Faculty of FCES Assessment Brief

Module Title: Secure Programming
Module Code: IS2S582
Module Leader/Tutor: Dr Emlyn Everitt
Assessment Type: Practical Coursework
Assessment Title: A Secure Banking System
Weighting: 40%
Word count/duration/equivalent: n/a
Submission Date: 09-Aug-24
Return Date: 20-Sept-24

Assessment Description

Guidance on Format of Assessment

Note: Students are reminded **not** to include this assignment brief with the assignment submission

Learning Outcomes Assessed

Marking Criteria/Rubric

Note: All grades are provisional until they are ratified by the exam board

Submission Details

Via Unilearn

What happens next?

Your marked assessment should be available 20 working days after submission. However, please be advised that this may be subject to change in the event of Bank Holidays, University Closure or staff sickness. If there is something about the feedback you have been given that you are unclear about, please see your module tutor.

Feedback Method

Feedback will be provided via Unilearn

Late Submission

Submit via Unilearn

Extenuating Circumstances

https://advice.southwales.ac.uk/a2z/extenuating-circumstances

Referencing, Plagiarism and Good Academic Practice

https://advice.southwales.ac.uk/a2z/referencing-plagiarism-and-good-academic-practice

Learning Support Resources

https://studyskills.southwales.ac.uk

Source (100 marks)

Criteria		Professional (1st Class)	Senior (Upper 2nd Class)	Intermediate (Lower 2nd Class)	Amateur (3rd Class)	Novice (Narrow Fail)	Beginner (Fail)
All Source Code (100 marks)	Encapsulation & Generalization	Elements are written in a way that invites reuse in other projects. The use of design patterns to facilitate program development is clearly demonstrated.	Elements are nearly always written in a way that supports reuse in other projects. The use of design patterns to facilitate program development is demonstrated.	Components are mostly well planned and implemented. Elements are generally written in a way that supports code reuse. There is some evidence that the use of design patterns facilitates program development.	Components exist, but are not well thought out or are used in an arbitrary fashion. Elements are seldom written in a way that invites code reuse. There is little to suggest that the use of design patterns facilitates program development.	Little evidence of modularisation or generalisation. Few elements are written in a way that supports code reuse. There is little if anything to suggest that the use of design patterns facilitates program development.	No meaningful evidence of modularisation or generalisation is presented. There is little if anything to suggest that the use of design patterns facilitates program development.
	Design, Structure & Efficiency	Program is designed in a logical manner. Control structures are used effectively and correctly. Data structures, algorithms and patterns are implemented efficiently and appropriately.	Program is generally designed in logical manner. Control structures are used correctly. Data structures, algorithms and patterns are implemented effectively.	Program design is mostly logical. Control structures are used correctly, although more appropriate structures may have been selected. Reasonable data structures, algorithms and patterns are generally implemented.	Program isn′t as clear or logical as it should be. Control structures are frequently used incorrectly. Steps that are obviously inefficient are used.	Program isn′t as clear or logical as it should be. Control structures are occasionally used incorrectly. Steps that are clearly inefficient are used.	No meaningful evidence of logical, efficient or effective decision making in terms of selected data structures, algorithms and patterns.

Tasks

Scenario

You work for Big Bank LTD, a well-known clearing bank. One day your boss comes to you to say that she wants you to develop their new and secure bank transaction software. She then provides you with a code specification (see Appendix A).

The transaction software should be able to:

- Validate and verify banking transaction w.r.t. customer accounts and currency
- Log impending transactions
- Implement atomic transactions – I.e. rollback any incomplete transactions (Command pattern)
- Create unique and strictly formatted account numbers (Singleton pattern)
- Implement a unique and privileged reserve account (Account subclass) that will be the only account able to introduce new money into the system.

Other criteria for software:

- The firmware should only display (print) basic success/ error messages: **no attempt should be made to implement user menus or prompt for user input in any way**.
- The software will only respond to method calls made to the public interfaces of the classes outlined in the specification – see appendix)
- **The code provided in Appendix A must be used as a starting point and the code structure provided must not be changed, only added to.**
- **No libraries should be used apart from sys.maxsize**
- The API specification together represent the formal specification for the project, and a failure to adhere to them will result in project failure.
- You should employ appropriate design patterns in the delivery of this project. Any solution that is not implemented using appropriate design patterns will be judge a failure, even if it works.
- **DO NOT INCLUDE the 'main' function** (if __name__ == '__main__') or any inline code in your submission, just classes.
- **Your code must work.** If the code you present is broken in any way, marks will be lost.
- Because of the marks awarded / lost for functional/ non-functional code, **you would be better off submitting simple working code than more complex non-functional code.**
- When submitting your work, please submit your code in **a single .py file** to Unilearn in the format: *put_your_student_ID_here*.py – **DO NOT** submit a zip/rar file, marks will be lost if you do.

Tasks

Appendix A

```
class Command: #abstract class
    def do(self):
        pass
    def undo(self):
        pass

class GenerateAccountNumber:
    #only one generator allowed (Singleton pattern) and it produces a 14 char (numerals) string
    #in the real world, the generator would be more sophisticated
    _instance = None
    def __new__(cls) -> str:



class AccountNumber:
    def __init__(self):
        #generates account number. Raises exception if not valid
    def _is_valid(self, account_number:str) -> bool:
        #validates that account number has 11-14 digits

    def id(self) -> str: #object return value
        #return account number (private attribute)

class Account:
    pass


class Currency:
    def __init__(self, amount:int):
        #takes in amount and validates that it is an appropriate value
    def _is_valid_amount(self, amount:int) -> bool:
         #validates that amount is an int and positive
    def value(self):
        #returns value (private attribute)


'''We have here a chain of trust:
    -we trust that Currency and AccountNumber are able to sanitise untrusted data
    -classes that only evoke the above do not need to sanitise anything passed to them, they only need to verify class type
    -BankTransaction class implements the Command pattern'''


class Account:
    '''Account object performs checks on its own account number and balance transfers.
    BTs are logged, but not committed until commit method is called'''
    def __init__(self):
        self._account_number = AccountNumber().id()
        self._balance = 0
        self._log = []
    def id(self):
        #returns account number (private attribute)
```

```
def _is_valid_account(self, account:Account) -> bool:
    #returns True if valid account

def _is_valid_currency(self, currency:Currency) -> bool:
    #returns True if valid currency

#class Account
def transfer_in(self, currency:Currency, source:Account, destination:Account):
        #check valid source and destination
        #check valid currency
        #if all valid, log transfer in


def transfer_out(self, currency:Currency, source:Account, destination:Account):#check valid BT and transfer source
        #check valid source and destination
        #check valid currency
        #if all valid, log transfer out

#class Account
def commit(self):
    #Balance transfers (in/ out) are logged, but don't actually happen until subject to a commit

def uncommit(self):
    #uncommit attempts to roll back a commit (invoked if a problem occurs with a commit)
    def clear_commit_log(self):
                #clears commit log when requested (after a successful commit – i.e. not rolled back)



class ReserveAccount(Account):
    '''Singleton pattern ensures that there will only ever be a single reserve account.
    The reserve account it unique in that it's balance can be set or added to directly
    (i.e. the money can be transferred in from nowhere. All other accounts require you
    to transfer money from a know account'''
    _instance = None
    def __new__(cls):
               #initialise and return single instance

    def __init__(self):
        pass

    def add_balance(self, currency:Currency):
              #checks valid currency and adds to balance
              #commit log not used because this is a side channel to introduce starting capital

class BankTransaction(Command):
    '''BankTransaction attempts to perform an atomic transaction.
    An atomic transaction is a balance transfer between two account that is undone
    ('rolled back') if a problem occurs. This rollback is implemented using the Command pattern'''
    def __init__(self, source_account:Account, destination_account:Account, currency:Currency):
        #no need to validate: validation performed at an account level
    self._source_account = source_account
    self._destination_account = destination_account
    self._currency = currency
    self._source_commit = False
    self._destination_commit = False

#class BankTransaction
def do(self):
    #commit transaction


def undo(self):
    #uncommit


    self._clear_commit_log()


def _clear_commit_log(self):
    #clear any commit logs
```