Dungeon Positioning System

Project Team: Elizabeth Dong, Gregory Purvine, Spencer Whyatt, Phillip Jones, Ray Cockerham

https://github.com/gnpurvin/CSC490DPS

Table of Contents

Project Definition

Playing tabletop RPGs using only pencil and paper can be very time consuming. Keeping track of all the tiny details wastes a lot of time that could be better used playing the game. Our system allows for all of this boring bookkeeping to be handled automatically. It will also allow for players to play tabletop RPGs across long distances, so that friends don't need to travel to meet in person in order to play their favorite tabletop RPG.

We want to create this software so that people who want to play games can spend more time doing just that, and less time doing behind the scenes work. Having that behind the scenes work done by a machine will allow the players to spend more time fully enjoying the game. We would also like to incorporate an online aspect, so that friends far away can play together without having to meet in person.

We will build a system that keeps track of players' character sheets and their locations on the game board. It will include panels that display a person's character sheet and display the game board, which will include the locations of all player-characters and non-player-characters currently in play. The system will be able to handle character movement along the game board. It will also give the Dungeon Master to make changes to the game board on the fly. This will allow them to add things to the game board during the session, like enemies or obstacles.

Git Hub Repository Link: https://github.com/gnpurvin/CSC490DPS

Project Requirements

Functional Requirements

1) Maps: The system will display a map to all players and the Dungeon Master, so that everyone can see where all players, enemies/non-player-characters, and terrain are on the field. The system will have the ability to randomly generate a map for the session, or allow the Dungeon Master to create/import their own map. The Dungeon Master will have the ability to make changes to the map during the game session. There will be a several tokens that can represent player characters, nonplayer characters, and monsters. These tokens are movable by the users by dragging and dropping. Token positions on the map updates live on each player's screen. DMs can make a custom map by starting off with a blank grid map with which they specify the size of. Then they can drag and drop environmental objects, such as trees and walls, on to the map. These maps are editable during the session and all players may be able to see the changes as their game progresses. DMs also have the option use a random map generator to create a map instead of starting from scratch. This random map generator takes in account to how big the DM wants the map as well as a specific setting. Then the generator creates the random map that is still editable to the DM only.

2) Characters: The system will display a player's character sheet. The Dungeon Master will also be able to see the character sheet for any non-player-characters or enemies on the board. For efficiency's sake, users have the option to only open an abridged version of their character sheet that only display vital information. Character sheets are saved as files on the player's local computer. It will also allow for users to create new characters. They will be able to either create a new character using preset stats for their particular class, be able to manually input stats for their character, or roll for their stats using the built in dice roller. DPS will use Dungeons and Dragons 5th Edition character sheets.

3) Game Sessions: The system will keep track of individual game sessions. A map, along with all tokens on the map, will be saved as part of that session. The Dungeon Master will have the ability to create new sessions. Players will then be able to join the session their Dungeon Master has created via a randomly generated code tied to the session.

4) Dice Roller: The system will have a built in dice roller. The user will be able to choose between 7 types of dice to roll, and the system will use a random number generator to simulate the rolling of these dice. The user will have the ability to roll 4, 6, 8, 10, 12, 20, or 100-sided dice.

5) Game Log: The game log will display the results of all rolls using the built in dice roller. It will also describe all movement on the game board and all additions to the game board made by the Dungeon Master. It will also function as a text chat for the game, allowing players to communicate with each other. This text chat should also add to the role-playing aspect of the game, as characters will be able to talk to one another.

Usability

The system will have a graphical user interface, to allow for ease of use for the user. The GUI will include sections for the game board, players' character sheets, the built in dice roller, and the game log/text chat. In terms of performance, the system needs to be efficient enough so that users do not experience any extraordinary delays or lag in gameplay. Delays are theoretically acceptable (though not desired), so long as they do not interfere with the users' game experience.

System

Hardware: Computer with Java installed, 2GB RAM
Software: Java, along with required libraries/extensions
Database: MySQL, JDBC MySQL Connector

Security

Security is not a major concern for our application. Our main method of security is ensuring that only the players and Dungeon Master are allowed to join the game session. To that end, we believe that having a randomly generated code serve as a passcode to join a game session may be sufficient. The code would be randomly generated by the Dungeon Master upon creation of the session. The Dungeon Master would then be able to share it with their players, who would use that code to join the session. Non-DM players of a session are restricted from accessing the map editor as well as other tokens that are not their own character. Only the DM can edit their own sessions' maps and all included tokens.

Project Specification

Domain

DPS is meant for tabletop hobbyists. Its intent is to be used as a companion application for tabletop roleplaying games.

Libraries / Frameworks / Development Environment

Most of our system is built from scratch. We use the MySQL JDBC connector to connect with the database, which is hosted on Amazon's AWS RDS. We used Java's built in multithreading to build the Connectivity subsystem, which follows a client/server model. Our user interface is built using the Java FXML framework. The Map and Quality of Life subsystems are built entirely from scratch.

Platform

Our system is built for use on a desktop platform. We believe this platform best suits the needs of our system. A mobile application would not be able to contain the information as well as a desktop, due to the smaller screen size. We also believe that making the system a desktop application will allow us to reach a wider audience, given our time frame. To properly reach a mobile audience would likely necessitate making iOS and Android applications, while a desktop audience only requires making one web application that can be run on all environments.

Genre

Application

System - Design

Subsystems

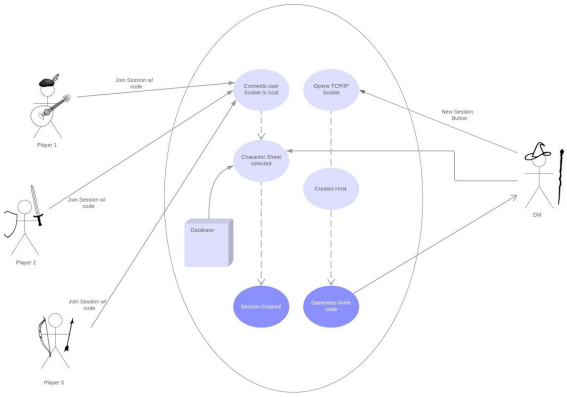
1. User Interface
2. Map
3. Connectivity
4. Database & Database Connector
5. Assorted Quality of Life Features (Sheets, Dice Roller, Game Log, etc.)

Overall Operation

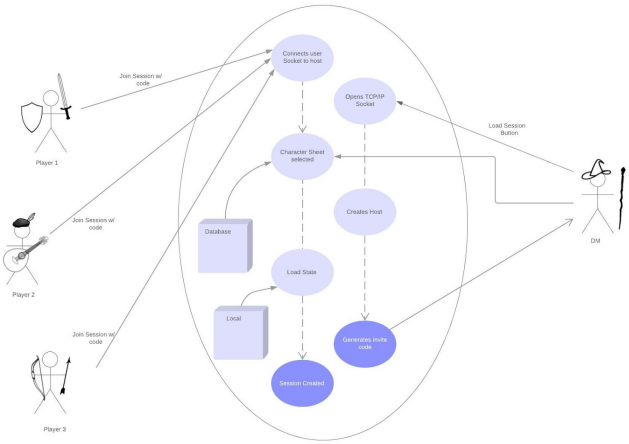
The User Interface subsystem will handle all interaction with the user. The Map subsystem will handle map creation, and editing. The Connectivity subsystem will handle sharing of all necessary assets to the players and connecting them to the system. The Database subsystem will contain the database where all assets are stored and the connector that allows the other subsystems to access the database. The Quality of Life subsystem will handle all other features in our system, hereby dubbed as quality of life features. These include the maintenance of character sheets, the operation of the dice roller, and the handling of the game log.

Use Case Diagrams

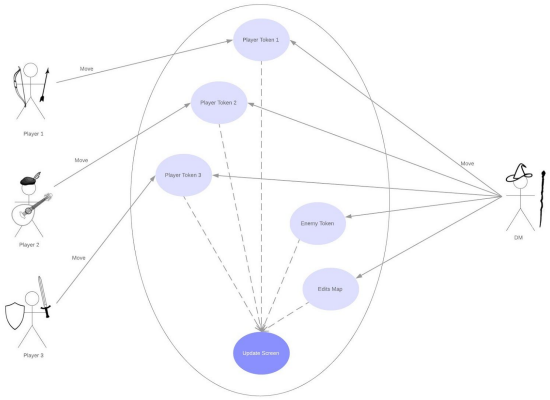
Creating a Session



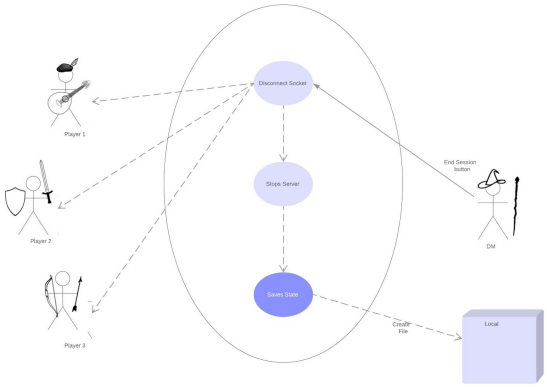
Loading a Session



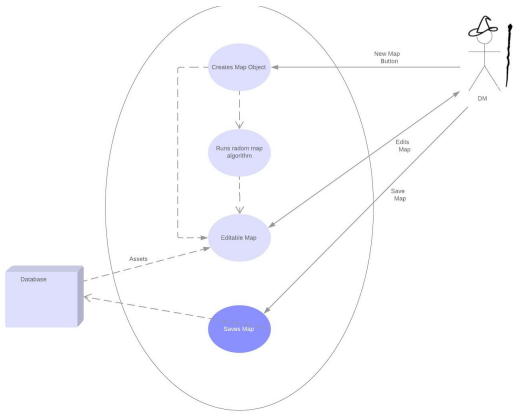
Playing a Session



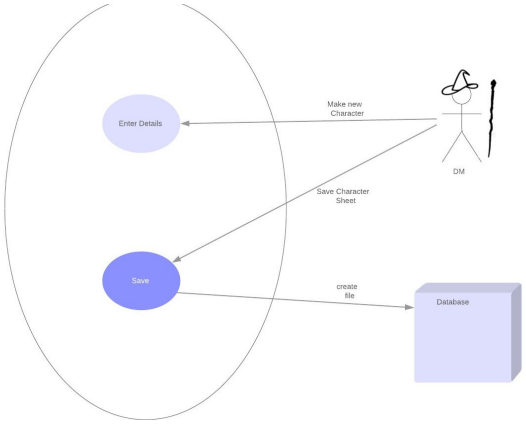
Ending a Session



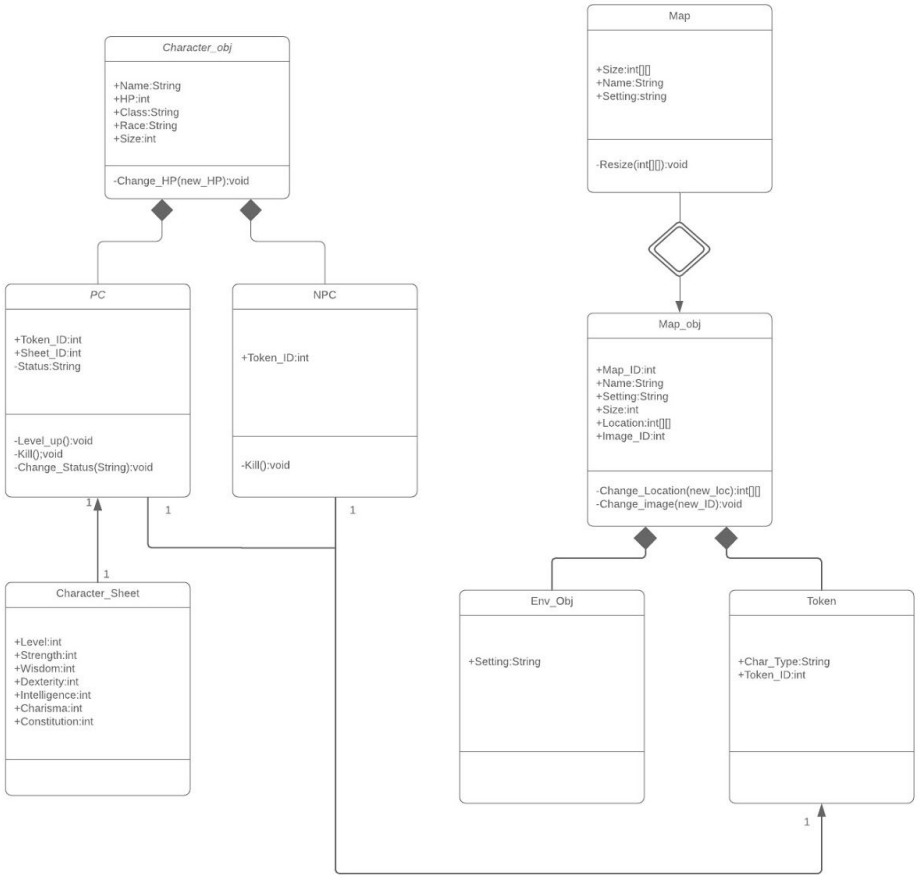
Making a Map



Making a Character

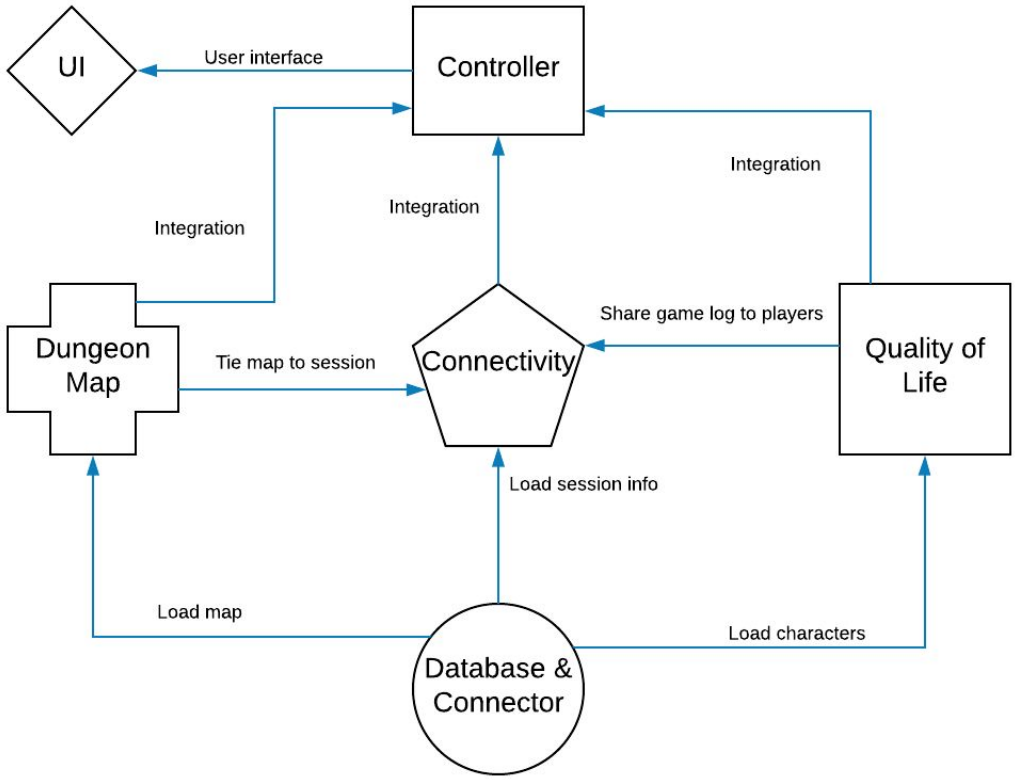


UML Diagram


Character_obj
+Name:String
+HP:int
+Class:String
+Race:String
+Size:int
-Change_HP(new_HP):void
PC
+Token_ID:int
+Sheet_ID:int
-Status:String
-Level_up():void
-Kill();void
-Change_Status(String):void
NPC
+Token_ID:int
-Kill():void
Character_Sheet
+Level:int
+Strength:int
+Wisdom:int
+Dexterity:int
+Intelligence:int
+Charisma:int
+Constitution:int
Map
+Size:int[][]
+Name:String
+Setting:string
-Resize(int[][]):void
Map_obj
+Map_ID:int
+Name:String
+Setting:String
+Size:int
+Location:int[][]
+Image_ID:int
-Change_Location(new_loc):int[][]
-Change_image(new_ID):void
Env_Obj
+Setting:String
Token
+Char_Type:String
+Token_ID:int
1
1
1
1
1
1

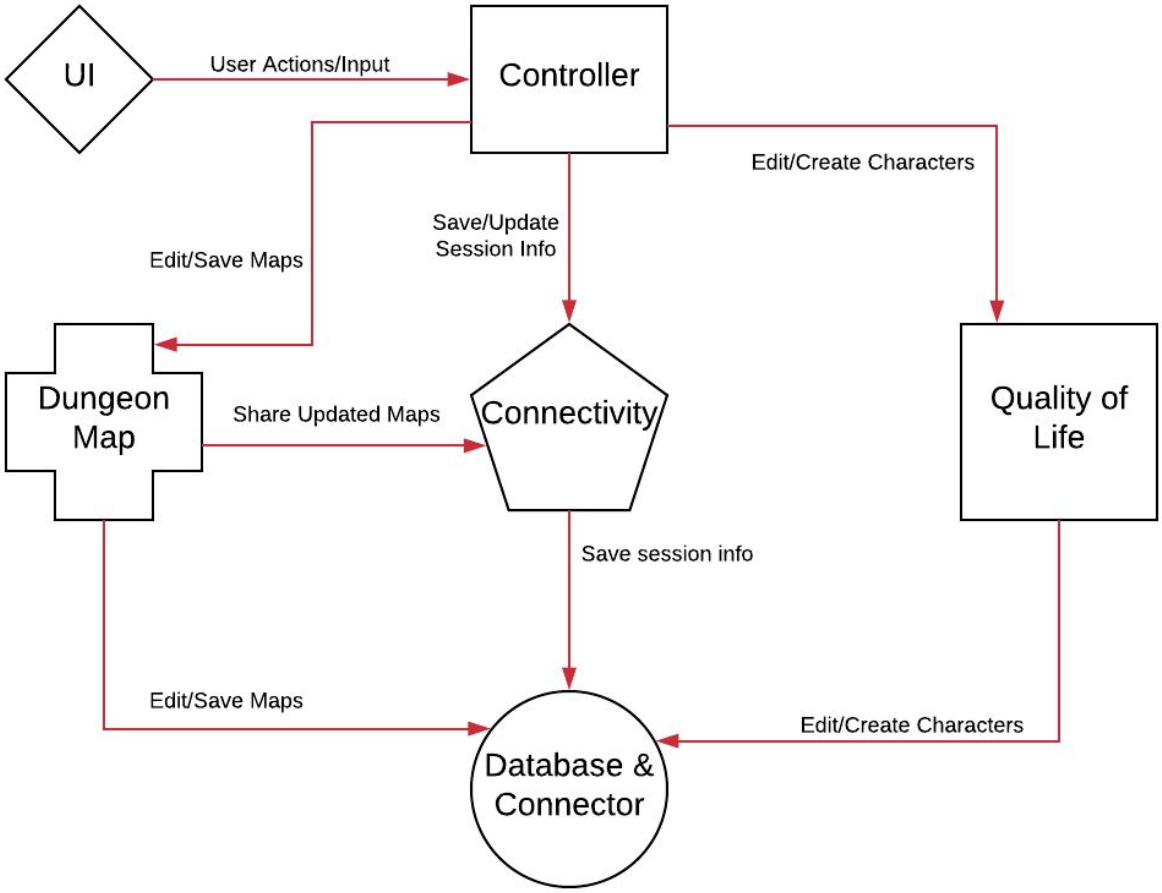
Sub-System Communication

DataFlow: Inputs



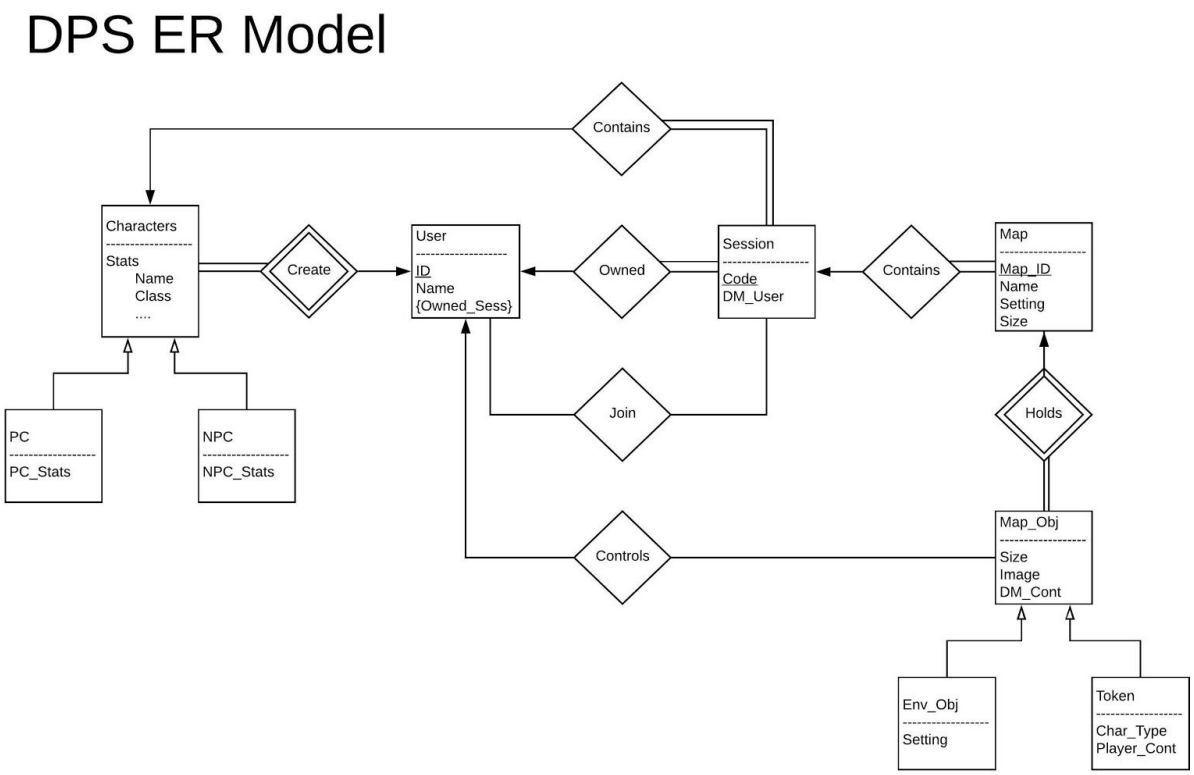
All system assets will be stored in the database. For each subsystem to access it's stored data, it will have to interact with the database connector. The database connector will give each of the subsystems the information it needs to operate from the database. The dungeon map subsystem will have to share the map with the connectivity subsystem, so that the map can be shared with all players. Similarly, the quality of life subsystem will share character sheets, NPC sheets, the dice roller, and the game log with the connectivity subsystem, so that the players have access to all of those resources. The controller then integrates the map, connectivity, and quality of life subsystems, so that the user interface can interact with the integrated system.

DataFlow: Outputs



The user interface will catch all user actions and input and pass them to the controller. The controller will then process these actions and input and determine what subsystem is being called to act by the user. It will give changes to game maps to the map subsystem, which will then give those changes to the connectivity subsystem, so they can be shared with all players, and to the database subsystem, so that the updated map can be saved. The controller will give updated session info to the connectivity subsystem, where it will be passed along to the database subsystem to be saved. Changes to character sheets will be given to the quality of life subsystem, where they will be sent to the database subsystem to be changed.

Entity Relationship Model (ER Model)



user
username
password

session
sessionID
sessionName

map
mapID
mapValues

token
tokenID
tokenSizeX
tokenSizeY
tokenValues

ownedSession
username
sessionID

sessionMaps
sessionID
mapID

mapContain
mapID
tokenID

ownedToken
username
tokenID

System - Analysis

Data Dictionary

(Minimum View) Name	Data Type	Description
Name	String	Name of a character
Race	String	Race of a character
Class	String	Character's Class
Alignment	String	Character's Alignment
Level	Int	Current level of a character
Max Hit Points	Int	Character's Hit Point Maximum
Strength	Int	Character's Strength Ability Score
Constitution	Int	Character's Constitution Ability Score
Dexterity	Int	Character's Dexterity Ability Score
Intelligence	Int	Character's Intelligence Ability Score
Wisdom	Int	Character's Wisdom Ability Score
Charisma	Int	Character's Charisma Ability Score
Armor Class	Int	Character's Armor Class
Proficiency	Int	Character's Proficiency Level
Hit Die	Int	Size of Character's Hit Die
Current Hit Points	Int	Character's current hit points

Algorithm Analysis

Big - O analysis of overall System and Sub-Systems

- O(n)
  - Insert to table with unique fields
  - Update
  - Select
  - Delete
- O(log(n))
  - Insert to table with unique fields and indexes
  - Update with indexes
  - Select with indexes
  - Delete with indexes
- O(1)
  - Insert to table without unique fields
  - Establishing connection to DB

Project Scrum Report

Product Backlog

Dice Roller	Client Program
Dice Roller Testing	Server Program
Database	GUI Home Page
Map Generation	Server Multithreading
Database Hosting	Token Object
GUI Connect Page	Game Log
Database Host Connection	Character Sheet
Map Controller	Game Log
Database Connector	GUI Game Page
Map Rendering	Map-Connectivity Integration
Quality of Life-Connectivity Integration	Database Connector Integration
Connectivity-GUI Integration	

Sprint Backlogs

Sprint 1

Task	Dice Roller & Testing	Client/Server Implementation	Database Implementation	GUI Home Page	Map Generation
Team Member(s)	Greg	Ray	Elizabeth	Spencer	Phillip
Story Points	5	5	21	21	13
Subsystem	QoL	Connectivity	Database	GUI	Maps

Sprint 2

Task	Firewall Bug	Server Multithreading	Database Hosting	GUI Connect Page	Token Implementation
Team Member(s)	Greg, Ray	Greg, Ray	Elizabeth	Spencer	Phillip
Story Points	3	8	8	13	3
Subsystem	Connectivity	Connectivity	Database	GUI	Maps

Sprint 3

Task	Game Log	Client/Server Implementation	Database Connection	GUI Character Sheet	Map Controller Implementation
Team Member(s)	Greg, Ray	Ray	Elizabeth	Spencer	Phillip
Story Points	8	8	13	13	8
Subsystem	QoL	Connectivity	Database	GUI	Maps

Sprint 4

Task	Game Log Integration	Sending Map over Connection	Database Connector	GUI Game Page	Basic Map Rendering
Team Member(s)	Greg, Ray, Spencer	Ray	Elizabeth	Spencer	Phillip
Story Points	13	8	13	13	21
Subsystem	QoL, Connectivity, GUI	Connectivity	Database	GUI	Maps

Sprint 5

Task	Documentation	Connectivity Integration	Database Integration	GUI Finalization	Map Integration
Team Member(s)	Greg	Ray	Elizabeth	Spencer	Phillip
Story Points	8	13		3	
Subsystem	General	Connectivity	Database	GUI	Maps

Burndown Report

Sprint 1:

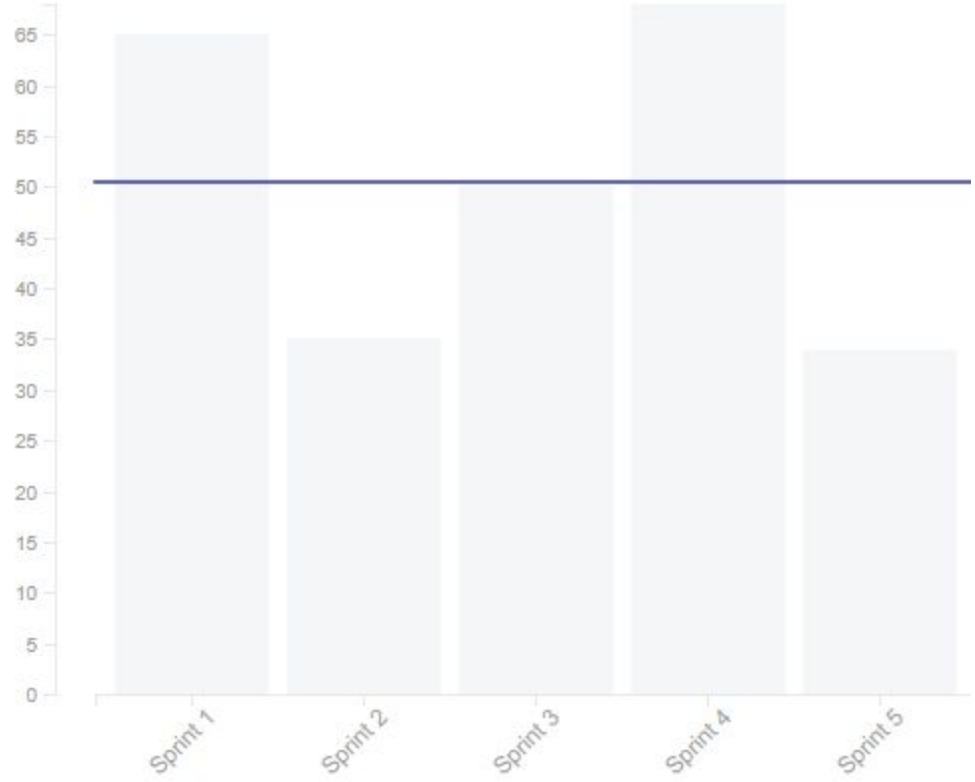
Sprint 2:

Sprint 3:

Sprint 4:

Sprint 5:

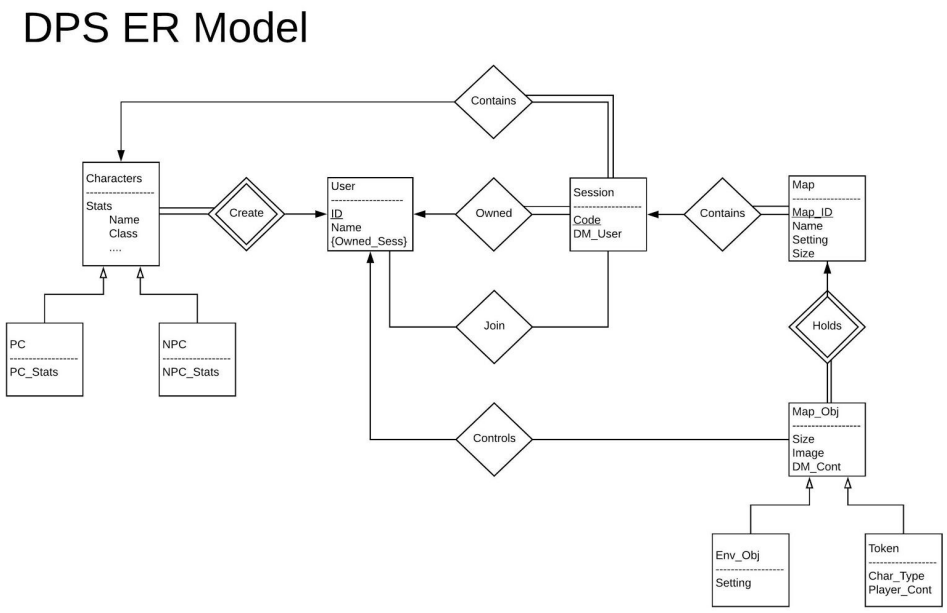
Velocity Report



Subsystems

7.1 Database – Elizabeth

Initial design and model



Design choices

The database includes 4 objects:

- Map
    - mapID (primary key)
    - mapValues
- Session
    - sessionID (primary key)
    - sessionName
- Token
    - tokenID (primary key)
    - tokenSizeX
    - tokenSizeY
    - tokenValues
- User
    - username (primary key)
    - password

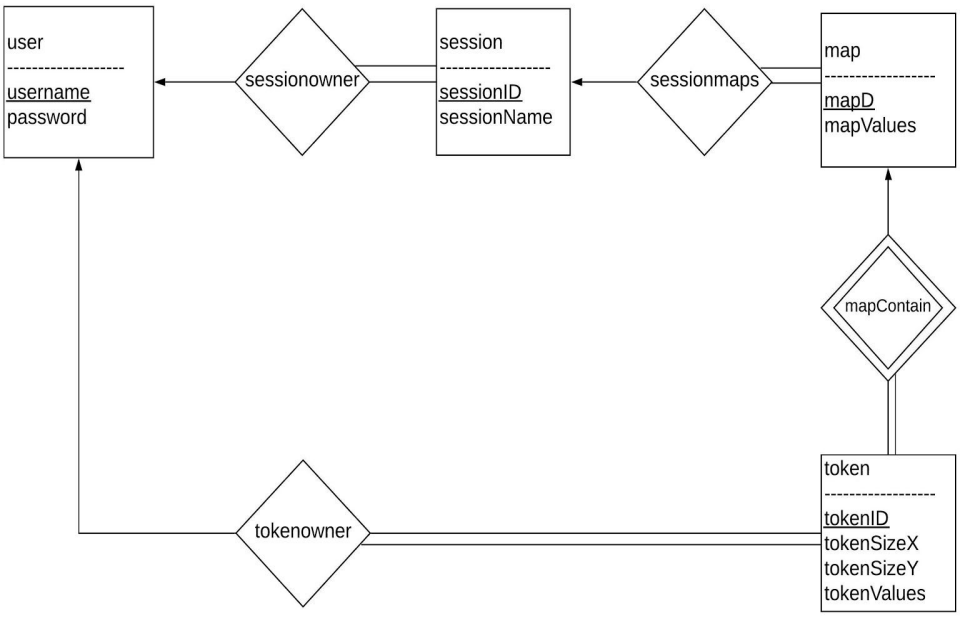
There also exists 4 relationships:

- mapContain
    - mapID (foreign key)
    - tokenID (foreign key)
- sessionmaps
    - sessionID (foreign key)
    - mapID (foreign key)
- sessionowner
    - username (foreign key)
    - sessionID (foreign key)
    - IPAddress
- tokenowner
    - tokenID (foreign key)
    - username (foreign key)

Data dictionary

- User: A person using the system. Can be either a Dungeon/Game Master or Player Character
    - A user is a DM whenever they create a session
    - A user is a player whenever they join a session made by someone else
- Session: Is an identifier to connect a users to certain maps
- Map: The game board being played on. A map contains tokens.
- Token: Represents a player or enemy on the game board. Is controlled by their token owner.

If refined (changed over the course of project)



A singular user type was implemented in the final database design rather than separating between players and DM. The dichotomy was not especially useful because they were completely alike aside their relationship to session and specializing a user offered no apparent advantage for this system. The two were combined for simplicity. There was a switch to using a single token type rather than divide it between environmental objects and players for a similar reasoning. Rather than use a joinedSession table to keep track of players in a session, tokens could be used to see if a player has joined a session. If a player possesses a token in a map, they are indirectly linked to the session that map is related to.

Scrum Backlog (Product and Sprint - Link to Section 6)

Sprint	**Task**	**Points**
Sprint 1	Database Implementation	21
Sprint 2	Host Database	8
Sprint 3	Connect to Database Host	13
Sprint 4	Database Connector	13
Sprint 5	Database Integration	5

Coding

The database is hosted on AWS RDS and was created using MySQL. The database connector was coded in Java using JDBC. Connector/J library was used to create a connection to the database.

User training

In order to establish a connection, the method connect() may be called. This will return a connection object. A connection must be closed by closeCon() method that will release the resources that were held by the connection.

To create a user, makeUser is called and must be passed a username and password. The username must be unique as it is the primary key for the user table. In order to authenticate a user, the method Authenticate is passed a user given username and password. This method will pass a true if it finds the inputted username and password to be a match in the database or false otherwise.

To make a session, the method makeSession is called and passed the username of the creator and a user inputted sessionName. This method will create and return a unique sessionID. Deleting a session is as simple as passing the sessionID to the deleteSession method. The database holds the current IP address of the DM in order for all players to be able to communicate. The method setIP is able to set the DM's IP address to the database given the sessionID and the DM's IP address. The getIP returns the IP address given the sessionID. In order to display all of a DM's created sessions, getSessionList must be called and passed the username. This will return an ArrayList<String> of session names that the user has created. It is possible to edit the session's name by passing the sessionID and the new session name to editSession.

Pass to makeMap a string of map values and a sessionID for it to be tied too. This method makes a unique mapID and returns it. deleteMap must be passed a mapID in order to remove a map from the database. To receive an ArrayList<Integer> of maps belonging to a session, pass the sessionID to getMapList. editMapValues will update a map's values in the database given the mapID and the new map values.

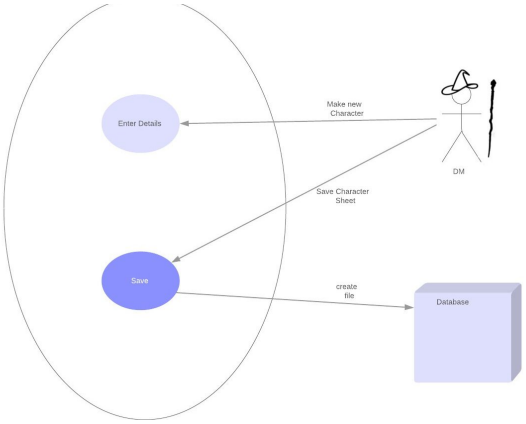
When a player joins a session, there must be a check to see if the player has existing tokens on a given map. To check, getTokenID is called and passed the username, sessionID, and mapID. This method will return the user's tokenID. If this is the user's first time using map on a session, a new token must be made. Create a token by calling makeToken and passing it the username, size x coordinate, size y coordinate (for a player token, it should by 1x1 on default), mapID, and the string of tokenValues. This method will return a unique tokenID. deleteToken only requires a tokenID to delete a session from the database. To edit a token's values, pass tokenID and the new token values to editTokenValues. To get a token's values in the database, call on getTokenValues with the tokenID.

Testing

```
//test connect
con = connector.connect();

//test makeUser
//connector.makeUser("Aesthellar", "letmein");

//tests Authenticate
//boolean bool = connector.Authenticate("Aesthellar", "letmein");
//System.out.println("Authenticating... " + bool);

//test makeSession
//int sessionID = connector.makeSession("DND Campaign", "Aesthellar");
//System.out.println("Your new session's ID is: " + sessionID);

//test setIP
//connector.setIP(89, "693.051.341.265");

//test getIP
//String IP = connector.getIP(89);
//System.out.println(IP);

//test getSessionID
//int sessionID = connector.getSessionID("Aesthellar", "DND Campaign");
//System.out.println(sessionID);

//test getSessionList
/*ArrayList<String> sessions = new ArrayList<String>();
sessions = connector.getSessionList("Aesthellar");
System.out.println("Aesthellar's owned sessions: ");
for (int i = 0; i < sessions.size(); i++) {
    System.out.println(sessions.get(i));
}*/

//test editSession
//connector.editSession(49, "Tester");

//test deleteSession
//connector.deleteSession(49);

//test makeMap
//int mapID = connector.makeMap("mapstuff", 849);
//System.out.println("Your new map ID is: " + mapID);

//test getMapList
/*ArrayList<Integer> maps = new ArrayList<Integer>();
maps = connector.getMapList(849);
System.out.println("Maps of session 849: ");
for (int i = 0; i < maps.size(); i++) {
    System.out.println("MapID: " + maps.get(i));
}*/

//test getMapValues
/*String mapValues = connector.getMapValues(24);
System.out.println("MapValues: " + mapValues);*/

//test editMapValues
//connector.editMapValues(24, "newmapstuff000000000000000000000000000000000

//test deleteMap
//connector.deleteMap(24);

//test makeToken
//int tokenID = connector.makeToken("Aesthellar", 1, 1, 594, "tokenstuff")
//System.out.println("Your new token ID is: " + tokenID);

//test getTokenID
//int tokenID = connector.getTokenID("Aesthellar", 849, 594);
//System.out.println("Token ID = " + tokenID);

//test getTokenValues
//String tokenValues = connector.getTokenValues(20);
//System.out.println(tokenValues);

//test editTokenValues
//String tokenValues = "map values 00000000000";
//connector.editTokenValues(20, tokenValues);

//test deleteToken
//connector.deleteToken(117);

//test closeCon
//connector.closeCon();
```

7.2 User Interface – Spencer

Initial design and model



The character sheets are implemented as XML files that contain all of the characters stats. We did this because all that's really necessary for the character sheets is to store and display them for the user; there is no need to use the data beyond that. We chose to implement the UI using Java FXML for its ease of use with XML files.

Design choices

We decided to Use Java FXML since all of the graphics can be implemented using java scene builder along with the readability of graphics and logic. We decided to use XML files for ease of use with FXML as well as an average user would not be able to change it as easily.

Data dictionary

DM- dungeon master
FXML document - document that defines all graphic and style options.
Controller - File that defines logic tied to specific FXML document.

If refined (changed over the course of project)

Only refinement was to log into the database at the beginning of the program rather than later. The pro to this is login is only done once rather than multiple times. Cons are that players will not be able to create characters if the database is down.

Scrum Backlog (Product and Sprint - Link to Section 6)

Sprint	Task	Points
Sprint 1	GUI Home Page	21
Sprint 2	Connect to Session Page	13
Sprint 3	Character Sheet	13
Sprint 4	Game Session Page	13
Sprint 5	GUI Finalization	3

Coding

GUI has been designed using Java FXML along side of other subsystems it can call for most of the difficult logic.

User training

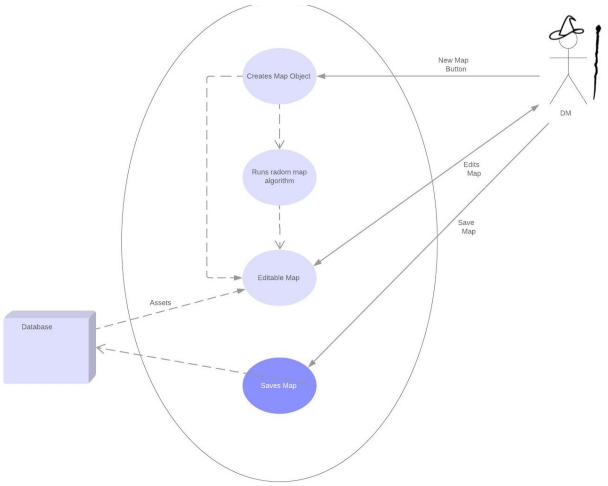
At login If the user has a username and password in the database they will login in using those credentials otherwise the user will have to create a new username and password. Once logged in the user will have the option to either play as a DM or player. If DM is selected, then the user will have to choose an existing session or create a new session. Once session has been selected the DM can choose to create/edit maps or characters, as well as start the session. Once session has been started the DM will receive a code in the bottom right of the screen, this code will be used by players to join the dm's session. Once player has been selected from the main menu they can either start session or create/edit characters. When all players are in the session they may begin playing.

Testing

All testing will be done on the integration of other subsystems.

7.3 Map Generation – Phillip

Initial design and model



Data dictionary

- Map Controller object: has a Map object, methods for drawing and editing the map, accessed by users through the GUI
- Map: the collection of data that represents the map in its entirety, consists of a few global variables that indicate the structure and look of the map, as well as an ordered 2D array of Tiles
- Tiles: a single piece of the map, has x and y coordinates and a few boolean variables that indicate properties of the tile, such as whether it's part of a room or hallway, if it's currently occupied or not, etc., etc.

If refined (changed over the course of project)

- Original choice of library for graphics rendering not compatible with GUI
- Switched from paint to canvas
- Actually works, requires instantiation of fewer objects
- Dungeon generation is now recursive, but graphical representation is still basically constant, takes only 2 steps per tile

Scrum Backlog (Product and Sprint - Link to Section 6)

Sprint	Task	Points
Sprint 1	Map Generation	13
Sprint 2	Token Implementation	3
Sprint 3	Map Controller	8
Sprint 4	Basic Map Rendering	21
Sprint 5	Map Integration	5

Coding

The Map subsystem was developed using object oriented methodologies in Java. We used a hierarchical approach, with each class using one or more instances of a lesser class

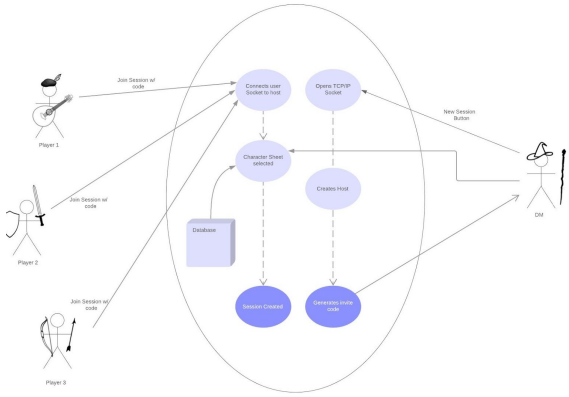
Testing

Testing consisted mostly of introducing changes incrementally and ensuring that the whole subsystem works even when given edge cases. Error checking and argument cleanup is needed at most portions of the process, but ensuring good inputs early on makes the rest of the algorithm flow smoothly. Because the algorithm is somewhat complex and can be inefficient under certain arbitrarily large inputs, safeguards were needed to protect against stack overflow errors. This was done by designing the algorithm so that it prevents excessive usage of recursion.

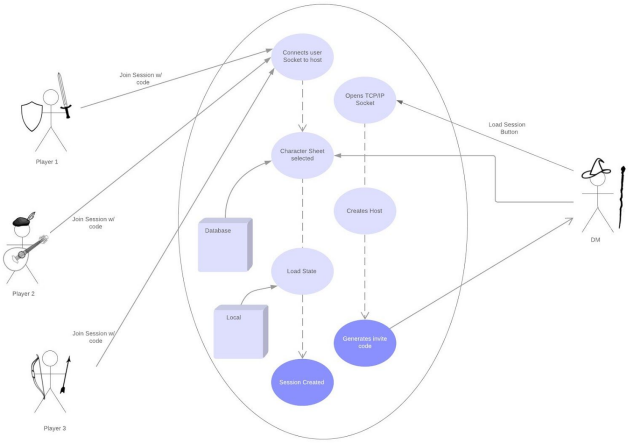
7.4 Connectivity – Ray

Initial design and model

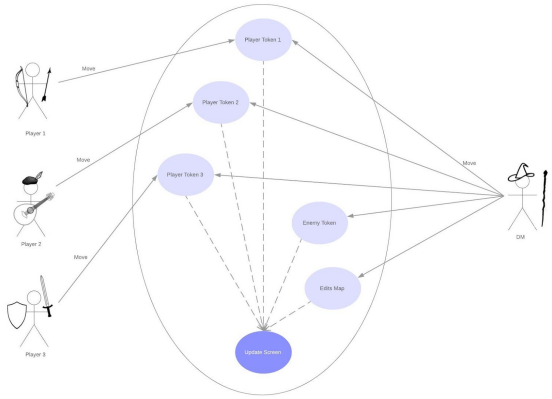
Creating a Session



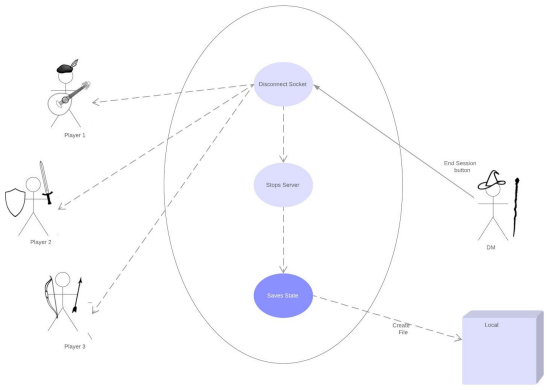
Loading a Session



Playing a Session



Ending a Session



- Design choices

Data dictionary:

sessionCode: this is the unique code generated to identify your session in the database.
ServerWorker: the subclass implemented to handle individual connections
Client: someone connecting to the game.

If refined (changed over the course of project)

The only design aspect that was changed over the course of the project in my subsystem was no longer authenticating usernames and passwords on my level. We decided that there is no need to authenticate users twice. So in an effort to be more efficient, the database authenticates usernames and afterwards spawns a client object simply passing a string for the username and a session code to get the host's IP address. This string sets the username in the chat, and nothing else. This proved to be a lot easier for everyone involved.

Scrum Backlog (Product and Sprint - Link to Section 6)

Sprint	**Task**	**Points**
Sprint 1	Basic Client/Server	5
Sprint 2	Server Multithreading	8
Sprint 3	Client/Server Implementation	8
Sprint 4	Sending Map	8
Sprint 5	Connectivity Integration	13

Coding

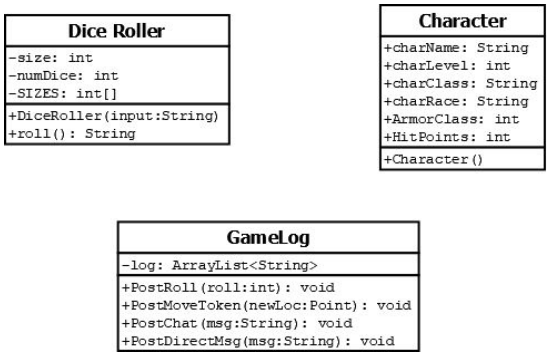
The connectivity subsystem was implemented using multithreading in Java and followed object oriented principles. The Server class extends the Thread class in order to be able to constantly listen for new connections and service those connections at the same time. When a new connection is received, the Server class creates an instance of the ServerWorker object, which handles everything about the connection to the server. The client class is also multithreaded, and implements two abstract interfaces in order to display messages from all users. One thread listens for messages, and the other can send data.

Testing: the majority of testing was done using telnet to locally connect to my own machine. The later stages testing was done using the GUI to send messages etc back and forth across the internet connection.

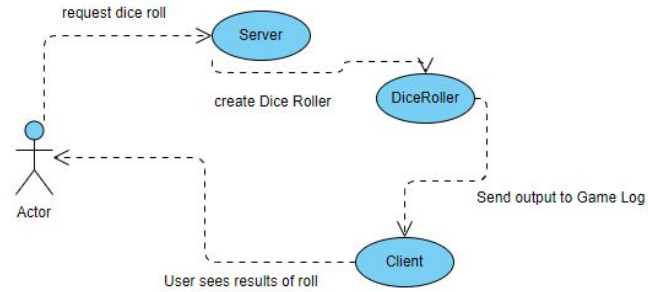
7.5 Quality of Life – Greg

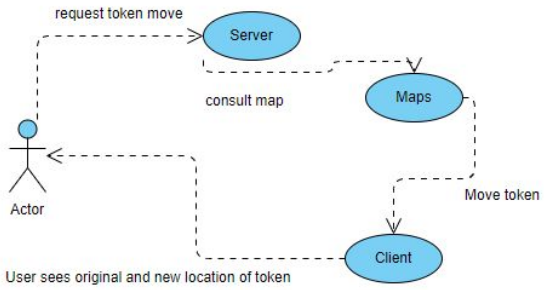
Initial design and model

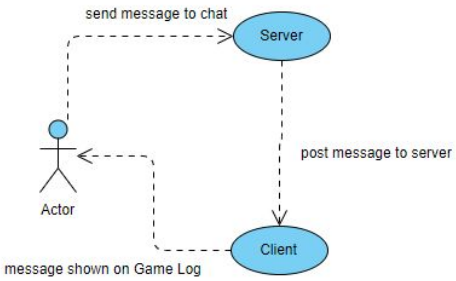
UML Diagram



Use Case Diagrams







If refined (changed over the course of project)
We chose to have the UI directly handle the creation and display of character sheets. Character sheet functionality as I had originally envisioned it was over-engineered. All we needed for our purposes was for the user to be able to create, save, and load character sheets. To that end, the UI creates an XML file that stores each character sheet, which can then be loaded for display to the user.

The game log functionality also ended up wrapped into another subsystem. Once it came time to implement the game log functionality, it became apparent that our implementation of the connectivity subsystem could easily serve that purpose with very little additional work. Rather than make an object to maintain a game log, all communication between the client and server programs would be outputted to a text box on the UI, which would serve as the game log. To facilitate this implementation of the game log, the dice roller was integrated with the connectivity subsystem.

Scrum Backlog (Product and Sprint - Link to Section 6)

Sprint	**Task**	**Points**
Sprint 1	Dice Roller	3
Sprint 2	Dice Roller Testing	2
Sprint 3	Game Log	8
Sprint 4	Game Log Integration	13

Coding
Coding was done using an object oriented design in Java.

Testing
The Dice Roller was tested for functionality and randomness using automated testing. The JUnit 5 library was used to perform whitebox testing.