Henry LeCates and Ryan Roberts
GRADE ARGUMENT:

Functionality: The program responds to user inputs exceptionally well. There are implemented cooldown timers for dashing and attacking that act as a safeguard against any graphics glitches. In the majority of cases where an object may be null, or a file not read properly, we implemented conditional statements which prevent an action from being done or reporting an error. A common error we encountered was concurrent modification. To avoid this error, we developed a system (a series of lines of code which can be applied to all linked lists) to prevent it. In this system we create a new linked list which we add to all elements to be removed and then remove it after iterating over all original elements (see updateEnemies method in world).

Given more time we would add further error statements, particularly if a wrong file is inputted to the image reader class. Furthermore, to improve the playability of the game, we would continue to finetune the collision systems and provide a more accurate system.

Design: The design of our program is intricate and carefully tracks different interactions between the players and elements of the game. The way in which we designed the interaction between the player, enemies, bosses, areas and environments, was very intricate. Each class has a large amount of variables and concurring events and therefore, needs to be carefully laid out ahead of time. Furthermore, in the way we structured the code, it allows it to be easily expanded. Levels can easily be added using the interface, and the levels[] in World.

Given more time we would make the levels system and how we increment them, handled in the main method of the program. Each level would then be an instance of the world and contain various different elements. In this way we could also potentially incorporate method overloading, or a more basic system of conditionals to modify each instance of the world. The basic idea of a levelIndex and a levels[] would remain the same, however, the variables would be stored and passed from main. Despite this, our system does work, however, we believe it could be made more elegant.

Creativity: Our game goes above and beyond in terms of creativity. We have created original levels and even an original final boss with varying attack phases. The program reflects a passion for the process of making a video game that can be seen in the variation of levels and gameplay content.

Furthermore, our graphics were all hand drawn and imported via png files. Although we may not be the most artistically skilled, we paid careful attention to the details, by making sure they are all the same art style. The style of our graphics attempted to be cartoonish, where each sprite is drawn in a cartoonish way. In addition, we tried to create a medieval theme, with our player being a knight and the title card and buttons on the first screen are modeled after tavern signs.

Sophistication: Our program has many examples of sophistication. It succeeds in tracking the collisions between the player, enemy, and projectiles with high accuracy. However, the main sources of sophistication come from the boss, and the use of user defined and java data structures to create a powerups system, printing actions to a file, and a randomized maze. For the implementation of the boss, we used a complex series of conditionals and interactions between three various classes (Boss, the health bar and the boss timer) to create a simplistic AI which follows and combats the player. The boss also transitions from phases, with each slightly changing the combat. For the user defined data structures, we used a generic queue. The generic queue is used for both printing to a file and the power ups system. For the printing, the queue takes strings and when called will output the actions stored in the queue one at a time. The queue was also used for the powerups system where it allows the player to collect a series of power ups and then use them in the order they were collected. The stack was used to create a randomly generated maze. For this implementation, we first researched various maze generation / solving algorithms, until we found one which was within the bounds of our knowledge. The algorithm uses recursive backtracking of a stack to find valid neighbors. If there are no valid neighbors, the program then backtracks and creates a new pathway.

Broadness: We checked boxes 1, 2, 3, 4, 5, 6, 7, and 8.

For 1) we used libraries which were not discussed. These included import javax.imageio.ImageIO. We used this for the purpose of drawing images for the game. To implement the library we had to research how to create and use the ReadImages class.

For 2) we used subclassing in numerous ways. First all of the enemies, player, and boss, extend a class person, which houses updating, and drawing for each of the sprites. In similar manner, this process is repeated with key and various powerup classes extending Collectables.

For 3) we used an interface for the creation of the levels. This allowed us to create a level[] in the world, despite the levels having different functions and methods. Furthermore, it provides a template for a further expansion of the levels.

For 4) we used built in data structures to house and control the enemies in level one, the projectiles throughout the game and other small implementations. We used a list to assist in randomly selecting a path, and a stack in the implementation of a maze generation algorithm.

For 5) we used a user defined queue to hold both strings for printing the game actions to a file and holding the powerups in the dueling mode.

For 6) we used both file input and output. Input was used when we read the image files, while output was used when we write the game actions to a txt file.

For 7) randomization was used consistently throughout the program. The majority of the areas, and anemone positions are all randomly defined allowing for a different experience each time the program is run. The largest implementation of randomization is in level three, where each time the game is played the node puzzle varies.

For 8) we used generics to create a generic user defined queue. This is because we use the queue to hold both strings and powerups and therefore, instead of repeated code, we have a singular generic queue.

Code Quality: Our code is consistently commented throughout, with each method having a short description and the various non obvious portions of code are explained. These portions include the maze generation, the levels and the node puzzle. Our code is also relatively organized, with the similar classes being grouped together in files and the methods in each class follow the same general structure. Our code is also formatted in the same way throughout, with the spacing being uniform and the variable names following the same structure throughout.