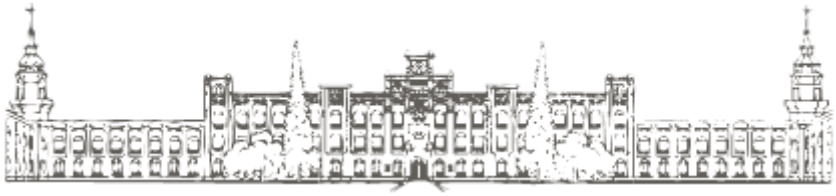


Міністерство освіти і науки України

Національний технічний університет України «КПІ

імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

ЗВІТ

лабораторної роботи №3

з дисципліни «Моделювання систем»

Перевірила:

Дифучина О. Ю.

Виконав:

Студент Гр. ІП-01

Пашковський Є. С.

Київ 2023

Завдання

1. Реалізувати універсальний алгоритм імітації моделі масового обслуговування з багатоканальним обслуговуванням, з вибором маршруту за пріоритетом або за заданою ймовірністю. **30 балів.**
2. Для наступного тексту задачі скласти формалізовану модель масового обслуговування та реалізувати її з використанням побудованого універсального алгоритму (**30 балів**):

   У банку для автомобілістів є два віконця, кожне з яких обслуговується одним касиром і має окрему під'їзну смугу. Обидві смуги розташовані поруч. З попередніх спостережень відомо, що інтервали часу між прибуттям клієнтів у годину пік розподілені експоненційно з математичним очікуванням, рівним 0,5 од. часу. Через те, що банк буває переобтяжений тільки в годину пік, то аналізується тільки цей період. Тривалість обслуговування в обох касирів однакова і розподілена експоненційно з математичним очікуванням, рівним 0,3 од. часу. Відомо також, що при рівній довжині черг, а також при відсутності черг, клієнти віддають перевагу першій смузі. В усіх інших випадках клієнти вибирають більш коротку чергу. Після того, як клієнт в'їхав у банк, він не може залишити його, доки не буде обслугований. Проте він може перемінити чергу, якщо стоїть останнім і різниця в довжині черг при цьому складає не менше двох автомобілів. Через обмежене місце на кожній смузі може знаходитися не більш трьох автомобілів. У банку, таким чином, не може знаходитися більш восьми автомобілів, включаючи автомобілі двох клієнтів, що обслуговуються в поточний момент касиром. Якщо місце перед банком заповнено до границі, то клієнт, що прибув, вважається втраченим, тому що він відразу ж виїжджає. Початкові умови такі: 1) обидва касири зайняті, тривалість обслуговування для кожного касира нормально розподілена з математичним очікуванням, рівним 1 од. часу, і середньоквадратичним відхиленням, рівним 0,3 од. часу; 2) прибуття першого клієнта заплановано на момент часу 0,1 од. часу; 3) у кожній черзі очікують по два автомобіля.

   Визначити такі величини: 1) середнє завантаження кожного касира; 2) середнє число клієнтів у банку; 3) середній інтервал часу між від'їздами клієнтів від вікон; 4) середній час перебування клієнта в банку; 5) середнє число клієнтів у кожній черзі; 6) відсоток клієнтів, яким відмовлено в обслуговуванні; 7) число змін під'їзних смуг.
3. Для наступного тексту задачі скласти формалізовану модель масового обслуговування та реалізувати її з використанням побудованого універсального алгоритму (**40 балів**):

   У лікарню поступають хворі таких трьох типів: 1) хворі, що пройшли попереднє обстеження і направлені на лікування; 2) хворі, що

бажають потрапити в лікарню, але не пройшли повністю попереднє обстеження; 3) хворі, які тільки що поступили на попереднє обстеження. Чисельні характеристики типів хворих наведені в таблиці:

Тип хворого	*Відносна частота*	*Середній час реєстрації, хв*
1	0,5	15
2	0,1	40
3	0,4	30

При надходженні в приймальне відділення хворий стає в чергу, якщо обидва чергових лікарі зайняті. Лікар, який звільнився, вибирає в першу чергу тих хворих, що вже пройшли попереднє обстеження. Після заповнення різноманітних форм у приймальне відділення хворі 1 типу ідуть прямо в палату, а хворі типів 2 і 3 направляються в лабораторію. Троє супровідних розводять хворих по палатах. Хворим не дозволяється направлятися в палату без супровідного. Якщо всі супровідні зайняті, хворі очікують їхнього звільнення в приймальному відділенні. Як тільки хворий доставлений у палату, він вважається таким, що завершив процес прийому до лікарні.

Хворі, що спрямовуються в лабораторію, не потребують супроводу. Після прибуття в лабораторію хворі стають у чергу в реєстратуру. Після реєстрації вони ідуть у кімнату очікування, де чекають виклику до одного з двох лаборантів. Після здачі аналізів хворі або повертаються в приймальне відділення (якщо їх приймають у лікарню), або залишають лікарню (якщо їм було призначено тільки попереднє обстеження). Після повернення в приймальне відділення хворий, що здав аналізи, розглядається як хворий типу 1.

У наступній таблиці приводяться дані по тривалості дій (хв):

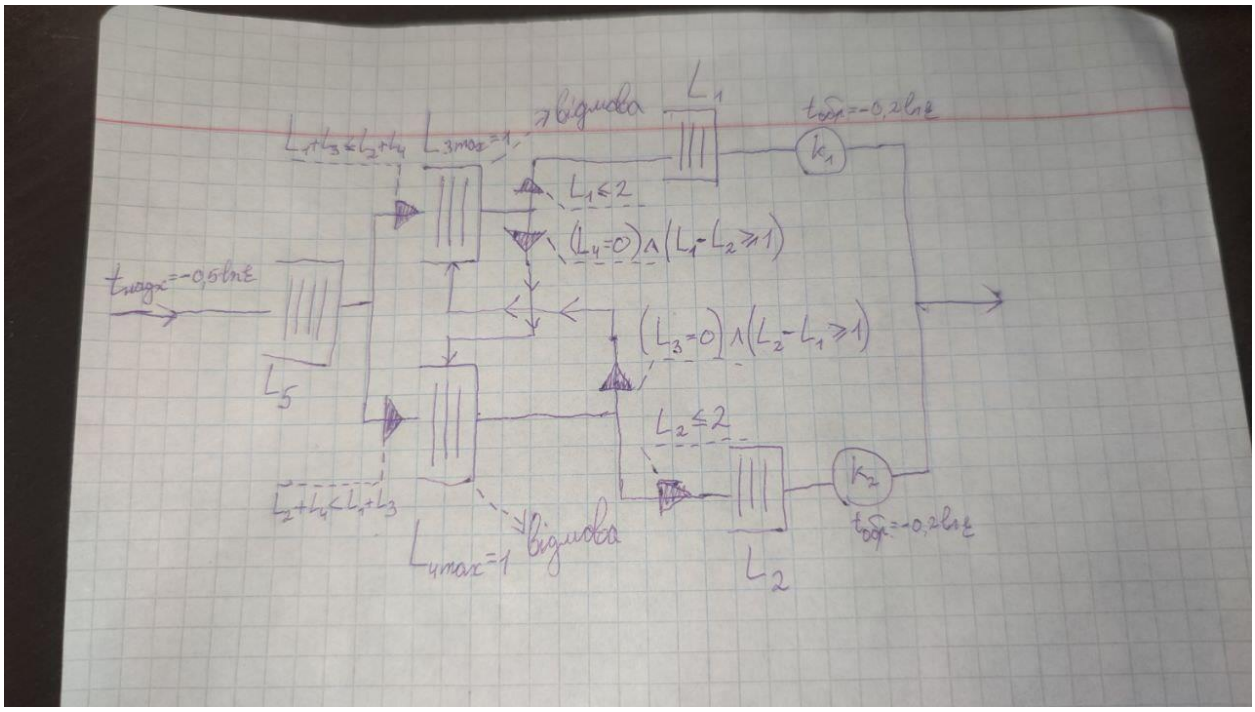
Величина	*Розподіл*
Час між прибуттями в приймальне відділення	Експоненціальний з математичним сподіванням 15
Час слідування в палату	Рівномірне від 3 до 8
Час слідування з приймального відділення в лабораторію або з лабораторії в приймальне відділення	Рівномірне від 2 до 5
Час обслуговування в реєстратуру лабораторії	Ерланга з математичним сподіванням 4,5 і k=3
Час проведення аналізу в лабораторії	Ерланга з математичним сподіванням 4 і k=2

Визначити час, проведений хворим у системі, тобто інтервал часу, починаючи з надходження і закінчуючи доставкою в палату (для хворих типу 1 і 2) або виходом із лабораторії (для хворих типу 3). Визначити також інтервал між прибуттями хворих у лабораторію.

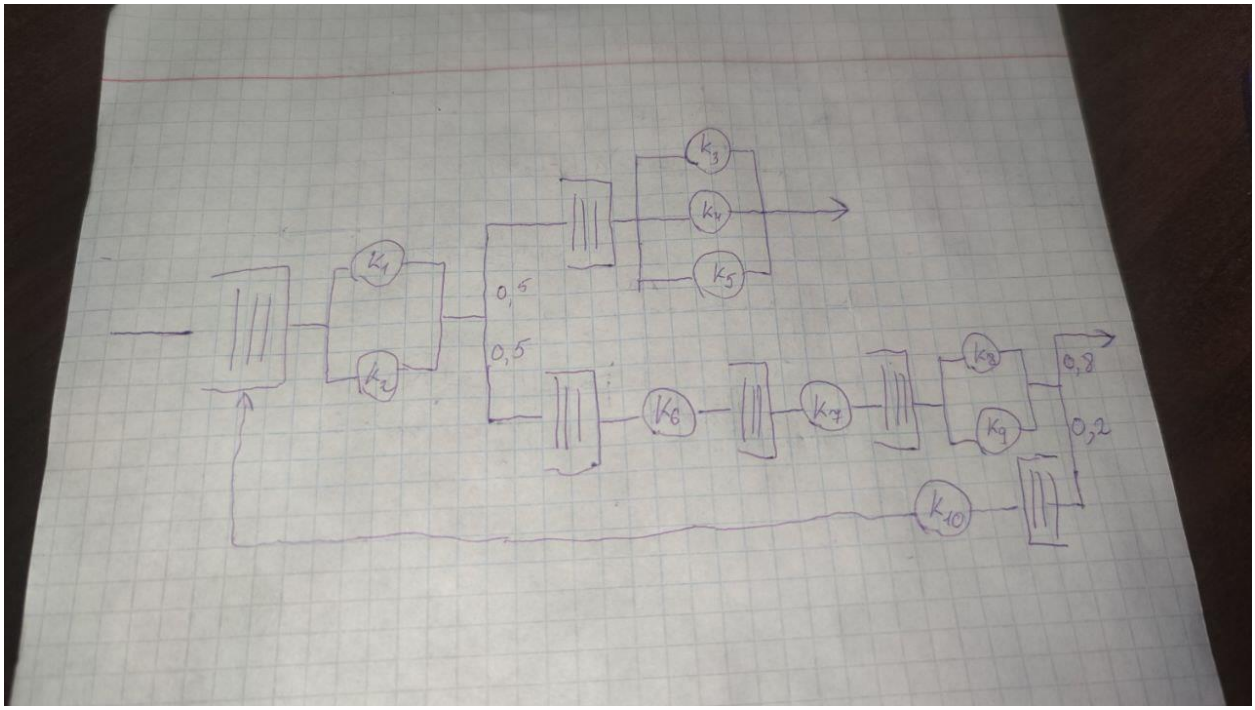
Хід роботи

Формалізовані моделі для виконання задвань:

1.



2.



Код для виконання завдання:

```
// index.ts
import BankClient from './BankClient';
import BlockingElement from './BlockingElement';
import Create from './Create';
```

```
import CustomRandom from './CustomRandom';
import Distribution from './Distribution';
import Model from './Model';
import Patient, { PatientType } from './Patient';
import PriorityQueue from './PriorityQueue';
import Process from './Process';
import Queue from './Queue';

const runTest = () => {
  const create = new Create(2);

  const p1 = new Process('p1', 1);
  const p2 = new Process('p2', 1);
  const p3 = new Process('p3', 1);

  const q1 = new Queue('q1', 2);
  const q2 = new Queue('q2', 2);
  const q3 = new Queue('q3', 3);

  const b1 = new BlockingElement('b1', (_obj, t) => t < 500);

  create.setDistribution(Distribution.EXPONENTIAL);
  p1.setDistribution(Distribution.EXPONENTIAL);
  p2.setDistribution(Distribution.EXPONENTIAL);
  p3.setDistribution(Distribution.EXPONENTIAL);

  create.setNextElements([q1]);
  q1.setNextElements([p1]);
  p1.setNextElements([q2]);
  q2.setNextElements([b1, q3]);
  b1.setNextElements([p2]);
  // p2.setNextElements([q3]);
  q3.setNextElements([p3]);

  const model = new Model([create, p1, p2, p3, q1, q2, q3, b1]);

  model.simulate(1000);
};

const runTask2Min = () => {
  const create = new Create(0.2);
  create.setDistribution(Distribution.EXPONENTIAL);

  const p1 = new Process('p1', 0.3);
  p1.setDistribution(Distribution.EXPONENTIAL);

  const p2 = new Process('p2', 0.3);
  p2.setDistribution(Distribution.EXPONENTIAL);

  const q1 = new Queue('q1', 3);
  const q2 = new Queue('q2', 3);
```

```
  const b1 = new BlockingElement('b1', () => q1.getLength() > q2.getLength());
  const b2 = new BlockingElement('b2', () => q2.getLength() >= q1.getLength());

  create.setNextElements([b1, b2]);
  b1.setNextElements([q1]);
  b2.setNextElements([q2]);
  q1.setNextElements([p1]);
  q2.setNextElements([p2]);

  const model = new Model([create, p1, p2, q1, q2, b1, b2]);
  model.simulate(1000);
};

const runTask2 = () => {
  const create = new Create(0.5, (t) => new BankClient(t));
  create.setDistribution(Distribution.EXPONENTIAL);

  const outputIntervals: number[] = [];
  const meanGeneralProcessingIntervals: number[] = [];
  let prevOutputT = 0;

  const p1 = new Process<BankClient>('p1', 0.3, (obj) => {
    if (prevOutputT) outputIntervals.push(p1.getCurrentT() - prevOutputT);
    prevOutputT = p1.getCurrentT();

    meanGeneralProcessingIntervals.push(p1.getCurrentT() - obj.bankEnterT);
    return obj;
  });
  p1.setDistribution(Distribution.EXPONENTIAL);

  const p2 = new Process<BankClient>('p2', 0.3, (obj) => {
    if (prevOutputT) outputIntervals.push(p2.getCurrentT() - prevOutputT);
    prevOutputT = p2.getCurrentT();

    meanGeneralProcessingIntervals.push(p2.getCurrentT() - obj.bankEnterT);

    return obj;
  });
  p2.setDistribution(Distribution.EXPONENTIAL);

  const qStart = new Queue<BankClient>('qStart');
  const q1 = new Queue<BankClient>('q1');
  const q2 = new Queue<BankClient>('q2');
  const q3 = new Queue<BankClient>('q3', 1);
  const q4 = new Queue<BankClient>('q4', 1);

  const b1 = new BlockingElement<BankClient>(
    'b1',
    () => q1.getLength() + q3.getLength() > q2.getLength() + q4.getLength()
  );
```

```
const b2 = new BlockingElement<BankClient>(
  'b2',
  () => q2.getLength() + q4.getLength() >= q1.getLength() + q3.getLength()
);
const b3 = new BlockingElement<BankClient>(
  'b3',
  () => !(q4.getLength() === 0 && q1.getLength() - q2.getLength() >= 1)
);
const b4 = new BlockingElement<BankClient>(
  'b4',
  () => !(q3.getLength() === 0 && q2.getLength() - q1.getLength() >= 1)
);

const b5 = new BlockingElement<BankClient>('b5', () => q1.getLength() === 2);
const b6 = new BlockingElement<BankClient>('b6', () => q2.getLength() === 2);

create.setNextElements([qStart]);
qStart.setNextElements([b1, b2]);

b1.setNextElements([q3]);
q3.setNextElements([b3, b5]);
b3.setNextElements([q4]);
b5.setNextElements([q1]);
q1.setNextElements([p1]);

b2.setNextElements([q4]);
q4.setNextElements([b4, b6]);
b4.setNextElements([q3]);
b6.setNextElements([q2]);
q2.setNextElements([p2]);

const t = 1000;

const model = new Model([
  create,
  p1,
  p2,
  qStart,
  q1,
  q2,
  q3,
  q4,
  b1,
  b2,
  b3,
  b4,
  b5,
  b6,
]);
model.simulate(t);
```

```
  console.log('\nTask results:');
  console.log(`cashier 1 mean work time: ${p1.getTotalWorkTime() / t}`);
  console.log(`cashier 2 mean work time: ${p2.getTotalWorkTime() / t}`);
  console.log(
    `mean clients in bank: ${
      (q1.getMeanQueueLength() +
        q2.getMeanQueueLength() +
        q3.getMeanQueueLength() +
        q4.getMeanQueueLength()) /
      t
    }`
  );
  console.log(
    `mean output interval: ${
      outputIntervals.reduce((acc, el) => acc + el, 0) / outputIntervals.length
    }`
  );
  console.log(
    `mean client time in bank: ${
      meanGeneralProcessingIntervals.reduce((acc, el) => acc + el, 0) /
      meanGeneralProcessingIntervals.length
    }`
  );
  console.log(
    `failure rate: ${
      q3.getFailuresCount() / (q3.getQuantity() + q3.getFailuresCount()) +
      q4.getFailuresCount() / (q4.getQuantity() + q4.getFailuresCount())
    }`
  );
  console.log(`line change count: ${b3.getQuantity() + b4.getQuantity()}`);
};

const runTask3 = () => {
  let lastLabInputT = 0;
  const labInputIntervals: number[] = [];
  const processingIntervals: number[] = [];

  const create = new Create(15, (t) => {
    const rand = Math.random();
    const type =
      rand < 0.5
        ? PatientType.ONE
        : rand < 0.6
        ? PatientType.TWO
        : PatientType.THREE;

    return new Patient(type, t);
  });
  create.setDistribution(Distribution.EXPONENTIAL);

  const doctorDelayFunc = (obj: Patient) =>
```

```
  obj.type === PatientType.ONE ? 15 : obj.type === PatientType.TWO ? 40 : 30;
const attendantDelayFunc = () => Math.random() * 5 + 3;
const labTransferDelayFunc = () => Math.random() * 3 + 2;
const labRegisterDelayFunc = () => CustomRandom.generateErlang(4.5, 3);
const labAssistantDelayFunc = () => CustomRandom.generateErlang(4, 2);

const attendantProcessingFunc = (obj: Patient, t: number) => {
  processingIntervals.push(t - obj.enterT);

  return obj;
};
const labAssistantProcessingFunc = (obj: Patient, t: number) => {
  if (obj.type === PatientType.THREE)
    processingIntervals.push(t - obj.enterT);

  return obj;
};

const doctor1 = new Process<Patient>('doctor1[p]', doctorDelayFunc);
const doctor2 = new Process<Patient>('doctor2[p]', doctorDelayFunc);

const attendant1 = new Process<Patient>(
  'attendant1[p]',
  attendantDelayFunc,
  attendantProcessingFunc
);
const attendant2 = new Process<Patient>(
  'attendant2[p]',
  attendantDelayFunc,
  attendantProcessingFunc
);
const attendant3 = new Process<Patient>(
  'attendant3[p]',
  attendantDelayFunc,
  attendantProcessingFunc
);

const transferToLabProcess = new Process<Patient>(
  'transfer to lab[p]',
  labTransferDelayFunc,
  (obj) => {
    if (lastLabInputT)
      labInputIntervals.push(
        transferToLabProcess.getCurrentT() - lastLabInputT
      );

    lastLabInputT = transferToLabProcess.getCurrentT();
    return obj;
  }
);
```

```
const labRegister = new Process<Patient>(
  'lab register[p]',
  labRegisterDelayFunc
);

const labAssistant1 = new Process<Patient>(
  'lab assistan1[p]',
  labAssistantDelayFunc,
  labAssistantProcessingFunc
);
const labAssistant2 = new Process<Patient>(
  'lab assistan2[p]',
  labAssistantDelayFunc,
  labAssistantProcessingFunc
);

const transferFromLabProcess = new Process<Patient>(
  'transfer from lab[p]',
  labTransferDelayFunc,
  (patient) => {
    patient.type = PatientType.ONE;
    return patient;
  }
);

const reception = new PriorityQueue<Patient>('reception[q]', (obj) =>
  obj.type === PatientType.ONE ? 1 : 0
);
const attendantWaiting = new Queue<Patient>('attendant waiting[q]');
const toLabTransfering = new Queue<Patient>('to lab transfering[q]');
const labReception = new Queue<Patient>('lab reception[q]');
const labWaitingRoom = new Queue<Patient>('lab waiting room[q]');
const fromLabTransfering = new Queue<Patient>('from lab transfering[q]');
const labExit = new Queue<Patient>('lab exit[q]');

const type1Block = new BlockingElement<Patient>(
  'type1Block',
  (obj) => obj.type !== PatientType.ONE
);

const type23Block = new BlockingElement<Patient>(
  'type23Block',
  (obj) => obj.type !== PatientType.TWO && obj.type !== PatientType.THREE
);

const type2Block = new BlockingElement<Patient>(
  'type2Block',
  (obj) => obj.type !== PatientType.TWO
);

const type13Block = new BlockingElement<Patient>(
```

```
  'type13Block',
  (obj) => obj.type !== PatientType.ONE && obj.type !== PatientType.THREE
);

create.setNextElements([reception]);
reception.setNextElements([doctor1, doctor2]);

doctor1.setNextElements([type1Block, type23Block]);
doctor2.setNextElements([type1Block, type23Block]);

type1Block.setNextElements([attendantWaiting]);
attendantWaiting.setNextElements([attendant1, attendant2, attendant3]);

type23Block.setNextElements([toLabTransfering]);
toLabTransfering.setNextElements([transferToLabProcess]);
transferToLabProcess.setNextElements([labReception]);
labReception.setNextElements([labRegister]);
labRegister.setNextElements([labWaitingRoom]);
labWaitingRoom.setNextElements([labAssistant1, labAssistant2]);
labAssistant1.setNextElements([type2Block, type13Block]);
labAssistant2.setNextElements([type2Block, type13Block]);
type13Block.setNextElements([labExit]);
type2Block.setNextElements([fromLabTransfering]);
fromLabTransfering.setNextElements([transferFromLabProcess]);
transferFromLabProcess.setNextElements([reception]);

const model = new Model([
  create,
  doctor1,
  doctor2,
  attendant1,
  attendant2,
  attendant3,
  transferToLabProcess,
  labRegister,
  labAssistant1,
  labAssistant2,
  transferFromLabProcess,
  reception,
  attendantWaiting,
  toLabTransfering,
  labReception,
  labWaitingRoom,
  fromLabTransfering,
  labExit,
  type1Block,
  type23Block,
  type2Block,
  type13Block,
]);
```

```
  model.simulate(10000);

  console.log('\nTask results:');
  console.log(
    `Mean processing time: ${
      processingIntervals.reduce((acc, el) => acc + el, 0) /
      processingIntervals.length
    }`
  );
  console.log(
    `Mean lab input interval: ${
      labInputIntervals.reduce((acc, el) => acc + el, 0) /
      labInputIntervals.length
    }`
  );
};

// runTest();
// runTask2();
runTask3();
```

```
// Element.ts
import CustomRandom from './CustomRandom';
import Distribution from './Distribution';
import ModelObject from './ModelObject';

export default abstract class Element<T extends ModelObject> {
  private static nextId = 0;

  protected id = Element.getNextId();
  private currentT = 0;
  protected nextT = 0;
  protected quantity = 0;
  protected nextElementsOptions:
    | { type: 'simple'; elements: Element<T>[] }
    | {
        type: 'possibilities';
        options: { element: Element<T>; probability: number }[];
      } = { type: 'simple', elements: [] };
  private delayFunction?: (obj: T) => number;

  constructor(
    protected name = '',
    private distribution: Distribution = Distribution.STATIC,
    private delayMean = 0,
    private delayVariance = 0
  ) {}

  public static getNextId() {
```

```
    return this.nextId++;
  }

  public getDelay(obj: T): number {
    if (this.delayFunction) return this.delayFunction(obj);

    switch (this.distribution) {
      case Distribution.NORMAL:
        return CustomRandom.generateNormal(
          Math.sqrt(this.delayVariance),
          this.delayMean
        );
      case Distribution.EXPONENTIAL:
        return CustomRandom.generateExponential(1 / this.delayMean);
      case Distribution.UNIFORM:
        return CustomRandom.generateUniform() * 2 * this.delayMean;
      case Distribution.STATIC:
      default:
        return this.delayMean;
    }
  }

  public inAction(obj: T) {}

  public outAction() {
    this.quantity++;
  }

  public setNextElements(elements: Element<T>[]) {
    this.nextElementsOptions = { type: 'simple', elements };
  }

  public setNextPossibleElements(
    options: { element: Element<T>; probability: number }[]
  ) {
    if (options.reduce((acc, el) => acc + el.probability, 0) !== 1)
      throw new Error('Sum of possibilities should equal 1');

    this.nextElementsOptions = { type: 'possibilities', options };
  }

  public getNextElement(obj: T) {
    if (this.nextElementsOptions.type === 'simple') {
      for (const element of this.nextElementsOptions.elements) {
        if (element.isReadyForIn(obj)) return element;
      }

      return;
    }

    const rand = Math.random();
```

```
  let sum = 0;

  for (const { element, probability } of this.nextElementsOptions.options) {
    sum += probability;

    if (rand < sum) return element;
  }

  return;
}

public getClosestNextElement(obj: T) {
  let closestElement: Element<T> | undefined = undefined;

  const elements = this.getNextElementsOptions();

  for (const element of elements) {
    if (
      element.isReadyForIn(obj) &&
      (!closestElement || element.nextT < closestElement.nextT)
    ) {
      closestElement = element;
    }
  }

  return closestElement;
}

public setCurrentT(t: number) {
  this.currentT = t;
}

public setNextT(t: number) {
  this.nextT = t;
}

public getCurrentT() {
  return this.currentT;
}

public getNextT() {
  return this.nextT;
}

public getQuantity() {
  return this.quantity;
}

public printResult() {
  console.log(`${this.name}[${this.id}] quantity = ${this.quantity}`);
```

```
  }

  public printInfo() {
    console.log(
      `${this.name}[${this.id}] quantity = ${this.quantity} tnext= ${this.nextT}`
    );
  }

  public setDistribution(distribution: Distribution) {
    this.distribution = distribution;
  }

  public setDelayFunction(func: (obj: T) => number) {
    this.delayFunction = func;
  }

  public getIdentifier() {
    return `${this.name}[${this.id}]`;
  }

  public doStatistics(delta: number) {}

  public isReadyForIn(obj: T) {
    return true;
  }

  public getNextElementsOptions() {
    return this.nextElementsOptions.type === 'simple'
      ? this.nextElementsOptions.elements
      : this.nextElementsOptions.options.map((option) => option.element);
  }
}
```

```
enum Distribution {
  EXPONENTIAL = 'exponential',
  NORMAL = 'normal',
  UNIFORM = 'uniform',
  STATIC = 'static',
}

export default Distribution;
```

```
import generateRandomOne from '../../lab1/src/server/generateRandomOne';
import generateRandomThree from '../../lab1/src/server/generateRandomThree';
import generateRandomTwo from '../../lab1/src/server/generateRandomTwo';

export default class CustomRandom {
```

```
  public static generateNormal(o: number, a: number) {
    return generateRandomTwo(o, a);
  }

  public static generateExponential(lambda: number) {
    return generateRandomOne(lambda);
  }

  public static generateUniform() {
    return generateRandomThree();
  }

  public static generateErlang(a: number, k: number) {
    let multEps = 1;

    for (let i = 0; i < k; i++) {
      multEps *= Math.random();
    }

    return -Math.log(multEps) / (k * a);
  }
}
```

```
import Distribution from './Distribution';
import Element from './Element';
import ModelObject from './ModelObject';

export default class Create<T extends ModelObject> extends Element<T> {
  constructor(
    delay: number,
    private readonly createObject = (t: number) => new ModelObject() as T
  ) {
    super('create', Distribution.STATIC, delay);
  }

  public inAction(): void {
    throw new Error('Tried to call inAction on Create');
  }

  public outAction() {
    const newObj = this.createObject(this.getCurrentT());

    super.outAction();
    super.setNextT(super.getCurrentT() + super.getDelay(newObj));
    super.getNextElement(newObj)?.inAction(newObj);
  }
}
```

```
import Element from './Element';
import ModelObject from './ModelObject';

export default class BlockingElement<T extends ModelObject> extends Element<T> {
  private currentObject?: T;

  constructor(
    name: string,
    public readonly isBlocked: (obj: T, t: number) => boolean
  ) {
    super(name);
    this.setNextT(Infinity);
  }

  public inAction(obj: T): void {
    if (this.isBlocked(obj, this.getCurrentT()))
      throw new Error('Tried to perform inAction on blocked element');

    this.currentObject = obj;
    this.setNextT(this.getCurrentT());
  }

  public outAction(): void {
    if (!this.currentObject)
      throw new Error('Tried to perform outAction on unbusy blocking element');

    super.outAction();

    const nextElement = this.getNextElement(this.currentObject);

    if (this.getNextElementsOptions().length > 0 && !nextElement)
      throw new Error(
        'Tried to perform outAction on blocking element without free next
element'
      );

    nextElement?.inAction(this.currentObject);
    this.currentObject = undefined;
    this.setNextT(Infinity);
  }

  public isReadyForIn(obj: T): boolean {
    const nextElement = this.getNextElement(obj);

    return (
      !this.currentObject &&
      !this.isBlocked(obj, this.getCurrentT()) &&
      !!nextElement &&
      nextElement.isReadyForIn(obj)
    );
  }
```

```
}
```

```
import ModelObject from './ModelObject';

export default class BankClient extends ModelObject {
  constructor(public readonly bankEnterT: number) {
    super();
  }
}
```

```
import Element from './Element';
import ModelObject from './ModelObject';
import Process from './Process';
import Queue from './Queue';

export default class Model<T extends ModelObject> {
  private elements: Element<T>[] = [];
  private currentT: number = 0;
  private nextT: number = 0;
  private event: number = 0;

  constructor(elements: Element<T>[]) {
    this.elements = elements;
  }

  public simulate(t: number) {
    while (this.currentT < t) {
      this.nextT = Infinity;
      for (let i = 0; i < this.elements.length; i++) {
        const element = this.elements[i];
        if (element.getNextT() < this.nextT) {
          this.nextT = element.getNextT();
          this.event = i;
        }
      }
      console.log(
        '\nIts time for event in ' +
          this.elements[this.event].getIdentifier() +
          ', time = ' +
          this.nextT
      );

      for (const element of this.elements) {
        element.doStatistics(this.nextT - this.currentT);
      }

      this.currentT = this.nextT;
```

```
      for (const element of this.elements) {
        element.setCurrentT(this.currentT);
      }
      for (const element of this.elements) {
        if (element.getNextT() === this.currentT) {
          element.outAction();
        }
      }
      this.printInfo();
    }
    this.printResult();
  }

  public printInfo() {
    for (const element of this.elements) {
      element.printInfo();
    }
  }

  public printResult() {
    console.log('\n-------------RESULTS-------------');
    for (const element of this.elements) {
      element.printResult();
      if (element instanceof Queue) {
        console.log(
          `mean length of queue = ${
            element.getMeanQueueLength() / this.currentT
          }\nfailure probability = ${
            element.getFailuresCount() /
            (element.getQuantity() + element.getFailuresCount())
          }\n`
        );
      }

      if (element instanceof Process) {
        console.log(
          `mean work time = ${element.getTotalWorkTime() / this.currentT}\n`
        );
      }
    }
  }
}
```

```
import ModelObject from './ModelObject';

export enum PatientType {
  ONE = 1,
  TWO = 2,
  THREE = 3,
```

```
}

export default class Patient extends ModelObject {
  constructor(
    public type: PatientType,
    public readonly enterT: number
  ) {
    super();
  }
}
```

```
import ModelObject from './ModelObject';
import Queue from './Queue';

export default class PriorityQueue<T extends ModelObject> extends Queue<T> {
  constructor(
    name: string,
    private readonly getPriority: (obj: T) => number,
    size?: number
  ) {
    super(name, size);
  }

  public enqueue(item: T): boolean {
    if (this.items.length === this.size) return false;

    const priority = this.getPriority(item);

    let i = this.items.length - 1;

    while (i >= 0 && priority > this.getPriority(this.items[i])) {
      i--;
    }

    this.items.splice(i + 1, 0, item);

    return true;
  }
}

import Distribution from './Distribution';
import Element from './Element';
import ModelObject from './ModelObject';

export default class Process<T extends ModelObject> extends Element<T> {
  private currentObject?: T;
  private totalWorkTime = 0;
  private prevWorkStart = 0;
```

```
  constructor(
    name: string,
    delayFunc: (obj: T) => number,
    modifyObj?: (obj: T, t: number) => T
  );
  constructor(name: string, delay: number, modifyObj?: (obj: T) => T);
  constructor(
    name: string,
    delay?: number | ((obj: T) => number),
    private readonly modifyObj: (obj: T, t: number) => T = (obj) => obj
  ) {
    const isDelayFunction = typeof delay === 'function';

    super(name, Distribution.STATIC, isDelayFunction ? 0 : delay);

    this.setNextT(Infinity);

    if (isDelayFunction) {
      this.setDelayFunction(delay);
    }
  }

  public inAction(obj: T): void {
    this.currentObject = obj;

    this.setNextT(this.getCurrentT() + this.getDelay(obj));
    this.prevWorkStart = this.getCurrentT();
  }

  public outAction(): void {
    super.outAction();

    if (!this.currentObject)
      throw new Error('Tried to perform outAction on unbusy process');

    const modifiedObject = this.modifyObj(
      this.currentObject,
      this.getCurrentT()
    );

    const nextElement = this.getNextElement(modifiedObject);

    if (this.getNextElementsOptions().length > 0 && !nextElement)
      throw new Error(
        'Tried to perform outAction on process without free next element'
      );

    nextElement?.inAction(modifiedObject);

    this.currentObject = undefined;
    this.setNextT(Infinity);
```

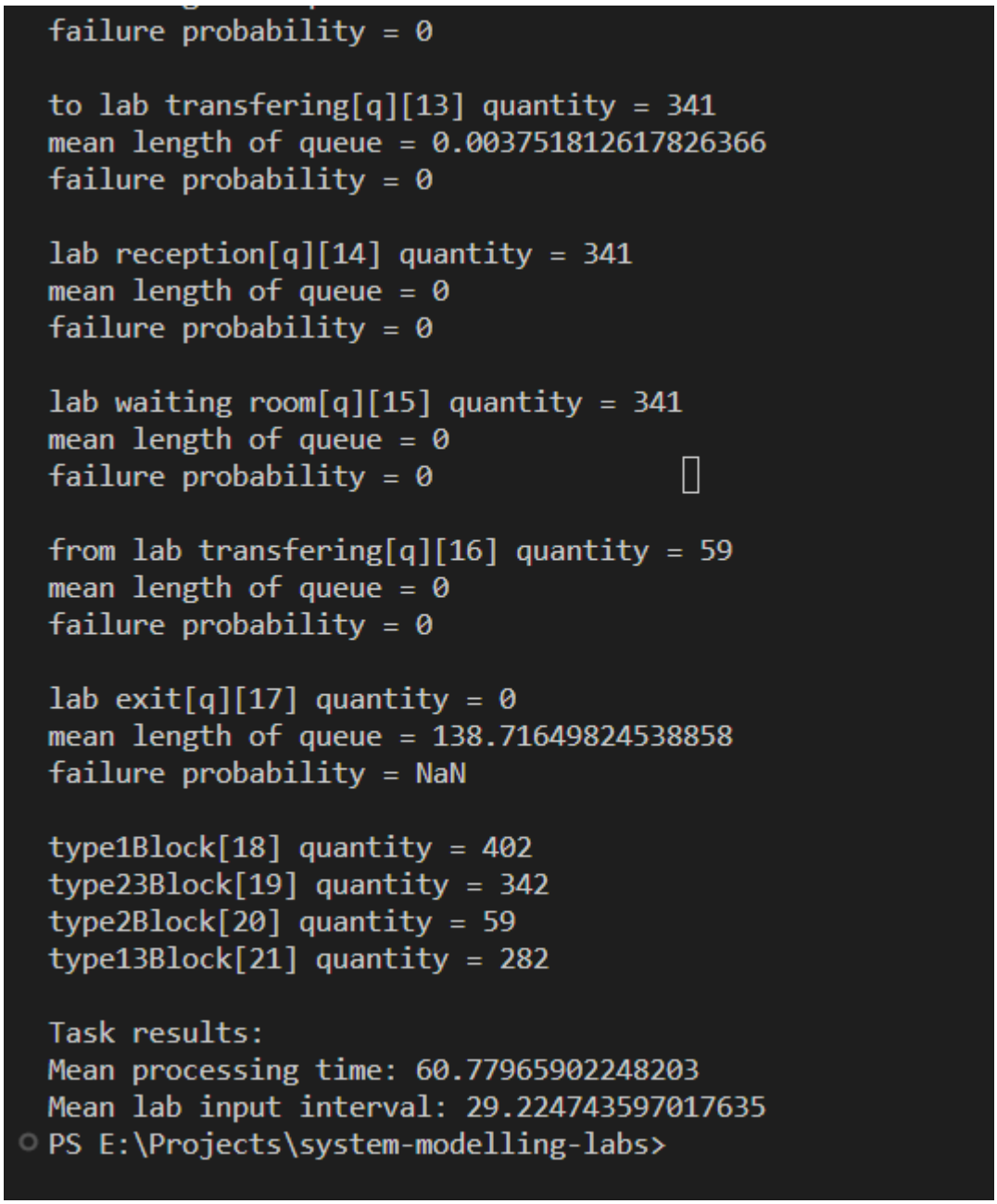
```
    this.totalWorkTime += this.getCurrentT() - this.prevWorkStart;
  }

  public isBusy() {
    return !!this.currentObject;
  }

  public getTotalWorkTime() {
    return this.totalWorkTime;
  }

  public isReadyForIn(): boolean {
    return !this.isBusy();
  }
}
```

```
import Element from './Element';
import ModelObject from './ModelObject';

export default class Queue<T extends ModelObject> extends Element<T> {
  protected items: T[] = [];
  private failuresCount = 0;
  private meanQueueLength = 0;

  constructor(
    name: string,
    public readonly size: number = Infinity
  ) {
    super(name);
    this.setNextT(Infinity);
  }

  public enqueue(item: T) {
    if (this.items.length === this.size) return false;

    this.items.push(item);

    return true;
  }

  public dequeue() {
    return this.items.shift();
  }

  public getLength() {
    return this.items.length;
  }

  public inAction(obj: T): void {
```

```
    if (!this.enqueue(obj)) this.failuresCount++;
  }

  public outAction(): void {
    const obj = this.dequeue();

    if (!obj) throw new Error('Tried to perfrom outAction on empty queue');

    const nextElement = this.getNextElement(obj);

    if (this.getNextElementsOptions().length > 0 && !nextElement)
      throw new Error(
        'Tried to perform outAction on queue without free next element'
      );

    super.outAction();

    nextElement?.inAction(obj);
  }

  public getNextT() {
    const closestNextElement = this.getClosestNextElement(this.items[0]);

    return this.items.length === 0 || !closestNextElement
      ? Infinity
      : closestNextElement.isReadyForIn(this.items[0])
      ? this.getCurrentT()
      : closestNextElement.getNextT();
  }

  public doStatistics(delta: number): void {
    this.meanQueueLength += this.items.length * delta;
  }

  public printInfo() {
    console.log(
      `${this.name}[${this.id}] quantity = ${this.quantity} tnext = ${this.nextT}
length = ${this.items.length}`
    );
  }

  public getFailuresCount() {
    return this.failuresCount;
  }

  public getMeanQueueLength() {
    return this.meanQueueLength;
  }
}
```

Результати роботи коду:

```
b1[8] quantity = 1652 tnext= Infinity
b2[9] quantity = 385 tnext= Infinity
b3[10] quantity = 0 tnext= Infinity
b4[11] quantity = 0 tnext= Infinity
b5[12] quantity = 1652 tnext= Infinity
b6[13] quantity = 385 tnext= Infinity

-------------RESULTS-------------
create[0] quantity = 2037
p1[1] quantity = 1650
mean work time = 0.4955066935961704

p2[2] quantity = 385
mean work time = 0.10373279301559829

qStart[3] quantity = 2037
mean length of queue = 0
failure probability = 0

q1[4] quantity = 1651
mean length of queue = 0.20045179404600036
failure probability = 0

q2[5] quantity = 385
mean length of queue = 0.021905002899131366
failure probability = 0

q3[6] quantity = 1652
mean length of queue = 0
failure probability = 0

q4[7] quantity = 385
mean length of queue = 0
failure probability = 0

b1[8] quantity = 1652
b2[9] quantity = 385
b3[10] quantity = 0
b4[11] quantity = 0
b5[12] quantity = 1652
b6[13] quantity = 385

Task results:
cashier 1 mean work time: 0.49562576538185754
cashier 2 mean work time: 0.10375772032547788
mean clients in bank: 0.2224102299687702
mean output interval: 0.49171384853832295
mean client time in bank: 0.40280895684725
failure rate: 0
line change count: 0
PS E:\Projects\system-modelling-labs>
```



Висновки: під час виконання цього завдання було побудовано декілька формалізованих моделей масового обслуговування та протестовано для вказаних задач та задля отримання певних метрик.