Banking System Simulation
Program Manual

Date of Creation

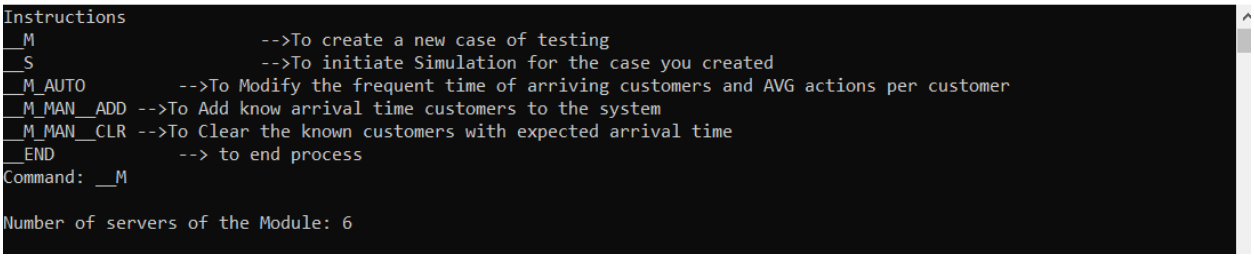
4/12/2022

Interface

The program comes with a command interface with given 6 commands for the user to control the program and simulations.

To run the program, we need to provide the program with the number of servers of our system which is by default = 0.

Command: __M



After that there are two ways to provide the system with information about incoming customers:

1. Specify a list of customers that will visit with expected time (in mins) **COMMAND: __M_MAN__ADD**

```
Command: __M_MAN__ADD

Number of customers: 3

Arrival time: 4

Number of actions needed by the customer 3

Arrival time: 5

Number of actions needed by the customer 6

Arrival time: 1

Number of actions needed by the customer 9

Command:
```

2. Specify the time between the arrival of each customer and the avg actions needed to be processed on each customer coming in. COMMAND: **__M_AUTO**

```
Command: __M_AUTO

Frequnt time for customers to come: 2

Number of actions for each customer: 3
```

The last thing that we could do other than modifying the values that we gave using the other commands (__MAN__ADD, __MAN__CLR)

Is to run simulation for a period in minutes COMMAND: **__S**

```
Command: __S

Enter the wanted simulation time in mins: 10
```

The simulation will print each server with its task for each minute of the given period of time we specified earlier

```
Enter the wanted simulation time in mins: 10

at minute 0
Server 1 : customer5
at minute 0
Server 2 : customer5
at minute 0
Server 3 : customer5
at minute 0
at minute 0
Server 5 : Idle
at minute 0
Server 6 : Idle
--------------------------------------------------------------------------------
at minute 1
Server 1 : customer3
at minute 1
Server 2 : customer3
at minute 1
Server 3 : customer3
at minute 1
Server 4 : customer3
at minute 1
Server 5 : customer3
at minute 1
Server 6 : customer3
--------------------------------------------------------------------------------
at minute 2
Server 1 : customer3
```

if the system can't handle the customers a message will be printed at the end of the simulation process

```
--------------------------------------------------------------------------------
Number of servers are not enough to handle the customer proparly
Please consider increamenting the number of the servers
Command:
```

Program Structure

The program code structure follows the **SOLID principle**. And next we will go through each file in the project.

CLASSES

Customer:

Customer.h:

```
class Customer
{
public:
    static int count;
    int expectedArrivalTime;
    int actions;
    int id;
    Customer();
    Customer(int expectedTime, int actionsNum);
    bool operator < (Customer const& obj);
};
```

The static attribute int count --> only exists to count the created objects and give the created customer object a valid unique id from the count.

The overloading of operator < done for sorting algorithm in the command file.cpp

The Customer.cpp file executes the Customer constructors and < operator overloading

Customer.cpp:

```
Customer::Customer() {
    this->id = count;
    this->actions = 0;
    this->expectedArrivalTime = 0;
}

Customer::Customer(int expectedTime, int actionsNum)
{
    Customer::count++;
    this->id = count;
    this->actions = actionsNum;
    this->expectedArrivalTime = expectedTime;
}
bool Customer::operator<(Customer const& obj)
{
    return this->expectedArrivalTime < obj.expectedArrivalTime;
}
 int Customer::count = 0;
```

Queue:

Queue.h:

```
#include "Customer.h"
#include "Node.h"
#include <iostream>
using namespace std;

class Queue
{
public:

    Node* front;
    Node* back;
    int size;
    Queue();
    void push(Customer);
    void pop();
};
```

only got 2 methods push (Customer) and pop ()

the Queue is obviously customized for Customer Class so as the Node class

Node:

Node.h:

```
#include "Customer.h"
class Node
{
public:
    Node* pre;
    Node* next;
    Customer val;
    Node(Customer value);
};
```

Module:

Module.h:

```
#include "Queue.h"
#include <Vector>
class Module {
public:
    int servers;
    Queue que;
    int fqTime;
    int numEvents;
    vector <Customer> manualInput;
    Module(int);
    Module();
};
```

Each Module has its number of servers, Queue of waiting customers, time between customers coming into the bank, average number of events for the Frequent coming customers and a list of given known customers.

The module attributes are used for the simulation process.

UTILS FILES

PrintUtil: This file is responsible for messages and printing operations in general.

printUtil.cpp:

```
void printAdvice(){
    cout << "Number of servers are not enough to handle the customer proparly\n"
        << "Please consider increamenting the number of the servers \n";
}
void printServer(int customerNumber, int serverNumber) {
    if (customerNumber == -1) { cout << "Server " << serverNumber << " : Idle \n"; }
    else { cout << "Server " << serverNumber << " : customer" << customerNumber << '\n'; }
}
void printBreakPoint() {
    cout << "--------------------------------------------------------------------------------\n";
}
void instructionsPrint() {
    cout << "Instructions\n"
        << "__M          -->To create a new case of testing\n"
        << "__S          -->To initiate Simulation for the case you created\n"
        << "__M_AUTO        -->To Modify the frequent time of arriving customers and AVG actions per customer\n"
        << "__M_MAN__ADD        -->To Add know arrival time customers to the system\n"
        << "__M_MAN__CLR        -->To Clear the known customers with expected arrival time\n"
        << "__END           --> to end process\n";
}
```

InputTemplete: this file is responsible of the input processes and taking data from user

InputTemplete.cpp:

```cpp
void manualInput(int* arrivalTime, int* events) {
    cout << "Arrival time: "; cin >> *arrivalTime;
    cout << "\nNumber of actions needed by the customer "; cin >> *events;
    cout << '\n';
}
void manualInput(int* numElements) {
    cout << "Number of customers: ";
    cin >> *numElements;
    cout << '\n';
}
int serversInput() {
    int num;
    cout << "Number of servers of the Module: ";
    cin >> num;
    cout << '\n';
    return num;
}
void automaticInput(int* fqTime, int* numEvents) {
    cout << "Frequnt time for customers to come: ";
    cin >> *fqTime;
    if (*fqTime != -1) {
        cout << "\nNumber of actions for each customer: ";
        cin >> *numEvents;
    }
    cout << "\n";
}
int inputSimulationTime() {
    int num;
    cout << "Enter the wanted simulation time in mins: "; cin >> num; cout << '\n';
    return num;
}
```

PROCESSING FILES

Time Simulation: The file is only responsible of the simulation that happens on a given module and a specific time in minutes.

TimeSimulation.cpp:

```cpp
35  void Simulation(int SimulationTime, Module* machine) {
36      bool serverIncreamentAdvice = false;
37      for (int j{ 0 }; j < SimulationTime; j++) {
38          checkingComingCustomers(machine, j);
39          serverIncreamentAdvice = check(machine, j);
40          printBreakPoint();
41      }
42      if (serverIncreamentAdvice) printAdvice();
43  }
```

-The simulation processes the module servers for each minute.

-It pushes elements to the queue of the machine if the we reached the time of expected arrival of the customer or time between arrival of customers equals 0 at a current time.

-Decrement the tasks of current handled customer **Given that a server can only handle 1 action in 1 min**

- When the current customer has no actions left to be processed the system will pop it out of the Queue and handle the next customer at the same time If there are servers available for tasks (Idle)

Command file: the file is responsible of handling the incoming commands of the main file.

Command.cpp:

```
void serversModification(Module* machine) {
    machine->servers = serversInput();
}
void simulationCommand(int SimulationTime, Module* machine) {
    Simulation(SimulationTime, machine);
}
void manualModification(vector <Customer> *lst) {
    int numElements;
    manualInput(&numElements);
    for (int i{ 0 }; i < numElements; i++) {
        int arrivalTime, events;
        manualInput(&arrivalTime, &events);
        lst->push_back(Customer(arrivalTime, events));
    }
    sort(lst->begin(), lst->end());
}
void autoMaticModification(Module* machine) {
    int fqTime, numEvents;
    automaticInput(&fqTime, &numEvents);
    machine->fqTime = fqTime;
    machine->numEvents = numEvents;
};
```

The manualModification function handles an incoming vector of class <Customer> and call-in input function to input customers into the list and after the input process. It sorts them based on the expected time of arrival as it will be used in Time Simulation.cpp file to append to the que when the expected time of arrival = the current simulated time.

Main file: It's only responsible of handling the command coming from the user and call in the appropriate command function

Main.cpp:

```
    else if (command == "__S") {
        int time = inputSimulationTime();
        machine.manualInput = lst;
        simulationCommand(time, &machine);
    }
    else if (command == "__M_AUTO")
        autoMaticModification(&machine);

    else if (command == "__M_MAN__ADD")
        manualModification(&lst);

    else if (command == "__M_MAN__CLR")
        lst.clear();

    else
        cout << "Invalid command\n";
}
int main() {
    string input;
    instructionsPrint();
    while (true)
    {
        cout << "Command: "; cin >> input; cout << '\n';
        if (endProcess(input))break;
        excuteCommand(input);
    }
    return 0;
}
```

-It checks at every loop if the input was "__END" in order to break the loop and exit.

-each command has it's function of execution.