

Ricardo Hernandez
Oscar Alvarez
Rogelio Ramirez

Dungeons and Dragons Encounter Calculator

Introduction

The genesis of our project came from Dr. Robert Schweller. He asked if the odds could be calculated for an encounter designed by the Dungeon Master. We undertook that question and decided that the best way for us was to simulate it and run it thousands of times. By taking the win rate of these thousands of simulation, the project would produce an approximate number for the odds of winning.

Overview

Dungeons and Dragons is a tabletop role playing game. Each player is given or makes a custom character for themselves to roleplay. The Dungeon Master is the one player without a custom character. Instead they are in charge of the adventure that is presented to the other players. The Dungeon Master acts as judge, narrator and antagonist to the players as they roleplay. During the course of play, the players will want to do certain actions(attack, climb up a wall, decipher text), and the Dungeon Master will give them a target number. This number is known by a few names such as Difficulty Check, and Armor Class. The player will then roll a die that generally has twenty sides and start off with the number rolled. That number then has a variety of modifiers added to it. This total is used to see if the target number is matched or

beaten, and if it is beaten or matched, the action succeeds. This is repeated throughout the whole entire game.

While this principle mechanic is used throughout all of Dungeons and Dragons, it is more structured in the area of combat. A simplistic version of combat follows the list below:

1. Each player and monster rolls a d20 to get turn order.
2. In the turn order, the players and monsters do an action and movement. (Attack, heal, etc)
3. Each attack has a roll to see if it hits the monster by meeting or exceeding a target number. The target number is referred to as Armor Class
4. If there is a hit, hit points are subtracted from the HP total.
5. Repeat for next player or monster in the order
6. Combat ends when one side is “dead”

Problem

The problem we are trying solve is if there is a way to provide an approximate number that will tell the Dungeon Master the odds of a party of players winning against an encounter they designed.

Current Solutions

There are a number of solutions to this problem, but they all seem to have inherent problems. These problems do mean that the solutions do not work, but rather that they have a cost that makes them less than ideal

The first solution is for the Dungeon Master to play out the encounter during their preparation time. The Dungeon Master would play the entire encounter by themselves, assuming they had access to their players' character sheets. The process can be time consuming and cause the Dungeon Master to focus away from more important aspects of running an adventure.

The second solution is quite a bit faster, but still tedious. The Dungeon Master can use the various rules in the Dungeon Master's Guide to design a level appropriate encounter. The tedious part in this solution would be finding all the players' levels, all the monsters levels, and referencing the various tables. Enacting this solution will still take a 10-20 minutes due to various things needed to be looked up, but the time needed can be drastically lowered with the use of online tools that automate all the math needed. One such example is the web application called "Kobold Fight Club" found at kobold.club. This solution still only provides an "Easy, Medium, Hard, or Deadly" scale and would still be difficult to judge without a clear win rate percentage.

The third solution is the most commonly used with experienced Dungeon Masters and is useless for new Dungeon Masters. They compare encounters based off their experience with other encounters. This experience is built over time, with the Dungeon Masters seeing different encounters while presenting the game to their players, many times with pre-prepared adventure modules.

Our Solution

The solution that we came up with was to have the Dungeon Master select the type of monster and how many for their encounter and also select the type of characters for the players.

These include the Cleric, Fighter, Rogue and Wizard. The Dungeon Master would input the number characters as well as their level. Afterwards the Dungeon Master would push the “FIGHT” button and the program would produce the win percentage that can be used for the odds of winning that encounter.

How it Works

We began with how would be simulate the players and creatures. Based on our experience with Dungeons and Dragons, we knew that both monsters and players share many stats. It was decided to create a Creature superclass that a Player class and Monster class would draw from. The Creature class would hold the most of the stats and have most of the methods needed throughout the simulation for either the player or monster. In the diagram at the end of the report, Figure 1, you can see the abstract Creature Class. The two things of note are that it is an abstract class, so another class needs to implement it, and the other is that it implements `Comparable<Creature>`. This interface lets all creatures be sorted. We created an `override compareTo` method that would let Java use to make comparisons for sorting purposes. The value that is being compared is the initiative.

The next major class we created was the Player abstract class, shown in Figure 2. We knew from the rules that each player character class was sufficiently different that they merited to have their own class for implementation, but still had enough common features that they needed a common class scaffold. The player class has the added properties of armor and weapons. It also contains sets of abstract methods that each player class needs to have but must

be custom created for each player class. Figure 2 will also show the final implementation of Cleric, Fighter, Rogue, and Wizard classes.

The Monster class has had the most changes in our plan. Initially we were going to follow the Player Class implementation with an abstract Monster class and individual monster classes. The remains of that can be seen in the code we have left over in the Monster package (folder). Currently the Monster class a specialized constructor that is fed information about the monster and creates an instance of that monster. This way allows us to extract the pertinent stats from a monster database (or JSON file in our case) and instantiate that particular type of monster. We will speak in further detail about how the statistics were extracted. Figure 3 at the end of the report has the implementation for Monster class.

Throughout creating this simulation, it was noted early that many code snippets were being performed multiple times. It was decided to create functions for these common snippets and save them to a Commands class. The commands class contained rolling dice, rolling for advantage, rolling for HP, rolling for stats for player characters, and checking the hit points from monsters in the combat list. The rolls would be performed by random generators found in JAVA. Figure 4 shows the Commands class.

The final major class for running the simulation was the Sim class. This class would accept a list of monsters and players, and then would begin conducting rounds until one side was finished. It would begin by having all the monsters and players roll for initiative, followed by being sorted in initiative order using Java's built in functions for lists and arrays. The combat rounds would begin and in turn order the players and monsters would begin to attack each other. In order to facilitate faster and more realistic target choice, both players and monsters have an

algorithm to find priority targets. In this early stage, the algorithm consists of monsters targeting player character with the lowest hit points. The player characters use the same algorithm but with customization to include ability use, such as: Wizards will use their spells, Clerics will heal, Fighters will use their natural healing and Rogues will use their sneak attack if able. Once the first round of combat is complete Sim will loop it again This will continue until one side is “dead”, meaning that one side has all its members at 0 or less hit points. Once that condition becomes true, the Sim class will determine which side won and add it to the win counter. Sim class will reinitialize the fight, and begin again until ten thousand simulations have been completed. The Sim class then finishes with returning a number as percentage of fights won. Figure 5 shows the Sim class.

Development Standards

During the first quarter of development, standards were not fully in place. For instance, while Git and GitHub were used by the team, different IDEs were being used. This became an issue when trying to make use of the version control within Netbeans. Since Oscar and Ricardo were experiencing these issues, Rogelio suggested to switch to the IntelliJ IDE. Oscar and Ricardo were in agreement with this suggestion, as well as holding a meeting to familiarize themselves with the IDE. During one of the weekly code reviews, Rogelio held a form of seminar in using IntelliJ and its version control functionality. In this one meeting, Oscar and Ricardo became familiar enough to be able to produce compilable code, as well as using the version control built-into IntelliJ. The version control included: commit, push, creation of branches, merging of branches. The use of branches let the team to work individually on their

code without it affecting the others during development. The few times when code ended up in conflict, the the merge tool would be used to resolve the conflicts and get all the code into the Master branch.

As mentioned previously, Git and Github were used among the team. A functionality built-into Github was also used, Github's Issue Tracker. This was used quite early in the development, to primarily assign tasks to team members. Though it was also used for its fundamental purpose, that is alerting team members of any issues such as bugs or miscalculations within the project.

As for the generation of the GUI, Oscar made use of Scene Builder for JavaFX. This standard was not primarily used by any other member other than Oscar. It is still a development standard, however, since no other member was using a variation of this technology. Specified details of this technology will be elaborated later on in the report.

Technologies

At the beginning of development, it was unanimously decided to produce the project in Java. Once development of the project was robust enough, a technology for creating the GUI was decided on. Rather than using the swing library, JavaFX was used. JavaFx is intended to replace swing as the standard GUI library for Java SE, sometime in the near future. Though it was soon discovered that JavaFX is as of now being shipped as a stand alone module rather than being included in Java, as it was originally. This was notable when development as team members had varying versions of Java. For instance Java 10 originally includes JavaFX, while Java 11 does not and it must be included as a module. To simplify the packing and shipping of an executable,

it was decided to build the project in Java 10.

Another major technology used for developing the project was the Jackson JSON Library. Why was this technology essential? Simply put, the monsters and the items were stored in JSON files so parsing the information when creating the monster and item objects was necessary. Given the complexity of the JSON files and objects, custom deserializers were created to help transfer the information into java class objects.

JavaFx

JavaFx was the software platform used for delivering our graphical user interface. It provided the necessary tools and libraries needed for designing and controlling the objects generated on our gui. Despite issues with the version changes between Java 10 & 11 (a few hiccups producing an executable jar), JavaFX is well documented and there were no issues implementing objects from the gui into the project.

JavaFx vs Java Swing

The integrated development environment provided by IntelliJ already came with its own platform for gui development known as Java Swing. Unfortunately, after a week of using it (early in the semester), Oscar (assigned to the gui) found it difficult to use and was unable to freely customize the layout. This is when Oscar introduced the possibility of using JavaFx at one of the meetings, which was then accepted.

It was easier to design the gui using JavaFx than it was using IntelliJ's Java Swing. JavaFx also had a better media library (audio, video, images) supporting it that Java Swing could not provide.

Scene Builder

Scene Builder is a visual design layout tool that was used to help create the user interface for the project and can be integrated directly into IntelliJ. You could open your fxml through IntelliJ with Scene Builder, and your fxml will be updated in IntelliJ as you customize it in Scene Builder.

Scene Builder provides an easy-to-use drag and drop feature, that uses no code, which made it easy to customize the gui. After dropping an object onto a newly created scene, you can associate that object with an initialized/declared variable in your code in IntelliJ. The scene can then be associated to a specified controller class in your code. Then using JavaFx features, methods added to that controller class handles any events associated with that object.

Jackson JSON Library

Making use of this technology was by no means straight forward. The “objects” stored in the JSON files were rather unpredictable. Allow us to elaborate, some objects had their values stored in ways that differed with others. For instance, armor class would be stored in an array as a number for some monsters, but then for others it would be stored as an object in the array with possible other arrays in the structure. IE [14] , vs [{ AC:14 , From : [natural armor] }],. As for parsing the items, there were three different types of items that would have to be accounted for in

one single JSON file. These three different types of objects all had values that were not present in others. For example, armor items have a value called “ac” or armor class which is something not present in the ammunition and weapon objects.

Given these issues, it was necessary to produce custom deserializers for the following objects: Monsters, Ammunitions, Armors, and Weapons. Each of these had to be created from scratch with only help from simple examples from other JSONS. Rogelio initiated in the creation of the custom monster deserializer. Once a greater understanding of the Jackson JSON Library was achieved Rogelio aided Ricardo in understanding custom deserializers for the items: Ammunitions, Armor, Weapons.

GUI Implementation

The first task for the GUI was to create a scene that would implement the D&D calculator simulator. Oscar was tasked with researching and making the GUI for the project. The stage, or scene, is initialized in the Main.java while where fxml loaded and the intro song is triggered. Labels make up the title for the opening scene, and several gifs, images, and an intro song set the tone for the simulator. The early 90’s graphics and music was an artistic preference by Oscar. The HomeController is shown in Figure 6. The actual scene that is rendered can be found in Figure 7. The scene shown after successfully running the simulation is in Figure 8.

The drop down menus are initialized in the scene’s respective controller and global variables are prepared to hold any data for nested items and monster stat scenes to access. Methods in the class are used to handle any events that occur with the items or stats buttons.

The “items” buttons calls on a new scene to open with its own drop down menus and text boxes to assign items for its respective player to equip. The “stats” button pulls the monster name from the monster drop down menu and pulls the information for the monster from the Json file and displays on a new scene. In figures 8 and 9, you can see the stats and item scene renderings.

The “Fight” button triggers the simulations to start, and any text fields and drop menus that are filled are pulled to be read and implemented. A system alert window pops up with the calculated results. The scene shown after successfully running the simulation is in Figure 10.

Bugs and Missing Features

In the project’s current state, there are multiple missing features as well as the presence of technical bugs. One of the primary ones is that Monster multi-attacks are not implemented. The reason is that information within the monster JSON file is structurally complex with the use of english language vs a key : value pair. An example of this is the sentence “The Umber Hulk makes two attacks with its claws, and one attack with its mandibles every turn.” This calls for a cleaner implementation of JSON parsing, as well as translation of an english language command into statistics for the monster.

Players do not have all abilities from levels 1-20. The player’s abilities should also be modified to take into consideration the level. For instance, the Rogue’s Sneak Attack rolls more dice when the Rogue’s level is higher. While this functionality is currently present, it must be made possible for other abilities.

Lastly, ammunition is not taken into consideration during battle. This means that the

player can use their ranged weapon, if equipped, infinitely. In order to implement finite ammunition in combat, the ammunition and weapon objects must be taken into consideration during combat. The ammunition must be compatible with the weapon, and a counter should be used to keep track of the amount of ammunition the player has. However, it is unclear as to what the player will resort to once the ammunition supply has depleted. Multiple items of the same category is not yet implemented, so the player does not have a fallback weapon to resort to. This is a problem that should be solved in the future.

Lessons Learned: Challenges to Overcome

Pre-Production planning was something that should have been thoroughly discussed prior to development of the project. During development it seemed as though we were creating solutions as the problems were encountered. These on-the-spot solutions were not ideal as they were oftenly re-worked as development continued. Before beginning the development of the project, in hindsight, we really should have spent more time in designing and planning the system. An example of this would be the implementation of items and the method of connecting them to the players. Ricardo was able to accomplish this, but a more elegant and efficient solution could have been applied if there was proper pre-planning.

Another issue in our execution was that some technologies that were not being explained in detail with the rest of the team. The best example of this was with JavaFX, Oscar was the only member of the team that was developing with this tool. Given this hypothetical example, if Oscar were to be unavailable the rest of the team would hastily search for resources to continue work with JavaFx. To overcome this issue, all members should be familiar with all major technologies.

The member working extensively with one technology should mentor the other unfamiliar members.

The scope of the problem should also be kept in mind throughout the entirety of the development. There were times where the Minimum Viable Product (MVP) was not being considered during development unfortunately. This issue was related to the project leader not considering the needs of an everyday user vs showing the simulation can be done. This would be rectified by starting with the needs of the user first, vs creating a solution that is difficult to use and adapt it to the needs of the user.

Lastly, well documented Github commits were not implemented until the final stages of development. If prior commits were to have been as detailed as the recent ones, then tasks such as merging of branches would have lessened the risk of leaving important changes unmerged. With good documentation, a general understanding of the purpose of the commit is also attained by the members of the team. This is especially helpful in giving a simplified explanation where changes made with a technology where others are not familiar with. Since two-thirds of the team were not familiar with JavaFX, commits done changing JavaFX should be detailed as much as possible to quickly understand what the changes do.

Lessons Learned: Successes to Continue

Our weekly code reviews helped keep the team stay on track. These reviews were done in person, in campus, preferably with a projector. We used this time to merge branches to master whenever tasks were completed. This time was also helpful to mentor others in using certain technologies, such as when Rogelio explained the Jackson JSON Library to Ricardo.

The development standards put into place early into the development of the project proved to be extremely helpful. All members were able to effectively use most of the major development tools such as IntelliJ, IntelliJ version control, and Github.

The ability to use multiple branches with Github allowed us to work freely with our assigned tasks without risking the stability of the master branch. Also, whenever possible we preferred meeting in person before merging branches. This allowed for the author to point out the changes that should be merged into the master branch.

By using Github's Issue Tracker, we were able to assign tasks to members of the team. This was useful in knowing which member of the team was working on what part of the project. As issues were completed, the member responsible for the task closed the issue ticket. Further into the development, we used the Issue Tracker as a way to gain a general understanding of the state of our project. It allowed us to quickly know what functionality the simulator had and which functionalities were missing. Also, the Issue Tracker was used for any member of the team to report any bugs encountered while executing or reviewing the code. Depending on the area where the bug was encountered, the responsible team member would be assigned to fixing or revising the malfunctioning code.

Basic coding standards that were put into place also helped minimize the amount of time spent when merging. These coding standards were not set in place until halfway into the development. Simply due to the fact that it was not necessary until the merging of multiple branches was necessary. During merging, extra time was spent deciding if changed lines were either functional or simply aesthetic. For instance, decision statements that were single line would either be braced or unbraced. In order to mitigate this, we dedicated some time during one

of our weekly reviews to decide on code standards. Once established, Ricardo went and refactored the code to reflect our code standard. Once completed, the changes were committed and pushed to the master branch. The end result was what we had expected, there was far less changed lines of code to sift through when merging branches.

Lastly, by setting standard references for use of implementing the combat rules of Dungeons & Dragons we were able to depend less on others when questions arose. This was especially true for Ricardo who has never played Dungeons & Dragons before. Simple knowledge such as what critical hits entailed were easily answered by reading the standard reference. Another benefit to keeping standard references was that we were able to stay true to the rules in Dungeons & Dragons 5th edition.

The Future

Oscar will be continuing this project for his Senior Design II class for the Fall Semester. Some goals will include fixing the bugs and features mentioned earlier as well as implementing more game mechanics. Such game mechanics will include: range attacks, spells, abilities, and player/monster movement. This should increase the accuracy of the simulator. Adding a customizable character function into the project can also provide multiple uses such as assisting the Dungeon Master with more relative simulation data, and as an easy tool to generate a customized player.

Another goal will be to see the possibility of implementing a database into the project rather than using JSON files to rely on for item and monster implementation. This may be useful for player, spells, abilities, etc. as well.

The Longer Future

In the longer future, instead of using an executable jar, we would like to see this program as a web application in order to cover software compatibility issues. Another addition, would be the possibility of including D&D expansions into the simulator. A final addition would be to play a game based on the D&D game mechanics implemented in the simulator.

Conclusion

The current version of the Dungeons and Dragons Encounter Calculator is marginally useful, more so at the lower levels. When a few improvements are implemented in the next semester, the utility will drastically increase for dungeon masters. This experience has shown the need of preplanning and the need to begin with a minimum viable product. The development cycle should have begun with a GUI prototype and program in the functionality as the team would advance. This was an essential learning experience for our team and let us experience a small part of what is needed for software development.

Figures

Creature		
f	diceNum	String[]
f	damConst	Integer
f	roll	Integer
f	sneakAttack	boolean
f	challengeRating	String
m	compareTo(Creature)	int
m	hpPercent()	Integer
m	generateInitiative()	void
m	attack()	Integer
m	sneakAttackDamage()	void
m	singleCombat(Creature)	void
m	spellCombat(Creature, Integer)	void
m	attackDamage()	Integer
m	receiveDamage(Integer)	void
m	recieveHealing(Integer)	void
m	setDamageDice()	void
m	generateDamage(String[])	Integer
m	setAttributes(Integer, Integer, Integer, Integer, Integer, Integer)	void
m	chooseAction(ArrayList<Creature>)	void
p	con	Integer
p	weap	String
p	wisMod	Integer
p	level	Integer
p	dexMod	Integer
p	init	Integer
p	prof	Integer
p	str	Integer
p	conMod	Integer
p	maxHp	Integer
p	intelMod	Integer
p	chaMod	Integer
p	ac	Integer
p	strMod	Integer
p	alive	boolean
p	intel	Integer
p	wis	Integer
p	hp	Integer
p	dex	Integer
p	cha	Integer

Powered by yFiles

Figure 1

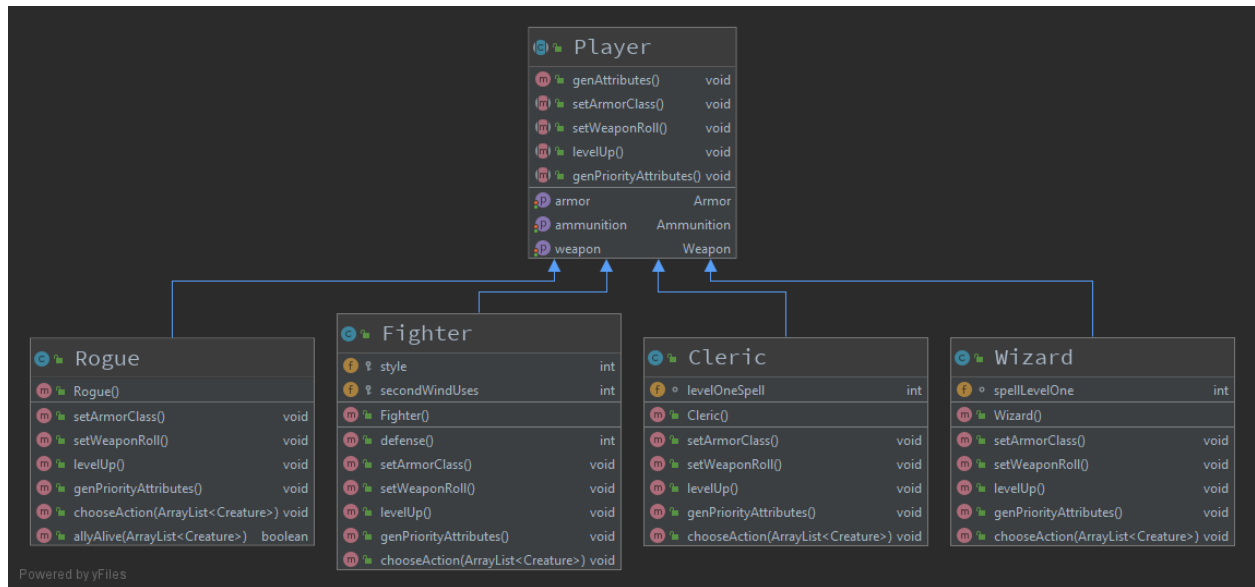


Figure 2

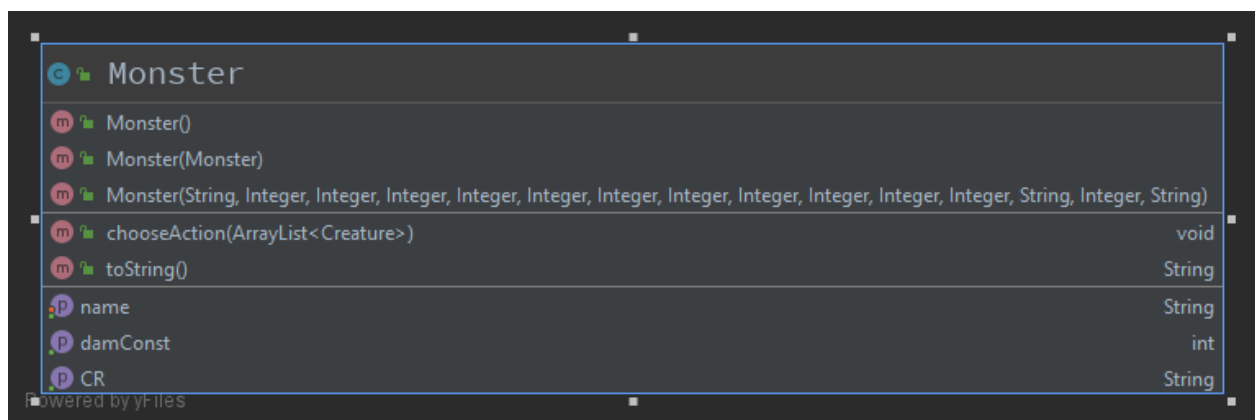
































Figure 3

Commands		
 	rollX(int)	int
 	roll10()	int
 	roll20()	int
 	selectClass(String)	int[]
 	disadvantage()	int
 	advantage()	int
 	rollHP(int, int)	int
 	rollHP(int, int, int)	int
 	hpPercent(Creature)	int
 	genAttribute()	int
 	poolAttribute()	ArrayList<Integer>
 	lowestMonHP(ArrayList<Creature>)	Monster
 	lowestPlayHP(ArrayList<Creature>)	Player
 	highestMonHP(ArrayList<Creature>)	Monster
 	isNumeric(String)	boolean

Powered by yFiles

Figure 4

Sim		
f	combatArrayList	ArrayList< Creature>
f	partySize	int
f	fighterArray	int[]
f	rogueArray	int[]
f	clericArray	int[]
f	wizardArray	int[]
f	goblinSize	int
f	orcSize	int
f	bugbearSize	int
f	simIterations	int
f	winNum	int
f	winRate	double
f	monstersArray	ArrayList< Monster>
m	Sim(int, ArrayList< Creature>)	
m	Sim(int[], int[], int[], int[], ArrayList< Monster>, int)	
m	newSimulation()	void
m	newEncounter()	void
m	newParty(int[], int[], int[], int[])	void
m	checkGroupAlive(ArrayList< Creature>)	boolean
m	setCombat()	void
m	combat(ArrayList< Creature>, ArrayList< Creature>)	void
m	round()	void
m	calcWinRate(int, int)	void
m	whoWon()	void
m	simulation()	void
m	simulationWinRate()	double

Powered by yFiles

Figure 5

HomeController	
m	initialize() void
m	showAlertWithoutHeaderText() void
m	fightButtonClicked() void
m	itemButtonClicked() void
m	itemButton1Clicked() void
m	itemButton2Clicked() void
m	itemButton3Clicked() void
m	statsButtonClicked() void
m	statsButton1Clicked() void
m	statsButton2Clicked() void
m	statsButton3Clicked() void
m	getMonsterList() void
m	addPlayer(ChoiceBox, TextField, TextField, Integer) void
m	checkPlayerBoxValid(ChoiceBox, TextField, TextField) boolean
m	addMonster(ChoiceBox, TextField) void
m	checkMonsterBoxValid(ChoiceBox, TextField) boolean
m	addPlayerToCombatList(String, Integer, Integer, Integer) void
m	setPlayerItems(Player, Integer) Player
m	selectClass(String) int[]
m	playSound(String) void
m	secretButtonClicked() void
p	armorMod String
p	weaponMod1 String
p	ammunitionCount2 String
p	armor Armor
p	armor3 Armor
p	ammunitionCount3 String
p	ammunitionCount String
p	weaponMod3 String
p	weaponMod2 String
p	weaponMod String
p	armor2 Armor
p	armor1 Armor
p	ammunitionCount1 String
p	armorMod3 String
p	armorMod2 String
p	armorMod1 String
p	ammunition2 Ammunition
p	ammunition1 Ammunition
p	weapon Weapon
p	ammunition Ammunition
p	weapon3 Weapon
p	ammunition3 Ammunition
p	weapon2 Weapon
p	weapon1 Weapon

Figure 6

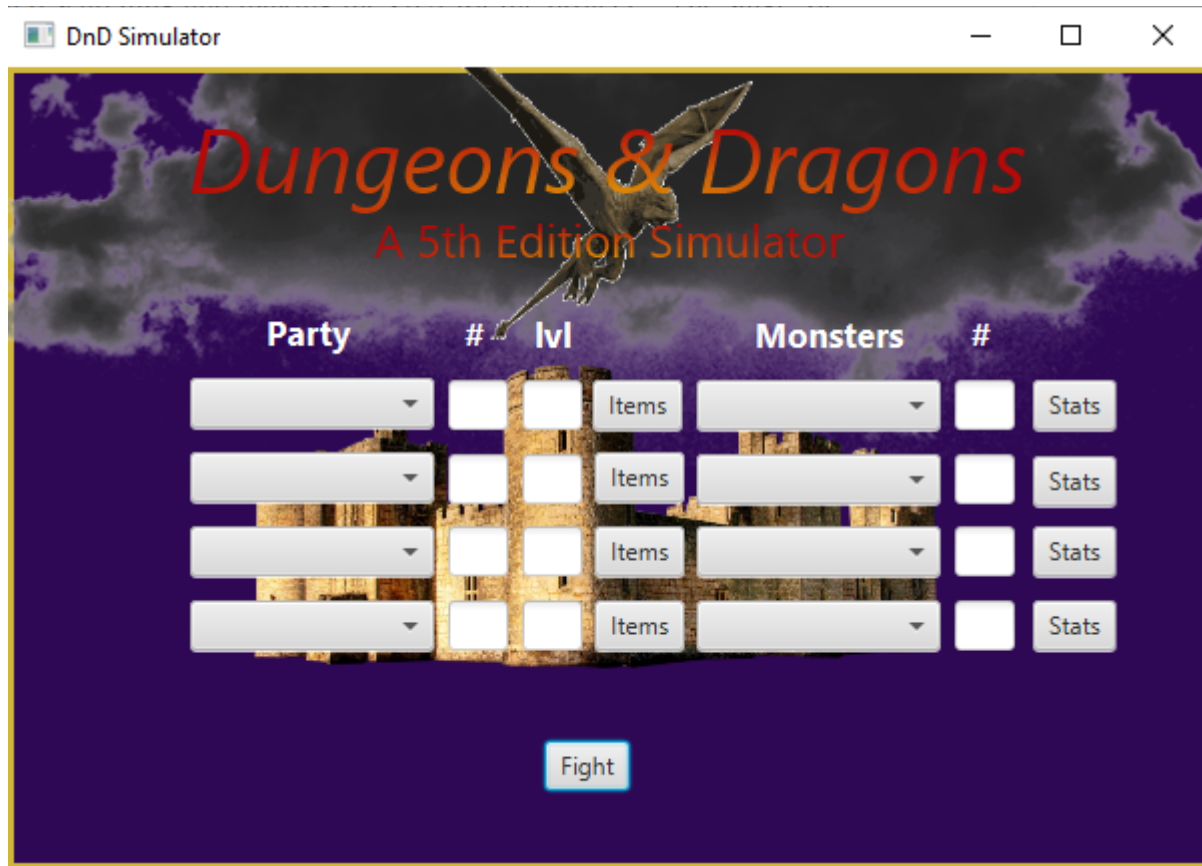


Figure 7

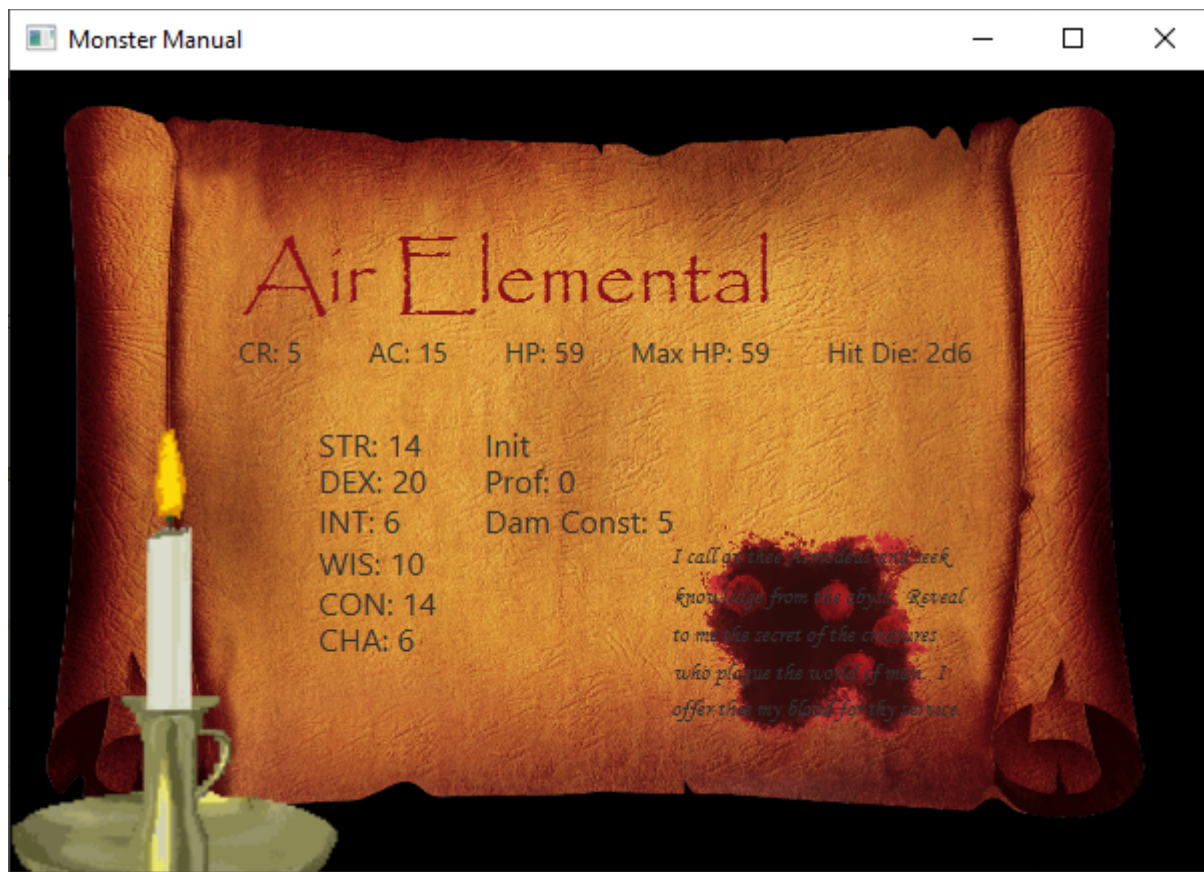


Figure 8

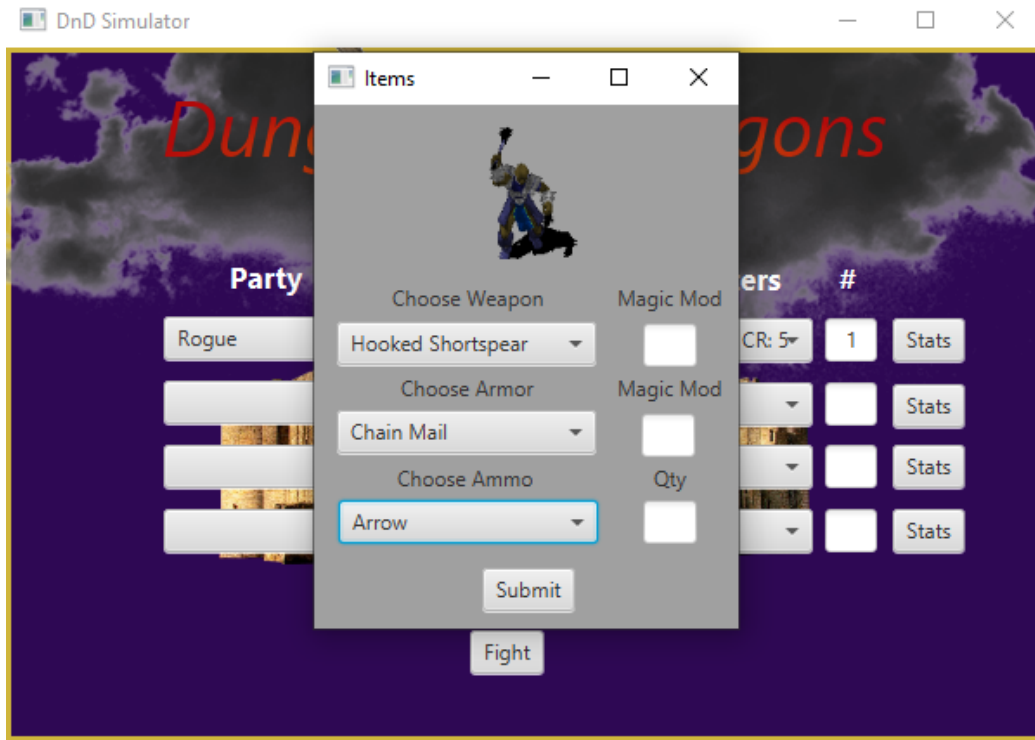


Figure 9

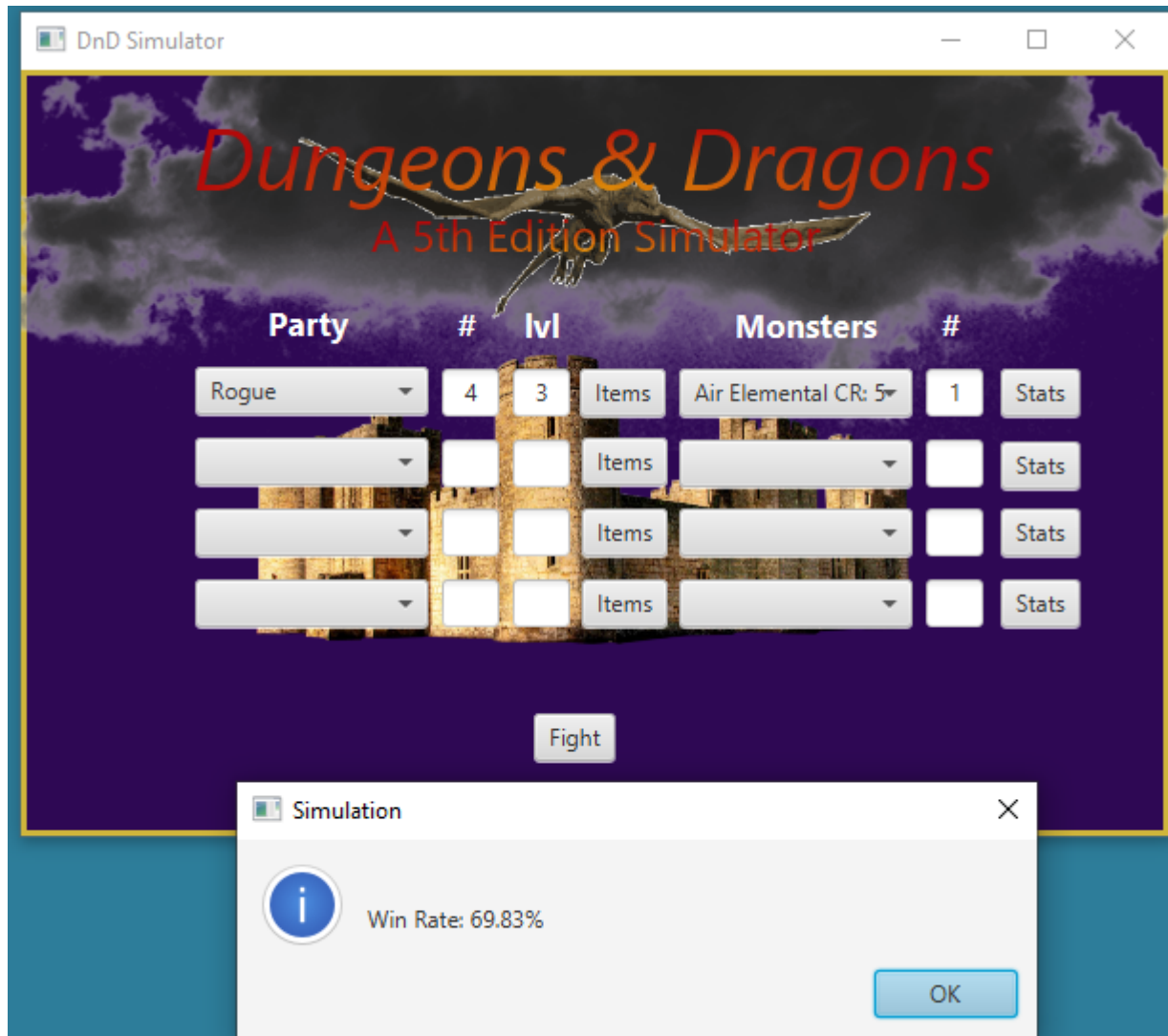


Figure 10

Resources

- IntelliJ: <https://www.jetbrains.com/idea/download/>
- Java: <https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html>
- JavaFX: <https://gluonhq.com/products/javafx/>
- Scene Builder: <https://gluonhq.com/products/scene-builder/>
- JSON Sources: <https://github.com/TheGiddyLimit/TheGiddyLimit.github.io>
- Dungeons and Dragons Rules: <https://www.dndbeyond.com/>
- JavaFx: <https://docs.oracle.com/javafx/2/>
- Art Assets: Intro Music by: Isabella Estrella
- Audio: <https://www.zapsplat.com>
 - <http://www.findsounds.com/ISAPI/search.dll?keywords=roar>
- Gifs: <https://wifflegif.com/gifs/602157-animation-reference-flight-cycle-gif>
 - <https://www.fg-a.com/medieval-clipart-2.shtml>
 - <http://www.animatedimages.org/cat-candles-88.htm>
- Images:
 - <https://www.kisspng.com/>