Assessment Support Sheet

Prepared by Dr. Shadi Basurra

CMP4266 – Computer Programming

11/1/17

Contents

Basic design of the Bank System

According to the assessment brief for this module (CMP4266 Computer Programming), the students need to design, develop and test a software that implements the basic functionalities of a banking system. A partial implementation of such a system has been released along with the assessment brief. The released implementation consist of the following classes: Customer, Account, Admin and the main class BankingSystem. The following diagram depicts the relationships among the classes of the partial implementation of the Banking System:

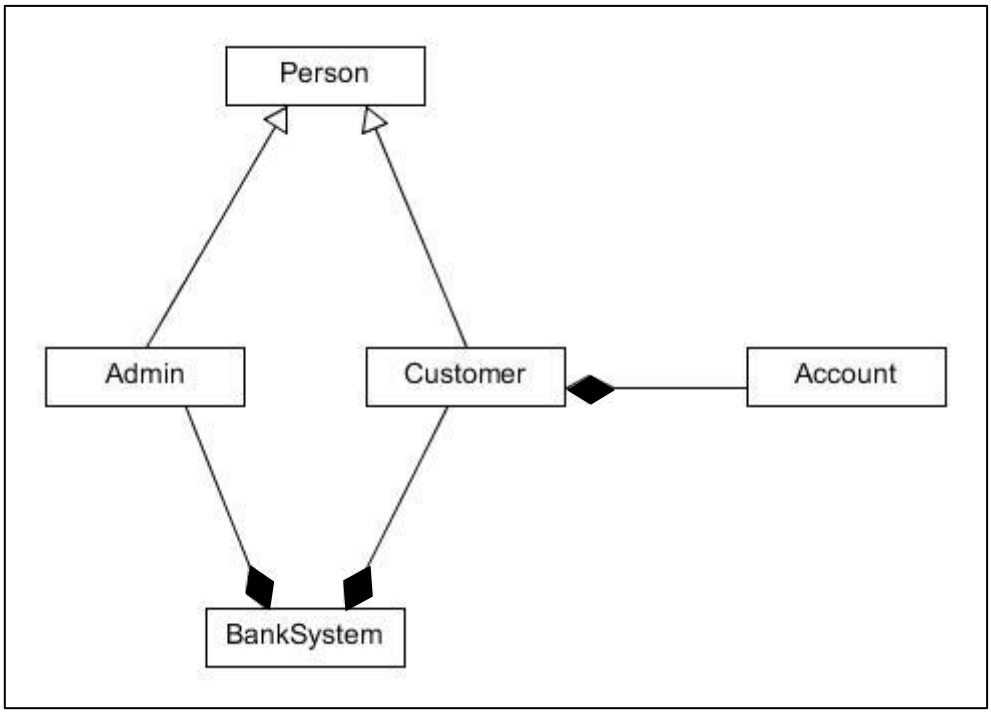

Figure 1: Basic class diagram

From the above Figure 1 the class relationships are defined as follows:

1. Admin and Customer inherits Person (Inheritance)
2. Customer has Account (Composition)
3. BankSystem owns customers and Admins (Composition)

The following sections will briefly discuss each class by explaining it's important attributes and the class role in the system.

The Person class

Looking at the code provided for the Person class we can see clearly that the class contains attributes and methods that are common to both classes Admin and Customer. For example, Admin and Customer will share attributes such as name, address and password. If the Person class does not exist then we will have to redefine all attributes and methods in both classes Customer and Admin that will cause repetition of code and data redundancy. Therefore, it is more efficient to create a parent class Person that includes all the common attributes between admins and customers. For this, the Customer class and the Admin class have become subclasses of the Person class, as a result, they inherit all the attributes and methods included in the class Person.

```
class Person(object):

    def __init__(self, name, password, address = [None, None, None, None]):
        self.name = name
        self.password = password
        self.address = address

    def get_address(self):
        return self.address

    def update_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def print_details(self):
        print("Name %s:" %self.name)
        print("Address: %s" %self.address[0])
        print("         %s" %self.address[1])
        print("         %s" %self.address[2])
        print("         %s" %self.address[3])
        print(" ")


    def check_password(self, password):
        if self.password == password:
            return True
        return False

    def profile_settings_menu(self):
        #print the options you have
         print (" ")
         print ("Your Profile Settings Options Are:")
         print ("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
         print ("1) Update name")
         print ("2) Print details")
         print ("3) Back")
         print (" ")
         option = int(input ("Choose your option: "))
         return option


    def run_profile_options(self):
        loop = 1
        while loop == 1:
            choice = self.profile_settings_menu()
            if choice == 1:
                name=input("\n Please enter new name\n: ")
                self.update_name(name)
            elif choice == 2:
                self.print_details()
            elif choice == 3:
                loop = 0
```

The Admin Class

```
from person import Person

class Admin(Person):

    def __init__(self, name, password, full_rights, address = [None, None,
None, None]):
        super().__init__(name, password, address)
        self.full_admin_rights = full_rights

    def has_full_admin_right(self):
        return self.full_admin_rights
```

The admin class is a template used to create admin instances. In fact, this class can instantiate two types of admins one with full admin rights whom can close customer accounts and a second admin type who can perform all admin operations but not closing customer accounts. As mentioned above Admin is a subclass of the Person class.

The statement `from person import Person` imports the module person and allows the Admin class to access the class definitions from the module. The class definition line (`class Admin(Person)`) signifies that the Admin class extends the Person class.

The Admin class is simple and small, but since it is a subclass of the Person class it inherits all the functionalities the Person class implements. The only special parameter the Admin class has is the self.full_admin_rights which is of type Boolean to state whether an admin instance has full administration rights or not. This is used in the BankSystem class to allow only administrators with full admin rights to close customers account i.e. remove customers from the system. The Admin class has one method has_full_admin_right(self) that returns the Boolean value True or False to indicate if the admin has full admin right or not.

The Customer class

This class Customer is a subclass of the Person class. The following is the code for the Customer class.

```
from person import Person

class Customer(Person):

    def __init__(self, name, password, address = "None, None, None, None"):
        super().__init__(name, password, address)

    def open_account(self, account):
        self.account = account

    def get_account(self):
        return self.account

    def print_details(self):
        super().print_details()
        bal = self.account.get_balance()
        print('Account balance: %.2f' %bal)
        print(" ")
```

In a similar way like the Admin class, Customer class also extends the Person class. The Customer class has three additional methods defined in it. The method open_account implements the composition relationship between the Customer class and the Account class. First, a customer object will be initialised at the class BankSystem, then, the object account is created and assigned to the customer instance using the Customer class's open_account(self, account) method. This method allows the customer object to encapsulate the account object. This ensures few advantages to the customer class, including direct access to the account in order to perform operations on the account (e.g. deposit money, withdraw money etc.), also it provides extensibility to the system as the system can be extended in the future to allow a customer to hold multiple bank accounts of different types!

Since the customer instance acts as the container of one or more bank account instance/s, then, the system will have to first identify the customer instance in order to perform any bank account operations to that particular customer.

Calling the method get_account(self) on the customer instance will supply the bank account instance for that customer.

print_details(self) method prints the state of the customer by exploiting the inheritance relationship between the Customer and Person class, and the relationship between Customer and Account classes through composition. The method first executes super().print_details() to ask the parent class Person to print the profile data such as name and address – check the method print_details inside the module person.py. The method print_details(self) also call the method get_balance() on the contained account instance to print the customer's current account balance.

The Account class

```
class Account:

        def __init__(self, balance, account_no):
                self.balance = float(balance)
                self.account_no = account_no

        def deposit(self, amount):
                self.balance+=amount


        def print_balance(self):
                print("Your account balance is %.2f" %self.balance)

        def get_balance(self):
                return self.balance

        def get_account_no(self):
                return self.account_no

        def account_menu(self):
                #print the options you have
                 print (" ")
                 print ("Your Transaction Options Are:")
                 print ("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
                 print ("1) Deposit money")
                 print ("2) Check balance")
                 print ("3) Back")
                 print (" ")
                 option = int(input ("Choose your option: "))
                 return option

        def run_account_options(self):
                loop = 1
                while loop == 1:
                        choice = self.account_menu()
                        if choice == 1:
                                amount=float(input("::\nPlease enter amount
to be deposited\n: "))
                                deposit = self.deposit(amount)
                                self.print_balance()
                        elif choice == 2:
                                balance = self.print_balance()
                        elif choice == 3:
                                loop = 0
                print ("Exit account operations")
```

Looking at the `__init__` method of the Account class, it is clear that an account can not be created without two input parameters, and these are the initial balance and the account number. This means that these two values should be defined by the class that initialises the account objects. The class Account provides basic accessor methods to retrieve and print account balance and account no.

The method `account_menu(self):` is a value returning method that prints the options the user can have for his/her account. This method also uses the `input` function to record and return the option the user wants to perform using integer values. For example, if the user inputs the value 1 it means he/she wants to deposit money.

The method `run_account_options(self):` consist of a while loop that is set to 1 before entering the loop. This means the while loop will continue iterating as long as the value for the variable loop is 1, but, once the value becomes anything else, the loop will terminate. This is why option 3 "Back" assign the loop the value 0, hence, the loop terminates and Python interpreter jumps back to the loop inside the customer_menu(self) or Admin menu at the class BankSystem. The while loop is responsible first print user menu options, then the asks the user to provide an integer value that represents on of the options displayed in the method account_menu as discussed above.

In the while loop there is a decision structure that checks the returned value from the method account_menu() and execute the desired functions accordingly. For example, if the customer choses to deposit money, then using the value 1 the system should ask the user for the amount to deposit into the account balance. Finally, the system will call the method print_balance() to show the latest balance the customer has after depositing the money.

The BankSystem class

The BankSystem class is the main class that handle most of the core functionalities of the system.

It is the container that holds and deals with Customer, Account and Admin classes, and this is why the class imports these three class as depicted in the beginning of the source code of the BankSystem class as shown below:

```
from customer import Customer
from admin import Admin
from account import Account
```

In this class, all customer and admin objects are instantiated and stored in lists customers_list = [] and admins_list = [] inside the BankSystem class. These lists are visible and globally accessible to all the methods of the class BankSystem.

To run the banking system we have to create a single instance of the class BankSystem which will call the __init__ that will create empty lists for the purpose of storing the admin and customer instances in these lists. The method load_bank_data(), which is called inside the __init__ method of the BankSystem class, will load 4 customer instances and 2 admin instances and store them in the lists accordingly. This is done purely because we need to test the system with basic data. All these objects are hard coded, but to achieve higher

marks for this assessment i.e. 70% and above, you will need to instantiate objects from data stored in csv files.

```
    def load_bank_data(self):
        customer_1 = Customer("Adam", "1234", ["14", "Wilcot Street",
"Bath", "B5 5RT"])
        account_no = 1234
        account_1 = Account(5000.00, account_no)
        customer_1.open_account(account_1)
        self.customers_list.append(customer_1)

        customer_2 = Customer("David", "password", ["60", "Holborn Via-
duct", "London", "EC1A 2FD"])
        account_no+=1
        account_2 = Account(3200.00,account_no)
        customer_2.open_account(account_2)
        self.customers_list.append(customer_2)


        customer_3 = Customer("Alice", "MoonLight", ["5", "Cardigan
Street", "Birmingham", "B4 7BD"])
        account_no+=1
        account_3 = Account(18000.00,account_no)
        customer_3.open_account(account_3)
        self.customers_list.append(customer_3)


        customer_4 = Customer("Ali", "150A",["44", "Churchill Way West",
"Basingstoke", "RG21 6YR"])
        account_no+=1
        account_4 = Account(40.00,account_no)
        customer_4.open_account(account_4)
        self.customers_list.append(customer_4)


        admin_1 = Admin("Julian", "1441", True, ["12", "London Road", "Bir-
mingham", "B95 7TT"])
        self.admins_list.append(admin_1)

        admin_2 = Admin("Eva", "2222", False, ["47", "Mars Street", "New-
castle", "NE12 6TZ"])
        self.admins_list.append(admin_2)
```

From the above 4 customer instances that have been created. It is clear that each customer instance has been created by providing 3 argument values name, password and customer address. The address is passed as a list of 4 String items, and each item represents the following information house number, street name, city name and post code.

Once the customer instance is created, an account will be created for that customer. Thus, two input parameters “balance” and “account number” will be used to instantiate a new bank account instance for each customer instance. To implement the composition relationship between a customer and his/her account, we have to call the method

open_account(Account) on the customer instance to allow that customer to hold a bank account.

Similar to customer instance, two admin instances have been created. To instantiate an admin instance from the Admin class 4 arguments should be passed to the Admin initialiser method of the Admin class. These are name, password, address and a Boolean value True/False that defines if the an admin has full administration rights or not.

Note, every time a customer instance or an admin instance is created it is added to the class BankSystem's respective list. For example, `self.customers_list.append(customer_3).` Here, the function append is called on the customer_list to store the instance customer_3 and located the end of the customers_list[].

The method load_bank_data (self) will instantiate and store all customer and admin instances in the BankSystem. Once the __init__ method is complete, and the bank system instance is created and assigned the variable app, the method run_main_option() is called to provide the user with the main menu of the programme. The menu will provide a welcome message and 3 options and these are as follows (See Figure 2)

```
>>> ================================ RESTART ================================
>>>

Welcome to the Python Bank System

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1) Admin login
2) Customer login
3) Quit Python Bank System

Choose your option:
```

Figure 2: Banking system main menu

The user will need to select whether to login as an admin or as a customer or quit from the bank system.

If the user inputs number 2 to log in as a customer, the user will be prompted to provide his/her name and password. Once the name and password have been entered, they will be used as input parameters to call the method self.customer_login(name, password). This method will first call another method search_customers_by_name(self, customer_name) to search for the customer in the customer_list. The method uses the input paramter customer_name to retrieve the customer object with the name that matches the customer_name parameter. The method uses a loop to iterate through the customers_list, and for each customer instance the method Customer.get_name() is called to retrieve the customer name which will be used for the comparison purpose. If the customer instance with the desired name is found then the inserted password will be checked against the

stored password in the customer instance. This has been implemented in the method Customer.check_password(password). This methods returns True if the password matches, otherwise, it will return False. If the password is correct, then, the software will call the method run_customer_options() on the found customer instance. The method run_customer_options()will provide the customer will the available banking options the system offers its customers.

To be able to perform the customer login process, the system should implement the methods customer_login(self, name, password) and search_customers_by_name(self, customer_name) that exist in the BankSystem class.

The partial implementation provided in the Moodle has some place markers (e.g. STEP A.1, STEP A.2, STEP A.3 and so on) in BankingSystem class where you need to add code to proceed with the implementation of the Banking System. In the following instructions are given that you can follow to add the needed code.

STEP A.1 Customer login function

First let us implement the method customer_login(self, name, password) located inside the BankSystem class . Copy and paste the following code inside the method customer_login(self, name, password). Remember to delete the statement pass before placing your code. Make sure the indentation is kept the same as shown in the code provided, because pasting text to Python Idle may corrupt the spacing and indentation of the source code.

```
#STEP A.1
found_customer = self.search_customers_by_name(name)
if found_customer == None:
    return("\n The customer has not been found!\n")
else:
    if (found_customer.check_password(password) == True):
        self.run_customer_options(found_customer)
    else:
        return("you have input a wrong password")
```

STEP A.2 – Search for a customer object

As discussed above, self.customer_login(name, password) method should call the search_customers_by_name(self, customer_name) to check whether the customer already exist in the system or not. Hence, for the customer login process to complete, we also need to implement the method search_customers_by_name(self, customer_name). Now,

you need to copy and paste the following code inside the method search_customers_by_name(self, customer_name). Remember to remove the pass statement beforehand.

```
        #STEP A.2
        found_customer = None
        for a in self.customers_list:
            name = a.get_name()
            if name == customer_name:
                found_customer = a
                break
        if found_customer == None:
            print("\nThe customer %s does not exist! Try again...\n" %cus-
tomer_name)

        return found_customer
```

STEP A.3 and STEP A.4

An admin will use a similar login process to a customer when attempting to use the banking system. Following the above examples in STEP A.1 and A.2, you will be able re-adjust them to work for admin_login(self, name, password) and search_admin_by_name(self, name) .

Note: No code is given here, as the code given for STEP A.1 and STEP A.2 should be used and adjusted to implement admin login process.

Implementation of administrative operations for Admins

Once STEP A3 and STEP A4 are completed correctly, then, when an admin attempts to login using the correct login details then he / she should be able to successfully get into the system and see the admin menu which offers different kind of administrative operations as a result to calling the method run_admin_options(self, admin) inside the class BankSystem. See Figure 3.

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>

Welcome to the Python Bank System

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1) Admin login
2) Customer login
3) Quit Python Bank System

Choose your option: 1

Please input your admin name:Julian
Please input your password: 1441
processing ......


   *logged in successfully!*

Your Transaction Options Are:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1) Transfer money
2) Customer account operations
3) Customer profile settings
4) Admin profile settings
5) Delete customer
6) Print all customers detail
7) Sign out

Choose your option:
```

Figure 3: Admin main menu

Although the admin options are visible, but when selected, most do not function because they are not implemented yet! Here, we will help you implementing some of critical functions that an admin should be able to exercise in the banking system. These functions are as follows

- STEP A.5 Customer account operations (option 2)
- STEP A.6 Customer profile settings (option 3)
- STEP A.7 Admin profile settings (option 4)
- STEP A.8 Delete a customer (option 5)
- STEP A.9 Print all customers (option 6)

STEP A.5 – Admins performing customer account operations

First, you need to know that the customer account operations exist inside the Account class, hence, account objects should provide the admin with all the necessary methods to perform operations for withdrawing and depositing money. It is worth noting that the provided partial implementation for the Banking System only implements few account operations such as print account balance, return the account balance and to deposit money.

As an admin, to be able to call these functions you need to first retrieve the customer object in which the customer bank account is stored. This will require the admin to search for the customer object by the customer name, and once the customer object is found, then the system will call the function get_account(self) to return the customer bank account. Once the system obtains the bank account object, then it runs its account options by calling the method run_account_options(self) which exits inside the class Account. When STEP A.5 is implemented correctly then the system should display the following text based the account options (See Figure 4):

```
Please input customer name :
Ali

Your Transaction Options Are:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1) Deposit money
2) Withdraw money
3) Check balance
4) Back

Choose your option:
```

Figure 4: Account transaction options

To implement the above you need to copy and paste the following code just below the comment #STEP A.5. which exist inside the method run_admin_options(self, admin), more specifically, in the statement block of the "elif" that checks if the user input is equal to number 2 to implement the option "Customer account operations".

```
#STEP A.5
customer_name = input("\nPlease input customer name :\n")
customer = self.search_customers_by_name(customer_name)
if customer != None:
    account = customer.get_account()
    if account != None:
        account.run_account_options()
```

STEP A.6 – Admin performing customer profile settings

Similar to the previous STEP A.5, in this STEP A.6 an admin will have to provide the name of the customer to access and perform profile settings on that customer instance. This operation is more straightforward than in STEP A.5 because once the system find the customer instance it immediately calls the method run_profile_options, inherited from the Person class, to show all available customer profile settings such as update name, address and password. Note, the partial implementation of the BankSystem only implements the functions to update the person name and to print the profile details. It is the student responsibility to implement other functions such as updating person's address and password etc.

In the method run_admin_options(self, admin), copy and paste the code below inside the functional block of the "elif" statement that implements option 3 "Customer profile settings".

```
#STEP A.6
  customer_name = input("\nPlease input customer name :\n")
  customer = self.search_customers_by_name(customer_name)
  if customer != None:
      customer.run_profile_options()
```

Place the above code below the comment # STEP A.6 in the source code. You also need to delete the pass statement.

STEP A.7 – Admin performing admin settings

For this option, there is no need to search for the admin instance to run its profile settings. This is because when the admin login successfully into the system, the system will call the method run_admin_options(self, admin) which should pass the logged in admin object as a parameter for this method. The admin object parameter will have a local scope to the method, hence, will always be accessible by all functions inside the method run_admin_options(self, admin).

To run admin profile settings, we simply need to call the method run_profile_options(self) on the current admin instance, a method that is inherited from the Person class.

The following code is what you need to have inside the method admin_options(self) where option 4 is implemented via the elif statement.

```
                #STEP A.7
                admin.run_profile_options()
```

Again, remember to remove the statement pass after placing the code above.

STEP A.8 – Admin deleting a customer from a system

To delete a customer from the system the admin needs to have full administration rights. This can be checked by calling the method has_full_admin_right(self) inside the Admin class. If the admin has full rights, then the system search for the customer by the name, and see if the customer actually exists or not. If the method search_customers_by_name returns a customer object denoting that the customer exists then the object gets removed by calling the function method remove on the self.customers_list. Note: because the Customer class owns the object Account via composition relationship, then removing a customer object will also mean deleting the account resides inside the customer instance.

Copy and paste the code below into the empty block where the comment #STEP A.8 is placed

```
                #STEP A.8
                if admin.has_full_admin_right() == True:
                    customer_name = input("\nPlease input customer name you
want to delete :\n")
                    customer_account = self.search_customers_by_name(cus-
tomer_name)
                    if customer_account != None:
                        self.customers_list.remove(customer)
                else:
                    print("\nOnly administrators with full admin rights can
remove a customer from the bank system!\n")
```

STEP A.9 – Admin printing all customer details

To print all customers details a for-loop is used to iterate through the entire self.customers_list from the start to the end of the list. At each iteration, one customer object is retrieved on which the method print_details(self) is called to print the person common details such as name and address, but also the method utilises the bank account instance, stored at the Customer class, to obtain the account balance by calling its method get_balance(self). All this is implemented in the method print_all_accounts_details already

defined in the BankSystem class. To implement this last option you only need to call the method print_all_accounts_details(self) in the statement block of the “elif” keyword that implements the option 6 of the admin menu.

To call the method in the source code place the method self.print_all_accounts_details() just under the comment #STEP A.9, and remove the pass statement.

```
#STEP A.9
self.print_all_accounts_details()
```

Test this function by login as Admin and select option 6 that should print all customer details.

Summary

If all the STEPs above have been implemented correctly, you should have a basic running system that can do the following

- Login
- Deposit money
- Check bank balance
- View customer details - partial implementation
- Update customer information - partial implementation

- Create the necessary classes and functions which allow admins to perform the following tasks:
  - Login
  - Search for a particular customer to perform various banking operations on his/her account i.e. check balance, deposit/withdraw money etc.
  - Close account i.e. remove customer from the system
  - Update customer information – partial implementation
  - Update admin own information - partial implementation
  - Print a customer details
  - Print all customers details

Please note the following:

- You have to use the provided partial implementation of the banking system as a starting point to complete this assessment.
- No comments were added to the source code.

- No exception handling have been implemented in the code, hence, the system will crash if the user inserts unexpected input. For example, if a user input a string instead of an integer for the selection of a menu option.
- The source code may contain errors or logical issues, and it is the student responsibility to identify and fix these errors if they exist.