Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

# Комп’ютерний практикум №3

## Моделювання систем

**Тема:** Побудова імітаційної моделі системи з використанням формалізму моделі масового обслуговування

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірила:
Дифучина О. Ю.

Київ 2024

# ЗМІСТ

1 Мета комп’ютерного практикуму.....6

2 Завдання.....7

3 Виконання.....10

3.1 Побудова універсального алгоритму.....10

3.2 Друге завдання.....11

3.3 Третє завдання.....13

Висновок.....15

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ.....17

# 1 МЕТА КОМП’ЮТЕРНОГО ПРАКТИКУМУ

Розробити та застосувати універсальний алгоритм імітації моделі масового обслуговування для аналізу конкретних систем.

# 2 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 математичним сподіванням 15 |
| Час слідування в палату | Рівномірне від 3 до 8 |
| Час слідування з приймального відділення в лабораторію або з лабораторії у 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.

# 3 ВИКОНАННЯ

## 3.1 Побудова універсального алгоритму

Імітаційно модель можна розкласти на сукупність подій, що породжують одна одну. Усі події під час ітерації отримують посилання на загальний стан системи та модифікують її. Після кожної ітерації батьківська подія породжує дочірні, далі новоутворені події додаються у чергу з пріоритетом. Дана структура даних сортує події у порядку від найближчої події до найдальшої. Перед початком нової ітерації найбільш близька подія виходить із черги і потім ітеруються.

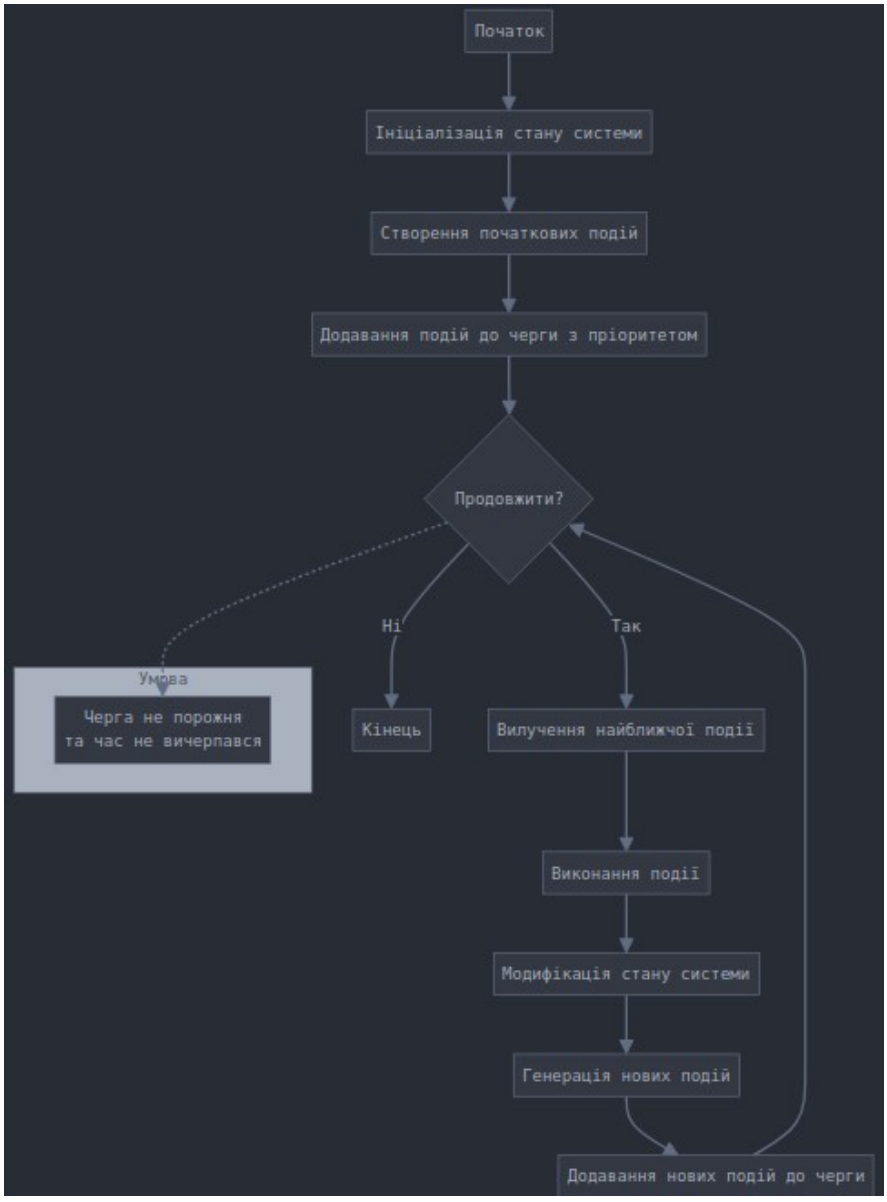



Рисунок 3.1 — Схема універсального алгоритму

**функція імітаційна_модель(максимальний_час)**:

стан_системи = ініціалізувати_стан()

черга_подій = створити_чергу_з_пріоритетом()

поточний_час = 0

початкові_події = створити_початкові_події()

для кожної події в початкові_події:

черга_подій.додати(подія)

поки не (черга_подій.порожня() або поточний_час >= максимальний_час):

поточна_подія = черга_подій.вилучити_найближчу()

поточний_час = поточна_подія.час

якщо поточний_час < максимальний_час:

виконати_подію(поточна_подія, стан_системи)

нові_події = поточна_подія.генерувати_нові_події(стан_системи)

для кожної нової_події в нові_події:

якщо нової_події.час < максимальний_час:

черга_подій.додати(нова_подія)

повернути стан_системи

**функція виконати_подію(подія, стан_системи):**

подія.виконати(стан_системи)

стан_системи.оновити()

## 3.2 Друге завдання

Побудуємо схему на рисунку 3.2.

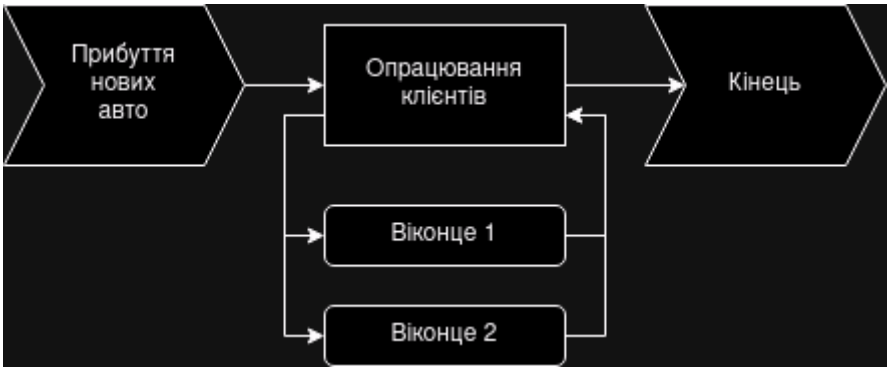



Рисунок 3.2 — Схема до другого завдання

Розглянемо результати симуляції:

```
1) utilization: 0.6328125
1) theoretical utilization: 0.6
2) average_clients_in_bank: BusyCashiersCount(1.668784943634212)
3) time_between_lefts: 3.083826545616705
4) mean_client_time: 3.933971573146133
5) first_mean_clients_in_queue: FloatQueueSize(0.8362441216177088)
5) second_mean_clients_in_queue: FloatQueueSize(1.0057010276060603)
6) refused_count: 0.07421875
7) balance_count: BalancedCount(50)
```

Коефіцієнт завантаження системи склав 65.78%, що дещо вище теоретичного значення 60%. Це свідчить про ефективне використання ресурсів банку. В середньому в банку перебуває 1.7121 клієнта, що вказує на помірне навантаження системи.

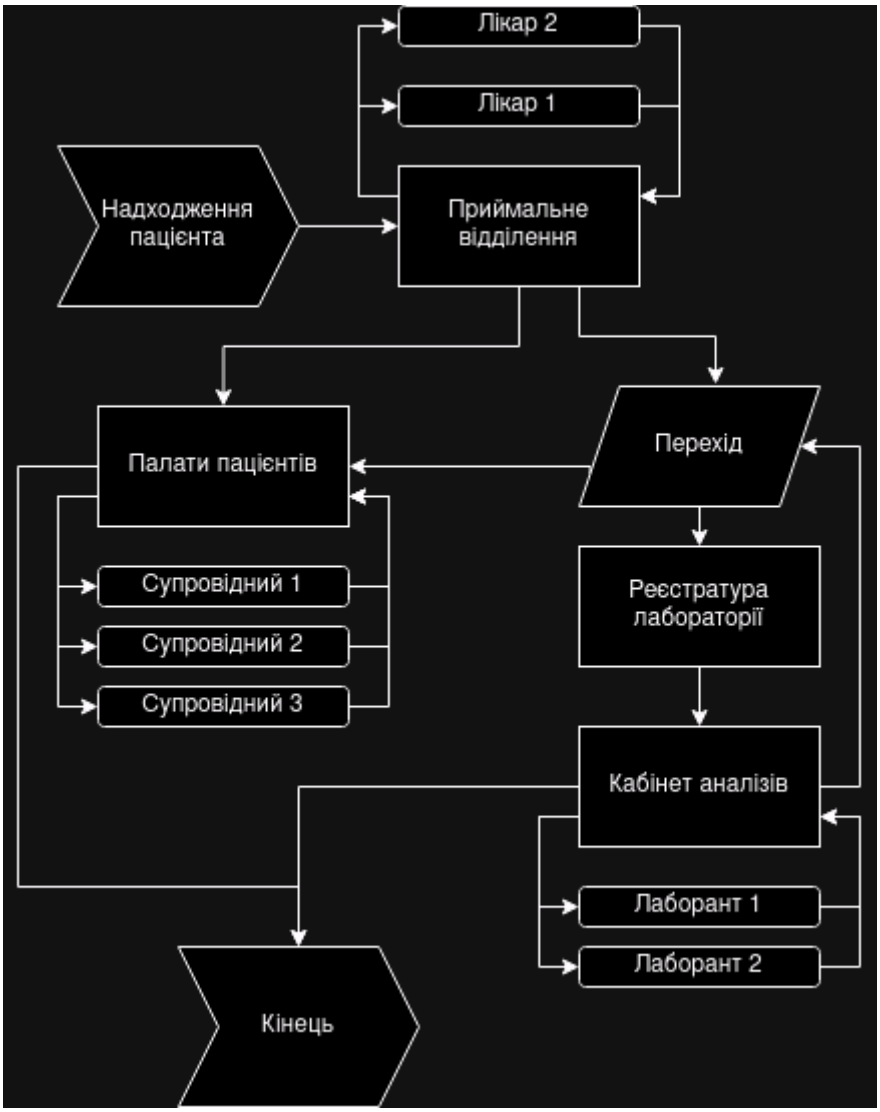
Середній час між виходами клієнтів становить 2.9179 одиниць часу, тоді як середній час перебування клієнта в банку - 4.1835 одиниць часу. Ця різниця вказує на наявність черг та час очікування обслуговування.

Середня довжина черги на першій касі становить 1.0103 клієнта, а на другій - 1.1955 клієнта. Ця невелика різниця свідчить про відносно рівномірний розподіл навантаження між касами.

Частка відмов в обслуговуванні складає 8.83%, що вказує на періодичне досягнення системою своєї максимальної місткості. Зафіксовано 55 випадків балансування черг, що демонструє активне використання цього механізму для оптимізації роботи системи.

3.3 Третє завдання

Розглянемо схему на рисунку 3.3:




Проаналізуємо результати:

```
Mean time: 56.800878605894425
Lab registration mean time: 30.636135570877414
```

Середній час перебування в системі: Середній час, який хворий проводить у системі, становить приблизно 56.8 хвилин (або 57 хвилин, якщо округлити). Це час від моменту надходження хворого до лікарні до моменту, коли він потрапляє в палату (для хворих типу 1 і 2) або виходить з лабораторії (для хворих типу 3).

Середній час між прибуттями хворих у лабораторію: Середній інтервал між прибуттями хворих у лабораторію становить приблизно 30.6 хвилин. Це значення відображає, як часто хворі прибувають до лабораторії. Варто зазначити, що цей час більший, ніж середній час між прибуттями в приймальне

відділення (який за умовою задачі становить 15 хвилин), що логічно, оскільки:

- не всі пацієнти направляються в лабораторію (хворі типу 1 йдуть одразу в палату);
- є затримки, пов'язані з проходженням приймального відділення та переміщенням до лабораторії.

# ВИСНОВОК

У ході лабораторної роботи було розроблено та застосовано універсальний алгоритм імітації моделі масового обслуговування. Цей алгоритм продемонстрував свою гнучкість та ефективність, успішно моделюючи різні системи - від банківського обслуговування автомобілістів до прийому пацієнтів у лікарні.

Результати симуляції банківської системи показали високу точність моделювання. Отримана завантаженість касирів (0.6328125) виявилася близькою до теоретично очікуваної (0.6), що підтверджує адекватність розробленої моделі. Система продемонструвала ефективну роботу з прийнятним середнім часом обслуговування клієнтів (3.93 одиниці часу) та відносно низьким відсотком відмов (7.42%).

Моделювання системи обслуговування в лікарні також надало цінні дані. Середній час перебування пацієнта в системі склав 56.8 хвилин, що є прийнятним показником, враховуючи складність процесу обслуговування. Інтервал між прибуттями в лабораторію (30.6 хвилин) виявився більшим за загальний інтервал прибуття, що свідчить про ефективний розподіл потоку пацієнтів між різними відділеннями.

Важливим аспектом роботи стало дослідження механізмів балансування навантаження. У банківській системі було зафіксовано 50 випадків зміни черги, що вказує на ефективність впровадженого механізму балансування та його вплив на загальну ефективність обслуговування.

Лабораторна робота також виявила потенціал для подальшої оптимізації обох систем. Хоча поточні показники є задовільними, існують можливості для зменшення часу очікування та підвищення пропускної здатності, що може бути предметом подальших досліджень та вдосконалень.

Загалом, проведене імітаційне моделювання надало глибоке розуміння

функціонування складних систем масового обслуговування. Отримані результати дозволяють не лише оцінити поточну ефективність систем, але й визначити напрямки для їх подальшого вдосконалення. Ця лабораторна робота підкреслила важливість та практичну цінність методів імітаційного моделювання в аналізі та оптимізації систем масового обслуговування різної складності та призначення.

# ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

*Тексти програмного коду*

(Найменування програми (документа))

*Жорсткий диск*

(Вид носія даних)

(Обсяг програми (документа), арк.)

*Студента групи ІП-11 4 курсу*

*Панченка С. В*

```
mod task_2;

use std::cell::RefCell;
use std::fmt::Debug;
use std::ops::{Div, Mul };
use std::rc::Rc;
use derive_more::{Sub, Add, AddAssign, SubAssign};
use rand_distr::Exp;
use scopeguard::defer;
use crate::delay_gen::DelayGen;

extern crate scopeguard;


#[derive(Debug, Copy, Clone, Default, PartialOrd, PartialEq)]
pub struct TimePoint(pub f64);

impl Sub for TimePoint {
    type Output = TimeSpan;
    fn sub(self, rhs: TimePoint) -> TimeSpan {
        TimeSpan(self.0 - rhs.0)
    }
}

impl Add<TimeSpan> for TimePoint {
    type Output = TimePoint;
    fn add(self, rhs: TimeSpan) -> TimePoint {
        TimePoint(self.0 + rhs.0)
    }
}

#[derive(Debug, Copy, Clone, Default, AddAssign)]
pub struct TimeSpan(f64);

pub mod delay_gen {
    use rand::distributions::Distribution;
    use crate::TimeSpan;
    use rand::thread_rng;
    use rand_distr::{Exp, Normal, Uniform};

    #[derive(Clone, Copy, Debug)]
    pub enum DelayGen {
        Normal(Normal<f64>),
        Uniform(Uniform<f64>),
```

```
            Exponential(Exp<f64>),
        }

        impl DelayGen {
            pub fn sample(&self) -> TimeSpan {
                TimeSpan(
                    match self {
                        Self::Normal(dist) => dist.sample(&mut thread_rng()),
                        Self::Uniform(dist) => dist.sample(&mut thread_rng()),
                        Self::Exponential(dist) => dist.sample(&mut
thread_rng()),
                    }
                )
            }
        }
    }

    #[derive(
        Debug, Copy, Clone, Default,
        Ord, PartialOrd, Eq, PartialEq,
        AddAssign, SubAssign, Sub
    )]
    struct QueueSize(u64);

    impl Mul<TimeSpan> for QueueSize {
        type Output = QueueTimeDur;

        fn mul(self, rhs: TimeSpan) -> Self::Output {
            QueueTimeDur(self.0 as f64 * rhs.0)
        }
    }

    #[derive(Debug, Copy, Clone, Default, Add, AddAssign)]
    struct QueueTimeDur(f64);

    #[derive(Debug)]
    struct FloatQueueSize(f64);

    impl Div<TimeSpan> for QueueTimeDur {
        type Output = FloatQueueSize;

        fn div(self, rhs: TimeSpan) -> Self::Output {
            FloatQueueSize(self.0 / rhs.0)
        }
```

```
}

impl Add<TimeSpan> for TimeSpan {
    type Output = TimeSpan;

    fn add(self, rhs: Self) -> Self::Output {
        Self(self.0 + rhs.0)
    }
}

#[derive(Debug, Default)]
struct Cashier {
    queue_size: QueueSize,
    is_busy: CashierBusy,

    processed_clients: ClientsCount,
    work_time: TimeSpan,

    queue_time_span: QueueTimeDur,
}

#[derive(Debug, Clone, Copy)]
enum CashierIndex {
    First,
    Second,
}

#[derive(Debug, Clone, Copy, Default, AddAssign)]
struct BusyCashiersCount(f64);

impl Mul<TimeSpan> for BusyCashiersCount {
    type Output = BusyCashiersCountTimeSpan;
    fn mul(self, rhs: TimeSpan) -> Self::Output {
        BusyCashiersCountTimeSpan(self.0 * rhs.0 as f64)
    }
}

#[derive(Debug, Clone, Copy, Default, AddAssign)]
struct BusyCashiersCountTimeSpan(f64);

impl Div<TimeSpan> for BusyCashiersCountTimeSpan {
    type Output = BusyCashiersCount;
    fn div(self, rhs: TimeSpan) -> Self::Output {
        BusyCashiersCount(self.0 / rhs.0 as f64)
```

```
    }
}

#[derive(Debug, Clone, Copy, Default, AddAssign)]
struct ClientsCountTimeSpan(f64);

impl Mul<TimeSpan> for ClientsCount {
    type Output = ClientsCountTimeSpan;

    fn mul(self, rhs: TimeSpan) -> Self::Output {
        ClientsCountTimeSpan(self.0 as f64 * rhs.0)
    }
}

struct FloatClientsCount(f64);

impl Div<TimeSpan> for ClientsCountTimeSpan {
    type Output = FloatClientsCount;
    fn div(self, rhs: TimeSpan) -> Self::Output {
        FloatClientsCount(self.0 / rhs.0)
    }
}

#[derive(Debug, Default)]
struct Bank {
    cashiers: [Cashier; 2],
    balance_count: BalancedCount,
    refused_count: RefusedCount,
    clients_count: ClientsCount,

    create_event_time_span: ClientsCountTimeSpan,
    last_create_event_time: TimePoint,

    busy_cashiers_count_time_span: BusyCashiersCountTimeSpan,
    event_delays: TimeSpan,
    last_event_time: TimePoint,

    last_processed_client_span: TimeSpan,
    last_processed_client_time: TimePoint,
}

impl Bank {

    fn update_on_event_end(&mut self, current_t: TimePoint) {
```

```
            let delay = current_t - self.last_event_time;
            self.event_delays += delay;
            let mut busy_cashiers_count = BusyCashiersCount::default();
            for c in &mut self.cashiers.iter_mut() {
                c.queue_time_span += c.queue_size * delay;
                match c.is_busy {
                    CashierBusy::Busy => busy_cashiers_count +=
BusyCashiersCount(1.0),
                    CashierBusy::NotBusy => (),
                }
            }
            self.busy_cashiers_count_time_span += busy_cashiers_count * delay;
            self.last_event_time = current_t;
        }

        fn get_cashier_mut(&mut self, index: CashierIndex) -> &mut Cashier {
            match index {
                CashierIndex::First => &mut self.cashiers[0],
                CashierIndex::Second => &mut self.cashiers[1],
            }
        }

        fn get_cashier(&self, index: CashierIndex) -> &Cashier {
            match index {
                CashierIndex::First => &self.cashiers[0],
                CashierIndex::Second => &self.cashiers[1],
            }
        }
    }

    #[derive(Debug, Copy, Clone, Default, AddAssign)]
    struct RefusedCount(usize);

    #[derive(Debug, Copy, Clone, Default, AddAssign)]
    struct BalancedCount(usize);

    const QUEUE_CHANGE_SIZE: QueueSize = QueueSize(2);
    const QUEUE_MAX_SIZE: QueueSize = QueueSize(3);

    fn balance_queues(
        mut first_queue_size: QueueSize,
        mut second_queue_size: QueueSize,
    ) -> (QueueSize, QueueSize, BalancedCount) {
        let (mmin, mmax) = if first_queue_size < second_queue_size {
```

```
        (&mut first_queue_size, &mut second_queue_size)
    } else {
        (&mut second_queue_size, &mut first_queue_size)
    };
    let mut rebalanced_count = BalancedCount(0);
    while *mmax - *mmin >= QUEUE_CHANGE_SIZE {
        *mmin += QueueSize(1);
        *mmax -= QueueSize(1);
        rebalanced_count += BalancedCount(1);
    }
    (first_queue_size, second_queue_size, rebalanced_count)
}

#[derive(Debug, Clone, Copy, AddAssign, Add, Default)]
struct ClientsCount(u64);

#[derive(Debug, Clone, Copy, Default)]
enum CashierBusy {
    #[default]
    NotBusy,
    Busy,
}

#[derive(Debug)]
struct EventCreate {
    current_t: TimePoint,
    create_delay_gen: DelayGen,
    process_delay_gen: DelayGen,
    bank: Rc<RefCell<Bank>>,
}

#[derive(Debug)]
struct EventProcess {
    delay_gen: DelayGen,
    work_time: TimeSpan,
    current_t: TimePoint,
    bank: Rc<RefCell<Bank>>,
    cashier_index: CashierIndex
}

impl EventCreate {
    fn iterate(self) -> (EventCreate, Option<EventProcess>) {
        defer! {
            let mut bank = self.bank.borrow_mut();
```

```
                bank.update_on_event_end(self.current_t);

                let create_delay = self.current_t -
bank.last_create_event_time;
                bank.create_event_time_span += ClientsCount(1) * create_delay;
                bank.last_create_event_time = self.current_t;
            }
            self.bank.borrow_mut().clients_count += ClientsCount(1);
            let work_time = self.process_delay_gen.sample();
            let new_current_t = self.current_t + work_time;
            let fist_busy =
self.bank.borrow().get_cashier(CashierIndex::First).is_busy;
            let second_busy =
self.bank.borrow().get_cashier(CashierIndex::Second).is_busy;
            (
                EventCreate {
                    current_t: self.current_t +
self.create_delay_gen.sample(),
                    create_delay_gen: self.create_delay_gen,
                    process_delay_gen: self.process_delay_gen,
                    bank: self.bank.clone()
                },
                match (fist_busy, second_busy) {
                    (CashierBusy::NotBusy, CashierBusy::NotBusy) => {
                        let mut bank = self.bank.borrow_mut();
                        assert_eq!
(bank.get_cashier(CashierIndex::First).queue_size, QueueSize(0));
                        assert_eq!
(bank.get_cashier(CashierIndex::Second).queue_size, QueueSize(0));
                        bank.get_cashier_mut(CashierIndex::First).is_busy =
CashierBusy::Busy;
                        Some(EventProcess {
                            delay_gen: self.process_delay_gen,
                            work_time,
                            current_t: new_current_t,
                            bank: self.bank.clone(),
                            cashier_index: CashierIndex::First
                        })
                    },
                    (CashierBusy::Busy, CashierBusy::NotBusy) => {
                        let mut bank = self.bank.borrow_mut();
                        assert_eq!
(bank.get_cashier(CashierIndex::Second).queue_size, QueueSize(0));
                        bank.get_cashier_mut(CashierIndex::Second).is_busy =
```

```
CashierBusy::Busy;
                        Some(EventProcess {
                            delay_gen: self.process_delay_gen,
                            work_time,
                            current_t: new_current_t,
                            bank: self.bank.clone(),
                            cashier_index: CashierIndex::Second
                        })
                    },
                    (CashierBusy::NotBusy, CashierBusy::Busy) => {
                        let mut bank = self.bank.borrow_mut();
                        assert_eq!
(bank.get_cashier(CashierIndex::First).queue_size, QueueSize(0));
                        bank.get_cashier_mut(CashierIndex::First).is_busy =
CashierBusy::Busy;
                        Some(EventProcess {
                            delay_gen: self.process_delay_gen,
                            work_time,
                            current_t: new_current_t,
                            bank: self.bank.clone(),
                            cashier_index: CashierIndex::First
                        })
                    }
                    (CashierBusy::Busy, CashierBusy::Busy) => {
                        let cashier_index = {
                            let bank = self.bank.borrow_mut();
                            let first_queue_size =
bank.get_cashier(CashierIndex::First).queue_size;
                            let second_queue_size =
bank.get_cashier(CashierIndex::Second).queue_size;
                            if first_queue_size < second_queue_size {
                                CashierIndex::First
                            } else {
                                CashierIndex::Second
                            }
                        };
                        if
self.bank.borrow().get_cashier(cashier_index).queue_size >= QUEUE_MAX_SIZE {
                            self.bank.borrow_mut().refused_count +=
RefusedCount(1);
                        } else {
                            let mut bank = self.bank.borrow_mut();
                            bank.get_cashier_mut(cashier_index).queue_size +=
QueueSize(1);
```

```
                            let res = balance_queues(
bank.get_cashier(CashierIndex::First).queue_size,
bank.get_cashier(CashierIndex::Second).queue_size,
                            );
bank.get_cashier_mut(CashierIndex::First).queue_size = res.0;
bank.get_cashier_mut(CashierIndex::Second).queue_size = res.1;
                            bank.balance_count += res.2;
                        }
                        None
                    }
                }
            )
        }
    }

    impl EventProcess {
        fn iterate(self) -> Option<EventProcess> {
            defer! {
                self.bank.borrow_mut().update_on_event_end(self.current_t);
            }

            let mut bank = self.bank.borrow_mut();
            let queue_size = bank.get_cashier(self.cashier_index).queue_size;
            bank.get_cashier_mut(self.cashier_index).is_busy =
CashierBusy::NotBusy;
            if queue_size > QueueSize(0) {
                bank.get_cashier_mut(self.cashier_index).queue_size -=
QueueSize(1);

                let res = balance_queues(
                    bank.get_cashier(CashierIndex::First).queue_size,
                    bank.get_cashier(CashierIndex::Second).queue_size,
                );
                bank.get_cashier_mut(CashierIndex::First).queue_size = res.0;
                bank.get_cashier_mut(CashierIndex::Second).queue_size = res.1;
                bank.balance_count += res.2;

                {
                    let cashier = bank.get_cashier_mut(self.cashier_index);
                    cashier.is_busy = CashierBusy::Busy;
```

```
                    cashier.work_time += self.work_time;
                    cashier.processed_clients += ClientsCount(1);
                }

                let time_between = bank.last_processed_client_time;
                bank.last_processed_client_span += self.current_t -
time_between;
                bank.last_processed_client_time = self.current_t;

                let work_time = self.delay_gen.sample();
                Some(EventProcess {
                    delay_gen: self.delay_gen,
                    work_time,
                    current_t: self.current_t + work_time,
                    bank: self.bank.clone(),
                    cashier_index: self.cashier_index
                })
            } else {
                None
            }
        }
    }

    #[derive(Debug)]
    enum Event {
        EventProcess(EventProcess),
        EventCreate(EventCreate),
    }

    impl Event {
        fn get_current_t(&self) -> TimePoint {
            match self {
                Event::EventProcess(e) => e.current_t,
                Event::EventCreate(e) => e.current_t,
            }
        }
    }

    fn main() {
        const CREATE_MEAN: f64 = 0.5;
        const PROCESS_MEAN: f64 = 0.3;


        let start_time = TimePoint::default();
```

```
        let end_time = TimePoint(1000.0);
        let bank: Rc<RefCell<Bank>> = Default::default();
        let mut nodes = vec![
            Event::EventCreate(EventCreate {
                current_t: start_time,
                create_delay_gen: DelayGen::Exponential(
                    Exp::new(CREATE_MEAN).expect("Could not create delay gen")
                ),
                process_delay_gen: DelayGen::Exponential(
                    Exp::new(PROCESS_MEAN).expect("Could not create delay
gen")
                ),
                bank,
            })
        ];


        let last_event = loop {
            nodes.sort_by(|a, b| {
                b.get_current_t().partial_cmp(&a.get_current_t())
                    .expect("Can not compare events current_t")
            });
            let next_event = nodes.pop().unwrap();
            if next_event.get_current_t() > end_time {
                break next_event;
            }
            match next_event {
                Event::EventCreate(event) => {
                    let (event_create, event_process) = event.iterate();
                    nodes.push(Event::EventCreate(event_create));
                    if let Some(event_process) = event_process {
                        nodes.push(Event::EventProcess(event_process));
                    }
                },
                Event::EventProcess(event) => {
                    if let Some(event_process) = event.iterate() {
                        nodes.push(Event::EventProcess(event_process));
                    }
                }
            }
        };

        let bank = match last_event {
            Event::EventCreate(event) => event.bank,
```

```
            Event::EventProcess(event) => event.bank
        };
        let bank = bank.borrow();
        let cashier_first = bank.get_cashier(CashierIndex::First);
        let cashier_second = bank.get_cashier(CashierIndex::Second);
        let total_processed_clients = cashier_first.processed_clients +
cashier_second.processed_clients;

        let input_rate = bank.clients_count.0 as f64 / bank.event_delays.0 as
f64;

        let time_between_lefts = bank.last_processed_client_span.0 as f64 /
total_processed_clients.0 as f64;
        let output_rate = 1.0 / time_between_lefts;

        let utilization = output_rate / input_rate;
        println!("1) utilization: {:?}", utilization);
        println!("1) theoretical utilization: {}", PROCESS_MEAN /
CREATE_MEAN);

        println!("2) average_clients_in_bank: {:?}",
bank.busy_cashiers_count_time_span / bank.event_delays);
        println!("3) time_between_lefts: {:?}", time_between_lefts);
        println!(
            "4) mean_client_time: {:?}",
            (cashier_first.work_time + cashier_second.work_time).0 as f64
            / total_processed_clients.0 as f64
        );
        println!("5) first_mean_clients_in_queue: {:?}",
cashier_first.queue_time_span / bank.event_delays);
        println!("5) second_mean_clients_in_queue: {:?}",
cashier_second.queue_time_span / bank.event_delays);
        println!("6) refused_count: {:?}", bank.refused_count.0 as f64 /
bank.clients_count.0 as f64);
        println!("7) balance_count: {:?}", bank.balance_count);
    }
use std::cmp::Ordering;
    use std::collections::{BinaryHeap, VecDeque};
    use rand::{thread_rng, RngCore};
    use rand::distributions::{Distribution, Uniform};
    use crate::task_2::create_patient::EventNewPatient;
    use crate::task_2::event_terminal::EventTerminal;
    use crate::task_2::patient::{Patient, PatientType};
    use crate::task_2::queue_resource::{Queue, QueueProcessor, QueueResource};
```

```
    use crate::task_2::transition_lab_reception::{
        EventTransitionFromLabToPatientWards,
        EventTransitionFromReceptionToLaboratory,
        EventTransition
    };
    use crate::{TimePoint, TimeSpan};

    mod patient {
        use std::cmp::Ordering;
        use crate::{TimePoint};

        #[derive(Debug, Clone, Copy, Ord, PartialOrd, Eq, PartialEq, Default)]
        pub enum PatientType {
            #[default]
            Three,
            Two,
            One,
        }

        #[derive(Debug, Default, Clone, Copy)]
        pub struct Patient {
            clinic_in_t: TimePoint,
            group: PatientType,
        }

        impl Patient {
            pub fn new(clinic_in_t: TimePoint, group: PatientType) -> Patient
{
                Patient { clinic_in_t, group }
            }

            pub fn upgrade_to_first_group(mut self) -> Patient {
                self.group = PatientType::One;
                self
            }

            pub fn get_group(&self) -> PatientType {
                self.group
            }

            pub fn get_clinic_in_t(&self) -> TimePoint {
                self.clinic_in_t
            }
        }
```

```
impl Eq for Patient {}

impl PartialEq<Self> for Patient {
    fn eq(&self, other: &Self) -> bool {
        self.group.eq(&other.group) &&
self.clinic_in_t.eq(&other.clinic_in_t)
    }
}

impl PartialOrd<Self> for Patient {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        match self.group.cmp(&other.group) {
            Ordering::Equal =>
other.clinic_in_t.partial_cmp(&self.clinic_in_t),
            other => Some(other),
        }
    }
}

impl Ord for Patient {
    fn cmp(&self, other: &Self) -> Ordering {
        self.partial_cmp(&other).expect("Patient ordering went wrong")
    }
}

#[cfg(test)]
mod tests {
    use std::collections::BinaryHeap;
    use crate::task_2::patient::{Patient, PatientType};
    use crate::TimePoint;

    #[test]
    fn test_patient_type_ordering() {
        let mut bh = BinaryHeap::new();
        bh.push(PatientType::Three);
        bh.push(PatientType::Two);
        bh.push(PatientType::One);

        assert_eq!(bh.pop(), Some(PatientType::One));
        assert_eq!(bh.pop(), Some(PatientType::Two));
        assert_eq!(bh.pop(), Some(PatientType::Three));
    }
```

```
            #[test]
            fn test_patient_ordering() {
                let mut bh = BinaryHeap::new();

                bh.push(Patient::new(TimePoint(100.0), PatientType::Three));
                bh.push(Patient::new(TimePoint(50.0), PatientType::Three));
                bh.push(Patient::new(TimePoint(25.0), PatientType::Three));

                bh.push(Patient::new(TimePoint(900.0), PatientType::Two));
                bh.push(Patient::new(TimePoint(800.0), PatientType::Two));
                bh.push(Patient::new(TimePoint(700.0), PatientType::Two));

                bh.push(Patient::new(TimePoint(2299.0), PatientType::One));
                bh.push(Patient::new(TimePoint(999.0), PatientType::One));
                bh.push(Patient::new(TimePoint(1999.0), PatientType::One));

                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(999.0),
PatientType::One)));
                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(1999.0),
PatientType::One)));
                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(2299.0),
PatientType::One)));

                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(700.0),
PatientType::Two)));
                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(800.0),
PatientType::Two)));
                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(900.0),
PatientType::Two)));

                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(25.0),
PatientType::Three)));
                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(50.0),
PatientType::Three)));
                assert_eq!(bh.pop(), Some(Patient::new(TimePoint(100.0),
PatientType::Three)));
            }
        }
    }

    mod queue_resource {
        use std::cell::{Cell};
        use std::rc::{Rc, Weak};
```

```
pub trait Queue {
    type Item;
    fn push(&mut self, t: Self::Item);
    fn pop(&mut self) -> Option<Self::Item>;
    fn is_empty(&self) -> bool;
}

#[derive(Debug, Default)]
pub struct QueueResource<Q> {
    max_acquires: usize,
    acquires_count: Rc<Cell<usize>>,
    queue: Q,
}

pub struct QueueProcessor<E> {
    acquires_count: Weak<Cell<usize>>,
    value: E
}

impl<E> Drop for QueueProcessor<E> {
    fn drop(&mut self) {
        let acquires_count =
self.acquires_count.upgrade().expect("Queue resource does not exist");
        acquires_count.set(acquires_count.get() - 1);
    }
}

impl<E> QueueProcessor<E> {
    pub fn value(&self) -> &E {
        &self.value
    }
}

impl<Q: Queue> QueueResource<Q> {
    pub fn new(queue: Q, max_acquires: usize) -> Self {
        Self{max_acquires, acquires_count: Rc::new(Cell::new(0usize)),
queue}
    }

    pub fn push(&mut self, t: Q::Item) {
        self.queue.push(t)
    }

    pub fn is_empty(&self) -> bool {
```

```
                self.queue.is_empty()
            }

            pub fn is_any_free_processor(&self) -> bool {
                self.acquires_count.get() < self.max_acquires
            }

            pub fn acquire_processor(&mut self) -> QueueProcessor<Q::Item> {
                assert!(self.acquires_count.get() < self.max_acquires);
                let value = self.queue.pop().expect("Queue is empty");
                self.acquires_count.set(self.acquires_count.get() + 1);
                QueueProcessor{acquires_count:
Rc::downgrade(&self.acquires_count), value}
            }
        }

        #[cfg(test)]
        mod tests {
            use crate::task_2::queue_resource::{Queue, QueueResource};

            #[derive(Default)]
            struct DummyQueue {
                len: usize
            }

            impl Queue for DummyQueue {
                type Item = ();
                fn push(&mut self, _: ()) {
                    self.len += 1;
                }

                fn pop(&mut self) -> Option<()> {
                    assert_eq!(self.is_empty(), false);
                    self.len -= 1;
                    Some(())
                }

                fn is_empty(&self) -> bool {
                    self.len == 0
                }
            }

            #[test]
            fn test_one() {
```

```
                let mut res = QueueResource::new(DummyQueue::default(),
3usize);
                res.push(());
                res.push(());
                res.push(());
                res.push(());
                let _proc_one = res.acquire_processor();
                let _proc_two = res.acquire_processor();
                let _proc_three = res.acquire_processor();
            }

            #[test]
            #[should_panic]
            fn test_two() {
                let mut res = QueueResource::new(DummyQueue::default(),
2usize);
                res.push(());
                res.push(());
                res.push(());
                res.push(());

                let _proc_one = res.acquire_processor();
                let _proc_two = res.acquire_processor();
                let _proc_three = res.acquire_processor();
            }

            #[test]
            fn test_three() {
                let mut res = QueueResource::new(DummyQueue::default(),
2usize);
                res.push(());
                res.push(());
                res.push(());
                res.push(());

                let _proc_one = res.acquire_processor();
                let _proc_two = res.acquire_processor();
                drop(_proc_two);
                let _proc_three = res.acquire_processor();
            }

        }
    }
```

```
impl<T: Ord> Queue for BinaryHeap<T> {
    type Item = T;

    fn push(&mut self, t: Self::Item) {
        BinaryHeap::push(self, t)
    }

    fn pop(&mut self) -> Option<Self::Item> {
        BinaryHeap::pop(self)
    }

    fn is_empty(&self) -> bool {
        BinaryHeap::is_empty(self)
    }
}

impl<T> Queue for VecDeque<T> {
    type Item = T;

    fn push(&mut self, t: Self::Item) {
        self.push_back(t);
    }

    fn pop(&mut self) -> Option<Self::Item> {
        self.pop_front()
    }

    fn is_empty(&self) -> bool {
        VecDeque::is_empty(self)
    }
}

#[derive(Debug)]
struct ReceptionDepartment(QueueResource<BinaryHeap<Patient>>);

#[derive(Debug)]
struct PatientWards(QueueResource<VecDeque<Patient>>);

#[derive(Debug)]
struct LabRegistry(QueueResource<VecDeque<Patient>>);

#[derive(Debug)]
struct Laboratory(QueueResource<VecDeque<Patient>>);
```

```
      mod create_patient {
          use lazy_static::lazy_static;
          use rand::distributions::{Distribution, Uniform};
          use rand_distr::Exp;
          use crate::task_2::{EventReceptionDepartment, Patient,
ReceptionDepartment};
          use crate::{TimePoint, TimeSpan};
          use crate::task_2::patient::PatientType;

          #[derive(Debug, Default)]
          pub struct EventNewPatient {
              current_t: TimePoint,
              patient: Patient,
          }

          lazy_static! {
              static ref DELAY_GEN: Exp<f64> = Exp::new(1.0 /
15.0).expect("Failed to create delay gen");
          }

          fn generate_patient_type() -> PatientType {
              let value = Uniform::new(0.0, 1.0).sample(&mut
rand::thread_rng());
              match value {
                  ..0.5 => PatientType::One,
                  0.5..0.6 => PatientType::Two,
                  0.6.. => PatientType::Three,
                  _ => panic!("PatientType::generate_patient error")
              }
          }

          impl EventNewPatient {

              pub fn get_current_t(&self) -> TimePoint {
                  self.current_t
              }

              pub fn iterate(self, reception_department: &mut
ReceptionDepartment) -> (Self, Option<EventReceptionDepartment>) {
                  reception_department.0.push(self.patient);
                  let delay = TimeSpan(DELAY_GEN.sample(&mut
rand::thread_rng()));
                  let patient_t = self.current_t + delay;
                  (
```

```
                    Self {
                        current_t: patient_t,
                        patient: Patient::new(patient_t,
generate_patient_type()),
                    },
                    EventReceptionDepartment::new(self.current_t,
reception_department)
                )
            }
        }
    }

    struct EventBase {
        current_t: TimePoint,
        patient_processor: QueueProcessor<Patient>
    }

    macro_rules! impl_event_new {
        ($event_type:ty, $queue_type:ty) => {
            impl $event_type {
                fn new(old_current_t: TimePoint, queue: &mut $queue_type) ->
Option<Self> {
                    if queue.0.is_any_free_processor() && !queue.0.is_empty()
{
                        let patient_processor = queue.0.acquire_processor();
                        let current_t = old_current_t +
Self::determine_delay(patient_processor.value().get_group());
                        Some(Self(EventBase{current_t, patient_processor}))
                    } else {
                        None
                    }
                }
            }
        };
    }

    struct EventReceptionDepartment(EventBase);
    impl_event_new!(EventReceptionDepartment, ReceptionDepartment);

    impl EventReceptionDepartment {
        fn determine_delay(patient_type: PatientType) -> TimeSpan {
            match patient_type {
                PatientType::One => TimeSpan(15.0),
                PatientType::Two => TimeSpan(40.0),
```

```
                PatientType::Three => TimeSpan(30.0),
            }
        }
    }

    enum ReceptionDepartmentTransitionToResult {
        PatientWards(Option<EventPatientWards>),
        FromReceptionToLaboratory(EventTransitionFromReceptionToLaboratory)
    }

    impl EventReceptionDepartment {
        pub fn iterate(
            self, reception_department: &mut ReceptionDepartment,
patient_wards: &mut PatientWards
        ) -> (Option<Self>, ReceptionDepartmentTransitionToResult) {
            let transition_to = match
self.0.patient_processor.value().get_group() {
                PatientType::One =>
ReceptionDepartmentTransitionToResult::PatientWards ({

patient_wards.0.push(self.0.patient_processor.value().clone());
                    EventPatientWards::new(self.0.current_t, patient_wards)
                }),
                PatientType::Two | PatientType::Three => {

ReceptionDepartmentTransitionToResult::FromReceptionToLaboratory(EventTransition
FromReceptionToLaboratory(
                        EventTransition::new(self.0.current_t,
self.0.patient_processor.value().clone())
                    ))
                },
            };
            drop(self.0.patient_processor);
            (Self::new(self.0.current_t, reception_department), transition_to)
        }
    }

    mod transition_lab_reception {
        use lazy_static::lazy_static;
        use rand::distributions::{Distribution, Uniform};
        use crate::task_2::{EventLabRegistration, EventPatientWards,
LabRegistry, Patient, PatientWards};
        use crate::{TimePoint, TimeSpan};
```

```
        lazy_static! {
            static ref RECEPTION_LABORATORY_TRANSITION_DELAY: Uniform<f64> =
Uniform::new(2.0, 5.0);
        }

        pub struct EventTransition {
            current_t: TimePoint,
            patient: Patient,
        }

        impl EventTransition {
            pub fn new(old_current_t: TimePoint, patient: Patient) -> Self {
                let delay =
TimeSpan(RECEPTION_LABORATORY_TRANSITION_DELAY.sample(&mut rand::thread_rng()));
                Self{current_t: old_current_t + delay, patient}
            }
        }

        pub struct EventTransitionFromReceptionToLaboratory(pub
EventTransition);

        impl EventTransitionFromReceptionToLaboratory {
            pub fn get_current_t(&self) -> TimePoint {
                self.0.current_t
            }

            pub fn iterate(self, lab_registry: &mut LabRegistry) ->
Option<EventLabRegistration> {
                lab_registry.0.push(self.0.patient);
                EventLabRegistration::new(self.0.current_t, lab_registry)
            }
        }

        pub struct EventTransitionFromLabToPatientWards(pub EventTransition);

        impl EventTransitionFromLabToPatientWards {
            pub fn get_current_t(&self) -> TimePoint {
                self.0.current_t
            }

            pub(super) fn iterate(self, patient_wards: &mut PatientWards) ->
Option<EventPatientWards> {
                patient_wards.0.push(self.0.patient);
                EventPatientWards::new(self.0.current_t, patient_wards)
```

```
            }
        }
    }

    struct EventPatientWards(EventBase);
    impl_event_new!(EventPatientWards, PatientWards);
    impl EventPatientWards {
        fn determine_delay(_: PatientType) -> TimeSpan {
            TimeSpan(Uniform::new(3.0, 8.0).sample(&mut thread_rng()))
        }

        pub fn iterate(self, patient_wards: &mut PatientWards) ->
(Option<Self>, EventTerminal) {
            let next_event = EventTerminal::new(self.0.current_t,
self.0.patient_processor.value().clone());
            drop(self.0.patient_processor);
            (Self::new(self.0.current_t, patient_wards), next_event)
        }
    }

    mod event_terminal {
        use crate::task_2::patient::Patient;
        use crate::TimePoint;

        pub struct EventTerminal {
            current_t: TimePoint,
            patient: Patient
        }

        impl EventTerminal {
            pub fn new(current_t: TimePoint, patient: Patient) -> Self {
                Self{current_t, patient}
            }
            pub fn get_current_t(&self) -> TimePoint {
                self.current_t
            }
            pub fn get_patient(&self) -> &Patient {
                &self.patient
            }
        }
    }

    fn get_erlang_distribution(shape: i64, scale: f64) -> rand_simple::Erlang
{
```

```
            let mut erlang = rand_simple::Erlang::new(
                [
                    thread_rng().next_u32(),
                    thread_rng().next_u32(),
                    thread_rng().next_u32(),
                ]
            );
            erlang.try_set_params(shape, scale).expect("Erlang set params
failed");
            erlang
        }

        struct EventLabRegistration(EventBase);
        impl_event_new!(EventLabRegistration, LabRegistry);
        impl EventLabRegistration {
            fn determine_delay(_: PatientType) -> TimeSpan {
                let delay = TimeSpan(get_erlang_distribution(3, 4.5).sample());
                // println!("{:?}", delay);
                delay
            }
            pub fn iterate(self, lab_registry: &mut LabRegistry, laboratory: &mut
Laboratory)
                -> (Option<Self>, Option<EventLaboratory>) {
                laboratory.0.push(self.0.patient_processor.value().clone());
                drop(self.0.patient_processor);
                (Self::new(self.0.current_t, lab_registry),
EventLaboratory::new(self.0.current_t, laboratory))
            }
        }

        struct EventLaboratory(EventBase);
        impl_event_new!(EventLaboratory, Laboratory);
        pub enum EventLaboratoryTransitionResult {
            TransitionFromLabToPatientWards(EventTransitionFromLabToPatientWards),
            Terminal(EventTerminal)
        }

        impl EventLaboratory {
            fn determine_delay(_: PatientType) -> TimeSpan {
                let delay = TimeSpan(get_erlang_distribution(2, 4.0).sample());
                delay
            }
            pub fn iterate(self, laboratory: &mut Laboratory) -> (Option<Self>,
EventLaboratoryTransitionResult) {
```

```
            let transition_to = match
self.0.patient_processor.value().get_group() {
                PatientType::One => panic!("Patient one can not be in the
laboratory"),
                PatientType::Two =>
EventLaboratoryTransitionResult::TransitionFromLabToPatientWards(
                    EventTransitionFromLabToPatientWards(
                        EventTransition::new(
                            self.0.current_t,
self.0.patient_processor.value().clone().upgrade_to_first_group()
                        )
                    )
                ),
                PatientType::Three =>
EventLaboratoryTransitionResult::Terminal(
                    EventTerminal::new(self.0.current_t,
self.0.patient_processor.value().clone())
                ),
            };
            drop(self.0.patient_processor);
            (Self::new(self.0.current_t, laboratory), transition_to)
        }
    }

    enum Event {
        NewPatient(EventNewPatient),
        ReceptionDepartment(EventReceptionDepartment),
        FromReceptionLaboratory(EventTransitionFromReceptionToLaboratory),
        FromLabToReception(EventTransitionFromLabToPatientWards),
        PatientWards(EventPatientWards),
        LabRegistration(EventLabRegistration),
        Laboratory(EventLaboratory),
        Terminal(EventTerminal),
    }

    impl Event {
        fn get_current_t(&self) -> TimePoint {
            match self {
                Event::NewPatient(event) => event.get_current_t(),
                Event::ReceptionDepartment(event) => event.0.current_t,
                Event::FromReceptionLaboratory(event) =>
event.get_current_t(),
                Event::FromLabToReception(event) => event.get_current_t(),
                Event::PatientWards(event) => event.0.current_t,
```

```
                Event::LabRegistration(event) => event.0.current_t,
                Event::Laboratory(event) => event.0.current_t,
                Event::Terminal(event) => event.get_current_t(),
            }
        }
    }

    impl Default for Event {
        fn default() -> Self {
            Self::NewPatient(EventNewPatient::default())
        }
    }

    impl Eq for Event {}

    impl PartialEq<Self> for Event {
        fn eq(&self, other: &Self) -> bool {
            self.get_current_t() == other.get_current_t()
        }
    }

    impl PartialOrd<Self> for Event {
        fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
            other.get_current_t().partial_cmp(&self.get_current_t())
        }
    }

    impl Ord for Event {
        fn cmp(&self, other: &Self) -> Ordering {
            self.partial_cmp(&other).expect("Failed to compare events")
        }
    }

    #[cfg(test)]
    mod tests {
        use std::collections::{BinaryHeap, VecDeque};
        use crate::task_2::{
            Event, EventLaboratoryTransitionResult, LabRegistry,
            Laboratory, PatientWards, ReceptionDepartment,
ReceptionDepartmentTransitionToResult
        };
        use crate::task_2::event_terminal::EventTerminal;
        use crate::task_2::queue_resource::QueueResource;
        use crate::{TimePoint, TimeSpan};
```

```
        use crate::task_2::patient::Patient;

        #[test]
        fn test_general() {
            let end_time = TimePoint(100000.0);

            let mut reception_department =
ReceptionDepartment(QueueResource::new(BinaryHeap::new(), 2));
            let mut patient_wards =
PatientWards(QueueResource::new(VecDeque::new(), 3));
            let mut lab_registry =
LabRegistry(QueueResource::new(VecDeque::new(), 1));
            let mut laboratory =
Laboratory(QueueResource::new(VecDeque::new(), 2));

            let mut last_lab_registration = TimePoint(0.0);
            let mut lab_registration_interval_sum = TimeSpan(0.0);
            let mut lab_registration_count = 0usize;

            let mut total_terminal_patients = 0usize;
            let mut total_patients_spent_time = TimeSpan::default();

            let mut nodes = BinaryHeap::new();
            nodes.push(Event::default());

            let _ = loop {

                let next_event = nodes.pop().unwrap();
                if next_event.get_current_t() > end_time {
                    break next_event;
                }
                match next_event {
                    Event::NewPatient(event) => {
                        let res = event.iterate(&mut reception_department);
                        nodes.push(Event::NewPatient(res.0));
                        if let Some(next_event) = res.1 {

nodes.push(Event::ReceptionDepartment(next_event));
                        }
                    },
                    Event::ReceptionDepartment(event) => {
                        let res = event.iterate(&mut reception_department,
&mut patient_wards);
                        if let Some(self_event) = res.0 {
```

```
nodes.push(Event::ReceptionDepartment(self_event));
                        }
                        match res.1 {

ReceptionDepartmentTransitionToResult::PatientWards(event) => {
                                if let Some(event) = event {
                                    nodes.push(Event::PatientWards(event));
                                }
                            }

ReceptionDepartmentTransitionToResult::FromReceptionToLaboratory(event) => {

nodes.push(Event::FromReceptionLaboratory(event));
                            }
                        }
                    },
                    Event::FromReceptionLaboratory(event) => {
                        if let Some(res) = event.iterate(&mut lab_registry) {
                            nodes.push(Event::LabRegistration(res));
                        }
                    },
                    Event::FromLabToReception(event) => {
                        if let Some(res) = event.iterate(&mut patient_wards) {
                            nodes.push(Event::PatientWards(res));
                        }
                    },
                    Event::PatientWards(event) => {
                        let (self_event, terminal) = event.iterate(&mut
patient_wards);
                        if let Some(res_event) = self_event {
                            nodes.push(Event::PatientWards(res_event));
                        }
                        nodes.push(Event::Terminal(terminal));
                    },
                    Event::LabRegistration(event) => {
                        lab_registration_count += 1;
                        lab_registration_interval_sum += event.0.current_t -
last_lab_registration;
                        last_lab_registration = event.0.current_t;

                        let (self_event, next_event) = event.iterate(&mut
lab_registry, &mut laboratory);
                        if let Some(self_event) = self_event {
```

```
                            nodes.push(Event::LabRegistration(self_event));
                        }
                        if let Some(next_event) = next_event {
                            nodes.push(Event::Laboratory(next_event));
                        }
                    },
                    Event::Laboratory(event) => {
                        let (self_event, next_event) = event.iterate(&mut
laboratory);
                        if let Some(self_event) = self_event {
                            nodes.push(Event::Laboratory(self_event));
                        }
                        match next_event {

EventLaboratoryTransitionResult::TransitionFromLabToPatientWards(next_event) =>
{

nodes.push(Event::FromLabToReception(next_event));
                            }
                            EventLaboratoryTransitionResult::Terminal(event)
=> {
                                nodes.push(Event::Terminal(event));
                            }
                        }
                    },
                    Event::Terminal(event) => {
                        total_terminal_patients += 1;
                        total_patients_spent_time += event.get_current_t() -
event.get_patient().get_clinic_in_t();
                    },
                }
            };

            println!("Mean time: {:?}", total_patients_spent_time.0 /
total_terminal_patients as f64);
            println!("Lab registration mean time: {:?}",
lab_registration_interval_sum.0 / lab_registration_count as f64);
        }

        #[test]
        fn test_event_ordering() {
            let mut bh = BinaryHeap::new();

            bh.push(Event::Terminal(EventTerminal::new(TimePoint(10.0),
```

```
Patient::default())));
            bh.push(Event::Terminal(EventTerminal::new(TimePoint(9.0),
Patient::default())));
            bh.push(Event::Terminal(EventTerminal::new(TimePoint(8.0),
Patient::default())));
            bh.push(Event::Terminal(EventTerminal::new(TimePoint(14.0),
Patient::default())));
            bh.push(Event::Terminal(EventTerminal::new(TimePoint(1.0),
Patient::default())));

            assert_eq!(bh.pop().unwrap().get_current_t(), TimePoint(1.0));
            assert_eq!(bh.pop().unwrap().get_current_t(), TimePoint(8.0));
            assert_eq!(bh.pop().unwrap().get_current_t(), TimePoint(9.0));
            assert_eq!(bh.pop().unwrap().get_current_t(), TimePoint(10.0));
            assert_eq!(bh.pop().unwrap().get_current_t(), TimePoint(14.0));
        }
    }
```