**Комп’ютерний практикум 3**

**Об’єктно-орієнтований підхід до побудови імітаційних моделей дискретно-подійних систем**

**2.1 Завдання до практичної роботи**

1. Реалізувати алгоритм імітації моделі обслуговування з використанням об’єктно-орієнтованого підходу. **10 балів.**
2. Модифікувати алгоритм, додавши обчислення статистичних характеристик функціонування моделі: ймовірність відмови в обслуговуванні, максимальне спостережуване та середнє спостережуване значення черг, максимальне спостережуване та середнє спостережуване значення завантаження пристроїв. **30 балів.**
3. Модифікувати клас PROCESS, щоб можна було його використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. **20 балів.**
4. Модифікувати клас PROCESS, щоб можна було організовувати вихід в два і більше наступних блоків, в тому числі з поверненням у попередні блоки. **20 балів.**
5. Створити модель за схемою, представленою на рисунку 2.1. **10 балів.**
6. Виконати верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі. Навести результати верифікації у таблиці. **10 балів.**

DESPOSE

CREATE

PROCESS 2

PROCESS 3

PROCESS 4

DESPOSE

PROCESS 1

Рисунок 2.1 – Схема моделі.

**2.2 Теоретичні відомості з побудови об’єктно-орієнтованих алгоритмів імітації систем масового обслуговування**

Розглянемо побудову алгоритму імітації для простої моделі масового обслуговування, представленої на рисунку 2.2. Вимоги на обслуговування генеруються елементом CREATE і відправляються на обслуговування до елементу PROCESS, який здійснює обслуговування з часовою затримкою, заданою випадковим числом, та обмеження на довжину черги, заданим невід’ємним числом.

CREATE

PROCESS

DESPOSE

Рисунок 2.2 – Структура моделі

Структуру об’єктно-орієнтованої програми представимо діаграмою класів (рис. 2.3). Модель складається з елементів, які є нащадками одного універсального типу Element. Цей клас містить основні поля та методи елементу моделі такі, як tcurr (поточний момент часу), tnext (момент часу наступної події), delayMean (середеє значення часової затримки), delayDev(середнє квадратичне відхилення 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 nextElement, що вказує на наступний (в маршруті слідування вимоги) елемент моделі.

Клас Model містить метод simulate(double time), 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.

Model

Element

Create

Process

FunRand

SimModel

Рисунок 2.3 – Діаграма класів

package simsimple;

public class FunRand {

/\*\*

\* Generates a random value according to an exponential distribution

\*

\* @param timeMean mean value

\* @return a random value according to an exponential distribution

\*/

public static double Exp(double timeMean) {

double a = 0;

while (a == 0) {

a = Math.random();

}

a = -timeMean \* Math.log(a);

return a;

}

/\*\*

\* Generates a random value according to a uniform distribution

\*

\* @param timeMin

\* @param timeMax

\* @return a random value according to a uniform distribution

\*/

public static double Unif(double timeMin, double timeMax) {

double a = 0;

while (a == 0) {

a = Math.random();

}

a = timeMin + a \* (timeMax - timeMin);

return a;

}

/\*\*

\* Generates a random value according to a normal (Gauss) distribution

\*

\* @param timeMean

\* @param timeDeviation

\* @return a random value according to a normal (Gauss) distribution

\*/

public static double Norm(double timeMean, double timeDeviation) {

double a;

Random r = new Random();

a = timeMean + timeDeviation \* r.nextGaussian();

return a;

}

}

public class Element {

private String name;

private double tnext;

private double delayMean, delayDev;

private String distribution;

private int quantity;

private double tcurr;

private int state;

private Element nextElement;

private static int nextId=0;

private int id;

public Element(){

tnext = 0.0;

delayMean = 1.0;

distribution = "exp";

tcurr = tnext;

state=0;

nextElement=null;

id = nextId;

nextId++;

name = "element"+id;

}

public Element(double delay){

name = "anonymus";

tnext = 0.0;

delayMean = delay;

distribution = "";

tcurr = tnext;

state=0;

nextElement=null;

id = nextId;

nextId++;

name = "element"+id;

}

public Element(String nameOfElement, double delay){

name = nameOfElement;

tnext = 0.0;

delayMean = delay;

distribution = "exp";

tcurr = tnext;

state=0;

nextElement=null;

id = nextId;

nextId++;

name = "element"+id;

}

public double getDelay() {

double delay = getDelayMean();

if ("exp".equalsIgnoreCase(getDistribution())) {

delay = FunRand.Exp(getDelayMean());

} else {

if ("norm".equalsIgnoreCase(getDistribution())) {

delay = FunRand.Norm(getDelayMean(),

getDelayDev());

} else {

if ("unif".equalsIgnoreCase(getDistribution())) {

delay = FunRand.Unif(getDelayMean(),

getDelayDev());

} else {

if("".equalsIgnoreCase(getDistribution()))

delay = getDelayMean();

}

}

}

return delay;

}

public double getDelayDev() {

return delayDev;

}

public void setDelayDev(double delayDev) {

this.delayDev = delayDev;

}

public String getDistribution() {

return distribution;

}

public void setDistribution(String distribution) {

this.distribution = distribution;

}

public int getQuantity() {

return quantity;

}

public double getTcurr() {

return tcurr;

}

public void setTcurr(double tcurr) {

this.tcurr = tcurr;

}

public int getState() {

return state;

}

public void setState(int state) {

this.state = state;

}

public Element getNextElement() {

return nextElement;

}

public void setNextElement(Element nextElement) {

this.nextElement = nextElement;

}

public void inAct() {

}

public void outAct(){

quantity++;

}

public double getTnext() {

return tnext;

}

public void setTnext(double tnext) {

this.tnext = tnext;

}

public double getDelayMean() {

return delayMean;

}

public void setDelayMean(double delayMean) {

this.delayMean = delayMean;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public void printResult(){

System.out.println(getName()+ " quantity = "+ quantity);

}

public void printInfo(){

System.out.println(getName()+ " state= " +state+

" quantity = "+ quantity+

" tnext= "+tnext);

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public void doStatistics(double delta){

}

}

public class Create extends Element {

public Create(double delay) {

super(delay);

}

@Override

public void outAct() {

super.outAct();

super.setTnext(super.getTcurr() + super.getDelay());

super.getNextElement().inAct();

}

}

public class Process extends Element {

private int queue, maxqueue, failure;

private double meanQueue;

public Process(double delay) {

super(delay);

queue = 0;

maxqueue = Integer.MAX\_VALUE;

meanQueue = 0.0;

}

@Override

public void inAct() {

if (super.getState() == 0) {

super.setState(1);

super.setTnext(super.getTcurr() + super.getDelay());

} else {

if (getQueue() < getMaxqueue()) {

setQueue(getQueue() + 1);

} else {

failure++;

}

}

}

@Override

public void outAct() {

super.outAct();

super.setTnext(Double.MAX\_VALUE);

super.setState(0);

if (getQueue() > 0) {

setQueue(getQueue() - 1);

super.setState(1);

super.setTnext(super.getTcurr() + super.getDelay());

}

}

public int getFailure() {

return failure;

}

public int getQueue() {

return queue;

}

public void setQueue(int queue) {

this.queue = queue;

}

public int getMaxqueue() {

return maxqueue;

}

public void setMaxqueue(int maxqueue) {

this.maxqueue = maxqueue;

}

@Override

public void printInfo() {

super.printInfo();

System.out.println("failure = " + this.getFailure());

}

@Override

public void doStatistics(double delta) {

meanQueue = getMeanQueue() + queue \* delta;

}

public double getMeanQueue() {

return meanQueue;

}

}

public class Model {

private ArrayList<Element> list = new ArrayList<>();

double tnext, tcurr;

int event;

public Model(ArrayList<Element> elements) {

list = elements;

tnext = 0.0;

event = 0;

tcurr = tnext;

}

public void simulate(double time) {

while (tcurr < time) {

tnext = Double.MAX\_VALUE;

for (Element e : list) {

if (e.getTnext() < tnext) {

tnext = e.getTnext();

event = e.getId();

}

}

System.out.println("\nIt's time for event in " +

list.get(event).getName() +

", time = " + tnext);

for (Element e : list) {

e.doStatistics(tnext - tcurr);

}

tcurr = tnext;

for (Element e : list) {

e.setTcurr(tcurr);

}

list.get(event).outAct();

for (Element e : list) {

if (e.getTnext() == tcurr) {

e.outAct();

}

}

printInfo();

}

printResult();

}

public void printInfo() {

for (Element e : list) {

e.printInfo();

}

}

public void printResult() {

System.out.println("\n-------------RESULTS-------------");

for (Element e : list) {

e.printResult();

if (e instanceof Process) {

Process p = (Process) e;

System.out.println("mean length of queue = " +

p.getMeanQueue() / tcurr

+ "\nfailure probability = " +

p.getFailure() / (double) p.getQuantity());

}

}

}

}

public class SimModel {

public static void main(String[] args) {

Create c = new Create(2.0);

Process p = new Process(1.0);

System.out.println("id0 = " + c.getId() + " id1=" + p.getId());

c.setNextElement(p);

p.setMaxqueue(5);

c.setName("CREATOR");

p.setName("PROCESSOR");

c.setDistribution("exp");

p.setDistribution("exp");

ArrayList<Element> list = new ArrayList<>();

list.add(c);

list.add(p);

Model model = new Model(list);

model.simulate(1000.0);

}

}

Результати роботи програми представлені на рисунку 2.4.

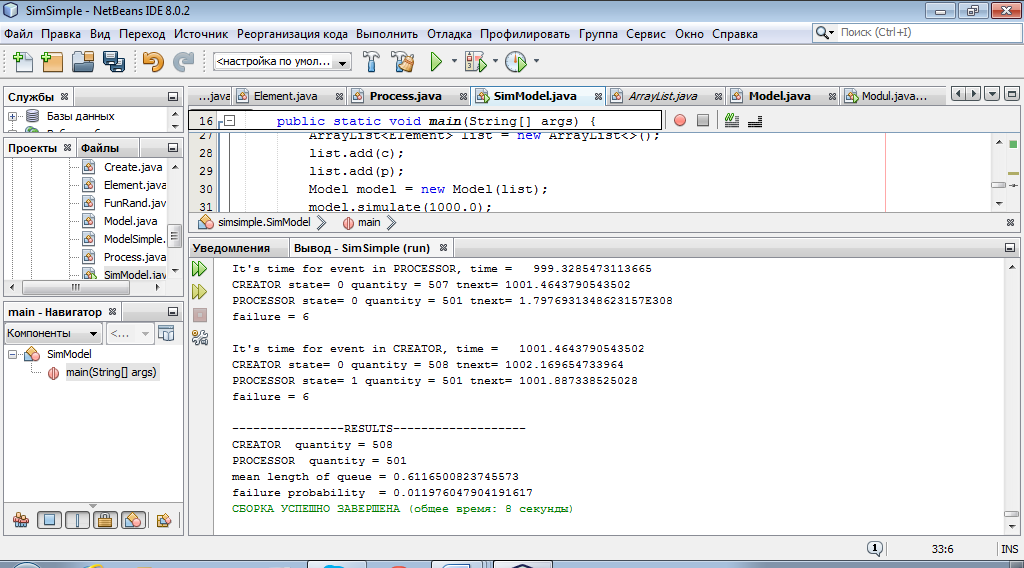


Рисунок 2.4 – Скріншот результатів моделювання