ATT
CSCI3060U Project Phase #4

<u>Back End Rapid Prototype</u>

Taylor Smith
Alexandar Mihaylov
Talha Zia

Introduction

Purpose

The purpose of this design document is to demonstrate the design and architecture of the Rapid Prototype Back End in our Banking system. The Banking system is designed to provide regular banking transactions that could be carried out by an admin or standard user. Requirements pertaining to the backend are provided by the client and highlighted throughout this design document.

Scope

The back end of our banking system is essential in order to keep an 'up to date' database. Back end reads in the data provided by the user through our front end and stores it in the file system.

Definitions, Acronyms and Abbreviations

Admin – Bank employee who has administrative rights to the system
mbaf - Master bank account file
cbaf - Current bank account file
tf - transaction file

Design Overview

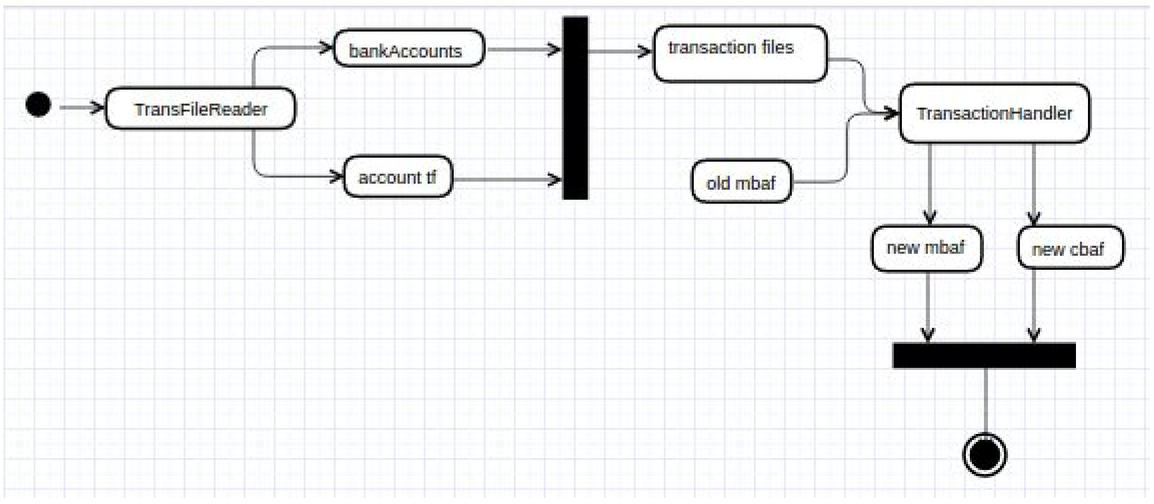
Description of problems

Back end is required to read the previous day's mbaf and merged bankaccount account transaction file, update all transactions in the old mbaf in order to produce the new mbaf. Back end is also required to calculate the daily cost of transactions based on the user's account plan. For student plans, the account is charged $0.05 for each transaction and $0.10 for non student.

For each bad input, the back end should immediately stop and log an error in the form of ERROR: <msg>.

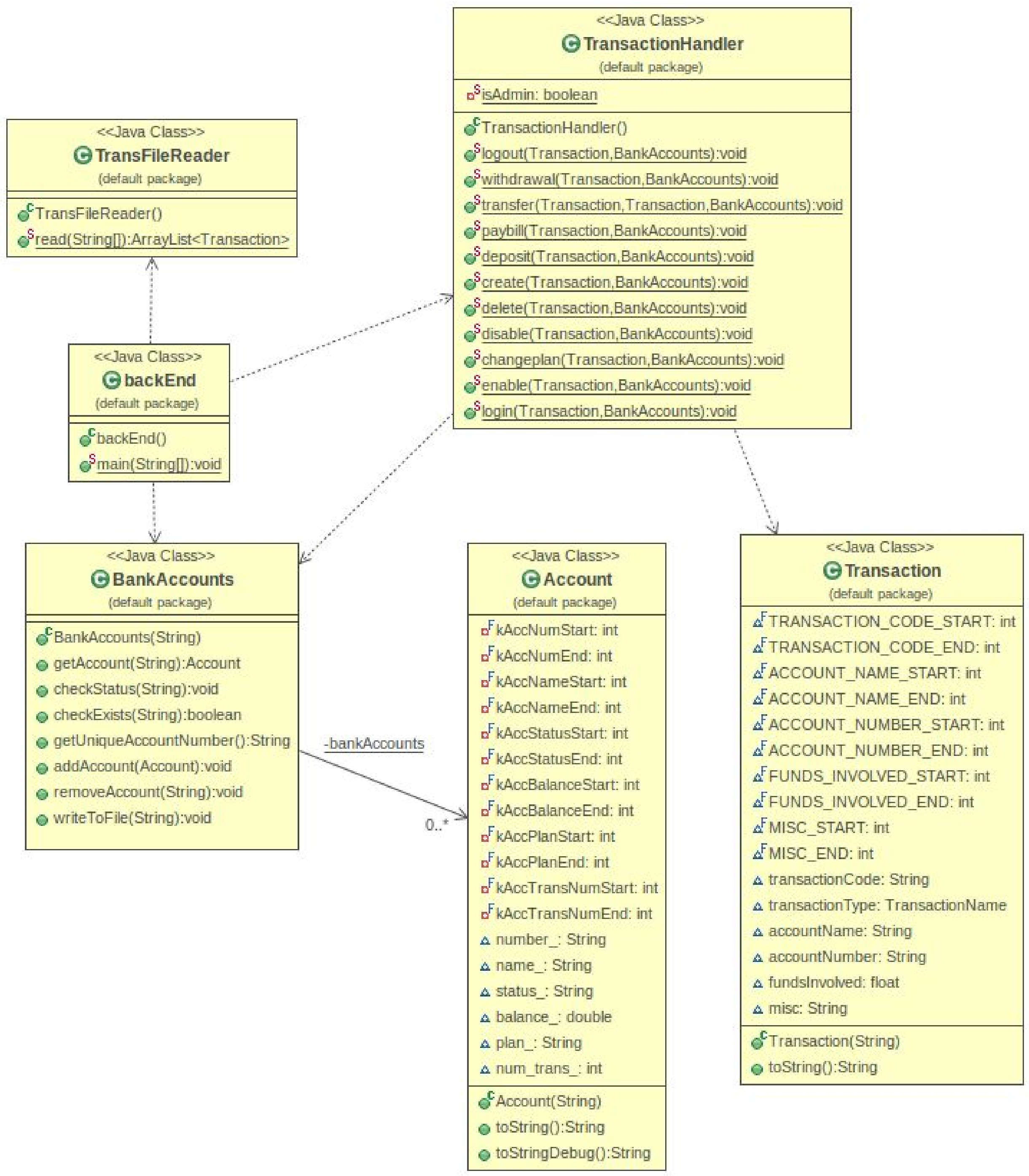
UML Activity Diagram

Inputs: transaction files **(trans-files/ -> mergedTransactions.tf)**, old mbaf **(old.mbaf)**
Outputs: new mbaf **(new.mbaf)**, new cbaf **(new.cbaf)**



System Architecture

UML Class Diagram



Class Name	Description	Methods
backEnd()	The back end of the banking system that contains the main function. Takes in two arguments in the form <old-mbaf> <list-of-transaction-files> and is responsible for creating instances of other classes that read in the master bank accounts file and all the transaction files. It then Iterates through all the transactions and calls their respective handlers	**main():**
BankAccounts(String)	Holds all data related to Bank Accounts in an internal list of Accounts. It is also responsible for reading in a Master Bank Accounts File and populate its internal data structure with all the Accounts in the file. Additionally BankAccounts is able to write the new Master Bank Accounts File by taking all the accounts it holds and writing out to a file in the required format. The Class also has a bunch of utility functions relating to interactions with Accounts, since the internal List of Accounts is private and other classes do not have direct access to it. These utility functions allow other classes to add/remove accounts, check if an account exists, and obtain a unique account number that doesn't yet exist in the "database".	**BankAccounts():** Constructor that is able to take in mbaf name and load each line into an account object, which is stored in private statis bankAccounts variable. **getAccount():** fetches account object from database **checkStatus():** prints information about a given account **checkExists():** checks if a given account exists in the database **getUniqueAccountNumber():** finds and returns unique account number **addAccount():** adds an account objects to the account database **removeAccount():** takes in an account number and removes it from the database

		writeToFile(): writes all the accounts in the database to a file provided as an argument
Account()	Holds all the necessary information relating to an Account like the number, name, status, balance, plan, and the number of transactions. Is able to take in a string from a master bank accounts file, parse it and fill in its internal members of all the required information. Is also able to produce a string in the same format of the current account using its toString() function.	**Account():** Constructor that parses the raw master bank account file line into the respective members **toString():** Returns a string with all the account information. If the boolean passed to this function is true, then the number of transactions are also included at the end of the string
TransFileReader()	Reads in a list of transaction file names and produces a mergedTransactions.tf containing all the transactions from every file provided. The constructor takes in an array of strings, with the first element being omitted (the old.mbaf file) and the rest of the strings in the array being names of transaction files.	**read(String[]):** Reads in a list of transaction files, parses through each transaction and stores them in an Array List.
TransactionHandler():	Used to handle every single type of transaction in the banking system. Has functions for every transaction and each handler typically takes in a Transaction object and the current bank account database. With each transaction it modifies the given account database accordingly.	l**ogout():** sets the isAdmin boolean to false **withdrawal():** obtains the account, takes off the withdrawn amount, applies fees if necessary and validates if the account balance is more than 0 after transaction **transfer():** obtains both accounts for transfer, takes off the transfer amount from the first account, applies fees if applicable, validate if

		account balance is more than 0 after transaction **paybill():** Obtains the amount, takes off the paybill amount, applies fees if applicable and validate if the account balance is less than 0 after transaction **deposit():** Obtains the amount, adds the deposit amount, applies fees if applicable and validate if the account balance is less than 0 after transaction **create():** creates a new account with unique account number and initialize it. **delete():** deletes the account, more to be added later. **disable():** sets the account status to ‘D’ for disable **changeplan():** sets the account plan to either student or non-student **enable():** sets the account status to ‘A” for active **login():** validates login for admin/standard user
Transaction()	Holds all information relating to a single transaction. This includes holding information like the transaction code/type, the account name, number and funds involved in the transaction, and also if the transaction was done by an admin or regular user. The constructor of this class is able to take in a raw string read in from a transaction (.tf)	**toString():** Converts the current transaction object into the same formatted string that it was originally read in by the constructor

	file, parse it and populate all the internal respective fields. The class is also able to convert the current transaction object into the same raw string format that it originally read it in.	