“Nightmare on Strand Campus”
Text-Based Adventure Game Report

Welcome to "Nightmare on Strand Campus," a text-based adventure game that plunges you into the heart of horror and suspense! You find yourself trapped in Strand Campus, haunted by iconic villains such as Jason Voorhees, Freddy Krueger, and the Joker.

The game places players in a suspenseful narrative where they must navigate through a simple map of rooms, each described, and encounter iconic movie personas while unraveling the mystery of why the player is there. Players can do a variety of actions, such as moving between rooms (including going back), grabbing, dropping, and giving items (to non-aggressive characters), and unlocking rooms.
The code features a well-organized base, encapsulating rooms, characters, and player actions within distinct class. Specialized classes like the Game Processor and the Battle Class handle initialization and player interaction with aggressive characters.

Base Tasks:

- The game incorporates a modular design, featuring the room class, which allows for the creation of various rooms. The GameFoundation.txt file is read during initialization, populating the game with 13 rooms in total. These rooms are initialized with a type, a name, a key if they are a lock type, a flag for if they are the transport room, and a description. For this game, 1 room was an instance of the Key Room class and one was a transport room. If they are an instance of Key Room, they are created as such, considering the key to unlock as well.

- Like rooms, items are created through the scan of the .txt file. They are initialized with a type, a name, a weight, a flag if they can be grabbed, an attack stat if an instance of weapon class, and a description. They are then placed in their respective rooms, also given by the .txt file. If they are an instance of weapon, they are created as such, otherwise created like an item, without taking into consideration the attack stat.

- The player’s inventory is implemented with a weight system, with a final max weight of 20. The game ensures players can only carry items up to 20. This is all managed through player class, as it’s the player that will interact with items.

- The winning condition in the game is met when the player enters the Exam Room. In the map, this is the central room, which needs to be unlocked first, before being able to enter. When enter, it will print a win message and inform the player they were successful.

- The logic of the back command is implemented in the player class, as it will be the player moving through rooms. A Stack is created to keep track of the player’s movements.

- Various commands were added. The ‘grab’ and ‘drop’ commands, executed with two words, will allow the player to manipulate items by adding or removing them from their inventory. The three-word command ‘give’ is a three-word player character interactive command, allowing for the exchange of items between the player and characters. Depending on the character, the player will receive different items. The ‘bag’ command is a simple one-word command, which provides a detailed overview of the player’s inventor: what items they have, what is the remaining weight they have to carry items, and what is the player’s attack stat. Lastly, the ‘unlock’ command is a two-word command, that will allow the player to unlock specific rooms if they meet specific criteria, some of those including if they are next to that room, if they have the key, and if the room is a locked room.

Challenge Tasks:

- Within the game foundation file, characters are initialized with an initial room where they are spawned in, as well as a type, a name, flags for if they can move and if they are aggressive, an attack level is their type is monster, and finally a story, which if a monster class object, will be used if they are defeated by the player, otherwise, it is printed after the 'give' command interaction with the player. When the players move, the characters move as well. This is done by accessing the characters current room, getting its exits, and generating a random number to select the room the character will move into. The characters only move when player moves which is a design choice, as very distinct and loosely coupled methods were created, as can be seen in the Game class. There are 5 characters in total, 3 aggressive and 2 friendly. Notably, Laurie Strode is kept stationary, requiring player interaction to obtain the key for unlocking the exam room. This intentional design choice avoids a luck based dynamic, ensuring a more exciting gameplay experience.

- The parser was extended to recognize and interpret three words commands, by simply adding a second string as a third word and adding an additional if condition inside the parser class, which checks if tokenizer has more input after a second word, to then retrieve it as the third word.
  The give command is currently the only command in this version of the game to use three words, but now that the parser has been extended to accept three-word commands, more intricate gameplay scenarios can be created.

- The transport room was implemented by giving rooms a flag for if they are transport rooms or not. This is then checked every time a player enters a room. If the flag for is transport room is true, the teleport method will be called, which creates an array containing all rooms excluding the transport room itself and locked rooms. It will randomly choose one and update the player's current room. While a separate class could have been created, one would argue that using a flag is a simpler but still effective method, as more rooms could be changed to teleport rooms.

Code Quality Considerations

Coupling:

An example where I have used loose coupling is between two of the most important classes of the program, the Player class, and the Game class. Loose coupling can be seen in the way the Player interacts with the Game class throughout the *movePlayer()* method. The Player class doesn't directly reference or depend on the Game Class. It uses the map parameter to interact with rooms directly. This allows the Player class to be more independent and reusable, without being tightly bound to the game class.
Furthermore, in the *goRoom()* method in the game class, where the *movePlayer()* method is used, the game class doesn't need to know the details of how player moves, it delegates that responsibility to the player class, creating a loosely coupled relationship, which is what is desired.

Cohesion:

There are various examples of where I have high cohesion, but to name a great representation that would be the Game Processor class. It was created to process the .txt file, with an additional level of complexity inside the class, where it has specific methods to process rooms, characters, and items, and create them. This specific class with a specific role, which is then divided into even more specific methods, is a great example of high cohesion, improving readability and maintainability of the code.

Responsibility-driven design:

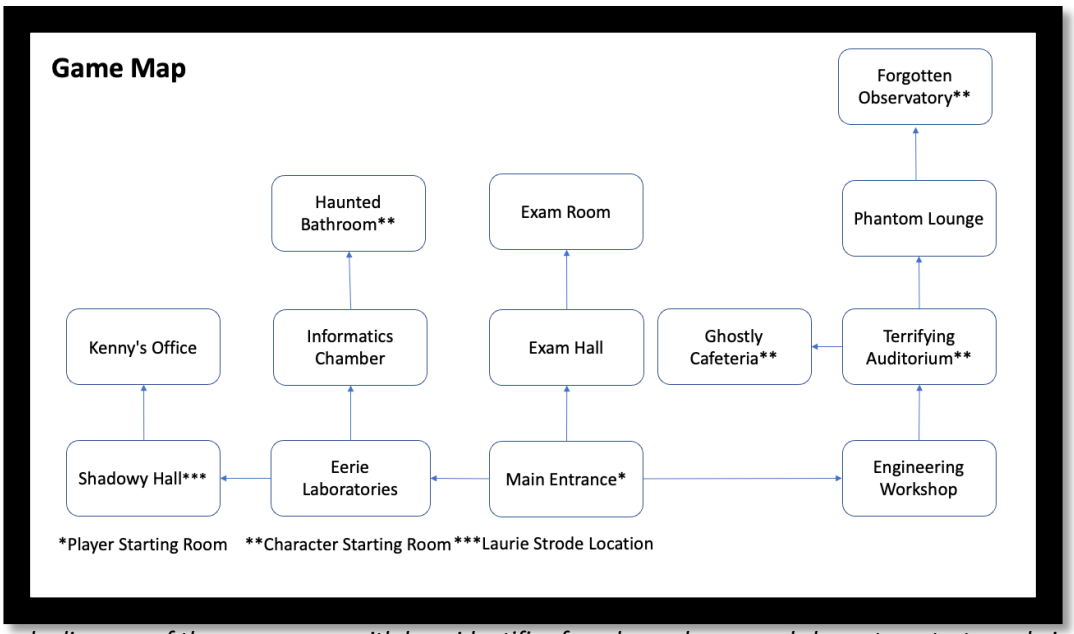
A strong example of RDD in my code is the Room Class. It is divided into 5 clear sections: a section with for the access of the room information, a section for the manipulation of exits, a section for the manipulation of items, a section for the teleport room, and finally a section for extras of the locked rooms. This code is organized into specific groups, with clear names as to what the role of each method is (such as *isItemInRoom* or *isAdjacent*). The code embraces high cohesion where each methods owns a single responsibility, increasing readability and maintainability.

Maintainability:

One noteworthy element of maintainability in my code is the utilization of an enumerated type for command words, as demonstrated by the CommandWord enum. This approach centralizes all valid command words in a single, easily extendable location. By encapsulating these commands within this enum, the code is more sustainable to change, as future modifications are implemented quickly, such as adding, removing, or modifying commands. This design choice not only minimizing the risk of errors but also simplifies the code for other programmers to use it as a base and update it.

Walkthrough:

1. Player will start in the main entrance greeted by an introductory message. An introduction will be printed. There will be an invitation that the player can grab, setting the stage for the unfolding narrative.
2. The player should now explore different rooms using the 'go' command and try and grab as many items as possible using the 'grab' command. There will be powerful weapons in the deepest areas of the map, such as the 'lightsaber', the most powerful weapon in the game.
3. The primary objective is to find the character Laurie Strode and offer her an item, as she holds the key to unlock the final room. While looking for Laurie, the player may encounter aggressive characters, which will cause a fight, potentially resulting in the death of the player. In that scenario, game will print a message and will stop taking inputs from player.
4. Along the way, the character may find one additional nonaggressive character, that being Nancy Thompson. It is recommended to give her an item too, as player will be reward with a weightless totem, that reduces chances of dying in every occurrence by 3%, a worthwhile investment of an item.
5. Once the key has been obtained, the player can now go to the exam hall, the only room adjacent to the exam room, and use the command 'unlock examRoom', printing a message telling the player so.
6. The player now uses the 'go north' command to enter the exam room, which will print a message explaining how player was successful, and game will stop taking input.



Simple diagram of the game map, with keys identifies for where players and characters start, made in PPT.