Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 2 з дисципліни
«Моделювання систем»

«ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО ПОБУДОВИ ІМІТАЦІЙНИХ МОДЕЛЕЙ ДИСКРЕТНО-ПОДІЙНИХ СИСТЕМ»

Виконав(ла) *ІП-14 Сергієнко Ю. В.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Дифучин А. Ю.* _____
(прізвище, ім'я, по батькові)

Київ 2024

Комп’ютерний практикум 2

Тема: об’єктно-орієнтований підхід до побудови імітаційних моделей дискретно-подібних систем.

Завдання:

1. Реалізувати алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу. **5 балів.**
2. Модифікувати алгоритм, додавши обчислення середнього завантаження пристрою. **5 балів.**
3. Створити модель за схемою, представленою на рисунку 2.1. **30 балів.**
4. Виконати верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі. Навести результати верифікації у таблиці. **10 балів.**

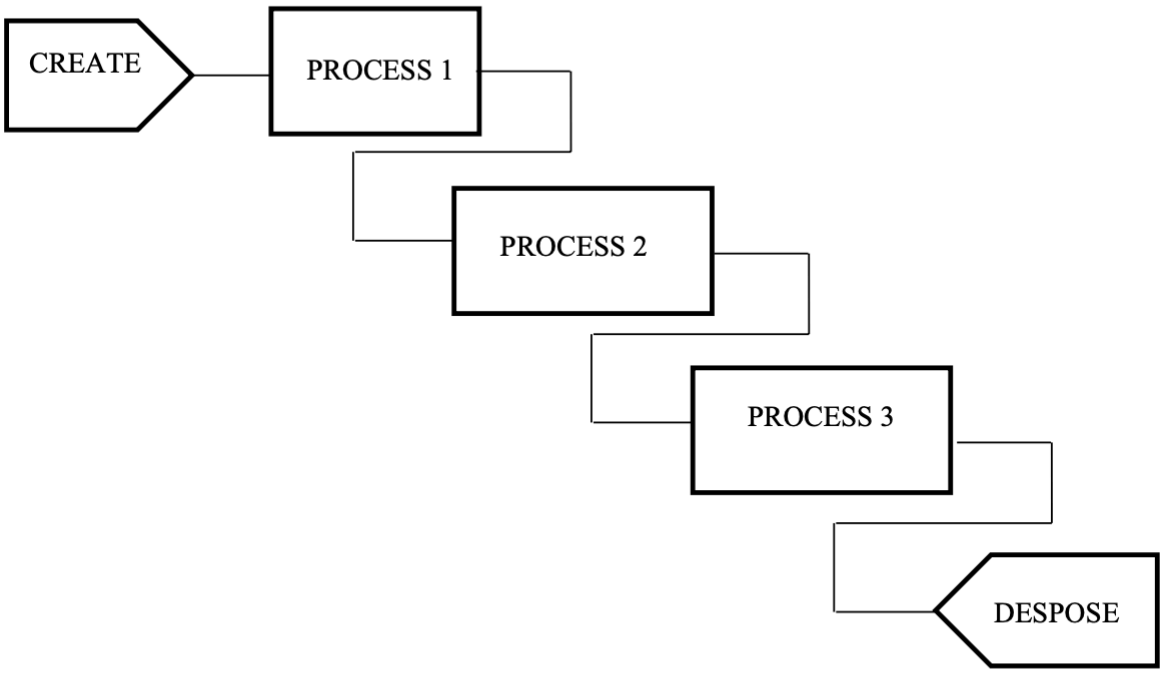

Рисунок 2.1 – Схема моделі.

5. Модифікувати клас PROCESS, щоб можна було його використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. **20 балів.**
6. Модифікувати клас PROCESS, щоб можна було організовувати вихід в два і більше наступних блоків, в тому числі з поверненням у попередні блоки. **30 балів.**

Виконання:

Було реалізовано алгоритм імітації з використанням об’єктно-орієнтованого підходу.

За основу взято клас Element — базовий клас, з нащадків якого будемо будувати модель. В моделі передбачено два типи класи-нащадки: Create та Process; їх можна пов’язати між собою за допомогою параметру nextElement.

Імітація починається з об’єкту Create - симуляції події надходження. По її виконанню буде займатись пристрій (якщо є вільний), займатись черга (якщо немає вільного пристрою) або відбуватись відмова (у випадку повної черги). Також після виконання даної події будемо генерувати час початку повтору надходження.

Після проходження даної події будемо перевіряти по часу, яка подія буде наступною. Якщо це Create — повторюємо попередні дії, інакше - відтворюємо дію на пристрої. По виходу з пристрою інкрементуємо к-сть обслугованих вимог та звільняємо пристрій. Також будемо перевіряти, чи є ще вимоги в черзі: якщо є, то займаємо нею пристрій, інакше — звільняємо його.

Цикл виконання даних подій будемо виконувати, поки поточний час менший модельний час, у класі Model. Паралельно циклу будемо обраховувати середню довжину черги, можливість відмови та середнє завантаження пристрою.

Для обрахування **середнього завантаження пристрою**, через інтервали часу будемо фіксувати, чи пристрій зайнятий в даний момент часу. Якщо так, то додаємо даний проміжок часу. В результаті отримаємо к-сть одиниць часу, скільки пристрій був у роботі. Для отримання середнього завантаження, розділимо результат на час моделювання.

```
meanLoad += getState() * delta;
```

Запустимо алгоритм за delayMeanCreate = 2, delayMeanProcess = 1, maxqueue = 5:

```
------------RESULTS------------

CREATOR quantity = 521

PROCESSOR quantity = 515
mean length of queue = 0.5413930550353855
failure probability = 0.0096153846153846 16
mean work load = 0.5121052831187309
```

Як бачимо, під час імітації була черга, що призвела до 5 відмов. Середня завантаженість = 0.512.

Створимо модель для 3 завдання. Необхідно налаштувати перехід з одного пристрою в інший, що можна зробити за допомогою nextElement параметру. Після завершення обробки вимоги пристроєм будемо перевіряти, чи існує наступний пристрій. У випадку істини спробуємо зайняти його, інакше — завершуємо прохід.

```
Element next = getNextElement();
if (next != null)
    next.inAct();
```

Виконаємо верифікацію моделі, змінюючи вхідні змінні та параметри моделі:

Таблиця - Верифікація моделі

№	T	T1	Q1	T2	Q2	T3	Q3	C	N1	MQ1	F1	MW1	N2	MQ2	F2	MW2	N3	MQ3	F3	MW3
1	2	1	5	1	5	1	5	544	541	0.437	0.006	0.513	535	0.6	0.09	0.557	532	0.535	0.005	0.561
2	2	1.3	5	1.5	5	1	5	517	496	1.148	0.04	0.677	476	1.15	0.036	0.72	475	0.309	0	0.456
3	1.8	1.5	5	2	5	1.2	5	556	504	1.675	0.09	0.781	442	1.879	0.123	0.846	439	0.442	0.002	0.511
4	2	1.5	2	1	5	1.2	5	478	420	0.439	0.12	0.587	419	0.286	0	0.445	411	0.49	0.01	0.5
5	2	1.3	2	1.3	2	1	5	473	414	0.453	0.12	0.577	391	0.242	0.055	0.482	390	0.28	0	0.424
6	2	1	1	1.5	3	1.5	3	510	425	0.156	0.165	0.442	408	0.515	0.04	0.578	375	0.564	0.08	0.6

В таблиці вказані наступні позначення:

T — середній час генерування вимоги;

T1-T3 — середній час виконання вимоги на СМО;

Q1-Q3 — черга на пристроях;

C — к-сть згенерованих вимог;

N1-N3 — к-сть оброблених вимог;

MQ1-MQ3 — середня довжина черги;

F1-F3 — ймовірність відмови;

MW1-MW3 — середнє завантаження пристрою.

Як бачимо з таблиці, результати є досить логічними, що свідчить про правильність функціонування моделі. Наприклад, за першого прогону бачимо, що ймовірність відмов на трьох СМО мізерна, і це схоже на правду, адже в середньому на одну подію створення вимоги приходить 2 події обробки. Також бачимо, що ймовірність відмов збільшується при зменшенні місць черги чи збільшенні часу обробки даної СМО. Наприклад, на 6 прогоні при Q1 = 1 майже кожна друга вимога відхиляється, бо не може стати у чергу.

Далі необхідно модифікувати клас Process, щоб можна було моделювати обслуговування кількома ідентичними пристроями. Фактично, потрібно надати можливість створювати n-канальну СМО. Для цього введемо клас Channel, що буде слугувати каналом для розгалуження. Тепер, після надходження вимоги буде виконуватись пошук вільного каналу, і в разі його знаходження — обробка на ньому. Якщо вільного каналу немає, той самий процес — відмова або черга.

При закінченні обробки необхідно:

- Звільнити канал;
- Знайти усі вільні канали;
- На основі знайденого списку (повного чи ні) змінити стан процесу;
- Вільні канали зайняти вимогами з черги.

```
@Override
```

```java
public void inAct() {
    boolean taken = false;
    for (Channel sp : channels) {
        if (sp.getState() == 0) {
            sp.setBusy(super.getTcurr() + super.getDelay());
            taken = true;
            break;
        }
    }
    setBusy();
    if (!taken) {
        if (getQueue() < getMaxqueue()) {
            setQueue(getQueue() + 1);
        } else {
            failure++;
        }
    }
}
```

```java
@Override
public void outAct() {
    // task 3
    Element next = getNextElement();
    if (next != null) {
        System.out.println("Into Processor " + next.getId());
        next.inAct();
    }
    List<Channel> freeChannels = channels.stream().filter((sp) -> {
        if (sp.getTnext() == getTcurr()) {
            super.outAct();
            sp.setFree();
        }
        return sp.getState() == 0;
    }).toList();
```

```
    super.setTnext(Double.MAX_VALUE);
    if (freeChannels.size() == channelSize) {
        super.setState(0);
    } else {
        setBusy();
    }
    for (Channel sp : freeChannels) {
        if (getQueue() > 0) {
            setQueue(getQueue() - 1);
            sp.setBusy(super.getTcurr() + super.getDelay());
            setBusy();
        }
    }
}
```

```
public void setBusy() {
    super.setState(1);
    for (Channel sp : channels) {
        if (sp.getTnext() < super.getTnext()) {
            super.setTnext(sp.getTnext());
        }
    }
}
```

При 3-канальних СМО з надходженням в середньому кожні 2 одиниці часу та обробці 1 (черга = 5), маємо наступний результат:

```
------------RESULTS------------

CREATOR quantity = 492

PROCESSOR_1 quantity = 492
mean length of queue = 0.014091033670944386
failure probability = 0.0
mean work load = 0.40761360313015554

PROCESSOR_2 quantity = 491
mean length of queue = 0.008250223556325717
failure probability = 0.0
mean work load = 0.38641994124792195

PROCESSOR_3 quantity = 491
mean length of queue = 0.005505268141983704
failure probability = 0.0
mean work load = 0.383422988583532
```

З результату бачимо, що можна підібрати більш оптимальні параметри, адже зараз використовується занадто багато ресурсів.

Спробуємо прогін 2-канальних СМО з середнім часом надходження та обробки 1 (черга = 5):

```
------------RESULTS------------

CREATOR quantity = 1071

PROCESSOR_1 quantity = 1062
mean length of queue = 0.3850936246428052
failure probability = 0.008403361344537815
mean work load = 0.6911817242802084

PROCESSOR_2 quantity = 1054
mean length of queue = 0.3687918569520822
failure probability = 0.004721435316336166
mean work load = 0.68645242729912

PROCESSOR_3 quantity = 1043
mean length of queue = 0.3675307361062955
failure probability = 0.00949667616334283
mean work load = 0.6857258230976924
```

Бачимо, що при даних параметрах результат виявився кращим: середня завантаженість СМО — майже 70%.

Для реалізації останнього завдання необхідно;

- Змінити nextElement на список nextElements;

- При створенні СМО вказувати список можливих переходів (у нашому випадку будь-куди крім Create);
- При виборі наступного кроку вибирати випадковий Element.

```
c.setNextElements(List.of(p1, p2, p3));
p1.setNextElements(List.of(p2, p3));
p2.setNextElements(List.of(p1, p3));
p3.setNextElements(Arrays.asList(p1, p2, null));
```

```
public Element getNextElement() {
    // handle for sequential (p3)
    if (nextElements == null)
        return null;
    return nextElements.get(new Random().nextInt(nextElements.size()));
}
```

Можливість виходу буде можлива лише з СМО3. Запустимо алгоритм при останніх вхідних даних (T = 1, channels = 2, Q = 5):

```
-------------RESULTS-------------

CREATOR quantity = 1004

PROCESSOR_1 quantity = 1622
mean length of queue = 1.6386881384528962
failure probability = 0.07420091324200913
mean work load = 0.9072819506683028

PROCESSOR_2 quantity = 1604
mean length of queue = 1.3087142684995208
failure probability = 0.059788980070339975
mean work load = 0.88198405339217

PROCESSOR_3 quantity = 1718
mean length of queue = 2.170266662827899
failure probability = 0.12122762148337596
mean work load = 0.9476195384884866
```

Бачимо, що найбільше роботи припадає на СМО3, адже ймовірність потрапити сюди з СМО1 та СМО2 більша, ніж навпаки.

Висновок

У ході виконання лабораторної роботи було створено та модифіковано алгоритм імітації на основі об’єктно-орієнтованого підходу. Була розроблено дві версії: для моделі з одним пристроєм обслуговування та трипристрійна модель. Були додані можливості створення n-канального пристрою та розгалуження шляху.

Було обраховано основні статистики для кожного пристрою, а саме: середня довжина черги, ймовірність відмови та середня завантаженість. Проведено верифікацію алгоритму, де ми досліджували вплив параметрів на вихідні дані.

Код програми доступний за посиланням на GitHub.