Design

Structure / Flow Charts

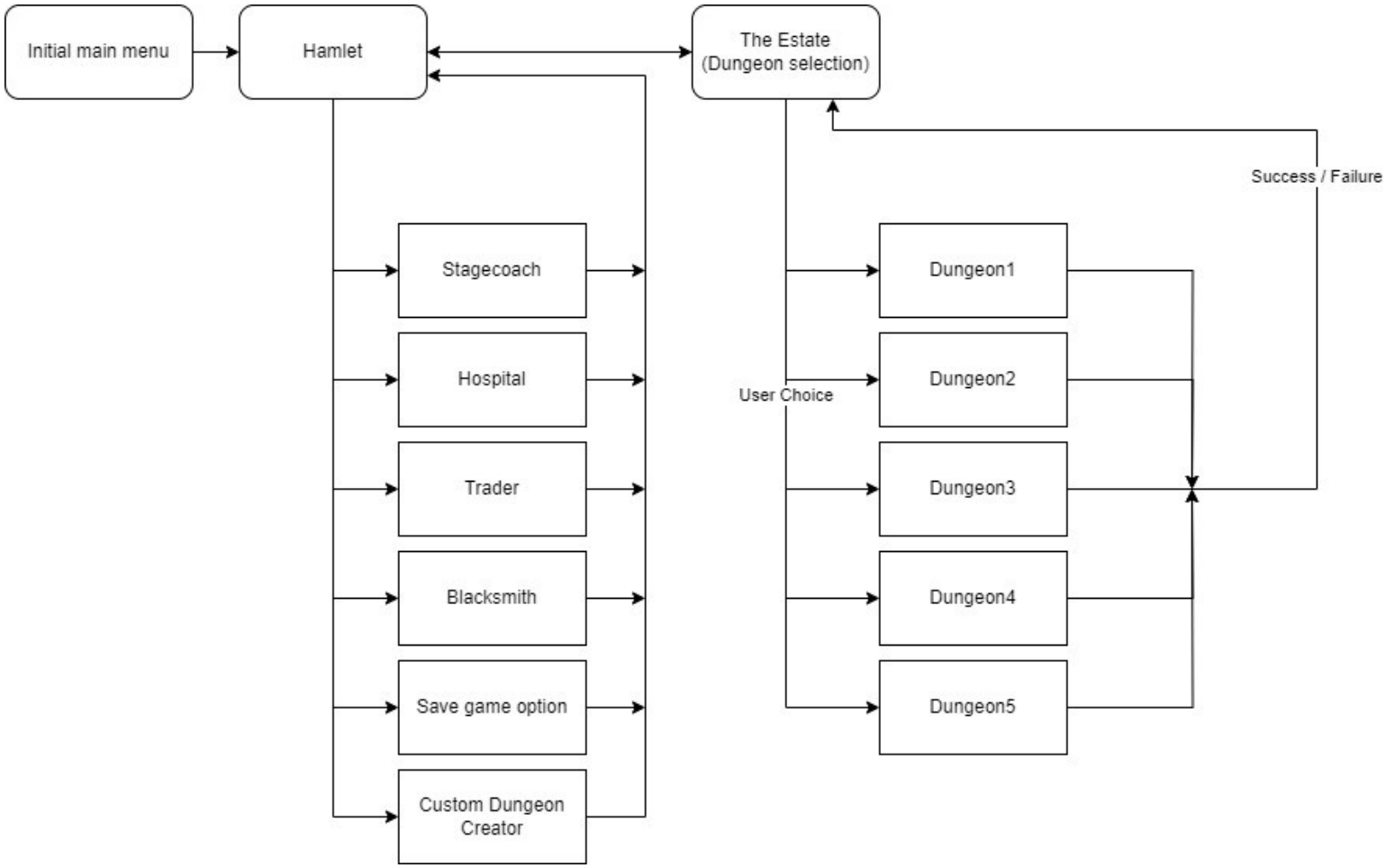


Figure 11: High-level menu navigation flow chart

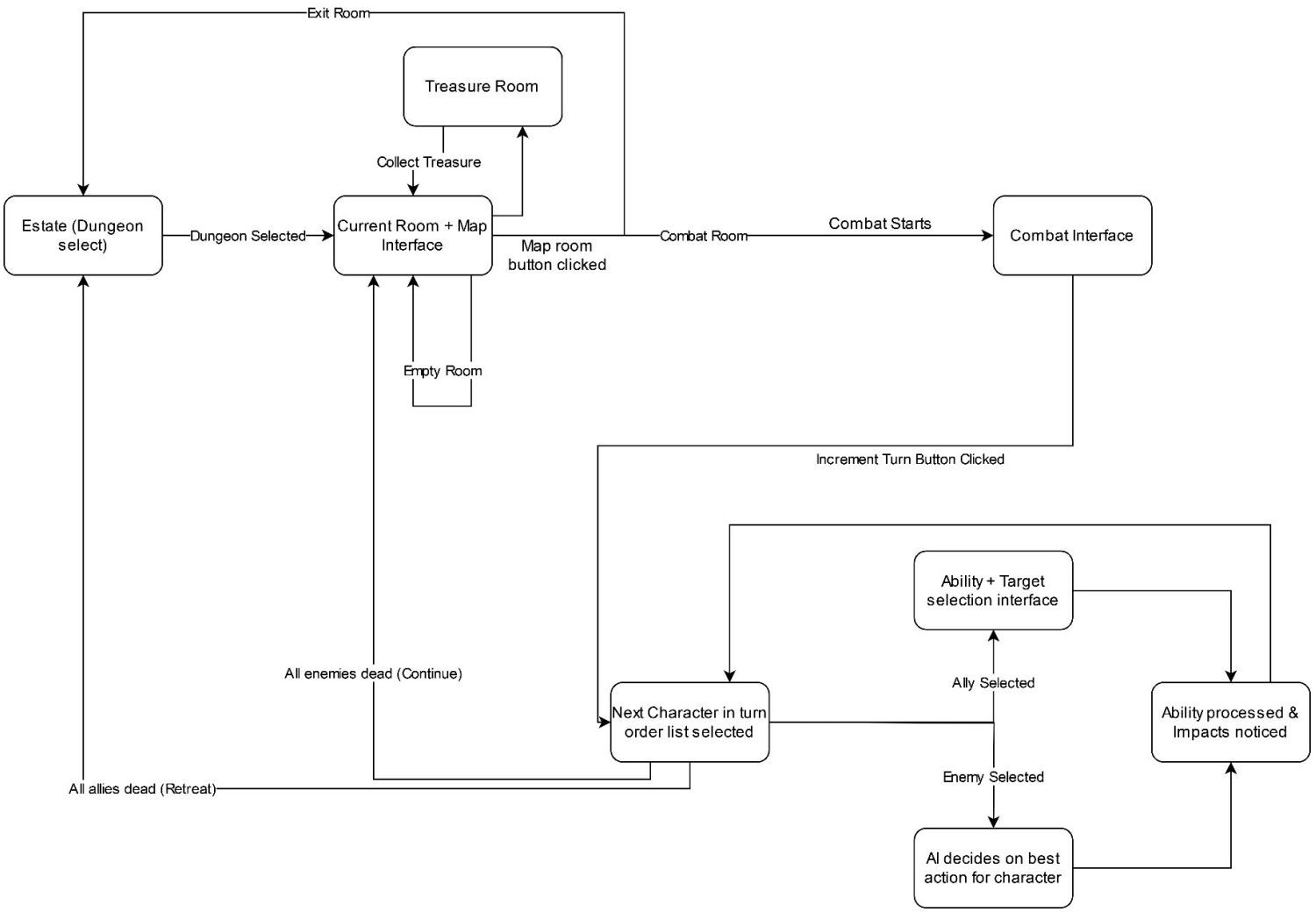


Figure 12: Dungeon navigation & Combat diagram

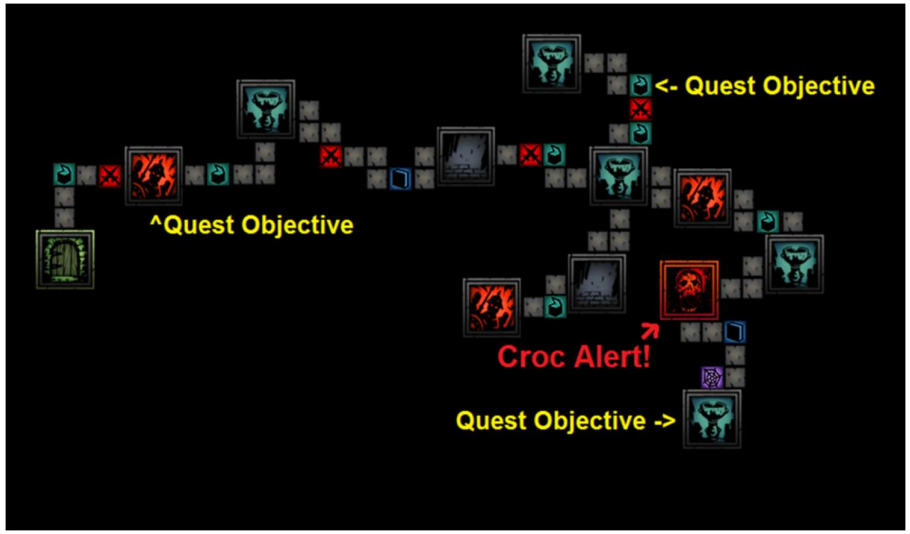
Issues / Bugs / Prototypes

Map Generation & Layout

Attempt 1

At the start of development I chose to begin by creating the dungeon map generation system, however before I could even do that I needed to establish how exactly a dungeon was going to be displayed to the player, and what the structure of the rooms was going to be.

Initially, I was going to try to replicate the map from Darkest Dungeon I (Existing system 1) as closely as possible.



When trying to replicate this map, I reviewed the individual aspects of it that I would need to create and then connect together:

- Rooms are randomly located with no fixed grid system.
- Corridors are randomly generated between the rooms but never intersect.
- Not all rooms link to each other.

After reviewing these requirements, I realised that to generate random maps of this nature it would be extremely complex, for a relatively minimal amount of reward.

I decided to research into how exactly the game generated these maps, and to see if there was anything I was missing that would allow me to replicate something similar more easily, however I ended up finding out that the maps in darkest dungeon 1 are actually all pre-made (at least in terms of structure).

This reaffirmed my current mindset at the time that generating maps that end up looking like that randomly was too challenging. However I knew I wanted random map generation in my game, so I tried coming up with other solutions.

Attempt 2

The next prototype I tried out was done in an attempt to generate seemingly randomly laid out maps, but in reality utilise a fixed structure behind the scenes.

```
enum DungeonWeighting
{
    Front = 1,
    Middle = 2,
    Back = 3,
}
```

This prototype would make use of the above enum type, which would determine whether or not a dungeon room would be in the front, middle, or back of a dungeon. Each of these positions would be on a vertical line on a form, which is where the picture for the room would be displayed.

This prototype didn't last long, I soon realised that this would result in every dungeon having an extremely "left to right" feel, and being far too linear. You would also know exactly where you are trying to get to in order to be able to exit the dungeon. This was not an effect I wanted the layout of a dungeon to have, and even though this method may look nice, randomness and uncertainty of where you are trying to reach was not something I was willing to sacrifice.

Additionally, this method did not solve the issue of connecting the rooms without corridors colliding, which was another significant issue I was facing.

Attempt 3 (Final)

After experiencing so many issues with the corridor aspect of the dungeons specifically, I decided not to go about trying to implement them, and thought that removing them would only have a minor negative impact on the user experience.

```
public enum RoomContent
{
    nulled = 0,
    Empty = 1,
    EasyEnemy = 2,
    HardEnemy = 3,
    Reward = 4,
    Exit = 5,
}
```

I settled on utilising a 5x5 2d array of integers to represent a map, which would then be converted into a 5x5 grid of picture boxes on a form to be displayed to the player. The generation algorithm would randomly create the room by checking adjacent rooms to existing rooms, starting from the middle, and randomly choosing one of them to turn into a new dungeon room with different integers representing different types of room, this process would then repeat until the dungeon was complete. This results in a large amount of variation from dungeon to dungeon, which was exactly what I was looking for.

Additionally, while at the time I did not realise it yet, utilising an integer 2d array to represent the maps before converting them into something the player can actually use would make my life much easier when it came to saving dungeons and being able to parse them in and out, as well as when it came to creating and sharing custom dungeons.

I considered evaluating the time complexity of the algorithm I had created, however it did not seem like a relevant aspect of the solution. It is standard and expected for games to have a loading time, and while I would attempt to reduce this as much as possible, I did not see a point in trying to reduce it by tiny increments. Additionally during testing, the algorithm ended up effectively not being noticeable in terms of time taken to complete and load, so I did not deem trying to make it more efficient as a good use of time as ultimately there would be no difference for the end user.

Relationship between dungeons and their rooms

Simultaneously while developing the dungeon generation algorithm, I began to think about designing the class architecture for Dungeons, and their Rooms.

I knew that I would need a class for Dungeon, and a class for DungeonRoom, however the relationship between the two was something I was not certain about yet.

```
class DungeonRoom
{
    private DungeonRoom[] AdjacenctRooms = new DungeonRoom[3];
}
```

I initially considered storing each dungeon room with an array of the rooms that were adjacent to it (Max 4).

This would then be linked back to the dungeon object of all the rooms by storing the origin (central) room as an attribute.

```
class Dungeon
{
    private DungeonRoom Origin;
}
```

This was when my idea for map generation centered around rooms being laid out with weighting, as described in Attempt 2 above.

I thought this idea would help me when it would come to deciding which rooms a user can navigate to from the room they are currently in, and while I do believe it would have made that aspect slightly easier, it made every other aspect of operating the dungeon class far harder, as the only way to get from a room to another room was going to be through navigating every room in between. While from a player's perspective that would be fine, for other features such as custom dungeon creation it would have introduced a lot of extra challenges. To me, this seemed like a worthwhile trade-off, as I was already calculating adjacent rooms dynamically when needed.

However, the primary reason for not utilising this method was that It would involve being able to know from which direction a room was from another. And then with that, being able to ensure that two rooms do not overlap. As if a room had an adjacent room above it, the room to the north west of that room would have the same room to its east. But due to the layout of the map, that may not always be the case.

With these points in mind, and also the fact I was moving on from the idea of laying out the dungeon's rooms in vertical lines across the form, I decided to move on from this idea.

I would eventually go on to store the dungeon that a dungeonroom belonged to within the dungeonroom object as an attribute, in order to be able to relate back to its container dungeon easily. As well as storing a 2d array of dungeon rooms as an attribute of the dungeon class.

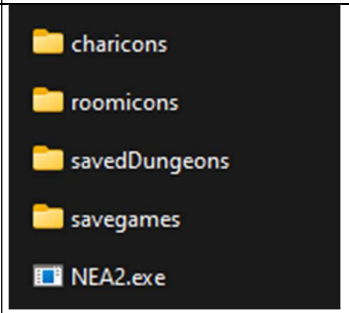
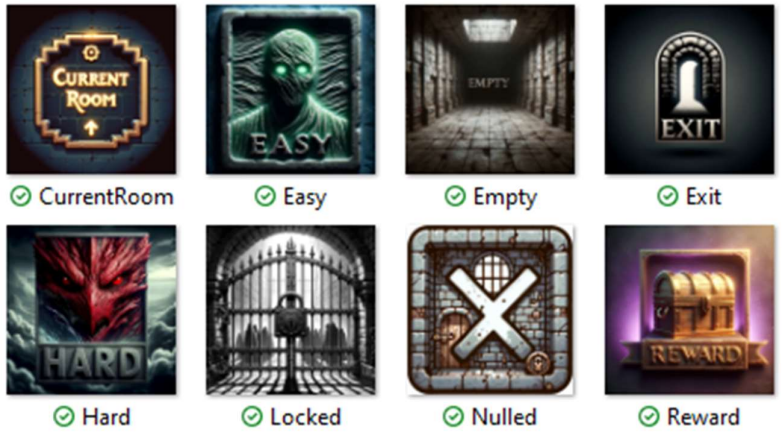
Turn System

Eventually during development, I arrived at the point of needing to design the overarching combat system. At that point, the primary element of it in my mind was how to make the

Techniques Used

Technique	What for	Example from project (Where)
Depth first search /	Used to check whether a custom map created by the user using the map-creation tool is valid. Specifically, whether or not all rooms are connected to each other (Ensuring every room is able to be reached)	Used within the CheckValidGrid() method, which is executed every time the user makes a change to the map in the custom dungeon creation form.
List Operations	Used throughout entire application regularly	
Recursion	Used in depth first search algorithm to create a visited array and passing it throughout multiple recursive calls.	Each call of the DepthFirstSearch() method calls 4 more DepthFirstSearch() methods, each for the nodes adjacent to the one being checked. The base case is when all nodes that can be visited have been visited.
Composition	Demonstrated throughout classes stemming from "Game" All hamlet forms such as stagecoach and doctor are attributes of the hamlet form itself.	All class attributes of Game are a composition relationship. All forms that are attributes of the hamlet form are a composition relationship.
Inheritance	Demonstrated throughout application, notably: - Characters - Abilities - Dungeon room types - RoomInteractions	Different classes (E.g. Barbarian, BloodHunter) of characters inherit from the base Character class. Different types of abilities inherit from the base Ability class

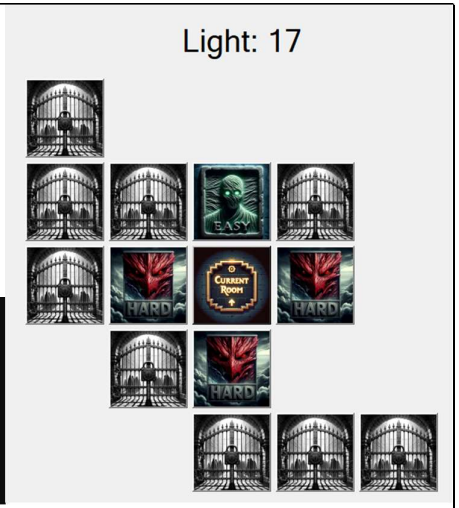
		Each type of dungeon room, such as reward & enemy rooms inherit from the base DungeonRoom class CombatInstance inherits from the RoomInteractions class
Classes	Whole application centers around classes/objects. Most notable classes: - Game - Dungeon - DungeonRoom - RoomInteractions - CombatInteraction - Ability - Character - Utils - CombatTurnHandler - KeyParser	Game class utilised as a container for an instance of a game, allows for a centralised collection of all the data about a game which makes saving easier. Utils is a static class which acts as a container for regularly used methods throughout the program, and miscellaneous methods which do not necessarily fit anywhere else. Examples: - Generate random character name - Generate random dungeon name - Custom method for showing forms - Method to remove temporary controls on a form based on the contents of their tag attribute Keyparser is a class which handles everything to do with converting data to and from keys (A form of hash) CombatTurnHandler is a static class which acts as a container for methods used surrounding all combat within the game. Including methods to: - Process the interaction of an ability use between a source and target - Process special & status effects during a turn. - Calculate whether or not an ability is going to hit - Calculate damage after taking into account armour & status effects such as strength.
Polymorphism	Polymorphism is used primarily within the inheritance of different character classes and different types of abilities.	Each ability overrides the “Execute” method of the Ability class, which opens up the opportunity for custom logic outside of the typical combat interactions to be put in place. For example, the “Divine intervention”

		ability makes the target invincible, due to the way combat interactions were programmed this logic needs to be separated from the ability's standard logic.
Files organised for direct access	Folders for aspects of the game have been organised into 1 directory: - Charicons – Contains all character portrait images. - Roomicons – Contains images for the icons displayed for the different types of room on a dungeon's map. - SavedDungeons - Contains text files which contain the save keys for custom created dungeons. - SaveGames – Contains the save keys generated when saving a game. This folder is then read from to load games.	 Example contents of Roomicons: 
Dictionaries	- Used to relate dungeon room type to map icon - Used to relate status effects to how many turns they last - Used to create an adjacency matrix with the user-defined coordinate class	Present within the grid map parser, which turns the 2d integer array generated by the map generation algorithm into the visual presentation the user sees via the map form. Each integer corresponds to a picturebox, which has a different image based on the room type, which is dependant on the integer value.

		Status effects are stored as a relationship between the number of turns they have left and the type of status effect that there is. This allows for interation through a character's status effect dictionary easily each turn. Adjacency matrix visible within the Dungeon class, being generated within the ProcessDungeonGrid() method.
Text Files / Hashing	- Used to save user generated map keys in order to be able to save them and share them later.	Visible within the KeyParser class, the ToFile method converts the dungeon a user creates within the custom dungeon creator into a saveable, single-line key, which is then stored in a file created with the same name as the dungeon. They key is also copied to the user's clipboard to allow for easy immediate sharing.
Multi-Dimensional Arrays	- Features all throughout code, most notable implementation is for storing map grids, where the integers represent the roomtypes. - Method "ProcessDungeonGrid" is re-used in order to load custom maps from keys.	Shown as an attribute of the dungeon class. - Represents the integer 2d array form of the map - Represents the 2d array of dungeon room objects form of the map Used to represent the grid buttons/picture boxes on the map form.

Description of main data structures and justification of their choices

Data Structure	Where & Justification

Arrays	Arrays are used throughout the application, bot one dimensional and two dimensional. 2 dimensional: - Int[,] used to represent dungeon map in number form - DungeonRoom[,] used to represent the same dungeon map, but where numbers are replaced by room instances. - Button[,] used to store the 5x5 array of Buttons on a dungeon's map form, where each button represents 1 room. On the example to the right, the map and the corresponding number grid representation is shown. Different numbers determine different types of rooms. The player begins in the middle, and the only rooms visible to the player are the ones they are adjacent to. 5 0 0 0 0 2 3 2 3 0 3 3 1 3 0 0 2 3 0 0 0 0 3 4 2  A 2d array was selected in these instances due to the fact that: - With each of these different representations of the map being represented by the same data structure, mapping between them and being able to translate from one to another is made much easier. - A map would always be the same size, with the same dimensions. Meaning a static data structure is appropriate and ideal. 1 dimensional: - Character[] used throughout rooms and room interactions. - Ability[] used to represent a character's abilities. - Type[] used to represent the different character types that an ability can belong to. A 1d array was selected in these instances due to the fact that: - The amount of data needing to be stored is either defined at runtime and never changed. - The amount of data being stored is guaranteed to be the same every time, as characters must all have 5 abilities and all combat rooms must have 4 allies and 4 enemies.
Lists	Lists are used throughout the application in any instance where multiple data points need to be stored together but with a potential variation in the size.

	The most notable example of this is a Character list, which is used to represent the order in which character take their turn during a combat instance. This data structure needs to be dynamic, as while it will always begin with 8 characters, as characters die they are removed from the list (as they can no longer execute an action during a turn)

Permanent Data Storage

Throughout my development I knew that developing the save system would ultimately come last. This was due to the fact that saving would have to rely on every other aspect of the game being completed, to ensure that the system wasn't developed, and then something extra that needed saving was added, as I believed that would cause problems.

When it came to beginning to develop the save system itself, I began to compare between JSON and XML as they are what I thought I would be deciding between. Upon further review of these two methods, they seemed ideal for saving individual instances of classes, however due to the structure of my classes, specifically the fact that there were many layers of classes having attributes that were other classes. This made utilizing these systems seem overly complex. Additionally, only small aspects of data from each class was actually required to re-construct it.

With that in mind, I made the decision to create my own form of storage. This would include:

- Translating data into a storable format
- Parsing stored data back into

The first stage of this process consisted of deciding on which data specifically needed to be stored, in order to be able to reconstruct the state of a game perfectly.

I concluded that the objects I would need to reconstruct were:

- Characters in party
- Characters owned by the player
- Dungeons
- Experience
- Gold

I then went through the classes for these objects, and identified the specific attributes that I would need to store to be able to reconstruct them.

- Characters
    - Name

    - Class
    - Level
    - Health
    - Max Health
    - Mana
    - Max Mana
    - Crit Chance
    - Dodge Chance
    - Armour
    - Speed
    - Status Effects
    - Abilities
        - Name
- Dungeons
    - Name
    - Difficulty
    - Map (As 2d integer array)
    - Attempted (Bool)
- Gold
- Experience

Following this, I introduced methods into each class that would generate that object's save key, by creating a formatted string with all of the data listed above. These strings would then be sectioned out in a file, similar to JSON.

Usefully, as I had already developed a method for players to create and share custom dungeons, as well as create a parser for the share key, in order to save a dungeon I simply converted it into the same format as a shared dungeon key with extra data added onto it. This allowed me to re-use a large amount of existing code for the parsing and saving of dungeons, which helped save time and keep the program efficient.

```
Gold:989344

Experience:1000

Party:
Rhys|Druid|4|Hero|213|213|95|95|0|0|15|20|Earthquake|Frostbolt|Heal|Fireball|Punch|
Brodie|Bloodhunter|4|Hero|246|246|38|38|0|0.11|16|4|Kick|BloodNova|DrainLife|BloodSuck|Punch|
Charlie|HighwayMan|4|Hero|131|131|38|38|0.165|0.11|9|9|Kick|Shoot|Backstab|PoisonArrow|Punch|
Dwayne|Paladin|4|Hero|328|328|51|51|0|0|23|3|Kick|DivineShield|Heal|Smite|Punch|

OwnedChars:
Nelson|HighwayMan|1|Hero|92|92|32|32|0.0525|0.035|2|7|Backstab|Shoot|PoisonArrow|PerfectShot|Punch|
Ellis|Paladin|1|Hero|234|234|43|43|0|0|10|2|Heal|Smite|SwordSlash|ThunderStrike|Punch|
Omari|Paladin|1|Hero|236|236|45|45|0|0|6|2|DivineShield|Heal|SwordSlash|Smite|Punch|

Dungeons:
False|The Lower Tombs|0.5|0:2,5:1,0:4,3:1,1:2,0:2,1:1,4:1,0:11
False|Dungeon of the Blooded Elf|0.8|0:1,5:1,0:4,4:1,0:1,1:1,0:2,4:1,1:1,2:2,0:2,3:1,0:7
False|Crypt of the Raging Queen|1|0:1,1:1,0:1,3:1,0:2,4:1,3:1,1:1,0:2,4:1,1:1,2:1,5:1,0:2,3:1,0:7
False|The Death Talon Quarters|1.2|0:1,1:1,0:1,5:1,3:1,0:1,4:1,3:1,1:1,0:2,4:1,1:1,2:2,0:2,3:1,0:1,2:1,0:5
False|The Iron Vault|1.5|0:1,5:1,0:1,3:1,0:2,3:2,4:1,3:1,0:1,4:1,1:2,4:1,0:1,3:1,2:1,0:2,3:1,2:1,0:3
```

Figure 13 shows an example of a save file.

I then wrote the parsers to effectively reverse the key conversion methods and reconstructed a game object from that data.

Save files are accessed by typing the name of the save file into a modal that pops up after clicking “Load Game” from the main menu. – When creating a save file, the user is prompted to enter a name for the save file. Additionally, if a user creates a save file and attempts to save again, it will not prompt them to enter the name again as each game object has a “savepath” attribute stored within it, to know where to save to.

UML diagrams

Key:

	Private
	Public
	Protected
	Constant
	Inheritance (Left inherits right)

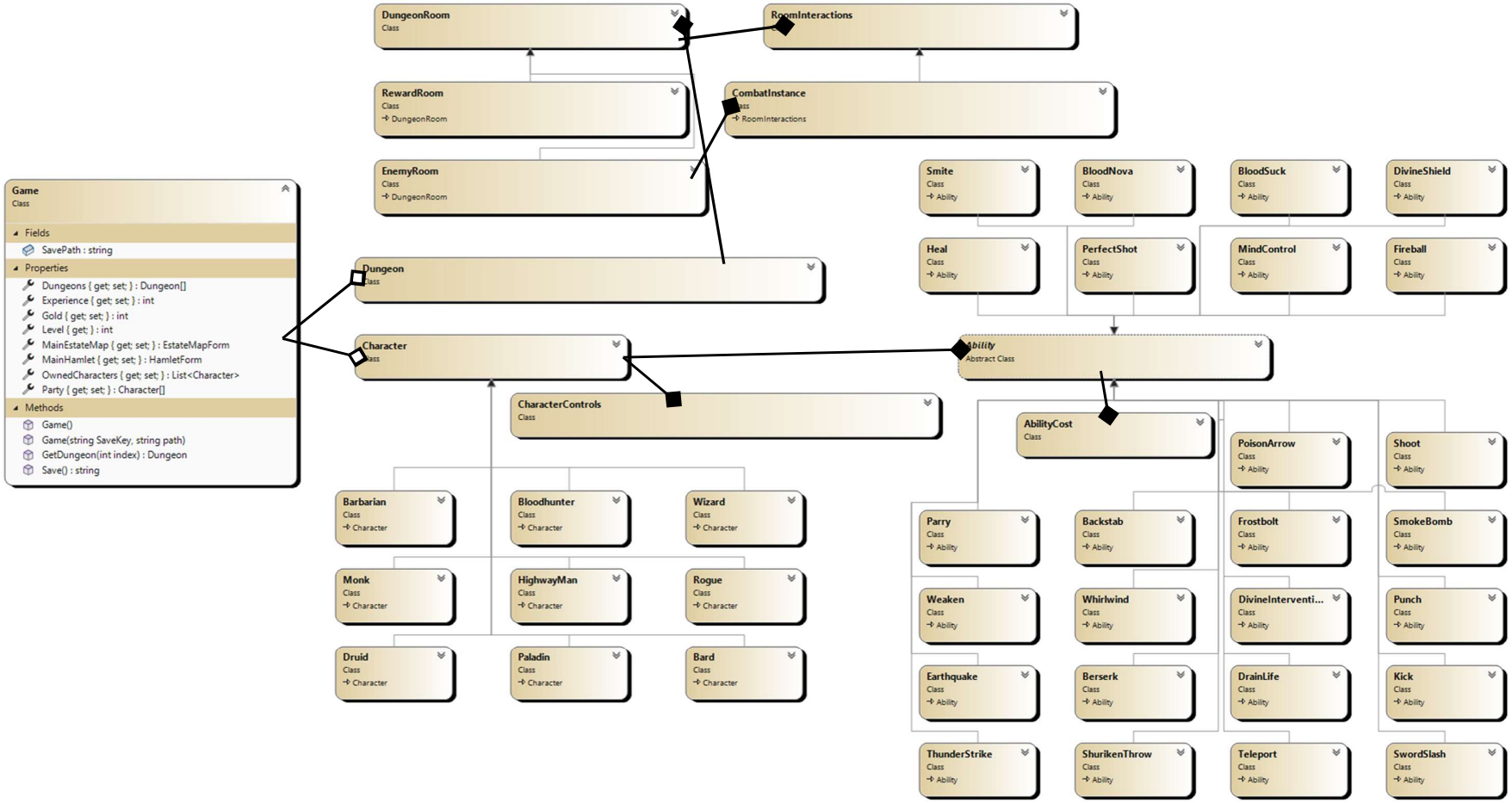
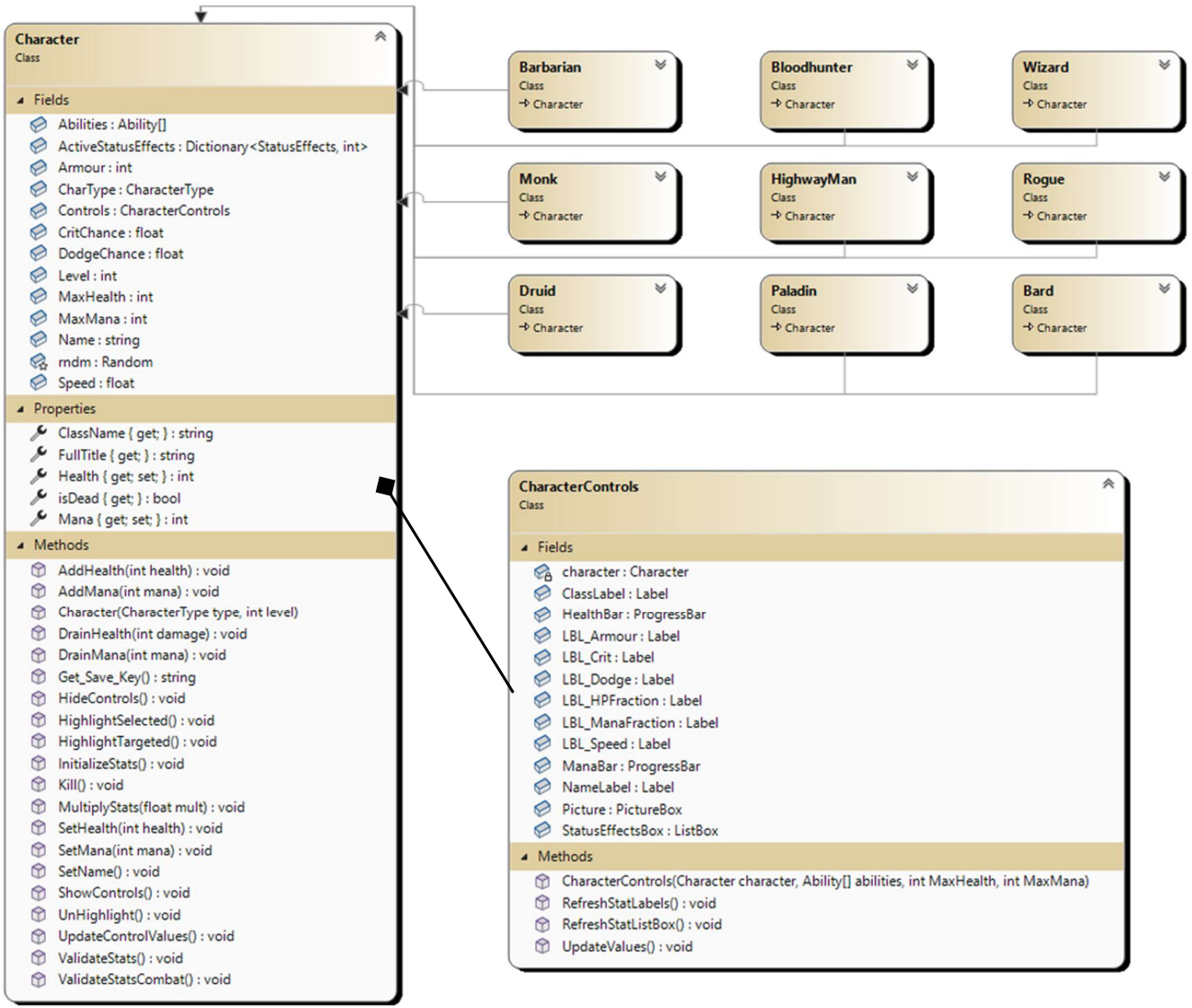
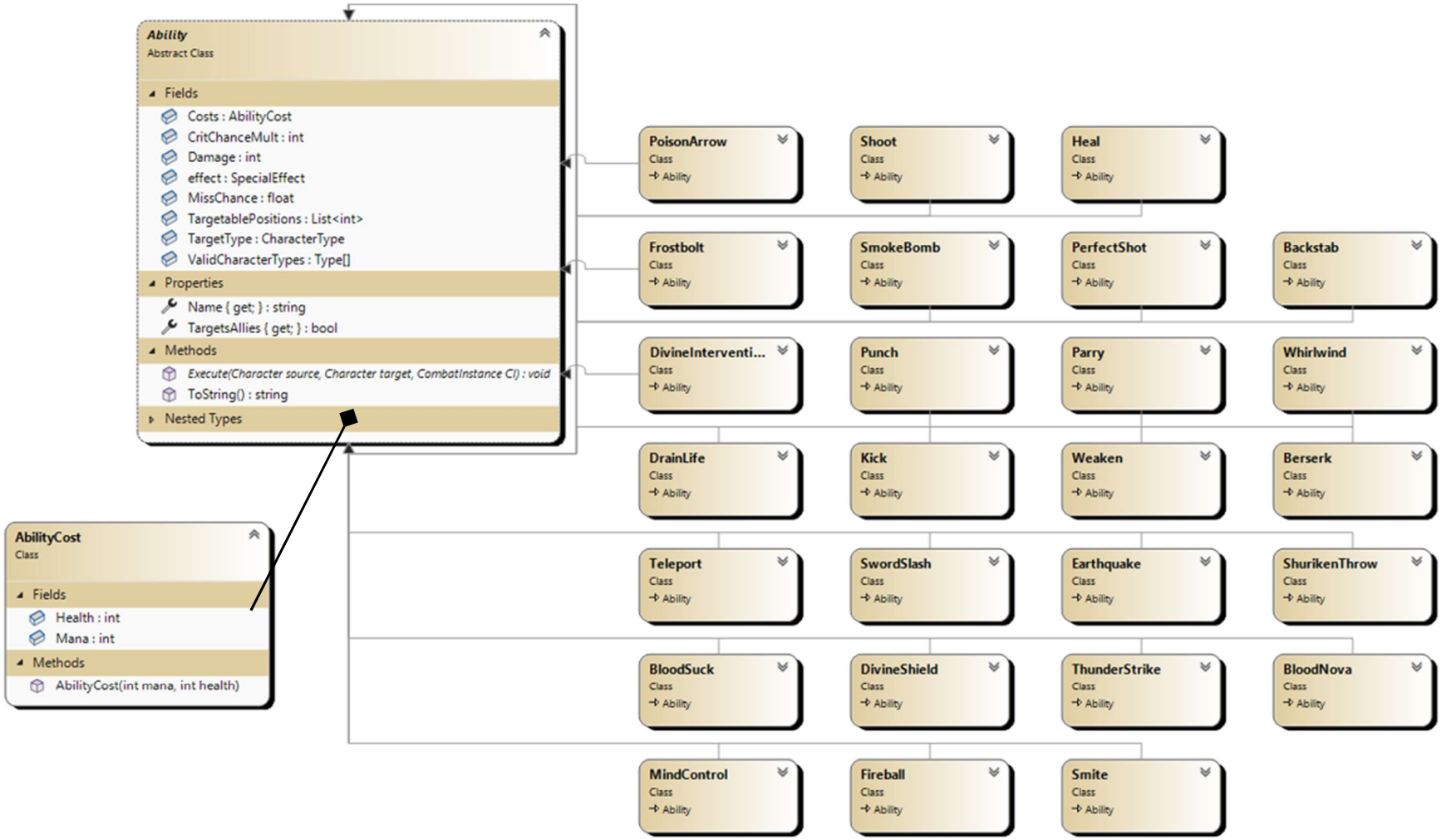
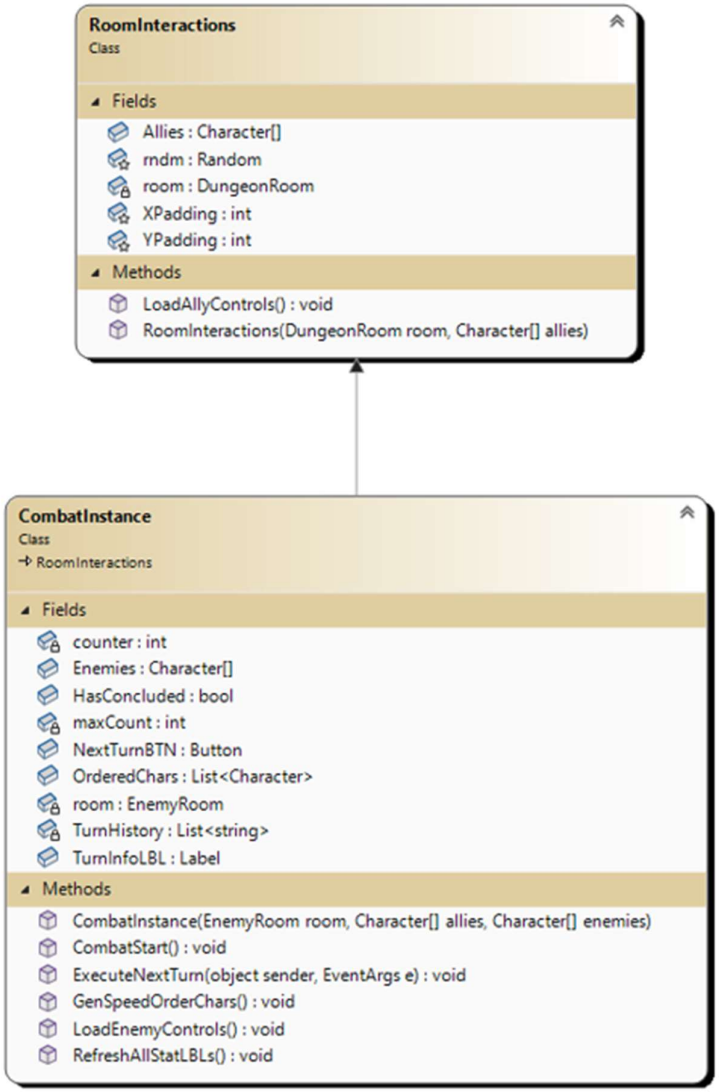
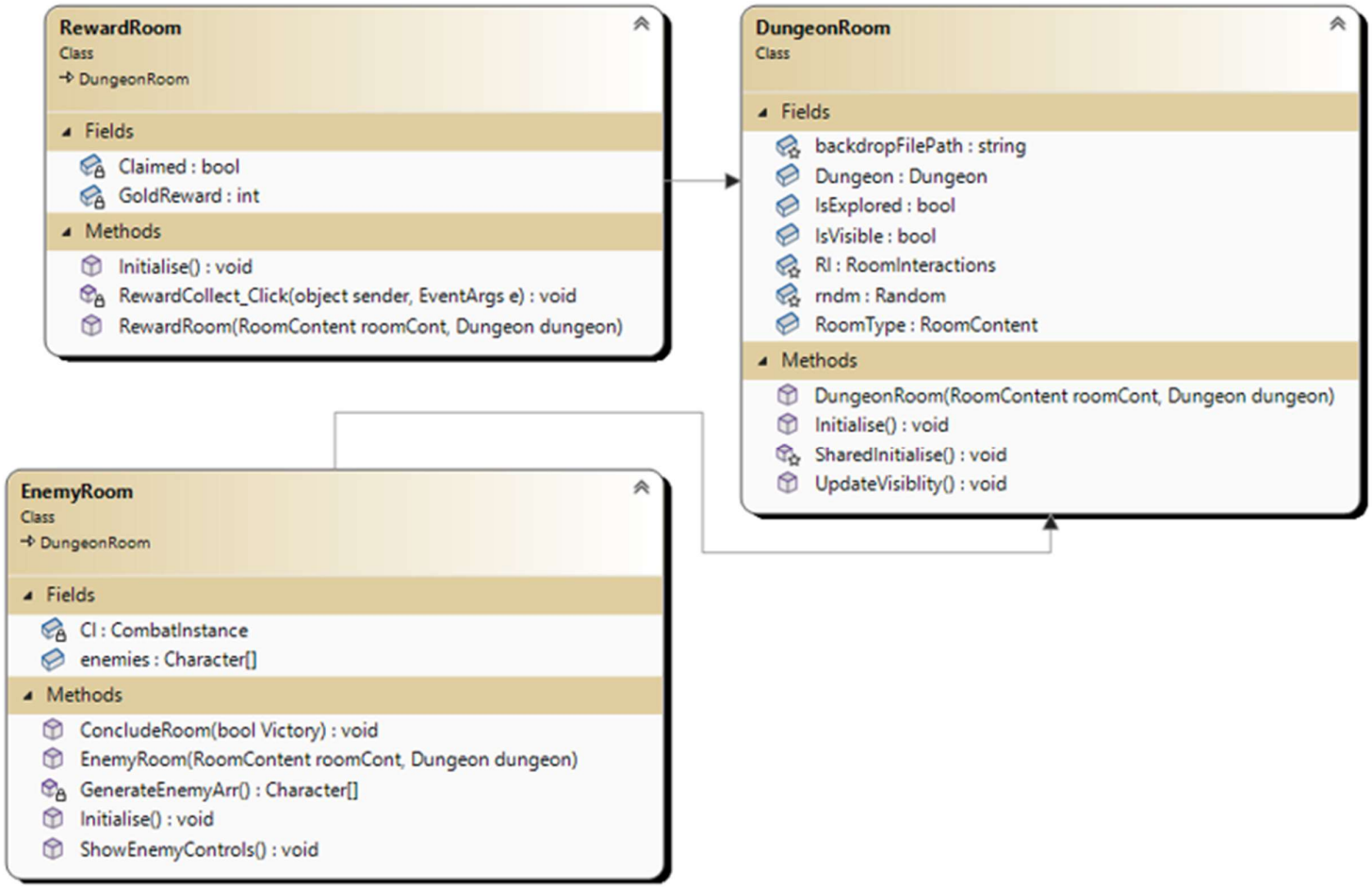


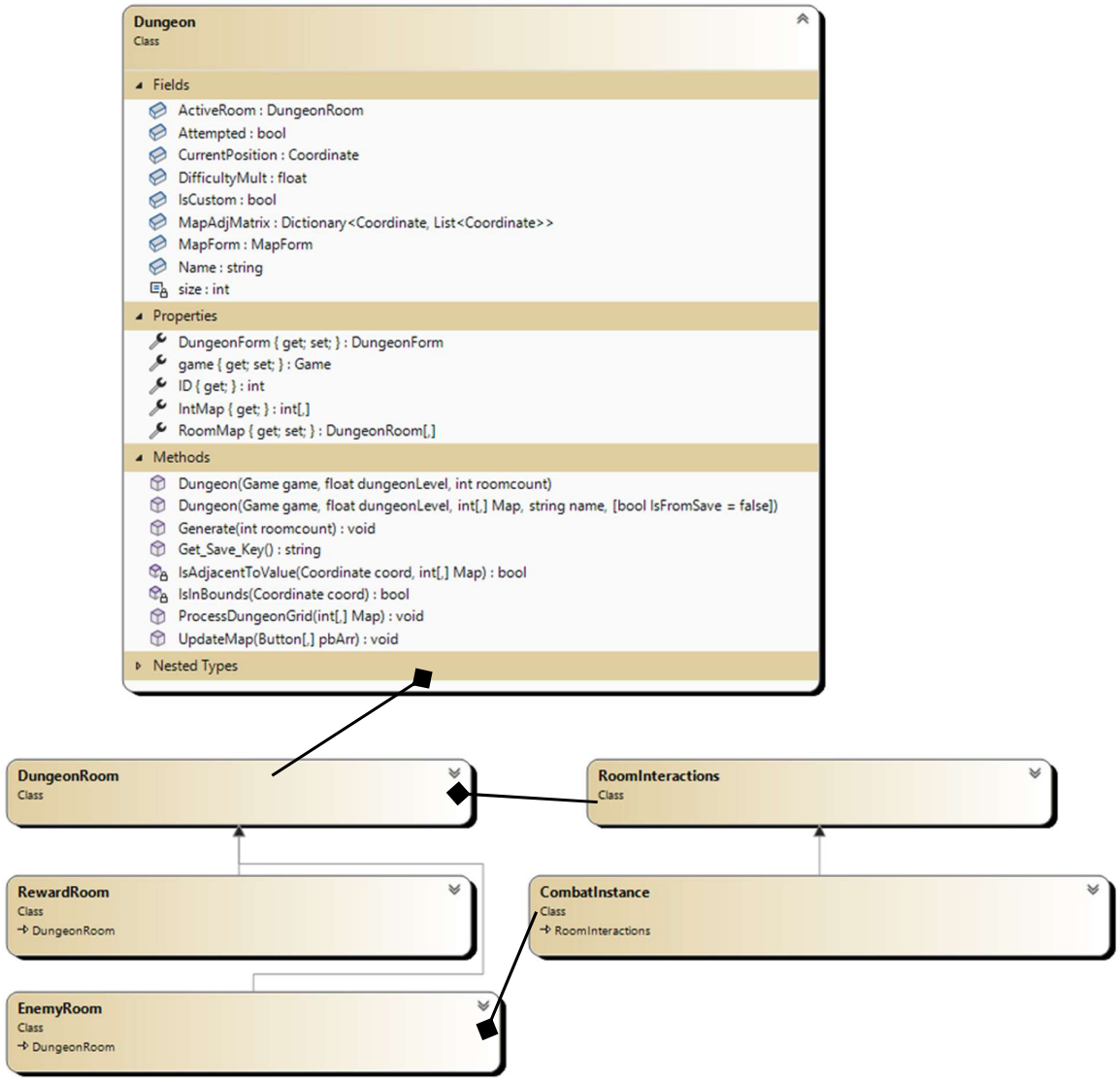
Figure 14 shows an overarching UML class diagram

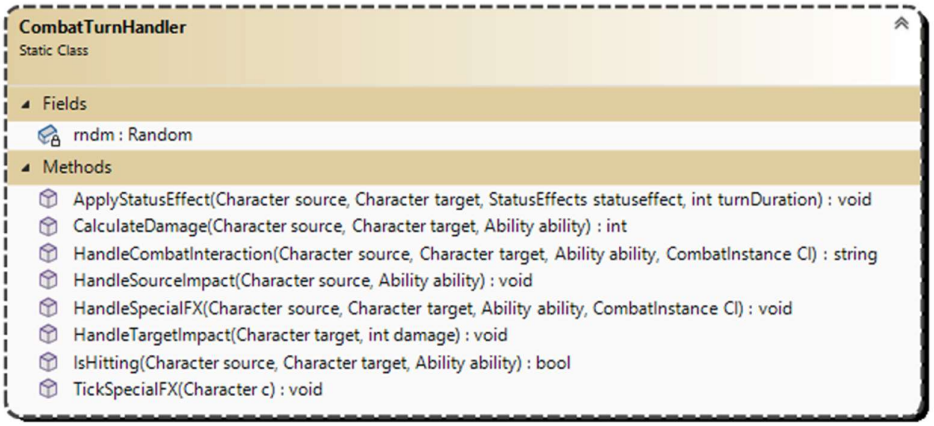

Character
Class
Fields
Abilities : Ability[]
ActiveStatusEffects : Dictionary<StatusEffects, int>
Armour : int
CharType : CharacterType
Controls : CharacterControls
CritChance : float
DodgeChance : float
Level : int
MaxHealth : int
MaxMana : int
Name : string
rndm : Random
Speed : float
Properties
ClassName { get; } : string
FullTitle { get; } : string
Health { get; set; } : int
isDead { get; } : bool
Mana { get; set; } : int
Methods
AddHealth(int health) : void
AddMana(int mana) : void
Character(CharacterType type, int level)
DrainHealth(int damage) : void
DrainMana(int mana) : void
Get_Save_Key() : string
HideControls() : void
HighlightSelected() : void
HighlightTargeted() : void
InitializeStats() : void
Kill() : void
MultiplyStats(float mult) : void
SetHealth(int health) : void
SetMana(int mana) : void
SetName() : void
ShowControls() : void
UnHighlight() : void
UpdateControlValues() : void
ValidateStats() : void
ValidateStatsCombat() : void
Barbarian
Class
Character
Bloodhunter
Class
Character
Wizard
Class
Character
Monk
Class
Character
HighwayMan
Class
Character
Rogue
Class
Character
Druid
Class
Character
Paladin
Class
Character
Bard
Class
Character
CharacterControls
Class
Fields
character : Character
ClassLabel : Label
HealthBar : ProgressBar
LBL_Armour : Label
LBL_Crit : Label
LBL_Dodge : Label
LBL_HPFraction : Label
LBL_ManaFraction : Label
LBL_Speed : Label
ManaBar : ProgressBar
NameLabel : Label
Picture : PictureBox
StatusEffectsBox : ListBox
Methods
CharacterControls(Character character, Ability[] abilities, int MaxHealth, int MaxMana)
RefreshStatLabels() : void
RefreshStatListBox() : void
UpdateValues() : void

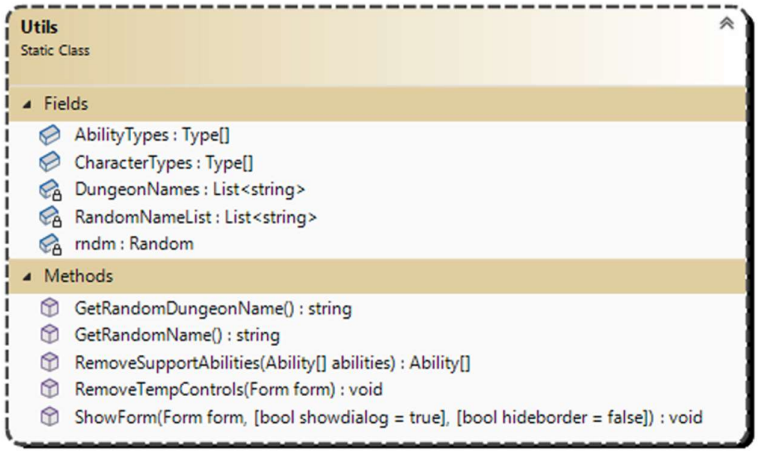

Ability
Abstract Class
Fields
Costs : AbilityCost
CritChanceMult : int
Damage : int
effect : SpecialEffect
MissChance : float
TargetablePositions : List<int>
TargetType : CharacterType
ValidCharacterTypes : Type[]
Properties
Name { get; } : string
TargetsAllies { get; } : bool
Methods
Execute(Character source, Character target, CombatInstance CI) : void
ToString() : string
Nested Types
AbilityCost
Class
Fields
Health : int
Mana : int
Methods
AbilityCost(int mana, int health)
PoisonArrow
Shoot
Heal
Frostbolt
SmokeBomb
PerfectShot
Backstab
DivineInterventi...
Punch
Parry
Whirlwind
DrainLife
Kick
Weaken
Berserk
Teleport
SwordSlash
Earthquake
ShurikenThrow
BloodSuck
DivineShield
ThunderStrike
BloodNova
MindControl
Fireball
Smite
Class
Ability


RoomInteractions
Class
Fields
Allies : Character[]
rndm : Random
room : DungeonRoom
XPadding : int
YPadding : int
Methods
LoadAllyControls() : void
RoomInteractions(DungeonRoom room, Character[] allies)
CombatInstance
Class
RoomInteractions
Fields
counter : int
Enemies : Character[]
HasConcluded : bool
maxCount : int
NextTurnBTN : Button
OrderedChars : List<Character>
room : EnemyRoom
TurnHistory : List<string>
TurnInfoLBL : Label
Methods
CombatInstance(EnemyRoom room, Character[] allies, Character[] enemies)
CombatStart() : void
ExecuteNextTurn(object sender, EventArgs e) : void
GenSpeedOrderChars() : void
LoadEnemyControls() : void
RefreshAllStatLBLs() : void


RewardRoom
Class
DungeonRoom
Fields
Claimed : bool
GoldReward : int
Methods
Initialise() : void
RewardCollect_Click(object sender, EventArgs e) : void
RewardRoom(RoomContent roomCont, Dungeon dungeon)
DungeonRoom
Class
Fields
backdropFilePath : string
Dungeon : Dungeon
IsExplored : bool
IsVisible : bool
RI : RoomInteractions
rndm : Random
RoomType : RoomContent
Methods
DungeonRoom(RoomContent roomCont, Dungeon dungeon)
Initialise() : void
SharedInitialise() : void
UpdateVisiblity() : void
EnemyRoom
Class
DungeonRoom
Fields
CI : CombatInstance
enemies : Character[]
Methods
ConcludeRoom(bool Victory) : void
EnemyRoom(RoomContent roomCont, Dungeon dungeon)
GenerateEnemyArr() : Character[]
Initialise() : void
ShowEnemyControls() : void


Dungeon
Class
Fields
ActiveRoom : DungeonRoom
Attempted : bool
CurrentPosition : Coordinate
DifficultyMult : float
IsCustom : bool
MapAdjMatrix : Dictionary<Coordinate, List<Coordinate>>
MapForm : MapForm
Name : string
size : int
Properties
DungeonForm { get; set; } : DungeonForm
game { get; set; } : Game
ID { get; } : int
IntMap { get; } : int[,]
RoomMap { get; set; } : DungeonRoom[,]
Methods
Dungeon(Game game, float dungeonLevel, int roomcount)
Dungeon(Game game, float dungeonLevel, int[,] Map, string name, [bool IsFromSave = false])
Generate(int roomcount) : void
Get_Save_Key() : string
IsAdjacentToValue(Coordinate coord, int[,] Map) : bool
IsInBounds(Coordinate coord) : bool
ProcessDungeonGrid(int[,] Map) : void
UpdateMap(Button[,] pbArr) : void
Nested Types
DungeonRoom
Class
RoomInteractions
Class
RewardRoom
Class
DungeonRoom
CombatInstance
Class
RoomInteractions
EnemyRoom
Class
DungeonRoom


CombatTurnHandler
Static Class
Fields
rndm : Random
Methods
ApplyStatusEffect(Character source, Character target, StatusEffects statuseffect, int turnDuration) : void
CalculateDamage(Character source, Character target, Ability ability) : int
HandleCombatInteraction(Character source, Character target, Ability ability, CombatInstance CI) : string
HandleSourceImpact(Character source, Ability ability) : void
HandleSpecialFX(Character source, Character target, Ability ability, CombatInstance CI) : void
HandleTargetImpact(Character target, int damage) : void
IsHitting(Character source, Character target, Ability ability) : bool
TickSpecialFX(Character c) : void


Utils
Static Class
Fields
AbilityTypes : Type[]
CharacterTypes : Type[]
DungeonNames : List<string>
RandomNameList : List<string>
rndm : Random
Methods
GetRandomDungeonName() : string
GetRandomName() : string
RemoveSupportAbilities(Ability[] abilities) : Ability[]
RemoveTempControls(Form form) : void
ShowForm(Form form, [bool showdialog = true], [bool hideborder = false]) : void

User Interface designs

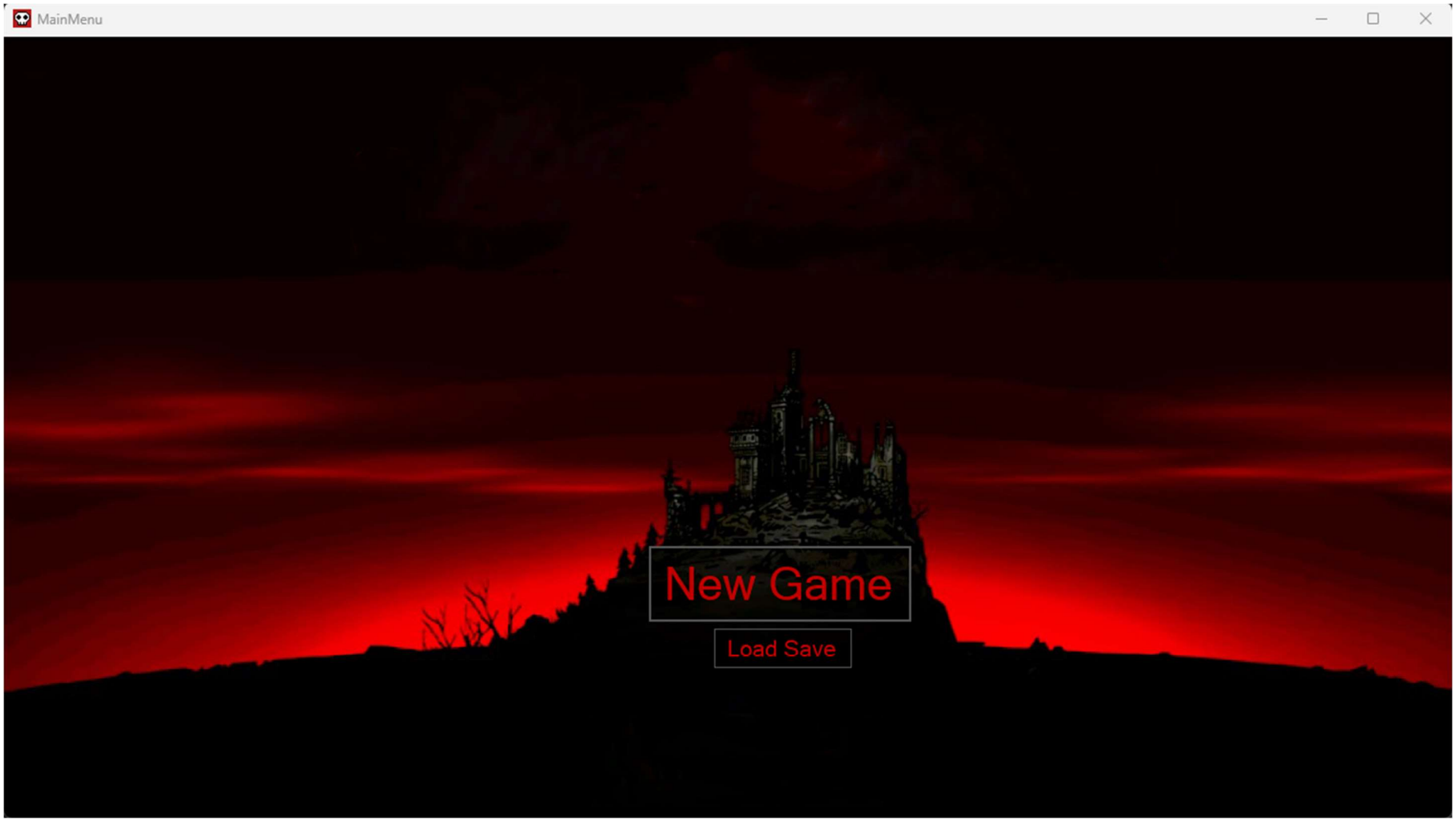


Figure 15 Shows the main menu screen

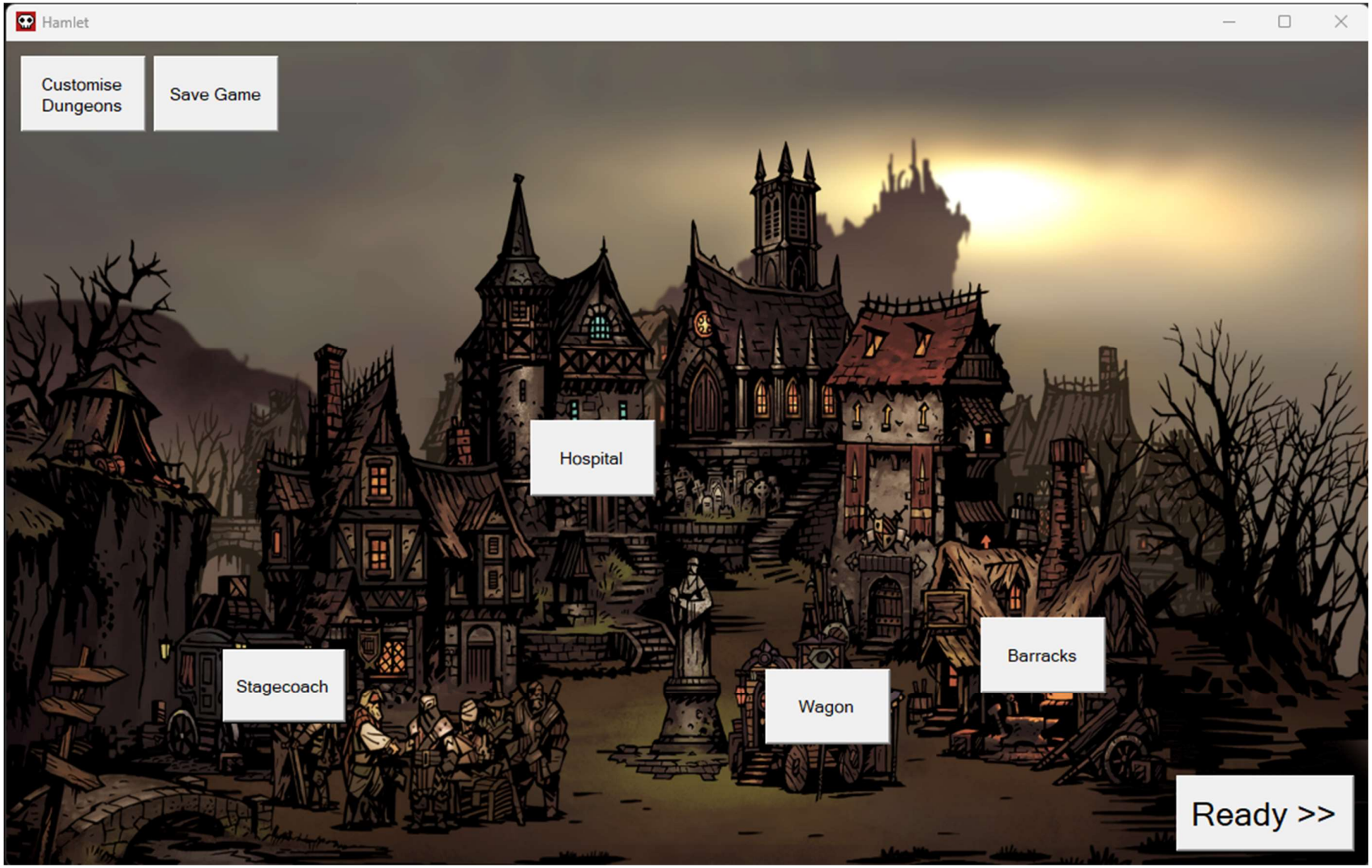


Figure 16 Shows the hamlet form

Figure 17 Shows the stagecoach form, the interface where the player buys new heroes

This interface is navigated to via the stagecoach button in the hamlet.

Initially, the form is blank except for the “Buy new character” button in the bottom right. Upon buying a new character, a new character for-purchase is added to the form, starting from the top. This includes detail on its stats, as well as their purchase price. The price is calculated dynamically where each point of each different stat is given a certain gold value, along with flat amounts depending on the class & level of the character.

Buying a new character should be treated as sourcing someone who can then be purchased. When the user attempts to purchase a new character after 4 characters have been purchased (i.e. the form is full), a random character will be replaced with a new character.

Generated characters are always within 1 level of the player’s current level.

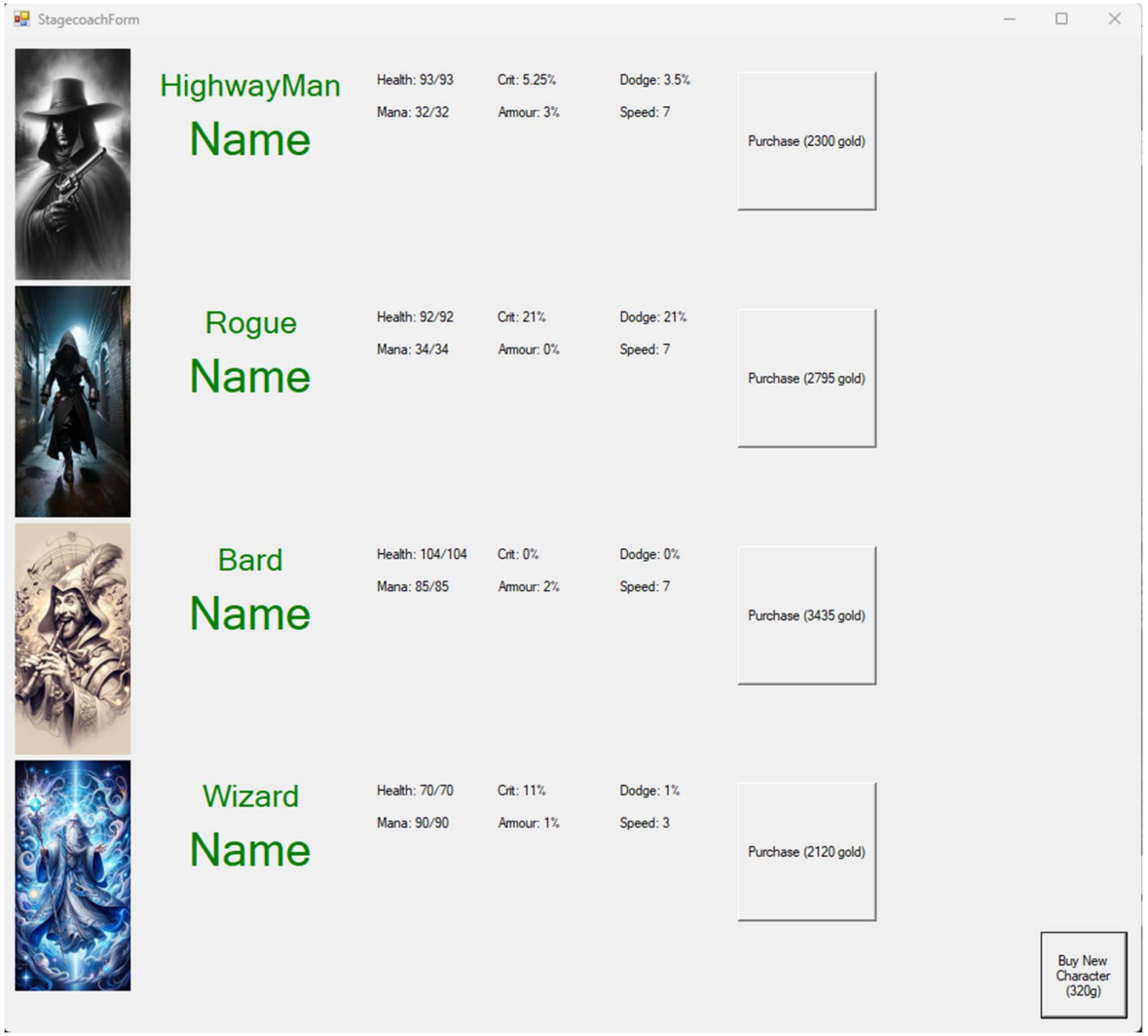


Figure 18 shows the hospital form, where players heal their heroes.

Within this form, the player can heal their heroes. Heling a hero will remove any status effects currently active as well as filling their health. The cost is solely calculated based on missing health, with a flat base cost of 100.

This means if a user tries to heal a full health character with a status effect, it will cost 100.

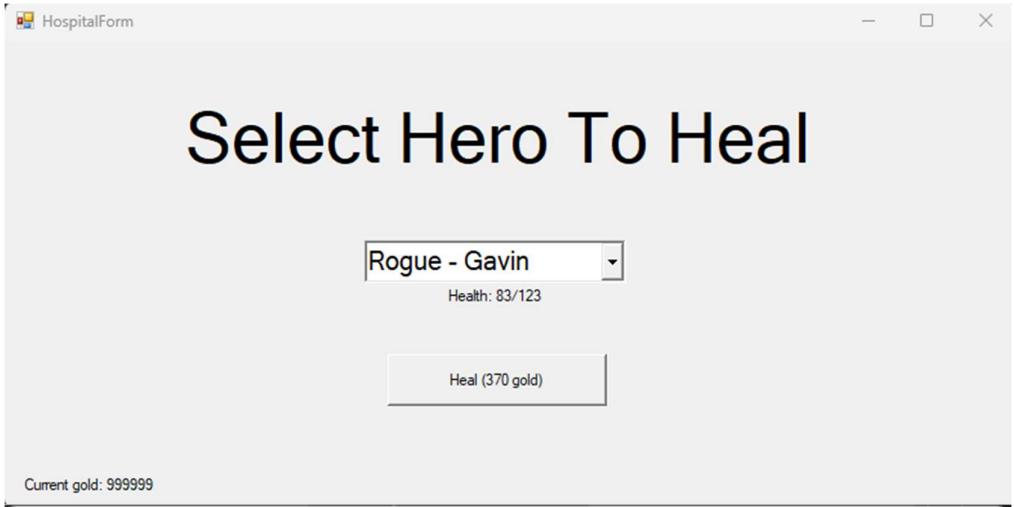


Figure 19 shows the interface for players to swap around the members of their party.

The party interface consists of 4 drop down menus, each drop down menu contains all characters that the player owns. Upon selecting a character, it is swapped into that position within the party of the player.

If the player select a character that is already in their party, the original position of the character in the party is made blank.

If the user attempts to continue with an incomplete party (less than 4 members), then the party is reverted to its original state.

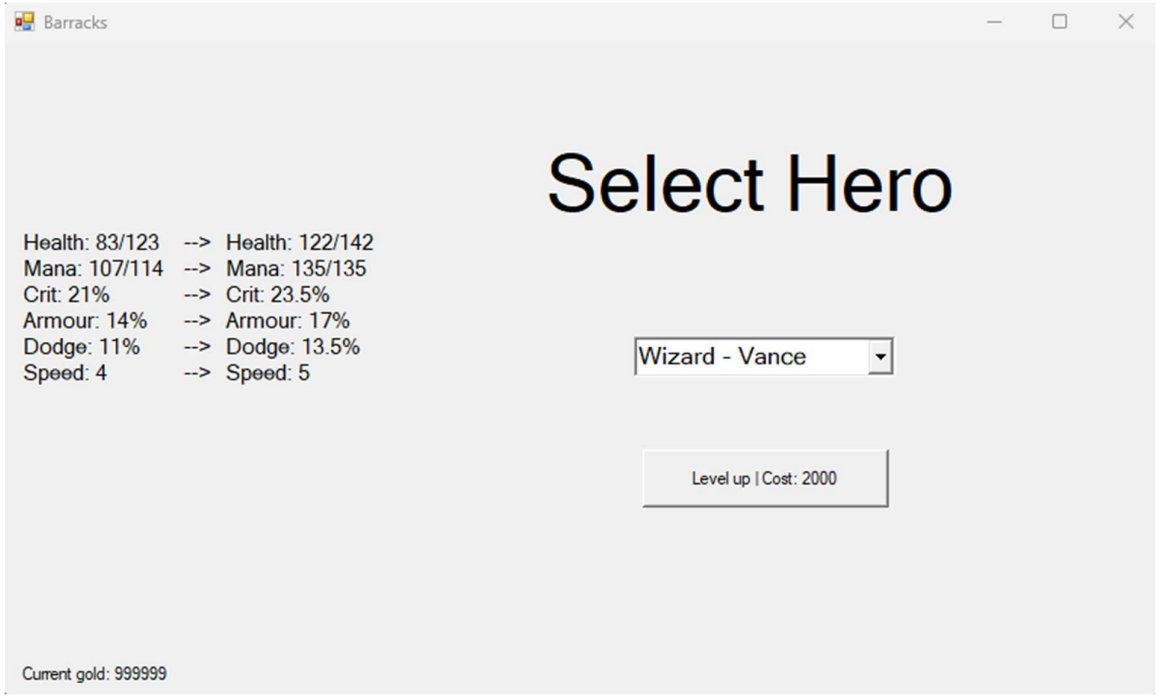
Figure 20 shows the interface for character upgrading

On this character upgrade interface, the current stats of the selected character are shown on the left of the arrows, and the potential future stats of the character after the upgrade are shown on the right.

The drop down menu is how the player selects their character, this drop down menu contains all of a player's current party.

The cost of levelling up changes based on the selected character's current level.

Figure 21 shows the interface for creating and loading custom dungeons.



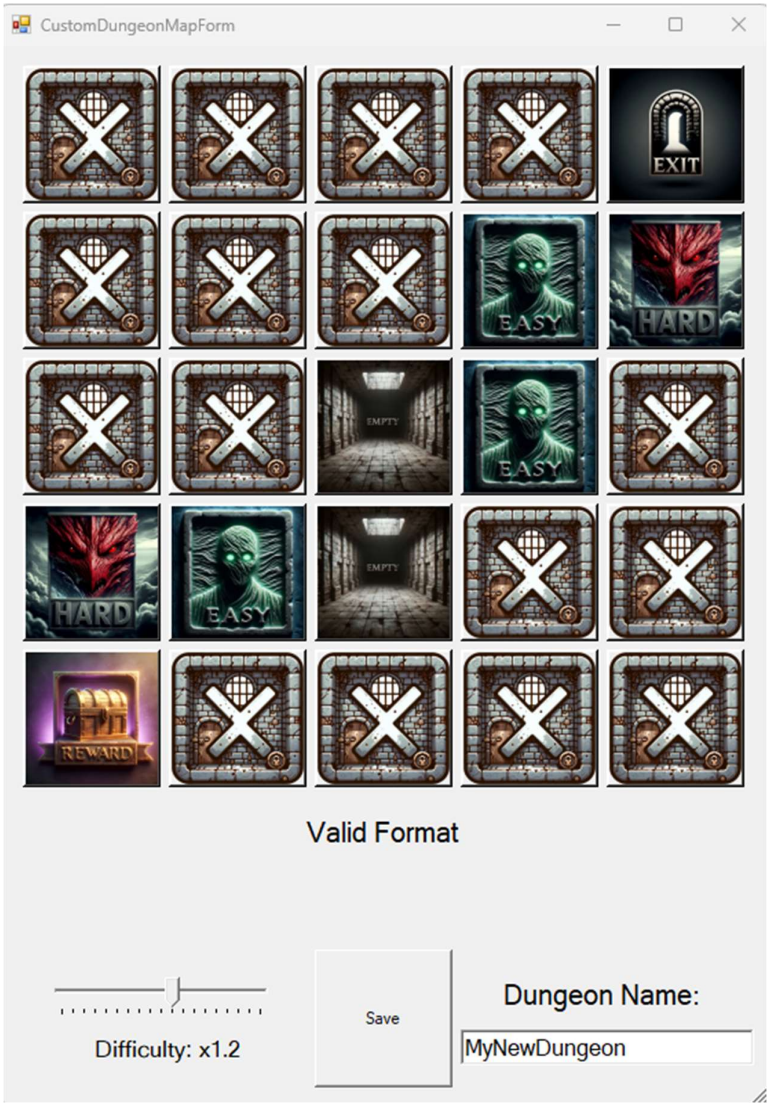


Figure 22 shows the interface for creating custom dungeons.

The custom dungeon creation form consists of a typical dungeon map interface, joined to a set of controls which take further inputs for data required to form a complete dungeon, rather than just a map. Including:

- Dungeon name
- Dungeon difficulty multiplier

To make the interface as intuitive and easy to use as possible, the way of selecting what goes in each room is done by clicking each room's icon, the X icon denotes a non-existent room.

Upon a room being clicked, it will cycle to the next type of room, going through:

- Void
- Empty
- Easy
- Hard
- Reward
- Exit

In the background, a searching algorithm is checking, with every room change, whether a dungeon is "Valid". This would mean checking for an exit room, that is connected to the central room via an accessible path. Additionally, the validation algorithm checks to ensure all rooms are connected to prevent inaccessible islands.

Upon pressing save, a dungeon key (which can be parsed into a playable dungeon as in figure 19) is placed into the users clipboard as well as saved to a text file.

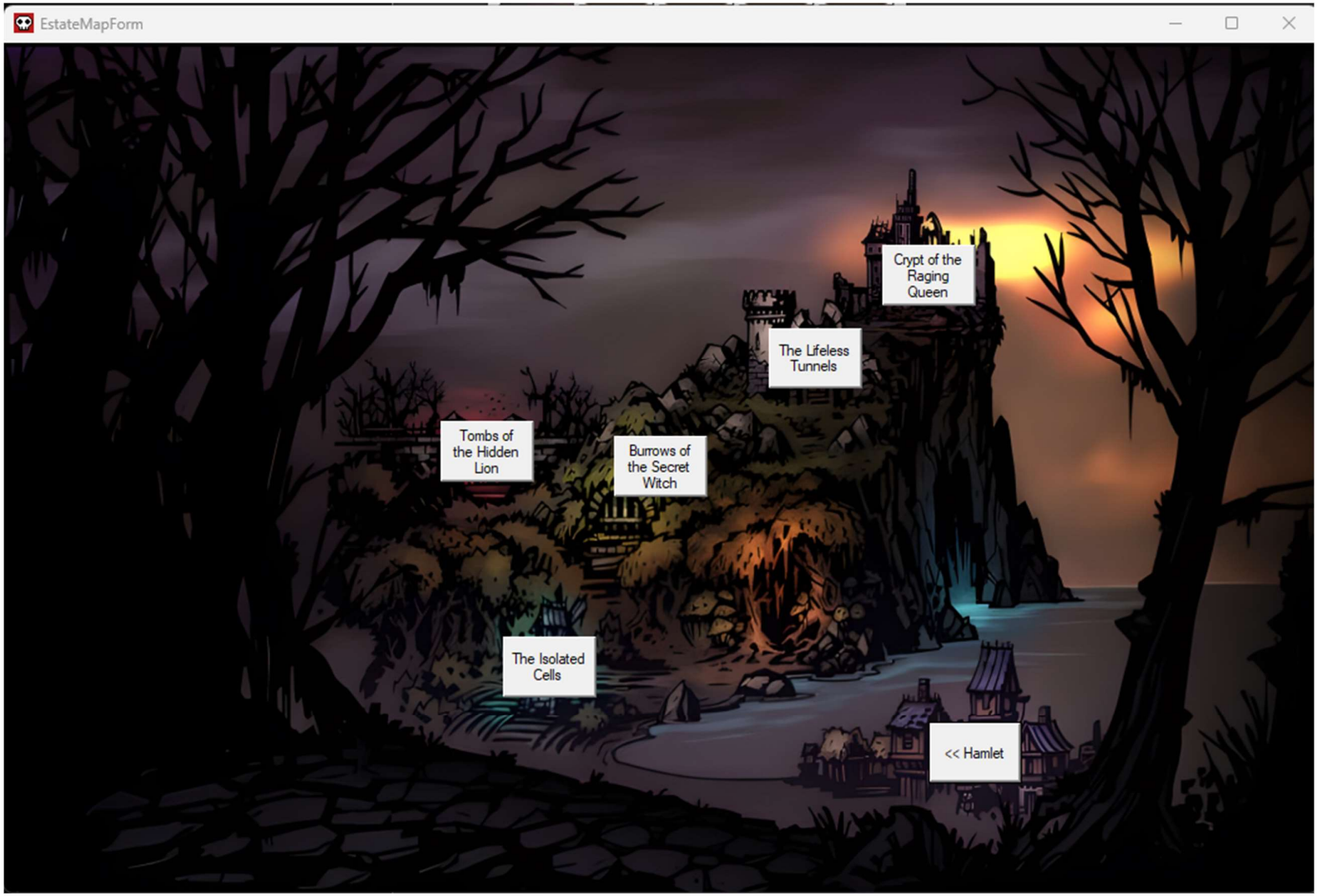


Figure 23 shows the interface for dungeon selection

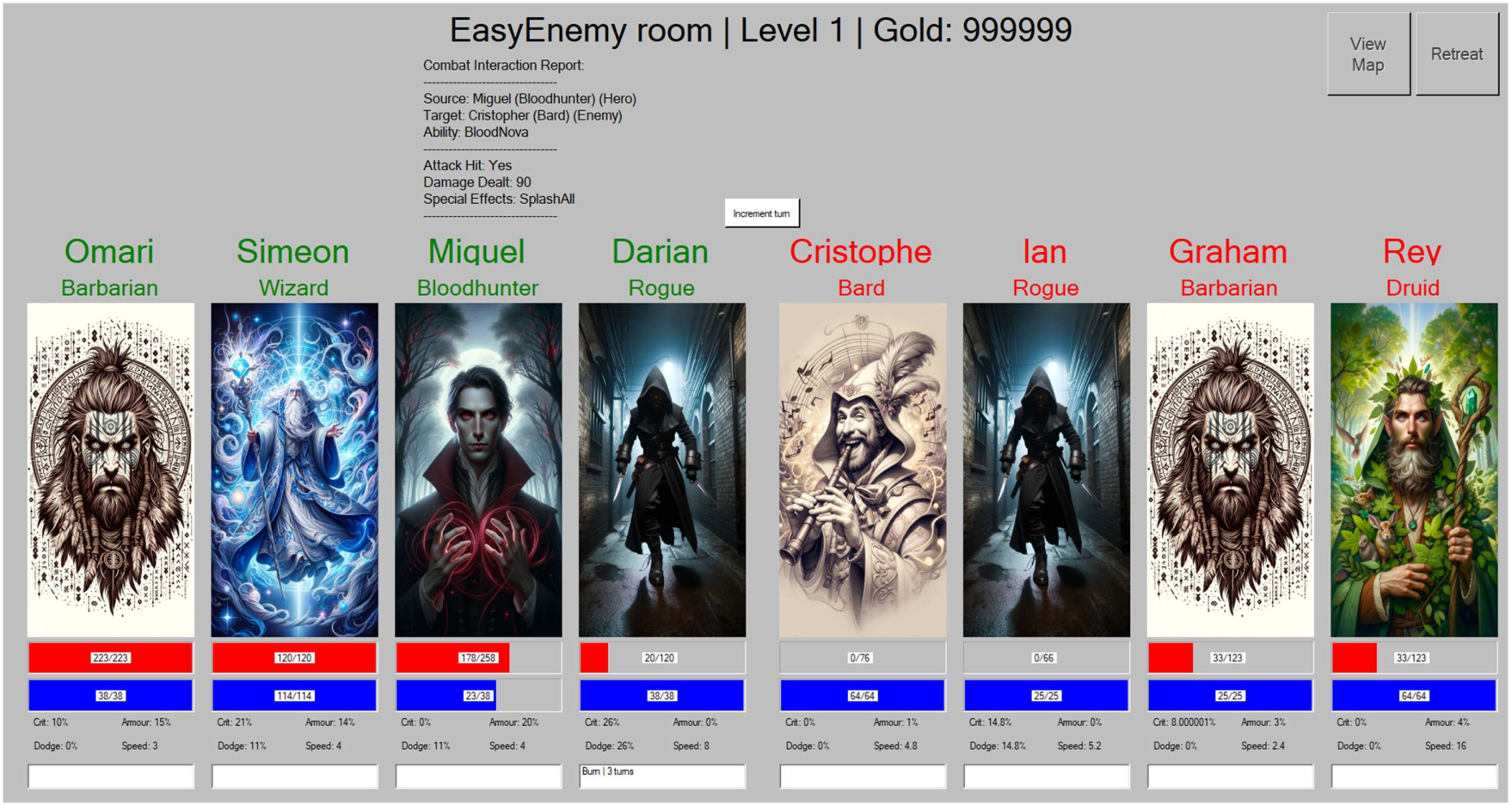
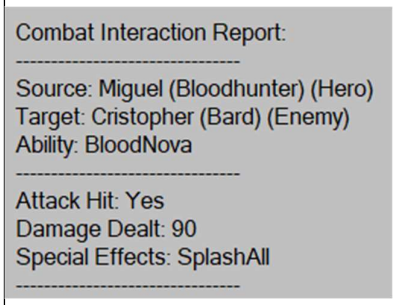
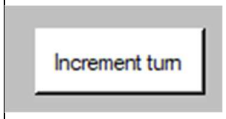
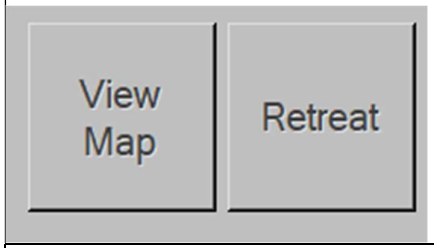


Figure 24 shows the combat interface.

Image Reference	Description
	A combat interaction report is displayed each time an ability is used and processed. During development, I noticed a need for the player to understand what had happened during an enemies turn, since there are no animations within the game health, mana, and status effects would seemingly mysteriously appear and it was hard to understand what was going on. The combat interaction report shows exactly what has happened, with important elements such as special effects and whether or not the attack hit.
	The increment turn button is central to the entire combat interface. From a development perspective, it was something I wasn't happy about having to introduce. My original idea was going to be to wait for a user's input, however upon further research I concluded that forms was not designed for time based inputs, nor was it possible to effectively "wait" for a certain button to be clicked before moving on. The increment turn button ended up being the compromise that had to be made to make a turn based combat system work.
	Two buttons that are always present on a dungeon's interface are "View Map" and "Retreat" During combat, both buttons are disabled. Both are only available when out of combat. This would be either after a combat interaction has ended, or when the player is in a non-combat room, such as a reward room.
	The status effect box shows the status effects that a character has been afflicted with, alongside the name are the number of turns left for the status effect to last. All status effects have a turn based effect, for example burn deals damage every turn it is activated.

Every character has a set of controls that are displayed to represent them.

The name of characters is randomly generated upon their creation, and when a character dies their name is re-added to the list that is used to source random names. This ensures that unique names will never run out.

All characters having unique names is not solely for decoration, without the unique name and class pairing, in various select menus when choosing characters (Upgrading, Healing, Targeting, etc…), it is ambiguous what character you are selecting if you have multiple of the same class of character.

Health and mana bars are also shown at the bottom, with a label overlayed on top of them to show the maximum and current amount. Both the bars and numbers update in real time as turns are executed.

Then, the characters individual stats are displayed, such as speed and crit chance.

The portrait of the character is intended to ensure a character feels to a degree like there is actually something there, as without it the game would feel extremely text and number based, and there would be no humanity or realism to what is actually going on.

	The target & ability select menu is displayed every time a player's turn begins. It allows them to first select an ability, and then to select a target. Due to the fact that abilities have limitations in terms of what positions they target, the ability must be selected before selecting the target. The target selection drop down is then limited based on the restrictions of the ability. If the player select an ability that can either only target one character, or that hits for splash damage (and so the target is irrelevant), the target box is automatically populated for the player. The purpose of the submit button is to allow the player to think about their selection before going through with it, as they may wish to change their mind.