КОМП'ЮТЕРНИЙ ПРАКТИКУМ 2
ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО ПОБУДОВИ ІМІТАЦІЙНИХ МОДЕЛЕЙ ДИСКРЕТНО-ПОДІЙНИХ СИСТЕМ

2.1 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 2.1. **30 балів.**
4. Виконати верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі. Навести результати верифікації у таблиці. **10 балів.**

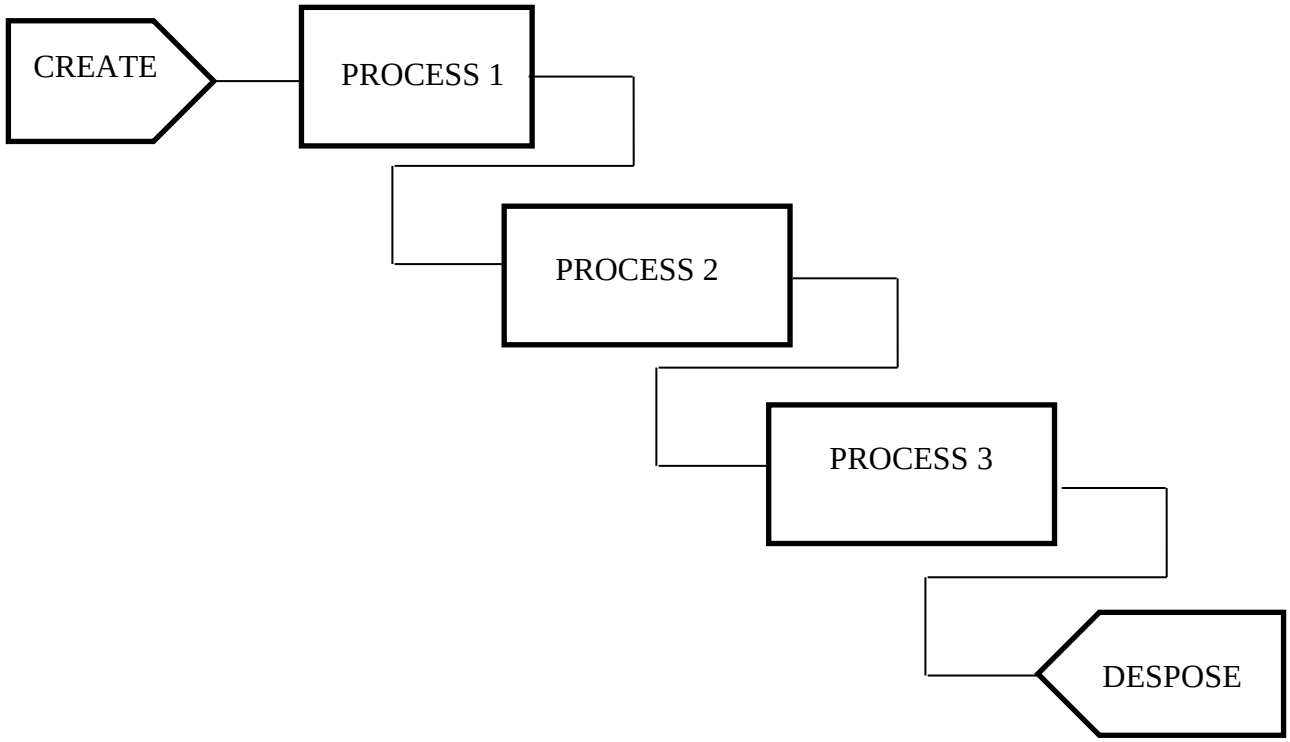

Рисунок 2.1 – Схема моделі.

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.**

2.2 Теоретичні відомості з алгоритмів імітації дискретно-подійних систем

Алгоритм, який відтворює функціонування системи, за допомогою комп'ютерної програми називається *алгоритмом імітації*.

Побудова алгоритму імітації складається з побудови:

1) алгоритму просування модельного часу;
2) алгоритму просування стану моделі в залежності від часу;
3) алгоритму збирання інформації про поведінку моделі у процесі імітації.

Просування модельного часу та просування стану моделі в залежності від часу становлять єдину задачу побудови алгоритму імітації. Якщо дослідник для реалізації алгоритму імітації використовує імітаційну мову моделювання, то моделюючий алгоритм закладений в самий засіб реалізації, і досліднику тільки корисно знати як він здійснюється, щоб не допустити помилки у застосуванні тієї або іншої імітаційної мови. Якщо ж дослідник використовує універсальну мову програмування, то задачу побудови моделюючого алгоритму йому потрібно вирішувати самостійно.

Узагальнена схема алгоритму імітації представлена на рисунку 2.2

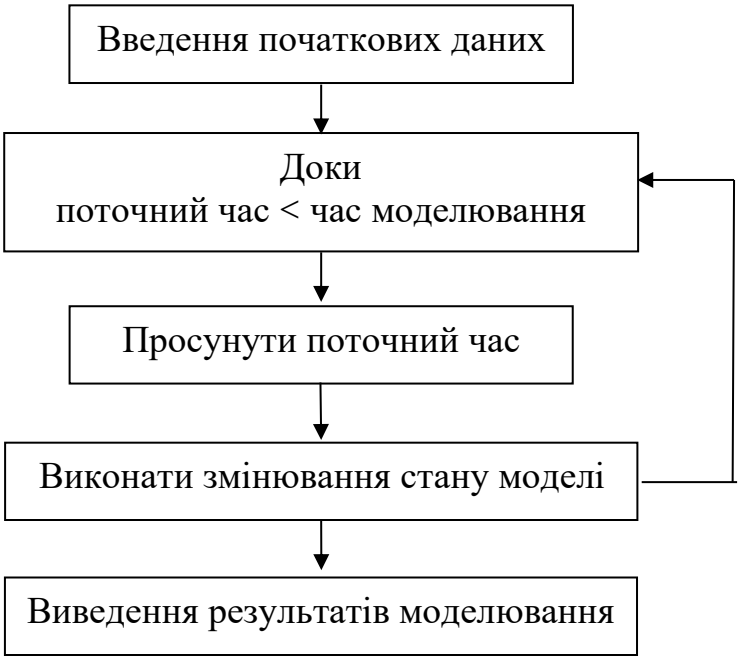

Рисунок 2.2 – Узагальнена схема алгоритму імітації.

Розглянемо відомі способи побудови алгоритмів імітації.

Способи побудови алгоритму просування модельного часу

Існують три способи просування модельного часу:

- за принципом Δt,
- за принципом найближчої події,
- за принципом послідовного проведення об’єктів уздовж моделі.

Принцип Δt

Весь інтервал часу, протягом якого моделюється система, поділяється на рівні інтервали довжиною $\Delta t = const$. При кожному просуванні модельного часу на величину Δt послідовно визначаються всі зміни, що відбуваються в моделі. Величина одного інтервалу Δt повинна бути настільки малою, щоб в одному

інтервалі Δt відбувалось не більш однієї події. В противному випадку логіка алгоритму імітації порушується і функціонування моделі суттєво відрізняється від функціонування реальної системи.

Принцип Δt є найбільш універсальним, але частіше він використовується для моделювання неперервних динамічних систем, оскільки вимагає у порівнянні з іншими способами більших затрат комп'ютерного часу при тій же точності моделювання.

Принцип найближчої події

Дискретні системи, імітаційне моделювання яких розглядається, мають певну особливість: змінювання стану в таких системах відбувається тільки в деякі моменти часу, а в усі інші моменти часу система не змінюється. Змінювання стану моделі спричиняється виникненням певної події у системі. Наприклад, подія «надходження деталі до технологічного процесу» спричиняє збільшення кількості деталей у системі, подія «виникнення поломки обладнання» спричиняє відправку поламаного обладнання до ремонту і т.д. Процес функціонування системи розглядається як послідовність подій, що відбуваються у моделі.

За принципом найближчої події модельний час просувається від моменту виникнення однієї події до моменту виникнення іншої, і після кожного просування часу реалізуються зміни стану моделі, відповідні до події, що виникла. Використання принципу найближчої події вимагає від дослідника побудови спеціальної процедури визначення моменту найближчої події, але при цьому він отримує виграш у затратах комп'ютерного часу, оскільки пропускається моделювання системи у моменти часу, коли події не відбуваються.

Розглянемо, наприклад, найпростішу модель обслуговування одним пристроєм потоку вимог (рисунок 2.3). В системі спостерігаються дві події: V - надходження вимоги, W– звільнення пристрою обслуговування. Представимо моменти часу, коли спостерігалась подія V, та моменти часу, коли спостерігалась подія W. Упорядкуємо моменти виконання подій V і W на вісі модельного часу t_M. На першому кроці алгоритму імітації встановлюється момент модельного часу t_{M1} та реалізується подія V_1; на другому – момент модельного часу t_{M2} та реалізується подія W_1, на третьому – t_{M3} та реалізуються події V_2 і W_2 у відповідності з пріоритетом процесів, і т.д.

Для дискретних систем просування модельного часу за принципом найближчої події є найефективнішим способом.

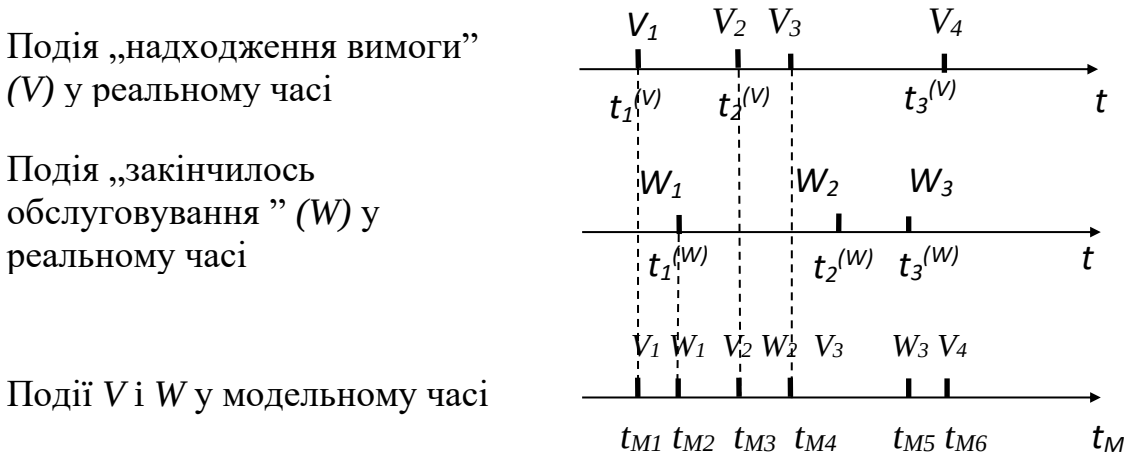

Рисунок 2.3 - Упорядкування подій у модельному часі

Принцип послідовного проведення об’єктів уздовж моделі

За принципом послідовного проведення об’єктів уздовж моделі алгоритм просування часу не будується. Кожний об’єкт проводиться по моделі з моменту його надходження у модель до моменту виходу з моделі. Історія кожного проведення запам’ятовується, так що наступний об’єкт проводиться уздовж моделі з урахуванням історії попередніх проведень. Такий алгоритм імітації дозволяє моделювати систему тільки в моменти виникнення подій, не займаючись побудовою алгоритму просування модельного часу. Проте використовується він дуже рідко, оскільки часто призводить до складних заплутаних алгоритмів.

Способи побудови алгоритму просування стану моделі в залежності від часу

Процес функціонування імітаційної моделі може бути описаний з точки зору:

1) змінювання стану системи, що відбуваються в момент появи подій;
2) дій, які виконуються елементами системи;
3) процесу, який відбувається у системі.

У відповідності з означеними способами опису функціонування моделі існують три способи просування стану моделі в часі:

- орієнтований на події;
- орієнтований на дії;
- процесно-орієнтований.

Спосіб, орієнтований на події

При підході, орієнтованому на події, дослідник визначає і описує події, які виникають у моделі. Імітація здійснюється виконанням упорядкованої у часі послідовності логічно взаємозв’язаних подій.

Проілюструємо на прикладі моделі обслуговування одним пристроєм потоку вимог (див. рисунок 4.1). Стан системи описується станом пристрою обслуговування та станом черги. Змінювання стану системи відбувається у моменти, коли у систему надходить вимога і коли закінчився процес обслуговування вимоги у пристрої. Назвемо ці події подія «надходження» та подія «закінчилось обслуговування у пристрої».

Подія «надходження» складається з таких дій:

- якщо пристрій обслуговування в стані вільний установити пристрій у стан «зайнятий», запам’ятати момент виходу вимоги з пристрою в момент часу - поточний момент часу плюс тривалість обслуговування у пристрої;
- інакше
  - якщо є вільне місце в черзі, то зайняти місце в черзі;
  - інакше збільшити кількість не обслугованих вимог на одиницю;
- генерувати момент надходження наступної вимоги у СМО.

Подія «закінчилось обслуговування в пристрої» складається з таких дій:

- збільшити кількість обслугованих вимог на одиницю;
- якщо черга вимог не пуста, перемістити одну вимогу із черги у канал обслуговування, запам’ятати момент виходу вимоги з пристрою у момент часу - поточний момент часу плюс тривалість обслуговування у пристрої;
- інакше установити канал у стан «вільний» вільний стан, запам’ятати момент виходу вимоги з пристрою у момент часу, що більший за час моделювання (тобто у найближчий час вихід вимоги із пристрою не очікується).

Після виконання події «надходження» запам’ятовуються моменти виникнення наступних подій «надходження» та «закінчилось обслуговування у пристрої». З цих двох моментів часу вибирається найменший і запам’ятовується, яка саме подія відповідає цьому моменту. Модельний час просувається у визначений момент найближчої події і виконується відповідна йому подія.

При виконанні події «закінчилось обслуговування у пристрої» запам’ятовується момент наступної події «закінчилось обслуговування у пристрої» або запам’ятовується, що найближчим часом не очікується звільнення пристрою.

Таким чином, події виконуються в упорядкованій у часі послідовності, а модельний час просувається від одного моменту виникнення події до найближчого наступного.

Програма, яка здійснює імітацію моделі, що розглядається, складається з таких дій:

1) ввести початкові значення змінних - поточний часу, час моделювання, стан пристрою, момент надходження вимоги у систему, момент звільнення пристрою обслуговування;

2) доки поточний час менший за час моделювання

3) знайти найменший із моментів часу «момент надходження вимоги у систему» та «момент звільнення пристрою обслуговування» і запам’ятати, якій події він відповідає;

4) просунути поточний час у момент найближчої події;

5) виконати подію, яка відповідає моменту найближчої події.

6) вивести результати моделювання – кількість обслугованих вимог та кількість не обслугованих вимог протягом часу моделювання.

Наведемо лістинг програми мовою Java :

```
package simsimple;

public class SimSimple {

    public static void main(String[] args) {
        Model model = new Model(2,1,5);
        model.simulate(1000);
    }
}

public class Model {
    private double tnext;
    private double tcurr;
    private double t0, t1;
    private double delayCreate, delayProcess;
    private int numCreate, numProcess, failure;
    private int state, maxqueue, queue;
    private int nextEvent;

    public Model(double delay0, double delay1){
        delayCreate = delay0;
        delayProcess = delay1;
        tnext=0.0;
        tcurr = tnext;
        t0=tcurr; t1=Double.MAX_VALUE;
        maxqueue=0;
    }
    public Model(double delay0, double delay1, int maxQ){
        delayCreate = delay0;
        delayProcess = delay1;
        tnext=0.0;
        tcurr = tnext;
        t0=tcurr; t1=Double.MAX_VALUE;
        maxqueue=maxQ;
    }

    public void simulate(double timeModeling){
        while(tcurr<timeModeling){

            tnext = t0;
            nextEvent = 0;

            if(t1<tnext){
                tnext = t1;
                nextEvent = 1;
            }
            tcurr = tnext;
            switch(nextEvent){
                case 0: event0();
```

```
                    break;
                case 1: event1();

            }
            printInfo();
        }
        printStatistic();
    }

   public void printStatistic(){
       System.out.println(" numCreate= " + numCreate+" numProcess
= "+numProcess+" failure = "+failure);
   }
    public void printInfo(){
        System.out.println(" t= " + tcurr+" state = "+state+"
queue = "+queue);
    }
    public void event0(){
        t0 = tcurr+getDelayOfCreate();
        numCreate++;
        if(state==0){
            state = 1;
            t1 = tcurr+getDelayOfProcess();
        } else {
            if(queue<maxqueue)
                queue++;
            else
                failure++;
        }


    }
    public void event1(){
        t1 = Double.MAX_VALUE;
        state=0;
        if(queue>0){
            queue--;
            state=1;
            t1 = tcurr+getDelayOfProcess();
        }
        numProcess++;
    }


    private double getDelayOfCreate() {
        return FunRand.Exp(delayCreate);
    }
    private double getDelayOfProcess() {
        return FunRand.Exp(delayProcess);
    }
}


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;
    }

     }
```

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

Розглянемо побудову алгоритму імітації для простої моделі масового обслуговування, представленої на рисунку 2.4. 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.

Рисунок 2.4 – Структура моделі

Структуру об’єктно-орієнтованої програми представимо діаграмою класів (рис. 2.5). 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.

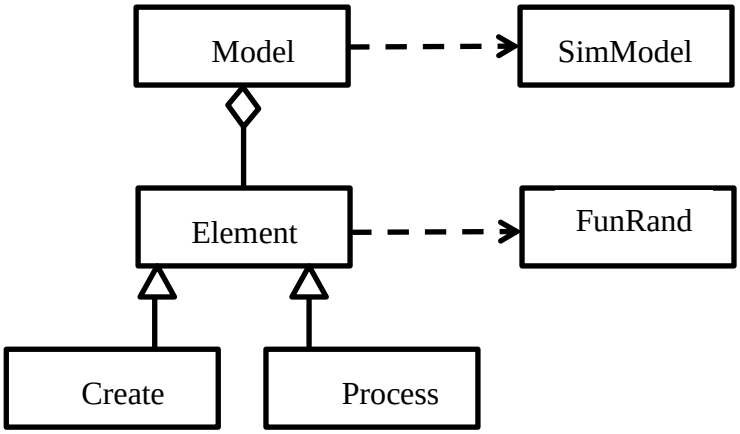

Рисунок 2.5 – Діаграма класів

```
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 = Double.MAX_VALUE;
        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);
       super.setTnext(0.0); // імітація розпочнеться з події Create
    }

    @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 2.6.

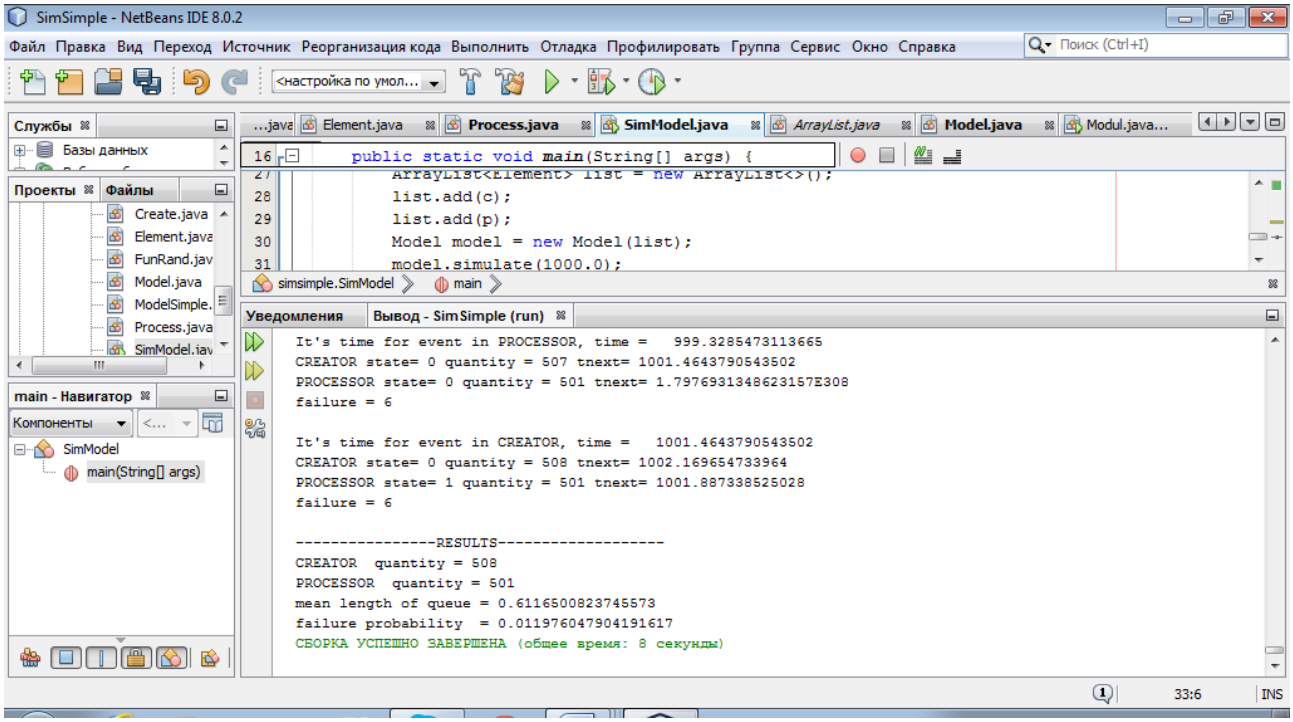

Рисунок 2.6 – Скріншот результатів моделювання