**Design Paradigm (250-350)**

In our design of project 3 we implemented a combination of the Object-Oriented Design paradigm and the Function-Oriented Design paradigm. Our design of the enemies and the player in the game are built with these design paradigms in mind. For example within the player object we developed certain moves within the player class that have their own functions connected with each move respectfully. Furthermore in the EnemyAI.js file there is an enemy AI object that has a very similar build to the player object. In the Enemy object it was created so that the prototype would have the same layout in the UI for enemies of different types. For example, we have a barbarian and a wizard enemy where they have different moves and heath, when implementing them we just let these new enemies be objects of the normal enemy class.

In the overWorld it utilizes both a function-oriented design paradigm and an object-oriented design paradigm. In the overWorld it utilizes many functions within the Kaboom.js library to create levels and create events within those levels. Our levels are just arrays of char's where each char is defined to be a certain image from our sprites, an example of this is the walls being defined to be the wall.png within our sprites folder. Within these definitions of chars we have functions associated with them, so a wall has a function called solid() where it doesn't allow the player to pass through it. The player is defined as an object of the level and is given specific functions to determine the flow of the game, if a player runs into an enemy the player.onCollide function will be called and take the player to the battle scene with that enemy.

**Software Architecture (250-350)**

The kaboom.js library uses an event-based architecture to allow users to interact with the game, and we extended that architecture for our own uses. The entire battle scene makes use of onClick and onHover events to update the game objects. For example, when the player hovers over one of the move options, the resulting onHover event sets the description text to a description of the move they are hovering over, and when they move the mouse off of the move, the next frame the game sets the description to the enemy’s flavor text. When the player clicks a move, the resulting onClick event starts a sequence of events that lead to a sort of “dialogue” where the text box advances each time the player clicks on the screen. Even for things outside the player’s control, we use a central eventQueue object that advances through onClick events.

The overworld is not as strongly event-based, as it only uses two event listeners: onUpdate, which runs every frame to check for player movement; and onCollide, which checks for game objects overlapping with one another. However, at its core this system is also event-based, the only difference being that many of those events happen very frequently and regularly.

In both of these scenes, the system for handling events and distributing them to the necessary listeners is handled internally by kaboom.js, and has proven to be a very useful architecture for game development, as a similar system is used by most game engines, such as Phaser and Unity.

**Design Patterns (250-350)**

The enemies in the game make use of the “prototype” pattern. We created a single base “enemyProto” object with several properties that all enemies will use (hp, maxHP, name, die(), takeDamage(), etc), and for each enemy we create a copy of that object and modify/add the necessary properties, such as specific moves and properties to track the enemy’s state. While the prototype does not make heavy use of this feature (considering it only has 1 or 2 different enemies), the system has been set up to be easily scalable and expandable into several different enemy types, each with unique behaviors and movesets, without having to repeat large sections of code for every single enemy type.

Our current prototype alone features 8-12 different moves, so making a separate function for each one could quickly become awkward and impractical. That’s why we implemented the “strategy” design pattern to implement both player and enemy movesets. Each move is represented by an object with the same interface: a name (string), a description (string), a “pretext” (ie what the game says when you select the move), and a function that actually performs the move, returning some feedback text to put on the screen afterwards. By doing this, we can store the player’s moveset by directly storing the move objects, and can modify it just as easily. This also allows moves to store internal information, so a move could behave differently on subsequent uses, without the player object needing to handle it in any way.