Splitwise - The settle up command

Settle up

One core feature of Splitwise is the ability to settle up in a group. The requirements are:

1. A user can add a transaction to a group.
2. A user can add multiple people that paid for the transaction, and also add multiple people that owe for the transaction.
3. When a user settles up, the app should calculate the minimum number of transactions required to settle up all the debts in the group.
4. Settling up means that no one owes anyone anything.

Let's look at an example to understand this better. Suppose there are 4 people in a group: A, B, C and D. A owes B 100, B owes C 200, C owes D 300 and D owes A 400. The app should calculate the minimum number of transactions required to settle up all the debts in the group.

This is known as the debt simplification problem. There are multiple ways to think about simplifying the debt. The most popular approach, and the one we'll be using, is to minimize the number of transactions required to settle up all the debts in the group. You can also minimize the total amount of money transferred, or minimize the number of people involved in the transactions.

Strategy

Before we start implementing the settle up command, let's think about the class design. Settling up requires an algorithm and there can be multiple different implementations of this algorithm. We can use the strategy pattern to implement this.

Let's create the strategy interface first:

```python
from abc import ABC, abstractmethod

class SettleUpStrategy(ABC):
    @abstractmethod
    def settle_up(self):
        pass
```

Expense model

The strategy interface right now has a method `settle_up` that is intended to return a list of transactions required to settle up the debts in a group. Think of the parameters that this method should take and the return type of this method.

The major parameter that this method should take is the list of expenses in the group. An expense is a transaction that has been added to the group. It has a list of people that paid for the transaction, and a list of people that owe for the transaction. Let's create the model for the expense:

```
from django.db import models

class Expense(models.Model):
    description = models.CharField(max_length=100)
    amount = models.DecimalField(max_digits=10, decimal_places=2)
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)
    paidBy = models.OneToManyField(UserExpense, on_delete=models.CASCADE)
    owedBy = models.OneToManyField(UserExpense, on_delete=models.CASCADE)

class UserExpense(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    amount = models.DecimalField(max_digits=10, decimal_places=2)
    expense_type = models.CharField(max_length=10, choices=[('PAID',
'Paid'), ('OWED', 'Owed')])
```

Here, we created an `Expense` model that has a description, amount and the user that created the expense. It also has a list of people that paid for the expense and a list of people that owe for the expense. We created a `UserExpense` model that has the user, the amount and the type of expense (paid or owed). Creating a separate model for the people that paid and the people that owe allows us to have an extensible design. We can add more fields to this model in the future if required.

So, now our interface looks like this:

```
from abc import ABC, abstractmethod

class SettleUpStrategy(ABC):
    @abstractmethod
    def settle_up(self, expenses: List[Expense]):
        pass
```

The `settle_up` method now takes a list of expenses as a parameter. This method should also return a list of transactions required to settle up the debts in the group. This transaction should have the from user, to user and the amount.

```
class SettleUpTransaction(models.Model):
    from_user_id = models.ForeignKey(User, on_delete=models.CASCADE)
    to_user_id = models.ForeignKey(User, on_delete=models.CASCADE)
    amount = models.DecimalField(max_digits=10, decimal_places=2)
```

Finally, our interface looks like this:

```
from abc import ABC, abstractmethod

class SettleUpStrategy(ABC):
    @abstractmethod
    def settle_up(self, expenses: List[Expense]) ->
List[SettleUpTransaction]:
        pass
```

Creating a concrete class for the algorithm

Now that you have an interface, you can create multiple implementations of the algorithm, and switch between them easily. Let's create a concrete class for our greedy algorithm:

```
class GreedySettleUpStrategy(SettleUpStrategy):
    def settle_up(self, expenses: List[Expense]) ->
List[SettleUpTransaction]:
        # Greedy algorithm to settle up the debts
        pass
```

Implementing the settle up strategy

Calculating the balance for each user

As mentioned before settling up is the process of finding the minimum number of transactions required to settle up all the debts in the group. Before we start implementing the algorithm, we need to figure out the balance for each user. The balance for a user is the net amount of money that the user owes or is owed across all the expenses in the group.

Every expense has a list of people that paid for the expense and a list of people that owe for the expense. The balance for a user is the sum of the amount that the user paid for the expense and the amount that the user owes for the expense. To find the balance for each user:

1. Iterate over all the users in the group.
2. For each user, iterate over all the expenses in the group.
3. For each expense, if the user paid for the expense, add the amount to the balance. If the user owes for the expense, subtract the amount from the balance.
4. The balance for the user is the net amount that the user owes or is owed across all the expenses in the group.

Let's take an example to understand this better. Suppose there are 4 people in a group: A, B, C and D.

- Expense 1 - A paid 1000 and owed 500
- Expense 2 - A paid 3000 and owed 1000
- Expense 3 - A paid 0 and owed 500
- Expense 4 - A paid 0 and owed 250

So the balance for A is:

- Expense 1: 1000 - 500 = 500
- Expense 2: 3000 - 1000 = 2000
- Expense 3: 0 - 500 = -500
- Expense 4: 0 - 250 = -250

Net balance for A = 500 + 2000 - 500 - 250 = 1750

Similarly, you can calculate the balance for B, C and D.

Let's implement this in the `GreedySettleUpStrategy` class:

```python
class GreedySettleUpStrategy(SettleUpStrategy):
    def settle_up(self, expenses: List[Expense]) ->
List[SettleUpTransaction]:
        # Calculate the balance for each user
        balance = self.calculate_balance(expenses)

    def calculate_balance(self, expenses: List[Expense]) -> Dict[User,
Decimal]:
        balance = {}
        for expense in expenses:
            for user_expense in expense.paidBy:
                balance[user_expense.user] =
balance.get(user_expense.user, 0) + user_expense.amount
            for user_expense in expense.owedBy:
                balance[user_expense.user] =
balance.get(user_expense.user, 0) - user_expense.amount
        return balance
```

The above method simply iterates over the expenses and returns the balance in a dictionary where the key is the user and the value is the balance.

Greedy algorithm to settle up the debts

The algorithm outlined is a strategy for settling debts among a group of individuals. It aims to minimize the number of transactions needed to settle all debts. You can implement any other

Algorithm:

1. **Initialization**:

- Initialize two heaps: one for the individuals who owe money (`max_heap`) and the other for the individuals who are owed money (`min_heap`).

2. **Iterative Process**:

    - While both sides of the transaction are not empty:

3. **Find the Individuals Involved**:

    - Identify the individual who is owed the most money (`largestPair`).
    - Identify the individual who owes the most money (`smallestPair`).

4. **Three Possibilities**:

    - There are three potential scenarios when considering the balances between the two individuals (`X` and `Y`):

    a. `X: 500, Y: -600`: In this case, `Y` owes more money than `X` is owed. `Y` pays `X` the amount owed by `X` or `Y` (whichever is smaller) to settle the transaction partially.

    b. `X: 600, Y: -500`: In this case, `X` is owed more money than `Y` owes. `Y` pays `X` the amount owed by `X` or `Y` (whichever is smaller) to settle the transaction partially.

    c. `X: 500, Y: -500`: In this case, both individuals owe each other an equal amount, so the transaction can be settled completely.

5. **Update Balances**:

    - After the transaction, update the balances of both individuals. If one individual's balance becomes zero, remove them from the respective heap.

6. **Repeat**:

    - Repeat steps 3 to 5 until one or both heaps become empty.

7. **Return Transactions**:

    - Return the list of transactions required to settle the debts.

Example:

Let's consider a scenario with three individuals (`A`, `B`, and `C`) and their respective balances:

- `A` owes `B` 300.
- `B` owes `A` 400.
- `C` owes `A` 200.

1. **Initialization**:

    - `max_heap`: `B (-400), C (-200)`
    - `min_heap`: `A (300)`

2. **Iteration 1**:

    - `largestPair`: `B (-400)`
    - `smallestPair`: `A (300)`
    - `B` pays `A` 300.
    - Updated balances:
        - `B` owes `A` 100.
        - Remove `A` from `min_heap`.

3. **Iteration 2**:

    - `largestPair`: `B (-100)`
    - `smallestPair`: `C (-200)`
    - `C` pays `B` 100.
    - Updated balances:
        - `B` owes `A` 100 (unchanged).
        - `C` owes `A` 100.
        - Remove `B` from `max_heap`.

4. **Iteration 3**:

    - `largestPair`: `C (-100)`
    - `smallestPair`: `A (100)`
    - `C` pays `A` 100.
    - Updated balances:
        - `C` owes `A` 0.
        - Remove `C` from `max_heap`.

5. **Conclusion**:

    - All debts are settled.
    - Transactions:
        - `B` pays `A` 300.
        - `C` pays `B` 100.
        - `C` pays `A` 100.

The algorithm efficiently settles debts among a group of individuals by iteratively identifying the individuals involved in the largest and smallest transactions, making partial or complete settlements, and updating the balances accordingly until all debts are cleared. It minimizes the number of transactions needed to settle the debts.

```
class GreedySettleUpStrategy(SettleUpStrategy):
    def settle_up(self, expenses: List[Expense]) ->
List[SettleUpTransaction]:
        # Calculate the balance for each user
        balance = self.calculate_balance(expenses)

        # Initialize lists for min and max heaps
        min_heap = []
        max_heap = []

        # Populate heaps
        for user_id, amount in balance.items():
            heappush(min_heap, (amount, user_id))
            heappush(max_heap, (-amount, user_id))

        transactions = []

        # Find transactions
        while len(min_heap) > 1:
            smallest_pair = heappop(min_heap)
            largest_pair = heappop(max_heap)

            from_user = largest_pair[1]
            to_user = smallest_pair[1]
            transaction_amount = -largest_pair[0]  # Negate to get the
actual amount

            transaction = SettleUpTransaction(from_user, to_user,
transaction_amount)
            transactions.append(transaction)

            combined_amount = smallest_pair[0] + largest_pair[0]

            if combined_amount < 0:
                heappush(max_heap, (combined_amount, largest_pair[1]))
            elif combined_amount > 0:
                heappush(min_heap, (combined_amount, smallest_pair[1]))

        return transactions

    def calculate_balance(self, expenses: List[Expense]) -> Dict[int,
Decimal]:
        balance = defaultdict(Decimal)
        for expense in expenses:
            for user_expense in expense.paidBy:
                balance[user_expense.user_id] +=
Decimal(user_expense.amount)
            for user_expense in expense.owedBy:
                balance[user_expense.user_id] -=
Decimal(user_expense.amount)
        return balance
```