**Delta**

**Detailed Design Document**

**October 2, 2017**

**By: Team Delta**

**Jack Crowley**

**Peter Julain**

**Chris Kelly**

**Matt Richardson**

**Nick Zazula**

**Table of Contents**

**Cover Page 1**

**Table of Contents 2**

**Chapter 1: Introduction 3**

**Chapter 2: Architecture 4**

**Chapter 3: Module Design 6**

Section A: Game 6

Section B: Communication 8

Section C: Database 11

**Chapter 4: Abstract Data Types 12**

**Chapter 5: Database Design 15**

**Chapter 6: COMPRISES Diagram 16**

**Chapter 7: USES Diagram 17**

**Chapter 8: Other Diagrams 18**

Class Diagram 18

Sequence Diagram 25

Activity Diagram 26

Object Diagram 28

**Chapter 9: Integration Test Plan 29**

**Chapter 1: Introduction**

This is the Detailed Design Document for our application called *Delta.* This document will expand on our original specifications from the SRS. Here we will provide more in-depth information and have high, mid and low-level designs of our system. *Delta* is a game focused on teaching programming skills to the customer in a fun and interactive way. Each level will represent a new idea to teach to the user. It is our goal that after the user completes this game they have the logical understanding of programming so they can better understand and use concepts they use in class.

The hope for our game project *Delta* is that it will captivate the user and make the task of learning programming fun. In order to accomplish this goal we will implement a system where a user can save their data and begin playing at that level whenever they find the time. This will require expert programming skills on top of the skills required to make the game itself. In order to entice the user to play in the first place we need to have an appetizing graphical user interface (GUI). The importance of a simple but strong GUI will help the user navigate with ease, without the need for extra instruction. When they begin the game, the GUI will be extremely important because it needs to run smooth and needs to be clear as to where the user can take their path. The hope is for this game to advance from a simple web-based game into a phone application with multi platform support and connectivity so users from around the globe on different devices can interact.

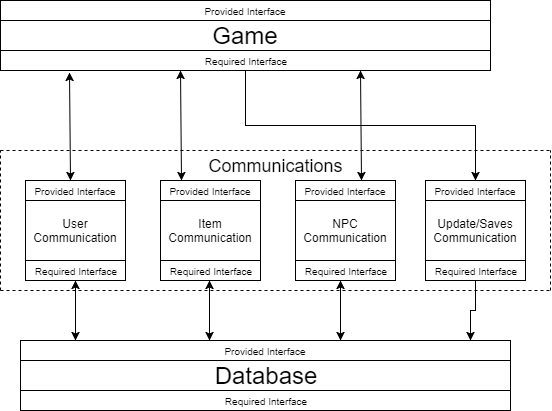
*Delta* will use third party software (Unity) and object-oriented programming. Doing this will allow us to create objects that can be loosely coupled to other classes and interfaces. The importance of having loosely coupled classes is extremely important because it allows our programming to be more efficient and reusable. We will have a single database responsible for containing the registered users and their characters as well as all the game items and NPC’s.

*Delta* will contain a three story architecture. Their will be the Game component where the users interact, a communications level, this serves as the bridge between the game and database. The communications level is the most vital component because this is where all the action happens to allow the game to run. The third level is the database, the database will contain all the game data and be stored in a local file.

The remainder of this document will provide more detail in how a our project will operate. Following the introduction we will bring you the Game Architecture, Class Diagram, Application classes, Database Diagram. Finally the document will conclude with the Integrated Test Plan to experiment the strength of our system.

**Chapter 2: Architecture**

Figure 2.1 Shows a high-level architectural design of the system.



*Figure 2.1: High-Level Architectural Design of Delta*

As the reader can see the game is separated into three sections, Database, the Communications area and the Game. These three separate layers all serve a vital but different role in the processes of our project. The most crucial part being the communications layer. This layer will be the most challenging layer to implement due to its complex handles that will have to be programmed. We also have to make sure it is not coupled to well will anything because we need it to be reusable; especially considering the time constraint on the project.

The first layer: Game, is what the user will interact with. Here the user will see the final product that the backend programming will develop. This is where the user can create his account, his character, and interact with items, other players and NPC’s. The communications layer is broken into sections. This allows us to search the database quicker, and keeps our data organized while the program is running. User Communication is designed to access the section of the database with a user's account and that accounts characters with their respective information. Item Communication is designed to access the section of the database that contains items, and their information. NPC Communication communicates with the database to update NPC interactions so the game can save what the NPC can see what it has already done with the character. Update/Saves Communication is when the User is in the game section and decides to save his game or update options in the game. It runs from the game, through the communication and saves the updated information to the database. The database section contains all the games data. The database is organized in such a way that it is simple to navigate and will be able to be searched effectively.

The importance of our High Level Architecture design is critical. Allowing the user to have direct access to the database can result in cheating, and cheating a game that is meant to teach the user something defeats the purpose of the game. The communication layer should only be holding data that the user requested in the game session. Therefore, the security of the data is practically meaningless at this layer. The game layer is designed for user interaction therefore it is the first and only layer the user will have direct access to.

**Chapter 3: Module Design**

*Section A: Game*

**Purpose**

The purpose of this module is for the user to interact with the Game, and the Game’s UI. The user interacts using their mouse and keyboard. Future development may include touch screen interaction.

**Provided Interface**

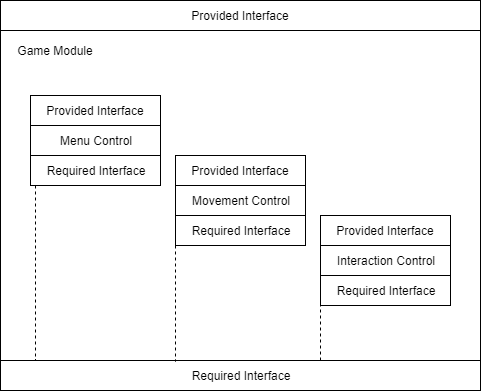
This module has no provided interface.

**Required Interface**

The required interface for the game module is the union of all sub-modules.

**High Level Module Design**

The game module is broken into movement controls, interaction controls, and menu controls, as shown in figure 3.1.



*Figure 3.1 High Level Module Design for Game Module*

**Menu Control Module**

**Purpose**

This module grants the user the ability to point, click and interact with the menu system. From here the user can login to their personal account, select the menu options and more.

**Provided Interface**

There is no provided interface.

**Required Interface**

void nextLevel()

void previousLevel

void quitRequest

void returnToStart()

void LoadLevel(string SceneName)

**Movement Control Module**

**Purpose**

This module is designed to allow the user to control their player. This includes, movement, and item usage depending on item.

**Provided Interface**

No Provided Interface

**Required Interface**

void moveNorth()

void moveEast()

void moveSouth()

void moveWest()

void armMove()

**Interaction Control Module**

**Purpose**

Allow user to interact with NPC’s, or interactions with some items depending on the usage of the item.

**Provided Interface**

No Provided Interface

**Required Interface**

void Interact()

void Talk()

void EndTalk()

void PickUp()

void Drop()

*Section B: Communications*

**Purpose**

The communications modules purpose is to provide the back end programming of the game and allow the connection of the database and its contents to the game module.

**Provided Interface**

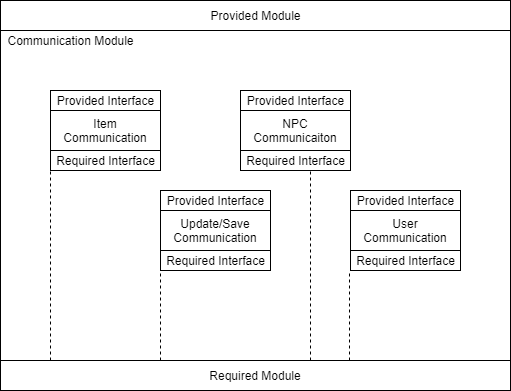
This module has no provided interface.

**Required Interface**

The required interface for the game module is the union of all sub-modules.

**High Level Module Design**

The game has three communication modules: User, Item, NPC, and Update/Save communications, as shown in figure 3.2.



*Figure 3.2 High Level Module Design of Communications Modul**e*

**Item Communication Module**

**Purpose**

This module is designed to find items saved in the database and send them over to the game world whether that be to a character, NPC, or the world itself.

**Provided Interface**

No Provided Interface

**Required Interface**

void GetIItem()

void DeleteItem()

void DeleteItemTimer()

void DeleteFromDatabase(string password)

**NPC Communication Module**

**Purpose**

This module is designed to respawn/spawn NPC’s that are not preloaded into the game. It will transfer data like character skins, or information about the NPC.

**Provided Interface**

No Provided Interface

**Required Interface**

void Spawn()

void SpawnTimer()

void GatherStrInfo()

void GatherIntInfo()

void DeleteFromDatabase(string password)

**Update/Save Communication Module**

**Purpose**

This module is designed to load a saved game from the users database, or save the user’s game or any updated information to the database.

**Provided Interface**

No Provided Interface

**Required Interface**

void DeleteSave()

void LoadGame()

void SaveGame()

void UpdateInfo()

**User Communication Module**

**Purpose**

This module is designed for the user to be able to log in to their 1) account and then 2) any characters associated with that account.

**Provided Interface**

No Provided Interface

**Required Interface**

void AccLogin()

void CharLoad(string name)

void createChar()

*Section C: Database*

**Purpose**

The database hold all the game information pertaining to items, characters, users, NPC’s. It hold almost all data except for a few level files.

**Provided Interface**

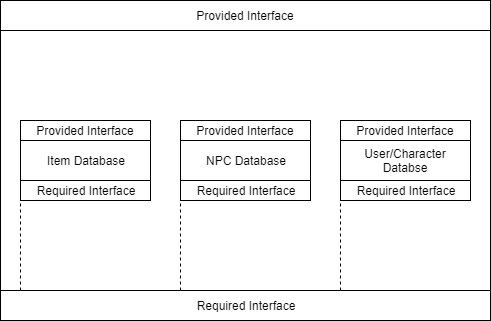
This module has no provided interface.

**Required Interface**

The required interface for the game module is the union of all sub-modules.

**High Level Module Design**

The database module is separated into three separate modules, Items, NPC, and User/Character Database. This can be seen in figure 3.3.



*Figure 3.3 High Level Design for Database Module*

**Item Database Module**

**Purpose**

This section of the database contains all item data.

**Provided Interface**

No Provided Interface

**Required Interface**

void LocateItem()

**NPC Database Module**

**Purpose**

This section of the database contains all NPC data.

**Provided Interface**

No Provided Interface

**Required Interface**

void LocateNPC()

**Item Database Module**

**Purpose**

This section of the database contains all user and character data.

**Provided Interface**

No Provided Interface

**Required Interface**

void LocateUser()

void LocateCharacter()

void LocateCharacterInUser()

**Chapter 4: Abstract Data Types**

|  |
| --- |
| CharacterADT |
| +health:int  +name:String  +item:Item[]  +xPos:Integer  +yPos:integer  +zPos:integer |
|  |

The **CharacterADT** contains five attributes that will define a character in the game. The first is the health that the character has remaining. The next is the character name and the array of items that the character can hold. Finally, the xPos, yPos and zPos state where the character is on the map at any given time.

|  |
| --- |
| ItemADT |
| +x-Coordinate:Integer  +y-Coordinate: Integer  +z-Coordinate:Integer  +itemID:Integer |
|  |

The **ItemADT** contains three integer values. The itemID tells the game which item is available in the level. The x, y and z coordinates show where the item is on the level map.

|  |
| --- |
| LevelADT |
| +LevelID:Integer  +Difficulty:Integer  +NPCList:NPC[]  +itemList:Item[]  +score:Integer |
|  |

The **LevelADT** contains five attributes that levels contain. First, each level contains a unique ID that differentiates each level from other levels. Another Integer attribute is difficulty. The first few levels will be simple, but as the game progresses they will get increasingly harder, as the programming concepts become more difficult.The NPCList and itemList are lists of the non-playable characters and items, respectively, that can be found in the level.

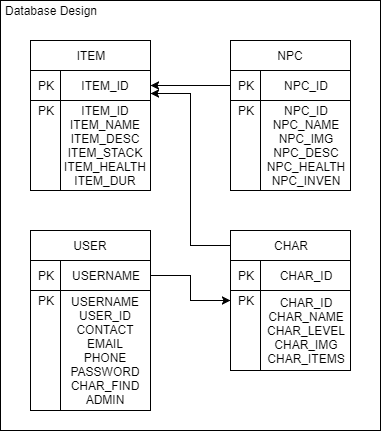
|  |
| --- |
| NPCADT |
| +x-Coordinate:Integer  +y-Coordinate: Integer  +z-Coordinate: Integer  +name:String  +item:itemID |
|  |

The **NPCADT** contains four attributes. The x, y and z Coordinates tell the player where the non-playable character is on the map, so that the player can interact with them. There is also a name attribute that shows the NPC’s name and it also shows if the character has any items that can be taken from them.

|  |
| --- |
| DirectionADT |
| +oldX:Integer  +oldY:Integer  +oldZ: Integer  +newX:Integer  +newY:Integer  +newZ:Integer |
| +newLoc(oldX, newX, oldY, newY, oldZ, newZ):Location |

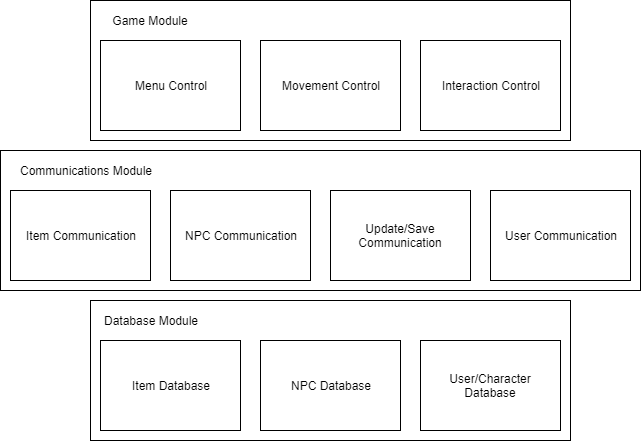
The final ADT is the DirectionADT. This ADT contains four different attributes and a single method. This contains four integer values: two x, y and z coordinates. The sole method takes these four values and determines the new location of the character on the map.

**Chapter 5: Database Design**

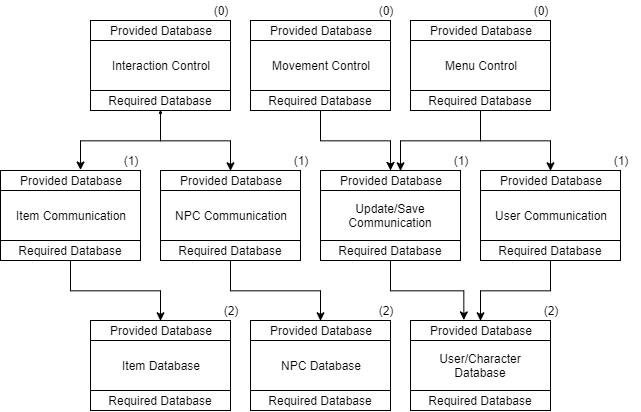


*Figure 5.1 Database Design*

**Chapter 6: COMPRISES Diagram**

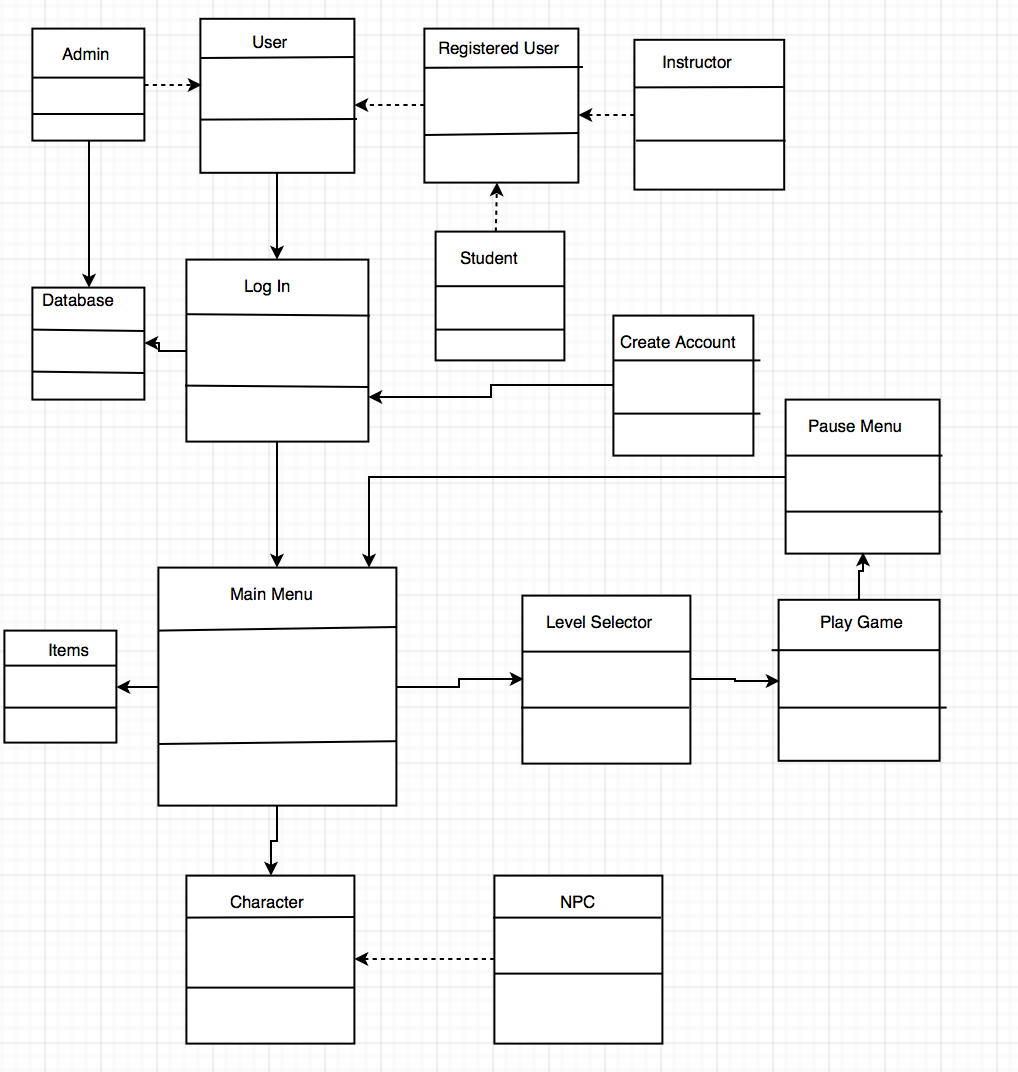


**Chapter 7: USES Diagram**



**Chapter 8: Other Diagrams**

Class Diagram (Class Attributes/Methods below)



|  |
| --- |
| User |
| +firstName:String  +lastName:String  +userName:String  +password:String  +emailAddress:String |
| +logIn() |

The **User** class contains all of the personal information for each of the game users. This includes their first and last names, along with their username, password and email address. The only method the user has is to call upon the logIn class.

|  |
| --- |
| RegisteredUser |
| +schoolName:String |
|  |

The **RegisteredUser** class is an extension of the User class. This will be an extension of the **User** class. It has one additional attribute, schoolName, which identifies the school that the player attends.

|  |
| --- |
| Instructor |
| +teacherID:Integer |
|  |

The **Instructor** class is an extension of the **RegisteredUser** class. It has one additional attribute, teacherID. This is an ID that we will give teachers who would like to use our software in their classes.

|  |
| --- |
| Student |
| +studentID:Integer |
|  |

The **Student** class is an extension of the **RegisteredUser** class. It has one additional attribute, studentID. This is an ID that we will give students after their teacher has signed up to use the game. It is a way that we will group the student users under their instructor, so it will be organized.

|  |
| --- |
| Admin |
| +adminID:Integer  +adminPassword:Integer |
| +editDatabase()  +viewDatabase() |

The **Admin** class is an extension of the **User** class. This class has two additional attributes and two additional methods. The adminID is a unique number given to certain employees, who manage the database for the game. This along with the adminPassword are the login credentials for them. These administrators will be able to view the users’ information in the database and edit the info is necessary.

|  |
| --- |
| LogIn |
| +userName:String  +password:String |
| +createAccount()  +verifyCredentials(userName, password):boolean |

The **LogIn** class allows the **User** to enter their userName and password in order to continue their progress in the game. There are two methods in this class. CreateAccount allows the new users to make a new account. If the player already has an account, then they will be able to enter their credentials, which calls on the verifyCredentials method. This compares their information with the database, and returns a boolean result of whether or not it matches. It the boolean returns “true,” then the MainMenu class loads

|  |
| --- |
| CreateAccount |
| +firstName:String  +lastName:String  +userName:String  +password:String  +emailAddress:String  +schoolID:Integer  +teacherID:Integer  +schoolName;String  +adminPassword:String  +AdminID:Integer |
| +createNewUser(firstName, lastName, userName, password, emailAddress)  +createNewStudentUser(firstName, lastName, userName, password, emailAddress, schoolName, schoolID)  +createNewTeacherUser(firstName, lastName, userName, password, emailAddress, schoolName, teacherID)  +createNewAdminUser(firstName, lastName, userName, adminPassword, emailAddress, adminID) |

The **CreateAccount** class has the **User** enter all of their information. Different users will need to enter different kinds of information. For example, **Admin** users will need to enter their AdminID and AdminPassword, while **RegisteredUsers** will need to enter their schoolName and their Student/Teacher IDs. There are 5 methods for each different kind of user.

|  |
| --- |
| Database |
| Item Database |
| NPC Database |
| User/Char Database |

The **Database** information was explained earlier in the document. You can see the database design in figure 5.1. The database is split into three separate sections. The first static section is the Item Section. This contains the items put into the game and their respective information as seen in figure 5.1. The seconds section is the NPC Database. This section contains all the information for the NPC’s, this section is also static. The third section is the only dynamic section, the only one that change change. This is the User data. Any number of users can be on any machine with as many characters as they wish. Users will be able to delete and create new characters.

|  |
| --- |
| MainMenu |
|  |
| +selectItem()  +selectCharacter()  +selectLevel()  +saveGame()  +updateInfo()  +logOut() |

The **MainMenu** class does not have any attributes, but has several methods. From the menu, there is an **Item** option, that will call on the selectItem method and bring the user to the Item screen. The same if for the **Character** option. The player will also be able to save his or her progress, update their **User** information, and logout. Finally, the Choose Level option, will call on the selectLevel method and bring the user to the Level Selection screen.

|  |
| --- |
| Character |
| +character:Character |
| +selectCharacter():Character |

The **Character** class will give the user an option of several characters to play as. More characters will be unlocked as the player progresses through the game. There is only one method in this class, which allows the user to select their character. It has no parameters and returns a Character

|  |
| --- |
| Item |
| +item:Item |
| +selectItem():Item |

The **Item** class works the same way as the Character class. It is a menu that allows the User to select items to use when they play the game next. The player will unlock certain items as they play the game. There is one method in the class, which allows the user to select their item. It takes no parameters and returns an Item

|  |
| --- |
| LevelSelector |
|  |
| +selectLevel(Integer):Level |

As stated above the , the LevelSelector class is accessed through an option in the MainMenu class. There are no attributes and one method. The method allows the user to select the desired level to play. The method requires one Integer value and returns the PlayGame class, with the desired level.

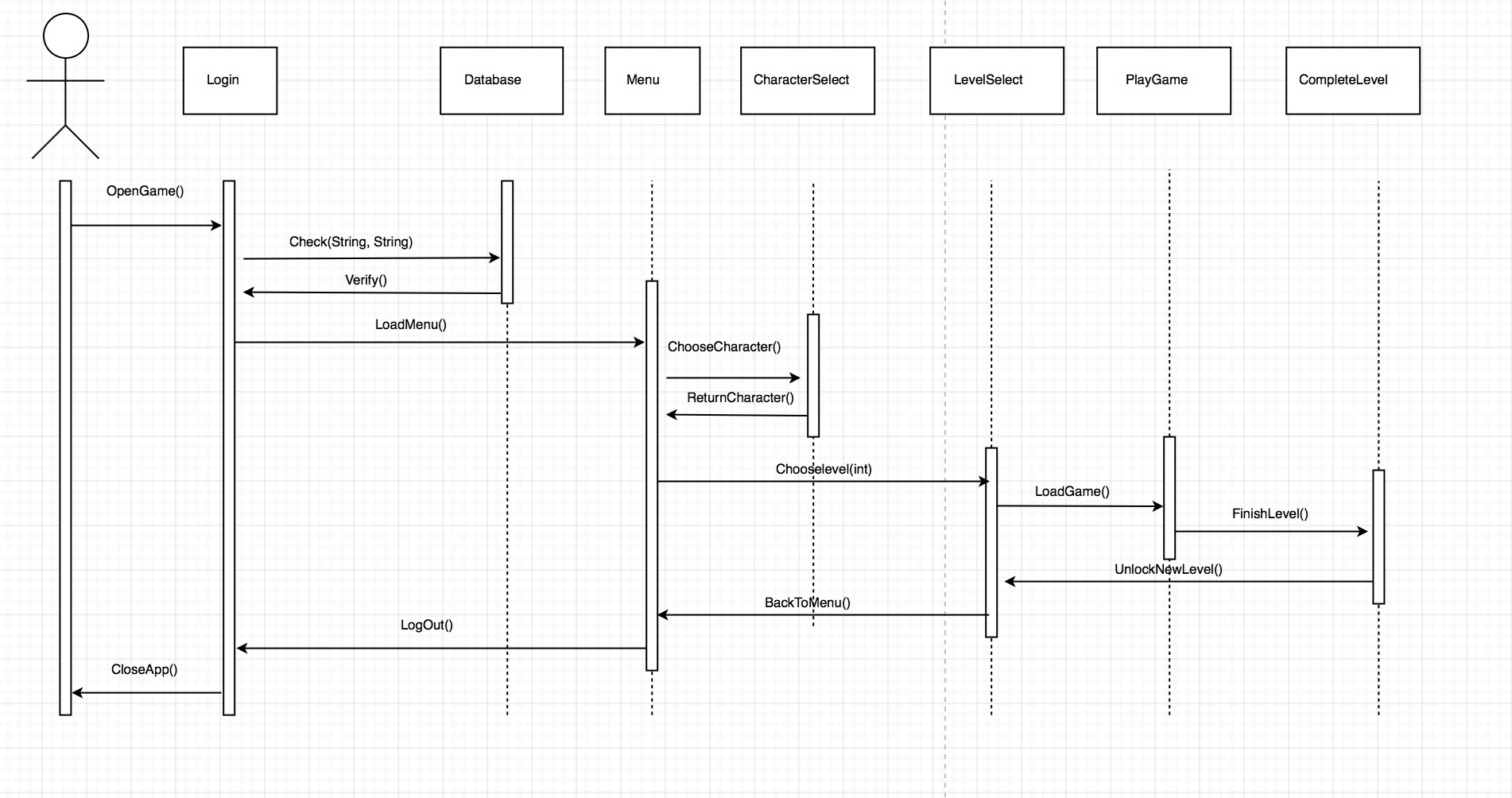
|  |
| --- |
| PlayGame |
| +Location |
| +Move(Direction):LocationChange  +PickupItem()  +Interact()  +Pause() |

The **PlayGame** class has one attribute and four methods. The only attribute is the player location, which tells the game where the player is on the map. The first method is move. This will take directional input from the user and return the change in location for the player on the map. The second method is the PickupItem(), where the user picks up an item from the map and adds it to their inventory. The third method is Interact, which is where the character interacts with an activity on the map. These will be the activities that will teach the player programming concepts. The final method is the pause method. This will allow the user to take a break from playing and pause their progress through the individual level.

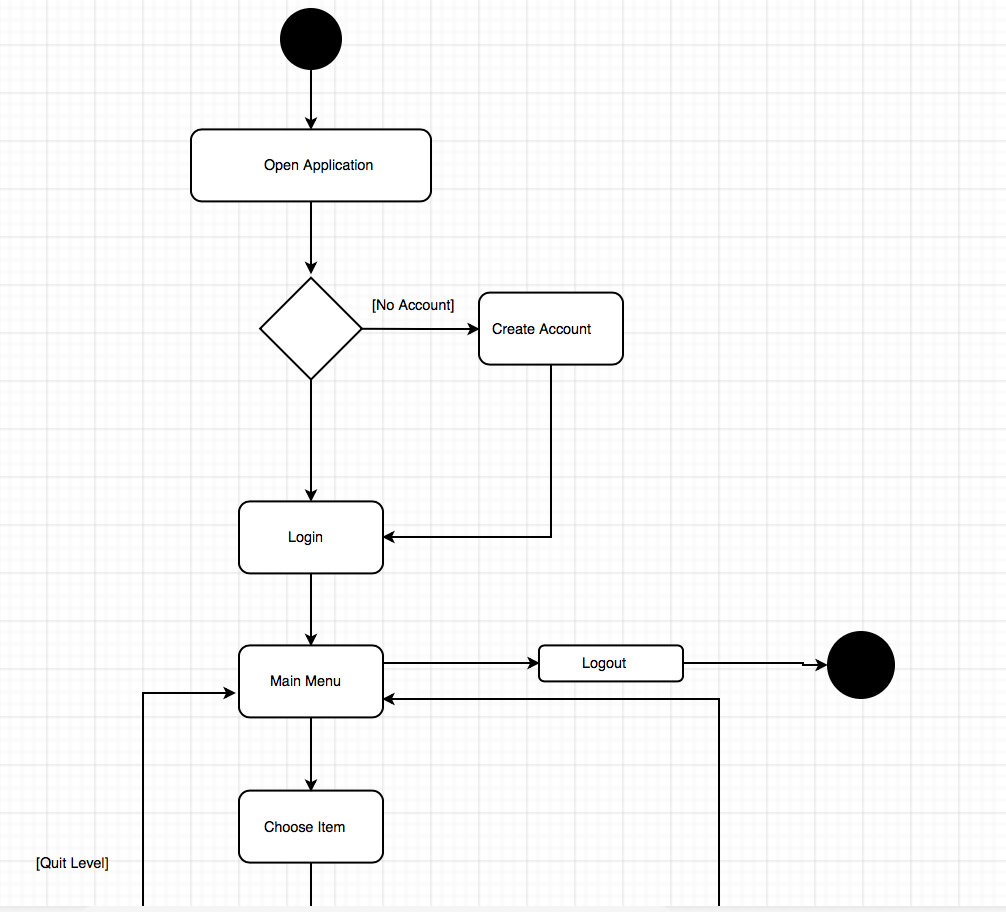
|  |
| --- |
| PauseMenu |
|  |
| +chooseNewItem():Item  +quitGame():MainMenu  +continueGame:PlayGame |

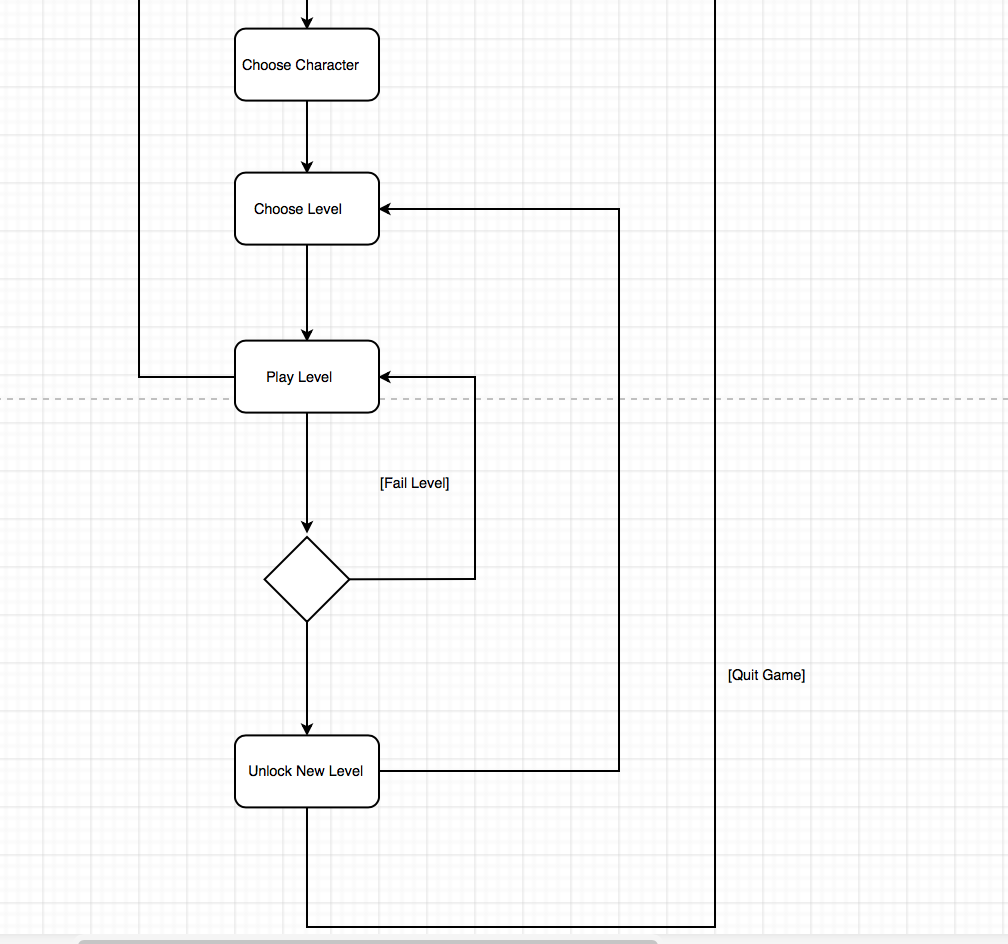
The **PauseMenu** method has no attributes and three methods. The first method allows the character to choose a new item to use in the level. The next method, quitGame, allows the user to quit the game and returns the MainMenu class. Finally, the continueGame method allows the user to continue the level that they are currently playing.

Sequence Diagram

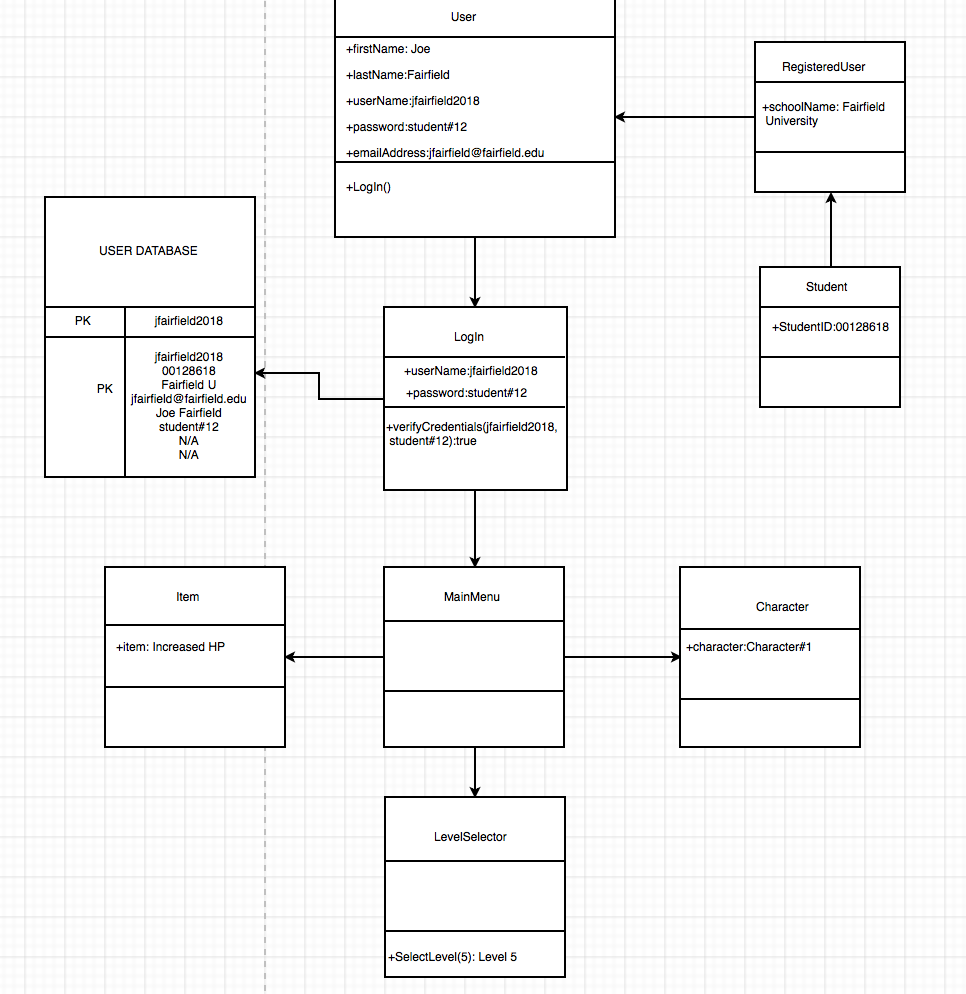


Activity Diagram





Object Diagram



**Chapter 9: Integration Test Plan**

**Approach**

For the integration and testing of *Delta* we are implementing a bottom-up approach. The bottom-up approach allows us to use drivers to test the low level components. When we do the final test, we know that every other sub component will successfully run.

**Order of Integration**

In the table below are listed the various phases that will be tested to ensure proper integration. We will begin with the database module, move to the communication module, and finish with the game module.

|  |  |  |
| --- | --- | --- |
| **Integration Group** | **Driver(s) Needed** | **Stub(s) Needed** |
|  |  |  |
| **Phase 1: Database Module** | | |
| N/A | N/A | N/A |
| **Phase 2: Communication Module** | | |
| Admin Module | User Driver | None |
| Registered User Module | User Driver | None |
| Student Module | User Driver | None |
| Instructor Module | User Driver | None |
|  |  |  |
| **Phase 3: Game Module** | | |
| Character Module | Level Driver | None |
| Level Module | Level Driver | None |
| User Module | User Driver | None |
| MainMenu Module | MainMenu Driver | None |
| Item Module | Level Driver | None |
| NPC Module | Level Driver | None |

**Driver Definitions**

User Driver: simulates the creation of the user accounts

Level Driver: simulates the level environment

MainMenu Driver: simulates a menu driver

**Tests Cases**

**Phase 1: Database Module**

**Phase 2: Communications Module**

**ID:** Admin

**What is being tested:** The admins ability to access and modify the database.

**Input Data:** Various attributes of a user account.

**Expected Output:** Changes are successfully made to the database.

**Environmental Constraints:** There is a connection to the database.

**Testing Constraints:** If adding, the user with that username must not exist already. If editing, the user must already exist.

**ID:** RegisteredUser

**What is being tested:** Create or edit a registered user account.

**Input Data:** Various profile information.

**Expected Output:** A created or update user account.

**Environmental Constraints:** There is a connection to the database.

**Testing Constraints:** If adding, the user with that username must not exist already. If editing, the user must already exist.

**ID:** Student

**What is being tested:** Create or edit a student user account and the ability to enroll in the class.

**Input Data:** Various profile information.

**Expected Output:** A created or updated student account.

**Environmental Constraints:** There is a connection to the database.

**Testing Constraints:** If adding, the user with that username must not exist already. If editing, the user must already exist.

**ID:** Instructor

**What is being tested:** The ability to create or edit an instructor account and to create a class. Also the ability to generate a report of a class.

**Input Data:** Various profile information.

**Expected Output:** A created or updated instructor account.

**Environmental Constraints:** There is a connection to the database.

**Testing Constraints:** If adding, the user with that username must exist already. If editing, the user must already exist. If generating a report, the class must exist already.

**Phase 3: Game Module**

**ID:** Character

**What is being tested:** The ability to create and modify the state of character.

**Input Data:** The name and ID of a new character.

**Expected Output:** A new character is made and can be modified.

**Environmental Constraints:** N/A.

**Testing Constraints:** N/A.

**ID:** Level

**What is being tested:** The creation for a level and that it can be modified.

**Input Data:** Various information regarding level characteristics.

**Expected Output:** The instantiation of a level.

**Environmental Constraints:** N/A.

**Testing Constraints:** N/A.

**ID:** User

**What is being tested:** The user is able to interact with the game.

**Input Data:** Input from the device.

**Expected Output:** Provided input is successfully executed.

**Environmental Constraints:** N/A.

**Testing Constraints:** N/A.

**ID:** MainMenu

**What is being tested:** The creation and successful interaction with a menu

**Input Data:** Input from the device.

**Expected Output:** Correct actions depending on which features for the menu are used.

**Environmental Constraints:** N/A.

**Testing Constraints:** N/A.

**ID:** Item

**What is being tested:** The item is created, can be used, and modified within a level.

**Input Data:** Information that identifies and modifies the item.

**Expected Output:** A created, usable, and modifiable item.

**Environmental Constraints:** You are within a level.

**Testing Constraints:** N/A.

**ID:** NPC

**What is being tested:** The creation of an intractable NPC.

**Input Data:** Information needed to create and identify a NPC.

**Expected Output:** A NPC that the user can interact with.

**Environmental Constraints:** You are within a level.

**Testing Constraints:** N/A.