1.1 Spec Summary:

3.1 Player
- Movement
- Inventory, attack, health

3.2 Static Entities
- Wall
- Exit
- Boulder
- Floor Switch
- Door
- Portal
- Zombie Toast Spawner

3.3 Moving Entities
- Spider
- Zombie Toast
- Mercenary

3.4 Collectable Entities
- Treasure
- Key
- Invincibility Potion
- Invisibility Potion
- Wood
- Arrows
- Bomb
- Sword

3.5 Buildable Entities
- Bow
- Shield

3.6 Battle
- Happens when player and enemy are in the same cell
- If health <= 0
    - player dies
    - be removed from the game
    - game is over
- If enemy <= 0
    - be removed from the game
- if both alive
    - repeat until one is died
- Allies provide an attack and defence bonus to the player
- 3.6.1: Rules

3.7 Goals: what must be achieved
- 3.7.1 Basic Goals: Goals are only evaluated after the 1st tick

        - a. Get to an Exit
        - b. Destroying a certain number of enemies AND spawners
        - c. Having a boulder on all floor switches
        - d. Collect a certain number of treasures.
    - 3.7.2 Complex Goals: sub-goal might be un-achieved. See 3.7.2 last line
        - a&b
        - c&d
        - a&(b||d)

3.8 Win & Lose
    - Win: all goals achieved
    - Lose: player dies and is removed

4. Dungeon Maps
    - Json Files
        - entities: an array of entities when start
        - goal-condition: goals to achieve for winning the game

4.1 Input Specification - Entities
    - entity in entities contains:
        - position x & y
        - type: see 4.1 types table

4.1.1 Extra Fields
    - portal has a filed “colour”, 2 portals with the same colour are linked
    - door and key will have a “key” filed.

4.2 Input - Goals
    - Basic goal: <goal> is one of “enemies”, “boulders”, “treasure”, or “exit” which represent 3.7.1 - 4 basic goals
    - Complex goal:
        - <supergoal>: “AND”, “OR”
        - <subgoals>: contains 2 basic goals

5. Configuration Files
    - Read in these values from the config_template.json when game is created

6. Interface
    - DungeonManiaController class which interacts with a HTTP layer of a web server.
    - 6.1 Interface Data Types
        - response/models: create objects of these types for the controller to return and communicate info to the server.
    - 6.2 Interface Methods
    - 6.3 Exceptions

12.3.1 Task Life Cycle

- For each task/ticket
    - Design:
        - filed and methods to add/change
        - classes and packages to create
    - Review by reporter(other teammate)
    - Test list: all the tests you will write for the ticket
    - Write tests -> Implementaition

14.1 Mark Breakdown

- Player movement
- Interface methods
- Boulders
- Doors & Keys
- Potions
- Portals
- Bombs
- Spiders
- Zombies
- Battles
- Mercenaries
- Basic Goals
- Complex Goals

2.1 Task Breakdown & Sequencing:

- Understand the spec and list all the entities first
- Start doing the UML and map out the relationships between entities
- Finish the UMLwith basic relationships and its cardinality
- Stubs out the classes and function prototypes for the MovementBehaviour interface and figuring out what are the different movement for each movingEntities, which includes Player, Boulder, Mercenary, Zombie and Spider.
- Write failing tests for the basic MovementBehaviour of Player
- Implement the PlayerMovement and finish related fields and methods in Player
- Stubs out the fields and methods for Spider and SpiderMovement
- Write failing tests for the basic SpiderMovement of Spider
- Implement the SpiderMovement and finish related fields and methods in Spider
- Stubs out the classes and method prototypes CollectableEntities, which includes arrows, key, sword, treasure, wood
- Write failing tests for each Collectable entity
- Implement all colleatable entities and add fields and methods as well
- Stubs out the classes and methods prototypes for all StaticEntites
- Write failing tests for each static entity

- Implement all static entities and add fields and methods as well
- Stubs out the classes for BuildableEntity and write failing tests for it, then implement it.
- Implement player states interface
- Implement the Player class and add Inventory to it.
- Stubs out classes for goal and decided to implement the composite pattern(see detailed planning on 4.1)
- Write failing tests for basic goal test: exitGoal, enemiesGoal, treasureGoal, bouldersGoal
- Implement methods for each subgoals and also complex goals (AND, OR)
- Create Dungeon class and add methods into it along we doing other classes
- Implementing the newGame() in DungeonController
- Implementing Battles
- Implement getDungeonResponseModel() in DungeonController
- Implement tick() in DungeonController
- Implement all other methods in DungeonController

3.1 UML Planning:

DungeonManiaController

- Has composition relation with basic goals
- Basic goals composite other goals: exist, enemies, boulder, treasure

Player

- May need to implement observer pattern with Mercenary/ Ally
- Player's Inventory has a composition relationship with player
- May need to implement state pattern when player using potion: invincible, invisible, died, lose

Item equipped

- Item that equipped with user, eg:treasure, wood, arrow, key, potions, sword, bow, shield

Static entity

- Wall, exist, boulder, floor switch, door(state pattern contains closed and open), portal, Zombie Toast Spawner

Enemies

- Zombies (affected if player is invincible)
- Mercenaries
- spider

Player's state (state pattern?)

- Invisible : can move past all other entities undetected
- Invincible : player win during battle
- Died
- Winning : all goal achieved

Movement (strategy pattern?)

- Interface MovementBehaviour implements:
    - PlayerMovement
    - SpiderMovement
    - ZombieMovement
    - MercenaryMovement
    - BoulderMovement

4.1 Planning on Goal Structure:

- Since goal can be basic goals or conjunction/disconjuction goals which represented as AND or OR goal and these will contain a list of subgoals. These subgoals can also hold some basic goal or conjuction/didconjuction goal and so on. Therefore a composite pattern will be suitable to implement this tree structure model.
- We need:
    - A common interface(act as component) called Goal.java which declare method that determine whether the given goal is achieved or not (client use this)
    - basicGoal (act as leaf) implements Goal.java and returns a basic goal which can be one of "enemies", "boulders", "treasure" or "exit". - in our case, we have multiple leaf classes.
    - ComplexGoal (act as composite) which go over each subgoal in the complexGoal(either AND or OR) and see if the goal is achieved based on the order of the goal and return whether the complexGoal is achieved or not.

Dependency Tree w/ Story Points: