НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Звіт по лабораторній роботі №2
«Моделювання систем»
«Об’єктно-орієнтований підхід до побудови імітаційних моделей дискретно-подійних систем»

Студент: Галько М.В.
Група: ІП-01

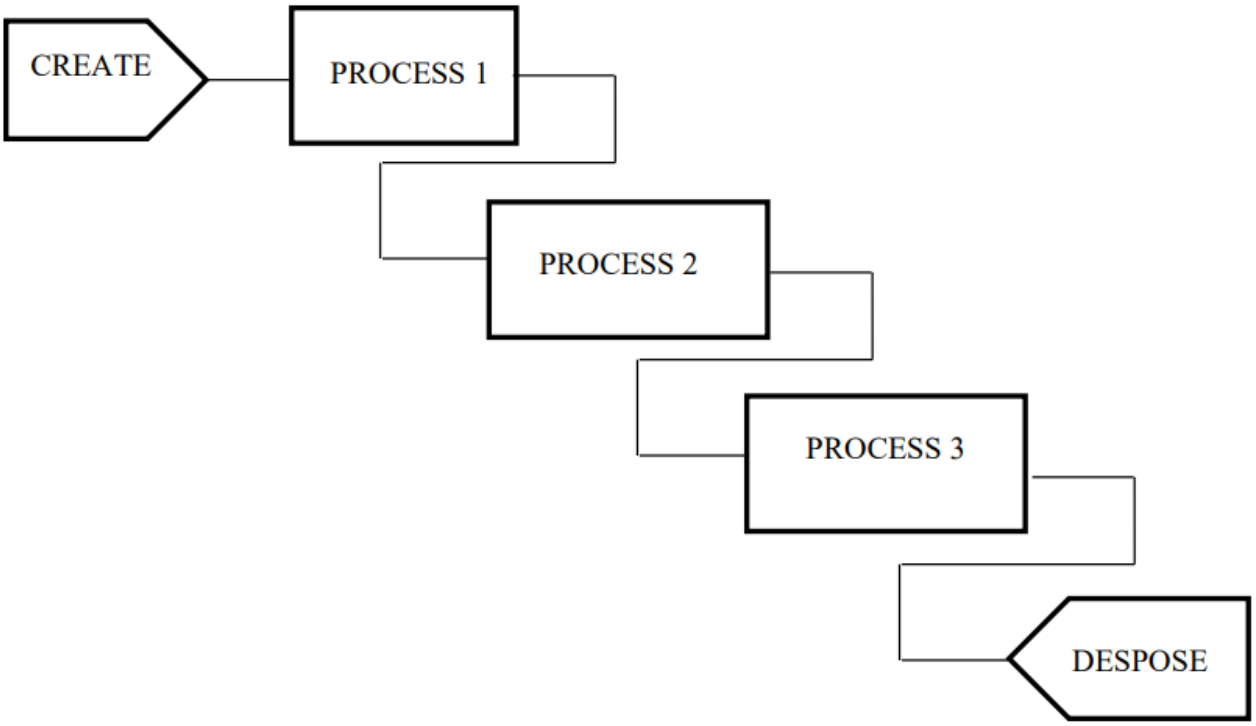
Київ, 2023

ЗМІСТ

Завдання до практичної роботи **2**
Виконання роботи **3**
Архітектура програми ООП **3**
Опис 3
Відповідальність блоків 4
Принцип роботи системи 4
Збір статистики **6**
Визначення необхідних параметрів 6
Побудова системи **7**
Схема моделі 7
Побудова моделі 7
Приклад виведення інформації про елементи 8
Виведення результату симуляції 8
Верифікація моделі **9**
Аналіз системи 9
Модифікація системи 9
Результати 10
Демонстрація роботи підпроцесів та переходів із ймовірністями **12**
Схема моделі 12
Побудова моделі 12
Результати 13
Модифікація системи 13
Результати 14
Висновок **15**
Код **16**
Model.cs 16
Element.cs 17
Create.cs 18
Process.cs 18
SubProcess.cs 19
NextElement.cs 20
NextElement.cs 20
IPrinter.cs 20

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.**



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 роботи

Архітектура програми ООП

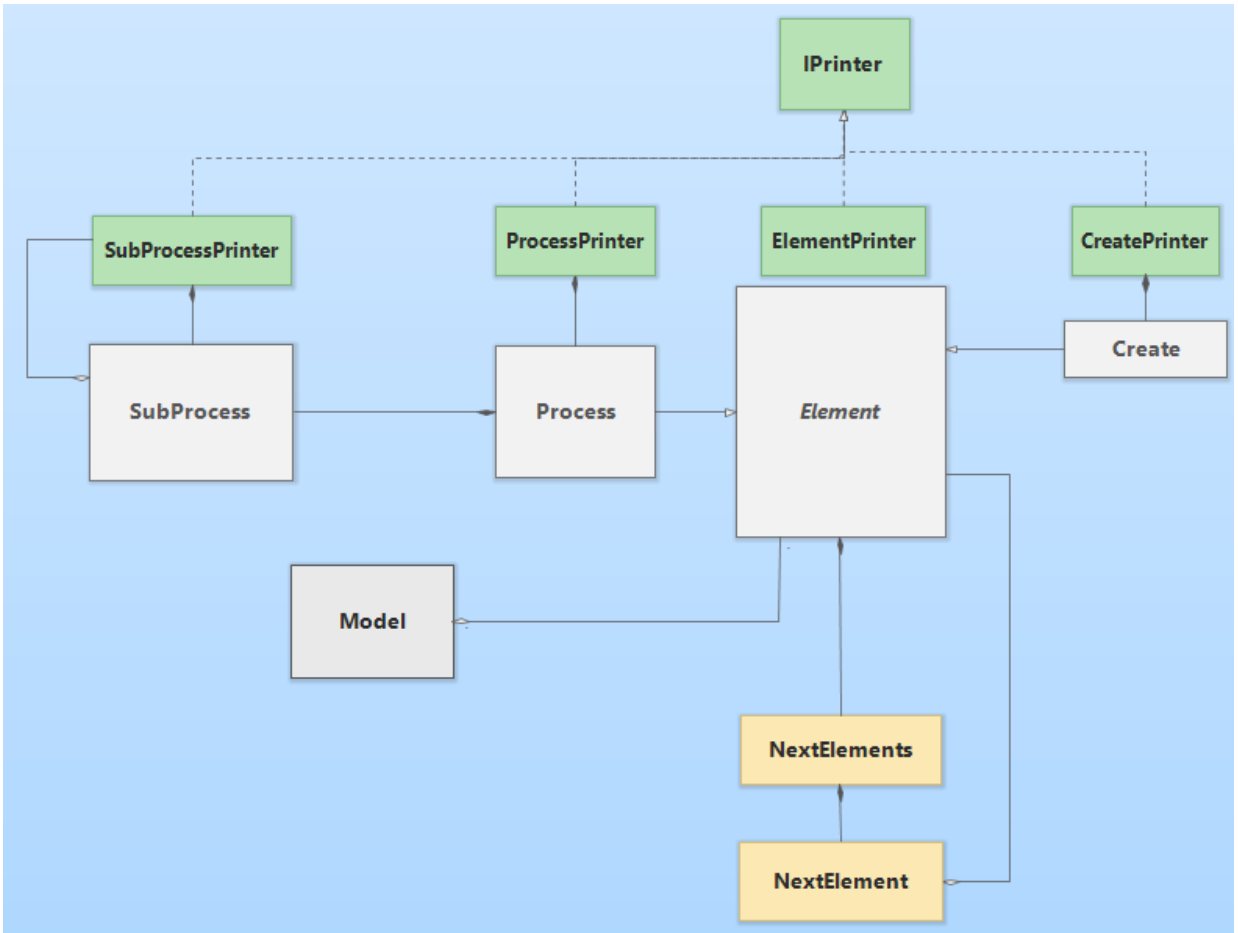
Опис

Для побудови самої моделі реалізуємо клас Model, її і будемо запускати. В модель будемо передавати елементи, які будуть обробляти об’єкти. З елементів маємо: Create та Process. Для виведення роботи елементів створимо інтерфейс IPrinter. Його реалізації мають методи для виведення статистики та інформації.

Оскільки в 5 пункті маємо пункт реалізації кількох ідентичних пристроїв, то реалізуємо його через застосування класу SubProcess у Process. Отже кожен Process має у собі від одного SubProcess-у.

6 завдання реалізуємо через застосування класу NextElements. Він утримує список посилань на наступні елементи із ймовірністю. Отже кожен елемент буде мати посилання на наступні.

Отже, маємо наступну структуру:



Відповідальність блоків

- Зелені: вивід в залежності від композиційного елементу
- Сірий: Модель, що отримує Елементи і запускає симуляцію
- Білі: абстрактний клас Елемент та його нащадки. SubProcess є композиційним атрибутом Процесу.
- Жовті: посилання на наступний елемент. NextElements утримує в собі список NextElement-ів. NextElement, в свою чергу, утримує Елемент та ймовірність попадання об’єкту у нього.

Принцип роботи системи

1. **Model** утримує атрибут List<Element> з усіма елементами системи. Має метод Simulate:

   WHILE tcurr< time

   DefineNextEvent // найближча подія

   FOREACH (element in elements) DoStatistics(Δt)

   tcurr = tnext; // перевод часу на найближчу подію

   UpgradeCurrTForAllElements(); // оновлення часу для всіх елементів

   OutActForAllCurrentElements(); // вихід з пропрацьованих елементів та перехід у наступний елемент
2. **Element** має атрибути: NextT (час закінчення процесу пристрою), CurrT (поточний час), IsWorking (відображає чи працює пристрій у поточний проміжок часу), _nextElement (посилання на наступний/-ні елемент/-ти) та інші допоміжні.

   Методи: InAct() та OutAct() віртуальні і симулюють вхід та вихід з пристрою. В залежності від нащадка має свою версію поведінки.
3. **Create** має NextT 0, отже починає свою роботу з самого початку і передає далі роботу до наступного Елементу.
4. **Process** додатково має такі атрибути як: черга, підпроцеси. При неможливості встати у чергу відбувається відмова. Реалізація методів:

   override InAct():

```
    Quantity++;
    IsWorking = true;
    IF WorkingSubProcessesCount < SubProcessesCount:
      FreeSubProcess.InAct(CurrT + GetDelay());
    ELSE:
      IF Queue < maxQueue: Queue++;
      ELSE Failure++;
    NextT = SubProcesses.Min(s => s.NextT)
```

override void OutAct():

```
  FOREACH subProcess in BusySubProcesses:
      subProcess.OutAct();
      QuantityProcessed++;
      IsWorking = false;
      nextElement?.InAct();
      IF Queue > 0:
            Queue--;
            subProcess.InAct(CurrT + GetDelay());
      NextT = SubProcesses.Min(s => s.NextT)
```

5. **SubProcess** має атрибути NextT (час закінчення дії пристрою), IsWorking (відображає чи працює пристрій у поточний проміжок часу) та допоміжні. Методи InAct та OutAct прості за призначенням, вони працюють до визначеного часу:

```
InAct(double nextT):
      IsWorking = true;
      NextT = nextT;
public void OutAct()
      IsWorking = false;
      NextT = double.MaxValue;
```

Збір статистики

Визначення необхідних параметрів

Під час симуляції необхідно збирати та обраховувати дані, щоб сформувати кінцеві висновки про якість роботи системи. Отже, маємо підраховувати наступні дані:

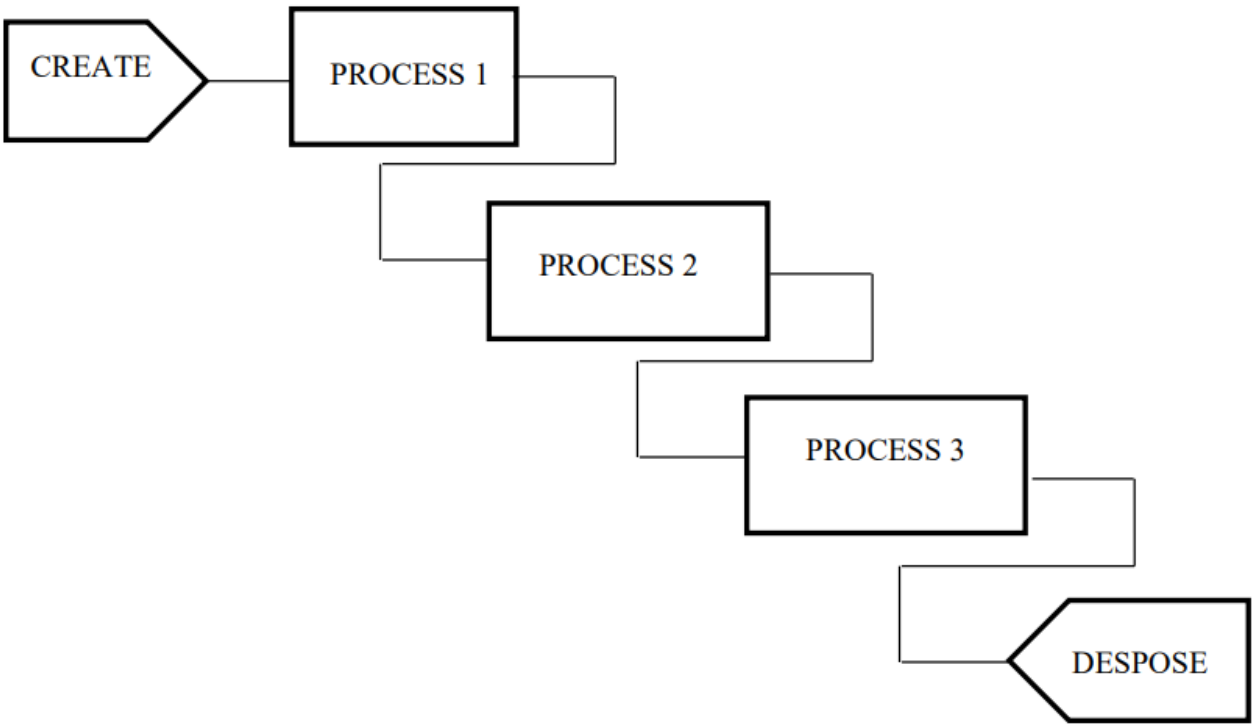
1) Загальна кількість запитів надходження у пристрій – Quantity;
2) Кількість об’єктів, що обробилася успішно – QuantityProcessed;
3) Кількість часу роботи пристрою – WorkTime;
4) Кількість відмов у надходження у пристрій – Failure;
5) Довжина черги за увесь час – FullMeanQueue += Queue * deltaT;

Ці дані підраховуються та оновлюються віртуальним методом DoStatistics, та у результаті виконання InAct() або OutAct(). В результаті кінця симуляції, підраховуємо фінальні значення:

1) Частка роботи пристрою – WorkTimePart = WorkTime / AllTime;
2) Середня довжина черги – MeanQueue = FullMeanQueue / AllTime;
3) Ймовірність відмови – FailureProbability = Failure / Quantity;

Побудова системи

Схема моделі



Побудова моделі

```
Create c = new Create(1);
Process p1 = new Process(1, 2.0, maxQueue: 3);
Process p2 = new Process(1, 2.0, maxQueue: 3);
Process p3 = new Process(1, 2.0, maxQueue: 3);

c.SetNextElement(p1);
p1.SetNextElement(p2);
p2.SetNextElement(p3);

Model model = new Model(new List<Element>() {c, p1, p2, p3});
model.Simulate(100.0);
```

Де у процес передаємо параметри, що відповідають за кількість підпроцесів, затримка та значення максимальної черги відповідно. У створюючий елемент передаємо тільки затримку.

Приклад виведення інформації про елементи

```
Next event: CREATE_0  Time: 22.313663340389418 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CREATE_0 quantity = 18 tnext= 22.61750720585366

PROCESS_1 state = True quantity = 18 tnext= 22.413966783624502 queue = 3 failure = 6
        subProcess_1.0 state = True quantity = 9 tnext= 22.413966783624502

PROCESS_2 state = False quantity = 8 tnext= ∞ queue = 0 failure = 0
        subProcess_2.0 state = False quantity = 8 tnext= ∞

PROCESS_3 state = False quantity = 8 tnext= 22.78464185768311 queue = 0 failure = 0
        subProcess_3.0 state = True quantity = 8 tnext= 22.78464185768311
```

Виведення результату симуляції

```
-------------RESULTS-------------
CREATE_0:
        Quantity = 107
PROCESS_1:
        Quantity = 107
        WorkTime = 0.6153839366889434
        Mean length of queue = 1.907826343855623
        Failure probability = 0.45794392523364486
        subProcess_1.0:
                Quantity = 58
PROCESS_2:
        Quantity = 57
        WorkTime = 0.48859670604624883
        Mean length of queue = 1.2790179657073748
        Failure probability = 0.15789473684210525
        subProcess_2.0:
                Quantity = 45
PROCESS_3:
        Quantity = 44
        WorkTime = 0.5145083482290176
        Mean length of queue = 1.0290061533626047
        Failure probability = 0.06818181818181818
        subProcess_3.0:
                Quantity = 40
```

Верифікація моделі

Аналіз системи

Як бачимо, кількість створених об’єктів – 107, а пройшовших всю симуляцію – 40. Як видно з результатів, найбільший показник відмови у Process_1. Це не є випадковим, оскільки затримка Create та Process_1 є 1 та 2 відповідно. Для зменшення значення відмови необхідно, щоб робота елементів мала +- однакове значення, або потребує більшої кількості підпроцесів у пристрої, що довго обробляє.

Для першого варіанту, просто візьмемо однаковий час затримки. Та перевіримо роботу при необмеженій черзі при довгій симуляції, щоб подивитися якою стане черга в самому кінці. Отже додатково ще будемо виводити кількість елементів у черзі в кінці симуляції.

Модифікація системи

```
Create c = new Create(1);
Process p1 = new Process(1, maxQueue: int.MaxValue);
Process p2 = new Process(1, maxQueue: int.MaxValue);
Process p3 = new Process(1, maxQueue: int.MaxValue);

c.SetNextElement(p1);
p1.SetNextElement(p2);
p2.SetNextElement(p3);

Model model = new Model(new List<Element>() {c, p1, p2, p3});
model.Simulate(3000000);
```

Результати

```
CREATE_0:
        Quantity = 2999789
PROCESS_1:
        Quantity = 2999789
        WorkTime = 0.4997399994260378
        Mean length of queue = 594.5418353915371
        Failure probability = 0
        Final queue = 578
        subProcess_1.0:
                Quantity = 2999211
PROCESS_2:
        Quantity = 2999210
        WorkTime = 0.49971697071646504
        Mean length of queue = 564.6833617376161
        Failure probability = 0
        Final queue = 311
        subProcess_2.0:
                Quantity = 2998899
PROCESS_3:
        Quantity = 2998898
        WorkTime = 0.4997867501132839
        Mean length of queue = 1592.9048808259686
        Failure probability = 0
        Final queue = 947
        subProcess_3.0:
                Quantity = 2997951
```

На майже 3 мільйони створених маємо черги 578, 311, 947, що є тисячною часткою. Спробуємо без черги, проаналізуємо тепер відмови:

```
------------RESULTS------------
CREATE_0:
        Quantity = 2999135
PROCESS_1:
        Quantity = 2999135
        WorkTime = 0.5001131418347128
        Mean length of queue = 0
        Failure probability = 0.5001982238212018
        subProcess_1.0:
                Quantity = 1498973
PROCESS_2:
        Quantity = 1498972
        WorkTime = 0.37479398796461194
        Mean length of queue = 0
        Failure probability = 0.24938357754514429
        subProcess_2.0:
                Quantity = 1125153
PROCESS_3:
        Quantity = 1125153
        WorkTime = 0.312396160916294
        Mean length of queue = 0
        Failure probability = 0.16661734004175432
        subProcess_3.0:
                Quantity = 937683
```

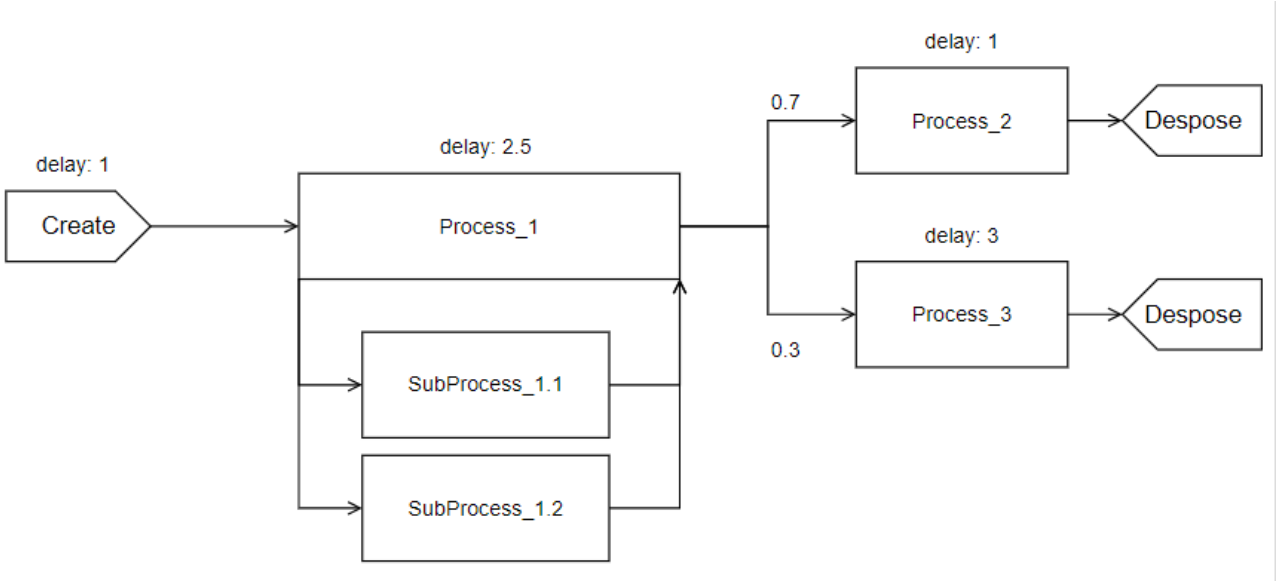
Результат свідчить про те наскільки наявність черги (навіть невеликої) допомагає зменшити кількість відмов. Нехай значення максимальної черги буде 100:

```
CREATE_0:
        Quantity = 2999691
PROCESS_1:
        Quantity = 2999691
        WorkTime = 0.49994406812904024
        Mean length of queue = 49.13951490042288
        Failure probability = 0.009847680977807381
        subProcess_1.0:
                Quantity = 2970145
PROCESS_2:
        Quantity = 2970144
        WorkTime = 0.4974306414037284
        Mean length of queue = 40.37434841667229
        Failure probability = 0.005163049333635002
        subProcess_2.0:
                Quantity = 2954756
PROCESS_3:
        Quantity = 2954755
        WorkTime = 0.49637083718069586
        Mean length of queue = 37.28156779430216
        Failure probability = 0.003683892573157504
        subProcess_3.0:
                Quantity = 2943857
```

В результаті можливість відмови стає менше 0.01%.

Демонстрація роботи підпроцесів та переходів із ймовірністями

Схема моделі



Побудова моделі

```
Create c = new Create(1);
Process p1 = new Process(2, delay:2.5, maxQueue: 10);
Process p2 = new Process(1, delay: 1, maxQueue: 10);
Process p3 = new Process(1, delay: 3, maxQueue: 10);

c.SetNextElement(p1);
p1.SetNextElement(p2, 0.8);
p1.SetNextElement(p3, 0.2);

Model model = new Model(new List<Element>() {c, p1, p2, p3});
model.Simulate(3000000.0);
```

Результати

```
-------------RESULTS-------------
CREATE_0:
        Quantity = 3002891
PROCESS_1:
        Quantity = 3002891
        WorkTime = 0.4805206832690428
        Mean length of queue = 2.219689740200375
        Failure probability = 0.021279160648854722
        subProcess_1.0:
                Quantity = 1043068
        subProcess_1.1:
                Quantity = 983951
        subProcess_1.2:
                Quantity = 911967
PROCESS_2:
        Quantity = 2351123
        WorkTime = 0.4397868093399153
        Mean length of queue = 2.146548456322503
        Failure probability = 0.01330980982279532
        subProcess_2.0:
                Quantity = 2319826
PROCESS_3:
        Quantity = 587860
        WorkTime = 0.3702257169130811
        Mean length of queue = 0.8176149468821741
        Failure probability = 0.0010104446636954377
        subProcess_3.0:
                Quantity = 587266
```

Модифікація системи

Бачимо занадто велику чергу у Process_1, можна додати ще один підпроцес. Також трохи збалансуємо WorkTime між Process_2 та Process_3. Нехай ймовірність переходу буде 0.76 та 0.24 відповідно.

Результати

```
------------RESULTS------------
CREATE_0:
        Quantity = 3001097
PROCESS_1:
        Quantity = 3001097
        WorkTime = 0.4797544603397925
        Mean length of queue = 2.194288329989743
        Failure probability = 0.02042186573776189
        subProcess_1.0:
                Quantity = 1044697
        subProcess_1.1:
                Quantity = 984349
        subProcess_1.2:
                Quantity = 910753
PROCESS_2:
        Quantity = 2234692
        WorkTime = 0.4270425076844919
        Mean length of queue = 1.7895946580048812
        Failure probability = 0.008462016242059308
        subProcess_2.0:
                Quantity = 2215779
PROCESS_3:
        Quantity = 705104
        WorkTime = 0.41489437346604296
        Mean length of queue = 1.4798220357076612
        Failure probability = 0.005851619051941274
        subProcess_3.0:
                Quantity = 700978
```

Як видно з результатів, черга зменшилася з 6.8 до 2.1 у Process_1. При цьому це не особливо повпливало на показник відмови у пристрої. Збалансування роботи пристроїв Process_2 та Process_3 у свою чергу зменшила кількість відмов Process_2 з 0.013 до 0.008.

Висновок

В ході виконання лабораторної необхідно було ознайомитись та відтворити об’єктно-орієнтований алгоритм імітації за принципом Δt, оскільки він є найбільш універсальним.

Була створена архітектура на основі схеми, яка загально розділяє свої функції між секторами: модель, елементи моделі, вивід та наступний елемент.

Принципи ООП роблять програму більш гнучкою для побудови більш складних моделей на основі сформованих простих елементів моделі із можливістю створювати підпроцеси та переходи до наступних блоків із ймовірністю.

Для дослідження моделей був необхідний збір статистики. Отже, були додані додаткові атрибути у елементи для утримання інформації у процесі роботи: загальна кількість запитів надходження у пристрій (Quantity); кількість об’єктів, що обробилася успішно (QuantityProcessed), загальна кількість часу роботи пристрою (WorkTime); кількість відмов (Failure) і так далі.

На основі значень вищезазначених атрибутів в кінці вираховується статистика наступних значень: середнього завантаження пристрою (WorkTimePart), середня довжина черги (MeanQueue) та ймовірність відмови (FailureProbability).

В кінці було продемонстровано декілька варіацій моделей різної поведінки із наступними аналізом та модифікацією значень параметрів для досягнення найкращих результатів роботи відповідної моделі.

Код

Model.cs

```
using Lab2.Elements;
using Lab2.Print;
namespace Lab2;

public class Model
{
   private readonly List<Element> _elements;
   private double _tnext, _tcurr;
   private int _event;

   public Model(List<Element> elements)
   {
       _elements = elements;
   }

   public void Simulate(double time)
   {
       int printTrigger = 0;
       while (_tcurr < time)
       {
           UpdateEventAndNextT();
           // IPrinter.PrintCurrent(_elements[_event]);
           foreach (var element in _elements)
               element.DoStatistics(_tnext - _tcurr);

           _tcurr = _tnext;
           UpgradeCurrTForAllElements();
           OutActForAllCurrentElements();
           // IPrinter.Info(_elements);
           if (_tcurr >= printTrigger)
           {
               printTrigger += 100000;
               Console.WriteLine($"tcurr = {_tcurr}");
           }
       }
       IPrinter.Result(_elements);
   }

    private void UpgradeCurrTForAllElements() => _elements.ForEach(e => e.CurrT =
_tcurr);

   private void UpdateEventAndNextT()
   {
       _tnext = double.MaxValue;
       for (int i = 0; i < _elements.Count; i++)
       {
           if (_elements[i].NextT < _tnext)
           {
               _tnext = _elements[i].NextT;
               _event = i;
           }
       }
   }

      private void OutActForAllCurrentElements()=> foreach (var element in _elements) if
(element.NextT == _tcurr) element.OutAct();
}
```

Element.cs

```
using DistributionRandomizer.DelayRandomizers;
using Lab2.NextElement;
using Lab2.Print;
namespace Lab2.Elements;

public abstract class Element
{
    public double NextT { get; protected set; }
    public double CurrT { get; set; }
    public int Quantity { get; private set; }
    public int QuantityProcessed { get; private set; }
    public double WorkTime { get; private set; }
    public bool IsWorking { private set; get; }
    public IPrinter Print { get; protected init; }
    public readonly string Name;

    private NextElements? _nextElement;
    private readonly double _delayMean;
    private readonly double _delayDeviation;
    private readonly Randomizer? _randomizer;

    protected readonly int Id = _nextId;
    private static int _nextId;

    protected Element(double delay, string name, string distribution = "exp")
    {
        _delayMean = delay;
        Name += $"{name}_{Id}";
        _randomizer = GetRandomizerByName(distribution);
        _nextId++;
        Print = new ElementPrinter(this);
    }

    public void SetNextElement(Element element, double probability = 1)
    {
        _nextElement ??= new NextElements();
        _nextElement.AddNextElement(element, probability);
    }

    public virtual void InAct()
    {
        Quantity++;
        IsWorking = true;
    }

    public virtual void OutAct()
    {
        QuantityProcessed++;
        IsWorking = false;
        _nextElement?.InAct();
        UpdateNextT();
    }

    public virtual void DoStatistics(double t)=>WorkTime += IsWorking ? t : 0;

    protected double GetDelay()
    {
        if (_randomizer != null) return _randomizer.GenerateDelay();
        return _delayMean;
```

```
    }

    protected abstract void UpdateNextT();
    private Randomizer GetRandomizerByName(string distribution)
    {
        switch (distribution.ToLower())
        {
            case "exp":
                return new ExponentialRandomizer(_delayMean);
            case "norm":
                return new NormalRandomizer(_delayMean, _delayDeviation);
            case "unif":
                return new UniformRandomizer(_delayMean, _delayDeviation);
            case "":
            case null:
                return null!;
            default:
                throw new ArgumentException("Unknown distribution");
        }
    }
}
```

Create.cs

```
using Lab2.Print;
namespace Lab2.Elements;
public class Create : Element
{
    public Create(double delay = 1.0, string distribution = "exp", string name =
"CREATE") : base(delay, name, distribution)
    {
        Print = new CreatePrinter(this);
    }
    protected override void UpdateNextT()=>NextT = CurrT + GetDelay();
}
```

Process.cs

```
using Lab2.Print;
namespace Lab2.Elements;
public class Process : Element
{
    public int Failure { get; private set; }
    public double MeanQueue { get; private set; }
    public int Queue { get; private set; }
    public List<SubProcess> SubProcesses { get; } = new();
    private readonly int _maxQueue;

    private int WorkingSubProcessesCount => SubProcesses.Count(s => s.IsWorking);
    private SubProcess FreeSubProcess => SubProcesses.First(s => !s.IsWorking);
    private List<SubProcess> BusySubProcesses => SubProcesses.Where(s => s.NextT <=
CurrT && s.IsWorking).ToList();

    public Process(int subProcessCount, double delay = 1.0, string distr = "exp",
string name = "PROCESS", int maxQueue = int.MaxValue) : base(delay, name, distr)
    {
        for (int i = 0; i < subProcessCount; i++)
            SubProcesses.Add(new SubProcess(Id, i));
        _maxQueue = maxQueue;
        NextT = double.MaxValue;
        Print = new ProcessPrinter(this);
    }
```

```
    public override void InAct()
    {
        base.InAct();
        if (WorkingSubProcessesCount < SubProcesses.Count)
            FreeSubProcess.InAct(CurrT + GetDelay());
        else
        {
            if (Queue < _maxQueue) Queue++;
            else Failure++;
        }
        UpdateNextT();
    }

    public override void OutAct()
    {
        foreach (var subProcess in BusySubProcesses)
        {
            subProcess.OutAct();
            base.OutAct();
            if (Queue > 0)
            {
                Queue--;
                subProcess.InAct(CurrT + GetDelay());
            }
        }
        UpdateNextT();
    }

    public override void DoStatistics(double delta)
    {
        base.DoStatistics(delta);
        foreach (var subProcess in SubProcesses) subProcess.DoStatistics(delta);
        MeanQueue += Queue * delta;
    }

    protected override void UpdateNextT() => NextT = SubProcesses.Min(s =>
s.NextT);
}
```

SubProcess.cs

```
using Lab2.Print;
namespace Lab2.Elements;
public class SubProcess
{
    public readonly string Name;
    public bool IsWorking { get; private set; }
    public double NextT { get; private set; } = double.MaxValue;
    public int Quantity { get; private set; }
    public double WorkTime { get; private set; }
    public SubProcessPrinter Printer { get; private init; }

    public SubProcess(int processId, int subProcessId)
    {
        Name = $"subProcess_{processId}.{subProcessId}";
        Printer = new SubProcessPrinter(this);
    }

    public void InAct(double nextT)
    {
        IsWorking = true;
```

```
        Quantity++;
        NextT = nextT;
    }

    public void OutAct()
    {
        IsWorking = false;
        NextT = double.MaxValue;
    }

    public void DoStatistics(double delta)=>WorkTime += IsWorking? delta : 0;
}
```

NextElement.cs

```
using Lab2.Elements;
namespace Lab2.NextElement;
public class NextElements
{
    private List<NextElement> NextElementsList { get; } = new();
    public void AddNextElement(Element element, double probability = 1) =>
NextElementsList.Add(new NextElement(element, probability));

    private NextElement GetNextElement()
    {
        double sum = NextElementsList.Sum(nextElement=>nextElement.Probability);
        double random = new Random().NextDouble() * sum;
        double current = 0;
        foreach (var nextElement in NextElementsList)
        {
            current += nextElement.Probability;
            if (random < current) return nextElement;
        }
        throw new Exception("NextElement not found");
    }

    public void InAct()
    {
        GetNextElement().Element.InAct();
    }
}
```

NextElement.cs

```
using Lab2.Elements;
namespace Lab2.NextElement;
public struct NextElement
{
    public readonly Element Element;
    public readonly double Probability;
    public NextElement(Element element, double probability)
    {
        Element = element;
        Probability = probability;
    }
}
```

IPrinter.cs

```
using System.Globalization;
using Lab2.Elements;
namespace Lab2.Print;
```

```
public interface IPrinter
{
   public void Statistics();
   public void Info();

   public static string Format(double num) => num == double.MaxValue ? "\u221E" :
num.ToString(CultureInfo.InvariantCulture);

   public static void Info(List<Element> elements)=> foreach (var element in
elements)element.Print.Info();

   public static void Result(List<Element> elements)
   {
       Console.WriteLine("\n-------------RESULTS-------------");
       foreach (var element in elements)element.Print.Statistics();
   }

   public static void PrintCurrent(Element element)=> Console.WriteLine($"\nNext
event: {element.Name}  Time: {element.NextT} ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
}
```

ProcessPrinter.cs

```
using Lab2.Elements;
namespace Lab2.Print;

public class ProcessPrinter : IPrinter
{
   private Process p;

   public ProcessPrinter(Process process)
   {
       p = process;
   }

   public void Info()
   {
       Console.WriteLine($"{p.Name} state = {p.IsWorking} quantity = {p.Quantity}
tnext= {IPrinter.Format(p.NextT)} queue = {p.Queue} failure = {p.Failure}");
       foreach (var subProcess in p.SubProcesses)subProcess.Printer.Info();
       Console.WriteLine();
   }

   public void Statistics()
   {
       Console.WriteLine($"{p.Name}:");
       Console.WriteLine($"\tQuantity = {p.Quantity}");
       Console.WriteLine($"\tWorkTime = {p.WorkTime / p.CurrT}");
       Console.WriteLine($"\tMean length of queue = {p.MeanQueue / p.CurrT}");
       Console.WriteLine($"\tFailure probability = {p.Failure /
(double)p.Quantity}");
       // Console.WriteLine($"\tFinal queue = {p.Queue}");
       foreach (var subP in p.SubProcesses) subP.Printer.Statistics();
   }
}
```

CreatePrinter.cs

```
using Lab2.Elements;
namespace Lab2.Print;

public class CreatePrinter : IPrinter
```

```
{
    private Create c;

    public CreatePrinter(Create create)
    {
        c = create;
    }

    public void Info()
    {
        Console.WriteLine($"{c.Name} quantity = {c.QuantityProcessed} tnext=
{IPrinter.Format(c.NextT)}\n");
    }

    public void Statistics()
    {
        Console.WriteLine($"{c.Name}:");
        Console.WriteLine($"\tQuantity = {c.QuantityProcessed}");
    }
}
```

SubProcessPrinter.cs

```
using Lab2.Elements;
namespace Lab2.Print;

public class SubProcessPrinter : IPrinter
{
    private SubProcess s;

    public SubProcessPrinter(SubProcess subProcess)
    {
        s = subProcess;
    }

    public void Statistics()
    {
        Console.Out.WriteLine($"\t{s.Name}:");
        Console.WriteLine($"\t\tQuantity = {s.Quantity}");
    }

    public void Info()
    {
        Console.WriteLine($"\t{s.Name} state = {s.IsWorking} quantity =
{s.Quantity} tnext= {IPrinter.Format(s.NextT)}");
    }
}
```

ElementPrinter.cs

```
using Lab2.Elements;
namespace Lab2.Print;

public class ElementPrinter : IPrinter
{
    private Element e;

    public ElementPrinter(Element element)
    {
        e = element;
    }
```

```
    public void Info()
    {
        Console.WriteLine($"{e.Name} state = {e.IsWorking} quantity = {e.Quantity}
tnext= {IPrinter.Format(e.NextT)}\n");
    }

    public void Statistics()
    {
        Console.Out.WriteLine($"{e.Name}:");
        Console.WriteLine($"\tQuantity = {e.Quantity}");
        Console.WriteLine($"\tQuantity processed = {e.QuantityProcessed}");
        Console.WriteLine($"\tWorkTime = {e.WorkTime}");
    }
}
```