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»

на тему: «Моделирование прохода болельщиков на стадион»

Студент ИУ7-71Б _____ Мицевич М. Д.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Рудаков И. В.
(Подпись, дата) (И. О. Фамилия)

2022 г.

1 Теоретический раздел

Моделируется проход фанатов на футбольный стадион ВТБ арена. Фанаты приходят с 2 станций метро: Динамо и Петровско-Разумовская. Далее для прохода на стадион нужно пройти проверку билетов, досмотр личных вещей и проверку на самом стадионе.

Структурная схема представлена на рисунке 1.1.

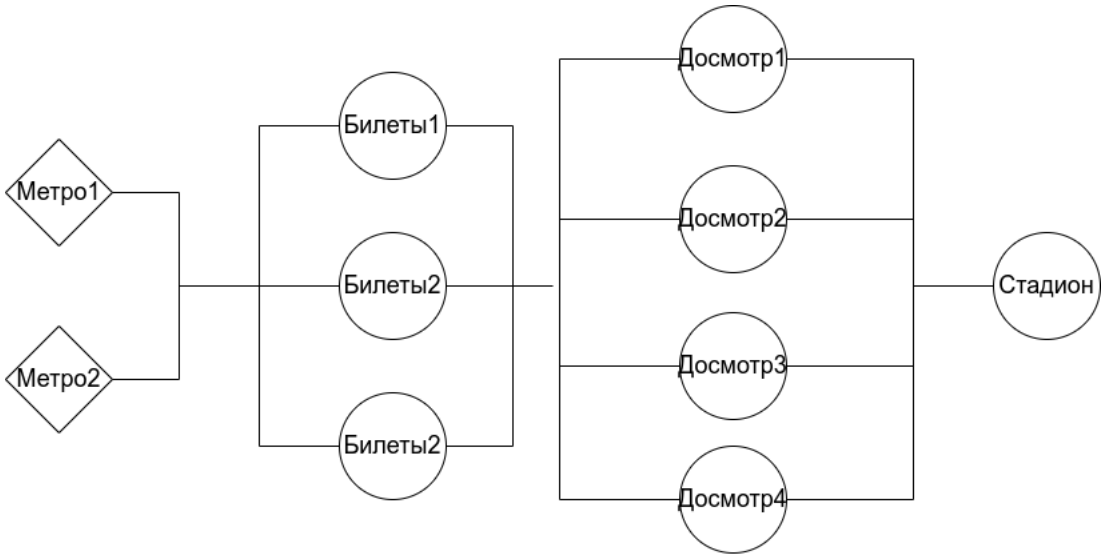


Рисунок 1.1 – Структурная схема модели прохода на стадион

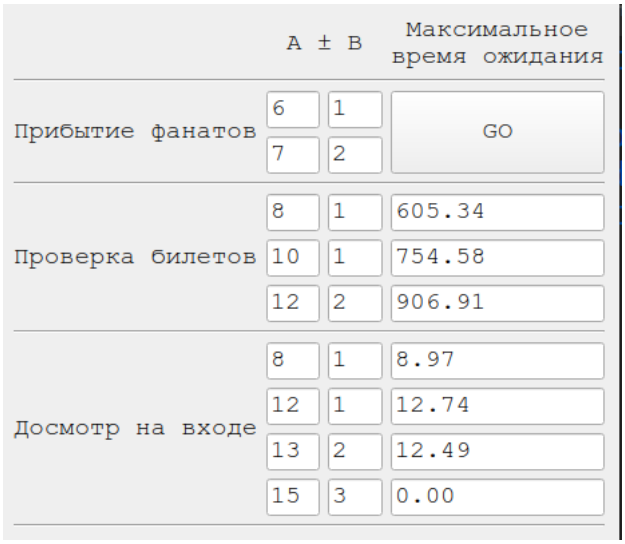
Переменные и уравнения имитационной модели

Эндогенные переменные: время проверки билетов, досмотра на входе и на стадионе.

Экзогенные переменные: число фанатов, которые прошли на стадион.

2 Практическая часть

Результат работы программы представлен на рисунке 2.1.



	A	±	B	Максимальное время ожидания
Прибытие фанатов	6	1		GO
	7	2		
Проверка билетов	8	1		605.34
	10	1		754.58
	12	2		906.91
Досмотр на входе	8	1		8.97
	12	1		12.74
	13	2		12.49
	15	3		0.00

Рисунок 2.1 – Результат работы программы

На листингах 2.1-2.2 представлен код программы.

Листинг 2.1 – Моделирование работы информационного центра

```
from collections import deque
from generator import ExponentialGenerator


class QueueSystemError(Exception):
    pass


class AllReceiversBusyError(QueueSystemError):
    pass


class Request(object):
    fmt = '{:.2f}'

    def __init__(self, creation_time):
        self._creation_time = creation_time
        self._processing_start = self._processing_end = 0
        self._dropped = False

    @property
```

```
    def creation_time(self):
        return self._creation_time

    @property
    def processing_start(self):
        return self._processing_start

    @processing_start.setter
    def processing_start(self, value):
        self._processing_start = value

    @property
    def processing_end(self):
        return self._processing_end

    @processing_end.setter
    def processing_end(self, value):
        self._processing_end = value

    def __str__(self):
        return self.fmt.format(self._creation_time)

    def __repr__(self):
        return self.fmt.format(self._creation_time)


class Device(object):
    def __init__(self, generator, name):
        self._generator = generator
        self._name = name
        self._receivers = []
        self._next_event_time = 0
        self._requests = 0
        self._dropped_requests = 0
        self._idle = False

    @property
    def name(self):
        return self._name

    def __str__(self):
```

```
        return '{} - {:.2f}'.format(self._name, self.
            _next_event_time)

    def __repr__(self):
        return '{} - {:.2f}'.format(self._name, self.
            _next_event_time)

    @property
    def next_event_time(self):
        return self._next_event_time

    @property
    def requests(self):
        return self._requests

    @property
    def dropped_requests(self):
        return self._dropped_requests

    @property
    def idle(self):
        return self._idle

    def add_receiver(self, receiver):
        if receiver not in self._receivers:
            self._receivers.append(receiver)

    def add_receivers(self, receivers):
        for receiver in receivers:
            self.add_receiver(receiver)

    def generate_time(self):
        return self._generator.next()

    def emit_request(self, request):
        potential_receivers = [receiver for receiver in self.
            _receivers if
                                receiver.can_receive_request()]
        if not potential_receivers or request._dropped:
            self._dropped_requests += 1
            return None
```

```
        potential_receiver = min(potential_receivers, key=lambda
           rcvr: rcvr.occupation)
        potential_receiver.receive_request(request)
        return potential_receiver

    def action(self):
        raise NotImplementedError


class RequestGenerator(Device):
    def __init__(self, generator, name, *, request_count=float('
       inf')):
        super().__init__(generator, name)
        self._count = request_count

    @property
    def generated_requests(self):
        return self._requests

    def __generate_request(self):
        self.emit_request(Request(self._next_event_time))
        self._next_event_time += self.generate_time()
        self._requests += 1

    def action(self):
        if not self._receivers:
            raise RuntimeError('No␣receivers␣bound␣to␣{}'.format(
               self._name))
        self.__generate_request()
        if self._requests >= self._count:
            self._idle = True


class RequestProcessor(Device):
    """
    Request processor

    Notes about implementation:
    1) Request processing is always performed over a request in
       queue at index 0 (zero). Therefore,
        real queued request count is _queued_requests - 1.
```

```
    """
    def __init__(self, generator, name, *, max_queue_size=float('
       inf'),
                 is_exit=False, can_drop=False):
        super().__init__(generator, name)
        self._queue = deque()
        self._max_queue_size = max_queue_size
        self._current_queue_size = 0
        self._queued_requests = 0
        self._max_waiting_time = 0
        self._idle_time = self._active_time = 0
        self._idle = True
        self._is_exit = is_exit
        self._can_drop = can_drop
        if can_drop:
            self.drop_prob_gen = ExponentialGenerator()
        else:
            self.drop_prob_gen = None


    @property
    def processed_requests(self):
        return self._requests

    @property
    def max_waiting_time(self):
        return self._max_waiting_time

    @property
    def queue_size(self):
        return self._current_queue_size

    @property
    def queued_requests(self):
        # See Note 1 in class docstring
        return self._queued_requests - 1

    @property
    def idle_time(self):
        return self._idle_time
```

```
    @property
    def active_time(self):
        return self._active_time

    @property
    def utilization(self):
        try:
            util = self._active_time / (self._active_time + self.
               idle_time)
        except ZeroDivisionError:
            util = 0.0
        return util

    def can_receive_request(self):
        # See Note 1 in class docstring
        if self._queued_requests - 1 < self._max_queue_size:
            return True
        return False

    @property
    def occupation(self):
        return self._queued_requests

    def receive_request(self, request):

        self._queue.append(request)
        self._queued_requests += 1
        if self._idle:
            self.__process_request()
        # See Note 1 in class docstring
        elif self._current_queue_size < self._queued_requests - 1
            <= self._max_queue_size:
            self._current_queue_size += 1
        elif self._queued_requests - 1 <= self._max_queue_size:
            pass
        else:
            self._queue.pop()
            self._queued_requests -= 1
            self._dropped_requests += 1
            raise QueueSystemError('Receiver␣is␣busy.␣Emitter␣
               should␣check␣availability␣with␣'
```

```
                                'can_receive_request()')

    def __process_request(self):
        if self._idle:
            request = self._queue[0]
            self._idle_time += request.creation_time - self.
               _next_event_time
            request.processing_start = request.creation_time
            self._next_event_time = request.creation_time + self.
               generate_time()
            self._idle = False
        else:
            request = self._queue.popleft()
            self._queued_requests -= 1
            current_time = self._next_event_time
            request.processing_end = current_time
            self._max_waiting_time = max(request.processing_start
                - request.creation_time,
                                         self._max_waiting_time)
            self._requests += 1
            self._active_time += request.processing_end - request
               .processing_start

            if self._queued_requests == 0:
                self._idle = True
            else:
                self._queue[0].processing_start = current_time
                self._next_event_time += self.generate_time()

            new_request = Request(current_time)
            if self._can_drop:
                prob = self.drop_prob_gen.next()
                if prob < 0.9:
                    new_request._dropped = True

            if not self._is_exit:
                self.emit_request(new_request)

    def action(self):
        if not self._receivers and not self._is_exit:
            raise RuntimeError('No␣receivers␣bound␣to␣{}'.format(
```

```
                self._name))
        self.__process_request()


def event_based_modelling(devices, condition):
    try:
        while not condition():
            device = min(filter(lambda x: not x.idle, devices),
               key=lambda x: x.next_event_time)
            device.action()
    except ValueError:
        print('Condition␣cannot␣be␣achieved␣with␣given␣model.')


if __name__ == '__main__':
    pass
```

Листинг 2.2 – Моделирование работы информационного центра

```
import sys

from PyQt5 import uic
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtWidgets import QApplication, QWidget

from generator import ConstGenerator, UniformGenerator, nr
from modeller import RequestGenerator, RequestProcessor,
   event_based_modelling


class MainWindow(QWidget):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self._ui = uic.loadUi("window.ui", self)

    @property
    def parameters(self):
        u = self._ui
        return {
            'pg0_m': float(u.le_pg0_m.text()),
            'pg1_m': float(u.le_pg1_m.text()),
            'pg0_d': float(u.le_pg0_d.text()),
```

```
            'pg1_d': float(u.le_pg1_d.text()),
            'ev0_m': float(u.le_ev0_m.text()),
            'ev1_m': float(u.le_ev1_m.text()),
            'ev2_m': float(u.le_ev2_m.text()),
            'ev0_d': float(u.le_ev0_d.text()),
            'ev1_d': float(u.le_ev1_d.text()),
            'ev2_d': float(u.le_ev2_d.text()),
            'cid0_m': float(u.le_cid0_m.text()),
            'cid1_m': float(u.le_cid1_m.text()),
            'cid2_m': float(u.le_cid2_m.text()),
            'cid3_m': float(u.le_cid3_m.text()),
            'cid0_d': float(u.le_cid0_d.text()),
            'cid1_d': float(u.le_cid1_d.text()),
            'cid2_d': float(u.le_cid2_d.text()),
            'cid3_d': float(u.le_cid3_d.text()),
            'c_count': 50000
        }

    @pyqtSlot()
    def on_pushButton_clicked(self):
        procfmt = '{0:13}|{1:5}|{2:5}|{3:5}'
        print(procfmt.format('name', 'reqs', 'drop', 'queue'))
        devices = self.start_modelling(**self.parameters)
        for dev in devices:
            if type(dev) is RequestGenerator:
                print(procfmt.format(dev.name, dev.requests, dev.
                   dropped_requests, ''))
            else:
                print(procfmt.format(dev.name, dev.requests,
                                     dev.dropped_requests, dev.
                                        queue_size))
        print('-' * len(procfmt.format('', '', '', '')))
        print(procfmt.format('',
                             sum(dev.requests for dev in devices)
                                ,
                             sum(dev.dropped_requests for dev in
                                devices),
                             ''))
        u = self._ui
        u.le_ev0_wt.setText('{:.2f}'.format(devices[2].
           max_waiting_time))
```

```
        u.le_ev1_wt.setText('{:.2f}'.format(devices[3].
            max_waiting_time))
        u.le_ev2_wt.setText('{:.2f}'.format(devices[4].
            max_waiting_time))
        u.le_cid0_wt.setText('{:.2f}'.format(devices[5].
            max_waiting_time))
        u.le_cid1_wt.setText('{:.2f}'.format(devices[6].
            max_waiting_time))
        u.le_cid2_wt.setText('{:.2f}'.format(devices[7].
            max_waiting_time))
        u.le_cid3_wt.setText('{:.2f}'.format(devices[8].
            max_waiting_time))

    def start_modelling(self, pg0_m, pg1_m, pg0_d, pg1_d,
                        ev0_m, ev1_m, ev2_m, ev0_d, ev1_d, ev2_d,
                        cid0_m, cid1_m, cid2_m, cid3_m, cid0_d,
                            cid1_d, cid2_d, cid3_d,
                        c_count):
        random = nr.RandomState()
        fans_generator0 = RequestGenerator(UniformGenerator(pg0_m
            , pg0_d, random),
                                           'metro␣0')
        fans_generator1 = RequestGenerator(UniformGenerator(pg1_m
            , pg1_d, random),
                                           'metro␣1')
        fans = (fans_generator0, fans_generator1)

        entrance_validator0 = RequestProcessor(UniformGenerator(
            ev0_m, ev0_d, random),
                                               'entrance␣0')
        entrance_validator1 = RequestProcessor(UniformGenerator(
            ev1_m, ev1_d, random),
                                               'entrance␣1')
        entrance_validator2 = RequestProcessor(UniformGenerator(
            ev2_m, ev2_d, random),
                                               'entrance␣2')
        entrance = (entrance_validator0, entrance_validator1,
            entrance_validator2)

        inspection0 = RequestProcessor(UniformGenerator(cid0_m,
            cid0_d, random),
```

```
                                        'inspection0')
        inspection1 = RequestProcessor(UniformGenerator(cid1_m,
            cid1_d, random),
                                        'inspection1')
        inspection2 = RequestProcessor(UniformGenerator(cid2_m,
            cid2_d, random),
                                        'inspection2')
        inspection3 = RequestProcessor(UniformGenerator(cid3_m,
            cid3_d, random),
                                        'inspection3')
        inspections = (inspection0, inspection1, inspection2,
            inspection3)

        stadium = RequestProcessor(ConstGenerator(1), 'stadium',
            is_exit=True)

        for f in fans: f.add_receivers(entrance)
        for e in entrance: e.add_receivers(inspections)
        for i in inspections: i.add_receiver(stadium)

        devices = fans + entrance + inspections + (stadium,)
        event_based_modelling(devices, lambda: stadium.
            processed_requests == c_count)
        return devices


def main():
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    return app.exec()


if __name__ == '__main__':
    sys.exit(main())
```