the wall is hit, it will get the robot’s direction facing and position. Then update it to its previous location by increasing or decreasing its x or y value accordingly.

RETURN PARAMETERS:

There is no return for this function.

TYPES:

This will use integers and strings.

goal Class

gotHit():

INPUT:

There will be no input for this function.

OUTPUT:

When the goal is hit, it will call the win function.

RETURN PARAMETERS:

There is no return for this function.

TYPES:

This will use integers and strings.

win():

INPUT:

There will be no input for this function.

OUTPUT:

When win is called, it will display a victory screen then load the next level.

RETURN PARAMETERS:

There is no return for this function.

TYPES:

This will use integers and strings.

Tile Class

update():

INPUT:

There will be no input for this function.

OUTPUT:

The update function will check to see if there is a robot present, then check if there is a block class. If there is, it will call the gotHit function of the child class that is there.

RETURN PARAMETERS:

There is no return for this function.

TYPES:

This will use integers and strings.

maze Class

exists():

INPUT:

There will be no input for this function.

OUTPUT:

The maze class will make sure that the size is > 0.

RETURN PARAMETERS:

Exists will return a 1 if size x and y > 0.

TYPES:

This will use integers.

Modules Used

We will use some modules that are provided in Godot, we do not currently know what additional modules we will use at this time other than the previously mentioned modules/classes.

Files Accessed

The user will download our program and export it from a zip file. Our program will store player saved data on .txt files for ease of use.

Real-Time Requirements

We want the game to feel smooth so it should work at a reasonable speed. Most computers will be able to run our game smoothly with no problem, as it is small, 2D, and non-taxing.

Messages

All messages will be displayed to the user in a pop-up window. Messages such as “Level completed,” “Did not find goal,” “Good job!”, etc.

Narrative / PDL

Menu:

* Play
* Settings
* Quit

PLAY

* + New game
  + Load game

New Game

* Program
* Run
* Quit

Run

* + Win
  + Lose

Win

* Next level
* Quit

Quit

* + bye

Programming language/Reuse/Portability

We will use GODOT for our project. This engine uses its own language, but also allows for the use of C/C++. GODOT is expected to be around for a while so the reuse of the program/code should be supported long term. GODOT supports other platforms besides windows.

Implementation Timeline

A diagram of a diagram

Description automatically generated

Figure 5: Implementation Flow Diagram

This is our simplified implementation timeline. It includes a simple example of how our timeline might look once next semester starts. It includes every month we have to work on the project. Starting in January, we will begin the implementation of our ideas. This will also include designing the map, levels, character, UI, etc. Deciding how baseline things will look and act. Next, in February, we will continue with implementation but with the additional level design. We may have the basics out of the way some by February, which will bring us to level design. Designing our puzzles, mazes, and obstacles and how they will be passed. March will include more implementation of these ideas, with the beginning of outsider testing. Our game should be functional enough by this time to be tested by outside players. Then, in April, the game should be near completion. All ideas should be founded and implemented by this time. At this point, the main things to be done will be bug testing/revising code and simple problems. We will gauge how users interact with our game here. In May, our game will be fully completed. All that is left to do at that point is present it to the board.

Design Testing

The design’s testing will be ongoing throughout the whole project. After programming anything, it is essential to proofread the code and run it to see if it is functional. Testing along the way will ensure that we do not miss small errors. It will also ensure that our code does as it should, it does as the user expects it to. The team will complete the main chunk of revision and error checking. After the code is more functional and cohesive for actual gameplay, we will have outside sources come in to help. These friends and family members will be essential for getting an outside user insight on our gameplay. They will help us find mistakes that we, as programmers, might not have found otherwise. Getting inputs from the team and various outside perspectives will help our game become complete.

**Resources**

*Godotengine/godot*. (2020, September 29). GitHub. <https://github.com/godotengine/godot>

*Introduction — Godot Engine latest documentation*. (2024). Readthedocs.io. <https://godot-doc.readthedocs.io/en/3.0/about/introduction.html>

GeeksforGeeks. (2024, October 4). *Differences between coupling and cohesion - software engineering*. https://www.geeksforgeeks.org/software-engineering-differences-between-coupling-and-cohesion/

**Appendix: Team Details**

This paper was written in a group manner including the whole ComPuzzle team. Different sections were allocated to each member to be completed; this is so that we ensure fairness and contribution of all. Each member’s section was reviewed by the other members and edits were made where necessary. Topics were decided on as a group before we started writing, to ensure we are all on the same page for how our game design should be. The workflow leader assigned the work to the team. Here is the list of each member’s main contributions.

* Abigail Dehart- Design Analysis, Data Flow or Transaction Analysis, Decision: Programming Language/Reuse/Portability, Implementation Timeline, Design Testing, Resources, Appendix: Team Details, Appendix: Writing Center Report
* Brianna Dulik- Abstract, Description of Document, Purpose and Use, Ties to Specification Document, Intended Audience, Appendix: Workflow Authentication, Modules Used, Files Accessed
* Brandon Mastin- Project Block Diagram, Design Details, System Modules and Responsibilities, Architectural Diagram, Module Cohesion, Module Coupling, Messages, Narrative/PDL, Real-Time Requirements
* Cameron McGill- Workflow Leader. Design Organization, Detailed Tabular Description of Classes/Objects, Description, Data Members/Types/Constraints, Member Function Listing/Description, Functional Description, Input/Output/Return Parameters/Types

**Appendix: Workflow Authentication**

I, Abigail Dehart, am in agreement with the design methods and proposals presented in this document; the authenticity of this paper is genuine and was discussed by all.

 11/28/2024

I, Brianna Dulik, am in agreement with the design methods and proposals presented in this document; the authenticity of this paper is genuine and was discussed by all.

 11/28/2024

I, Brandon Mastin, am in agreement with the design methods and proposals presented in this document; the authenticity of this paper is genuine and was discussed by all.

 11/28/2024

I, Cameron McGill, am in agreement with the design methods and proposals presented in this document; the authenticity of this paper is genuine and was discussed by all.

A black and white logo

Description automatically generated 11/28/2024

**Appendix: Writing Center report**

Completed via email by Emily Esposito from foundrywritingcenter@pennwest.edu.