Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 6

по курсу: «Моделирование»

Тема Моделирование работы почтового отделения

Студент Якуба Д. В.

Группа ИУ7-73Б

Оценка (баллы) _____

Преподаватель Рудаков И.В.

Москва, 2021

1. Задание

В некоторое почтовое отделение с тремя окнами (отправка посылок, получение посылок и оплата коммунальных и муниципальных платежей) приходят клиенты с интервалом времени 5 ± 3 минуты и становятся в очередь к терминалу для получения талонов на обслуживание.

Получение одного талона у клиента занимает 3 ± 2 минуты.

Далее с вероятностью 30% клиент становится в очередь на отправку посылок, 50% - в очередь на получение посылок и 20% - в очередь на оплату коммунальных и муниципальных платежей.

Окно отправки посылок обслуживает каждого клиента 13 ± 5 минут.

Окно выдачи посылок обслуживает каждого клиента 5 ± 3 минуты.

Окно оплаты коммунальных и муниципальных платежей обслуживает каждого клиента 20 ± 5 минут.

После получения каждой услуги клиент вновь становится в очередь к терминалу с вероятностью $p_{\text{возвр}} = 0.2$.

Размер очереди к терминалу не ограничен.

Если клиент видит 10 человек в очереди к окну, он уходит.

2. Теория

2.1 Концептуальная модель системы

На рисунке 2.1 предоставлена концептуальная модель моделируемой системы.

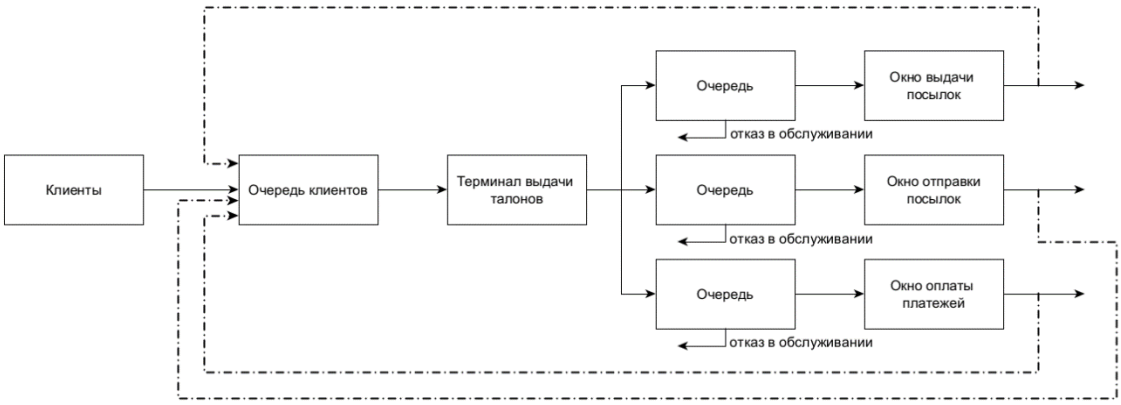

Рис. 2.1, концептуальная модель системы

2.2 Схема элементов СМО

На рисунке 2.2 предоставлена схема элементов СМО моделируемой системы.

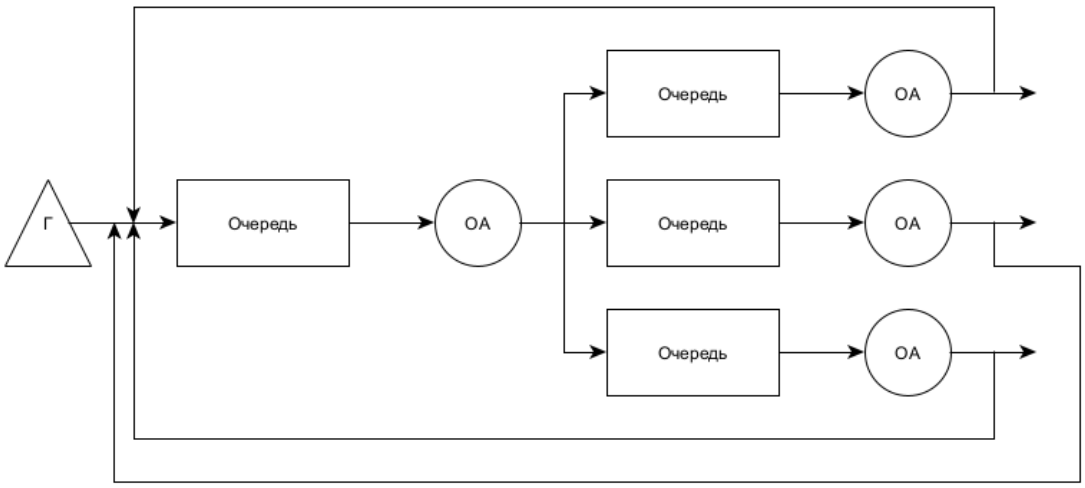

Рис. 2. 2, схема элементов СМО

2.3 Уравнение модели

Вероятность отказа в обслуживании клиента окном:

$$\mathrm{P}_{\text{отказа}} = \frac{C_{\text{отказанных}}}{C_{\text{отказанных}} + C_{\text{обслуженных}}}$$

где $C_{\text{отказанных}}$ – количество заявок, которым было отказано в обслуживании, $C_{\text{обслуженных}}$ – количество заявок, которые были обслужены.

Данная вероятность рассчитывается для каждого окна.

3. Выполнение

Моделирование проводилось с использованием событийного принципа.

На рисунках 3.1 – 3.2 предоставлены примеры работы реализованного приложения.

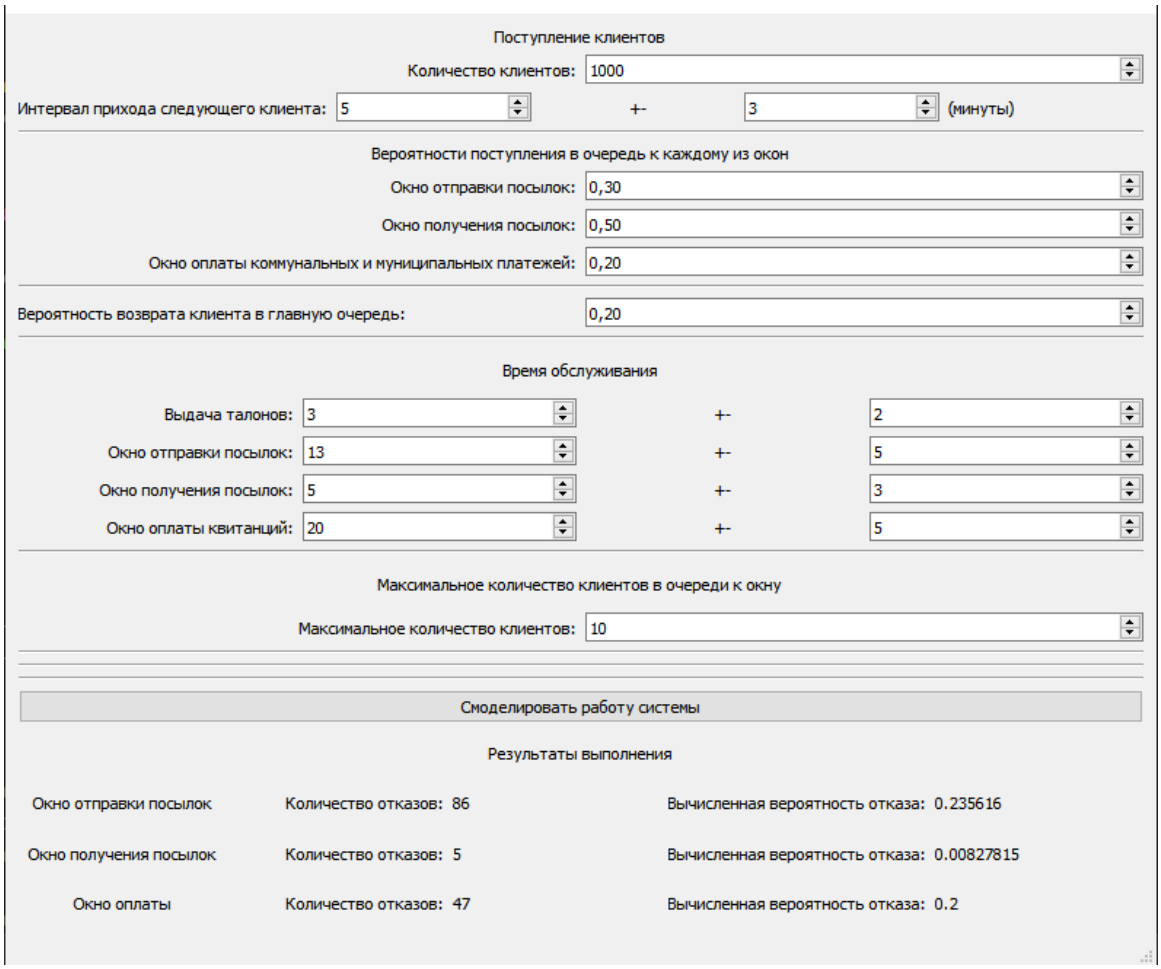

Рис. 3.1, пример работы реализованного приложения

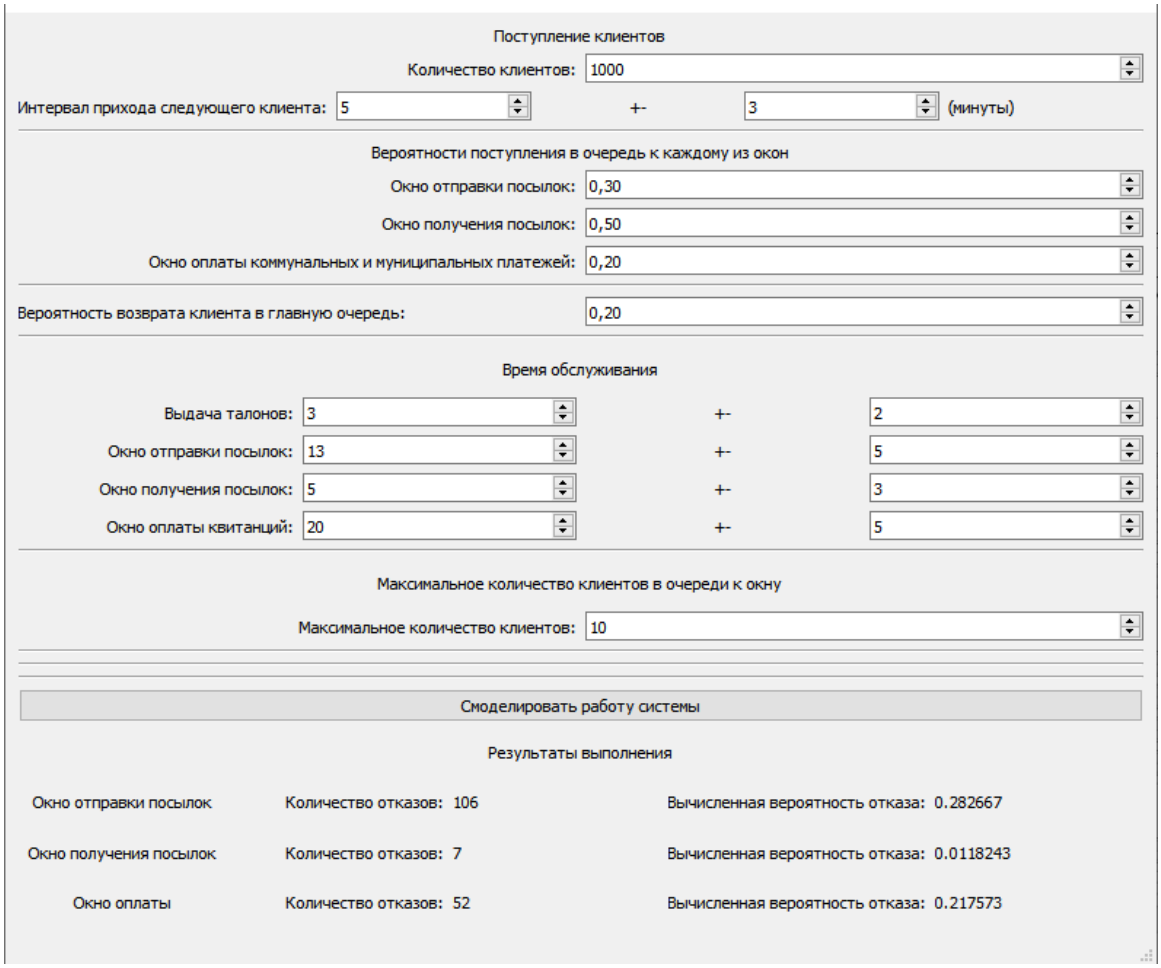

Рис. 3.2, пример работы реализованного приложения

Таким образом, в случае если заказчику требуется уменьшить вероятность отказа заявки для определённого окна, а, соответственно, и количество таких заявок, он может прибегнуть к уменьшению времени обработки заявок, например, увеличив количество работников или прибегнув к иным оптимизациям.

На рисунке 3.3 можно увидеть, что если уменьшить интервал обработки заявок третьего окна на 5 минут (до интервала (10; 20)), то вероятность отказа **может** упасть до 7.2%.

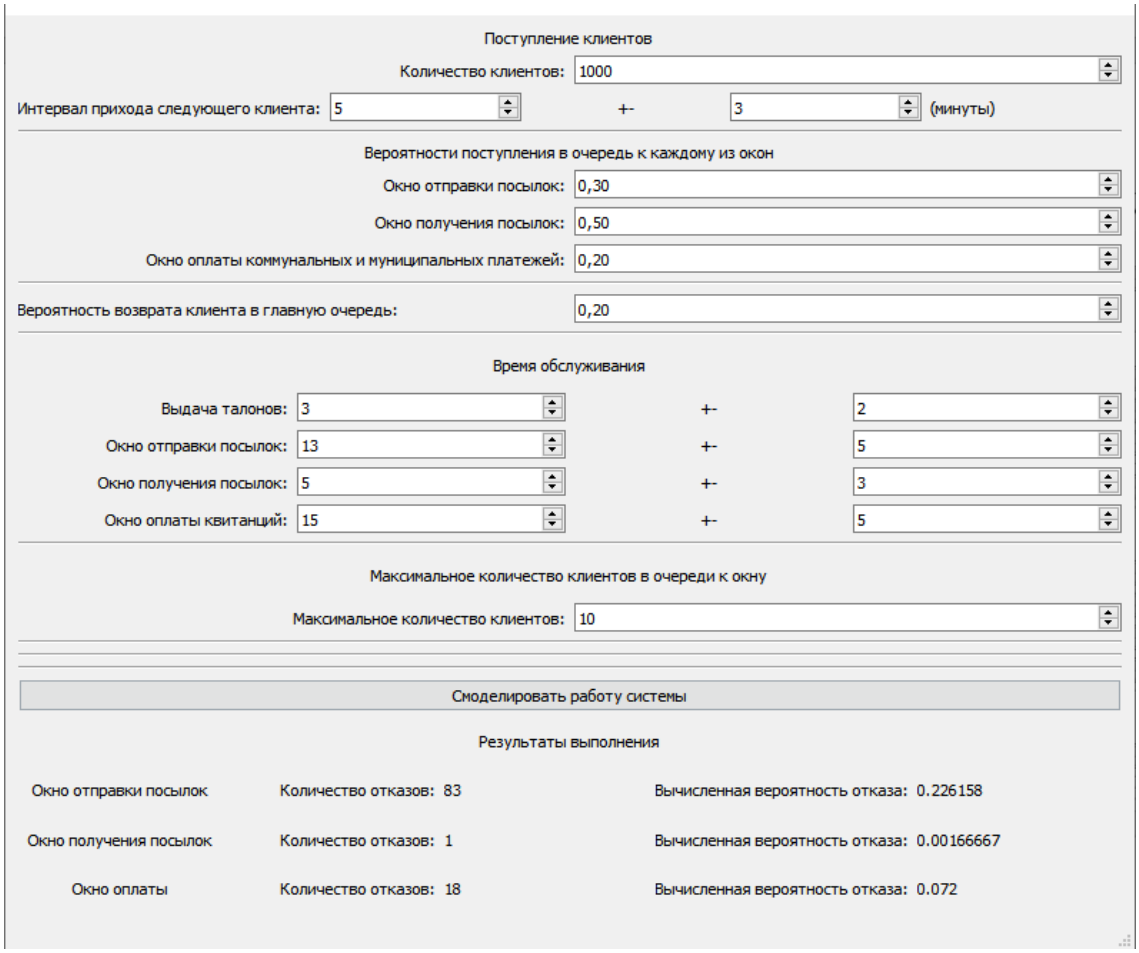

Рис. 3.3, пример работы приложения при проведении оптимизации обработки заявок в третьем окне

4. Листинг

В данном разделе предоставлены используемые для работы приложения классы (используемый ЯП – C++).

```
double currentTime;

class Processor
{
public:
    virtual ~Processor() {}

    virtual void getRequest() = 0;
};

class RequestGenerator
{
public:
    using Generator = std::function<double()>;

    RequestGenerator(Generator generator_)
    {
        numberOfGeneratedRequests = 0;
        returnReceiver = nullptr;
        generator = generator_;
        receivers = std::vector<Processor *>();
```

```
        timeOfNextEvent = 0;
    }

    virtual ~RequestGenerator() {}

    void subscribeReceiver(Processor *receiver) {
receivers.push_back(receiver); }

    double getNextTime() { return generator(); }

    void sendRequest()
    {
        numberOfGeneratedRequests++;
        if (receivers.size() == 3)
        {
            Processor *receiver = receivers[0];

            double currentSum = 0;
            double chooseRandom = getUniformReal(0, 1);

            for (size_t i = 0; i < pValuesToSendTo.size(); i++)
            {
                currentSum += pValuesToSendTo[i];
                if (chooseRandom < currentSum)
                {
                    receiver = receivers[i];
                    break;
                }
            }

            receiver->getRequest();
        }
        else
        {
            for (auto &&receiver : receivers) { receiver->getRequest(); }
        }
    }

    void returnRequestToSubscriber()
    {
        if (returnReceiver)
        {
            returnReceiver->getRequest();
        }
    }

public:
    int numberOfGeneratedRequests;
    Processor *returnReceiver;

private:
    Generator generator;
    std::vector<Processor *> receivers;

public:
    double timeOfNextEvent;
    std::vector<double> pValuesToSendTo;
};

class RequestProcessor : public RequestGenerator, public Processor
{
public:
    using Generator = RequestGenerator::Generator;
```

```
    RequestProcessor(
        Generator generator, int maxOfQueue_ = 0, double
probabilityOfReturn_ = 0.0)
    : RequestGenerator(generator), Processor()
    {
        numberOfRequestsInQueue = 0;
        numberOfProcessedRequests = 0;
        numberOfSkippedRequests = 0;
        maxOfQueue = maxOfQueue_;
        numberOfReturns = 0;
        probabilityOfReturn = probabilityOfReturn_;
    }

    ~RequestProcessor() override {}

    void process()
    {
        if (numberOfRequestsInQueue > 0)
        {
            numberOfProcessedRequests++;
            numberOfRequestsInQueue--;
            sendRequest();

            if (getUniformReal(0, 1) < probabilityOfReturn)
            {
                numberOfReturns++;
                returnRequestToSubscriber();
            }
        }

        timeOfNextEvent = numberOfRequestsInQueue ? currentTime +
getNextTime() : 0.0;
    }

    void getRequest() override
    {
        if (maxOfQueue == 0 || numberOfRequestsInQueue < maxOfQueue)
        {
            numberOfRequestsInQueue++;
            timeOfNextEvent = numberOfRequestsInQueue ? currentTime +
getNextTime() : 0.0;
        }
        else
        {
            numberOfSkippedRequests++;
        }
    }

public:
    int numberOfRequestsInQueue;
    int numberOfProcessedRequests;
    int numberOfSkippedRequests;

private:
    int maxOfQueue;
    int numberOfReturns;
    double probabilityOfReturn;
};

Results doSimulate(const SimulationParameters &parameters)
{
    RequestGenerator generatorOfClients([=]() {
```

```
        return getUniformReal(parameters.client.timeOfCome -
parameters.client.timeDelta,
            parameters.client.timeOfCome + parameters.client.timeDelta);
    });

    RequestProcessor terminal([=]() {
        return getUniformReal(
            parameters.terminal.timeOfService -
parameters.terminal.timeDelta,
            parameters.terminal.timeOfService +
parameters.terminal.timeDelta);
    });
    std::vector<double> pValuesToSendTo =
{parameters.client.probForSendWindow,
        parameters.client.probForGetWindow,
parameters.client.probForMoneytalksWindow};
    terminal.pValuesToSendTo = pValuesToSendTo;
    RequestProcessor sendWindow(
        [=]() {
            return getUniformReal(
                parameters.sendWindow.timeOfService -
parameters.sendWindow.timeDelta,
                parameters.sendWindow.timeOfService +
parameters.sendWindow.timeDelta);
        },
        parameters.sendWindow.maxQueueSize,
        parameters.client.probabilityOfReturnToMainQueue);
    RequestProcessor getWindow(
        [=]() {
            return getUniformReal(
                parameters.getWindow.timeOfService -
parameters.getWindow.timeDelta,
                parameters.getWindow.timeOfService +
parameters.getWindow.timeDelta);
        },
        parameters.getWindow.maxQueueSize,
        parameters.client.probabilityOfReturnToMainQueue);
    RequestProcessor moneytalksWindow(
        [=]() {
            return getUniformReal(parameters.moneytalksWindow.timeOfService
-
                                  parameters.moneytalksWindow.timeDelta,
                parameters.moneytalksWindow.timeOfService +
                    parameters.moneytalksWindow.timeDelta);
        },
        parameters.moneytalksWindow.maxQueueSize,
        parameters.client.probabilityOfReturnToMainQueue);

    generatorOfClients.subscribeReceiver(&terminal);
    terminal.subscribeReceiver(&sendWindow);
    terminal.subscribeReceiver(&getWindow);
    terminal.subscribeReceiver(&moneytalksWindow);
    sendWindow.returnReceiver = &terminal;
    getWindow.returnReceiver = &terminal;
    moneytalksWindow.returnReceiver = &terminal;

    std::array<RequestGenerator *, 5> devices{
        &generatorOfClients, &terminal, &sendWindow, &getWindow,
&moneytalksWindow};

    generatorOfClients.timeOfNextEvent = generatorOfClients.getNextTime();
    terminal.timeOfNextEvent =
        generatorOfClients.timeOfNextEvent + terminal.getNextTime();
```

```
    while (generatorOfClients.numberOfGeneratedRequests <
parameters.client.amount)
    {
        currentTime = generatorOfClients.timeOfNextEvent;
        for (auto &&device : devices)
        {
            if (device->timeOfNextEvent != 0 && device->timeOfNextEvent <
currentTime)
            {
                currentTime = device->timeOfNextEvent;
            }
        }

        for (auto &&device : devices)
        {
            if (currentTime == device->timeOfNextEvent)
            {
                RequestProcessor *processor = dynamic_cast<RequestProcessor
*>(device);
                if (processor)
                {
                    processor->process();
                }
                else
                {
                    generatorOfClients.sendRequest();
                    generatorOfClients.timeOfNextEvent =
                        currentTime + generatorOfClients.getNextTime();
                }
            }
        }
    }

    while (terminal.numberOfRequestsInQueue > 0 ||
           sendWindow.numberOfRequestsInQueue > 0 ||
           getWindow.numberOfRequestsInQueue > 0 ||
           moneytalksWindow.numberOfRequestsInQueue > 0)
    {
        currentTime = std::numeric_limits<double>::max();
        for (size_t i = 1; i < devices.size(); i++)
        {
            if (devices[i]->timeOfNextEvent != 0 &&
                devices[i]->timeOfNextEvent < currentTime)
            {
                currentTime = devices[i]->timeOfNextEvent;
            }
        }

        for (size_t i = 1; i < devices.size(); i++)
        {
            if (currentTime == devices[i]->timeOfNextEvent)
            {
                dynamic_cast<RequestProcessor *>(devices[i])->process();
            }
        }
    }

    auto res = [](RequestProcessor &processor) {
        int numberOfSkippedRequests = processor.numberOfSkippedRequests;
        int numberOfProcessedRequests = processor.numberOfProcessedRequests;
        double probabilityOfRequestToBeSkipped =
            static_cast<double>(numberOfSkippedRequests) /
            (numberOfSkippedRequests + numberOfProcessedRequests);
```

```
        return Results::Element{numberOfSkippedRequests,
probabilityOfRequestToBeSkipped};
    };

    return {res(terminal), res(sendWindow), res(getWindow),
res(moneytalksWindow)};
}
```