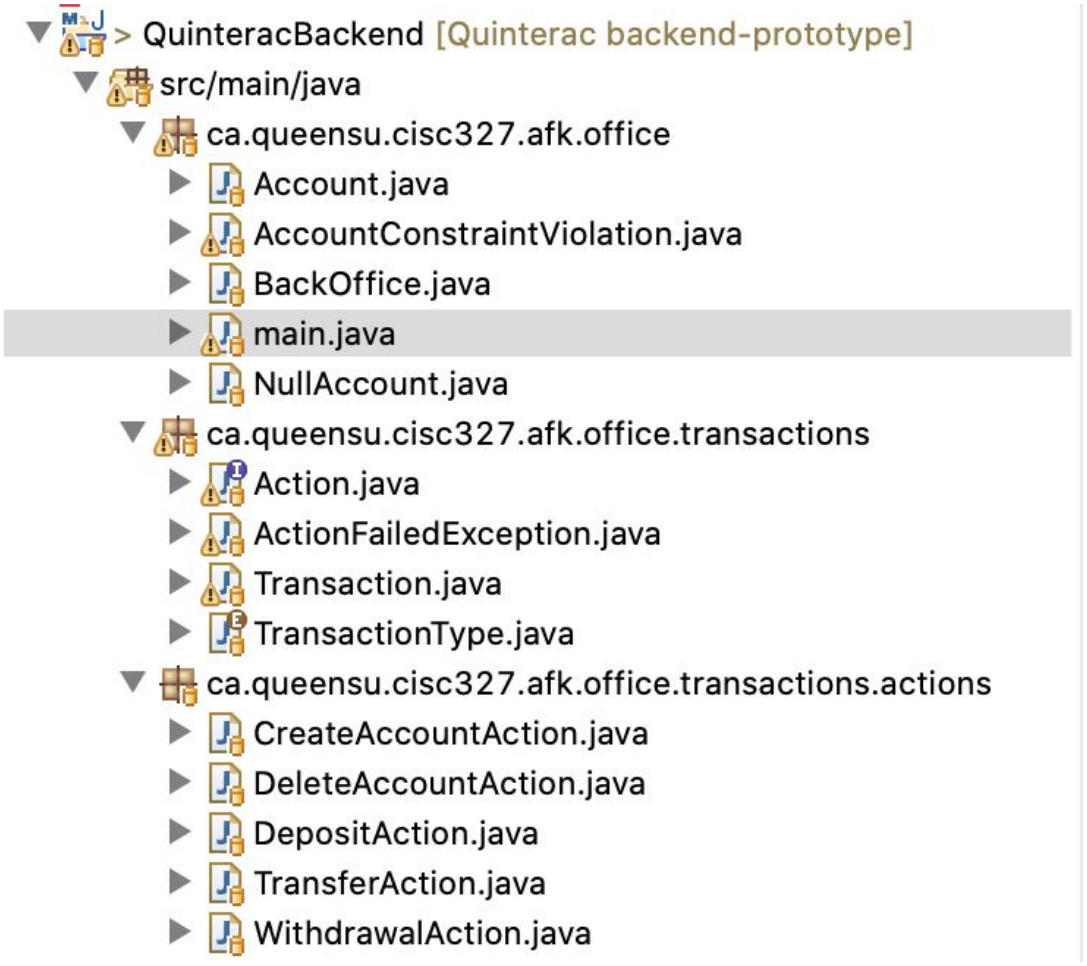
Design Document

Group 17
Shihao Lu, Ziping Li, Ryan Kerr

Below is an overview of the file structure of the backend. There are three packages for the back office. The office package contains the I/O and main functions for the back office and account. The other two are specifically for the transactions, one contains the general functions and the other one contains the actions for each specific type of transaction.



Package afk.office

Main Class

The main class contains the file I/O functions for the backend, including reading in the merged transaction summary file and the master account file, and writing to the two files. There are several exception classes created for the functions as well.

Method

main	Initializes the backend application.
readTransactionList	Read in the merged transaction summary file line by line and save it to an array list of strings. Check if each line meets the constraints(the length of each of line cannot be longer than 61 characters, the last line have to start with EOS, etc.)
readMasterAccountList	Read in the master account file line by line and save it to an array list of strings. Each line cannot have a length of over 47 characters. Each input block is split exactly by one space.
transactionReader	Create a list of transaction objects based on the read in transactions.
accountReader	Create a list of account objects based on the read in master accounts.
writetiFile	Overwrite a file with a list. Create a copy of the old file as a log for future review.
exceedMaxLenthExceptio n	If the length of a line exceeds the max length allowed
transactionCodeException	If the transaction is not recognized
moreThanOneSpaceExce ption	If the blocks are split by more than one space

Account Class

Represents an account with a name and account number. This account class differs slightly with the version found in the front-end as it keeps track of the account balance rather than individual transaction limits.

Method

Constructor	Creates an account with a name, number, and starting balance
getNumber	Returns the account number as a string
getName	Returns the account name
getBalance	Returns the account balance
applyTransaction	Checks the source and destination fields of a given transaction. The amount specified is either subtracted or added to the balance for the respective cases
adjustBalance	Adds or subtracts the given value from the balance. Throws an **AccountConstraintViolation** if the resulting balance would be negative; Sets the balance otherwise
toString	Returns a string with all the account information formatted for the master accounts file
clone	Makes a non-aliased copy of the account
compareTo equal	Allows for comparing of Accounts

Children

NullAccount	Represents the special account #0000000 to be used as a placeholder in unused transaction fields. No transactions may be applied to this class.

BackOffice Class

The back office class is responsible for holding the master account list and handling the high-level logic. It is very similar to the **Session** class of the front-end. Unlike the **Session** class, however, the **BackOffice** does not rely on user input.

Methods

Constructor	Converts a list of Accounts into a map of Accounts. Creates a copy of the original account list as a backup
execute	The back office runs through each transaction one at a time and calls on their respective **Action**s.

	If an **ActionFailedException** occurs, the execute function will return the original master account list from the constructor. Otherwise, this function will return a *sorted* master accounts list.

Package afk.office.transaction

Action Interface

The **Action** interface declares the *execute* function and defines a number of helper functions. Implementations of this interface can call these functions to safely lookup accounts in the master account list and wrap exceptions into **ActionFailedException**.

Methods

assertAccountsExist	Performs a lookup in the master account list to ensure that that both the accounts listed in a transaction exist. If the account does not exist, an **ActionFailedException** specifying which account (source or destination) was missing will be thrown
apply	Resolves the given account from the master accounts list and calls the *applyTransaction* function for that account and the given transaction. Wraps any exceptions in an **ActionFailedException**
execute	Implementations of this interface can use this function to make modifications to the master accounts list (create/delete accounts) or modify the balances of accounts by using the *apply* function

Children

WithdrawalAction	Calls the *apply* function for the source account
DepositAction	Calls the *apply* function for the destination account
TransferAction	Calls the *apply* function for the source account then the destination account (Order is important! Calling destination first would result in money being transferred even if the source account had insufficient funds!)
DeleteAccountAction	Checks that the given account exists, that the name on the transaction matches the name of the account, and that the account has a balance of zero before removing it from the master accounts

	list. If any of those criteria fail, the function will throw an **ActionFailedException**
CreateAccountAction	Checks that the source account number on a given transaction does not exist in the master accounts list and creates the account if the number is not taken. Throws an **ActionFailedException** if the account number is already taken.
(a, t) -> {}	A no-op lambda implementation

Transaction Class

Represents a transaction with a given transaction code, source and destination accounts, value, and optional account name. The back-end implementation of **Transaction** is nearly identical to that of the front-end, but the source and destination accounts are just the account numbers rather than a full **Acconut** object.

Method

Constructor	Creates a new transaction with the fields **TransactionType**, source account, destination account, amount, and optional account name
getAmount	Returns the amount stored in transaction
getSourceAccountNumber / getDestinationAccountNumber	Returns the relevant account numbers stored in the transaction
getType	Returns the enumerated **TransactionType**
toString	Returns the transaction in the format specified for the transaction summary file

TransactionType Enum

The TransactionType enum serves as the /usr/bin of this program. It defines all the available transaction types available, each with their own short code and action script.

Method

codeToEnum	Performs a lookup and returns the enum corresponding to a given transaction code. Throws an **IllegalArgumentException** if no corresponding enumeration is found.
getShortCode	Returns the 3 character transaction code

getAction	Returns a **TransactionScript** corresponding to this transaction type