Master of Molecular Science and Software Engineering

Final Project

CHEM 247B – Software Engineering Fundamentals for Molecular Science

DUE Date: December 13, 2024 by 11:59 PM Pacific

Group Assignment (3-4 students per group)

The main goal of this assignment is to design input, output, and processing components for an application developed as part of cross-functional teams. This assignment explicitly tests your abilities in code implementation, data structures & data processing, refactoring & encapsulation, problem solving, and software design patterns. The coding in this assignment involves refactoring and adapting to new requirements, without any highly artificial scenarios. You'll be asked to implement a simple coding project from a spec, broken into four levels.

This assignment includes group deliverables and individual deliverables. As a whole, the final project constitutes 25% of your grade. 80% of the final project grade (20% of final course grade) consists of the group deliverables. 20% of the final project grade (5% of final course grade) consists of the individual deliverables. Regardless of a deliverable's status as a group or individual deliverable, all students should submit their own files for each deliverables under the final project assignment tab in BCourse.

Each student should submit their final code for grading to each level in Gradescope. Note that there are separate submissions for each level. Students should submit their responses as part of the final project under the final project assignment in BCourse. Format the name of your PDF based on the following: "<Berkeley ID>-Answers-Final.pdf".

Document your programs well (e.g., using meaningful names for variables and functions, file names, relevant documentation). Make sure your computer programs are readable (i.e., exercise writing computer programs with code readability in mind). If a problem requests that you provide any programming files, place them all within a zipped folder containing all requested scripts for this problem assignment. Title the folder with the following format: <Berkeley ID>-Answers-Final-code and upload it to BCourses. If you use external code, e.g., StackOverflow (LLM outputs not allowed), please cite the source and provide rationale to prove you understand the code and that it is correct.

Your task is to implement a simplified version of a banking system. All operations that should be supported are listed below. Solving this task consists of several levels. Your code will need to pass all tests in a given level to move forward to the next level. The later levels will take more time to complete than the earlier ones. Note that subsequent levels may require modifying functionality implemented as part of a prior level. Your group will need to plan a design according to the level specifications below:

- Level 1: The banking system should support creating new accounts, depositing money into accounts, and transferring money between two accounts.
- Level 2: The banking system should support ranking accounts based on outgoing transactions.
- Level 3: The banking system should allow scheduling payments with cashback and checking the status of scheduled payments.

- Level 4: The banking system should support merging two accounts while retaining both accounts' balance and transaction histories.

  Note. All operations will have a **timestamp** parameter — a stringified timestamp in milliseconds. It is guaranteed that all timestamps are unique and are in a range from **1** to $\mathbf{10^9}$. Operations will be given in order of strictly increasing timestamps.

Your team is given starter code to frame the problem implementation. The file banking_system.py contains the boilerplat functionality that you will implement in the banking_system_impl.py file. Your team will implement your design and functionality in the banking_system_impl.py file. You should not need to modify any other files. You are free to add any additional helper function or class implementations. Your final submission will be to the corresponding final project entry on Gradescope. To aid in your implementation. All Gradescope tests are available to you in the "test" folder within the starter_code. There are separate test files for each level (one through four). You are provided with bash scripts to help with running the full suite of unit tests for a given test file, though you may run them via Python or however you want. You can also execute a single test case by running the following command in the terminal: **bash run_single_test.sh "<test_case_name>"**

Level 1

Initially, the banking system does not contain any accounts, so implement operations to allow account creation, deposits, and transfers between 2 different accounts.

- **create_account(self, timestamp: int, account_id: str) -> bool** — should create a new account with the given identifier if it doesn't already exist. Returns **True** if the account was successfully created or **False** if an account with **account_id** already exists.
- **deposit(self, timestamp: int, account_id: str, amount: int) -> int | None** — should deposit the given **amount** of money to the specified account **account_id**. Returns the balance of the account after the operation has been processed. If the specified account doesn't exist, should return **None**.
- **transfer(self, timestamp: int, source_account_id: str, target_account_id: str, amount: int) -> int | None** — should transfer the given amount of money from account **source_account_id** to account **target_account_id**. Returns the balance of **source_account_id** if the transfer was successful or **None** otherwise.
    - Returns **None** if **source_account_id** or **target_account_id** doesn't exist.
    - Returns **None** if **source_account_id** and **target_account_id** are the same.
    - Returns **None** if account **source_account_id** has insufficient funds to perform the transfer.

Level 1 Examples

The example below shows how these operations should work (see test cases for additional examples:

Queries	Explanations
create_account(1, "account1")	**returns True**
create_account(2, "account1")	**returns False; this account already exists**
create_account(3, "account2")	**returns True**
deposit(4, "non-existing", 2700)	**returns None**
deposit(5, "account1", 2700)	**returns 2700**
transfer(6, "account1", "account2", 2701)	**returns None; this account has insufficient funds for the transfer**
transfer(7, "account1", "account2", 200)	**returns 2500**

Level 2

The bank wants to identify people who are not keeping money in their accounts, so implement operations to support ranking accounts based on outgoing transactions.

- **top_spenders(self, timestamp: int, n: int) -> list[str]** — should return the identifiers of the top **n** accounts with the highest outgoing transactions - the total amount of money either transferred out of or paid/withdrawn (the **pay** operation will be introduced in level 3) - sorted in descending order, or in case of a tie, sorted alphabetically by **account_id** in ascending order. The result should be a list of strings in the following format: **["<account_id_1>(<total_outgoing_1>)", "<account_id_2>(<total_outgoing_2>)", ..., "<account_id_n>(<total_outgoing_n>)"]**.
    - If less than **n** accounts exist in the system, then return all their identifiers (in the described format).
    - Cashback (an operation that will be introduced in level 3) should not be reflected in the calculations for total outgoing transactions.

Level 2 Examples

The example below shows how these operations should work (see test cases for additional examples):

Queries	Explanations
create_account(1, "account3")	**returns True**
create_account(2, "account2")	**returns True**
create_account(3, "account1")	**returns True**
deposit(4, "account1", 2000)	**returns 2000**
deposit(5, "account2", 3000)	**returns 3000**
deposit(6, "account3", 4000)	**returns 4000**
top_spenders(7, 3)	**returns ["account1(0)", "account2(0)", "account3(0)"];**
transfer(8, "account3", "account2", 500)	**returns 3500**
transfer(9, "account3", "account1", 1000)	**returns 2500**
transfer(10, "account1", "account2", 2500)	**returns 500**
top_spenders(11, 3)	**returns ["account1(2500)", "account3(1500)", "account2(0)"]**

Level 3

The banking system should allow scheduling payments with some cashback and checking the status of scheduled payments.

- **pay(self, timestamp: int, account_id: str, amount: int) -> str | None** — should withdraw the given amount of money from the specified account. All withdraw transactions provide a 2% cashback - 2% of the withdrawn amount (rounded down to the nearest integer) will be refunded to the account 24 hours after the withdrawal. If the withdrawal is successful (i.e., the account holds sufficient funds to withdraw the given amount), returns a string with a unique identifier for the payment transaction in this format: **"payment[ordinal number of withdraws from all accounts]"** - e.g., **"payment1"**, **"payment2"**, etc. Additional conditions:
    - Returns **None** if **account_id** doesn't exist.
    - Returns **None** if **account_id** has insufficient funds to perform the payment.
    - **top_spenders** should now also account for the total amount of money withdrawn from accounts.
    - The waiting period for cashback is 24 hours, equal to **24 * 60 * 60 * 1000 = 86400000** milliseconds (the unit for timestamps). So, cashback will be processed at timestamp **timestamp + 86400000**.
    - When it's time to process cashback for a withdrawal, the amount must be refunded to the account before any other transactions are performed at the relevant timestamp.

- **get_payment_status(self, timestamp: int, account_id: str, payment: str) -> str | None** — should return the status of the payment transaction for the given **payment**. Specifically:
    - Returns **None** if **account_id** doesn't exist.
    - Returns **None** if the given **payment** doesn't exist for the specified account.
    - Returns **None** if the payment transaction was for an account with a different identifier from **account_id**.
    - Returns a string representing the payment status: **"IN_PROGRESS"** or **"CASHBACK_RECEIVED"**.

Level 3 Examples

The example below shows how these operations should work (see test cases for additional examples):

Queries	Explanations
create_account(1, "account1")	returns True
create_account(2, "account2")	returns True
deposit(3, "account1", 2000)	returns 2000
pay(4, "account1", 1000)	return "payment1"
pay(100, "account1", 1000)	return "payment2"
get_payment_status(101, "non-existing", "payment1")	returns None; this account does not exist
get_payment_status(102, "account2", "payment1")	returns None; this payment was from another account
get_payment_status(103, "account1", "payment1")	returns "IN_PROGRESS"
top_spenders(104, 2)	returns ["account1(2000)", "account2(0)"]
deposit(3 + MILLISECONDS_IN_1_DAY, "account1", 100)	returns 100; cashback for "payment1" was not refunded yet
get_payment_status(4 + MILLISECONDS_IN_1_DAY, "account1", "payment1")	returns "CASHBACK_RECEIVED"
deposit(5 + MILLISECONDS_IN_1_DAY, "account1", 100)	returns 220; cashback of 20 from "payment1" was refunded
deposit(99 + MILLISECONDS_IN_1_DAY, "account1", 100)	returns 320; cashback for "payment2" was not refunded yet
deposit(100 + MILLISECONDS_IN_1_DAY, "account1", 100)	returns 440; cashback of 20 from "payment2" was refunded

Level 4

The banking system should support merging two accounts while retaining both accounts' balance and transaction histories.

- **merge_accounts(self, timestamp: int, account_id_1: str, account_id_2: str) -> bool** — should merge **account_id_2** into the **account_id_1**. Returns **True** if accounts were successfully merged, or **False** otherwise. Specifically:
    - Returns **False** if **account_id_1** is equal to **account_id_2**.
    - Returns **False** if **account_id_1** or **account_id_2** doesn't exist.
    - All pending cashback refunds for **account_id_2** should still be processed, but refunded to **account_id_1** instead.
    - After the merge, it must be possible to check the status of payment transactions for **account_id_2** with payment identifiers by replacing **account_id_2** with **account_id_1**.
    - The balance of **account_id_2** should be added to the balance for **account_id_1**.
    - **top_spenders** operations should recognize merged accounts - the total outgoing transactions for merged accounts should be the sum of all money transferred and/or withdrawn in both accounts.
    - **account_id_2** should be removed from the system after the merge.
- **get_balance(self, timestamp: int, account_id: str, time_at: int) -> int | None** — should return the total amount of money in the account **account_id** at the given timestamp **time_at**. If the specified account did not exist at a given time **time_at**, returns **None**.
    - If queries have been processed at timestamp **time_at**, **get_balance** must reflect the account balance **after** the query has been processed.
    - If the account was merged into another account, the merged account should inherit its balance history.

Level 4 Examples

The example below shows how these operations should work (see test cases for additional examples):

Queries	Explanations
create_account(1, "account1")	returns True
create_account(2, "account2")	returns True
deposit(3, "account1", 2000)	returns 2000
deposit(4, "account2", 2000)	returns 2000
pay(5, "account2", 2000)	returns "payment1"
transfer(6, "account1", "account2", 500)	returns 1500
merge_accounts(7, "account1", "non-existing")	returns False; account `"non-existing"` does not exist
merge_accounts(8, "account1", "account1")	returns False; account `"account1"` cannot be merged into itself
merge_accounts(9, "account1", "account2")	returns True
deposit(10, "account1", 100)	returns 3800
deposit(11, "account2", 100)	returns None; account `"account2"` doesn’t exist anymore
get_payment_status(12, "account2", "payment1")	returns None; account `"account2"` doesn’t exist anymore
get_payment_status(13, "account1", "payment1")	returns "IN_PROGRESS"
get_balance(14, "account2", 1)	returns None; "account2" was not created yet
get_balance(15, "account2", 9)	returns None; "account2" was already merged, doesn’t exist anymore
get_balance(16, "account1", 11)	returns 3800
deposit(5 + MILLISECONDS_IN_1_DAY, "account1", 100)	returns 3906

Another example:

Queries	Explanations
create_account(1, "account1")	returns True
deposit(2, "account1", 1000)	returns 1000
pay(3, "account1", 300)	returns "payment1"
get_balance(4, "account1", 3)	returns 700
get_balance(5 + MILLISECONDS_IN_1_DAY, "account1", 2 + MILLISECONDS_IN_1_DAY)	returns 700
get_balance(6 + MILLISECONDS_IN_1_DAY, "account1", 3 + MILLISECONDS_IN_1_DAY)	returns 706; cashback for "payment1" was refunded

Final Deliverables:

- **Delivery I:**
    - Every team member submits a repository with the full software library distribution and modeling example (i.e., complete cross-functional team's final software distribution making sure it works)
    - Every team member submits their solutions for grading to each assignment (levels 1, 2, 3, and 4) in the Gradescope platform
- **Delivery II**: Every team member submits a pdf with a software engineering reflection that includes:
    - A description of their role in the project.
    - How much do they contribute to the successful project completion (e.g. lead work, help others, coordinated meetings, etc).
    - Describe any challenges/problems you and/or your team dealt with and how you solved them.
    - Algorithmic and performance analysis of each method.
    - A UML diagram of your choice relevant to modeling the design of your project implementation. The UML diagram need not reflect the full implementation of levels 1 through 4, but should at minimum reflect the implementation details of at least 1 level.
    - What can you have done differently:
        - Ex. Software project management
        - Ex. Final product improvements

The final project accounts for 25% of the final Chem 274B grade.
Every student will receive two grades:

1. Cross-functional software development team Grade (80% of final project grade)
2. Individual student Grade (20% of final project grade)