Chem274B Final Project – DeliveryII

Software Engineering Reflection

Name: Dongwan Kim

Group: Group 6

Team Members: Trinity Ho, Priscilla Vaskez, Dongwan Kim

1. Description of My Role in the Project

My role in this project evolved across all four levels of the banking system implementation.

In Level 1, I implemented the deposit method. This function checks whether an account exists and updates the account balance accordingly. Although this was a relatively simple task, it was an important foundational component for the entire system.

In Level 2, I implemented the majority of the functionality. I designed and implemented the `top_spenders` method, which ranks accounts based on their total outgoing transactions. To support this feature, I added outgoing transaction tracking and integrated it into the transfer and pay operations so that all spending activity was accurately recorded across the system.

In Level 3, the core payment and cashback functionality was primarily implemented by my teammates. After completing the Level 4 implementation, I revisited the Level 3 features to ensure they continued to work correctly within the finalized system. My role focused on verifying that the payment and cashback logic remained consistent with deposits, transfers, balance updates, and account merging, and fixing any issues that arose during final integration.

In Level 4, I took responsibility for final integration and completion of the system. I debugged complex edge cases related to account merging, particularly scenarios involving queries before and after merge timestamps. I added comprehensive docstrings to all methods to improve readability and maintainability. I also created the project README to document system structure and usage, and ensured that all test cases passed by resolving various bugs and edge cases.

2. Contribution to Successful Project Completion

Our team worked collaboratively, with all members sharing responsibility for meetings, coordination, and implementation. We communicated frequently and supported one another when encountering difficulties.

My personal contributions were substantial throughout the project. I implemented the `deposit` method in Level 1, completed the Level 2 functionality including `top_spenders` and outgoing transaction tracking, assisted with integration of Level 3 post Level4, and handled the final debugging, documentation, and test stabilization in Level 4.

In addition to coding, I helped others understand edge cases, reviewed logic during integration, and ensured the final system met the project specifications. These efforts contributed directly to the successful completion of the project.

3. Challenges and How They Were Solved

The most challenging aspect of the project was handling edge cases in Level 4, particularly those involving merged accounts. Once an account is merged, new operations on the merged account must return None, while historical queries prior to the merge must still work correctly. Ensuring correct behavior across different timestamps required careful reasoning and extensive testing.

Another major challenge was refactoring and code organization as the project grew in complexity. Several methods became difficult to read and maintain. To address this, I introduced helper functions such as `_is_merged_account` to centralize logic and reduce repetition. I also simplified overly complex code paths, removed unnecessary checks, and replaced confusing logic with clearer alternatives while preserving functionality.

Maintaining code readability was also difficult under time pressure. To solve this, I added detailed docstrings to every method and created a README explaining system design and usage. Because refactoring can easily introduce bugs, I tested the system carefully after each change to ensure all existing tests continued to pass.

4. Algorithmic and Performance Analysis of Each Method

- **deposit**

  The deposit method runs in O(1) time and space complexity. It performs a constant-time dictionary lookup and update, making it highly efficient.

- **top_spenders**

  This method uses bubble sort on account–outgoing pairs, resulting in $O(n^2)$ time complexity and O(n) space complexity, where n is the number of accounts. While not optimal for large datasets, bubble sort was chosen for simplicity and clarity. A more efficient alternative would be Python's built-in sorted() function, which runs in O(n log n) time.

- **pay**

The pay method runs in $O(1)$ time complexity since it relies on constant-time dictionary operations.

- **_process_cashback**

  This helper method processes pending cashback events and runs in $O(p)$ time, where p is the number of pending payments. Since it is called at the beginning of operations and the number of pending payments is typically small, this approach is acceptable.

- **get_balance**

  This method uses binary search over a sorted balance history, resulting in $O(\log m)$ time complexity, where m is the number of balance records for the account. This allows efficient balance queries even with many transactions.

- **merge_accounts**

  The merge operation runs in $O(m_1 + m_2 + \log(m_1 + m_2))$ time, where m_1 and m_2 are the balance history sizes of the two accounts. Histories are merged linearly and sorted once, enabling fast future queries.

- **_resolve (alias resolution)**

  Alias resolution runs in $O(k)$ time, where k is the length of the alias chain. Since alias chains are typically short, this operation remains efficient in practice.

5. What Could Have Been Done Differently

Software Project Management

One area for improvement is project management. We tested each level independently, which caused integration issues when combining features across levels. Introducing continuous integration testing earlier would have helped detect these issues sooner.

Regular code reviews after each level could have reduced complexity and improved consistency. Additionally, we occasionally had multiple versions of files, which made tracking changes harder. More disciplined use of git branches and clearer commit messages would have improved workflow organization.

Documentation was added near the end of the project. Writing docstrings and comments incrementally during development would have saved time and improved clarity earlier.

Final Product Improvements

From a performance perspective, `top_spenders` could be optimized by replacing bubble sort with Python's built-in sorting, reducing time complexity from $O(n^2)$ to $O(n \log n)$.

Error handling could also be improved. Instead of returning None for all error cases, custom exception classes could provide clearer error reporting and easier debugging.

Some methods grew large and handled multiple responsibilities. Extracting additional helper functions would further improve readability and maintainability. Adding more unit tests for edge cases beyond the provided test suite would also strengthen reliability.

Finally, while basic type hints are present, runtime validation for inputs such as positive transaction amounts and valid timestamps could improve robustness.