2021315385

이건 Gun Daniel Lee

Programming Assignment 5 Report

This report will thoroughly discuss and explain the contents of the text-based console application, "Adventurer's Guild". The report will divide the explanation into 3 main parts, system's general flow, JSON schema and storage, and the implementation of required programming techniques.

1. System's General Flow

The application is all designed into one Python file named "guild.py". The general flow begins with initialization, where the system calls the 'load()' function. This function attempts to open the JSON files; if a file is missing, it automatically creates fresh files with empty structures into two global dictionaries, users and quests. The 'users' dictionary stores all the user information and 'quests' stores all the quest information. If 'quests' is an empty dictionary, the program will load the preset quest data and save the quests into the JSON file right away using the 'save()' function.

Once the initialization succeeds, the program enters the 'main()' function, which triggers the entire program. The main function uses a while loop to continually trigger the initial screen until the user selects to register, login, or exit the program. This portion represents the authentication of the program. If the user chooses to register, the 'registration() function will run. This function also uses a while loop to continuously receive inputs from the user until the inputs are valid, or the user chooses to cancel the registration. The helper functions 'is_username_taken()' and 'password_check()' are used to validate if the username is unique and the password follows the specified rules. Once the username, password, and pin are validated, the helper function 'user_profile()' will be called to create a user profile dictionary with default values. The 'user_profile()' function also uses two separate helper functions 'generate_unique_id()' and 'hash()' which, as the name suggests, generates a unique 5-digit ID and hashes both the password and pin. This information will be updated in the 'users' dictionary, then the updated 'users' dictionary will be saved into the JSON file using the 'save' function. On the other hand, when the user chooses to login, the 'login()' function will be triggered. This function authenticates the user by finding the username instance in the 'users' dictionary and comparing the hash of the inputted password and stored password.

Upon successful authentication, the 'main()' function will call the 'main_menu()' function which contains the main program logic. This function takes the username as the parameter and constantly displays the user's key information, including username, ID, rank, and stamina. In this function, the user can choose between a couple actions such as checking history, accepting or submitting quests, training, and logging out.

The 'history()' function receives the username and accesses the history, which is stored as a list, where actions are stored chronologically. This is achieved by using the 'log_history()' function. This helper function formulates a history entry and appends it to the end of the history list, which ensures chronological order of the list. The functionality of it will be discussed in detail as it implements one of the required programming techniques. Since the action history is stored chronologically, all the 'history()' function does is to iterate through the history list in reverse order using a for loop, and display the information based on the 4 action types, which are train, submit, accept, and rest.

The 'accept_quest()' function manages the Quest Board. It iterates through the 'quests' dictionary to display all the available quests. Once the user inputs a quest ID, this function thoroughly validates several rules, including user PIN is valid, quest exists, quest not accepted yet, skill requirements met, and not past due date. Once the quest is accepted, both the quest and user's accepted quest will be updated. Then, this action will be logged using 'log_history()', and the updated values will be saved into both JSON files.

The 'submit_quest()' function handles the completion logic. It first checks whether the quest ID input was accepted by the current user and validates the user using the PIN. Once valid, the function will print the list of proof items required to submit the quest. Then, the user will be able to input each proof item using a while loop until a blank input is read, which breaks out of the loop. Then, the function uses the 'validate_proofs()' function to verify the proof items and checks if the user has enough stamina to submit the quest (10). When the user successfully submits the quest, the user and quest information are all updated as needed. The user will get all the benefits of submitting the quest, the quest will clear its 'accepted' value, and this action will be logged using the 'log_history()' function. Once all of these are achieved, the updated 'users' and 'quests' dictionaries will be saved into their respective JSON files.

The 'train_skill()' function handles character skills progression and resting. The function receives the skill name, minutes, and PIN as inputs. Stamina cost and gain will be calculated using the Train class. Then, it creates or reuses a training session (closure) for the user by calling the 'training()' function. These training sessions are stored in a global dictionary

named 'active' to ensure the logic for diminishing returns for each user. The closure function will be explained in detail during the discussion of implementation. For this program, due to the lack of stamina, a resting option was added to the skills. The stamina, skill level, and action history will all be updated using the 'log_history()' function then saved to the user's JSON file.

2. JSON Schema and Storage

The system uses two primary JSON files for storage, 'adventurers.json' and 'quests.json'. As shown in the explanation above, these JSON files are updated after every significant change to either the 'users' or 'quests' dictionaries. Every state-changing operation (e.g., register, accept quest, submit quest, train) triggers the 'save()' function to immediately update the JSON files while read-only operations like viewing history do not.

All user-specific data and states are stored in 'adventurers.json'. The keys are the usernames, and the values are the schema. The schema includes:

- ID: A unique, randomly generated 5-digit number
- Password: hashed password value
- PIN: hashed PIN value
- Exp: integer value of user's experience points
- Rank: User's current rank (e.g., BRONZE)
- Stamina: an integer value of stamina
- Skills: dictionary including the user's skill points
- Inventory: a dictionary of user's inventory (mainly coin pouch)
- Accepted: List of accepted quest IDs (strings)
- Completed: List of completed quests objects (dictionaries)
- History: List of user actions listed chronologically. Each entry is a dictionary containing the type and data, and context-specific keys (e.g., exp, skill, loot, etc.)

All quest-specific data are stored in 'quests.json'. The keys are the quest IDs, and the values are the schema. The schema includes:

- Title: type of quest in string
- Difficulty: difficulty level in string (e.g., EASY)
- Require: nested dictionary defining the required skill and its minimum level
- due_date: due date of the quest (YYYY-MM-DD)

- rewards: object containing exp (int), fame (int), and loot (dictionary of item name and counts)
- required_proofs: a list of proof items needed to submit quest
- accepted: a string that shows the user that currently accepted the quest

3. Implementation of Required Programming Techniques

A. Parameter Binding

The system implements flexible parameter binding to handle input validation and logging efficiently.

Keyword arguments (**kwargs) were used in the 'log_history()' function to accept arbitrary data points depending on the action type. This functionality was necessary since each action type receives a different set of values. The 'log_history()' function takes all these keyword arguments and creates a dictionary entry to be added to the user's history list. The function also iterates through the arguments to detect reserved keys (e.g., date, action type), printing a binding error if detected.

Variable positional arguments (*args) were used in the 'validate_proofs()' function. The *args was used to accept a variable list of proof items provided by the user. It validates that no duplicate submissions exist in the input and that the items match the `required` proof items, printing a binding error if detected.

To show that the binding error works, the possible binding errors were added to the end of the code, printing the binding errors before exiting the program as shown below. These codes will be deleted in the final code submission.

```
main()
# attempt to show binding error
log_history("test_user", type="Broken", date="2025-01-01")
log_history("test_user", date="2025-01-01")
validate_proofs("hello", "hello", required="hello")
```

```
=== Adventurer's Guild ===
[1] Register    [2] Login       [0] Exit
Select: 0
Bye!
Binding error: 'type' is a reserved key and cannot be used in history logging.
Binding error: 'date' is a reserved key and cannot be used in history logging.
Binding error: Duplicate proof names detected.
PS C:\Users\samsung\Desktop\SKKU\Programming-Languages\Assignment\Assignment5-PL>
```

Figures 1 & 2: Deliberate binding errors caused before exiting the program

B. Closure

To simulate the diminishing returns during training, the system uses a closure. The 'training()' function initializes a local state with a base exp of 10 and a diminishing factor. It returns the inner 'train' function that modifies the state using the 'nonlocal' keyword.

To ensure that the state persists across several trainings, the closure instance of a user will be stored into the global dictionary named 'active'. This ensures that the diminishing returns will continue and that each user will have a different training return. As shown in the figure below, training 2 times had an exp gain of 10 and 8, respectively. This is because the diminishing factor is set to 0.8.

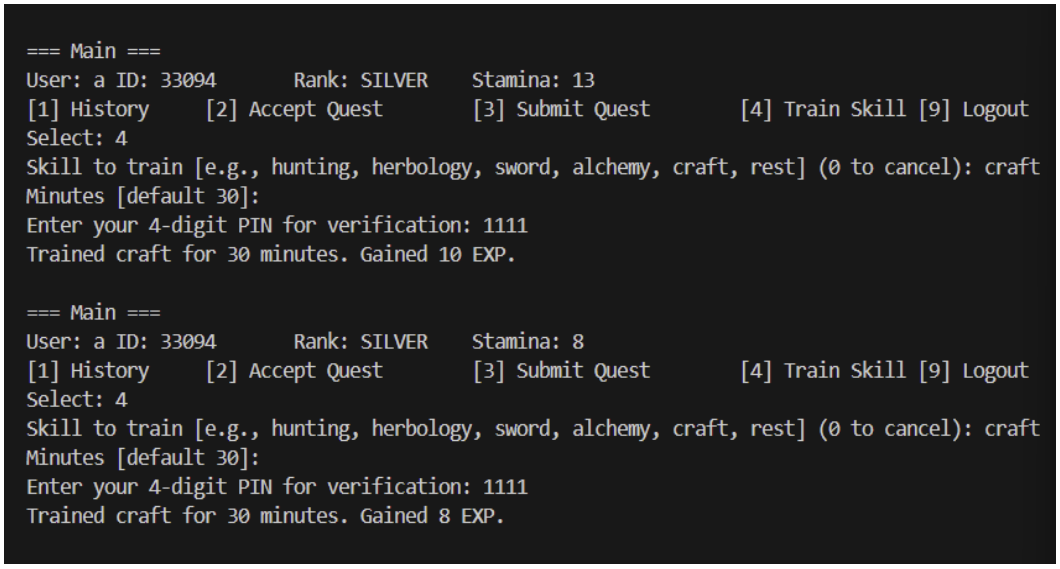

Figure 3: Diminishing return of continuous training

C. User-Defined Overloaded Operators

Three classes were created to create 3 different overloaded operators.

The Rank class overloads the greater than or equal to (>=) operator. This operator was overloaded to directly compare a user's exp to the rank objects. The string operator 'str()' was also overloaded to update a person's rank based on the exp they have using the overloaded >= operator. This update occurs in the 'submit_quest()' function as this is the only function that gains exp. Once the quest is submitted, the str() will be called on the user's exp and the rank will be updated correspondingly. For example, if the user has 50 exp, they're rank will become SILVER.

```
=== Main ===
User: a ID: 33094       Rank: BRONZE    Stamina: 50
[1] History     [2] Accept Quest        [3] Submit Quest        [4] Train Skill [9] Logout
Select: 3
Quest to submit (0 to cancel): Q007
Enter your 4-digit PIN for verification: 1111
Provide proof item:  herb_bundle
Add proof item (blank to stop): herb_bundle
Add proof item (blank to stop):
Submit OK. You gained EXP: 12, Fame: 1, Loot: {'coin_pouch': 3}
Congratulations!!! You have been promoted to the rank SILVER
```

Figure 4: Rank Promotion caused by submitting quest

The Inventory class overloaded the "+" operator to merge loot directories. In the 'submit_quest()' function, when a user submits a quest correctly, the corresponding loots are added to the user's inventory. The actual code in the function is "profile['inventory'] = current_inventory + quest['rewards']['loot']". When the two dictionaries are added, the class will look through the 'inventory' dictionary and check whether the item already exists. If the item exists, the item count will increase, and if not, a new key and value for the corresponding item will be added to the inventory. The figures below show the before and after results of the user after submitting the quest in figure 4.


```
"pin": "0ffe1abd1a08215353c233
"rank": "BRONZE",
"exp": 49,
"fame": 14,
"stamina": 50,
"skills": {
    "hunting": 31,
    "herbology": 5,
    "sword": 47,
    "alchemy": 5,
    "craft": 41
},
"inventory": {
    "coin_pouch": 10
},
```

```
"exp": 61,
"fame": 15,
"stamina": 40,
"skills": {
    "hunting": 31,
    "herbology": 5,
    "sword": 47,
    "alchemy": 5,
    "craft": 41
},
"inventory": {
    "coin_pouch": 13
},
```


Figure 5 & 6: Before and After of the Submission in Figure 4

The Train class overloads the "+" and "-" operators to calculate stamina recovery or costs, respectively. These operators will receives the time in minutes, and a self-defined stamina gain or cost formula will be applied. These values will be used in the 'train_skill()' function to determine how much stamina will be wasted based on the time. Figure 7 shows how the user stamina decreases by 5, when training for 30 minutes, which is the default cost of training. While resting for 50 minutes had a stamina gain of 10.

```
=== Main ===
User: a ID: 33094       Rank: SILVER    Stamina: 33
[1] History     [2] Accept Quest        [3] Submit Quest        [4] Train Skill [9] Logout
Select: 4
Skill to train [e.g., hunting, herbology, sword, alchemy, craft, rest] (0 to cancel): craft
Minutes [default 30]:
Enter your 4-digit PIN for verification: 1111
Trained craft for 30 minutes. Gained 6 EXP.

=== Main ===
User: a ID: 33094       Rank: SILVER    Stamina: 28
[1] History     [2] Accept Quest        [3] Submit Quest        [4] Train Skill [9] Logout
Select: 4
Skill to train [e.g., hunting, herbology, sword, alchemy, craft, rest] (0 to cancel): rest
Minutes [default 30]:50
Rested for 50 minutes. Stamina + 10
```

Figure 7: Results of Stamina After Training

4. Demonstration Checklist

For the demonstration checklist, all JSON files were started fresh in order to show the functionality even with initialization.

- Registration of 2 users

```
=== Adventurer's Guild ===
[1] Register    [2] Login       [0] Exit
Select: 1

--- Register ---
Username (0 to cancel): a
Password (>=8, 1 uppercase, 1 special) (0 to cancel): Aaaaaaa!
4-digit PIN (0 to cancel): 1111
Registration successful! Welcome, a. Your Adventurer ID is 81755.

=== Adventurer's Guild ===
[1] Register    [2] Login       [0] Exit
Select: 1

--- Register ---
Username (0 to cancel): b
Password (>=8, 1 uppercase, 1 special) (0 to cancel): Bbbbbbb!
4-digit PIN (0 to cancel): 1111
Registration successful! Welcome, b. Your Adventurer ID is 75952.
```

- Accept and submit quest (with proofs) and rewards applied:
  - Rewards as seen in figure: 12 exp, 1 fame, 3 coin pouch

```
=== Adventurer's Guild ===
[1] Register    [2] Login       [0] Exit
Select: 2

--- Login ---
Username (0 to cancel): a
Password: Aaaaaaa!
Login successful! Welcome back, a.


=== Main ===
User: a ID: 81755       Rank: BRONZE    Stamina: 100
[1] History     [2] Accept Quest        [3] Submit Quest        [4] Train Skill [9] Logout
Select: 2

=== Quest Board ===
- Q007: 'Herb Gathering'    Difficulty: EASY   Req: herbology>=5 Due: 2025-11-28
- Q027: 'Boar Hunting'    Difficulty: EASY   Req: hunting>=10 Due: 2025-11-29
- Q030: 'Escort Caravan'    Difficulty: HARD   Req: sword>=15 Due: 2025-11-30
- Q033: 'Slime Nest Cleanup'    Difficulty: EASY   Req: alchemy>=2 Due: 2025-11-27
- Q010: 'Bridge Repair'    Difficulty: MEDIUM   Req: craft>=10 Due: 2025-11-28
Enter Quest ID to accept (0 to cancel): Q007
Enter your 4-digit PIN for verification: 1111
Priority (NORMAL / HIGH) [default: NORMAL]:

=== Main ===
User: a ID: 81755       Rank: BRONZE    Stamina: 100
[1] History     [2] Accept Quest        [3] Submit Quest        [4] Train Skill [9] Logout
Select: 3
Quest to submit (0 to cancel): Q007
Enter your 4-digit PIN for verification: 1111
Provide proof item:  herb_bundle
Add proof item (blank to stop): herb_bundle
Add proof item (blank to stop):
Submit OK. You gained EXP: 12, Fame: 1, Loot: {'coin_pouch': 3}
```

- Train, with stamina change

```
=== Main ===
User: b ID: 75952       Rank: BRONZE    Stamina: 100
[1] History     [2] Accept Quest        [3] Submit Quest        [4] Train Skill [9] Logout
Select: 4
Skill to train [e.g., hunting, herbology, sword, alchemy, craft, rest] (0 to cancel): hunting
Minutes [default 30]:60
Enter your 4-digit PIN for verification: 1111
Trained hunting for 60 minutes. Gained 18 EXP.

=== Main ===
User: b ID: 75952       Rank: BRONZE    Stamina: 93
[1] History     [2] Accept Quest        [3] Submit Quest        [4] Train Skill [9] Logout
Select: 1
```

- History shows actions in reverse chronological order

```
=== Main ===
User: b ID: 75952       Rank: BRONZE    Stamina: 83
[1] History     [2] Accept Quest        [3] Submit Quest        [4] Train Skill [9] Logout
Select: 1

--- History (latest first) ---
[2025-11-28T00:48:44.280093] Submit     Q027 EXP:20 Fame:2 Loot:{'coin_pouch': 5}
[2025-11-28T00:48:33.268238] Accept     Q027 difficulty=EASY priority=NORMAL
[2025-11-28T00:48:20.600711] Train      hunting +18 (60m)

=== Main ===
```

The checklist for the 3 programming techniques are given from figures 1 – 7 in the implementation section of this report.