МІНІСТЕРСТВО ОСВІТИ І НАУКИ 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-13 Бабіч Денис* _____

(шифр, прізвище, ім'я, по батькові)

Перевірив(ла) *Дифучин А. Ю.* _____

(посада, прізвище, ім'я, по батькові)

Київ 2024

ОСНОВНА ЧАСТИНА

Мета роботи: Реалізувати алгоритм імітації дискретно-подійних систем різної складності.

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.**

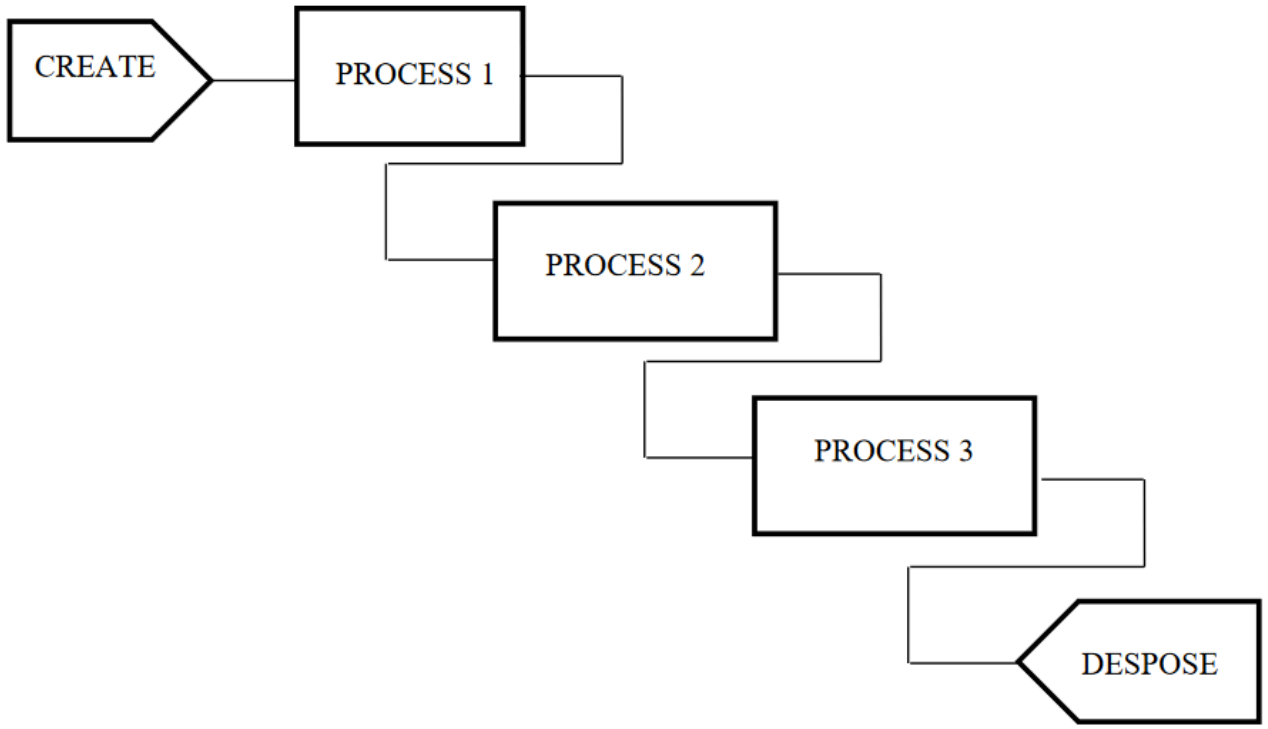

Рисунок 2.1 – Схема моделі.

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.**

Рисунок 1.1 – Завдання лабораторного практикуму

```
|REPORT| [create] Tasks: 190; Delay mean: 5.2761636.
|REPORT| [processor1] Delay mean: 5.367814; Successes: 163; Queue mean: 2.125111; Failures: 26; Failure probability: 0.13756613;
|REPORT| [processor2] Delay mean: 5.699886; Successes: 140; Queue mean: 1.4642352; Failures: 18; Failure probability: 0.11392405;
|REPORT| [processor3] Delay mean: 4.851845; Successes: 129; Queue mean: 0.6578967; Failures: 11; Failure probability: 0.07857143;
```

Рисунок 1.2 – Приклад виводу результатів роботи моделі

Таблиця 1.1 – Верифікація імітаційної моделі при різних вхідних даних

Час імітації	Середня затримка створення	Середня затримка обробки	Кількість відмов	Кількість успішних операцій	Середня довжина черги	Відсоток відмов
1000	4.85	5.329	32	172	2.6	0.157
	4.85	4.62	11	158	1.16	0.065
	4.85	4.5	9	147	0.63	0.058
1000	4.5	5.3	45	171	2.5	0.2
	4.5	4.94	12	159	1.1	0.091
	4.5	4.65	26	129	1.06	0.07
5000	5	5.033	137	851	2.12	0.14
	5	4.93	96	753	1.3	0.113
	5	4.87	65	686	0.75	0.087
5000	4.75	4.95	159	872	2.256	0.15
	4.75	4.88	84	783	1.46	0.097
	4.75	4.83	77	706	0.85	0.098

ВИСНОВКИ

У рамках виконання лабораторної роботи було реалізовано алгоритм імітації моделі обслуговування, використовуючи об’єктно-орієнтований підхід.

Спочатку було створено базову модель, що дозволяє відслідковувати процеси обслуговування з одним пристроєм. Після цього була здійснена модифікація алгоритму, зокрема додано обчислення середнього завантаження пристрою, що дозволило отримати статистичні дані для перевірки ефективності роботи. Для забезпечення коректності моделі було проведено верифікацію шляхом зміни значень вхідних змінних та параметрів моделі, і результати верифікації були оформлені у вигляді таблиці, що дозволило виявити потенційні недоліки або помилки в моделюванні.

В рамках подальших модифікацій було доопрацьовано клас PROCESS для моделювання процесу обслуговування кількома ідентичними пристроями, що дозволило змоделювати більш складні системи обслуговування.

Останнім етапом роботи стала модифікація цього класу для організації виходу в два та більше наступних блоків, у тому числі з можливістю повернення у попередні блоки. Це дозволило значно розширити можливості моделювання, створивши гнучку систему для реалізації складних процесів обслуговування з кількома шляхами.

В результаті виконання лабораторної роботи було досягнуто поставлених цілей: створено працюючу модель обслуговування з одним і кількома пристроями, а також реалізовано додаткові функціональні можливості для більш складних варіантів обслуговування з умовами зворотного зв'язку. Модель пройшла верифікацію, і були отримані наочні результати, що підтвердили коректність реалізації.

ДОДАТОК ПРОГРАМНИЙ КОД

Programs.cs

```
using System.Collections.Generic;
using LabWork2.Framework.Core.Controllers;
using LabWork2.Framework.Components.Modules.Common;
using LabWork2.Framework.Components.Modules.Concrete;
using LabWork2.Framework.Components.Workers.Concrete;

namespace LabWork2.Application;

file sealed class Program
{
   private static void Main()
   {
      Program.RunExampleModel();
      // Program.RunAdvancedModel();
   }

   private static void RunExampleModel()
   {
                  CreateModule create = new CreateModule("create", new
MockExponentialWorker(5.0f));
         ProcessorModule processor1 = new ProcessorModule("processor1", new
MockExponentialWorker(5.0f), 5);
         ProcessorModule processor2 = new ProcessorModule("processor2", new
MockExponentialWorker(5.0f), 4);
         ProcessorModule processor3 = new ProcessorModule("processor3", new
MockExponentialWorker(5.0f), 3);
```

```
        create.AttachModule(processor1, 1.0f);
        processor1.AttachModule(processor2, 1.0f);
        processor2.AttachModule(processor3, 1.0f);

            new SystemModelController(new List<Module>() { create, processor1, processor2, processor3 }).RunSimulation(1000);
    }

    private static void RunAdvancedModel()
    {
                    CreateModule create = new CreateModule("create", new MockUniformWorker(1.0f, 5.0f));
            ProcessorModule processor1 = new ProcessorModule("processor1", new MockExponentialWorker(2.5f), 3);
            ProcessorModule processor2 = new ProcessorModule("processor2", new MockNormalWorker(2.5f, 0.5f), 2);
                                MultiProcessorModule multiProcessor3 = new MultiProcessorModule("multiProcessor3", new MockExponentialWorker(2.25f), 0, 2);

        create.AttachModule(processor1, 1.0f);
        processor1.AttachModule(processor2, 0.5f);
        processor1.AttachModule(multiProcessor3, 0.5f);
        processor2.AttachModule(processor1, 0.5f);
        processor2.AttachModule(multiProcessor3, 0.5f);
            new SystemModelController(new List<Module>() { create, processor1, processor2, multiProcessor3 }).RunSimulation(1000);
    }
}
```

SystemModelController.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using LabWork2.Framework.Common;
using LabWork2.Framework.Components.Modules.Common;

namespace LabWork2.Framework.Core.Controllers;

internal sealed class SystemModelController : IStatisticsPrinter
{
    private readonly IList<Module> modules;

    private float timeNext;
    private float timeCurrent;

    internal SystemModelController(IList<Module> modules)
    {
        this.timeNext = 0.0f;
        this.modules = modules;
        this.timeCurrent = 0.0f;
    }

    internal void RunSimulation(float simulationTime)
    {
        IList<Module> nextModules;
        this.timeNext = this.modules.Min(module => module.TimeNext);

        while (timeNext < simulationTime)
```

```
        {
            this.timeCurrent = this.timeNext;
            foreach (Module module in this.modules)
                module.TimeCurrent = this.timeCurrent;
                    nextModules = this.modules.Where(module => module.TimeNext ==
this.timeCurrent).ToList();

            foreach (Module module in nextModules)
                module.CompleteTask();

            this.timeNext = this.modules.Min(module => module.TimeNext);

            this.PrintIntermediateStatistics();
        }
        this.PrintFinalStatistics();
    }

    public void PrintFinalStatistics()
    {
        Console.Write('\n');
        foreach (Module module in this.modules)
            module.PrintFinalStatistics();
    }
    public void PrintIntermediateStatistics()
    {
        foreach (Module module in this.modules)
            module.PrintIntermediateStatistics();
    }
}
```

CreateModule.cs

```
using System;
using LabWork2.Framework.Components.Workers.Common;
using LabWork2.Framework.Components.Modules.Common;

namespace LabWork2.Framework.Components.Modules.Concrete;

internal sealed class CreateModule : Module
{
   private readonly IMockWorker mockWorker;

   private int tasksCount;
   private float totalDelayPayloads;

         internal CreateModule(string identifier, IMockWorker mockWorker) :
base(identifier)
   {
      if (mockWorker == null)
            throw new ArgumentNullException($"{nameof(mockWorker)} cannot be
null.");

      this.mockWorker = mockWorker;

      this.MoveTimeline();
   }

   internal override sealed float TimeCurrent { get; set; }

   internal sealed override void AcceptTask()
```

```
    {
        throw new InvalidOperationException($"{base.Identifier} ({this.GetType()}) is not able to accept tasks.");
    }

    internal override sealed void CompleteTask()
    {
        base.NextModule?.AcceptTask();
        this.MoveTimeline();
        ++this.tasksCount;
    }
    private protected override sealed void MoveTimeline()
    {
        float delayPayload = this.mockWorker.DelayPayload;
        this.TimeNext = this.TimeCurrent + delayPayload;
        this.totalDelayPayloads += delayPayload;
    }
    public override sealed void PrintFinalStatistics()
    {
        Console.Write($"|REPORT| [{base.Identifier}] ");
                    Console.WriteLine($"Tasks: {this.tasksCount}; Delay mean: {this.totalDelayPayloads / this.tasksCount}.");
    }
    public override sealed void PrintIntermediateStatistics()
    {
        Console.Write($"|LOG| (STATS) [{base.Identifier}] ");
        Console.WriteLine($"Tasks: {this.tasksCount}; Time: {base.TimeNext}.");
    }
}
```

Processor.cs

```
using System;
using LabWork2.Framework.Components.Modules.Common;
using LabWork2.Framework.Components.Workers.Common;

namespace LabWork2.Framework.Components.Modules.Concrete;

internal sealed class ProcessorModule : Module
{
   private readonly int maxQueueLength;
   private readonly IMockWorker mockWorker;

   private int queueLength;
   private float queueLengthSum;

   private float timeCurrent;
   private int failuresCount;
   private int successesCount;
   private float totalDelayPayloads;

        internal ProcessorModule(string identifier, IMockWorker mockWorker, int maxQueueLength) : base(identifier)
   {
      if (mockWorker == null)
            throw new ArgumentNullException($"{nameof(mockWorker)} cannot be null.");

      if (maxQueueLength < 0)
```

```
            throw new ArgumentException($"{nameof(maxQueueLength)} cannot be less than 0.");

        this.mockWorker = mockWorker;
        this.maxQueueLength = maxQueueLength;
    }

    internal bool IsBusy { get; private set; }

    internal override sealed float TimeCurrent
    {
        get => this.timeCurrent;
        set
        {
            this.queueLengthSum += (value - this.timeCurrent) * this.queueLength;
            this.timeCurrent = value;
        }
    }

    internal sealed override void AcceptTask()
    {
        if (this.IsBusy)
        {
            if (this.queueLength < this.maxQueueLength)
                ++this.queueLength;
            else
                ++this.failuresCount;
        }
        else
```

```
    {
        this.IsBusy = true;
        this.MoveTimeline();
    }
}

internal override sealed void CompleteTask()
{
    ++this.successesCount;

    if (this.queueLength == 0)
    {
        this.IsBusy = false;
        base.TimeNext = Single.MaxValue;
    }
    else
    {
        --this.queueLength;
        this.MoveTimeline();
    }

    base.NextModule?.AcceptTask();
}

private protected override sealed void MoveTimeline()
{
    float delayPayload = this.mockWorker.DelayPayload;
    this.TimeNext = this.TimeCurrent + delayPayload;
    this.totalDelayPayloads += delayPayload;
```

```
    }

    public override sealed void PrintFinalStatistics()
    {
        float averageQueueLength = this.queueLengthSum / this.TimeCurrent;
        float averageDelayPayload = this.totalDelayPayloads / this.successesCount;
                    float failureProbability = (this.successesCount == 0 ? 0 :
(float)this.failuresCount / (this.failuresCount + this.successesCount));

        Console.Write($"|REPORT| [{base.Identifier}] ");
                Console.WriteLine($"Delay mean: {averageDelayPayload}; Successes:
{this.successesCount}; Queue mean: {averageQueueLength}; Failures:
{this.failuresCount}; Failure probability: {failureProbability}; ");
    }

    public override sealed void PrintIntermediateStatistics()
    {
        Console.Write($"|LOG| (STATS) [{base.Identifier}] ");
                        Console.WriteLine($"Queue: {this.queueLength}; Failures:
{this.failuresCount}; Time: {this.TimeNext}.");
    }
}
```

MultiProcessorModule.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using LabWork2.Framework.Components.Modules.Common;
using LabWork2.Framework.Components.Workers.Common;

namespace LabWork2.Framework.Components.Modules.Concrete;

internal sealed class MultiProcessorModule : Module
{
    private readonly int maxQueueLength;
    private readonly IMockWorker mockWorker;
    private readonly IList<ProcessorModule> subProcessorsModules;

    private bool isBusy;
    private int queueLength;
    private float queueLengthSum;
    private IList<ProcessorModule>? busySubProcessorsModules;

    private float timeCurrent;
    private int failuresCount;
    private int successesCount;

      internal MultiProcessorModule(string identifier, IMockWorker mockWorker, int
maxQueueLength, int subProcessorsCount) : base(identifier)
    {
       if (mockWorker == null)
```

```
            throw new ArgumentNullException($"{nameof(mockWorker)} cannot be
null.");

        if (maxQueueLength < 0)
            throw new ArgumentException($"{nameof(maxQueueLength)} cannot be
less than 0.");

        if (subProcessorsCount <= 0)
            throw new ArgumentException($"{nameof(subProcessorsCount)} cannot be
less or equals 0.");

        this.mockWorker = mockWorker;
        this.maxQueueLength = maxQueueLength;
        this.subProcessorsModules = new List<ProcessorModule>();

        for (int i = 0; i < subProcessorsCount; ++i)
            this.subProcessorsModules.Add(new ProcessorModule($"{identifier}_{i}",
mockWorker, 0));
    }

    internal override sealed float TimeCurrent
    {
        get => this.timeCurrent;
        set
        {
            this.queueLengthSum += (value - this.timeCurrent) * this.queueLength;
            this.timeCurrent = value;

            foreach (ProcessorModule processorModule in this.subProcessorsModules)
```

```
            processorModule.TimeCurrent = this.timeCurrent;
        }
    }

    internal sealed override void AcceptTask()
    {
        if (this.isBusy)
        {
            if (this.queueLength < this.maxQueueLength)
                ++this.queueLength;
            else
                ++this.failuresCount;
        }
        else
        {
                                                ProcessorModule? processorModule =
this.subProcessorsModules.Where(processorModule                                    =>
!processorModule.IsBusy).FirstOrDefault();

            if (processorModule != null)
            {
                Console.WriteLine($"|LOG| (TRACE) {this.Identifier} sends task to the
{processorModule.Identifier}.");
                processorModule.AcceptTask();
            }
            else
            {
                Console.WriteLine($"|LOG| (TRACE) {this.Identifier} disposes task.");
            }
```

```
                    // this.subProcessorsModules.Where(processorModule => !processorModule.IsBusy).FirstOrDefault()?.AcceptTask();
                this.isBusy = this.subProcessorsModules.All(subProcessorModule => subProcessorModule.IsBusy);
            this.MoveTimeline();
        }
    }

    internal override sealed void CompleteTask()
    {
        ++this.successesCount;
                                        this.busySubProcessorsModules = this.subProcessorsModules.Where(processorModule => processorModule.TimeNext == this.TimeNext).ToList();

        foreach (ProcessorModule processorModule in this.busySubProcessorsModules)
        {
            processorModule.CompleteTask();

            if (this.queueLength != 0)
            {
                --this.queueLength;
                processorModule.AcceptTask();
            }
        }

                this.isBusy = this.subProcessorsModules.All(subProcessorModule => subProcessorModule.IsBusy);
```

```
        this.MoveTimeline();
        base.NextModule?.AcceptTask();
    }
    private protected override sealed void MoveTimeline()
    {
                this.TimeNext = this.subProcessorsModules.Min(processorModule => processorModule.TimeNext);
    }
    public override sealed void PrintFinalStatistics()
    {
        float averageQueueLength = this.queueLengthSum / this.TimeCurrent;
                    float failureProbability = (this.successesCount == 0 ? 0 : (float)this.failuresCount / (this.failuresCount + this.successesCount));

        Console.Write($"|REPORT| [{base.Identifier}] ");
                        Console.WriteLine($"Failures: {this.failuresCount}; Successes: {this.successesCount}; Failure probability: {failureProbability}; Queue mean: {averageQueueLength}");

        foreach (ProcessorModule processorModule in this.subProcessorsModules)
            processorModule.PrintFinalStatistics();
    }
    public override sealed void PrintIntermediateStatistics()
    {
        Console.Write($"|LOG| (STATS) [{base.Identifier}] ");
                            Console.WriteLine($"Queue: {this.queueLength}; Failures: {this.failuresCount}; Time: {this.TimeNext}.");
    }
}
```