**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
Чернігівський національний технологічний університет

# ПРОГРАМУВАННЯ ПАРАЛЕЛЬНИХ ПРОЦЕСІВ

МЕТОДИЧНІ ВКАЗІВКИ
до виконання розрахунково-графічної роботи з дисципліни
**« Об'єктно-орієнтоване програмування»**
для студентів напрямів підготовки
6.050102 –"Комп'ютерна інженерія"
6.050103 –"Програмна інженерія"

ЗАТВЕРДЖЕНО
на засіданні кафедри
програмної інженерії
протокол № 7 від 27.02.14

**Чернігів ЧНТУ 2014**

Програмування паралельних процесів. Методичні вказівки до виконання розрахунково-графічної роботи з дисципліни «Об’єктно-орієнтоване програмування» для студентів напряму підготовки 6.050102 –“Комп’ютерна інженерія” 6.050103 – „Програмна інженерія”. /Укл.: Бивойно П.Г. Бивойно Т.П. – Чернігів: ЧНТУ, 2014. – 35 с.

Укладачі: Бивойно Павло Георгійович, кандидат технічних наук, доцент
Бивойно Тарас Павлович, старший викладач

Відповідальний за випуск: Литвинов В.В., завідуючий кафедрою програмної інженерії, доктор технічних наук, профессор

Рецензент: Скітер І.С., кандидат техніч. наук, доцент кафедри програмної інженерії Чернігівського національного технологічного університету

ЗМІСТ

# Вступ

Поняття розрахунково-графічна робота (РГР) прийшло у вищу школу з дисциплін навчальних планів інженерно-механічних спеціальностей. У межах розрахунково-графічної роботи студент повинен був виконати деякі розрахунки і графічні побудови (креслення, ескіз, діаграму), що були необхідні для вирішення деякої інженерної проблеми. До того ж часто використовувалися так звані графоаналітичні методи розрахунків. Тому назва «розрахунково-графічна робота» була, безсумнівно, доречною.

Проте потім, завдання студентам, що були пов'язані з вирішенням проблем інших галузей знань, теж почали називати розрахунково-графічними роботами. З'явилися РГР з хімії, бухгалтерського обліку та інші. У цих завданнях, зазвичай, ніяких графічних елементів вже не було, залишалися тільки розрахунки, а інколи і їх не було, та назва збереглася.

У курсі «Об’єктно-орієнтоване програмування» виконання розрахунково-графічної є частиною самостійної роботи студентів над дисципліною. РГР передбачає проведення об’єктно-орієнтованого аналізу деякої предметної області і подальшу побудову програми, що вирішує деякі проблеми цієї області. Ніяких розрахунків робота не передбачає, але має буди програмна реалізація додатку.

Окрім аналізу предметної області студент має проаналізувати можливості Java та визначити перелік інтерфейсів та класів, що можуть бути використані або безпосередньо, або як батьківські класи для реалізації класів майбутнього додатку. У процесі програмної реалізації мають бути візуальна композиція та класи, що відповідають складовим частинам предметної області

Роботу виконує команда з двох студентів.

Дизайнер перш за все розробляє перелік компонентів для візуальної частини і після цього створює візуальну частину проекту.

Кодер створює класи для компонентів.

Результатом розрахунково-графічної роботи є звіт обсягом приблизно 30 сторінок друкованого тексту оформленого відповідно до стандартів кафедри. Бали за розрахунково-графічну роботу виставляються з урахуванням своєчасності та якості виконання, а також особистого внеску у розробку і включаються як складова до результатів семестру.

За звичай номер варіанту завдання для РГР призначається викладачем, та студент може і сам обрати предметну область для моделювання, але завдання обов'язково має бути погоджено з викладачем

# 1 ЗАВДАННЯ ДО РОЗРАХУНКОВО - ГРАФІЧНОЇ РОБОТИ

## Завдання 1. Моделювання продажу квитків в аеропорті, варіант KassaInAirport

У курортному аеропорті малої авіації до продажу квитків залучено стюардес. На продаж квитка стюардеса витрачає випадковий час.

Друге завдання стюардеси - проводити групи пасажирів до літаків, які чекають завантаження, тому періодично стюардеси йдуть із групою пасажирів на літовище. А якщо літака нема, пасажири чекають.

В аеропорті обмежена кількість малих літаків, які летять і повертаються.

Якщо довжина всіх черг на покупку квитків до вільних стюардес перевищують критичне значення, пасажир іде на залізничний вокзал.

## Завдання 2. Моделювання роботи ділянки тестування комп'ютерів, варіант TestPC

На останній стадії складання системних блоків комп’ютера виконується їх тестування та пакування. Блоки з головного конвеєра надходять на ділянку тестування через випадкові проміжки часу. На ділянці тестування є площадка для розміщення блоків, розмір якої обмежений. Якщо на площадці немає місця для чергового блоку, він відправляється на пакування без тестування. Блоки, що не пройшли тестування відправляються на ділянку налагодження, після чого знову проходять тестування. У тому випадку, якщо блок не проходить тестування повторно, він відправляється у брак. Блоки, що пройшли тестування йдуть на пакування.

## Завдання 3. Моделювання посівної компанії, варіант BeanFeast

Сівбу зернових на полі забезпечують декілька сівалок. Зерно для сівалок підвозять вантажівки. З однієї вантажівки можна завантажити декілька сівалок. Вантажівки завантажуються на складі.

## Завдання 4. Моделювання авіаперевезень, варіант AviaBridg

У аеропорт прибувають контейнери з гуманітарною допомогою для відправлення у зону стихійного лиха. Контейнери потрапляють на навантажувальну площадку, а з площадки бригада вантажників завантажує контейнери у літак, якщо він готовий для навантаження. Літак відправляється відразу після заповнення. Після повернення на свій аеродром літак знову готовий до завантаження.

### Завдання 5. Моделювання розвантаження автомобілів, варіант TransAvto

Вантажівки прибувають на розвантажувальну станцію у випадкові моменти часу, причому інтенсивність прибуття залежить від часу доби (день - ніч). Час розвантаження вантажівки випадковий. Розвантаженням вантажівки займається бригада вантажників. На роботу виходить кілька бригад вантажників.

### Завдання 6. Моделювання виборів, варіант Election

На виборчій дільниці працює кілька регістраторів, до кожного з яких надходить свій потік виборців. Зареєструвавшись, виборець стає у чергу до якоїсь кабінки, потім заходить до неї і вибирає свого кандидата. Після цього він опускає бюлетень до однієї з урн.

### Завдання 7. Моделювання роботи станції швидкої допомоги, варіант Ambulance

На станції швидкої допомоги бригади лікарів чекають виклику до хворих. Коли надходить заявка, бригада їде до хворого й надає йому допомогу. Виконавши необхідну роботу, бригада повертається на станцію.

### Завдання 8. Моделювання зернозбиральних робіт, варіант Zerno

Сільськогосподарський загін по збиранню зерна складається з декількох зернозбиральних комбайнів й автомобілів для вивезення зерна з поля. Комбайни працюють цілодобово. Коли комбайн намолотить повний бункер зерна, він зупиняється й чекає автомобіля, якщо його ще немає. Коли автомобіль під'їжджає, зерно перевантажується в кузов автомобіля, і комбайн продовжує роботу. Автомобіль відвозить зерно на елеватор, чекає розвантаження, потім повертається й стає в чергу на завантаження зерном від комбайнів.

### Завдання 9. Моделювання роботи кар'єру, варіант Ruda

У кар'єрі самоскиди доставляють руду від екскаватора до каменедробарки й там вивантажують руду на площадку. Для завантаження дробарки із площадки використається навантажувач.

### Завдання 10. Моделювання роботи відділу технічного контролю, варіант TestTV

На заключній стадії виробництва телевізорів здійснюється їх перевірка. Якщо під час перевірки виявилося, що телевізор працює неправильно, то він направляється в пункт налаштування. Після налаштування телевізор знову направляється в пункт контролю для перевірки. Телевізори, які пройшли пе-

ревірку, направляються в цех пакування.

### Завдання 11. Моделювання роботи супермаркету, варіант SuperMarket

Покупці приходять у магазин через випадкові проміжки часу. У магазині покупець може зробити кілька покупок. Покупець розраховується за покупки в касах на виході. Час розрахунку на касі й час перебування покупця в магазині залежить від кількості покупок.

### Завдання 12. Моделювання роботи потокової лінії, варіант Line

На потокову лінію, де виконуються послідовно дві операції (відповідно є два робочих місця), оброблювані деталі надходять через випадкові інтервали часу. Оброблювані деталі громіздкі, тому кількість деталей, що можна розмістити перед робочими місцями обмежена. Якщо на площадці до другого робочого місця немає вільних місць, то перше робоче місце блокується. Якщо немає вільних місць перед першим робочим місцем, чергова деталь передається на склад без обробки.

### Завдання 13. Моделювання роботи технічного ракетного дивізіону, варіант Raketa

У технічному дивізіоні ракетного полку складаються й перевіряються ракети перед відправленням у вогневий дивізіон. На складання ракети витрачається якийсь час, а потім ракета перевіряється послідовно на 2-х іспитових стендах. Перед кожним стендом є площадка для розміщення ракет, розмір якої обмежений. Якщо на площадці перед першим стендом немає місця для нової ракети, що надійшла з ділянки складання, вона відправляється без перевірки. Якщо немає місця на площадці перед другим стендом, то робота на першому стенді припиняється.

### Завдання 14. Моделювання обслуговування ветеранів варіант Veteran

9 травня 2011 року ветеранам, що зареєстровані у місті Києві, після урочистого мітингу на Майдані Незалежності видавали продуктові пакети з гречкою. Спочатку ветеран мав показати посвідчення і отримати талон, потім отримував пакет. Внаслідок поганої організації цього процесу у пунктах видачі талонів та пакетів створилися великі черги. Деяким ветеранам довелося чекати пакету майже годину. Президент, дізнавшись про це, наказав у наступному році підготуватися до видачі пакетів краще і створити імітаційну модель, яка дозволить промоделювати цей процес и визначити потрібну кількість обслуговуючого персоналу в залежності від кількості ветеранів.

Можливо у наступному році ветеранів у автобусах, після отримання пакетів будуть відвозити до палацу «Україна» на концерт. Це теж треба передбачити у моделі.

# 2 ВИМОГИ ДО РОБОТИ

## 2.1 Створення команди та розподіл обов'язків

Особливість роботи полягає у тому, що вона має бути розроблена командою. Студенти мають об'єднатися у команди по 2 особи. Кожна команда обирає із свого складу керівника (team leader). Розподіл ролей і зон відповідальності між членами команди орієнтовно може бути таким:

– керівник команди організує роботу команди, розробляє технічне завдання, визначає об'єкти, що мають входити до моделі, розподіляє роботу. За звичай керівник реалізує правила дії активних об'єктів і створює класи для них.

– дизайнер розробляє графічний інтерфейс користувача та пов'язує його з моделлю. Допомагає керівникові у створенні деяких класів. Він же виконує тестування проекту і оформлення звіту.

Робота має виконуватися на протязі останніх чотирьох тижнів другого семестру. У цей час закінчується вивчення відповідних тем курсу.

Виконання роботи починається з детального аналізу завдання, визначаються абстракції предметної області, формується перелік класів, що мають бути створені та складається графік виконання робіт з переліком контрольних точок (milestones).

Команда має періодично проводити збори. На зборах кожний член команди доповідає про виконану роботу і про проблеми, що виникли. Команда обговорює стан справ і приймає необхідні рішення. Стан справ по роботі в цілому і по кожному виконавцю, а також проблеми, що виникли і прийняті рішення обговорюються з викладачем під час консультацій.

## 2.2 Обов'язкові розділи пояснювальної записки

Пояснювальна записка до роботи створюється відповідно до вимог кафедри викладених у методичних вказівках [1] (СОККР-2002). Наведені нижче пункти можна вважати доповненням до вимог СОККР-2002.

### *2.2.1 Технічне завдання*

У технічному завданні має бути наведено опис системи, для якої створюється модель.

Обумовлені спрощення та обмеження, що будуть прийняті при створенні моделі.

Докладно викладені цілі моделювання, які команда має визначити сама, виходячи з опису системи та обраного рівня складності проекту.

Визначено, які компоненти мають бути передбачені у проекті та вимоги до інтерфейсу користувача.

### *2.2.2 Аналіз системи що підлягає моделюванню*

У цьому розділі слід перш за все навести схематичне зображення пред-

метної області. Це можна зробити у вигляді діаграми бізнес процесів, як це зроблено у прикладі, який наведено у додатку А, але можна використовувати будь який спосіб зображення.

Розділ має складатися з двох підрозділів.

Перший підрозділ має бути зорієнтований на виділення абстракцій системи.

У другому підрозділі слід проаналізувати поведінку активних абстракцій

#### 2.2.2.1 Виділення основних абстракцій системи

У цьому розділі слід провести аналіз системи, що підлягає моделюванню, і визначити так звані абстракції, на основі яких будуть створені класи, необхідні для побудови моделі. Поняття «абстракція» передбачає, що при розгляді якоїсь частини реальної системи ми беремо до уваги тільки ті її властивості, які мають значення для вирішення поставленого завдання. Інколи може бути і так, що абстракція не відповідає жодній частині реальної системи.

Результатом цього етапу має бути перелік абстракцій. Доцільно це зробити у вигляді таблиці, де навести назви абстракцій та перелік завдань, що вони вирішують.

#### 2.2.2.2 Аналіз поведінки активних об'єктів системи

У цьому підрозділі слід проаналізувати поведінку абстракцій та представити її у вигляді схем алгоритмів, або діаграм діяльності.

### *2.2.3 Реалізація системи*

У цьому розділі слід докладно представити результати реалізації у складових частин проекту. Кожний з шарів має бути описаний у окремому підрозділі.

#### 2.2.3.1 Реалізація візуальної частини проекту

У цьому розділі пояснювальної записки слід дати зображення інтерфейсу користувача і пояснити призначення його складових частин.

Далі слід перелічити вимоги до окремих компонентів та зв'язків між ними. При необхідності, дати перелік подій компоненту, що будуть оброблятися. Слід також обґрунтувати вибір менеджерів компоновки, що мають бути налаштовані у візуальній частині.

Обов'язково потрібно надати перелік публічних методів шару подання, що будуть надавати доступ до компонентів візуальної частини.

Слід також навести методи запуску процесу моделювання у запроектованих режимах роботи, а текст усього класу наводити не слід.

#### 2.2.3.2 Реалізація класів моделі

Тут у окремих підрозділах слід охарактеризувати класи для компонент моделі, що створювалися. Якщо тексти класів добре прокоментовані, то їх можна наводити цілком. У іншому випадку слід окремо навести перелік полів,

конструктор та методи і дати пояснення до них.

### *2.2.4 Результати тестування програми*

У цьому розділі слід докладно представити результати тестування програми. Тестування проводиться з метою з'ясування працездатності моделі, адекватної реакції на зміну налаштувань та підтвердження можливості її використання для дослідження реальної системи.

Слід вибрати налаштування моделі, які б могли відповідати реаліям системи.

У цьому розділі має бути достатня кількість копі екранів із результатами роботи застосування у різних режимах.

Слід також навести фрагменти протоколу для різних варіантів роботи системи.

# 3 ТЕОРЕТИЧНІ ВІДОМОСТІ

## 3.1 Багатопоточність

Головна мета даної розрахунково-графічної роботи полягає у тому, щоб студенти отримали навички реалізації багатопоточних задач.

Багатопоточнсть має на увазі одночасне виконання декількох программ. Поділом процесорного часу між цими програмами займається операційна система. При цьому слід розрізняти два поняття - потік і процес.

Процес (process) - це об'єкт, який створюється операційною системою, коли користувач запускає додаток. Процесу виділяється окремий адресний простір, причому цей простір фізично недоступний для інших процесів. Процес може працювати з файлами або з каналами зв'язку локальної або глобальної мережі. Коли користувач запускаєте текстовий процесор або програму калькулятора, він створюєте новий процес.

Для кожного процесу операційна система створює один головний потік (thread), який є потоком команд центрального процесора, що виконуються по черзі. При необхідності головний потік може створювати інші потоки, користуючись для цього програмним інтерфейсом операційної системи.

Усі потоки, створені процесом, виконуються в адресному просторі цього процесу і мають доступ до ресурсів процесу. Однак потік одного процесу не має ніякого доступу до ресурсів потоку іншого процесу, бо вони працюють в різних адресних просторах.

Кожному потоку дається певний інтервал часу, протягом якого він перебуває в активному стані. Розподілом часу центрального процесора між потоками займається спеціальний модуль операційної системи - планувальник. Планувальник по черзі передає управління окремим потокам, так що навіть у однопроцессорной системі створюється повна ілюзія паралельної роботи запущених потоків.

Найбільш очевидна область застосування багатопоточності - це програмування інтерфейсів. Потоки незамінні тоді, коли необхідно, щоб графічний інтерфейс продовжував відповідати на дії користувача під час виконання деякої обробки інформації. Наприклад, потік, що відповідає за інтерфейс, може чекати завершення іншого потоку, що завантажує файл з інтернету, і в цей час виводити деяку анімацію або оновлювати прогрес-бар. Крім того, він може зупинити потік завантажує файл, якщо була натиснута кнопка «скасувати ».

Ще одна популярна область застосування багатопоточності - ігри. В іграх різні потоки можуть відповідати за роботу з мережею, анімацію, розрахунки.

Дуже корисна багатопоточність і при розробці програм моделювання систем з паралельними процесами, що характерно для большиство предметних областей. Першу об'єктно орієнтовану мову Симула і було створено для вирішення саме таких задач. Прикладом такого завдання може послужити моделювання роботи операційної системи.

## 3.2 Реалізація багатопоточності у Java

### *3.2.1 Створення потоків у Java*

У Java потік є об'єктом класу java.lang.Thread, який має усе, що необхідно для створення і запуску потоків, а також управління їх станом. Для створення потоків клас Thread надає різні конструктори, за допомогою яких можна реалізувати кілька способів створення потоків. Однак у кожному разі, код, який повинен виконуватися у потоці, повинен бути оформлений у вигляді публічного методу run ( ). Такий метод реалізовано у класі Thread, проте він майже нічого не робить. Можна в цьому переконатися, відкривши його код. Метод, який реально буде виконуватися у потоці, має бути створений в іншому класі.

Є три можливості опису такого методу:

– створити клас на базі класу Thread і у новому класі перевизначити метод run ();

– реалізувати в деякому класі інтерфейс Runnable, що вимагає реалізації методу run (), а потім створити об'єкт класу Thread, передавши через конструктор цей об'єкт як параметр;

– створити дочірній клас на базі класу TimerTask, в ньому реалізувати метод run (), а потім створити об'єкт класу Timer, встановивши час активації завдання.

Другий спосіб особливо зручний у тих випадках, коли клас, метод якого повинен виконуватися в окремому потоці, успадкований від будь-якого іншого класу.

### *3.2.2 Функціональність класу Thread*

За допомогою конструкторів класу Thread, таблиця 3.1, можна створювати потоки різними способами, вказуючи при необхідності для них ім'я та групу. Ім'я призначене для ідентифікації потоку не є обов'язковим атрибутом. Що ж стосується груп, то вони призначені для організації захисту потоків один від одного в рамках однієї програми.

Таблиця 3.1 – Конструктори класу Thread

| Конструктор | Коментарі |
|---|---|
| 1 | 2 |
| public Thread(); | Створення нового об’єкту Thread |
| public Thread(String name); | Створення об’єкта Thread із зазначенням його імені |
| public Thread(Runnable target); | Створення нового об’єкту Thread із використанням об'єкту Runnable |
| public Thread(Runnable target, String name); | Як і попередній, але додатково задається ім'я нового об'єкта Thread |
| public Thread(ThreadGroup group, String name); | Створення нового об’єкта Thread у заданій групі потоків та імени об’єкту |

Продовження таблиці 3.1

| 1 | 2 |
|---|---|
| public Thread(ThreadGroup group, Runnable target); | Створення нового об’єкту Thread з використанням об'єкту Runnable та зазначенням групи потоку. |
| public Thread(ThreadGroup group, Runnable target, String name); | Аналогічно попередньому, але додатково задається ім'я нового об'єкта Thread |

### *3.2.3 Пріоритети потоків в додатках Java*

Якщо процес створив кілька потоків, то всі вони виконуються паралельно, причому час центрального процесора (або декількох центральних процесорів в мультипроцесорних системах) розподіляється між цими потоками.

Кожному потоку дається певний інтервал часу, протягом якого він перебуває в активному стані. Інтервал часу, що виділяється залежить від пріоритету потоку.

У додатках Java можна вказувати три значення для пріоритетів потоків.

За замовчуванням потік має нормальний пріоритет. Якщо інші потоки в системі мають той же самий пріоритет, то всі вони користуються процесорним часом на рівних правах.

У разі необхідності можна підвищити або знизити пріоритет окремих потоків. Потоки з підвищеним пріоритетом виконуються в першу чергу. Потоки а з пониженням пріоритетом теж виконуються, але часу їм виділяється менше.

Для завдання пріоритету потоку використовується метод public final void setPriority (int). Як параметр в цей метод можна передавати константи NORM_PRIORITY, MAX_PRIORITY і MIN_PRIORITY.

### *3.2.4 Методи класу Thread*

У класі Thread реалізовано багато методів і ви можете ознайомитися з ними, відкривши вихідний текст класу. Тут же ми розглянемо тільки деякі з них.

Метод public void start () використовується для запуску потоку. Не слід його плутати з методом run (). Метод run () містить код, який повинен виконуватися в окремому потоці, а метод start () активізує метод run () та забезпечує його виконання в окремому потоці.

Звичайно, можна і просто викликати метод run (), і він теж буде виконуватися, але тільки не в окремому потоці.

Метод public static void sleep (long millis [, int nanos]) призупиняє потік на заданий час. Оскільки як цей метод статичний, то його можна викликати через ім'я класу в будь-якому методі. Наносекунди, що передаються до цього методу округлюютья до мілісекунди.

Метод public final void join () використовується для того, щоб призупинити поточний потік до завершення роботи потоку, якому надіслано повідомлення join (). Тобто join () ніби вставляє інший потік в поточний.

### *3.2.5 Організація взаємодії потоків при використанні спільних ресурсів*

Основна складність, з якою стикаються програмісти, що створюють багатопотокові програми, це організація взаємодії одночасно працюючих потоків.

Однопотокова програма після запуску отримує в монопольне розпорядження всі ресурси комп'ютера і використовує ці ресурси в тій послідовності, яка відповідає логіці роботи програми.

У багатопотокової системі можливі стіуації, коли деякий потік змушений чекати завершення виконання етапу алгоритму іншого потоку. Потоки можуть також намагатися звертатися одночасно до одних і тих же ресурсів, що може призвести до неправильної роботи програми. Спроба одночасної модифікації спільно використовуваних даних може привести до порушення цілісності цих даних.

Таким чином, в багатопотоковому середовищі необхідна синхронізація спільної роботи потоків для забезпечення коректності використання критичних ресурсів.

У Java передбачені різні засоби для вирішення цього завдання.

#### 3.2.5.1 Синхронізація доступу до даних, що використовуються спільно

Для захисту критичних ділянок програми використовується ключове слово synchronized. Використання ключового слова synchronized дає гарантію, що в даний момент часу якийсь оператор або блок буде виконуватися тільки в одному потоці. Інші потоки автоматично припиняються при спробі звернення до ресурсу зайнятому іншим потоком.

Синтаксис створення синхронізованого блоку при використанні критичного ресурсу object наведено нижче:

```
synchronized(object){
// critical operations
}
```

При необхідності можна синхронізувати і метод:

```
 public synchronized void anyMethod(){
   //тело метода
}
```

Синхронізація використовується як засіб блокування потоків, що не дозволяє одному потоку спостерігати об'єкт в проміжному стані, поки той модифікується іншим потоком.

У межах синхронізованою області потрібно виконувати якомога менше роботи:

- заблокувати ресурс;
- перевірити спільно використовувані дані;

– перетворити дані при необхідності;
– розблокувати ресурс.

### 3.2.5.2 Методи wait() і notify()

Метод final void wait () застосовується в тому випадку, коли потрібно змусити потік дочекатися виконання деякої умови.

Цей метод визначений у класі Object, тому його можна викликати для будь-якого об'єкту. Єдина вимога до цього об'єкту полягає у тому, що б цей об'єкт на момент виклику методу wait () був синхронізований. У документації такі об'єкти називають моніторами і потік повинен монопольно володіти цим монітором в момент виклику методу wait () .

Для виведення потоку з призупиненого стану використовується метод notify (), який надсилається тому самому монітору, для якого був викликаний метод wait (). Монітор на момент виклику методу notify () теж повинен бути синхронізований.

Виклик методу wait (), як правило, здійснюється всередині циклу, де перевіряється умова очікування. Якщо умова виконується, то метод wait () не викликається. Саме цикл, а не if, потрібен для забезпечення безпеки.

Існує кілька причин, по яких виконання умови слід перевіряти у циклі:

– за час від моменту, коли один потік викликає метод notify, і до того моменту, коли потік, що очикує, прокинеться, інший потік може встигнути змінити параметри умови.

– інший потік може випадково або умисне викликати notify, коли умову ще не виконано.

– потік, що сповіщає про виконання умови, може викликати notifyAll, хоча умову пробудження виконано лише для одного з потоків, що очікують.

– потік може прокинутися і без сповіщення. Це зветься помилковим пробудженням. У багатьох реалізаціях JVM застосовуються механізми управління потоками, у яких помилкові пробудження інколи трапляються.

Стандартна схема використання методу wait виглядає так:

```
synchronized (obj) {
        try {
          while (< умова не виконана >)
            obj.wait();
        } catch (InterruptedException e) {
          //TODO обробка виключної ситуації
        }

        //Дії, що слід виконувати з obj після виконання умови
}
```

### *3.2.6 Засоби для зупинки потоків*

Зазвичай потік зупиняється після завершення методу run() . Але іноді виникають ситуації, коли необхідно перервати виконання потоку примусово.

Але в Java НЕМА засобів для примусової зупинки потоку. Точніше, є метод stop, але він оголошений deprecated і його використовувати не варто. Справа в тому, що після примусової зупинки потоку (не призупинення!), можуть виникати небажані ситуації. Наприклад, потік може відкрити мережеве з'єднання, а потік зупинять посередині транзакції? Хто її буде закривати ? Хто буде розблокувати ресурси?

Тому в Java прийнятий інформативний порядок зупинки потоку. Якщо поток треба зупинити, використовуют метод interrupt класу Thread. Цей метод не зупиняє потік, а тільки виставляє ознаку, яка свідчить про те, що потік необхідно завершити. Цю ознаку можна перевірити за допомогою методу isInterrupted. Існує також статичний метод interrupted, який перевіряє ознаку для поточного потоку. Причому виклик цього методу скидає ознаку, що має на увазі відповідальність розробника з обробки цієї ситуації.

Метод interrupt здатний вивести потік і зі стану очікування. Тобто якщо у потоку були викликані методи sleep або wait - очікування перерветься і буде викинуто виключення InterruptedException. Ознака у цьому випадку не виставляється.

Для того, щоб корректно зупинити потік, можна періодично викликати метод interrupted. Якщо перевірка спрацювала то треба прийняти рішення - або продовжувати роботу (якщо з якихось причин не можна зупинитися), або треба звільнити ресурси, які використовував потік і вийти з методу run.

Якщо доводиться переривати потік ззовні, то зручніше успадкувати від Thread. За посиланням на екземпляр Thread, легко викликати метод interrupt.

### *3.2.7 Класи Timer і TimerTask*

Ці класи також дозволяють забезпечити виконання коду в окремому потоці, причому, виклик потоку може бути відстрочений на деякий час та / або періодично повторюватися.

Абстрактний клас java.util.TimerTask використовується як батьківський для класів, в яких визначається поведінка потоку, керованого таймером. У цьому класі заявлено інтерфейс Runnable, але метод run () не реалізовано. Цей метод повинен бути реалізований у класі-спадкоємці.

Об'єкт класу java.util.Timer використовується для управління об'єктами класу TimerTask. Клас Timer надає набір конструкторів, які використовуються для створення таймерів з різними характеристиками. З цими конструкторами ви можете познайомитися, відкривши вихідний текст класу. Тут ми згадаємо тільки про один з них - public Timer (boolean isDaemon). Цей конструктор дозволяє встановити потік пов'язаний з таймером, як демон. Потоки демони існують поки існує додаток, з яким вони пов'язані. Завершення додатки закриває і потік демон.

Один таймер може обслуговувати скільки завгодно завдань, які є об'єктами класу TimerTask. Завдання передаються таймеру за допомогою одного з методів schedule(), які відрізняються параметрами. Наприклад, метод public void schedule (TimerTask task, long delay, long period) забезпечує перший запуск завдання task через delay мілісекунд і повторні запуски через кожні period мілісекунд після першого запуску.

Для зупинки виконання завдання можна викликати метод cancel () класу TimerTask.

### *3.2.8 Ще один спосіб створення потоку*

При роботі з GUI для відрисовки компонентів і реакції на всі події компонентів створюється потік диспетчера подій Abstract Window Toolkit (AWT) з ім'ям AWTEventQueue. Для того, щоб забезпечити коректну взаємодію між цим потоком та іншими потоками, що зачіпають GUI, рекомендується створювати потоки, що зачіпають GUI, за допомогою статичного методу invokeLater утілітного класу SwingUtilities. Потік запуститься на виконання після того, як будуть оброблені усі події з черги java.awt.EventQueue.

Ось приклад створення і запуску такого потоку:

```
SwingUtilities.invokeLater(new Runnable(){
  public void run() {
    System.out.println("Hello from SwingUtilities");
  }
});
```

# 4 ПРИКЛАД ПОБУДОВИ СИСТЕМИ

## 4.1 Опис предметної області

Є декілька пристроїв для обробки інформаційних повідомлень (транзакций), наприклад сервери. Ці пристрої отримують транзакції із буфера обмеженного разміру, де транзакції стоять у черзі. Інтервал, між появами транзакцій та час їх обробки випадковий. До буферу транзакциії потрапляють від джерел інформації (наприклад, робочі станції). Якщо буфер заповнено, джерело інформації чекає, поки у буфері зявиться місце. Переміщення транзакції потребує деякого часу, тому можливі випадки, коли буфер не може прийняти транзакцію, він повертає її назад. Кількість оброблених транзакцій та транзакцій, що були відправлені має бути підраховано.

Завданння

Створити графічний імітатор роботи такої системиический имитатор такой системы із музичним супроводом та регістратором подій у системі.

## 4.2 Аналіз системи що підлягає моделюванню

### *4.2.1 Виділення абстракцій системи*

Аналізуючи опис предметної області, технічне завдання та схематичне зображення можна виділити наступні абстракції у системі:

- джерело транзакцій;
- транзакція;
- черга транзакцій на обробку;
- пристрій для обробки транзакцій;
- лічильник транзакцій;
- регістратор подій у системі;
- форма для візуального відображення роботи системи.

Схематичне зображення предметної області наведено на рисунку 4.1.

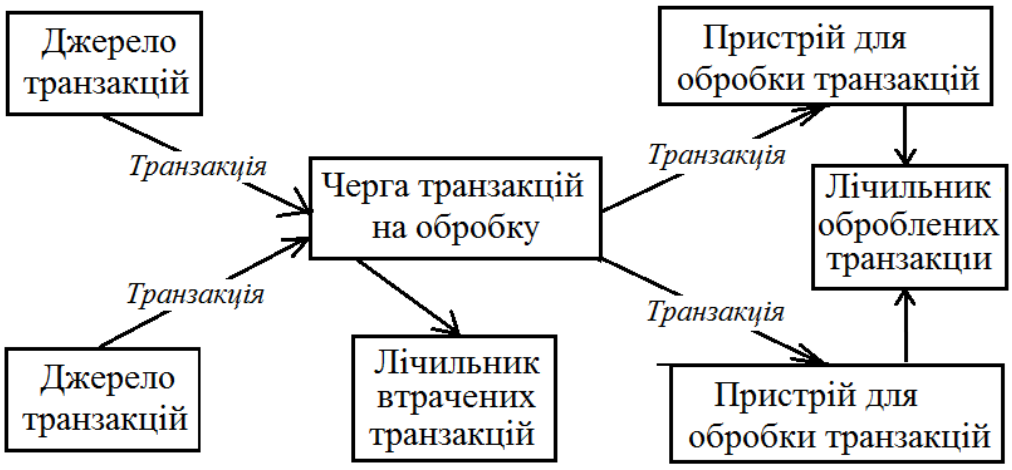


Рисунок 4.1 – Схематичне зображення предметної області

### *4.2.2 Аналіз поведінки абстракцій*

**Джерело транзакцій:** імітує процес створення транзакції і відображає його на формі. Створивши транзакцію, джерело транзакцій аналізує стан черги. Якщо чергу заповнена, об'єкт чекає появи місця в черзі. При появі місця в черзі, об'єкт ініціює процес переміщення транзакції від себе до черги, і чекає завершення цього процесу. Цикл діяльність джерела транзакцій продовжеється, поки не завершиться поток музичного супроводу.

Діаграма діяльності джерела транзакцій наведена на рисунку 4.2.

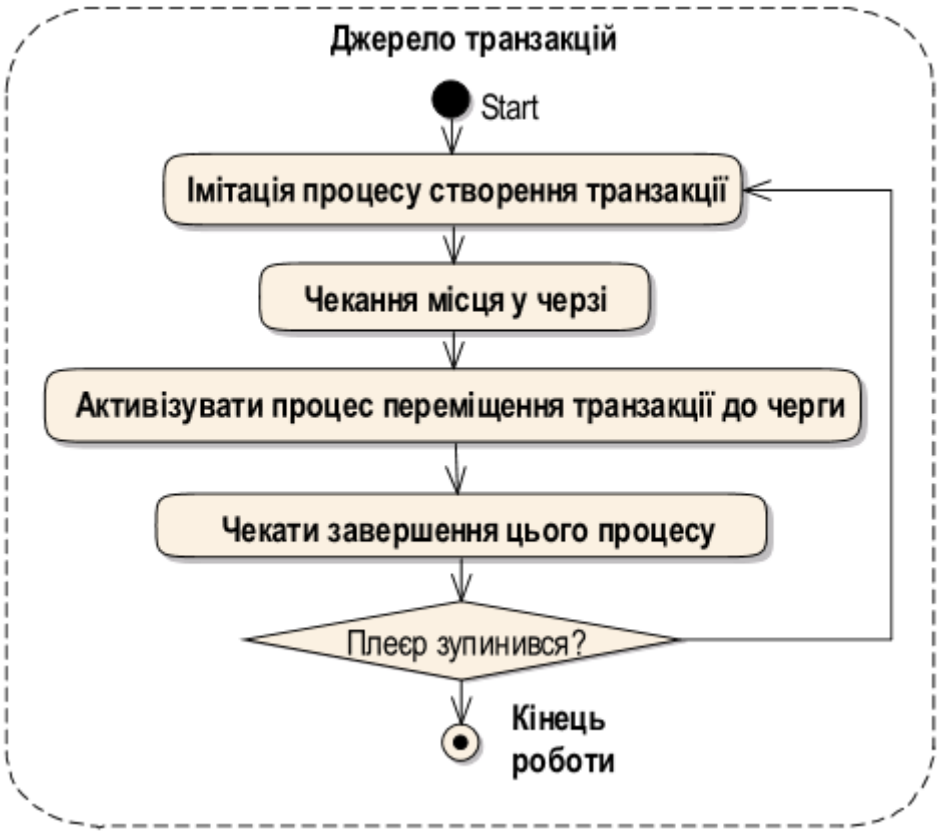


Рисунок 4.2 – Діаграма діяльності джерела транзакцій

**Транзакція:** має переміщатися від однієї абстракції до іншої. Перед початком процесу переміщення транзакція має якимось чином повідомити про це об'єкт, від якого вона починае рухатись, а після закінчення процесу переміщення сповістити про це об'єкт призначення.

**Черга транзакцій на обробку:** приймає, зберігає та віддає транзакції. Має реагувати на повідомлення транзакцій, які приходять до черги. Якщо у черзі немає місць, відправляє транзакцію, що прийшла, до лічильника неприйнятих транзакцій. Інформацію про зміни свого стану джерело транзакцій має надсилати ло регістратора подій у системі

**Пристрій для обробки транзакцій:** аналізує стан черги. Якщо черга пуста, об'єкт чекає появи транзакції в черзі. Коли транзакція зявляється у черзі, ооб’єкт вилучає її з черги і ініціює процес переміщення транзакції від черги до себе та чекає завершення цього процесу. Далі імітує процес обробкт транзакції і

відображає його на формі. Закінчивши обробку транзакції, пристрій ініціює процес переміщення транзакції від себе до лічильника оброблених транзакцій, після цього цикл діяльності пристрою повторюється.

Цикл діяльність продовжеється, поки у черзі є транзакції та активні джерела транзакцій.

Діаграма діяльності пристрою для обробки транзакцій наведена на рисунку 4.3.

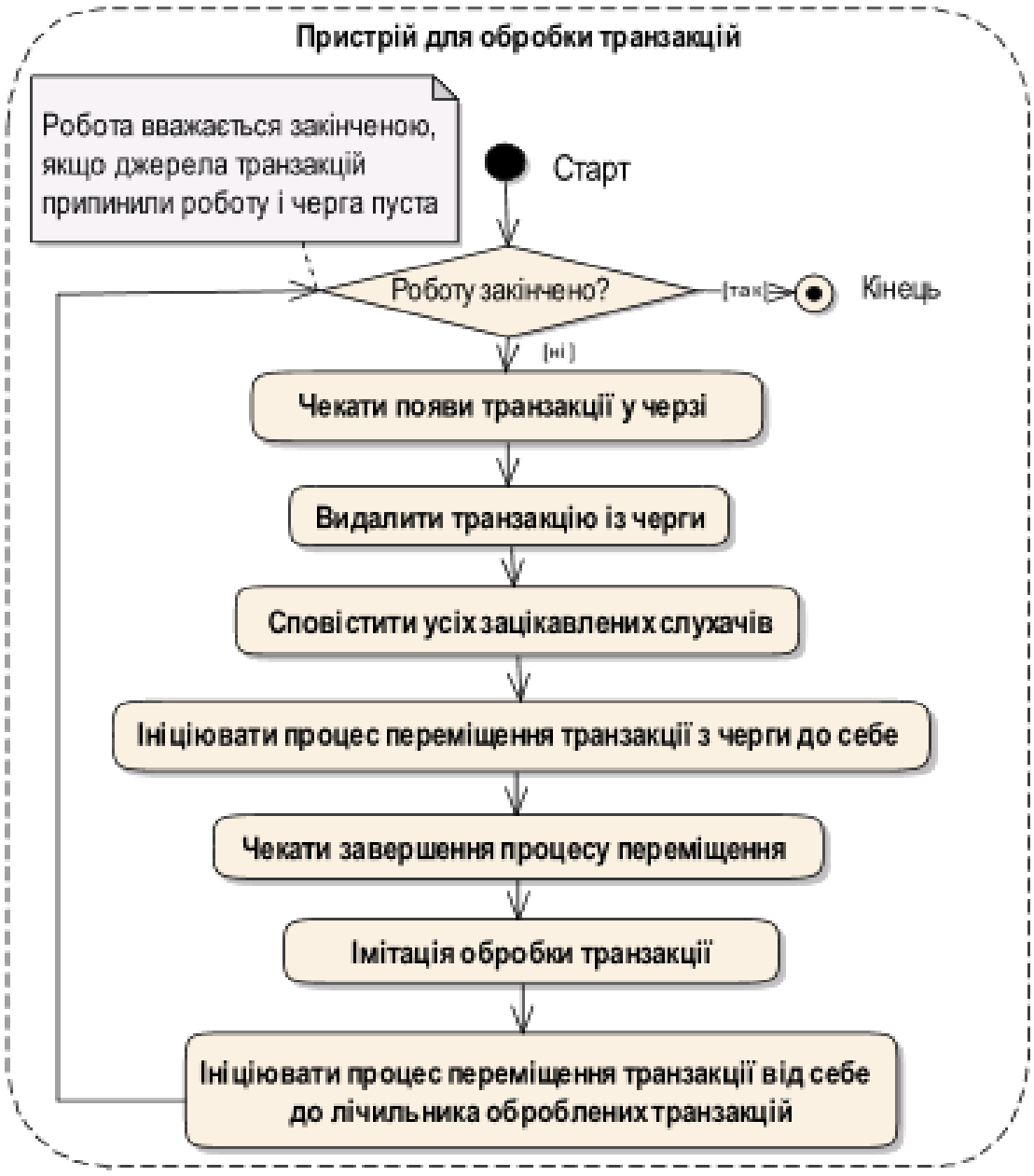


Рисунок 4.3 – Діаграма діяльності пристрою для обробки транзакцій

**Лічильник транзакцій:** має підраховувати кількість транзакцій, що прийшли до нього та відображати це число на візуальному компоненті.

**Регістратор подій у системі:** має приймати інформацію про зміни стану усіх компонент системі та зберігати її. У свою чергу усі компоненти системи

мають надсилати цю інформацію регістратору.

**Форма:** використовується для візуального відображення роботи системи та розташування компонентів управління системою.

## 4.3 Реалізація системи

4.3.1.1 Діаграма класів системи

На рисунку 4.4 наведено діаграму класів проекту.

Клас VisualPart2 реалізує візуальну частину проекту. Цей же клас відповідає за формування протоколу роботи системи.

Клас QueueWihSlider реалізує чергу транзакцій.

Клас Counter використовується для створення лічильників транзакцій.

Клас AbstractWorker є батьківським для класів Creator та Handler, що визначають властивості та поведінку джерела транзакцій та приладу для їх обробки.

Клас Transaction використовується для створення транзакцій.

Клас Player забезпечує музичний супровід. Цей клас входить до складу бібліотеки jlme0.1.3.jar, що додана до buildPath проекту і була знайдена в Інтернет.

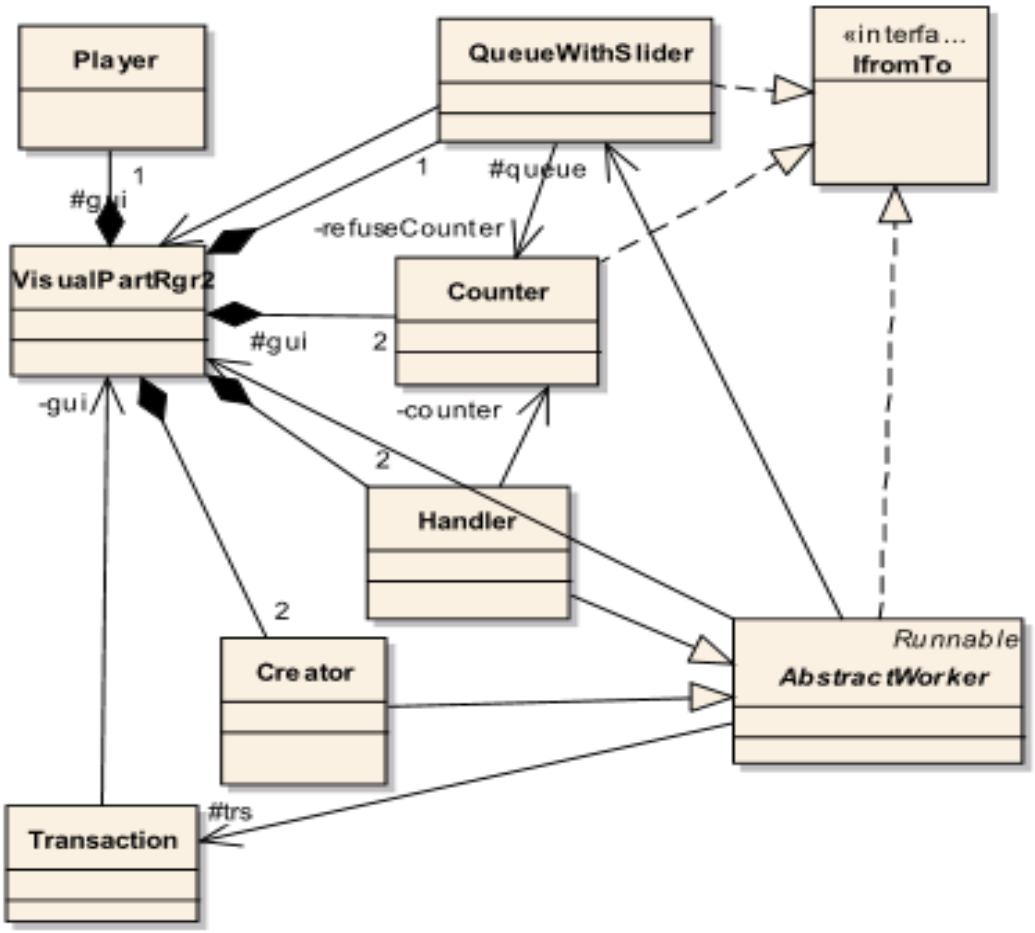


Рисунок 4.4 – Діаграма класів проекту

4.3.1.2 Реалізація візуальної частини проекту

На рисунку 4.5 наведено зовнішній вигляд візуальної частини проекту.

Для відображення двох джерел транзакцій Creator1 та Creator2 використовуються компоненти типу Jlabel, яким шляхом налаштування властивості icon втановлено початкові зображення.

Зображення взято із набору файлів .png, що завантажені у спеціально для цього створений пакет проекту.

Такі само компоненти типу Jlabel використовуються для відображення двох пристроїв обробки транзакцій Handler1 та Handler2.

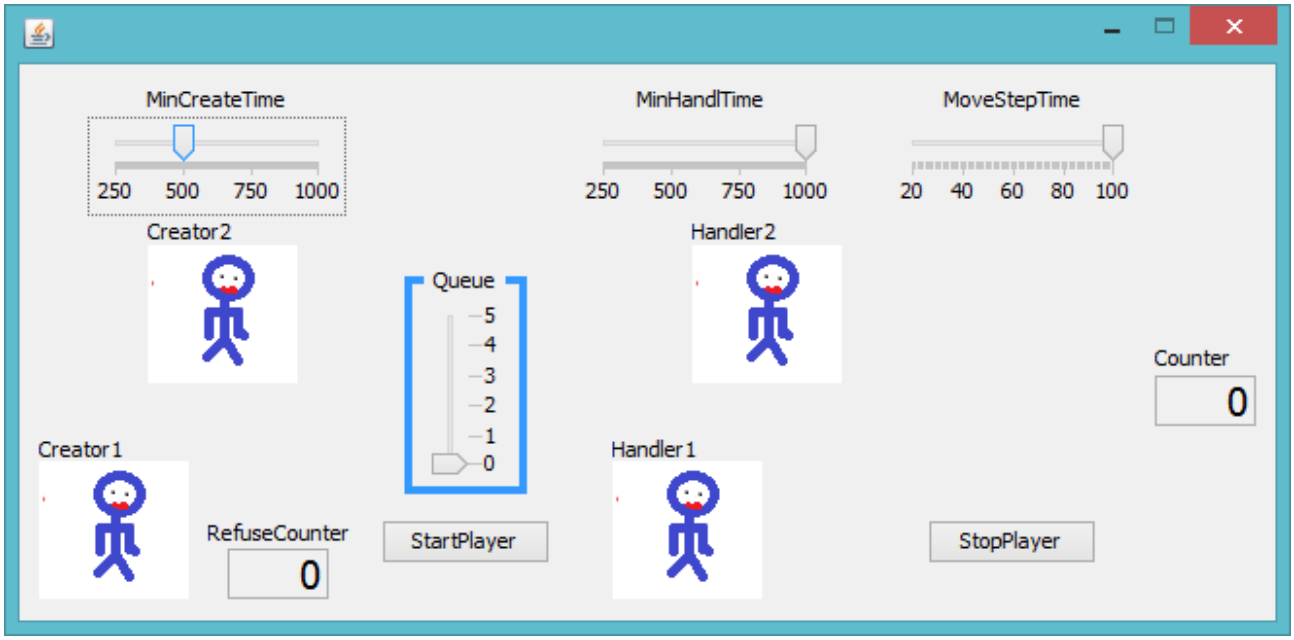


Рисунок 4.5 – Візуальна частина проекту перед стартом

Діяльність об'єктів Creator1, Creator2, Handler1 та Handler2 імітується шляхом зміні файлів .png для властивості icon, які завантажені у спеціально для цього створений пакет проекту. Але цю функцію реалізують самі об'єкти, а не візуальна частина, рисунок 4.6.

Продуктивність об'єктів Creator1, Creator2, Handler1 та Handler2 налаштовується за допомогою компонентів MinCreateTime та MinHandlTime типу JSlider, що задають мінімальний час обробки однієї транзакції у mls. Реальній час обробки кожної транзакції формується самим об'єктом, шляхом додавання до мінімального часу випадкової величини, внаслідок чого реальний час може приймати значення від одного до трьох мінімальних значень.

Для відображення лічильників оброблених транзакцій Counter та втрачених транзакцій RefuseCounter використовуються компоненти типу JTextField.

Черга транзакцій Queue відображується за допомогою компоненту типу JSlider. Цим же компонентом у режимі дизайну налаштовується і місткість черги. На рисунку вона дорівнює 5. Під час роботи проекту сладер показує фак-

тичну завантаженість черги.

Слайдер MoveStepTime час у mls на один шаг переміщення транзакції. Шаг дорівнює розміру транзакції. Але за переміщення транзакцій відповідають самі транзакції а не візуальна частина. А ініціюють ці переміщення джерела транзакцій, пристрої обробки, а також черга, у разі відмови від прийняття транзакції.

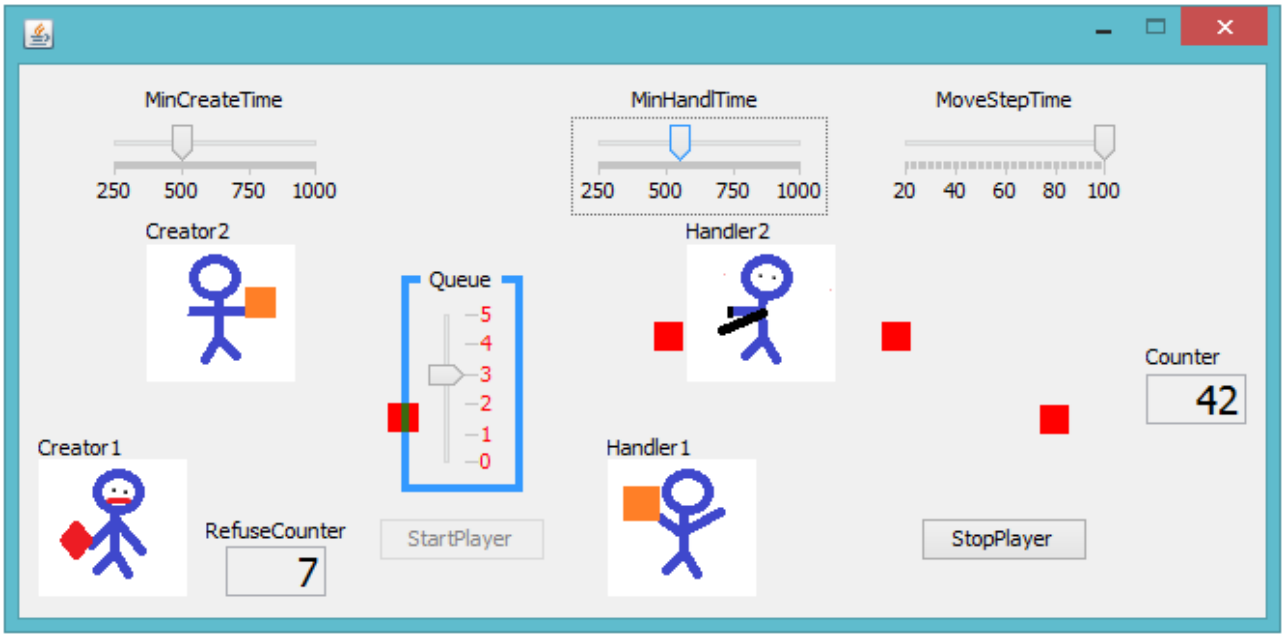


Рисунок 4.6 – Візуальна частина проекту під час роботи

Кнопка StartPlayer викликає приватний метод doRun() візуальної частини. Цей метод створює об'єкти Player, Queue, Creator1, Creator2, Handler1, Handler2, Counter та RefuseCounter за допомогою конструкторів з необхідною кількістю параметрів. Зокрема, ім передаються посилання на візуальні компонентм, що відображкють їх діяльність. Об'єкту Player передається файл .mp3, який знаходиться у тому ж самому пакеті, що й файли зображень. Після цого для об'єктів Player, Creator1, Creator2, Handler1, Handler2 створюються і запускаються потоки, у яких реалізується діяльність цих об'єктів.

Цей метод також блокує кнопку StartPlayer, що унеможливлює повторний запуск потоків реалізації діяльності об'єктів Player, Creator1, Creator2, Handler1, Handler2.

Джерела транзакцій працюють поки є музика. Після завершення потоку виконання музичного супроводу, завершується робота і джерел транзакцій. Після цього завершують роботу і пристрої обробки транзакцій, але тільки після того, як будуть оброблені усі транзакції із черги.

Коли усі потоки завершують роботу, розблоковується кнопка StartPlayer і система готова до повторного запуску.

Кнопка StopPlayer викликає приватний метод doStop() візуальної частини. Цей метод припиняє виконання музичного супроводу, що призводить до завершення роботи усіх потоків.

Діаграма класу VisualPartRgr2, що реалізує візуальну частину проекту, наведена на рисунку 4.7.

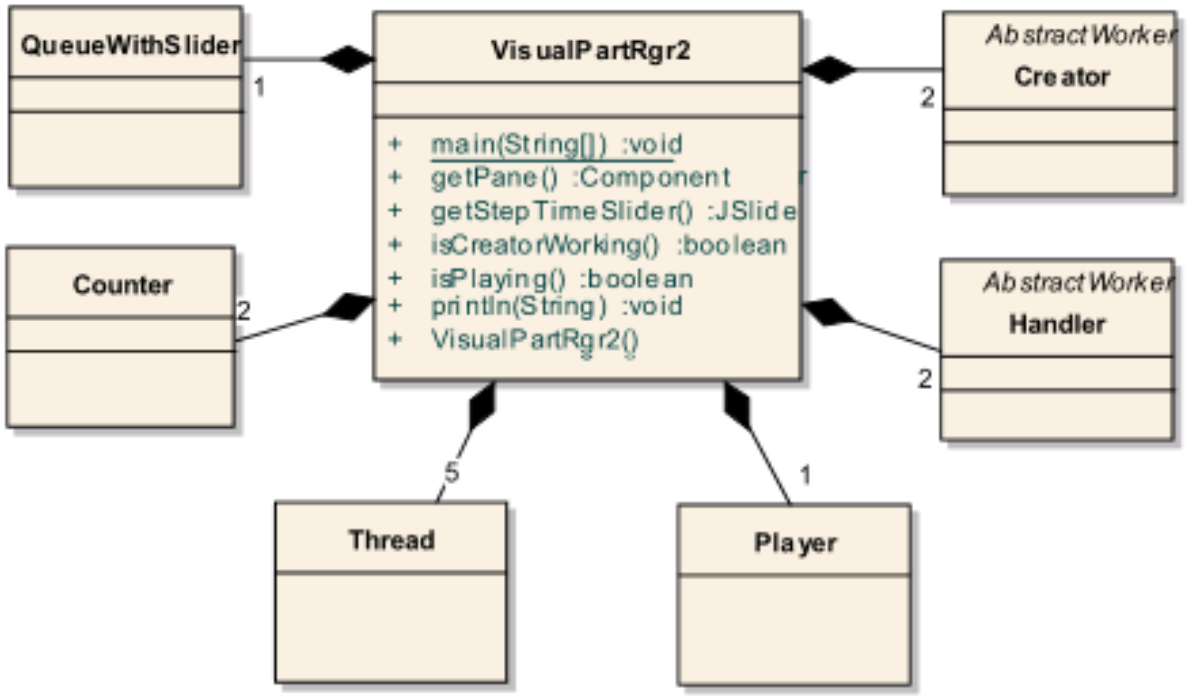


Рисунок 4.7 – Діаграма класу VisualPartRgr2, для візуальної частини проекту

Метод getPane() надає доступ до головної панелі інтерфейсу користувача. Він використовується транзакціями для відображення свого переміщення. Транзакції також використовують метод getStepTimeSlider() для налаштування швидкості переміщення.

Методи isPlaying() та isCreatorWorking() використовуються джерелами транзакцій та приладами обробки ля формування умов закінчення роботи.

Метод println(String) використовується для виведення повідомлень про роботу моделі.

Діаграма 4.7 свідчить про те, що візуальна частина, окрім візуальних компонент має справу з багатьма невізуальним об’єктами. Це одна черга (обїкт класу QueueWithSlider), два джерела транзакцій (об’єкти класу Creator), два прилади обробки транзакцій (об’єкти класу Handler), два лічильника транзакцій (об’єкти класу Counter), плеєр (об’єкт класу Player) та пять потоків (об’єкти класу Thread), у яких виконують свої дії джерела транзакцій, прилади обробки та плеєр.

Робота застосування починається з виконання приватного методу doRun(), який ствоює об’єкти, показані на діаграмі, та запускає потоки. Текст методу наведено нжче.

```
protected void doRun() {
  btnStartPlayer.setEnabled(false);
```

```
    Counter counter = new Counter(textFieldCounter);
    Counter refuseCounter = new Counter(textFieldRefuseCounter);
    QueueWithSlider queue = new QueueWithSlider(this, queueSlider,
        refuseCounter);
    Creator creator1 = new Creator(this, lblCreator1, queue,
        minCreateTimeSlider);
    Creator creator2 = new Creator(this, lblCreator2, queue,
        minCreateTimeSlider);
    Handler handler1 = new Handler(this, lblHandler1, queue,
        minHandlTimeSlider, counter);
    Handler handler2 = new Handler(this, lblHandler2, queue,
        minHandlTimeSlider, counter);
    startTime = System.currentTimeMillis();
    thPlay = playMusic();
    (tc1 = new Thread(creator1)).start();
    (tc2 = new Thread(creator2)).start();
    (th1 = new Thread(handler1)).start();
    (th2 = new Thread(handler2)).start();
  }
```

Для запуску потоку музичного супроводу використовується приватний метод playMusic(), текст якого наведено нижче.

```
    private Thread playMusic() {
        Thread t = new Thread() {
            public void run() {
        try {
                    URL u = getClass().getResource("/other/Osen.mp3");
                    player = new Player(new
                      BufferedInputStream(u.openStream(),2048));
                    player.play();
                    onEndOfPlay();
        } catch (Exception e) {
          e.printStackTrace();
        }
            }
        };
        t.start();
        return t;
    }
```

Після зупинки плеєра викликається приватний метод onEndOfPlay(), який чекає, поки завершать роботу прилади обробки транзакцій, після цього робить доступною кнопку запуску додатку.

```
    private void onEndOfPlay() {
  new Thread() {
    public void run() {
      try {
        th1.join();
```

```
            th2.join();
            btnStartPlayer.setEnabled(true);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }

        }
      }.start();
    }
```

Для примусової зупинки роботи додатку використовується приватний метод doStopPlay(), який зупиняє роботу плеєра.

```
    private void doStopPlay() {
        player.stop();
    }
```

#### 4.3.1.3 Інтерфейс IfromTo

Як видно на рисунку 4.4, транзакції переміщуються від одного компоненту візуальної частини до іншого. З точки зору транзакції усі ці компоненти однакові, бо її цікавлять тільки координати розташування компонентів та їх розміри. Доступ до цієї інформації надають методи, що прописані у класі Component, який успадковують усі візуальні компоненти. Але створюють та обробляють транзакції не візуальні компоненті, а об'єкти різних класів, до складу яких входять посилання на візуальні компоненти.

Для того, щоб дії транзакції не залежали від того, з яким об'єктом вона спілкується, потрібно створити єдиний тип для усіх об'єктів, з якими спілкується транзакція. Це джерела транзакцій, черга, прилади обробки та лічильники.

Роль такого типу буде виконувати інтерфейс IfromTo. До складу інтерфейсу влючимо метод getComponent(), який буде повертати посилання типу Component на візуальні компоненти, що повязані з об'єктами моделі.

Транзакція також має мати можливість повідомляти обектам, з якими вона спілкується, про своє відправлення та прибуття. Щоб транзакція мала можливість реалізувати таку функцію, до складу інтерфейсу IfromTo слід влючити відповідні методі, які транзакція буде викликати у момент відправлення та прибуття.

Інтерфейс IfromTo має такий вигляд:

```
public interface IfromTo {

  /**
   * Метод обработки "события" начало движения транзакции.
   * Вызывается перед началом движения tr.
   * @param tr транзакция, которая уходит от об'єкта
   */
  public void onOut(Transaction tr);

  /**
```

```
   * Метод обработки "события" конец движения транзакции.
   * Вызывается, когда tr приходит к об’єкту.
   * @param tr транзакция, которая пришла к об’єкту
   */
  public void onIn(Transaction tr);

  /**
   *Метод, предоставляющий доступ к компоненту, который представляет
об’єкт на GUI
   * @return ссылку на компонент, который отображает об’єкт на GUI
   */
  public Component getComponent();
}
```

#### 4.3.1.4 Реалізація класу Transaction

Класс Transaction використовується для створення транзакцій, які відображають своє переміщення від одного компоненту до іншого.

Конструктор класу має один параметр – посилання на візуальну частину.

```
    public Transaction(VisualPartRgr2 gui) {
    this.gui = gui;
    this.stepTimeSlider = gui.getStepTimeSlider();
    this.g = gui.getPane().getGraphics();
    // Налаштування коольору графічного контексту
    Color color = Color.RED; //Колір транзакції
    Color back = gui.getPane().getBackground();
    int rgb = back.getRGB() ^ color.getRGB();
    g.setXORMode(new Color(rgb));
  }
```

У конструкторі формується колір графічного контексту, який у режимі XOR забезпечить потрібний колір транзакції та збереження малюнку панелі по якій рухається транзакція.

Для того, щоб ініціювати процес переміщення транзакції, використовується публічний метод moveFromTo(IfromTo, IfromTo). У методі визначаються координати маршруту транзакції, обчислюється кількість необхідних кроків та створюється потік переміщення транзакцію. Посилання на цей поток метод повертає.

Перед початком переміщення транзакця викликає метод onOut(Transaction). Після завершення переміщення транзакця викликає метод onIn(Transaction).

Текст методу наведено нижче:

```
public Thread moveFromTo(final IfromTo from, final IfromTo to) {
  // Создаем поток движения транзакции
  Thread t = new Thread() {
    public void run() {
      //Розміри транзакції
      int hT = 15, wT=15 ;
```

```
      // Параметры маршрута транзакции по X
      int xFrom = pointFrom(from).x;
      int xTo = pointTo(to).x;
      if (xFrom > xTo) {
        // Если движение справа-налево
        xFrom = pointTo(from).x;
        xTo = pointFrom(to).x;
      }
      int lenX = xTo - xFrom;
      // Параметры маршрута транзакции по У
      int yFrom = pointFrom(from).y;
      int yTo = pointTo(to).y;
      int lenY = yTo - yFrom;
      // Длина маршрута
      int len = (int) (Math.round(Math
          .sqrt(lenX * lenX + lenY * lenY)));
      // середній розмір транзакції
      int lenT = (hT + wT) / 2;
      // Число шагов продвижения
      int n = len / lenT + 1;
      // Шаги продвижения
      int dx = lenX / n;
      int dy = lenY / n;
      gui.println("Транзакция начинает перемщаться от "
          + from.getComponent().getName() + " к "
          + to.getComponent().getName());
      // Вызов метода обработки события "отправление"
      from.onOut(Transaction.this);
      // Цикл перемещения
      for (int x = xFrom, y = yFrom, i = 0; i < n; x += dx, y += dy, +) {
        // Рисуем транзакцию
        g.fillRect(x, y, wT, hT);
        try {
          // Задержка
          Thread.sleep(stepTimeSlider.getValue());
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        // Повторный xor возвращает картинку фона
        g.fillRect(x, y, wT, hT);
      }
      // Вызов метода обработки события "прибытие"
      to.onIn(Transaction.this);
    }
  };
  // Запускаем созданный поток движения транзакции
  t.start();
  return t;
}
```

Наведений метод використовує приватний метод pointFrom(IfromTo), який визначає координати середини правої кромки візуального компоненту, та приватний метод pointTo(IfromTo), який визначає координати середини лівої кромки візуального компоненту.

#### 4.3.1.5 Реалізація класу QueueWithSlider

Діаграма класу QueueWithSlide наведена на рисунку 4.8.

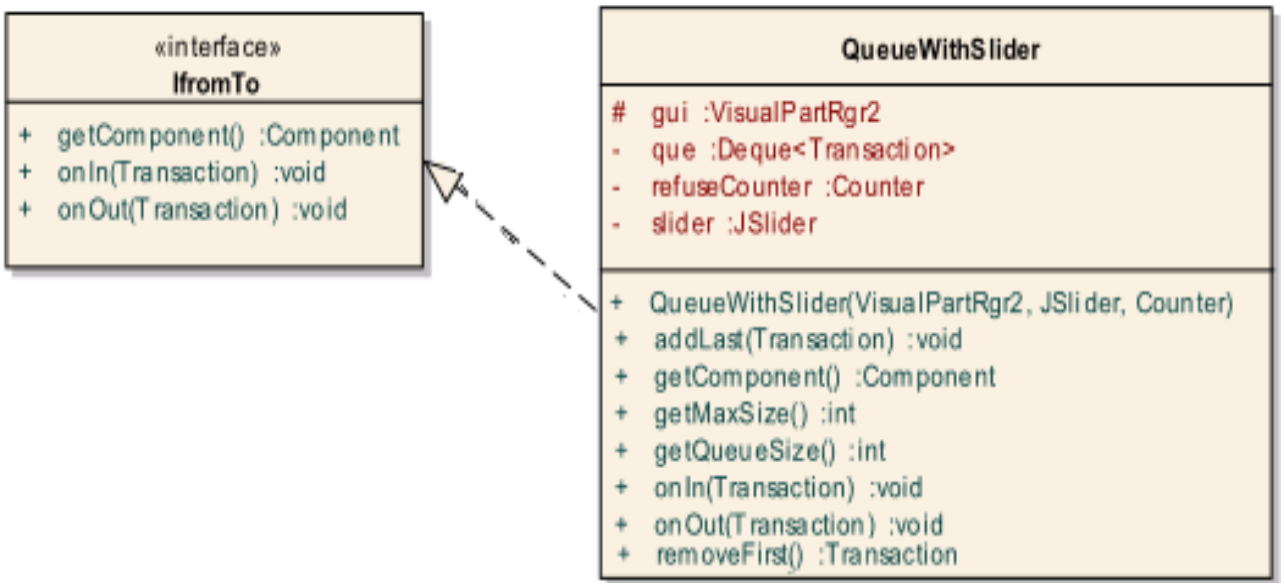


Рисунок 4.8 – Діаграма класу QueueWithSlider

Конструктор класу має три параметри, через які передаються посилання на візуальну частину, на слайдер, який відображує параметри черги та на лчильник неприйнятих транзакцій, куди черга відправляє транзакції, що не можна прийняти.

Для збереження транзакцій у класі використовується об’єкт que типу Deque, який проініціалізовано коллекцією ArrayDeque.

Узвязку з тим, що черга співпрацює з транзакціями, вона має реалізовувати інтерфейс IfromTo. Метод getComponent() повертає посилання на слайдер. Метод onIn() викликається транзакцією, коли вона підійшла до черги. Якщо у черзі є місце, транзакція приймається і зацікавленим об’єктам неадсилається повідомлення про це. Якщо у черзі місця нема, то створюється потік переміщення транзакції від черги до лічильника транзакцій, що не були прийняті чергою.

Текст методу наведено нижче.

```
public void onIn(Transaction tr) {
   synchronized (this) {
     if (getQueueSize() < getMaxSize()) {
       addLast(tr);
         this.notify();
       return;
```

```
            }
        }
        tr.moveFromTo(this, refuseCounter);
    }
```

Метод onOut() черга не використовує.

Методи addFirst(transaction) додає транзакцію до коллекції que, а метод removeFirst() вилучає транзакцію з коллекції. Окрім того ці методи корегують значення value слайдера та виводять повідомлення про зміну розміру черги у протокол за допомогою методу println(String) візуальної частини.

Методи getMaxSize() та getQueueSize() повертають значення максимально допустимого та поточного розміру черги.

#### 4.3.1.6 Реалізація класу Counter

Діаграма класу Counter наведена на рисунку 4.9.

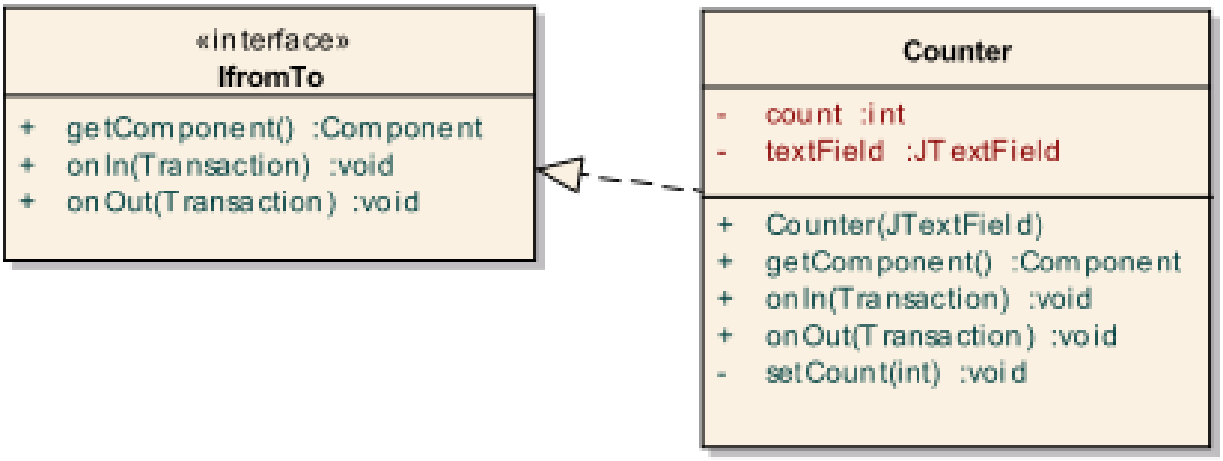


Рисунок 4.9 – Діаграма класу Counter

Цей клас забезпечує підрахунок транзакцій, що надійшли до нього.

Конструктор класу має один параметр, через який передається посилання на компонент типу JTextField, який відображає кількість транзакцій, що надійшло.

Кількість транзакцій зберігається у полі count. Підрахунок реалізується за допомогою методу onIn(Transaction), шляхом інкременту поля count. Одночасно змінюється і текст у компоненті textField.

#### 4.3.1.7 Реалізація класу AbstractWorker

Це базовий клас для об’єктів, дії яких повязані із створенням та обробкою транзакций. Такими класами у нашому проекті є класи Creator та Handler.

Діаграма низки цих класів наведена на рисунку 4.10.

Конструктор класу AbstractWorker має чотири параметри, через які передаються такі посилання:

gui – посилання на візуальну частину;

label – посилання на компонент типуJLabel, у якому відображується діяльність об’єкту;

minWorkTimeSlider – посилання на слайдер, який визначає мінімальний час роботи з транзакцією;

queue – посилання на чергу, з якою працює об’єкт.

У класі заявлено реалізацію інтерфейсу IfromTo, але реалізовано тільки метод getComponent(), який повертає посилання на компонент label. Реалізація методів onIn() та onOut() цього інтерфейсу передбачена у класах спадкоємцях, тому клас обявлено як абстрактний.

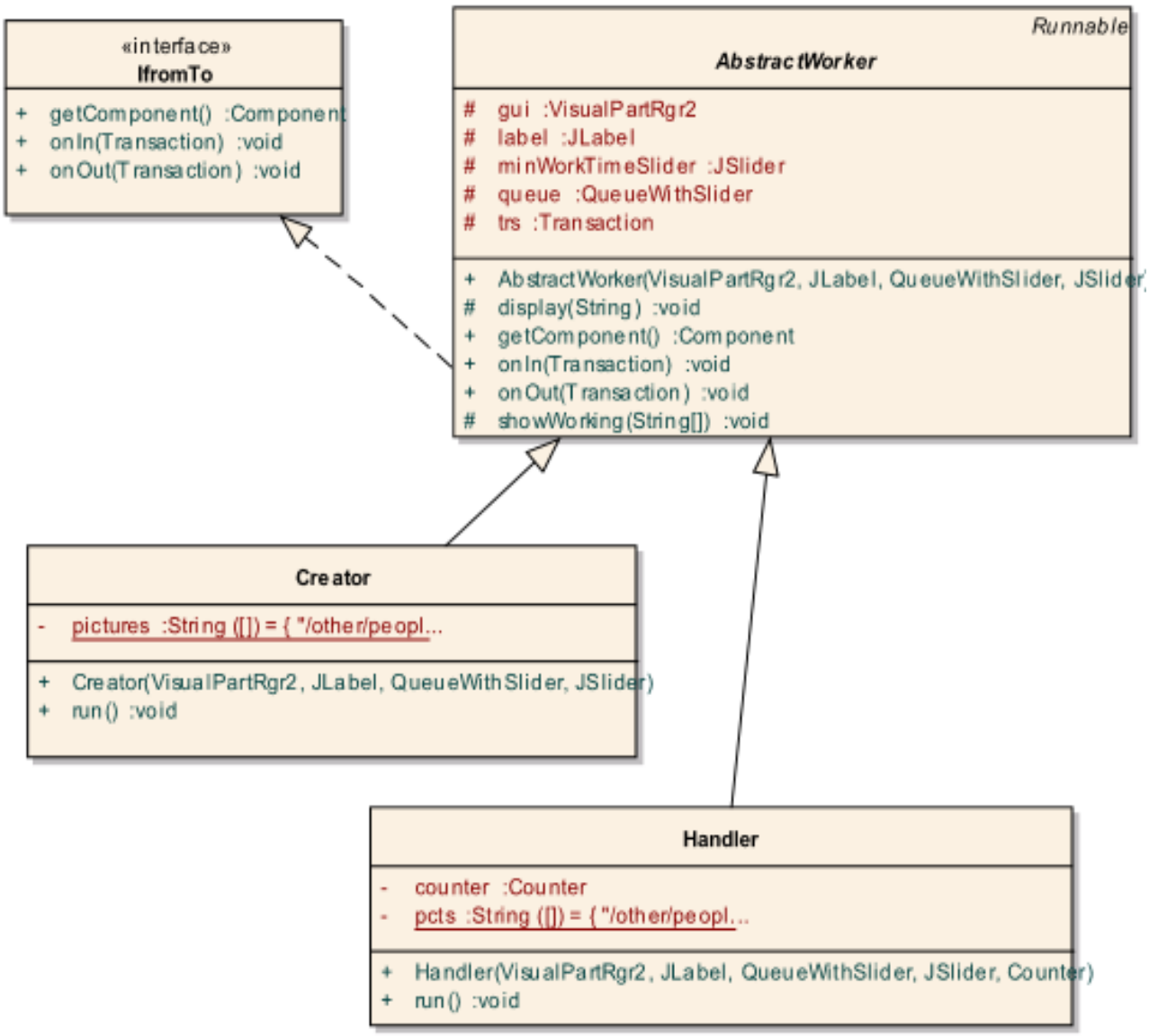


Рисунок 4.10 – Діаграма класів, дії яких повязані із створенням та обробкою транзакций

У класі також заявлено реалізацію інтерфейсу Runnable. Це зроблено тому, що об’єкти класів спадкоємців будуть реалізовувати свою діяльність у окремих потоках. Саме у цих класах і буде реалізовно метод run цього інтерфейсу.

Основним функціональним навантаженням класу AbstractWorker є відображення діяльності об’єкту на візуальній частині у компоненті label. Цей функціонал реалізовано у методі showWorking(String[]). Параметром методу є масив рядків, які містять відносні імена файлів картинок, які по черзі будуть відображатися у компоненті label.

У методі формується час роботи об’єкту шляхом добавки випадкового числа до мінімального значення, яке задає слайдер. Внаслідок цього час роботи може дорівнювати від одного до трьох мінімальних значень. Інтервал між змінами картинок обчислюється шляхом ділення мінімального часу роботи на 10. Таким чином, із збільшенням мінімального часу збільшується і інтервал між змінами кадрів.

Затримка реалізується методом slieep(long) класу java.lang.Thread.

Для виведення одного кадру використовується метод displayString(String), текст якого наведено нижче:

```
protected void display(String pct) {
    //pct – це рядок символів,
    //що визначає місце знаходження ресурсу,
    //наприклад, "/other/peoplWait.png"
    //URL це Uniform Resource Locator -
    //об’єкт, що визначає місце знаходження ресурсу.
    URL u = getClass().getResource(pct);
    ImageIcon image = new ImageIcon(u);
    label.setIcon(image);
}
```

#### 4.3.1.8 Реалізація класу Creator

Діаграма класу Creator вже була наведена на рисунку 4.10.

Конструктор класу дублює конструктор суперкласу.

Поле pictures містить масив, у якому наведено відносні імена файлів картинок, які по черзі будуть відображати діяльність по створенню транзакції.

У методі run розписано порядок діїї об’єкту. Цей метод наведено нижче.

```
/**
 * Метод, що визначає поведінку об’єкту,
 * яка полягає у циклічному повторенні наступних дій:
 * 1.Імітує процес створення транзакції
 *   та відбражає його у компоненті label.
 * 2.Аналізує стан черги, і якщо вона повна
 *   чекає появи місця у черзі.
 * 3.При появі місця у черзі
 *   ініціює процес переміщения транзакціїї від себе до черги.
 * 4.Чекає завершення цього процесу. (Черга прийме транзакію,
 * коли вона дійде до неї, але якщо одночасно працює декілка таких
 * об’єктів, транзакція може до чергиі не попасти, бо за час її руху
 * місце у черзі може заняти інша транзакія)
 * 5.Процес повторюється проки працює плеєр.
 *
 * @see java.lang.Runnable#run()
 */
```

```
  @Override
  public void run() {
    do {
      // Імітує процес створення транзакції
      showWorking(pictures);
      // Створює транзакцію
      trs = new Transaction(gui);
      gui.println(label.getName() + " создал транзакцию " + trs);
      // Цикл перевірки та, можливо, чекання місця у черзі
      synchronized (queue) {
        while (queue.getQueueSize() >= queue.getMaxSize()) {
          try {
            display("/other/peoplWait1.png");
            gui.println(label.getName()
                + " чекає, нема місця у черзі ");
            queue.wait();
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }// кінець блоку synchronized (queue) і циклу чекання
      gui.println(label.getName() + " отправляет транзакцию в очередь");
      // Створює поток переміщення транзакції
      Thread t = trs.moveFromTo(this, queue);
      // Призупиняється на час переміщення транзакції
      display("/other/peopljoin1.png");
      gui.println(label.getName()
          + " чекає, поки транзакція не дойде до черги");
      try {
        t.join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      gui.println("транзакція від " + label.getName()
          + " досягла черги");
    }while (gui.isPlaying());
    // Завершення роботи
    display("/other/peoplWait.png");
    gui.println(label.getName() + " роботу завершив");
  }
}
```

#### 4.3.1.9 Реалізація класу Handler

Діаграма класу Handler вже була наведена на рисунку 4.10.

Конструктор класу майже дублює конструктор суперкласу, але має одне додаткове поле, через яке передається посилання на лічильник оброблених транзакцій counter.

Поле pictures містить масив, у якому наведено відносні імена файлів картинок, які по черзі будуть відображати діяльність по створенню транзакції.

У методі run розписано порядок діїї об’єкту. Цей метод наведено нижче.

```
/**
 * Метод, що визначає поведінку об’єкту,
 * яка полягає у циклічному повторенні наступних дій:
 * 1.Якщо черга пуста, об’єкт чекає появи транзакції.
 * 2.Коли транзація зявляється у черзі, об’єкт звбирає її
 *   та сповіщає про це усі зацікавлені об’єкты.
 * 3.Ініціює процес переміщения транзакціїї від черги до себе.
 * 4. Чекає завершення цього процесу.
 * 5.Імітує процес обробки транзакції
 *   та відбражає його у компоненті label.
 * 6.Ініціює процес переміщения транзакціїї від себе до лічильника.
 *
 * @see java.lang.Runnable#run()
 */
@Override
public void run() {
  while (gui.isCreatorWorking() || queue.getQueueSize() > 0) {
    // Цикл перевірки та, можливо, чекання транзакції у черзі
    synchronized (queue) {
      while (queue.getQueueSize() <= 0) {
        display("/other/peoplWait.png");
        gui.println(label.getName()
            + " чекає появи транзакції у черзі");
        try {
          queue.wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }// кінець цикду чекання
      gui.println(label.getName()
          + " влучає транзакцію із черги");
      trs = (Transaction) queue.removeFirst();
      queue.notify();
    }// конец блока synchronized (queue)
    // Створює поток переміщення транзакції до себе
    Thread t = trs.moveFromTo(queue, this);
    display("/other/peopljoin.png");
    //Призупиняється на час переміщення транзакції
    try {
      gui.println(label.getName()
          + " чекає, поки транзакція дійде до нього");
      t.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    // Імітує процес обробки транзакції
    showWorking(pcts);
    // Створює поток переміщення транзакції від себе до лічильник
```

```
      trs.moveFromTo(this, counter);
    }
    //Завершення роботи
    display("/other/peoplWait.png");
    gui.println(label.getName() + " роботу закінчив");
}
```

## Рекомендована література


1. Бадд Т. Объектно-ориентированное программирование в действии. Санкт-Петербург: Питер, 1997.
2. Буч Г. Объектно-ориентированное проектирование с примерами применения: Пер. с англ. – М.:Кокорд,1992. – 519с
3. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++: Пер. с англ. – М.:Диалект,1999
4. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. – СПб.:Питер,2001
5. Фаулер Мартин Архитектура корпоративных программных приложений. СПб.: Издательский дом «Вильямс», 2007. – 544с
6. Simon Kendal. Object oriented programming using Java. Ventus Publishing ApS-, 2009. – 209 с.
7. http://omg.org Object Management Group
8. \\kid\incoming\byvoino\4_курс\Simulation