

Monsters and Metal: Complete Documentation

Ian Bunner¹ Connor Valore-Kemmerer² Nathan Shangkala³ Vincent Liang⁴

Baiyu Zhu⁵ Christina Guan⁶

November 25, 2018

¹bunner@usc.edu

²valoreke@usc.edu

³shangkal@usc.edu

⁴liangvin@usc.edu

⁵baiyuzhu@usc.edu

⁶christcg@usc.edu

Contents

1	Proposal	2
1.1	Two Hour Weekly Meeting Time:	2
1.2	Meeting Location:	2
1.3	Brief Overview of the Project:	2
1.4	How we will use a GUI:	2
1.5	How we will use multi-threaded code:	2
1.6	How we will use networking:	2
2	High-Level Requirements	2
3	Technical Specifications	6
3.1	GUI Design	6
3.2	Game Logic	6
3.3	Database	7
3.4	Backend	7
4	Detailed Design	7
4.1	Difficult Algorithms	7
4.2	Graphical User Interface	7
4.3	Database	10
4.4	Hardware/Software Requirements	11
4.5	Classes/Hierarchies	11
4.6	Implementation	12
5	Testing	12
5.1	Black Box Testing	12
5.2	GUI Testing	12
5.3	Marketplace Testing	13
5.4	Database Testing	13
6	Deployment	14

1 Proposal

1.1 Two Hour Weekly Meeting Time:

Fridays from 4:00 pm to 6:00 pm.

1.2 Meeting Location:

Reserved room in Leavey Library (LVL) or group study room in Fertitta Hall (JFF).

1.3 Brief Overview of the Project:

The goal of our project is to build a 2D, turn-based game using Unity. Generally, it will be similar to monster collecting games like Pokémon. In the game, users can explore randomly generated dungeons and capture various monsters. Different maps can have their own types of monsters. Additionally, there can be a final boss, which drops rare items, to defeat. There are markets that allow users to trade for different monsters or items like EXP double tickets and treasure boxes. There are rewards for battles. Users will be guests before they register. Guest users can only access one map and limited monster types. Also, there are items that can only be collected by registered users. Overall, we will complete the core structure of the game, then expand on it and add more features.

1.4 How we will use a GUI:

Users can interact with the game through a GUI. Some art and aesthetics act as a GUI as well.

1.5 How we will use multi-threaded code:

Parallelize lists where an object's update is not dependent on other objects in the list (ie. particles, or entities that don't need other entities to update) with different threads updating different objects in those lists. Also, resource Allocation (loading/calculating textures, meshes) can be done by multi-threading.

1.6 How we will use networking:

Items or monsters can be uploaded to a marketplace where users can request trading their monster for another type of monster they need. This function can be done by using networking.

2 High-Level Requirements

Definition 1.1: The game should have a 2-Dimensional display for the user to interact with. This display should provide the user access to all of the functionality they need.

Requirements:

1.1.1- The display must allow the user to move with their arrow keys or wasd.

1.1.2- The display must have a button users can click to save their game state or return to a previous game state depending on how saving is done. See definition 3.

1.1.3- The display must have a button users can click to exit the game.

1.1.4- The display must have a way for users to visualize their inventory. This can be a popup from clicking a button or a separate window within the main window. See definition 4.

1.1.5- Items in a user's inventory must be clickable so that users can read a description of the item and any special properties the item confers. See definition 4.

1.1.6- A button must be provided so that users can access a map of the high level layout of their virtual world. This can be a popup from clicking a button or a separate window with the main window. See definition 5.

The following will likely not be able to be implemented in an 8-week project.

1.1.4- The game must enable configurable hotkeys.

1.1.5- The game should automatically map hotkeys for all graphical buttons, and these hotkeys should be displayed when hovering over a button.

Definition 1.2: The game must use consistent keyboard commands to respond to in game events.

Requirements:

1.2.1- These commands should be provided in a document that users can consult. This document should be packaged with the game.

1.2.2- These commands should be included in the text generated by the first event that requires their specific response.

The following will likely not be able to be implemented in an 8-week project.

1.2.3- This document should include default hotkey mappings and instructions on how to change them.

Definition 1.3: The game must have the functionality to save a user's game state. This can optionally save their exact status, or their status at well-defined checkpoints. If checkpoints are used, users should not have to wait for the save process to finish before continuing and the save button should allow users to see their most recent checkpoint and return to that state if desired.

Requirements:

1.3.11- A user should be returned to their exact location with their exact state from their most recent save upon opening the game. (if checkpoints are not used)

1.3.12- Users should be prompted to save before exiting. (if checkpoints are not used)

1.3.21- Users should only be able to save their state at a checkpoint the first time they reach it. (if checkpoints are used)

1.3.22- Users should get a message when they reach a checkpoint. (if checkpoints are used)

Definition 1.4: The user's inventory must be modifiable by the user from its graphical interface and limited in size.

Requirements:

1.4.1- The inventory must have fixed size so that the user cannot simply keep every item they ever come across.

1.4.2- Users should be able to remove items from their inventory at all times through the inventory's graphical display.

1.4.3- Users should be prompted to go straight to their inventory display every time they come across an item they decide to keep that puts their inventory past max capacity.

Definition 1.5: A map of the high level layout of the virtual world should be accessible by the user.

Requirements:

- 1.5.1- If checkpoints are used, the map should visually distinguish between checkpoints that have already been reached and those that haven't.
- 1.5.2- The map should clearly mark where the current user is located.
- 1.5.3- The map should intuitively show the main paths a user can take as well as any barriers that will prevent user movement GLOBALLY. Local barriers should not be visible.

Definition 1.6: Users should be able to access an online marketplace where they can trade items in their inventory with items from another user's inventory. NOTE: Due to the fact that this is only an 8-week project "online" will be implemented by storing a few fake user inventories in a database that our users can view and trade with.

Requirements:

- 1.6.1- Users should be able to place items in this marketplace to be traded.
- 1.6.2- When a user's item is in the marketplace it should still be visible in their inventory.
- 1.6.3- When a user's item is in the marketplace it should still count towards the inventory cap.
- 1.6.4- When a user's item is in the marketplace it should not be usable or discardable.
- 1.6.5- When a user accesses the marketplace they should be able to see objects available for trade.
- 1.6.6- A user should be able to sort and search objects in the marketplace.
- 1.6.7- A user should be able to see all offers made for items they have listed in the Marketplace.
- 1.6.8- Users should get a notification when their offer on an object has been accepted. This item should then be added to their inventory and removed from the trader's Inventory.
- 1.6.9- Users should be able to remove offers that have not yet been accepted.
- 1.6.10- Users should be able to remove items they no longer wish to trade.

The following will likely not be able to be implemented in an 8-week project.

- 1.6.11- Users should be notified if an outstanding offer of theirs is rejected, or an item they have an outstanding offer on is no longer available.
- 1.6.12- Users should be able to see the name of the account listing an object and message that account.
- 1.6.13- The marketplace should have a chat interface that is visible by all users currently "in" the marketplace.

Definition 1.7: When a user enters a battle, they should be able to perform all battle-actions through the GUI in a well-defined control sequence.

Requirements:

- 1.7.1- All parts of the main GUI should still be visible and accessible, though how they may be accessed can alter slightly to improve aesthetics.
- 1.7.2- The location of the save button and exit button may NOT be altered.

1.7.3- A user should be prompted with a limited number of actions they can perform during their turn.

Definition 1.8: Interactive items and characters automatically sends users into battle when encountered in their grid space.

Requirements:

1.8.1- The requirements for this are to be interpreted by the developer. It depends largely on other implementations that the developers have control over.

The following will likely not be able to be implemented in an 8-week project.

Definition 2.1: Users should have control over the appearance of their character.

Requirements:

2.1.1: Users should be able to pick from default characteristics when starting a new game.

2.1.2: Users should be able to access a GUI for changing their character's appearance from a button and by clicking on a wearable item in their inventory.

Definition 2.2: Users should be able to access a secondary storage mechanism.

Requirements:

2.2.1: This secondary storage mechanism should only be able to store items that would reasonably keep well in storage.

2.2.2: This secondary storage should only be accessible from a single fixed point in the virtual world, or a series of sparse and related locations with some sort of connection mechanism.

Definition 2.3: Environmental variables should impact a player's vitals and should change depending on location and a seasonal model.

Requirements:

2.3.1: The climate should be well defined and minimally include temperature related effects.

2.3.2: User's should be able to collect and equip wearables that mitigate environmental effects on their character's vitals.

2.3.3: The game should keep implement a seasonal model consistent with seasons in comparable real world biomes.

Definition 2.4: The wild dungeon monsters should fight optimally.

Requirements:

2.4.1: They must have some artificial intelligence directing their attacks.

2.4.2: This artificial intelligence should allow the users to play on varying difficulty levels.

3 Technical Specifications

3.1 GUI Design

- Main graphical user interface top bar implemented (2 hours)
 - Save Button
 - Pull up inventory button
 - Map button
 - Exit button
- Main character movement screen implemented (4 hours)
 - WASD binding
 - Separate thread for displaying this main screen
- Inventory GUI implemented (2 hours)
 - Separate thread for displaying inventory screen
 - Option to remove items from inventory
- Map GUI implemented (2 hours)
 - Show current location
- User sign-in frontend designed and implemented (1 hour)
 - Simple DB to store users and hashed passwords
 - Simple graphical interface to verify
- Fighting cutscene must be designed (1 hour)
 - Graphics to change main display designed

3.2 Game Logic

- Fight logic written (4 hours)
 - Battles occur randomly when the player is moving in a dungeon. (2 hours)
 - Navigating the option menu. (1 hour)
 - Enemy will randomly choose an option (30 min)
 - Damage is proportional to attack. Battle ends when someone faints. (30 min)
- Access to Dungeons and Marketplace should be restricted to guest users (30 min)
- Users must be taken to a fighting cutscene when they run into a dungeon monster (1 hour)
- When a monster is captured it should be added to the user's inventory unless the inventory is at capacity, in which case the user should be asked if they would like to delete an item to make room. (1 hour)

3.3 Database

- Database needs to be setup to store variables for save states (2 hours)
- JDBC file needs to be setup to connect Unity to the Database (2 hours)
- JDBC file needs to be setup to connect backend logic to the Database (2 hours)

3.4 Backend

- The user sign in system front-end needs to be connected to the backend for validation (2 hours)
- User loading once validated needs to be implemented (1 hour)
- Guest users must get an auto-created guest user account if not signed in (2 hours)
- Marketplace must be populated with tradeable items from fake accounts (5 hours)
- User accounts and the password must be stored (1 hour)
- Trading functions must be implemented for the marketplace.
 - Completing a trade must swap items between player inventories (2 hours)
 - Users should be notified when an offer is accepted/rejected (2 hours)
- Function of acquiring/collecting monsters needs to be implemented. (2 hours)
 - Capturing mechanism designed and implemented (up to dev)
- Every monster needs to have specific value for their HP, attack damage, etc.
 - Auto generated on dungeon load (1 hour)
 - Healing mechanism implemented for user monsters (2 hours)

4 Detailed Design

4.1 Difficult Algorithms

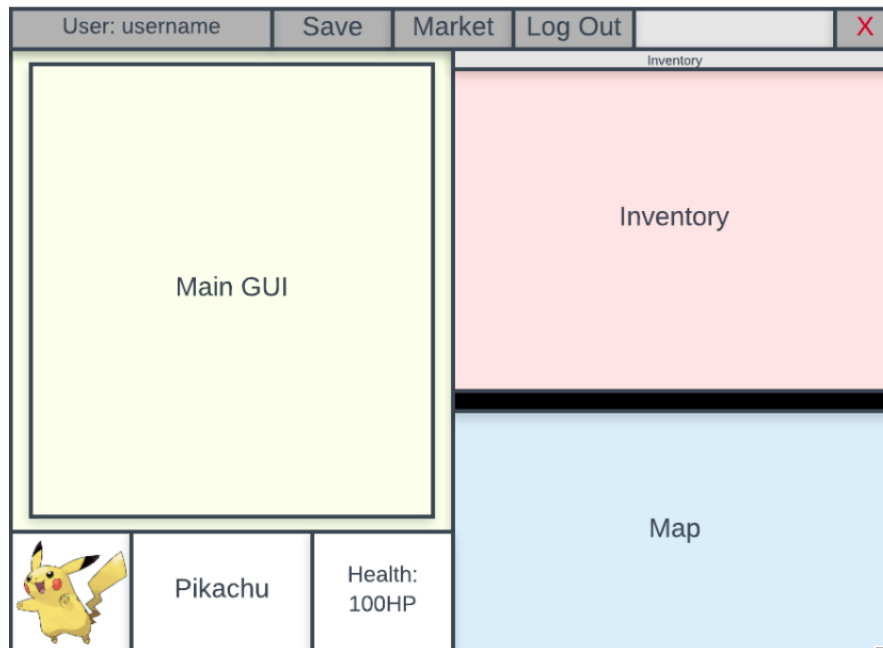
None, this project is pretty light on theory. The bulk of the work is discovering how to use the features in Unity and connect the Unity front end to a Java backend.

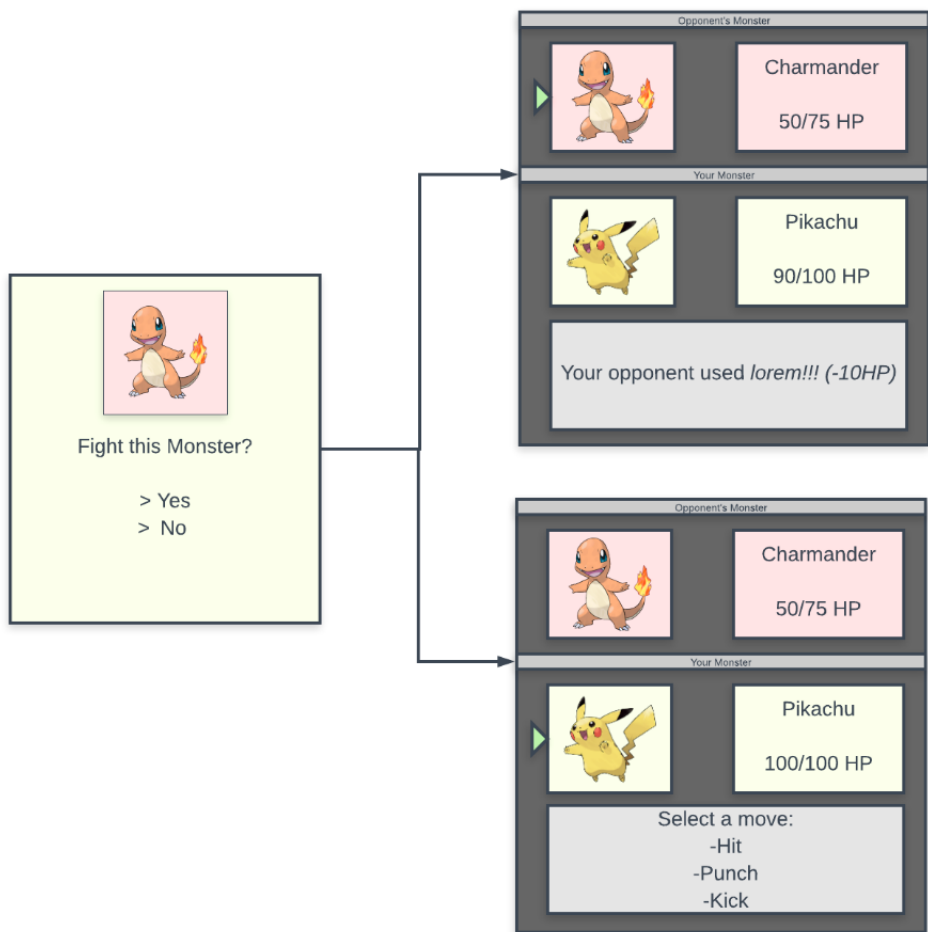
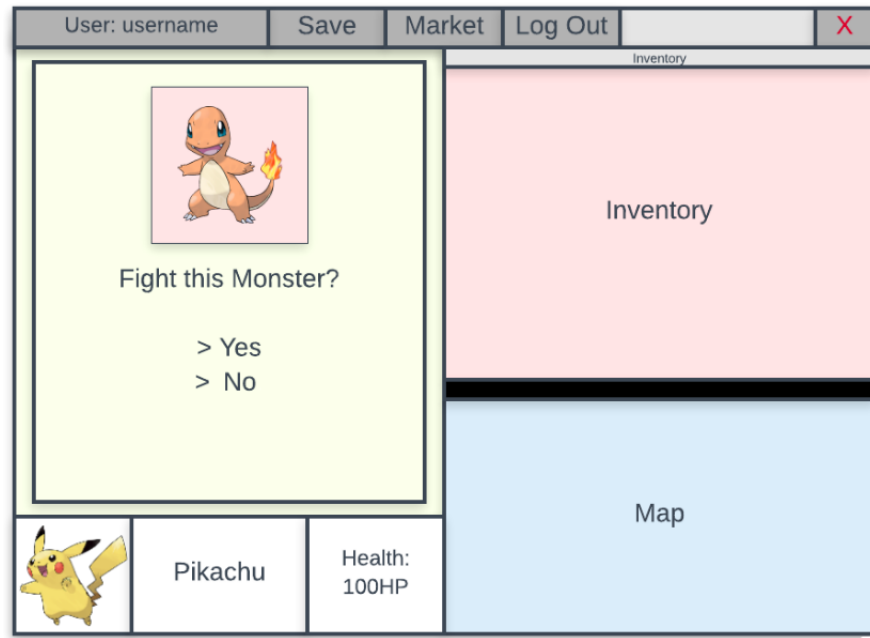
4.2 Graphical User Interface

The GUI isn't finished being built in Unity, so we designed a series of mockups that demonstrate the exact layout that the GUI will have. The coloring is important only to improve aesthetics and group elements that are the same. We will work around the free art available in the Unity store to determine an appropriate color theme for the GUI. The different aspects to our GUI are shown below.


1. This is the screen that a user will be presented with upon opening our game. They will be able to sign in and be redirected to (3) or to create an account (2).
2. This is the page that will allow new users to create an account. Once they submit the form, if all of the information is valid they will be redirected to (1). Otherwise, they will be returned back to this screen.

3. This is the main window that logged in users will interact with. The main GUI will show the player in the virtual world. The lens will be 2D and show the character from directly above. The top bar contains the user's username, a button to save their game state, a button to enter the marketplace, a button to logout, and a button to close the window. The inventory screen is a separate screen that shows the items that the user is carrying around. The form of this screen is shown in (6) and (7).
4. This shows how the player will be notified that they have encountered a monster. Selecting "No" will return them to (3). Selecting "Yes" will send them to (5).
5. This shows how the area occupied by the main GUI will alternate during a fight. When it is an opponents turn to attack, the top screen will be shown, and when it's a user's turn to attack the bottom screen will be shown.
6. This is the inventory screen that is shown in (3). Each item in the inventory will be displayed with it's image, name, type, and description. Clicking on an item will create the popup shown in (7)
7. A popup allowing users to manipulate the items in their inventory.





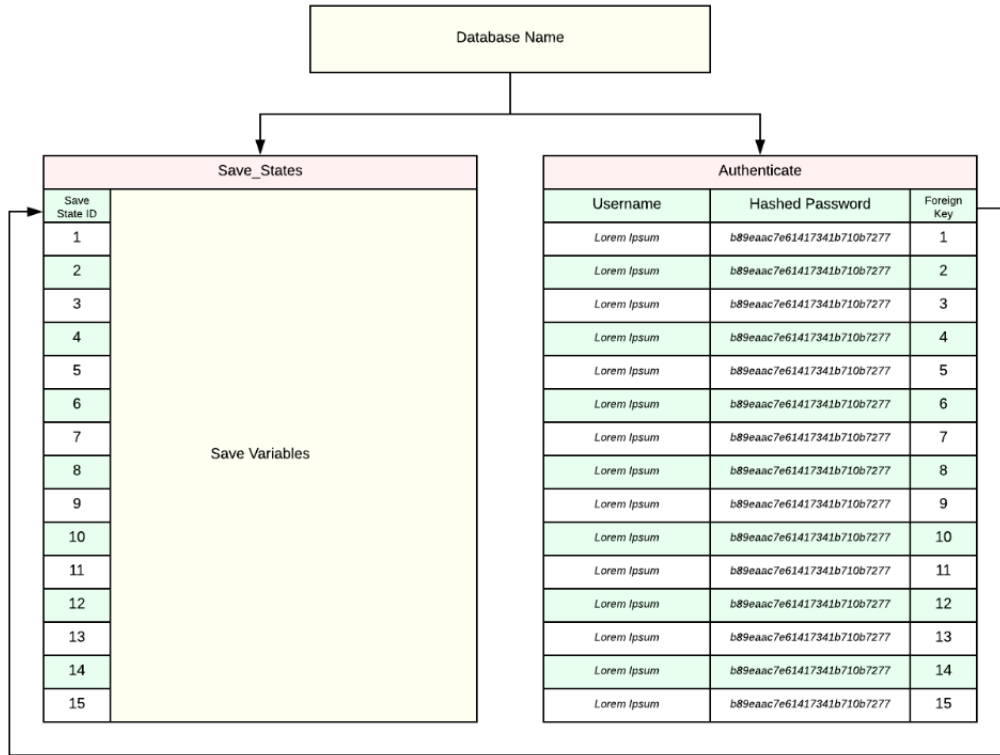
Inventory		
Image	Item Name	Notes/Type Dependent Stats
	Item Type	
Image	Item Name	Notes/Type Dependent Stats
	Item Type	
Image	Item Name	Notes/Type Dependent Stats
	Item Type	
Image	Item Name	Notes/Type Dependent Stats
	Item Type	
Image	Item Name	Notes/Type Dependent Stats
	Item Type	

Inventory (On Click of Item)		
Image	Item Name	
	Pikachu	
Image		Item Description
Image		
Image		
Image	Pikachu (Item Name) Item Status (on market, HP, etc...)	<div> <div>Add/Remove from Market</div> <div> Equip (if wearable) Sleep (if monster) Use (if modifier) </div> <div>Discard</div> </div>
Image	Item Type	Notes/Type Dependent Stats

4.3 Database

Due to the time and resource constraints we are dealing with, we will implement the user authentication and game save state features as tables in the same database. Ideally, user authentication would be handled by a separate central server so that users could access their account from any device with the game installed.

However, since the game prototype is only being developed with single machine usage in mind and we don't have easy access to a server to host our authentication database we will be spending our resources elsewhere. The passwords will be stored hashed, time permitting. The Save.States table is intentionally vague to allow flexibility in what is stored. This is entirely dependent on what features we have time to implement. We will be passing JSON objects to and from the front end to save and load user states.



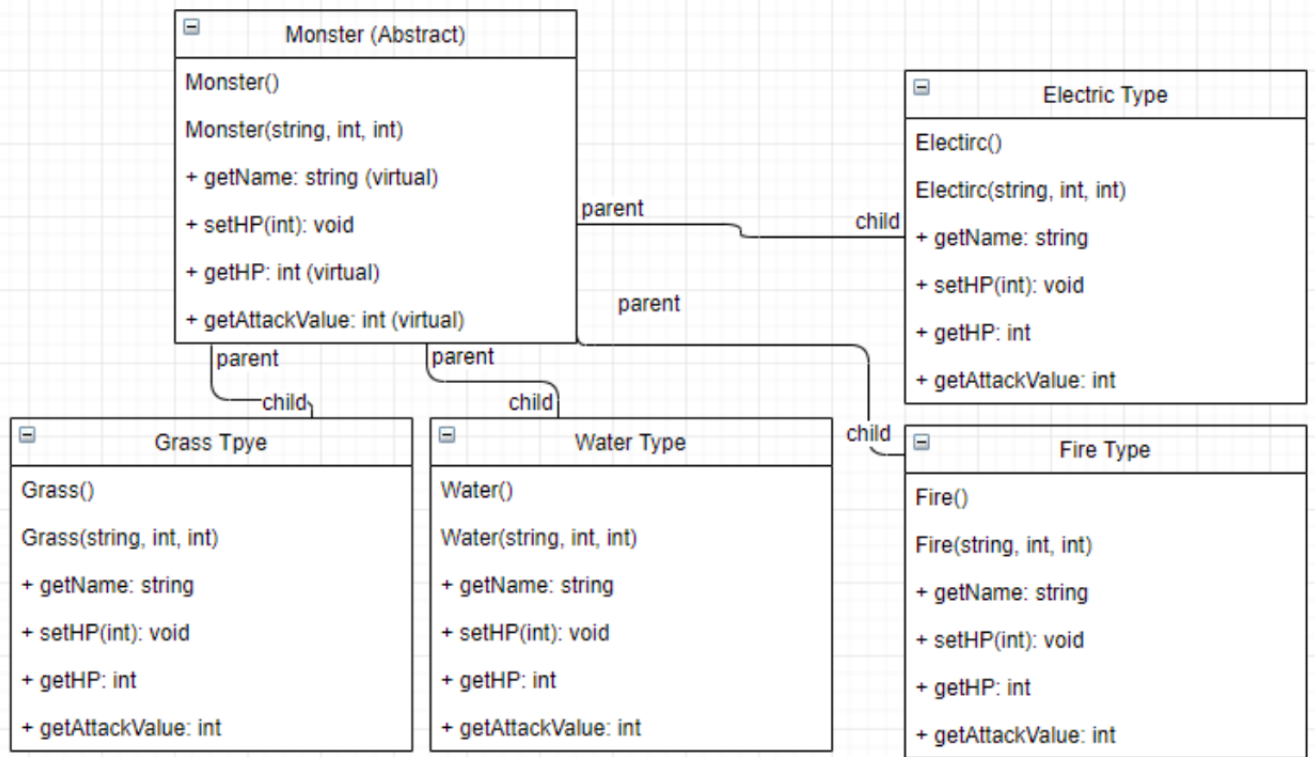
4.4 Hardware/Software Requirements

The game will be available for both Windows and Mac users. In order to access their account users will need to be logged in to a computer that has the game downloaded. For best performance, users should have at least a 2.0 GHz processor and 4 GB of RAM.

4.5 Classes/Hierarchies

As we likely won't have time to implement items, the monsters in our game represent the only family of classes we will need. We will have an abstract type monster that all other monsters inherit from. Each monster must implement its own attacks and vitals, but will share a common implementation for features that all monsters share such as getters and setters for universal variables.

Due to time constraints, we will limit our monsters to types. We will not further divide types into individual "species". The types will correspond to the element of the dungeon that they are found in. They will have fixed maximum HP and movesets that do not change throughout the course of their life, even through training. Sample class diagrams and an inheritance hierarchy are shown below.



4.6 Implementation

Our front end will be mostly written in Unity, with C scripts. This has the benefits of inheriting Unity's `MonoBehaviour` class that automatically employs `Start`, `Awake`, and `Update` methods and automatically loops through the game state of each frame. It also has the benefit of using Unity's `MonoBehaviour` packages that allows easy communication with the GUI by allowing visual serialization of fields. Unity also allows usage of prefabs, which will be how most of the level will be built. We will use `MonoBehavior`'s inherited `Instantiate` and `Destroy` methods to build grid type levels without needing to individually set each object. Player and monster information will be stored as an object that doesn't inherit `MonoBehavior`, while player and monster behavior will be controlled by Behavior scripts that inherit `MonoBehavior`. Overall game state will be tracked by a Game Manager object, which will also be responsible for tracking JSON requests and responses.

5 Testing

5.1 Black Box Testing

Recruit users to play for 5-10 minutes and provide feedback based on their experience.

5.2 GUI Testing

- Click on an item in inventory. This should bring up a menu with a list of actions and a description of the item.
- Click on the marketplace button. This should take you to the marketplace.
- In the save thread, add a print statement that prints to console when the game state is being saved. Click on the save button. Ensure that this causes one output to the console.
- The username in the GUI should be the username used to successfully log in.

- WASD bindings should move the character.
- Arrow key bindings should move the character.
- Users should be taken to a fight scene when encountering monsters in a dungeon.
- The fight scene should have the functionality enumerated in the detailed design document.
- Monster health should be updated appropriately based on the damage done by an opponent's attack (for both the user's monster and the wild monster).
- Monster health at the end of a battle should be the same as the monster's health when exiting the fight scene.

5.3 Marketplace Testing

- Log into an account with at least one monster in inventory. In the inventory, click on a monster. This should give you a menu with the option to add the monster to the marketplace.
- Add the monster to the marketplace.
- Go to the marketplace via the button in the top bar.
- Ensure that the monster is viewable as on the market.
- Go to a dungeon and walk around until you encounter a monster. If that monster was the only one in your inventory you should get a message indicating that you have no monsters in your inventory to fight with. Otherwise, you should be able to enter battle with any monster that is not the one you put on the market.
- Log out and into a different account with at least one inventory item.
- Go to the marketplace and offer an item for trade on the item just added by the other account.
- Ensure this item is not usable, as per the instructions above.
- Log out and then log into the first account. Accept the trade. Ensure that this user's inventory is properly updated.
- Log out and the log into the second account. Ensure this user's inventory has also been updated appropriately.

5.4 Database Testing

For these tests let ψ be the set of users already in the database. All results will assume that the returned set is some set $\psi \cup A$, where A is the set of elements that are added as a result of our tests and for all $a \in A$, $a \notin \psi$.

Add a New User

- Navigate to login screen, click on “create new account”, and enter “Joe_User” for username and “password” for password.

- Run the following java class:

```

1 public Class ADD_NEW_USER_TEST {
2     public static void main() {
3         try {
4             Connection conn = null;
5             PreparedStatement ps = null;
6             ResultSet rs = null;
7             String email = request.getParameter("userEmail");
8             Class.forName("com.mysql.jdbc.Driver");
9             conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/Info?user=root&password=BBB_Confirm3d&useSSL=false&useLegacyDatetimeCode=false&serverTimezone=UTC");
10            ps = conn.prepareStatement("SELECT COUNT(*) FROM Info.UserInfo WHERE (Username='Joe_User' and HashedPassword='password') AS count;");
11            rs = ps.executeQuery();
12            int count = rs.getInt("count");
13            if (count == 0) {
14                System.out.println("Test passed");
15            }
16            else {
17                System.out.println("Test failed. Expected result '0', but got result '" + count + "'");
18            }
19        } catch (SQLException se) {
20            System.out.println(se.getMessage());
21        } catch (ClassNotFoundException cfne) {
22            System.out.println(cfne.getMessage());
23        }
24    }
25 }

```

- Log out, then repeat the first bullet point, using a different password.
- Run the following java class:

```

1 public Class ADD_EXISTING_USER_TEST {
2     public static void main() {
3         try {
4             Connection conn = null;
5             PreparedStatement ps = null;
6             ResultSet rs = null;
7             String email = request.getParameter("userEmail");
8             Class.forName("com.mysql.jdbc.Driver");
9             conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/Info?user=root&password=BBB_Confirm3d&useSSL=false&useLegacyDatetimeCode=false&serverTimezone=UTC");
10            ps = conn.prepareStatement("SELECT COUNT(*) FROM Info.UserInfo WHERE (Username='Joe_User') AS count;");
11            rs = ps.executeQuery();
12            int count = rs.getInt("count");
13            if (count == 1) {
14                System.out.println("Test passed");
15            }
16            else {
17                System.out.println("Test failed. Expected result '1', but got result '" + count + "'");
18            }
19        } catch (SQLException se) {
20            System.out.println(se.getMessage());
21        } catch (ClassNotFoundException cfne) {
22            System.out.println(cfne.getMessage());
23        }
24    }
25 }

```

- Log out, and try to log in with the username “Joe.User” and password “password”. You should not be successful.
- Run the following class to verify that users initially receive a proper blank inventory. You should see the right username and a blank inventory array.

```

1 public Class LOAD_EXISTING_USER_INVENTORY {
2     public static void main() {
3         try {
4             Connection conn = null;
5             PreparedStatement ps = null;
6             ResultSet rs = null;
7             String email = request.getParameter("userEmail");
8             Class.forName("com.mysql.jdbc.Driver");
9             conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/Info?user=root&password=BBB_Confirm3d&useSSL=false&useLegacyDatetimeCode=false&serverTimezone=UTC");
10            ps = conn.prepareStatement("SELECT Inventory FROM Info.UserInfo WHERE (Username='Joe_User') AS inventory;");
11            rs = ps.executeQuery();
12            JSONObject inventory = new JsonParser().parse(rs.getString("inventory")).getAsJsonObject();
13            Gson gson = new GsonBuilder().setPrettyPrinting().create();
14            System.out.println(gson.toJson(inventory));
15        } catch (SQLException se) {
16            System.out.println(se.getMessage());
17        } catch (ClassNotFoundException cfne) {
18            System.out.println(cfne.getMessage());
19        }
20    }
21 }

```

Testing Inventory Update

Log in to “Joe.User”. Enter a dungeon and capture a monster. Save. Then run the LOAD_EXISTING_USER_INVENTORY class from above and ensure that the monster’s info is in the inventory array.

6 Deployment

To deploy our game, we would make use of the built in build features that the Unity game engine offers. For the purposes of demonstrating, we will build for 64-bit Windows machines. If we were to do a wide release we would also include Mac builds and potentially builds for popular Linux distros. For Windows, the build will give an .exe and a Data folder that, together, provide access to the full functionality of the

game. For Mac, the build process creates a standalone app that is self contained. For the first build we would have to worry about appropriately importing scenes and specifying their order of execution. Since scene transitions are mostly handled in game, this would not be too difficult. In all likelihood, the bulk of the work would be in creating a loading animation to play and then switching to login. For our build for demonstrating we will not have an introductory animation, and will instead just pull up the login screen on load. The nice thing about building through Unity is that it automatically handles the bundling of all dependencies. This is especially helpful if we were to package the game for a wide variety of platforms, as it would reduce the need for domain specific knowledge to provide our code to a wide user base.

In addition to game build concerns, we would have to migrate our database to some central server so that users are able to centrally authenticate and access their accounts from different computers. For such a small game, we would likely use something like Amazon Web Services instead of building our own infrastructure. Additionally, we would have to make a decision of how we wanted to handle backing up data. This would likely consist of storing all records in a backup database, and keeping user game data in a file on the local machine that would be updated after every save. For our simple deployment, we have to run our Java code before the game, since one of our computers will be acting as the server, giving access to the login/register functions. This would be a non issue if we used a hosted database service.

The medium on which our game would be available depends on the platform. Depending on how many platforms we want to provide our game for, it would likely be easiest to make a website for the game and host the executables/binaries for all platforms there. In order to gain attention for our game, it would be beneficial to offer the game on the Apple Store and the Microsoft Store. For hosting a website, we would likely use Amazon Web Services to keep costs low.

End users would simply have to get and run the executables/binaries we provide to play the game (with no extra configuration). The only possible exception to this rule would be if we packaged for Linux and users had to make the build themselves. In this case, we would provide a CMake file or something similar, along with documentation, so that building is not a headache for these users. Another option would be to package the game with Flatpak or Lutris, as both of these options are in line with the movement of the Linux gaming community and would help make our program easier to use and easier to find.