

**Національний Технічний Університет України КПІ**

Факультет Інформатики та Обчислювальної Техніки  
Кафедра інформатики та програмної інженерії

**Практична робота №2**

З дисципліни «Моделювання систем»

ОБ’ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО ПОБУДОВИ ІМІТАЦІЙНИХ

МОДЕЛЕЙ ДИСКРЕТНО-ПОДІЙНИХ СИСТЕМ

**Перевірив:**

Асистент

Дифучин А.Ю.

Оцінка:\_\_\_\_\_\_\_\_\_\_\_**Виконав:**

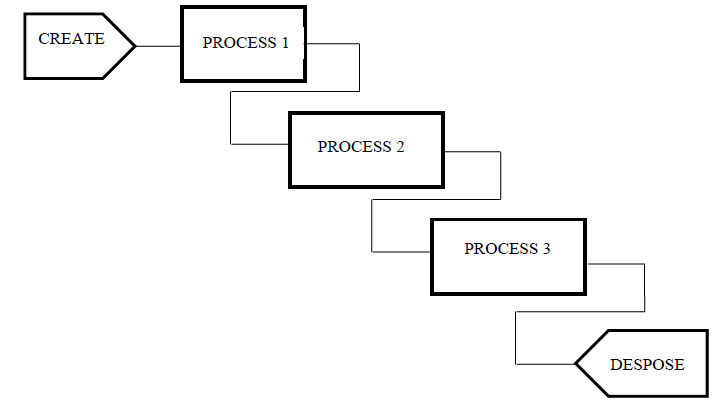
Студент групи ІТ-04

Чапча С.О.

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_

**Завдання до практичної роботи**

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 нище. **30 балів.**



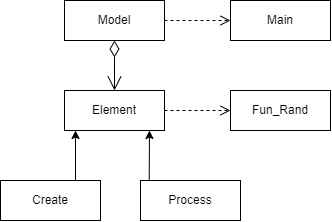
1. Виконати верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі. Навести результати верифікації у таблиці. **10 балів.**
2. Модифікувати клас PROCESS, щоб можна було його використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. **20 балів.**
3. Модифікувати клас PROCESS, щоб можна було організовувати вихід в два і більше наступних блоків, в тому числі з поверненням у попередні блоки. **30 балів**

**Виконання Лабораторної роботи**

Побудуємо алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу, та розрахуємо середнє завантаження пристрою. Наш алгоритм буде виглядати наступним чином:



Приблизна структура об’єктно-орієнтованої програми:



Модель складається з елементів, які є нащадками одного універсального типу Event. Цей клас містить основні поля та методи елементу моделі такі, як tcurr (поточний момент часу), tnext (момент часу наступної події), delayMean (середнє значення часової затримки), getDelay() (розрахунок часової затримками), inAct() (вхід в елемент), outAct() (вихід з елементу). Використання універсального класу надає можливість уніфікувати використання різних елементів в імітаційній програмі.

Важливим для з’єднання елементів в єдину модель є поле next, що вказує на наступний (в маршруті слідування вимоги) елемент моделі.

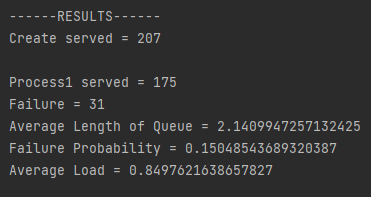
Клас Model містить метод simulate(double timeModeling), що здійснює імітацію на інтервалі часу time. Імітація здійснюється за відомим з попередньої теми принципом: визначається момент найближчої події, просувається час в момент найближчої події та здійснюється відповідна подія. Щоб зменшити обсяг обчислень, введемо також здійснення відповідної події для всіх елементів, час наступної події яких співпадає з поточним моментом часу. Оскільки ми не розглядаємо можливість присвоєння пріоритету елементам моделі, то ця дія цілком допустима.

Емуляцію будемо виконувати на проміжку часу рівним у 1000 одиниць. Затримка (delay) для CREATE та PROCESS буде рівним 5. Для розрахунку коли відбудиться наступна подія ми будемо використовувати експоненційний розподіл.

Побудова данної моделі мовою Java:

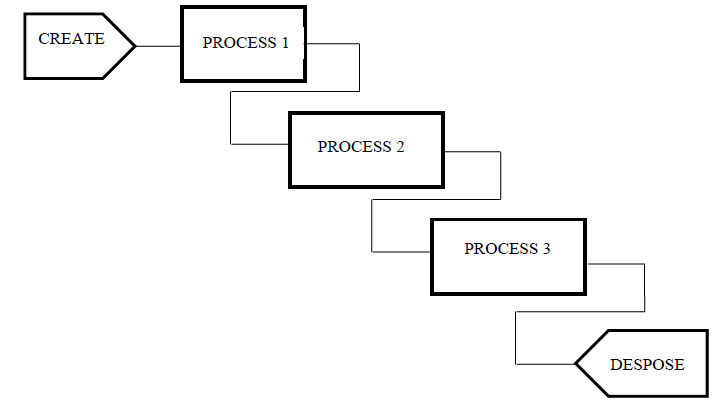
Create create = new Create(5, "Create");  
Process process1 = new Process(5, "Process1", 5.0, 1);  
  
create.setNextElement(process1);  
  
List<Event> events = List.*of*(process1);  
Model model = new Model(create, events);  
model.simulate(1000);

Виконуємо програму та отримаємо наступні результати:



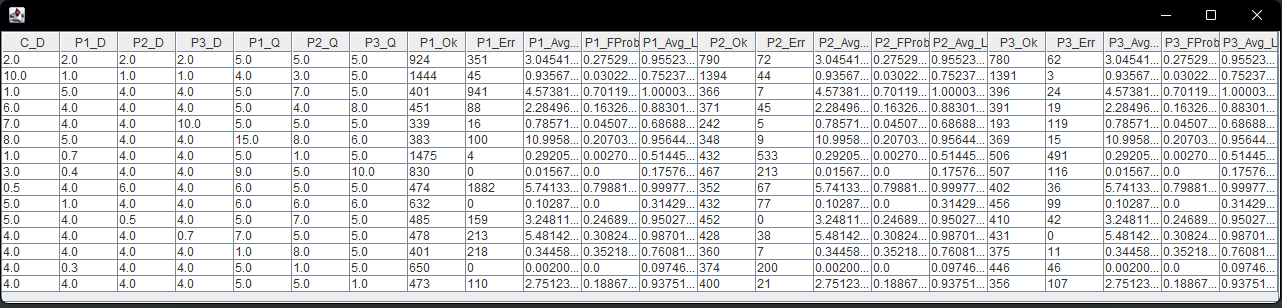
* “Create served” – кількість опрацьованих елементів процесом CREATE;
* “Process1 served” – кількість опрацьованих елементів процесом PROCESS;
* “Average Length of Queue” – середня довжина черги для PROCESS;
* “Failure Probability” – ймовірність не опрацювання завдання для PROCESS;
* “Average Load” – середнє завантаження PROCESS.

Побудуємо модель за схемою наведеною у завданні до практичної роботи разом із верифікацією моделі:



public static void task4(){  
 JFrame frame = new JFrame();  
 int n = 15;  
 Double[] delay\_create = {2.0, 10.0, 1.0, 6.0, 7.0, 8.0, 1.0, 3.0, 0.5, 5.0, 5.0, 4.0, 4.0, 4.0, 4.0};  
 Double[] delay\_process1 = {2.0, 1.0, 5.0, 4.0, 4.0, 5.0, 0.7, 0.4, 4.0, 1.0, 4.0, 4.0, 4.0, 0.3, 4.0};  
 Double[] delay\_process2 = {2.0, 1.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 6.0, 4.0, 0.5, 4.0, 4.0, 4.0, 4.0};  
 Double[] delay\_process3 = {2.0, 1.0, 4.0, 4.0, 10.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 0.7, 4.0, 4.0, 4.0};  
 Double[] maxQ\_list1 = {5.0, 4.0, 5.0, 5.0, 5.0, 15.0, 5.0, 9.0, 6.0, 6.0, 5.0, 7.0, 1.0, 5.0, 5.0};  
 Double[] maxQ\_list2 = {5.0, 3.0, 7.0, 4.0, 5.0, 8.0, 1.0, 5.0, 5.0, 6.0, 7.0, 5.0, 8.0, 1.0, 5.0};  
 Double[] maxQ\_list3 = {5.0, 5.0, 5.0, 8.0, 5.0, 6.0, 5.0, 10.0, 5.0, 6.0, 5.0, 5.0, 5.0, 5.0, 1.0};  
  
 String[] columnNames = {"C\_D", "P1\_D", "P2\_D", "P3\_D", "P1\_Q", "P2\_Q", "P3\_Q",  
 "P1\_Ok", "P1\_Err", "P1\_Avg\_Q", "P1\_FProb", "P1\_Avg\_L",  
 "P2\_Ok", "P2\_Err", "P2\_Avg\_Q", "P2\_FProb", "P2\_Avg\_L",  
 "P3\_Ok", "P3\_Err", "P3\_Avg\_Q", "P3\_FProb", "P3\_Avg\_L"};  
  
 Object[][] data = new Object[n][columnNames.length];  
  
 for (int i = 0; i < n; i++) {  
 Create create = new Create(delay\_create[i], "Create");  
 Process process1 = new Process(delay\_process1[i], "Process1", maxQ\_list1[i], 2);  
 Process process2 = new Process(delay\_process2[i], "Process2", maxQ\_list2[i], 2);  
 Process process3 = new Process(delay\_process3[i], "Process3", maxQ\_list3[i], 2);  
  
 create.setNextElement(process1);  
 process1.setNextElement(process2);  
 process2.setNextElement(process3);  
  
 List<Event> events = List.*of*(process1, process2, process3);  
 Model model = new Model(create, events);  
 model.simulate(1000);  
  
 data[i] = new Object[]{delay\_create[i], delay\_process1[i], delay\_process2[i], delay\_process3[i],  
 maxQ\_list1[i], maxQ\_list2[i], maxQ\_list3[i],  
 process1.served, process1.failure, process1.meanQueue/1000, process1.failure / ((double) process1.served + process1.failure),  
 (process1.meanLoad/process1.workerStates.size())/1000,  
 process2.served, process2.failure, process1.meanQueue/1000, process1.failure / ((double) process1.served + process1.failure),  
 (process1.meanLoad/process1.workerStates.size())/1000,  
 process3.served, process3.failure, process1.meanQueue/1000, process1.failure / ((double) process1.served + process1.failure),  
 (process1.meanLoad/process1.workerStates.size())/1000};  
 }  
  
 JTable table = new JTable(data, columnNames);  
 JScrollPane sp = new JScrollPane(table);  
 frame.add(sp);  
 frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 frame.setSize(1200,300);  
 frame.setLocationRelativeTo(null);  
 frame.setVisible(true);  
}

Після запуску програми отримуємо результати і наступну таблицю



Як можемо бачити, при зміні параметрів моделі відбуваються очікувані зміни у результатах. Так, при піднятті значень черг, значення не опрацьованих елементів буде знижуватися. Або при піднятті затримки, кількість не опрацьованих елементів на поточному пристрої буде високою, тоді як на наступних – низькою. Також це працює і в зворотньому напрямі, тому можна зробити висновок, що модель працює правильно.

Далі ми модифікували клас PROCESS, щоб його можна було використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. Для цього було обрано метод системи каналів - у одного Process може бути декілька каналів які будуть виконувати роботу.

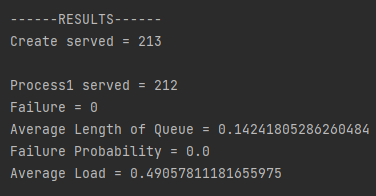
Разом із цим ми також модифікуємо клас PROCESS, щоб можна було організовувати вихід в два і більше наступних блоків, в тому числі з поверненням у попередні блоки. Для цього ми зробили властивість next\_element масивом у якому будуть зберігатися наступні елементи і в залежності від деякої ймовірності буде обиратися деяка наступна дія.

public class Process extends Event {  
 public List<Integer> states = new ArrayList<>();  
 private List<Double> tnext = new ArrayList<>();  
 public Process(double delay, String name, Double maxQueue, int countOfWorkers) {  
 super(delay, name, maxQueue);  
 tstate = Double.*MAX\_VALUE*;  
 for (int i = 0; i < countOfWorkers; i++) {  
 states.add(0);  
 tnext.add(Double.*MAX\_VALUE*);  
 }  
 }  
  
 @Override  
 public void inAct(double tcurr) {  
 int index = states.indexOf(0);  
 if (index != -1) {  
 states.set(index, 1);  
 tnext.set(index, tcurr + getDelay());  
 tstate = Collections.*min*(tnext);  
 } else {  
 if (queue < maxQueue) {  
 queue += 1;  
 } else {  
 failure++;  
 }  
 }  
 }  
  
 public void setNextEvent(List <Event> events) {  
 int processEventsCount = events.size() - 1;  
 double step = 1.0 / processEventsCount;  
 double randomNumber = Math.*random*();  
 for (int i = 0; i < processEventsCount; i++) {  
 if (randomNumber >= i \* step && randomNumber < (i + 1) \* step) {  
 Event nextEvent = events.get(i + 1);  
 if(!Objects.*equals*(nextEvent.name, this.name)) {  
 next = nextEvent;  
 }  
 break;  
 }  
 }  
 }  
  
 @Override  
 public void outAct(double tcurr, List <Event> events) {  
 super.outAct(tcurr, events);  
 int index = tnext.indexOf(tcurr);  
 states.set(index, 0);  
 tnext.set(index, Double.*MAX\_VALUE*);  
 tstate = Collections.*min*(tnext);  
 if (queue > 0) {  
 queue -= 1;  
 states.set(index, 1);  
 tnext.set(index, tcurr + getDelay());  
 tstate = Collections.*min*(tnext);  
 }  
 setNextEvent(events);  
 if (next != null) {  
 next.inAct(tcurr);  
 }  
 }  
  
 @Override  
 public void doStatistics(double delta) {  
 meanQueue += queue \* delta;  
 for(int state : states){  
 meanLoad += state \* delta;  
 }  
 }  
   
 @Override  
 public void printResult(double tcurr) {  
 super.printResult(tcurr);  
 System.*out*.println("Failure = " + this.failure);  
 System.*out*.println("Average Length of Queue = " + meanQueue / tcurr);  
 System.*out*.println("Failure Probability = " + failure / ((double) served + failure));  
 System.*out*.println("Average Load = " + (meanLoad/states.size())/tcurr);  
 }  
  
}

Дещо змінимо алгоритм імітації простої моделі обслуговування одним пристроєм, для перевірки модифікацій класу Process.

public static void task5 (){  
 Create create = new Create(5, "Create");  
 Process process1 = new Process(5, "Process1", 5.0, 2);  
  
 create.setNextElement(process1);  
  
 List<Event> events = List.*of*(process1);  
 Model model = new Model(create, events);  
 model.simulate(1000);  
}

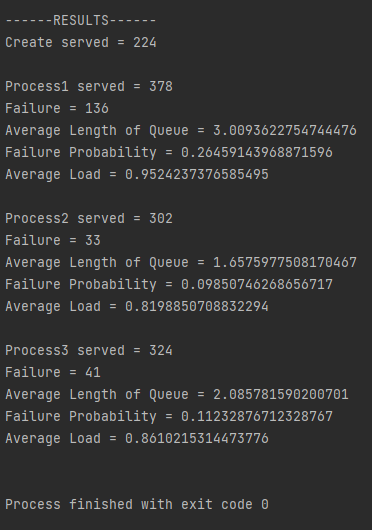
Після виконання програми отримаємо наступні результати



Можна бачити що із використанням каналів ми отримали в майже нульове значення завантаження пристрою.

Додаймо ще два пристрої. Тепер Process1 буде виходити у декілька пристроїв (Process2 та Process3) одночасно. Шанси потрапляння у кожний пристрій будуть 50 на 50:

Create create = new Create(5, "Create");  
Process process1 = new Process(5, "Process1", 5.0, 2);  
Process process2 = new Process(5, "Process2", 5.0, 2);  
Process process3 = new Process(5, "Process3", 5.0, 2);  
  
List<Event> nextEvents = List.*of*(process2, process3);  
  
create.setNextElement(process1);  
process1.setNextEvent(nextEvents);  
  
List<Event> events = List.*of*(process1, process2, process3);  
Model model = new Model(create, events);  
model.simulate(1000);



Як можна побачити, дійсно, тепер задачі розподілені 50/50 між другим та третім пристроєм.

**Висновок**

У даній лабораторній роботі ми успішно побудували алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу. Модифікували його для можливості будувати більш складні моделі. Провели верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі та зрозуміли, що модель побудована та працює вірно.

Посилання на гітхаб: <https://github.com/whitetark/modelling-systems>