Group members:

- Vi Nguyen - 501240732
- Leo Lai - 500414796
- Jeffry Lien - 500638398
- Alireza Iman - 500901122
- Viri Nguyen - 501132181

Design Patterns Used:

1. Creational Patterns
    - **Factory Method Pattern**
        - **Where it's used:** In pretty much all classes that were created, there is a method that reads and parses data from json files and instantiating objects dynamically. For example, Items.load_items, Actions.load_actions, etc.
        - **Why it's used:** These methods centralize object creation, making it easier to modify how `Item` and `Action` objects are constructed without changing the rest of the codebase.

    - **Singleton Pattern**
        - **Where it's used:** In the main game logic file, there is a global variable called 'current_room' that acts as a singleton for tracking the player's current location.
        - **Why it's used:** This usage although does not fully demonstrate the full picture of Singleton Pattern, it's still following the principle of a single source of truth for the current state.

2. Behavioral Patterns
    - **Strategy Pattern**
        - **Where it's used:** All the rooms' logic like home_logic, bakery_logic, restaurant_logic, etc. Each room's logic is encapsulated in a dedicated module of functions, combining with the logic to move_rooms, allowing the main game logic to dynamically decide which strategy (room-specific logic) to execute when the player enters a room.
        - **Why it's used:** Simplifies room-specific interactions, making it easier to extend the game with new rooms.

    - **Command Pattern**

        - **Where it's used:** When handling a user's input as command to behave a variety of actions: moving between rooms, solving puzzles, performing specific actions to prompt more interaction. Each of these commands is associated with their own logic in various functions such as move_player, process_home_command, etc.
        - **Why it's used:** Detaches input handling from game logic, simplifying the addition of new commands.

    - **Template Method Pattern**
        - **Where it's used:** Room-specific logic (e.g., start_home_room, start_flower_shop) acts as a high-level template defines the general flow (e.g., displaying the room, processing commands), while room-specific logic handles details.
        - **Why it's used:** Reduces code duplication and ensures consistency across rooms.

    - **Observer Pattern**
        - **Where it's used:** The game's progression depends on the player's actions (e.g., collecting items or helping characters).
        - **Why it's used:** Events like solving puzzles or fulfilling quests could notify other components of changes in game state.

3. Single Responsibility Principle (SRP)
    - **Where it's used:** Classes such as Rooms, Items, Actions, etc. Each class has a well-defined responsibility, such as managing item properties, game actions, or room data.
    - **Why it's used:** Improves code readability, maintainability, and testability.

4. Don't Repeat Yourself (DRY) Principle
    - **Where it's used:** The modular structure of classes like Item, Action, and room logic functions avoids redundancy by encapsulating reusable logic.
    - **Why it's used:** This principle aligns with general clean-code practices, preventing duplication and ensuring maintainability.

Anti-Patterns Encountered and Refactoring:

1. **Spaghetti Code**

- **Anti-Pattern:** In the early stages of building the code base, the game logic was scattered across multiple files, leading to tight attachment and difficulty in navigation. It was also hard when updating logic and oftentimes the update was inconsistent as some logic was forgotten nested somewhere in the code without clear structure or indicator of their roles..
- **Refactoring:** Modularized the codebase into logical units (Modules and game_engine), grouping related functionality into cohesive classes and files.

2. **Hardcoding Values**
    - **Anti-Pattern:** Initial versions hardcoded items, actions, and room descriptions directly in code, which worked, but it became super redundant and heavily violated clean-code principle.
    - **Refactoring:** Moved game data into JSON data files (rooms.json, items.json, actions.json), enabling easier updates and scalability.

3. **One-object-solve-all**
    - **Anti-Pattern:** Before breaking game logic into smaller functional components, one single file (main.py) tried to manage too many responsibilities, such as loading data, processing commands, and handling room transitions.
    - **Refactoring:** Delegated responsibilities to specific modules and classes (Item, Action, and individual room logic modules), then just import and use them in main file instead.

4. **Code Duplication**
    - **Anti-Pattern:** Repeated code for parsing JSON files and creating objects for Item's, Room's and Action's methods.
    - **Refactoring:** Extracted common logic into factory methods (load_items and load_actions).

5. **Features Heavy**
    - **Anti-Pattern:** Some methods relied too heavily on data from other classes, such as handling room transitions directly in main.py, which made main.py receive too many imported separated data in order to perform certain logic.
    - **Refactoring:** Encapsulated room-related logic in the Room class and delegated specific interactions to room-specific modules.