Design Documents division/actual document:

Anton Nosov 0727096

Vincent Yong 0744993 (tell your group to remember to log off :) )

Josiah Menezes 0757039

Santiago Florez 0758977

Yasser Kassim 0769076

Santiago Florez 0758977

Alex Newton 0761500

Andrew Macaulay 0726671

INTRODUCTION

Tower Defense is a strategy game. The player has to be able to carefully plan a strategy that will make him succeed the enemy’s waves of attack. The player controls the towers that are set along the edges of a path. As the player sets fire to creeps coming from the start of the path, money is awarded to the player, which can be use to make more towers or upgrade the existing ones.

In an actual game the player chooses a map that contains a path. Units (creeps) begin to emerge and start walking along the route towards the end. The player has to stop them from reaching the end line or else a life will be taken from the player, and when all lives are gone the game will be over. Simple game and controls combine to give an easy to learn but engaging play experience.

1.1 Goals and Objectives

- The players’ goal is always to be ready for different waves of attack by building towers among the edges of the path. These towers will destroy the enemies preventing them from reaching the end line.

- As soon as all the creeps are gone from a wave. A new wave will come containing more creeps than the one before.

- Advanced players will attempt to destroy all the creeps before any of them reach the end of the path.

- There is a limit on the amount of towers that can be made. Every time the player destroys an enemy, money will be awarded, this will allow the player to invest that money on either upgrading a tower or building a new one. This will prepare the player for upcoming attacks.

- The Objective is to survive all the attacks of the enemies and it is met by managing the towers the most efficient way.

1.2 Statement of scope

- Program is coded in java language

- Run as a java program

- User is capable of selecting the path

- Functionality of towers is determined by button clicked at running time

o Every time the player wants to build a new tower they need to clicked on the spot they want it

o In order to upgrade a tower, the player clicks on an existing one and upgrades it

- Outputs:

o Wins: whenever a wave of creeps is destroyed completely

o Looses: Whenever a creep reaches the end line. Certain number will be game over.

- Game is composed of:

o Enemies: Come out from the start of the path in different waves

o Towers: Players ability to destroy the enemy(creeps)

o Goals: Destroy all creeps before reaching end line.

o Scores: Players progress, and method to upgrade and/or buy towers

o Lives: Every time a creep crosses the end line a live will be subtracted

- Players abilities:

o Fire from each tower whenever it is clicked

o Upgrade and/or buy more towers to prepare for more intense waves

1.3 Software context

- This software is to be available to the public in different applications stores such as android and apple store. It also intended to be a web browser game using add on store, Internet explorer and safari.

1.4 Major constraints

- In order to use this software the hardware has to support Java

- It has to meet the minimum requirements of the program.

- Screen resolution has to be met

Section 2: Yasser, Anton

2.1 Internal software data structures

* Methods for calculating currency, highscore, lives.
* Methods for all tower(range, damage/damage type, cost, attack type) types + Sell and Build towers.
* Methods for all enemy(health, speed, armor/armor type, gold value) types + Enemy and wave counters.

The main 3 internal data structures we are going to use in our project are: Gameplay data, Tower data, and Enemy data. Note that data structures in our project are going to be in the form of Java methods.

Gameplay data structure/method:

The role of the Gameplay data structure is to store and update gold(game currency), player lives, and enemy wave counter. The gold data will be updated each time an enemy is killed or a tower is built. Player lives will be updated each time an enemy passes through the exit tile. The wave counter is going to be a simple 'current wave/total waves' type of counter. This structure will interact with both Tower and Enemy data to make the necessary updates during gameplay.

Tower data structure/method:

This structure will have multiple instances of it for every type of tower in the game. The basic tower stats will be: type of damage, amount of damage, attack range, cost, resale value and upgrade level. As the towers get upgraded, all of their base values will update accordingly. This structure will be interacting with Enemy data structure the most.

Enemy data structure/method:

Just as the Tower structure above, this structure will have multiple instances for different enemies as well. The enemy stats will be: max health, current health, type of armor, amount of armor, movement speed and reward value. As the enemies get attacked by the towers the values will get updated.

2.2 Global software data structures

* Text file to save high scores
* Text file to save user data between levels(?)

The only global data structure in our project will be a text file to store highscores of all the users who have played the game. The file will be updated every time the user beats a level. If the user has the highest score, it will be put on top of the list. Otherwise it will be added below the score that is higher. The list of highscores will be 10 records maximum, with lower scores being replaced by higher scores. This structure will be accessed from the "High scores"

menu to display all current scores recorded.

2.3 Temporary data structures

One of the temporary data structures that we will be using is the enemy. Although their path and their strength are predefined, they are temporary for a specific instance. Another temporary data structure that will be included is the towers that will be used to destroy the enemies.

2.4 Database description

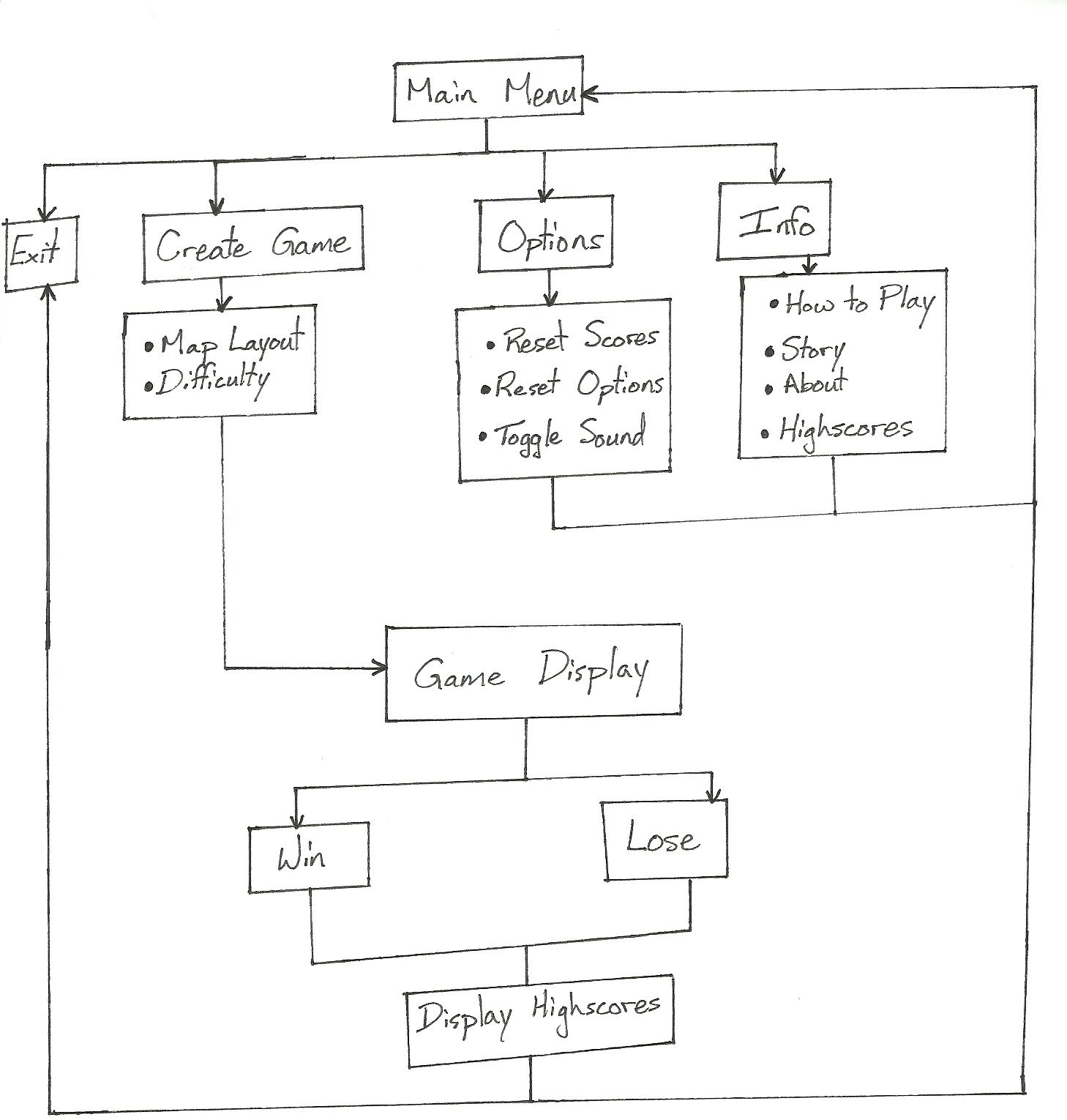
Our program may contain a database that will be used to set the settings for the game. This will be used to control things such as; difficulty of the levels or the strength of the characters, possibly a time limit, potential toggling of any added sound effects, and possibly a high score list.

Section 3:

3.1 - Program Structure

The structure of the program, as seen in 3.1.1, was chosen to have a good up front menu choice for the user. Allowing them to change options, such as: reset scores, reset options and toggle sound from the main menu was an important feature we wanted. Having an info menu option allows the user to read up on how to play, the background story to the game, and the about option, as well as look at the current high-scores. Both the options and info choices return the user back to the main menu once finished in the respective submenu. Create game brings the user to a submenu requesting the map layout and the difficulty of the game to be played. On Top of these three menu choices, there is an exit game choice which ends the program. Once the player has selected the map and difficulty, the game loads. Once the user has played through the game, they get sent to a respective Win or Lose screen, afterwards a display of the high-scores. From here, the user can choose whether to return to the main menu, or to exit the game.

3.1.1 - Architecture Diagram



3.1.2 - Alternatives

There are many alternatives to the current structural design. Many of these include possible extra features, which range from a colour change featured in the options submenu, to having a save feature which in turn would create a need for a "Continue Game" option in the main menu. As for more aesthetic alternatives, the info submenu could be accessed through the options submenu, reducing some clutter on the main menu. Although these are reasonable alternatives, the need for extra features is not a necessity to the game, and the info submenu is a better fit on the main menu.

->3.2 Josiah

3.2.Main Menu

3.2.1.Main Menu

The very start of the program is the Operating System telling the program to run. Global variables and the GUI are initialized here. The main menu pops up. If a high score has been created and is stored in the correct location the high score button will be available like every other button. From here one is able to access options, new game, information about the product, and exit the program.

3.2.2.Main Menu

The user sees the logo of the program, a simple background and four buttons before him. They can be used to begin the game, open the options menu to change settings, see critical information, and leave the game.

3.2.3.1.Main Menu.Play

Press this button to Begin the process of starting to play a game. Takes mouse click. Outputs the event of pressing the button.

3.2.3.2.Main Menu.Play

Upon button\_click do method\_x

3.2.3.3.Main Menu.Play

Game must be initialized, main menu must be created. Relevant button and related event needs to exist. The method it connects to needs to exist.

3.2.3.4.Main Menu.Play

No immediate local data structures.

3.2.3.5.Main Menu.Play

No known current performance issues.

3.2.3.6.MainMenu.Play

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes.

3.2.3.1.MainMenu.Info

Press this button to learn more about the game. Takes mouse click. Outputs the event of pressing the button.

3.2.3.2.MainMenu.Info

Upon button\_click do method\_x

3.2.3.3.MainMenu.Info

Game must be initialized, main menu must be created. Relevant button and related event needs to exist. The method it connects to needs to exist.

3.2.3.4.MainMenu.Info

No immediate local data structures.

3.2.3.5.MainMenu.Info

No known current performance issues.

3.2.3.6.MainMenu.Info

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes.

3.2.3.1.MainMenu.Options

Press this button to change settings within the game. Takes mouse click. Outputs the event of pressing the button.

3.2.3.2.MainMenu.Options

Upon button\_click do method\_x

3.2.3.3.MainMenu.Options

Game must be initialized, main menu must be created. Relevant button and related event needs to exist. The method it connects to needs to exist.

3.2.3.4.MainMenu.Options

No immediate local data structures.

3.2.3.5.MainMenu.Options

No known current performance issues.

3.2.3.6.MainMenu.Options

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design.

3.2.3.1.MainMenu.Exit

Press this button to change settings within the game. Takes mouse click. Outputs the event of pressing the button.

3.2.3.2.MainMenu.Exit

Upon button\_click do method\_x

3.2.3.3.MainMenu.Exit

Game must be initialized, main menu must be created. Relevant button and related event needs to exist. The method it connects to needs to exist.

3.2.3.4.MainMenu.Exit

No immediate local data structures.

3.2.3.5.MainMenu.Exit

No known current performance issues.

3.2.3.6.MainMenu.Exit

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design.

3.2.1.CreateGame

This process is for creating the game. It has permanently set settings chosen in it. Initializes world. This section of the program chooses the settings. There are numerous settings to be chosen. Upon clicking next begins creating the chosen settings and creates the world.

3.2.2.CreateGame

User sees a series of buttons before them. They control the map layout and difficulty of the game. Increasing or decreasing the difficulty will control how hard the opponents are to kill but will have no effect on the quantity or gold gain.

3.2.3.1.CreateGame.layout

Press this button to change the layout of the map that will be played on. Takes mouse click. Outputs the event of pressing the button.

3.2.3.2.CreateGame.Layout

Upon button\_click do method\_x

If map = =x, map=y

3.2.3.3.CreateGame.Layout

Program must be initialized, supporting functions need creation. Methods need to exist for the buttons to link to. Must have GUI elements, there must be valid options selected.

3.2.3.4.CreateGame.Layout

Storage for the options chosen

3.2.3.5.CreateGame.Layout

No known current performance issues.

3.2.3.6.CreateGame.Layout

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Knowledge of data types.

3.2.3.1.CreateGame.Difficulty

Press this button to change the difficulty that the game will be played on. Takes mouse click. Outputs the event of pressing the button.

3.2.3.2.CreateGame.Difficulty

Upon button\_click do method\_x

Set.difficulty(hard)

3.2.3.3.CreateGame.Difficulty

Program must be initialized, supporting functions need creation. Methods need to exist for the buttons to link to. Must have GUI elements, there must be valid options selected.

3.2.3.4.CreateGame.Difficulty

Storage for the options chosen

3.2.3.5.CreateGame.Difficulty

No known current performance issues.

3.2.3.6.CreateGame.Difficulty

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Knowledge of data types is useful in this scenario

3.2.3.1.CreateGame.Start

Press this button to load the map and begin the game. Takes mouse click. Outputs the event of pressing the button.

3.2.3.2.CreateGame.Start

Upon button\_click do method\_x

Method\_x: start game, load game initialization

Set.difficulty(hard)

3.2.3.3.CreateGame.Start

Program must be initialized, supporting functions need creation. Methods need to exist for the buttons to link to. Must have GUI elements, there must be valid options selected.

3.2.3.4.CreateGame.Start

Storage for the options chosen

Storage for initial values

3.2.3.5.CreateGame.Start

No known current performance issues.

3.2.3.6.CreateGame.Start

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Knowledge of data types is useful in this scenario

3.2.1.GameDisplay

This process is for Playing the game. It has permanently set settings utilized in it. There are many events that occur in this section of the architecture. It records your current score, wave level, controls your towers, moves minions, reduces life if you lose lives, spawns mobs, gains gold. Everything involved in the game itself is held here.

3.2.2.GameDisplay

User sees a series of buttons before them. They control what tower you are choosing to build. In addition there will be a menu option to leave the game immediately, a panel that shows gold count and wave number, another showing lives remaining.

3.2.3.1.GameDisplay.Tower

Press one of these buttons to select a tower to create. Takes mouse click. Outputs the event of pressing the button.

3.2.3.2.GameDisplay.Tower

Upon button\_click do method\_x

Method\_x – create tower(i) at location mouseclick

3.2.3.3.GameDisplay.Tower

Program must be initialized, supporting functions need creation. Methods need to exist for the buttons to link to. Must have GUI elements, there must be valid options selected. Gold values must be acceptable. Must be a permissible location to create tower

3.2.3.4.GameDisplay.Tower

Methods for tower creation must exist. Must store where legal tower placements are, this will change based on the existence of towers.

Storage for initial values

3.2.3.5.GameDisplay.Tower

No known current performance issues.

3.2.3.6.GameDisplay.Tower

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Knowledge of data types is useful in this scenario.

3.2.3.1.GameDisplay.Upper Interface

Does not allow for user input. This component holds the values like current gold, wave number, and lives remaining. It also allows the user to pause the game.

3.2.3.2.GameDisplay.UpperInterface

On\_mob\_kill-gold++

On\_mob\_end-life--

3.2.3.3.GameDisplay.UpperInterface

Program must be initialized, supporting functions need creation. Methods need to exist to update interface. Structures must exist to hold interface values.

3.2.3.4.GameDisplay.UpperInterface

Simple yet accessible variables could be used to hold the gold value, life total, wave number and current score.

3.2.3.5.GameDisplay.UpperInterface

No known current performance issues.

3.2.3.6.GameDisplay.UpperInterface

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Knowledge of data types is useful in this scenario. Space onscreen is a limitation. The ability to pause simulations is a critical function and will require a large amout of work.

3.2.3.1.GameDisplay.Board

This is the physical playing field. It can be interacted with by either placing towers upon it or possibly selling them to return one’s investment in them. There is a predetermined path for the creeps that spawn to follow. They will be fired upon by the placed towers. In this way the two parts interact. There will have to be checks to ensure the placement of the tower is legal

3.2.3.2.GameDisplay.Board

update\_gameStatus();

3.2.3.3.GameDisplay.Board

Program must be initialized, supporting functions need creation. Methods need to exist for each square to link to. Must be clearly visible to the user, things must be distinct and apart from each other

3.2.3.4.GameDisplay.Board

Storage for what is where on the map, where legal tower placements are, where towers exist and how much they can be sold for.

3.2.3.5.GameDisplay.Board

No known current performance issues.

3.2.3.6.GameDisplay.Board

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Knowledge of data types is useful in this scenario. Time is a huge constraint here. Personal knowledge of code structure is a huge thing here.

3.2.3.1.GameDisplay.creep

This component contains the player’s enemies. They move in a linear path to a distant location. They intend to make it without triggering their ‘death.’

3.2.3.2.GameDisplay.creep

Upon\_Wave\_start: for(x){spawn creep}

Upon\_creep\_0:remove creep from game, award gold.

3.2.3.3.GameDisplay.creep

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Must be clear and distinct from towers, their defeat should be clear to the player.

3.2.3.4.GameDisplay.creep

Storage for all current creeps’ health values, states, and locations.

3.2.3.5.GameDisplay.creep

No known current performance issues.

3.2.3.6.GameDisplay.creep

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Knowledge of data types is useful in this scenario.

3.2.1.HScore

This process is called after the game ends. It has permanently stored scores within it. It can also be accessed from the info menu. This menu allows a user to view the current and top high scores.

3.2.2.Hscore

The user will see a series of names and values before them representing the highest scores ever achieved in this game. If they have recently completed the game and won they will have the option of placing their name in the list. Displays whether they user won or lost the game if they are accessing this from ending the game.

3.2.3.1.Hscore.Score

This component holds the highest of scores. With it the user is also able to see the most recent high scores.

3.2.3.2.Hscore.Score

Display:highscore(0)

3.2.3.3.Hscore.Score

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Each score must clearly show which name it is related to.

3.2.3.4.Hscore.Score

There needs to be storage for all the high scores, this must be external to the program’s operation in order to be accessible after closing.

3.2.3.5.Hscore.Score

No known current performance issues.

3.2.3.6.Hscore.Score

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Knowledge of reading and writing to file is absolutely necessary here. Screen space is a limitation here.

3.2.3.1.Hscore.Menu

This subcomponent sends the user back to the main menu.

3.2.3.2.Hscore.Menu

Upon\_button\_press: open\_menu\_Window

3.2.3.3.Hscore.Menu

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible and linked to main menu

3.2.3.4.Hscore.Menu

No large or special storage system needed for this subcomponent

3.2.3.5.Hscore.Menu

No known current performance issues.

3.2.3.6.Hscore.Menu

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Window/GUI manipulation is an important skill needed here.

3.2.1.Options

This process opens up the options menu. It includes options for color, high score reset, options reset, and to toggle the sound running in the program.

3.2.2.Options

The user will see a small menu with a series of buttons for choosing options on it. One will change the menu’s color, another resets the score, another resets all options, and a final one toggles the in game sound. In addition to those buttons there is a title of Options for the menu and a back button that will return the user to the main menu.

3.2.3.1.Options.Color

This subcomponent will permanently change the color of the menus.

3.2.3.2.Options.Color

Upon\_buttonx \_click: do method\_colorChange()

3.2.3.3.Options.Color

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. Color changing method needs to be connected and referenced on every menu.

3.2.3.4.Options.Color

Small variable is likely to be what needs changing to determine menu color.

3.2.3.5.Options.Color

No known current performance issues.

3.2.3.6.Options.Color

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. System architecture needs to be built to allow this with ease.

3.2.3.1.Options.Score

This subcomponent will permanently clear the high score records.

3.2.3.2.Options.Score

Upon\_buttonx \_click: do method\_clearHScore()

3.2.3.3.Options.Score

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. File that accesses high score needs deletion.

3.2.3.4.Options.Score

No data structure needed here, only the deletion of a text file.

3.2.3.5.Options.Score

No known current performance issues.

3.2.3.6.Options.Score

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Need to ensure that program does not fail when presented with the lack of a file or a blank file.

3.2.3.1.Options.Option

This subcomponent will reset the chosen options to their default values.

3.2.3.2.Options.Option

Upon\_buttonx \_click: do method\_resetOption

3.2.3.3.Options.Option

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. Options need default values.

3.2.3.4.Options.Option

Variables used by other buttons will be changed

3.2.3.5.Options.Option

No known current performance issues.

3.2.3.6.Options.Option

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. No other major constraints

3.2.3.1.Options.Sound

This subcomponent will decide whether or not there is sound in the game. It is simply a yes or no option.

3.2.3.2.Options.Sound

Upon\_buttonx \_click: do method\_soundToggle

3.2.3.3.Options.Sound

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. Sound needs to immediately stop or start when button pressed.

3.2.3.4.Options.Sound

A simple Boolean will be used here.

3.2.3.5.Options.Sound

No known current performance issues.

3.2.3.6.Options.Sound

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Should the sound simply be muted rather than disabled?

3.2.3.1.Options.Menu

This part will return the user to the main menu. The main menu is the only way to access this screen.

3.2.3.2.Options.Menu

Upon\_buttonx \_click: close\_option(); open\_menu();

3.2.3.3.Options.Menu

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. Main menu needs to work. Button needs to link to main menu.

3.2.3.4.Options.Menu

No data structures but for those that hold the options will be transferred here. None will be created or changed within this component

3.2.3.5.Options.Menu

No known current performance issues.

3.2.3.6.Options.Menu

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design.

3.2.1.Info

This process opens up the information menu. It includes buttons that will detail information on how to play the game, what the background story of the game is, general information about the game and its creators, in addition one can read the list of high scores from this menu.

3.2.2.Info

The user will see a small menu with a series of buttons for choosing information to display on it. One will reveal how to play the game, another details the background story, a third button reveals information about the game, and the last two buttons direct the user to the high score menu and return to the main menu. In addition to those buttons there is a title for the menu labeled ‘Info.’

3.2.3.1.Info.Menu

This part will return the user to the main menu. The main menu is the only way to access this screen.

3.2.3.2.Info.Menu

Upon\_buttonx \_click: close\_info(); open\_menu();

3.2.3.3.Info.Menu

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. Main menu needs to work. Button needs to link to main menu.

3.2.3.4.Info.Menu

None will be created or changed within this component, neither will there be any mass data access.

3.2.3.5.Info.Menu

No known current performance issues.

3.2.3.6.Info.Menu

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design.

3.2.3.1.Info.How

This subcomponent will show the user how to play the game through a pop-up window. It is critical to inform a new user how your system works in a precise and efficient yet detailed manner.

3.2.3.2.Info.How

Upon\_buttonx \_click: open\_howto();

3.2.3.3.Info.How

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. Submenu system needs to work, their links need to function.

3.2.3.4.Info.How

Minimal use of data structures in this area, will be using a couple to track progress through tutorial.

3.2.3.5.Info.How

No known current performance issues.

3.2.3.6.Info.How

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Length and tone of tutorial for the uninformed is a constraint on the project.

3.2.3.1.Info.Story

This part will provide the user with information on the “supposed reason” behind the game’s events.

3.2.3.2.Info.Story

Upon\_buttonx \_click: open\_story(); createButton\_storyClose(); Create text: “a long time ago…”;

3.2.3.3.Info.Story

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. Button within menu must work and be linked back to the info menu.

3.2.3.4.Info.Story

None will be created or changed within this component, neither will there be any mass data access.

3.2.3.5.Info.Story

No known current performance issues.

3.2.3.6.Info.Story

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Must be within reason.

3.2.3.1.Info.About

This part will provide the user with information regarding the game and its creators.

3.2.3.2.Info.About

Upon\_buttonx \_click: open\_about(); createButton\_aboutClose(); Create text: “Created for CIS3250…”;

3.2.3.3.Info.About

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. Button within menu must work and be linked back to the info menu.

3.2.3.4.Info.About

None will be created or changed within this component; neither will there be any mass data access.

3.2.3.5.Info.About

No known current performance issues.

3.2.3.6.Info.About

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. The information contained within must be within the realm of possibility and at least partially true.

3.2.3.1.Info.HScore

This part will provide the user with the high score screen. It does not feature a screen for returning to the information screen and as such will be placed lowest on the screen of the relevant buttons, just above the menu button.

3.2.3.2.Info.HScore

Upon\_buttonx \_click: open\_HighScoreWindow(hsfile.txt);

3.2.3.3.Info.HScore

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Button to click must be visible. HighScore Menu must be working.

3.2.3.4.Info.HScore

No data structure will be used significantly by this subprocess.

3.2.3.5.Info.HScore

No known current performance issues.

3.2.3.6.Info.HScore

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Must read from file.

3.2.1.Quit

This process terminates the program. Any and all high scores are saved.

3.2.2.Quit

Upon clicking a button that is labeled quit, the user will be prompted with a confirmation window asking if they are sure they wish to leave. If they click “no” they will return to what they were doing. Otherwise the program will close.

3.2.3.1.Quit.Menu

This part will confirm the user’s desire to leave the program

3.2.3.2.Info.Menu

Button\_click\_x:exit()

Button\_click\_y:close\_exit();

3.2.3.3.Info.Menu

Program must be initialized, supporting functions need creation. Methods need to exist for the events to reference. Buttons to click must be visible. Must have close function linked to close.

3.2.3.4.Info.Menu

High scores will be written if they have not been already at this stage.

3.2.3.5.Info.Menu

No known current performance issues.

3.2.3.6.Info.Menu

Main constraints placed on the button: GUI knowledge and possibilities, lack of classes. Need for effective design. Simplicity, do not allow for accidental termination of program. Cautious button placement.

3.3 Software Interface Description

The software's interface(s) to the outside world are described. These in our case would be the database interface, actually interactive interface for the user, and the interface connecting the program with other machines.

3.3.1 External machine interfaces

At the present time no External Machine interfaces are needed as the program is not going to be going over a network for any of the functionality.

3.3.2 External system interfaces

The only interface that would need to be set up for this program that could be considered External is the interface between the database and the program itself to actually pull data from it and update the database. This is essential as it is the way for us to store settings, high scores, etc. and will lead to a better overall experience for the user.

3.3.3 Human interface

The human interface for the program will allow the user to easily navigate through the various menus for options and for the extra features that could be toggled. The user is presented with most, if not all, of the necessary functions on the first user interface page(home page) to make it much more easy for the user to find what they really want to do in the program without having to delve into the menus and options further and further making the program more complicated for them to interact with. To make our interface with the human more accessible a way to navigate the menu with only one of the keyboard or the mouse should be possible.

Section 4: Vince, Justin

Section 5: Restrictions, limitations, and constraints

Restrictions - No use of classes or objects is allowed.

Limitations - … same as above.

Constraints - Java version used on the lab Macs.

Section 6: Testing issues (Do we have to predict this?) I think we do, but it shouldn’t be too hard

Section 7: Appendices

7.1

As of 26/10/12 completed design document.

7.2

Making the program an exe over a java file must be considered for ease of use and access.

7.3

No currently implemented design metrics.

7.4

No Supplemental information as of 26/10/12

JPanel Array List Code

int count=0;

ArrayList<JPanel> battleField = new ArrayList<>();

for(int i=0;i<25;i++){

for(int j=0;j<10;j++){

battleField.add(new JPanel());

battleField.get(count).setSize(40,40);

battleField.get(count).setLocation(i\*40,j\*40);

gridWindow.add(battleField.get(count));

count++;

}

}

In the Variables section

private static ArrayList<Integer> enemyPath = new ArrayList<>();

private static ArrayList<JButton> tower3 = new ArrayList<>();

private static ArrayList<JButton> tower2 = new ArrayList<>();

private static ArrayList<JButton> tower1 = new ArrayList<>();

In the Ingame section below battlefield panel creation

tower3.add(new JButton());

tower3.get(count).setSize(40,40);

tower3.get(count).setLocation((i\*40)-40,(j\*40)+50);

gridWindow.add(tower3.get(count));

tower3.get(count).setVisible(false);

tower3.get(count).setEnabled(false);

tower1.add(new JButton());

tower1.get(count).setSize(40,40);

tower1.get(count).setLocation((i\*40)-40,(j\*40)+50);

gridWindow.add(tower1.get(count));

tower1.get(count).setVisible(false);

tower1.get(count).setEnabled(false);

tower2.add(new JButton());

tower2.get(count).setSize(40,40);

tower2.get(count).setLocation((i\*40)-40,(j\*40)+50);

gridWindow.add(tower2.get(count));

tower2.get(count).setVisible(false);

tower2.get(count).setEnabled(false);

Tower Placement

towerOne.setSize(200, 72);

towerOne.setLocation(400, 490);

Action towerOneLocation = new AbstractAction(""){

@Override

public void actionPerformed(ActionEvent e){

for(int i=0;i<275;i++){

if((JButton)e.getSource()==tower1.get(i)){

battleField.get(i).setBackground(Color.ORANGE);

tower1.get(i).setEnabled(false);

tower1.get(i).setVisible(false);

int x=i/11;

int y=i%11;

mapLayout[y][x+1]="1";

gridWindow.repaint();

}

}

for(int i=0;i<26;i++){

for(int j=0;j<11;j++){

System.out.print(mapLayout[j][i]);

}

System.out.println("");

}

}

};

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

tower1.get(count).addActionListener(towerOneLocation);

count++;

}

}

Action addTowerOne = new AbstractAction("") {

@Override

public void actionPerformed(ActionEvent e) {

mouseClickCountOne++;

towerThree.setEnabled(false);

towerTwo.setEnabled(false);

if (mouseClickCountOne == 1) {

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

if (mapLayout[j][i].equals("0")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.GREEN);

tower1.get(count).setEnabled(true);

tower1.get(count).setVisible(true);

count++;

}

else if (mapLayout[j][i].equals("1")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.gray);

count++;

}

}

}

gridWindow.repaint();

}

else {

mouseClickCountOne = 0;

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

if (mapLayout[j][i].equals("0")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.blue);

tower1.get(count).setEnabled(false);

tower1.get(count).setVisible(false);

count++;

}

else if (mapLayout[j][i].equals("1")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.gray);

count++;

}

}

}

towerThree.setEnabled(true);

towerTwo.setEnabled(true);

gridWindow.repaint();

}

}

};

towerOne.addActionListener(addTowerOne);

gridWindow.add(towerOne);

gridWindow.repaint();

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

towerTwo.setSize(200, 72);

towerTwo.setLocation(600, 490);

Action towerTwoLocation = new AbstractAction(""){

@Override

public void actionPerformed(ActionEvent e){

for(int i=0;i<275;i++){

if((JButton)e.getSource()==tower2.get(i)){

battleField.get(i).setBackground(Color.PINK);

tower2.get(i).setEnabled(false);

tower2.get(i).setVisible(false);

int x=i/11;

int y=i%11;

mapLayout[y][x+1]="1";

gridWindow.repaint();

}

}

}

};

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

tower2.get(count).addActionListener(towerTwoLocation);

count++;

}

}

Action addTowerTwo = new AbstractAction("") {

@Override

public void actionPerformed(ActionEvent e) {

mouseClickCountTwo++;

towerOne.setEnabled(false);

towerThree.setEnabled(false);

if (mouseClickCountTwo == 1) {

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

if (mapLayout[j][i].equals("0")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.GREEN);

tower2.get(count).setEnabled(true);

tower2.get(count).setVisible(true);

count++;

}

else if (mapLayout[j][i].equals("1")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.gray);

count++;

}

}

}

gridWindow.repaint();

}

else {

mouseClickCountTwo = 0;

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

if (mapLayout[j][i].equals("0")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.blue);

tower2.get(count).setEnabled(false);

tower2.get(count).setVisible(false);

count++;

}

else if (mapLayout[j][i].equals("1")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.gray);

count++;

}

}

}

towerOne.setEnabled(true);

towerThree.setEnabled(true);

gridWindow.repaint();

}

}

};

towerTwo.addActionListener(addTowerTwo);

gridWindow.add(towerTwo);

gridWindow.repaint();

/\* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

//Button

towerThree.setSize(200, 72);

towerThree.setLocation(800, 490);

Action towerThreeLocation = new AbstractAction(""){

@Override

public void actionPerformed(ActionEvent e){

for(int i=0;i<275;i++){

if((JButton)e.getSource()==tower3.get(i)){

battleField.get(i).setBackground(Color.BLACK);

tower3.get(i).setEnabled(false);

tower3.get(i).setVisible(false);

int x=i/11;

int y=i%11;

mapLayout[y][x+1]="1";

gridWindow.repaint();

}

}

}

};

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

tower3.get(count).addActionListener(towerThreeLocation);

count++;

}

}

Action addTowerThree = new AbstractAction("") {

@Override

public void actionPerformed(ActionEvent e) {

mouseClickCountThree++;

towerOne.setEnabled(false);

towerTwo.setEnabled(false);

if (mouseClickCountThree == 1) {

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

if (mapLayout[j][i].equals("0")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.GREEN);

tower3.get(count).setEnabled(true);

tower3.get(count).setVisible(true);

count++;

}

else if (mapLayout[j][i].equals("1")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.gray);

count++;

}

}

}

gridWindow.repaint();

}

else {

mouseClickCountThree = 0;

count = 0;

for (int i = 0; i < 26; i++) {

for (int j = 0; j < 11; j++) {

if (mapLayout[j][i].equals("0")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.blue);

tower3.get(count).setEnabled(false);

tower3.get(count).setVisible(false);

count++;

}

else if (mapLayout[j][i].equals("1")) {

//battleField.get(count).setBorder(BorderFactory.createLineBorder(Color.black, 1));

// battleField.get(count).setBackground(Color.gray);

count++;

}

}

}

towerOne.setEnabled(true);

towerTwo.setEnabled(true);

gridWindow.repaint();

}

}

};

towerThree.addActionListener(addTowerThree);

gridWindow.add(towerThree);

gridWindow.repaint();

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

}

else if (mapLayout[j][i].equals("S")) {

count++;

}

else if (mapLayout[j][i].equals("F")) {

count++;

}

http://pastebin.com/Z7J69tyX