ATM based project

MonkePay

Severhin Oleksandr

Yashchenko Pavlo

Bolotov Yehor

created by DonkeyKong in 2022

Plan

Plan

1 Introduction

2 Our expectations,visions

Plan

1 Introduction

2 Our expectations,visions

3 Class planning

Plan

1 Introduction

2 Our expectations,visions

3 Class planning

4 How we implemented it

Plan

1 Introduction

2 Our expectations,visions

3 Class planning

4 How we implemented it

5 Errors we encountered an the test

Plan

1 Introduction

2 Our expectations,visions

3 Class planning

4 How we implemented it

5 Errors we encountered an the test

6 What we got in the end

Plan

1 Introduction

2 Our expectations,visions

3 Class planning

4 How we implemented it

5 Errors we encountered an the test

6 What we got in the end

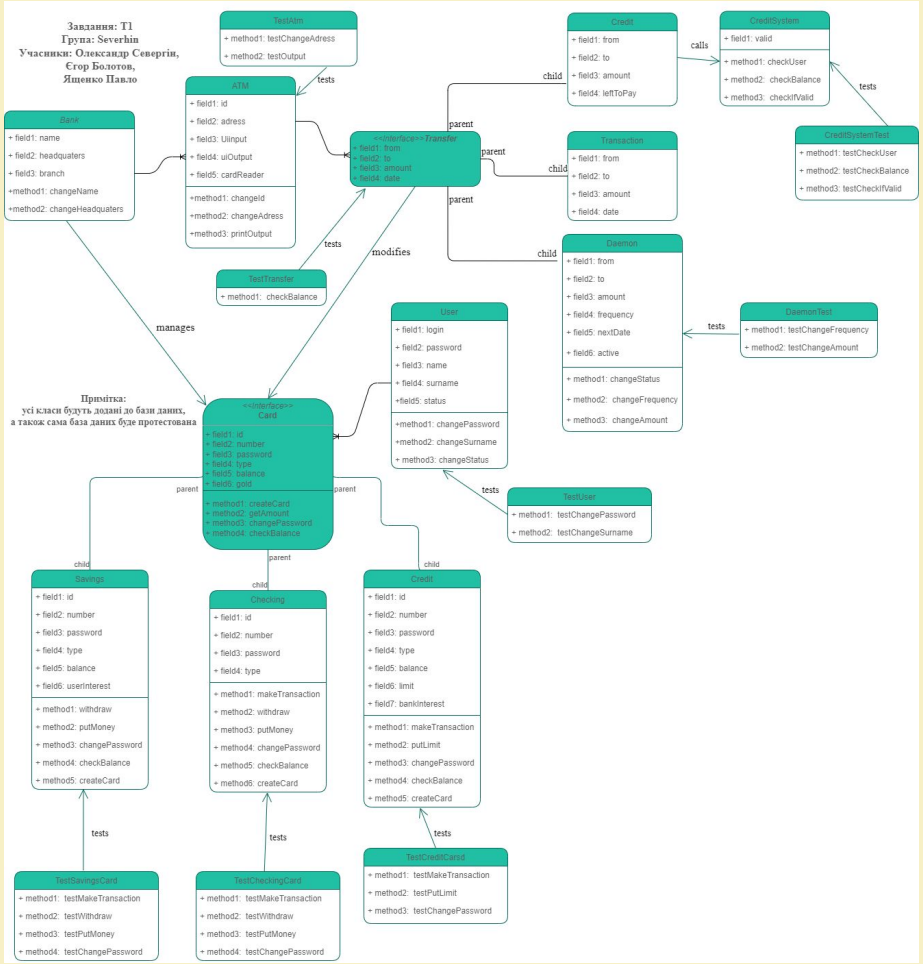
7 What's next? What could be changed? Consclusion

MonkePay
ATM System

Our expectations,visions

1. Знати / покласти / перевести гроші
2. Використання кредитних коштів
3. Виставити максимум на використання кредитних коштів
4. Авторизуватися
5. Змінити ПІН-код
6. Заблокувати картку
7. Видалити (знищити) картку
8. Оформити картки трьох типів

Class planning



Implementation

User

```python
from ConnectToDB import ConnectToDb as con

"""
    This class is responsible for an user entity
"""
class User:
    id = int(0)
    login = str("default")
    password = str("default")
    money = float(0)
    restore = False

    """
    Constructor
    :param: self, login, password, money
    :type: User, str, str, int
    :returns: nothing
    """
    def __init__(self, login, password, money, restore):
        assert type(login) is str, "Login must be a string!"
        assert type(password) is str, "Password must be a string!"
        assert type(money) is float or type(money) is int, "Money must be a float or an int!"
        if restore == False:
            assert self.checkLogin(login) == True, "This login is already used! Sign in or use another login!"
            assert len(password) >= 8, "Password must be longer than 7 symbols!"
            self.id = con.getLastId("user") + 1
            self.login = login
            self.password = password
            self.money = money
            self.createUser()
        else:
            id = int(self.findUserId(login))
            self.restoreUser(id, login, password, money)
```

```
"""
This method restored data about the user from the database
:param: self, id, login, password, money
:type: User, int, str, str, int
:returns: nothing
"""
```

Oleksandr Severhin +1

```
def restoreUser(self, id, login, password, money):
        self.id = id
        self.login = login
        self.password = password
        self.money = money


"""
This method finds a user id
:param: self, login
:type: User, str
:returns: id
:rtype: int
"""
```

Oleksandr Severhin

```
def findUserId(self, login):
    query = "SELECT id FROM user WHERE login = '" + str(login) + "';"
    return tuple(con.executeReturn(query)).__getitem__(0)[0]
```

```
"""
Checks if User with login given is already in a database
:param: self, login
:type: User, str
:returns: True or False
:rtype: bool
"""
```

Oleksandr Severhin

```
def checkLogin(self, login):
    query = "SELECT * FROM user WHERE login = '" + login + "';"
    res = con.executeReturn(query)
    if res.__len__() == 0:
        return True
    else:
        return False


"""
This method checks password when logging in
:param: self, password, id
:type: User, str, int
:returns: True/False
:rtype: bool
"""
```

Oleksandr Severhin +1

```
def checkPassword(self, password, id):
    query = "SELECT password FROM user WHERE id = '" + str(id) + "';"
    res = con.executeReturn(query)
    if res.getitem(0)[0] == password:
        return True
    return False
```

Account

```
    else:
        id = self.findAccountId(user_id)
        number = self.findAccountNumber(user_id)
        self.restoreAccount(id, name, surname, number, status, user_id)
```

```
from ConnectToDB import ConnectToDb as con
import random

"""
    This class is responsible for an Account entity
"""
 Oleksandr Severhin + 1
class Account:
    id = int(0)
    name = str("default")
    surname = str("default")
    number = int(0)
    status = str("default")
    user_id = int(0)
    bank_id = int(1)

    """
    Constructor
    :param: self, name, surname, status, user_id, bank_id
    :type: Account, str, str, str, int, int
    :returns: nothing
    """
     Oleksandr Severhin
    def __init__(self, name, surname, status, user_id, restore):
        assert type(name) is str, "Name must be a string!"
        assert type(surname) is str, "Surname must be a string!"
        assert type(status) is str, "Status must be a string!"
        assert type(user_id) is int, "You must give int that is user_id!"
        assert self.validName(name) == True, "Name is invalid! Enter the valid one"
        assert self.validSurname(surname) == True, "Surname is invalid! Enter the valid one"
        assert self.validStatus(status) == True, "Stauts is invalid! Enter the valid one"
        if restore == False:
            self.id = con.getLastId("account") + 1
            self.name = name
            self.surname = surname
            self.number = self.generateNumber()
            self.status = status
            self.user_id = int(user_id)
            self.createAccount()
```

```
"""
This method finds an account id
:param: self, user_id
:type: Account, int
:returns: id
:rtype: int
"""
Oleksandr Severhin
def findAccountId(self, user_id):
    query = "SELECT id FROM account WHERE user_id = '" + str(user_id) + "';"
    return tuple(con.executeReturn(query)).__getitem__(0)[0]

"""
This method finds an account number
:param: self, user_id
:type: Account, int
:returns: id
:rtype: int
"""
Oleksandr Severhin
def findAccountNumber(self, user_id):
    query = "SELECT number FROM account WHERE user_id = '" + str(user_id) + "';"
    return tuple(con.executeReturn(query)).__getitem__(0)[0]
```

```
"""
Constructor for restoring Account
:param: self, id, name, surname, status, user_id, bank_id
:type: Account, int, str, str, str, int, int
:returns: nothing
"""
Oleksandr Severhin
def restoreAccount(self, id, name, surname, number, status, user_id):
    self.id = id
    self.name = name
    self.surname = surname
    self.number = number
    self.status = status
    self.user_id = user_id

"""
Creates Account in database
:param: self
:type: Account
:returns: nothing
"""
Oleksandr Severhin
def createAccount(self):
    query = "INSERT INTO account (id, name, surname, number, status, user_id, bank_id) VALUES (%s, %s, %s, %s, %s, %s, %s);"
    val = (self.id, self.name, self.surname, self.number, self.status, self.user_id, self.bank_id)
    con.executeWithVal(query, val)
```

Card

```python
import random

"""
    This class is responsible for Card entity
"""
Oleksandr Severhin
class Card:
    id = int(0)
    number = int(0)
    password = int(0)
    type = str("default")
    balance = float(0)
    limit = float(0)
    valid = bool(0)
    leftToPay = int(0)          # left to pay for the credit
    account_id = int(0)

    """
    Constructor
    :param: self
    :type: Card
    :returns: nothing
    """
    Oleksandr Severhin
    def __init__(self, password, cardType, account_id, restore):
        assert type(password) is int, "Card password must be an int!"
        assert self.validPassword(password) == True, "Password must be be 4 numbers from 0 to 9!"
        assert self.validType(cardType) == True, "You've entered invalid Card type!"
        assert type(account_id) is int, "You must give an int (account_id) as the last parameter!"
        if restore == False:
            assert self.thisCardExists(cardType, account_id) == True, "You can't create the card of type " + str(cardType) + " as you already have one!"
            self.id = con.getLastId("card") + 1
            self.number = self.generateNumber()
            self.password = password
            self.type = cardType
            self.balance = float(0)
            self.valid = True
            self.limit = float(0)
            self.account_id = account_id
            self.createCard()
```

```python
"""
    This class is responsible for Checking (Card) entity
"""
class Checking(Card):
    id = int_(0)
    number = int_(0)
    password = int_(0)
    type = str_("checking")
    balance = float_(0)
    limit = float_(0)
    valid = bool_(1)
    account_id = int_(0)

    """
    Constructor
    :param: self, password, cardType, balance, limit, account_id
    :type: Checking, int, str, float/int, float/int, int
    :returns: nothing
    """
    def __init__(self, password, cardType, account_id, restore):
        if restore == False:
            super(Checking, self).__init__(password, "checking", account_id, restore)
        if restore == True:
            id = self.findCardId(account_id, cardType)
            number = self.findCardNumber(account_id, cardType)
            balance = self.findCardBalance(account_id, cardType)
            self.valid = True
            limit = self.findCardLimit(account_id, cardType)
            leftToPay = self.findLeftToPay(account_id, cardType)
            self.restoreCard(id, number, password, cardType, balance, limit, leftToPay, account_id)
```

```python
"""
    This class is responsible for Credit (Card) entity
"""
Oleksandr Severhin +1
class Credit(Card):
    id = int_(0)
    number = int_(0)
    password = int_(0)
    type = str_("credit")
    balance = float_(0)
    limit = float_(0)
    valid = bool_(1)
    leftToPay = int_(0)          # left to pay for the credit
    bankInterest = float_(10)    #
    account_id = int_(0)

    """
    Constructor
    :param: self, password, cardType, balance, limit, account_id
    :type: Checking, int, str, float/int, float/int, int
    :returns: nothing
    """
    Oleksandr Severhin
    def __init__(self, password, cardType, account_id, restore):
        if restore == False:
            super(Credit, self).__init__(password, "credit", account_id, restore)
        if restore == True:
            id = self.findCardId(account_id, cardType)
            number = self.findCardNumber(account_id, cardType)
            balance = self.findCardBalance(account_id, cardType)
            self.valid = True
            limit = self.findCardLimit(account_id, cardType)
            leftToPay = self.findLeftToPay(account_id, cardType)
            self.restoreCard(id, number, password, cardType, balance, limit, leftToPay, account_id)
```

```python
"""
    This class is responsible for Savings (Card) entity
"""
class Savings(Card):
    id = int(0)
    number = int(0)
    password = int(0)
    type = str("savings")
    balance = float(0)
    process = 0
    limit = float(0)
    valid = bool(1)
    account_id = int(0)
    userInterest = float(8)

    """
    Constructor
    :param: self, password, cardType, balance, limit, account_id
    :type: Savings, int, str, float/int, float/int, int
    :returns: nothing
    """
    def __init__(self, password, cardType, account_id, restore):
        if restore == False:
            super(Savings, self).__init__(password, "savings", account_id, restore)
        if restore == True:
            id = self.findCardId(account_id, cardType)
            number = self.findCardNumber(account_id, cardType)
            balance = self.findCardBalance(account_id, cardType)
            self.valid = True
            limit = self.findCardLimit(account_id, cardType)
            leftToPay = self.findLeftToPay(account_id, cardType)
            self.restoreCard(id, number, password, cardType, balance, limit, leftToPay, account_id)
            if self.balance != 0:
                self.process = Daemon(self.number, self.number, self.balance, 1, self.id, self.account_id)
```

Transfer

```
self.toCard = toCard
self.amount = amount
self.date = self.getTime()
self.transferType = transferType
self.active = bool(1)
self.leftToPay = float(leftToPay)
self.frequency = float(frequency)
self.card_id = card_id
self.card_account_id = card_account_id
self.createTransfer()
```

```
"""
    This class is responsible for a Transfer entity
"""
Oleksandr Severhin
class Transfer:
    id = int(0)
    fromCard = int(0)
    toCard = int(0)
    amount = float(0)
    date = str("default")
    transferType = str("default")
    active = bool(0)
    frequency = int(0)
    card_id = int(0)            # from which card (id) is transfer
    card_account_id = int(0)    # id of the account that has a card from which transfer takes place
    atm_id = int(1)             # id of the atm where transaction is taken
    atm_bank_id = int(1)        # bank id that has an atm

    """
    Constructor
    :param: self, fromCard, toCard, amount, date, transferType, card_id, card_account_id, atm_id, atm_bank_id
    :type: Transfer, int, int, float/int, date, str, int, int, int, int
    :returns: nothing
    """
    Oleksandr Severhin
    def __init__(self, fromCard, toCard, amount, transferType, leftToPay, frequency, card_id, card_account_id):
        assert self.cardExists(fromCard) == True, "Card from which you want to make a transfer doesn't exist!"
        assert self.cardExists(toCard) == True, "Card on which you want to make a transfer doesn't exist!"
        assert amount > 0, "You can't make a Transaction with amount less than 1!"
        if transferType == "transaction" or transferType == "withdraw":
            assert self.getBalance(fromCard) >= amount, "You can't transfer more money than you have on the Card!"
        assert self.validType(transferType) == True, "You can't make a transfer with type different from Transaction, Daemon or Credit"
        assert type(leftToPay) == float or int, "You can't use other type than float or int for initialising leftToPay!"
        assert type(frequency) == float or int, "You can't set frequency for a daemon using other type than float or int!"
        self.id = con.getLastId("transfer") + 1
        self.fromCard = fromCard
```

```
"""
    This class is responsible for a Transaction entity
"""
Oleksandr Severhin +1
class Transaction(Transfer):
    id = int(0)
    fromCard = int(0)
    toCard = int(0)
    amount = float(0)
    date = str("default")
    type = str("transaction")
    active = bool(0)
    card_id = int(0)             # from which card (id) is transfer
    card_account_id = int(0)     # id of the account that has a card from which transfer takes place
    atm_id = int(1)              # id of the atm where transaction is taken
    atm_bank_id = int(1)         # bank id that has an atm

    """
    Constructor
    :param: self, fromCard, toCard, amount, date, transferType, card_id, card_account_id, atm_id, atm_bank_id
    :type: Transaction, int, int, float/int, date, str, int, int, int, int
    :returns: nothing
    """
    Oleksandr Severhin +1
    def __init__(self, fromCard, toCard, amount, card_id, type, card_account_id):
        super(Transaction, self).__init__(fromCard, toCard, amount, type, 0, 0, card_id, card_account_id)
        if type == "transaction":
            self.transaction(fromCard, toCard, amount)
            if self.getCardType(toCard) == "credit":
                if self.getLeftToPay(toCard) == 0:
                    self.creditInactive(toCard, card_account_id)
        if type == "withdraw":
            self.withdraw(fromCard, amount)
            self.userChangeMoney(amount, type, card_account_id)
        if type == "putMoney":
            self.putMoney(toCard, amount)
            self.userChangeMoney(amount, type, card_account_id)
            if self.getCardType(toCard) == "credit":
```

```
if self.getCardType(toCard) == "credit":
    if self.getLeftToPay(toCard) == 0:
        self.creditInactive(toCard, card_account_id)
```

```
"""
    This class is responsible for a Credit Entity
"""
Oleksandr Severhin +1
class Credit(Transfer):
    id = int(0)
    fromCard = int(0)
    toCard = int(fromCard)
    amount = float(0)
    date = str("default")
    type = str("credit")
    active = bool(0)
    bankInterest = int(10)       # bank interest
    card_id = int(0)             # from which card (id) is transfer
    card_account_id = int(0)     # id of the account that has a card from which transfer takes place
    atm_id = int(1)              # id of the atm where transaction is taken
    atm_bank_id = int(1)         # bank id that has an atm

    """
    Constructor
    :param: self, fromCard, toCard, amount, date, transferType, card_id, card_account_id, atm_id, atm_bank_id
    :type: Credit, int, int, float/int, date, str, int, int, int, int
    :returns: nothing
    """
    Oleksandr Severhin
    def __init__(self, fromCard, toCard, amount, card_id, card_account_id):
        try:
            self.checkIfValid(toCard, amount)
        except Exception as e:
            raise AssertionError(e)
        leftToPay = amount * float(1.1) # amount of the money with bank interest
        super(Credit, self).__init__(fromCard, toCard, amount, "credit",  leftToPay, 0, card_id, card_account_id)
        self.changeBalance(toCard, amount, True)
```

```
"""
    This class is responsible for a Daemon entity
"""
```
Oleksandr Severhin
```
class Daemon(Transfer):
    id = int(0)
    fromCard = int(0)
    toCard = int(0)
    amount = float(0)
    date = str("default")
    type = str("daemon")
    frequency = int(0)
    nextDate = str(0)
    increase = bool(0)
    active = bool(1)
    card_id = int(0)              # from which card (id) is transfer
    card_account_id = int(0)      # id of the account that has a card from which transfer takes place
    atm_id = int(1)               # id of the atm where transaction is taken
    atm_bank_id = int(1)          # bank id that has an atm

    """
    Constructor
    :param: self, fromCard, toCard, amount, date, transferType, card_id, card_account_id, atm_id, atm_bank_id
    :type: Credit, int, int, float/int, date, str, int, int, int, int
    :returns: nothing
    """
```
Oleksandr Severhin
```
    def __init__(self, fromCard, toCard, amount, frequency, card_id, card_account_id):
        super(Daemon, self).__init__(fromCard, toCard, amount, "daemon",  0, frequency, card_id, card_account_id)
```

ATM

```
"""
    This class is responsible for an ATM entity
"""
```
Oleksandr Severhin
```
class ATM:
    id = int(0)
    address = str("default")
    bank_id = int(1)

    """
    Constructor
    :param: self, address, bank
    :type: ATM, str, Bank
    :returns: nothing
    """
```
Oleksandr Severhin
```
    def __init__(self, address):
        assert type(address) is str, "Address must be a string!"
        assert self.checkAddress(address, self.bank_id) == True, "ATM with such address is already existing! Create an ATM with another address!"
        self.id = con.getLastId("atm") + 1
        self.address = address
        self.createAtm()
```

Bank

```
"""
    This class is responsible for a bank entity
"""
Oleksandr Severhin
class Bank:
    id = int(0)
    name = str("default")
    headquarters = str("default")
    branch = str("default")

    """
    Constructor
    :param: self, name, headquarters, branch
    :type: Bank, str, str, str
    :returns: nothing
    """
    Oleksandr Severhin
    def __init__(self, name, headquarters, branch):
        assert type(name) is str, "Name must be a string!"
        assert type(headquarters) is str, "Headquarters must be a string!"
        assert type(branch) is str, "Branch must be a string!"
        assert self.checkName(name) == True, "Bank with such name is already existing! Create a bank with another name!"
        self.id = con.getLastId("bank") + 1
        self.name = name
        self.headquarters = headquarters
        self.branch = branch
        self.createBank()
```

Tests

Test User

```python
""" Creating new User """
user = User("Testing", "11111111", 30000, False)
print("User:")
print(user)
print()

""" Changing user password """
""" Error as password is less than 8 symbols """
try:
    user.changePassword("1234567")
except Exception as e:
    print(e)

""" Success """
user.changePassword("12345678")
print("User:")
print(user)
print()
```

```
User:
id: 12, login: Testing, password: 11111111, money: 30000

You can't change the password on a new one with length < 8!
User:
id: 12, login: Testing, password: 12345678, money: 30000
```

Test Account

```
""" Creating user's account """
account = Account("Taras", "Shevchenko", "workless", user.id, False)
print("Account:")
print(account)
print()

""" Changing name and status of the account """
account.changeName("Taras Grygorovych")
account.changeStatus("working")
print("Account:")
print(account)
print()
```

```
Account:
id: 11, name: Taras, surname: Shevchenko, number: 9459559, status: workless, user_id: 12, bank_id: 1

Account:
id: 11, name: TarasGrygorovych, surname: Shevchenko, number: 9459559, status: working, user_id: 12, bank_id: 1
```

Test Checking

```python
""" Creating checking card """
checking = Checking(1111, "checking", account.id, False)
print("Checking Card:")
print(checking)
print()

""" Make a transaction with 0 money """
try:
    checking.makeTransaction(601226214, 2000)
except Exception as e:
    print(e)

""" Withdraw with 0 money """
try:
    checking.withdraw(2000)
except Exception as e:
    print(e)

""" Put money user has on the card """
""" Let's put more than the user has """
try:
    checking.putMoney(user.id, 2000000)
except Exception as e:
    print(e)
print("User:")
print(user)
print()
```

```python
""" Let's put money that the user has on the card """
checking.putMoney(user.id, 10000)
print("Checking:")
print(checking)
print()

"""Let's make a transaction """
checking.makeTransaction(601226214, 2000)
print("Checking:")
print(checking)
print()

""" Let's withdraw some money """
checking.withdraw(2000)
print("Checking:")
print(checking)
print("User:")
print(user)
print()

""" Put the card limit """
checking.changeLimit(4000)
print("Checking:")
print(checking)
print()

""" Not let's make a transaction on amount that is higher than the limit """
try:
    checking.makeTransaction(601226214, 4000)
except Exception as e:
    print(e)
print("Checking:")
print(checking)
print()
```

```
Checking Card:
id: 22, number: 886007466, password: 1111, type: checking, balance: 0.0, valid: True, limit: 0.0, leftToPay: 0, account_id: 11

You can't make Transaction with more money than you have on your card!
You can't withdraw more money than you have on your card!
User can't put on the Card more money that he has!
User:
id: 12, login: Testing, password: 12345678, money: 30000

Checking:
id: 22, number: 886007466, password: 1111, type: checking, balance: 10000.0, valid: True, limit: 0.0, leftToPay: 0, account_id: 11

Checking:
id: 22, number: 886007466, password: 1111, type: checking, balance: 8000.0, valid: True, limit: 0.0, leftToPay: 0, account_id: 11
```

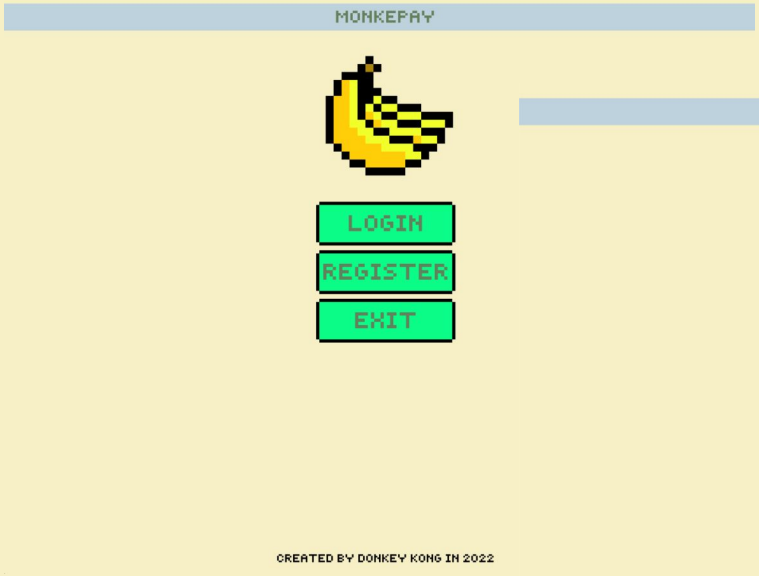
Test Credit

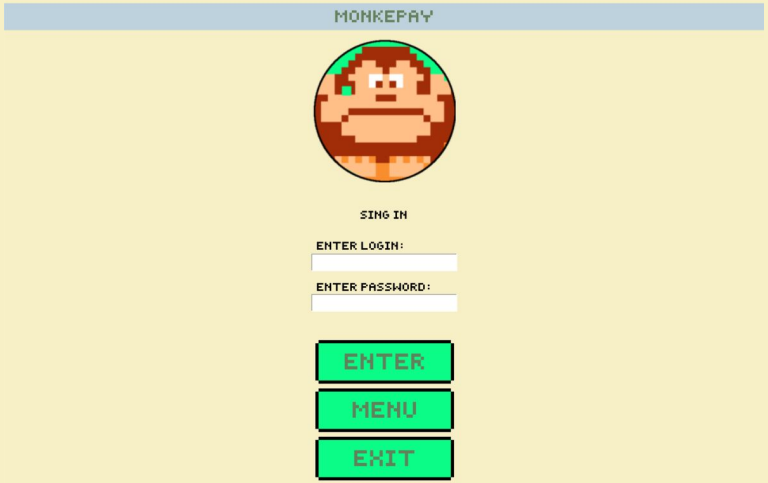
```
""" Change limit again """
credit.changeLimit(40000)
print("Credit:")
print(credit)
print()

""" Let's take a credit """
credit.takeCredit(30000)
print("Credit:")
print(credit)
print()

""" Make a transaction """
credit.makeTransaction(601226214, 10000)
print("Credit:")
print(credit)
print()

""" Withdraw money """
credit.withdraw(10000)
print("Credit:")
print(credit)
print()

""" Put money some money to decrease left to pay amount"""
credit.putMoney(user.id, 3000)
print("Credit:")
print(credit)
print()
```

```
""" Let's create a credit card """
credit = credit(2222, "credit", account.id, False)
print("Credit:")
print(credit)
print()

""" Let's do impossible things now """
try:
    credit.withdraw(2000)
except Exception as e:
    print(e)
print("Credit:")
print(credit)
print()

try:
    credit.makeTransaction(601226214, 2000)
except Exception as e:
    print(e)
print("Credit:")
print(credit)
print()

""" Let's change the credit limit """
credit.changeLimit(90000)
print("Credit:")
print(credit)
print()

""" Now, let's take a credit """
try:
    credit.takeCredit(89500)
except Exception as e:
    print(e)
print("Credit:")
print(credit)
print()
```

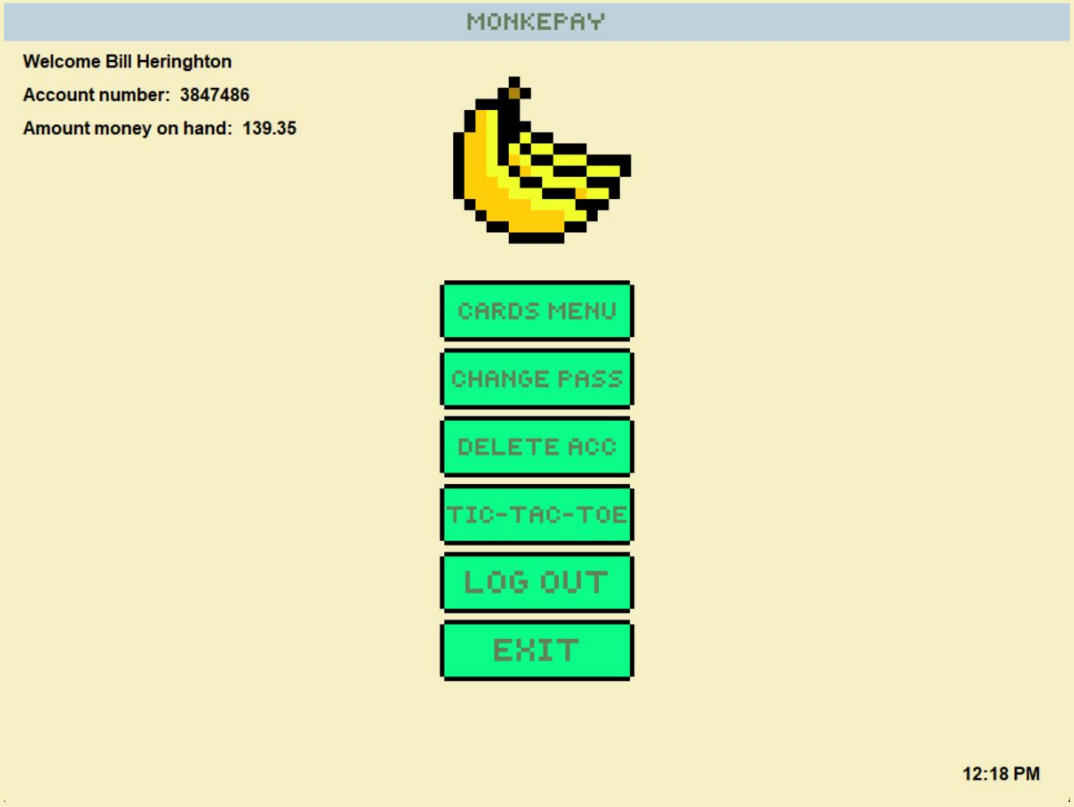
What we got in the end

What we got in the end

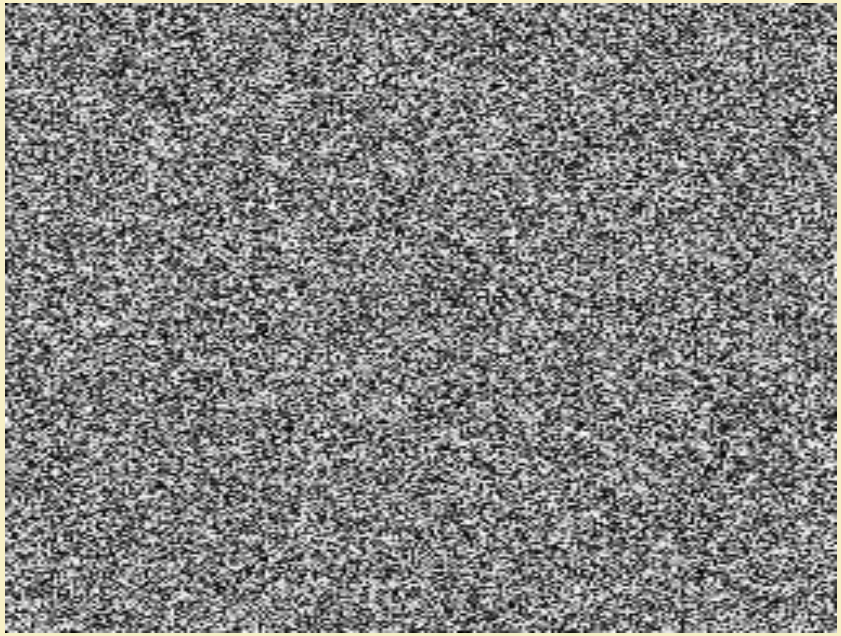




What we got in the end (after login)

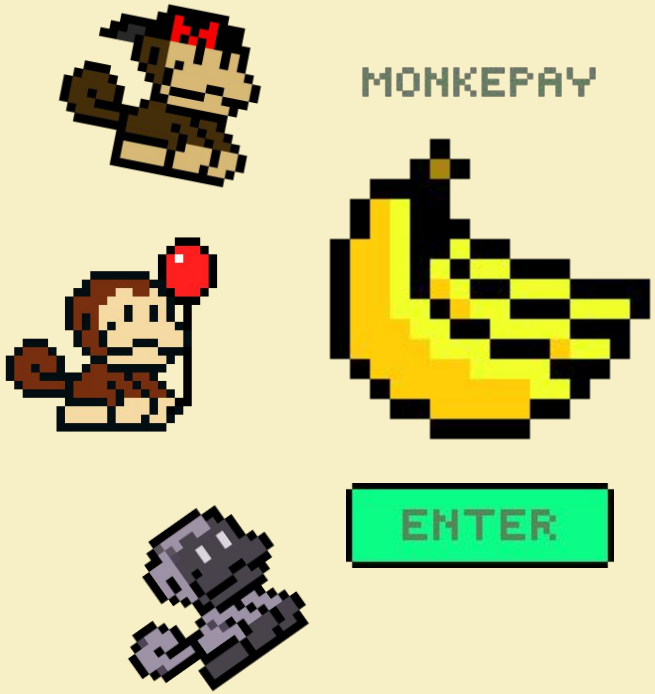


What we got in the end (main menu)



It seems that there was a problem,it turns out that the presentation of the interface will have to be carried out directly

What's next? What could be changed? Conclusion



End