1. **Introduction**
   1. **Purpose of the system**

Dread of the Evil Wizard aims to provide a unique gaming experience with presenting a timeless concept with a new infrastructure, a brand new implementation for a text based RPG. The main purpose of this game is to save the sister of our main character from the evil wizard. The way to defeat the evil wizard is completing the missions given by the characters in the game and fighting the monsters and become victorious along the way.

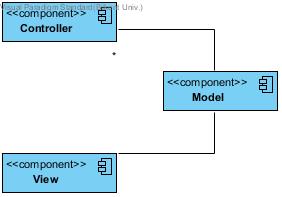
* 1. **Design goals**
* **Usability:** Text based nature of Dread of the Evil Wizard provides a very user friendly and easy to understand game for our players. The explanations regarding storyline, items, game dynamics, creatures, powers and quest goals provided in the help page using plain English. No special interface experience or knowledge about special key combinations are needed since the game is text based but player needs to know the valid commands provided in the help section.
* **Robustness:**  To have a reliable infrastructure Dread of the Evil Wizard especially command interpreter must differentiate valid and invalid commands entered by the user and only execute the valid commands. Command validation depends on the current situation of the character for example attack commands outside of battle mode ignored by default. Also while loading game if the save file is missing or corrupt game starts with default settings since dying results with deletion of the save file game is already compatible with missing files. These properties of File Manager increase the fault tolerance of the game.
* **Extensibility:** Additional characters or quests can be added or modified in the game later on. In order to achieve this, creating a good documentation regarding the inner dynamics and object models of the game was necessary. This documentation would minimize the compatibility problem may occur extending the features of the game.
* **Efficiency:** The throughput is limited to only one move at a time (since the game is turn based) so calculations and responses are not pose a big problem and to make the game more efficient the map itself (objects that are non-interactive or stays still) is not rendered but displayed as an image in the form of background then the objects that are dynamic is rendered by the game on top of this background. This method must be applied for both battle mode and regular game mode.
* **Portability:** The Java implementation provides a platform independent game where every system compatible with running Java software would be able to run this game successfully.

**Trade-offs:** Some trade-offs needed to be decided to implement these design goals that listed below.

* **Efficiency vs. Portability:** By implementing the system with Java we obtain a portable game which would be functional in any system capable of running Java but less efficient compared to beforehand compiled languages like C/C++.
* **Space vs Speed:** Handling the actions with two separate components with Battle Manager and Map Manager instead of just one. This eats up more space but runs more quickly because of the purpose built nature of the separate components.
* **Functionality vs. Usability:** The game provides the player with options about classes, character attributes, status affects etc. This options affect the player’s choices regarding the selection of a character type, a skill or an attack to be used in an arbitrary moment of the game. To simplify the learning curve relating the matter number of these factors must not overwhelm the user but simultaneously present a diversity of options to maximise the gaming experience.

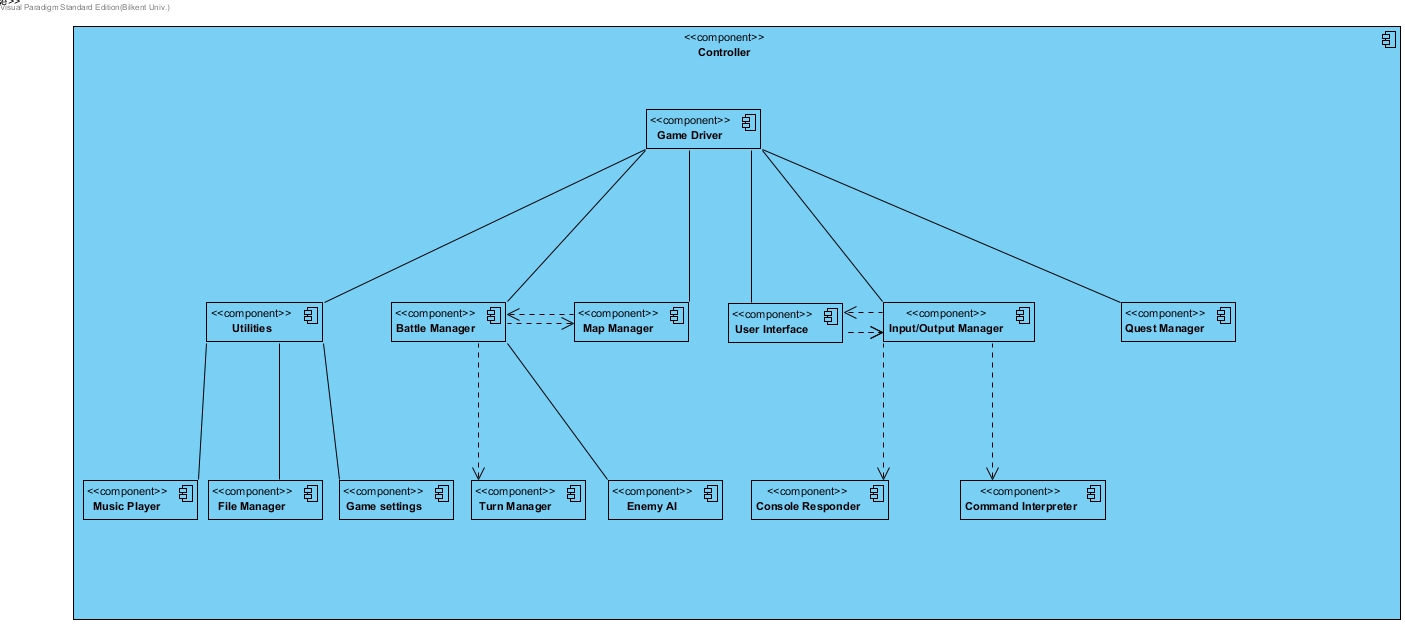
1. **Software Architecture**
   1. **Subsystem Decomposition**

We divided our system into relatively independent Sub-systems to better understand and explain it. For our architectural choice, we went with MVC. This means that we divided the program into 3 big subsystems. Model part hold our game entity objects, View part hold the user interfaces and lastly, the controller part hold the system logic (Business Intelligence). Controller part has its own sub-systems. These sub-systems are responsible for the core features of the game.



**Figure 1**

Figure 1 shows the interactions between model-view-controller parts. Figure 2 Shows the inside of the Controller part. We used 3 layered architecture to design this part. 3 layered architecture gives us better maintainability and flexibility, so if we need to change something or add something to these systems, it will be relatively easy.



**Figure 2**

* 1. **Architectural Styles**
     1. **3 Model View Controller**

We classify subsystems into three parts, called model, view and controller.

We add a controller part between the domain knowledge and the user interface and we are able to separate the domain knowledge from the user interface.

Here, the domain objects of our system is only accessed and controlled by manager classes which are grouped under Game Driver layer. Under this, main subsystems exist that interacts with both users and system. Below the main subsystem layer, small subsystems are exist. They will be used for providing features to the main subsystems. With the MVC system, we can ensure that changes made on the interfaces will not affect the model of the system, therefore it is a good idea to use MVC for our game.

* 1. **Hardware/Software Mapping**

The game requires only Java software to execute. No special hardware devices are needed to play the game except a regular computer system with a monitor a mouse and a keyboard. Our game does not have high performance demands since the game is 2D and throughput is limited to one turn at a time. Also Dread of Evil Wizard is a single player game so there is no need for an external connection like internet. For gameplay no special keys mapped to special functions since the game is text based. Players will be using predetermined commands to play the game. Any other input than this set of commands will be functionless. Outside the game session regular mouse controls will be used for most functions such as navigation in the menu, load/save operations and so on.

* 1. **Persistent Data Management**

Dread of Evil Wizard uses its own file extension to save/load the games to storage. Reason behind this is to avoid misread or corrupt other files which may be in the same save directory. Also death of the main character results in deletion of previous save files so the game must differentiate which files are relevant. A Database system is not needed to implement this specifications since there won’t be any concurrent file accesses.

* 1. **Boundary Conditions**

In case of data loss or corrupt file game returns to default settings and starts the story from the scratch. To be able to save the game player’s character must be in the town center so any attempt to use the save function of the game will be resulted in a message saying the character must be in the town center. Player will be able to have multiple save files with different characters. And these saves of different characters would not be affect one another. Program does not allow to instantiate multiple games at the same time.

1. **Subsystem Services**

In this section we will talk about our sub-systems and their functions. We used MVC for our basic division of our sub-systems and 3 layered opaque architecture to design the Controller part.

* 1. **Game Driver**

This our core sub-system. This is where our main class will be. Game Driver is our 1st layer of the 3 layer architecture. This will be responsible for communicating any necessary information between other the sub-systems and other parts of the program. We can think this sub-system as the boss of the whole system.

* 1. **Battle Manager**

Battle Manager is responsible for anything battle related in the game. It uses map manager, turn manager and enemy ai to advance the battles. It gets the user info from the game driver and acts accordingly.

* 1. **Map Manager**

Map manager is responsible for anything map related in the game. Map manager will handle all the things related to both the zone maps and the battle arena maps.

* 1. **Turn Manager**

Turn manager is responsible for the turn advancements and turn based effects on the characters. Turn manager keeps track of whose turn it is in the battle and relays this information to the battle manager so that it can do its job.

* 1. **Enemy AI**

Enemy ai will be responsible for controlling the opponents in battles. It will act as a user in a sense but its outputs will directly go to the battle manager. Turn manager will tell whose turn it is, if it’s an opponent’s turn battle manager will ask enemy ai to play the specific characters turn.

* 1. **Utilities**

This subsystem is responsible for managing the functions that are outside the game such as save files, music or settings of the game. Utilities invoke the relevant subsystems to these needs accordingly and the invoked subsystems makes the necessary arrangements about their specified tasks.

* 1. **Music player**

Music Player handles the music will be played or not. When user changes the settings in the menu Game Settings informs the Utilities and Utilities activate Music Player according to this settings. When the player is battling Music Player plays a different song when the player is roaming in the map.

* 1. **Game Settings**

Game Settings manages the player’s preferences about the game. This preferences are whether the music is on or and the difficulty level user is selecting to play the game in.

* 1. **File Manager**

File Manager controls the data management of this game. Player saving or loading is done through this subsystem. The controls about files are made by File Manager. Control conditions for loading are checking if the file exists, if exists checking the extension of the file to make sure the file is game relevant and the last step checking the file for corruption. If any of these steps fails the load operation is failed and Utilities is informed to display the relevant problem to the user. The save case is much simpler only control is about the main character’s location on the map and if so game creates the save file. The death of the player results in deletion of all saves of the player.

* 1. **Input/Output Manager**

Input/Output manager subsystem is on the 2nd layer of controller subsystem. Input/Output manager is responsible for receiving the commands entered by the user and sending the commands to Game Driver in a way that it can be processed. Also this subsytem is responsible for informing the user via displaying appropriate messages on the screen when it is necessary. Input/ Output Manager has relationship with 4 different subsystems.

It has partition relationship with User Interface subsystem so that it can use the services to receive commands entered through User Interface.

It has runtime dependency with Command Interpreter subsystem, so it uses the services of Command Interpreter Subsystem to convert the commands taken from the user as input to a form that Game Driver can understand.

It has run time dependency with Console responder. Console will be ready after the user selects to play game via either selecting Starting a new game or successfully loading a saved game. Input/Output Manager Subsystem will get the commands from user through Console Responder. Furthermore, Input/Output manager will send appropriate messages to the user through Console responder as well.

Lastly, Game Driver has compile time relationship with the Input/Output manager to keep Interaction with user by processing the commands entered by the user In a way to keep the game alive!

* 1. **Command Interpreter**

Command Interpreter is responsible turning the commands entered by the user into appropriate form that Game Driver can understand. Having such a subsystem enables user have a fault tolerance. In other words, if the user enters a command that does not match exactly the same input asked from the user, command Interpreter still can convert it into a form that game will understand. For example, if the user wants to move to the North and enters the command without case sensitivity, or if the user enters “nOrtH” instead of “North”, Command manager can still correct it and send it to Input/Output Manager. Adding such a subsystem increases the robustness of overall system.<s

* 1. **Console Responder**

Console responder subsystem is responsible for giving the necessary response information to the user. When user enters a command and game driver does what it supposed to do, console responder gives an appropriate feed back to the user via writing. Since the only visual feedback user will be getting will be from the map we will rely heavily on the console responder. These feed backs can be exemplified like, giving a description of what is around the user when user says “Look around” or telling how much damage the user did to a monster after he attacked.

* 1. **Quest Manager**

Quest Manager provides services to the Game Driver about Quest functionalities. It is responsible for keeping track of the overall Quests of the Player. In other words, Quest Manager keeps track of the ongoing and completed Quests. This subsystem is responsible for updating the states of each quest that player is still doing. Furthermore, whenever player tries to start a quest, it’s Quest Manager’s responsibility to compute the necessary requirements for completing the quest and updating the quest to “ongoing” status.