Plan of Attack

Project Breakdown & Schedules:

Task & Milestone	Details	Assignee	Completion Date
Brainstorm, discussion & general planning	Interpret project requirements, communicate mutual expectations, and consolidate understanding of object-oriented design	Simon & George	Monday, Nov. 17
Specific planning & UML prototype	Plan hierarchical relations between game components and experiment with various design patterns Produce rough paper draft of UML diagram	Simon & George	Tuesday, Nov. 18
Plan of attack draft	Note down tentative answers for design questions and vague schedule	George	Wednesday, Nov. 19
Final UML design diagram	Finalize details of object hierarchy and game logic Produce organized UML diagram	George	Thursday, Nov. 20
Final plan of attack	Proofread and edit plan of attack Revise schedule to reflect changes	Simon	Thursday, Nov. 20
Header files & Skeleton code	Translate UML design to header files and make relational linkages	Simon	Friday, Nov. 21
Milestone 1: Design phase completed			Friday, Nov. 21

Main driver & Game	Implement driver function, game loop, command interpreter and command-line interface	George	Saturday, Nov. 22
Character & Entity	Implement overall framework for spawnable entities	Simon	Saturday, Nov. 22
Player & Enemy	Implement concrete character subclasses with specific overrides for each race	George	Sunday, Nov. 23
Floor, Cell & Tile	Implement game map components and file loading/parsing logic	Simon	Sunday, Nov. 23
Potion, Treasure & Stair	Implement status effect items, and make relational linkages to game map and characters	George	Tuesday, Nov. 25
Unit testing	Write unit tests for complex, error-prone, or intricately overridden methods	Simon	Tuesday, Nov. 25
Integration testing	A comprehensive testing of the entire game system, with particular focus on: - Spawning mechanism - Combat system (special effects) - Potion effects - Level advancement & end game	George & Simon	Wednesday, Nov. 26
Milestone 2: Core implementation phase completed			Wednesday, Nov. 26
DLC: Inventory	Implement inventory space to store items	Simon	Thursday, Nov. 27
DLC: WASD with curses library	Implement a more natural command input system	George	Thursday, Nov. 27
DLC: Equipment system	Implement equipment slots for permanent status effect items	Simon	Friday, Nov. 28
DLC: Random floor generation	Implement system to generate floors with random layouts	George	Friday, Nov. 28

Integration testing of enhancement features	Ensure additional features does not break any core game functionalities,	George & Simon	Friday, Nov. 28
Milestone 3: Enhancement Implementation phase completed			Friday, Nov. 28
Revised UML design diagram	Revise UML to reflect actual structure of the project	Simon	Saturday, Nov 29
Revised answers to design questions	Revise answers to reflect changes made during the implementation phase	George	Saturday, Nov 29
Design document draft	Provide overview of all aspects of the project, including high level implementation and design patterns	Simon & George	Sunday, Nov 30
Final design document	Proof read and edit final design document	Simon & George	Sunday, Nov 30
Milestone 4: Final design & documentation phase completed			Sunday, Nov 30

Preliminary Design of Core Components

General Interaction Framework

For the project, we decided that a map-oriented representation would be most suitable. In other words, the overall game controller keeps track of the game map, and only the specific tile of the game map is aware of its own state and the entity that rests atop. Interaction between the entities that rest atop map tiles is performed through tile-to-tile communication. A tile encapsulates the entity that it is associated with, and notifies the entity of any change in its state. On the other hand, the entity also notifies its associated tile of any change in its state. In essence, a bi-directional Observer Design Pattern is used to keep tiles in sync with their associated entities.

To make the interaction as natural as possible, we model objects with their most distinguishing characteristics:

Class	Characteristic
Entity	Spawnable on a game Tile
Character	Movable Entity
Player	Controllable Character
Enemy	Un-controllable Character

Potion	Entity that wraps Player once picked up
Treasure	Entity that modifies Player once stepped on
Stair	Entity that modifies Game state once stepped on
Cell	Objects in a game map
Tile	Cell that spawns Entity
Wall, Door, Passage	Cell that does not spawn Entity, but has other special effects

Characters and Combat System

To model the similarity of player and enemy objects, we created a character superclass that provides a generic implementation of attack and defense methods. Concrete subclasses of player and enemy representing different races override the generic combat methods to account for their special abilities.

As described in the previous section, interactions between entities are initiated by tile-to-tile communication. During player's turn, if an attack action is received from the command interpreter, the current player notifies the corresponding tile that it is attacked. Similarly, during the enemy turns, each tile on board performs specific actions based on its associated entity. Given the current implementation, only the tiles associated with an enemy needs to process the entity's actions. The enemy receiving the notification of its turn to move will query its surrounding tiles and either attack a nearby tile, or randomly move to a nearby tile.

Potion and Treasure System

To implicitly track the status effect of permanent and temporary potions, we make use of the Decorator Design Pattern. A potion inherits all behaviors from a player, but delegates all of the methods to the internal player object. Each potion applies its status effect to the player when it is first wrapped, and reverses its status effect when it is discarded. Treasures, however, are implemented differently. Since there is no need to reverse a status change caused by picking up a treasure, treasures are directly applied to the player without any additional tracking.

Spawning System

To ensure the future customizability of the entity generation for different floors, we delegated the task of entity generation to the floor object. To generate each entity, the floor provides the pointer to a random tile from a random chamber, and then use the static spawning method provided by the specific entity class to place the entity on the given tile.

Design Questions:

1. **[Question]** How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?

   [Answer] Our system will have concrete classes representing each race that extend off an abstract superclass. The superclass will contain fields common to all the races, as well as a static method that returns a certain race based off the user's input. This will

make race generation easy as the mapping between 's' to Shade for example, will be encapsulated in one method. It will also be easy to add additional classes by simply inheriting off the superclass.

2. **[Question]** How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

   [Answer] Our system will use the Factory Design Pattern to generate enemies, as there can be multiple instances of them. The race of the generated enemies is purely probabilistic (with the exception of dragons, which are spawned with treasure piles). On the other hand, the player character will be generated with the Singleton Design Pattern, as there is only one instance of the player object. The race of the player character is also pre-determined by user selection.

3. **[Question]** How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

   [Answer] We will use the Visitor Design Pattern to implement the abilities for both the enemies and player races, basing effects on the attacker and defender. Other effects such as Drow's potion multiplier or Human's gold drop will be implemented through overriding method in their respective classes. Thus the techniques used for both the enemies and the player will be similar in nature.

4. **[Question]** What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

   [Answer] We will use the Decorator Design pattern so that the potion itself will wrap around the original player. Thus, multiple effects can be stacked and implicitly tracked. To remove the temporary status effects, one can simply retrieve the underlying player object by recursively unwrapping the decorators.

5. **[Question]** How could you generate items so that the generation of Treasure and Potions reuses as much code as possible? That is, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code?

   [Answer] The floor generates specific items by selecting a random chamber and a random cell. It, then, delegates the task of item generation to the individual cells. The cells generate specific items with a Factory Design Pattern. Code will be reused, as finding the random cell to spawn an item will be the same for any item.