Quinterac Front End System Design Documentation

CISC 327 Fall 2019

Assignment 2

Instructor: Dr. Stephen Ding

Contributors:

Denny,	Matt	(16mmmd1)
Laxdal,	Kai	(16kibl)
Littlefield,	Brent	(16bml1)
Liu,	Tong	(9tl14)

1 System Specifications

Please refer to https://l1nna.com/327/Course%20Project.pdf for functional specifications of the Quinterac Front End System (QFES).

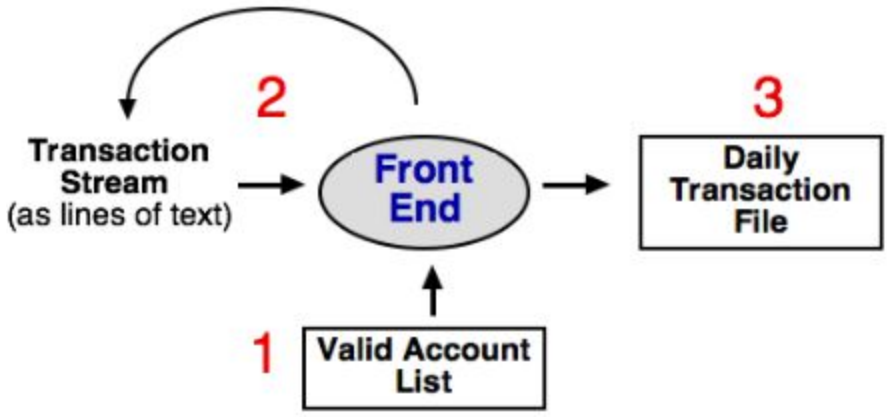
Qualitative requirements are not considered to be within the scope of this assignment.

2 System Design

For the design process of creating the front End System for Quinterac, we made it a goal of creating a robust design that can be easily maintained while still being simple to change incase of changing requirements that might occur in the future. We started by breaking down the program into 3 different sections. These three sections we call User Interaction, Logic and finally Back End Interaction. In the User interaction section we handle all of the possible events that can occur on our given system. The Logic Section is where we make sure that the program is behaving correctly and the user inputs are valid. Finally in the back end interaction section we make sure that the outputted file is formatted correctly and in chronological order so we know what transactions took place and in what order.

2.1 General Front End Design

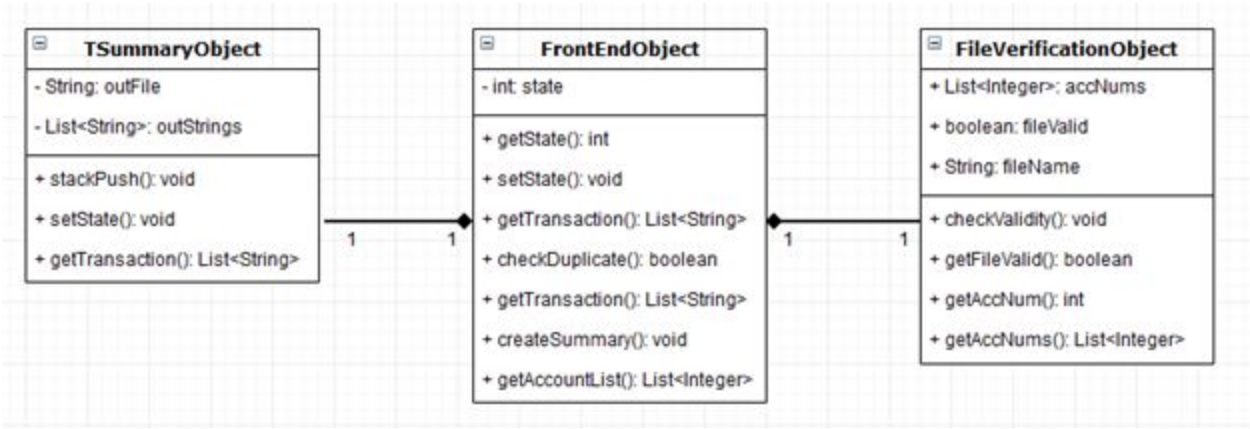
The following diagram below summarizes the inputs and outputs of QFES.



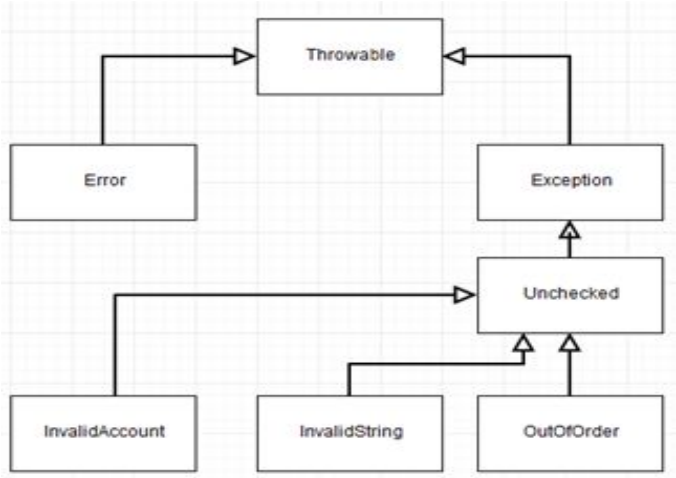
2.2 Class Hierarchy Diagrams

The current implementation of QFES is the first of a series of evolutionary prototypes. The prototype has been simplified in terms of several lower-level details for ease of development and testing. The following class diagrams describe the intended design and architecture of QFES.

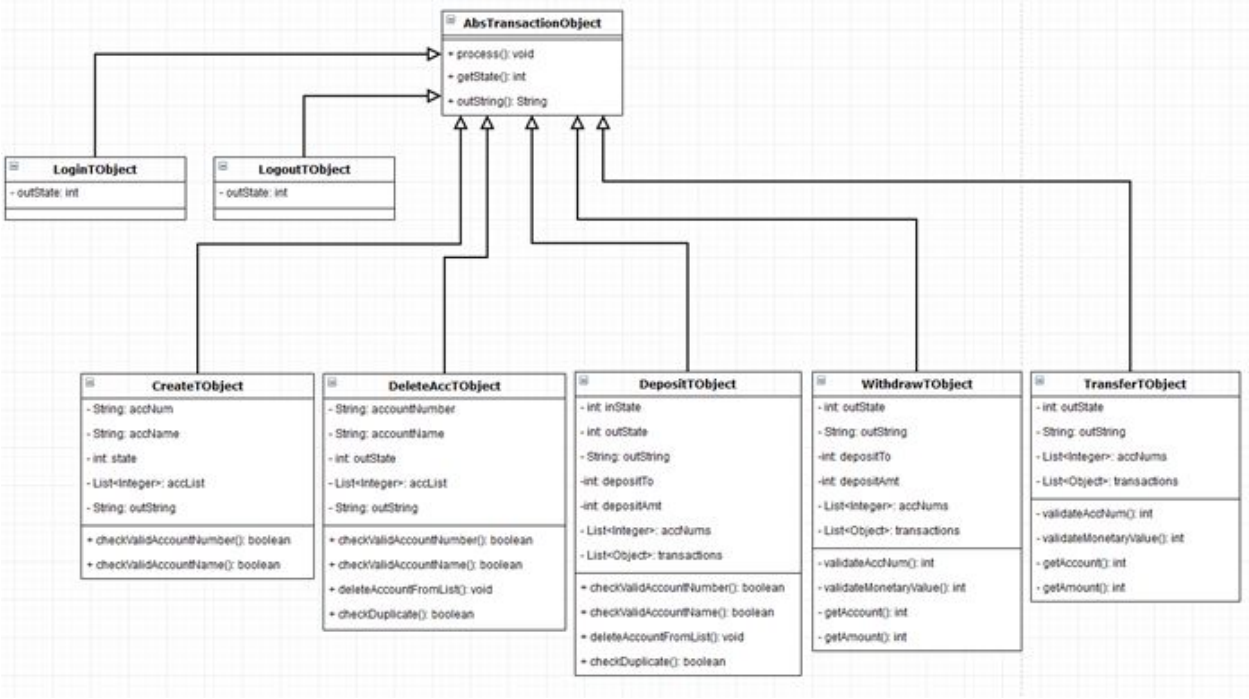
FrontEndObject:



Exception Hierarchy:



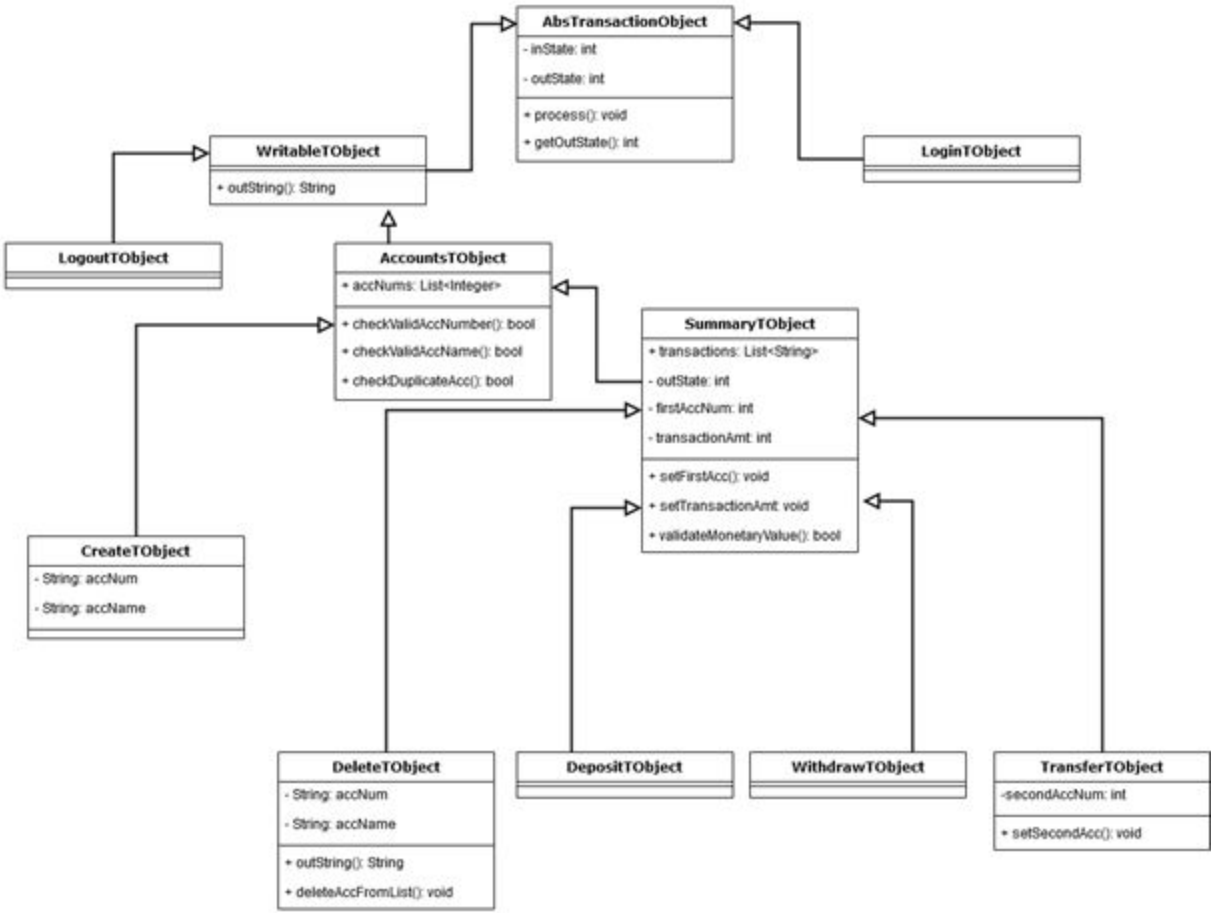
Class Hierarchy:



In line with the principles of evolutionary development, the current prototype has been implemented with ease of development and testing in mind. Several code refactoring and optimizations can be performed on the current hierarchy of transaction objects in order to improve the structure of QFES.

After analysis of the current implementation, the following structure would represent a system with a

Planned Improvements to Transaction Object Hierarchy:



3. Structure of Solution

In this section we will be outlining each of the 15 Java files that create the Front End System of Quinterac. Down below there are 15 tables, one per file. In these tables there will be a short description which will briefly outline the intentions of that file as list all of the methods inside of the file and what they do.

3.1 Main Files

File: **Quinterac.java**	This file is intended to be the main “workhorse” of the front end system.
main()	This main function instantiates a new FrontendObject and then will execute a while loop that will keep executing the behaviour of the system.

File: **FrontendObject.java**	This File's intention was to create a way that we can track the state of the system whether that be 0 for not logged in, 1 for logged in using the ATM mode, 2 for logged in as agent, 3 for logged out of system, -1 for an error when creating the object.
FrontendObject	This method is the constructor that is called to create a new frontendObject
getState	Returns the state of the frontEnd when called
setState	Will manually override the state of the system when called and passed an integer
getTransactions	Returns a stack of transactions from the TSummaryObject instantiated in the constructor
checkDuplicate	Will check to see if n is in the accounts Number ArrayList
stackPush	Will push a String to a stack to be written to the output file when the user logs out
createSummary	Will prompt the TSummaryObject instantiated in the constructor to write the output to the file
getAccountList	Will return a pointer to the accounts number arraylist

File: **FileVerification.java**	This File will check to make sure that the file passed as input contains only valid account numbers as well as to create an arraylist of each of the account numbers in the file
FileVerification	This is the constructor of the FileVerification Object and it will call the checkValidity Method and if that returns false then it will throw an InvalidAccountException
checkValidity	This method goes through all the lines in the file and makes sure each line in the file represents a valid account number as well as the file ends with "0000000"
getFileValid	Will return true or false depending on if the file is valid or not
checkDuplicate	Will check for a duplicate to the account number n in the accNums arrayList
getAccNum	This method will return an account number at a given index in the array list
getAccNums	This method will return a pointer to the accounts Number List

3.2 Transaction Files

File: **AbsTransactionObject.java**	This file is intended to outline the abstract objects we use to define different concrete transactions

File: **CreateTObject.java**	This file is used to handle the create account transaction that can be done when a user is logged in using the “agent” mode
CreateTObject	This constructor accepts an integer for the state and a n arraylist for the accounts list and will check what state was passed due to the Create Account transaction being only able to be used by “agents”
process	Performs the actions needed to process a valid create account transaction
getState	Will return 0
outString	Will return the String that needs to be outputted to the output file for record of the transaction that took place
checkValidAccountNumber	This checks to see if the account number inputted by the user is correctly formatted
checkValidAccountName	This checks to see if the account name inputted by the user is correctly formatted

File: **DeleteAcctTObject.java**	This file’s intention is to be used when a user logged in as an agent wants to process a delete account transaction.
DeleteAcctTObject	This is the constructor for the class. It will check if the state is not 2 and if it is will set the outstate to -1 since there was an error due to only agent mode being able to delete accounts.
process	This will perform the correct actions needed to record a valid delete account
getState	Return the state of the session
outString	Will return the String that needs to be outputted to the output file for record of the transaction that took place
deleteAccountFromList	Delete the account from the accountsList so we cannot perform

	any more transactions on this account
checkDuplicate	Checks if the account number n is actually in the accounts list
checkValidAccountNumber	This checks to see if the account number inputted by the user is correctly formatted
checkValidAccountName	This checks to see if the account name inputted by the user is correctly formatted

File: **DepositTObject.java**	The intention of the file is to handle the Deposit transaction
DepositTObject	This constructor takes in a state of the session, arrayList of account numbers and a FrontendObject
process	This will perform the correct actions needed to record a valid delete account
getState	Return the state of the session
outString	Will return the String that needs to be outputted to the output file for record of the transaction that took place
checkValidAccountNumber	This checks to see if the account number inputted by the user is correctly formatted
checkValidAccountName	This checks to see if the account name inputted by the user is correctly formatted
validateMonetaryValue	This method takes in a string representing the amount the user would like to enter as well as an integer representing the account number they would like to deposit to. This method checks all of the previous transactions on the day and determines if the amount entered by the user is valid to deposit or not.

File: **loginTObject.java**	The intention of this is to handle the ability to log into the front end under “machine” or “agent”
LoginTObject	This constructor takes in a state and will determine if the user is able to log in or not. It will throw an exception if the state is in anything but 0 and set the outstate to -1.
process	This method is used to act out the behaviours that are required for a

	valid login transaction
getState	Returns the outState variable
outString	Returns the string needed to write to the output file upon logout

File: **logoutTObject.java**	The intention of this file is to handle the logout transaction that a user may want to use
LogoutTObject	This constructor takes in a state and checks if it is 1 or 3 and if so will throw an exception because you cannot logout when not logged in.
process	Will set the outstate variable to 3 indicating the user logged out of the Front End System
getState	Will return the int variable outState
outString	Will return the required string that needs to be added to the output file to indicate a processing day has concluded

File: **TransferTObject.java**	The intention of this file is to handle the transfer transaction that a user may use
TransferTObject	This is the constructor for the class. It will check if the state of the use is 0 (logged out) or 3 (logging out) and throw an error if it is, otherwise it will just make necessary assignments.
validateAccNum(String)	Checks to see if the account number entered as a string is valid or not and return the account number as an int .
validateAccNum(String, int)	Checks to see if the account number entered as a string is valid or not and checks to see if it is the same as the previous account number entered as an int.
validateMonetaryValue	Checks to see if the amount entered as a String is valid, checks if the amount is within constraints and then returns it as an int.
getAccount	Asks the user to enter the account number until a valid account number is selected.
getAmount	Asks the user to enter the amount until a valid amount is entered.
process	Does all that is necessary to process the transaction.

getState	Returns the state.
outString	Returns the transaction summary.

File: **WithdrawTObject.java**	The intention of this file is to handle the withdraw transaction that a user may use
WithdrawTObject	This is the constructor for the class. It will check if the state of the use is 0 (logged out) or 3 (logging out) and throw an error if it is, otherwise it will just make necessary assignments.
validateAccNum	Checks to see if the account number entered as a string is valid or not and return the account number as an int .
validateMonetaryValue	Checks to see if the amount entered as a String is valid, checks if the amount is within constraints and then returns it as an int.
getAccount	Asks the user to enter the account number until a valid account number is selected.
getAmount	Asks the user to enter the amount until a valid amount is entered.
process	Does all that is necessary to process the transaction.
getState	Returns the state.
outString	Returns the transaction summary.

3.3 Transaction Summary

File: **TSummaryObject.java**	The use of this File is to output all of the required transaction summary lines to the transaction Summary Output File.
TSummaryObject	This constructor takes in a string and also creates a stack to be able to push the outLines from each of the Transaction Objects onto.
stackPush	This adds the Strings to the bottom of the stack to keep all of the transactions that were performed in chronological order.
createFile	This method is used to perform the writing to the Output File
getTransactions	Returns a Stack of all the transactions that have occured in the current processing day.

3.4 Exception Files

File: **OutOfOrderException.java**	The intention of this file is to create a custom tailored exception we can throw when the user tries to do something that is not allowed when the system is in the logged out state (state = 3)

File: **InvalidAccountException.java**	The intention of this file is to create a custom tailored exception we can throw when the user tries to enter an invalid account number when trying to proceed with a transaction

File: **InvalidStringException.java**	The intention of this file is to create a custom tailored exception we can throw when the user tries to enter an invalid string when trying to proceed with a transaction