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 №2

Моделювання систем

Тема: Об’єктно-орієнтований підхід до побудови імітаційних моделей дискретно-подійних систем

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

Перевірив:
Стеценко І.В.

Київ 2024

ЗМІСТ

1 МЕТА КОМП’ЮТЕРНОГО ПРАКТИКУМУ

Мета лабораторної роботи - розробити об'єктно-орієнтовану модель системи обслуговування, використовуючи алгоритм імітації за принципом Δt.

2 ЗАВДАННЯ

1. Реалізувати алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу. 5 балів.
2. Модифікувати алгоритм, додавши обчислення середнього завантаження пристрою. 5 балів.
3. Створити модель за схемою, представленою на рисунку 2.1. 30 балів.
4. Виконати верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі. Навести результати верифікації у таблиці. 10 балів.
5. Модифікувати клас PROCESS, щоб можна було його використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. 20 балів.
6. Модифікувати клас PROCESS, щоб можна було організовувати вихід в два і більше наступних блоків, в тому числі з поверненням у попередні блоки. 30 балів.

3 ВИКОНАННЯ

Під час симуляції збираємо такі дані:

1. Кількість запитів - Quantity
2. Успішно оброблені об'єкти - QuantityProcessed
3. Час роботи пристрою - WorkTime
4. Кількість відмов - Failure
5. Довжина черги за весь час - FullMeanQueue += Queue * deltaT

Після завершення симуляції обчислюємо:

1. Частка роботи пристрою = WorkTime / AllTime
2. Середня довжина черги = FullMeanQueue / AllTime
3. Ймовірність відмови = Failure / Quantity

3.1Загальний приклад

Розглянемо схему моделі. Опишемо параметри в коді.

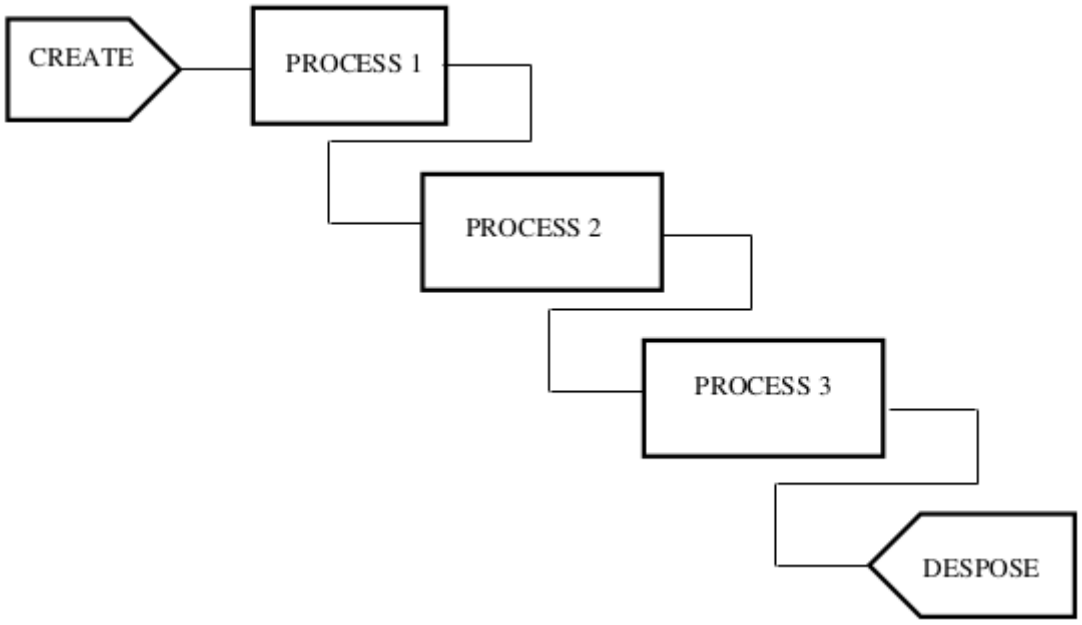

Рисунок 3.1 — Загальна схема моделі

3.1.1 Параметри симуляції в коді:

```
      let delay_gen = DelayGen::Uniform(rand_distr::Uniform::<f64>::new(0.,
10.));
      let element_process_3 = Rc::new(RefCell::new(ElementProcess::new(
          ElementBase::new("Process 3", delay_gen, Default::default()), 3,
          vec![Device::new("Device 1", delay_gen)]
      )));
      let element_process_2 = Rc::new(RefCell::new(ElementProcess::new(
          ElementBase::new(
              "Process 2", delay_gen,
              ProbabilityElementsMap::new(vec![(element_process_3.clone(),
1.0)])
          ),
          3,
          vec![Device::new("Device 1", delay_gen), ]
      )));
      let element_process_1 = Rc::new(RefCell::new(ElementProcess::new(
          ElementBase::new(
              "Process 1", delay_gen,
              ProbabilityElementsMap::new(vec![(element_process_2.clone(),
1.0)])
          ),
          3,
          vec![Device::new("Device 1", delay_gen)]
      )));

      let element_create = Rc::new(
          RefCell::new(ElementCreate(ElementBase::new(
              "Create", delay_gen,
              ProbabilityElementsMap::new(vec![(element_process_1.clone(),
1.0)])
          )))
      );
```

3.1.2 Результати симуляції

```
      name = Create
           quantity_processed = 200
      name = Process 1
           quantity_processed = 174
           work_time = 491
```

```
        mean_queue = 1.447
        count_rejected = 25
        max_queue = 3
        failure_probability = 0.125
    name = Process 2
        quantity_processed = 161
        work_time = 486
        mean_queue = 0.899
        count_rejected = 9
        max_queue = 3
        failure_probability = 0.05172413793103448
    name = Process 3
        quantity_processed = 160
        work_time = 436
        mean_queue = 0.686
        count_rejected = 0
        max_queue = 3
        failure_probability = 0
```

Бачимо, що для першого процесу імовірність відмови близько 12.5%. Ми можемо зменшити відсоток відмови, якщо черга буде нескінченна.

3.2 Приклад з нескінченною чергою

Нескінченна черга повинна зменшити відсоток відмови до числа, що дуже близьке до 0.

3.2.1 Параметри симуляції в коді

```
    let delay_gen = DelayGen::Uniform(rand_distr::Uniform::<f64>::new(0.,
10.));
    let element_process_3 = Rc::new(RefCell::new(ElementProcess::new(
        ElementBase::new("Process 3", delay_gen, Default::default()),
usize::MAX,
        vec![Device::new("Device 1", delay_gen)]
    )));
    let element_process_2 = Rc::new(RefCell::new(ElementProcess::new(
        ElementBase::new(
            "Process 2", delay_gen,
            ProbabilityElementsMap::new(vec![(element_process_3.clone(),
1.0)])
        ),
        usize::MAX,
        vec![Device::new("Device 1", delay_gen), ]
```

```
    )));
    let element_process_1 = Rc::new(RefCell::new(ElementProcess::new(
        ElementBase::new(
            "Process 1", delay_gen,
            ProbabilityElementsMap::new(vec![(element_process_2.clone(),
1.0)])
        ),
        usize::MAX,
        vec![Device::new("Device 1", delay_gen)]
    )));

    let element_create = Rc::new(
        RefCell::new(ElementCreate(ElementBase::new(
            "Create", delay_gen,
            ProbabilityElementsMap::new(vec![(element_process_1.clone(),
1.0)])
        )))
    );

    let elements: Vec<Rc<RefCell<dyn Element>>> = vec![
        element_create,
        element_process_1,
        element_process_2,
        element_process_3,
    ];

    simulate_model(elements, 1000);
```

3.2.2 Результати симуляції

```
    name = Create
         quantity_processed = 194
    name = Process 1
         quantity_processed = 190
         work_time = 424
         mean_queue = 4.956
         count_rejected = 0
         max_queue = 18446744073709551615
         failure_probability = 0
    name = Process 2
         quantity_processed = 180
         work_time = 474
         mean_queue = 2.578
         count_rejected = 0
         max_queue = 18446744073709551615
```

```
        failure_probability = 0
    name = Process 3
        quantity_processed = 174
        work_time = 447
        mean_queue = 1.228
        count_rejected = 0
        max_queue = 18446744073709551615
        failure_probability = 0
```

3.3 Приклад переходу з імовірностями

Побудуємо схему, що зображує імовірнісний перехід між процесами.

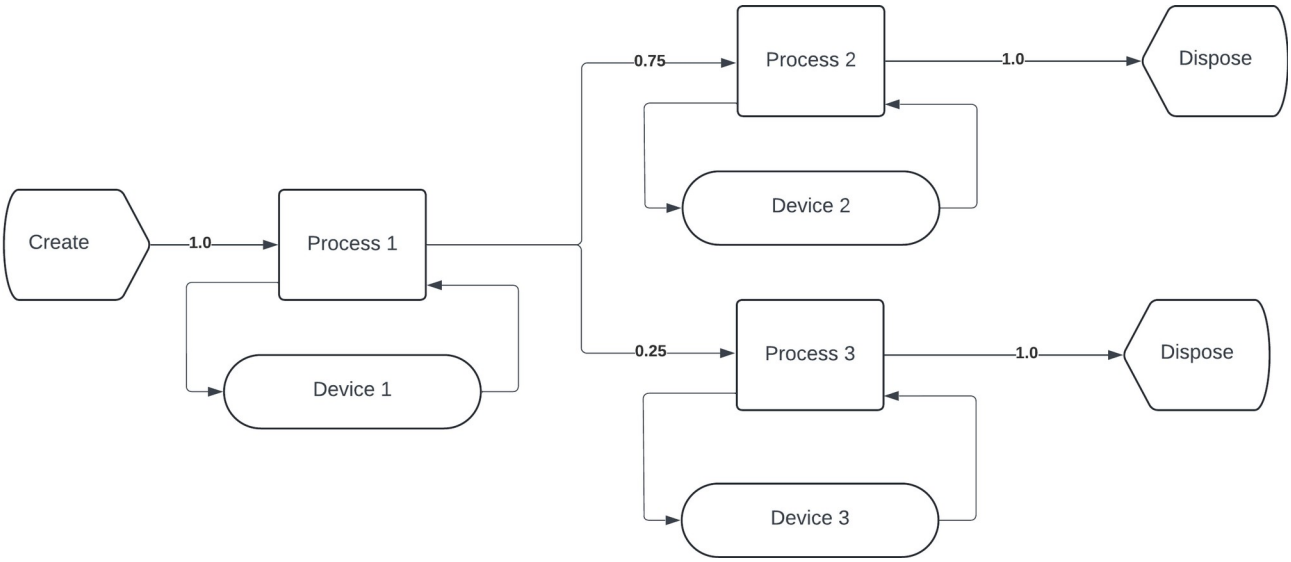

Рисунок 3.2 — Схема переходу з імовірностями

3.3.1 Параметри симуляції в коді

```
    let queue_size: usize = 8;
    let delay_gen = DelayGen::Uniform(rand_distr::Uniform::<f64>::new(0.,
10.));
    let element_process_3 = Rc::new(RefCell::new(ElementProcess::new(
        ElementBase::new("Process 3", delay_gen,
Default::default()),queue_size,
        vec![Device::new("Device 1", delay_gen)]
    )));
    let element_process_2 = Rc::new(RefCell::new(ElementProcess::new(
        ElementBase::new("Process 2", delay_gen, Default::default()),
        queue_size,
        vec![Device::new("Device 1", delay_gen), ]
    )));
    let element_process_1 = Rc::new(RefCell::new(ElementProcess::new(
        ElementBase::new(
            "Process 1", delay_gen,
            ProbabilityElementsMap::new(
```

```
                vec![
                    (element_process_3.clone(), 0.25),
                    (element_process_2.clone(), 0.75),
                ]
            )
        ),
        queue_size,
        vec![Device::new("Device 1", delay_gen)]
    )));

    let element_create = Rc::new(
        RefCell::new(ElementCreate(ElementBase::new(
            "Create", delay_gen,
            ProbabilityElementsMap::new(vec![(element_process_1.clone(),
1.0)])
        )))
    );

    let elements: Vec<Rc<RefCell<dyn Element>>> = vec![
        element_create,
        element_process_1,
        element_process_2,
        element_process_3,
    ];

    simulate_model(elements, 1000);
```

3.3.2 Результати симуляції

```
    name = Create
         quantity_processed = 204
    name = Process 1
         quantity_processed = 191
         work_time = 441
         mean_queue = 4.507
         count_rejected = 5
         max_queue = 8
         failure_probability = 0.024509803921568627
    name = Process 2
         quantity_processed = 145
         work_time = 415
         mean_queue = 0.747
         count_rejected = 0
         max_queue = 8
         failure_probability = 0
```

```
name = Process 3
     quantity_processed = 43
     work_time = 188
     mean_queue = 0.022
     count_rejected = 0
     max_queue = 8
     failure_probability = 0
```

Як бачимо, у попередньому пункті час роботи другого та третього процесів майже однаковий. У поточному прикладі різниця між часом роботи приблизно в 2 рази на користь другого процесу.

ВИСНОВОК

В ході виконання лабораторної роботи було досліджено об'єктно-орієнтований алгоритм імітації за принципом Δt. Створена архітектура розподіляє функції між моделлю, її елементами, виводом та наступними елементами, що забезпечує гнучкість при побудові складних моделей.

Застосування парадигми об'єктно-орієнтованого програмування (ООП) значно підвищує гнучкість та масштабованість розробленої системи. ООП дозволяє створювати складні моделі шляхом комбінування та розширення базових компонентів. Це забезпечує можливість легко впроваджувати нові функціональності, такі як підпроцеси або ймовірнісні переходи між елементами моделі. Така архітектура сприяє модульності коду, полегшує його підтримку та адаптацію до нових вимог, що особливо цінно при моделюванні комплексних систем.

Для аналізу ефективності систем було впроваджено збір статистики, включаючи кількість запитів, оброблених об'єктів, час роботи та кількість відмов. На основі цих даних обчислювалися ключові показники: середнє завантаження пристрою, середня довжина черги та ймовірність відмови.

Проведено серію симуляцій з різними параметрами:

1. Модель з обмеженою чергою
2. Модель з необмеженою чергою
3. Модель з імовірнісним розподілом між процесами

Аналіз результатів показав, що зміна розміру черги та введення імовірнісного розподілу суттєво впливають на ефективність системи. Зокрема, вдалося знизити ймовірність відмови з 10.7% до 3.9% при збалансованому навантаженні.

Такий підхід до моделювання дозволяє експериментально визначати оптимальні параметри для різних конфігурацій систем, що є цінним

інструментом для проектування та оптимізації складних процесів.

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

```
    use crate::delay_gen::DelayGen;
    use crate::device::Device;
    use crate::element_process::ElementProcess;
    use crate::prob_el_map::ProbabilityElementsMap;
    use std::cell::RefCell;
    use std::fmt::{Debug, Formatter};
    use std::rc::Rc;

    type TimeUnit = u64;

    pub mod delay_gen {
        use crate::TimeUnit;
        use rand::distributions::Distribution;
        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) -> TimeUnit {
                match self {
                    Self::Normal(dist) => dist.sample(&mut
thread_rng()).round() as TimeUnit,
                    Self::Uniform(dist) => dist.sample(&mut
thread_rng()).round() as TimeUnit,
                    Self::Exponential(dist) => dist.sample(&mut
thread_rng()).round() as TimeUnit,
                }
            }
        }
    }

    pub mod prob_el_map {
        use crate::Element;
        use rand::random;
        use std::cell::{RefCell, RefMut};
        use std::fmt::{Debug, Formatter};
        use std::rc::Rc;
```

```
        #[derive(Default)]
        pub struct ProbabilityElementsMap(Vec<(Rc<RefCell<dyn Element>>,
f64)>);

        impl Debug for ProbabilityElementsMap {
            fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
                write!(f, "")
            }
        }

        impl ProbabilityElementsMap {
            pub fn new(next_elements_map: Vec<(Rc<RefCell<dyn Element>>,
f64)>) -> Self {
                const EPSILON: f64 = 0.001;
                assert!(
                    next_elements_map.iter()
                        .map(|e| (*e).1).all(|v| v >= 0.0 && v <= 1.0),
                    "Values must be between 0.0 and 1.0"
                );
                assert!((next_elements_map.iter().map(|e| (*e).1).sum::<f64>()
- 1.0).abs() < EPSILON);
                Self(next_elements_map)
            }

            pub fn sample(&self) -> Option<RefMut<dyn Element>> {
                if self.0.is_empty() {
                    return None;
                }

                let total_prob: f64 = self.0.iter().map(|e| (*e).1).sum();
                let rand_value = random::<f64>() * total_prob;
                let mut current_sum = 0.0;

                let mut target_index = self.0.len() - 1;
                for (index, (_, prob)) in self.0.iter().enumerate() {
                    current_sum += *prob;
                    if rand_value < current_sum {
                        target_index = index;
                        break;
                    }
                }
                Some(self.0.iter().nth(target_index)?.0.borrow_mut())
            }
        }
```

```
}

pub trait Element : Debug {
    fn in_act(&mut self);
    fn out_act(&mut self);
    fn set_current_t(&mut self, current_t: TimeUnit);
    fn get_next_t(&self) -> TimeUnit;
    fn update_statistic(&mut self, next_t: TimeUnit, current_t: TimeUnit);
    fn update_next_t(&mut self);
    fn print_stats(&self);
}

pub struct ElementBase {
    name: &'static str,
    next_t: TimeUnit,
    current_t: TimeUnit,
    work_time: TimeUnit,
    quantity: usize,
    quantity_processed: usize,
    is_working: bool,
    delay_gen: DelayGen,
    next_elements: ProbabilityElementsMap,
}

impl Debug for ElementBase {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(
            f, "name: {:?}, next_t: {:?}, current_t: {:?}, quantity: {:?},
quantity_processed: {:?}, work_time: {:?}",
            self.name, self.next_t, self.current_t, self.quantity,
self.quantity_processed, self.work_time
        )
    }
}

impl ElementBase {
    pub fn new(name: &'static str, delay_gen: DelayGen, next_elements:
ProbabilityElementsMap) -> Self {
        Self{
            name, next_t: 0, current_t: 0, quantity: 0, work_time: 0,
            quantity_processed: 0, is_working: false,
            delay_gen, next_elements
        }
    }
```

```
    }

    impl ElementBase {
        fn in_act(&mut self) {
            self.quantity += 1;
            self.is_working = true;
        }

        fn out_act(&mut self) {
            self.quantity_processed += 1;
            self.is_working = false;
            if let Some(mut next_element) = self.next_elements.sample() {
                next_element.in_act();
            }
        }

        fn set_current_t(&mut self, current_t: TimeUnit) {
            self.current_t = current_t;
        }
        fn get_next_t(&self) -> TimeUnit {
            self.next_t
        }

        fn update_statistic(&mut self, next_t: TimeUnit, current_t: TimeUnit)
{
            self.work_time += if self.is_working { next_t - current_t } else {
0 };
        }

        fn print_stats(&self) {
            println!("name = {}", self.name);
            println!("\tquantity_processed = {}", self.quantity_processed);
        }
    }

    #[derive(Debug)]
    struct ElementCreate(ElementBase);

    impl Element for ElementCreate {
        fn in_act(&mut self) {
            self.0.in_act()
        }

        fn out_act(&mut self) {
```

```
            self.0.out_act();
            self.update_next_t();
        }

        fn set_current_t(&mut self, current_t: TimeUnit) {
            self.0.set_current_t(current_t)
        }

        fn get_next_t(&self) -> TimeUnit {
            self.0.get_next_t()
        }

        fn update_statistic(&mut self, next_t: TimeUnit, current_t: TimeUnit)
{
            self.0.update_statistic(next_t, current_t)
        }

        fn update_next_t(&mut self) {
            self.0.next_t = self.0.current_t + self.0.delay_gen.sample();
        }

        fn print_stats(&self) {
            self.0.print_stats()
        }
    }

    pub mod device {
        use crate::delay_gen::DelayGen;
        use crate::TimeUnit;

        #[derive(Debug)]
        pub struct Device {
            name: &'static str,
            next_t: TimeUnit,
            delay_gen: DelayGen,
            is_working: bool
        }

        impl Device {
            pub fn new(name: &'static str, delay_gen: DelayGen) -> Self {
                Self{name, next_t: 0, delay_gen, is_working: false}
            }

            pub fn in_act(&mut self, current_t: TimeUnit) {
```

```
                self.next_t = current_t + self.delay_gen.sample();
                self.is_working = true;
            }

            pub fn out_act(&mut self) {
                self.next_t = TimeUnit::MAX;
                self.is_working = false;
            }

            pub fn get_next_t(&self) -> TimeUnit {
                self.next_t
            }

            pub fn get_is_working(&self) -> bool {
                self.is_working
            }
        }
    }

    mod element_process {
        use crate::device::Device;
        use crate::{Element, ElementBase, TimeUnit};
        use std::fmt::{Debug, Formatter};

        pub struct ElementProcess {
            base: ElementBase,
            queue: usize,
            max_queue: usize,
            count_rejected: usize,
            mean_queue: f64,
            devices: Vec<Device>
        }

        impl Debug for ElementProcess {
            fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
                write!(
                    f, "{:?}, queue: {:?}, max_queue: {:?}, count_rejected:
{:?}, mean_queue: {:?}",
                    self.base, self.queue, self.max_queue,
self.count_rejected, self.mean_queue
                )
            }
        }
```

```
impl ElementProcess {
    pub fn new(mut base: ElementBase, max_queue: usize, devices:
Vec<Device>) -> Self {
        base.next_t = TimeUnit::MAX;
        Self{base, queue: 0, max_queue, count_rejected: 0, mean_queue:
0.0, devices}
    }
}

fn find_free_device(devices: &mut [Device]) -> Option<&mut Device> {
    devices.iter_mut().find(|d| !(*d).get_is_working())
}

impl Element for ElementProcess {
    fn in_act(&mut self) {
        self.base.in_act();
        if let Some(free_device) = find_free_device(&mut self.devices)
{
            free_device.in_act(self.base.current_t);
        } else if self.queue < self.max_queue {
            self.queue += 1;
        } else {
            self.count_rejected += 1;
        }
        self.update_next_t();
    }
    fn out_act(&mut self) {
        for device in &mut self.devices {
            device.out_act();
            self.base.out_act();
            if self.queue > 0 {
                self.queue -= 1;
                device.in_act(self.base.current_t);
            }
        }
        self.update_next_t();
    }

    fn set_current_t(&mut self, current_t: TimeUnit) {
        self.base.set_current_t(current_t);
    }

    fn get_next_t(&self) -> TimeUnit {
        self.base.get_next_t()
```

```
            }
            fn update_statistic(&mut self, next_t: TimeUnit, current_t:
TimeUnit) {
                self.base.update_statistic(next_t, current_t);
                self.mean_queue += (self.queue as u64 * (next_t - current_t))
as f64;
            }
            fn update_next_t(&mut self) {
                self.base.next_t = self.devices.iter().map(|d|
d.get_next_t()).min().unwrap()
            }
            fn print_stats(&self) {
                self.base.print_stats();
                println!("\twork_time = {}", self.base.work_time);
                println!("\tmean_queue = {}", self.mean_queue);
                println!("\tcount_rejected = {}", self.count_rejected);
                println!("\tmax_queue = {}", self.max_queue);
                println!("\tfailure_probability = {}", self.count_rejected as
f64 / self.base.quantity as f64);
            }
        }
    }


    fn simulate_model(mut elements: Vec<Rc<RefCell<dyn Element>>>, max_time:
TimeUnit) {
        let mut current_t: TimeUnit = 0;
        while current_t < max_time {
            let next_t = elements.iter()
                .map(|d| d.borrow().get_next_t()).min()
                .expect("elements can not be empty");
            for el in &mut elements {
                el.borrow_mut().update_statistic(next_t, current_t);
            }
            current_t = next_t;
            for el in &mut elements {
                el.borrow_mut().set_current_t(current_t)
            }
            for el in &mut elements {
                if el.borrow().get_next_t() == current_t {
                    el.borrow_mut().out_act();
                }
            }
            // for el in &mut *elements {
```

```
            //     println!("{:?}", &*el.borrow_mut());
            // }
        }
        for el in &elements {
            el.borrow().print_stats();
        }
    }


    fn main() {}

    #[cfg(test)]
    mod tests {
        use super::*;

        #[test]
        fn test_general() {
            let delay_gen =
DelayGen::Uniform(rand_distr::Uniform::<f64>::new(0., 10.));
            let element_process_3 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new("Process 3", delay_gen, Default::default()),
3,
                vec![Device::new("Device 1", delay_gen)]
            )));
            let element_process_2 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new(
                    "Process 2", delay_gen,
                    ProbabilityElementsMap::new(vec!
[(element_process_3.clone(), 1.0)])
                ),
                3,
                vec![Device::new("Device 1", delay_gen), ]
            )));
            let element_process_1 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new(
                    "Process 1", delay_gen,
                    ProbabilityElementsMap::new(vec!
[(element_process_2.clone(), 1.0)])
                ),
                3,
                vec![Device::new("Device 1", delay_gen)]
            )));

            let element_create = Rc::new(
```

```
                RefCell::new(ElementCreate(ElementBase::new(
                    "Create", delay_gen,
                    ProbabilityElementsMap::new(vec!
[(element_process_1.clone(), 1.0)])
                )))
            );

            let elements: Vec<Rc<RefCell<dyn Element>>> = vec![
                element_create,
                element_process_1,
                element_process_2,
                element_process_3,
            ];

            simulate_model(elements, 1000);
        }

        #[test]
        fn test_infinite_queue() {
            let delay_gen =
DelayGen::Uniform(rand_distr::Uniform::<f64>::new(0., 10.));
            let element_process_3 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new("Process 3", delay_gen, Default::default()),
usize::MAX,
                vec![Device::new("Device 1", delay_gen)]
            )));
            let element_process_2 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new(
                    "Process 2", delay_gen,
                    ProbabilityElementsMap::new(vec!
[(element_process_3.clone(), 1.0)])
                ),
                usize::MAX,
                vec![Device::new("Device 1", delay_gen), ]
            )));
            let element_process_1 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new(
                    "Process 1", delay_gen,
                    ProbabilityElementsMap::new(vec!
[(element_process_2.clone(), 1.0)])
                ),
                usize::MAX,
                vec![Device::new("Device 1", delay_gen)]
            )));
```

```
            let element_create = Rc::new(
                RefCell::new(ElementCreate(ElementBase::new(
                    "Create", delay_gen,
                    ProbabilityElementsMap::new(vec!
[(element_process_1.clone(), 1.0)])
                )))
            );

            let elements: Vec<Rc<RefCell<dyn Element>>> = vec![
                element_create,
                element_process_1,
                element_process_2,
                element_process_3,
            ];

            simulate_model(elements, 1000);
        }

        #[test]
        fn test_probability_transition() {
            let queue_size: usize = 8;
            let delay_gen =
DelayGen::Uniform(rand_distr::Uniform::<f64>::new(0., 10.));
            let element_process_3 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new("Process 3", delay_gen,
Default::default()),queue_size,
                vec![Device::new("Device 1", delay_gen)]
            )));
            let element_process_2 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new("Process 2", delay_gen, Default::default()),
                queue_size,
                vec![Device::new("Device 1", delay_gen), ]
            )));
            let element_process_1 = Rc::new(RefCell::new(ElementProcess::new(
                ElementBase::new(
                    "Process 1", delay_gen,
                    ProbabilityElementsMap::new(
                        vec![
                            (element_process_3.clone(), 0.25),
                            (element_process_2.clone(), 0.75),
                        ]
                    )
                ),
```

```
                queue_size,
                vec![Device::new("Device 1", delay_gen)]
            )));

            let element_create = Rc::new(
                RefCell::new(ElementCreate(ElementBase::new(
                    "Create", delay_gen,
                    ProbabilityElementsMap::new(vec!
[(element_process_1.clone(), 1.0)])
                )))
            );

            let elements: Vec<Rc<RefCell<dyn Element>>> = vec![
                element_create,
                element_process_1,
                element_process_2,
                element_process_3,
            ];

            simulate_model(elements, 1000);
        }
    }
```