Київський національний університет

імені Тараса Шевченка

Кулябко П.П.

Проблеми рефакторінгу

Частина 1.

Навчально-методичний посібник

Київ

2022

УДК 681.3

Кулябко П.П. Проблеми рефакторінгу. Частина 1:

Навчальний посібник для студентів факультету комп’ютерних наук та кібернетики. –

Київ. – 2022. – 56 с.

Затверджено вченою радою

факультету

комп'ютерних наук та кібернетики,

протокол № 12 від 30 серпня 2022 р.

Поліпшення проекту існуючого коду.

Поняття рефакторінгу виникло у колах, близьких до Smalltalk, але дуже швидко перейшло до інших мов програмування. Оскільки оптимізація є невід'ємною частиною розвитку програмного забезпечення, то цей термін з'являється, як тільки проектувальники починають вести професійні бесіди. Він потрібен при уточненні ієрархії класів, а також при обговоренні кількості рядків коду для вилучення. Рефакторінг є ключем для зручності читання та змінюваності коду – як загалом для програмного забезпечення, так і для окремої програми.

У чому ж проблема? Рефакторінг – це ризик. Він вимагає зміни робочого коду, при цьому можуть виникати не тільки поліпшення, але і нові помилки. Недбало виконаний рефакторінг може відкинути розробку назад на дні і навіть тижні. Далі розглянемо основні принципи та передовий досвід рефакторінгу.

Рефакторінг – це процес змін програмного проекту, при якому зовнішня поведінка залишається незмінною, а внутрішня структура поліпшується. Це систематизований спосіб очищення коду для мінімізації можливості появи нових помилок.

Приклади рефакторінгу пропонуються на мові Java.

1. Перший приклад рефакторінгу.

Програма прикладу дуже проста. Вона розраховує і виводить звіт про покупки клієнта у відеопрокаті. Програма отримує інформацію про те, які фільми брав клієнт і на який термін. Далі вона розраховує вартість прокату, виходячи з тривалості і типу фільму. Фільми можуть бути 3-х типів: звичайні, дитячі та новинки. Окрім розрахунку вартості, нараховуються бонуси, у залежності від того, чи є фільм новинкою.

Movie		Rental		Customer
priceCode: int	1 ← *	daysRented: int	* ← 1	
				statement()

```
public class Movie {
```

```
        public static int CHILDRENS  = 2;
        public static int REGULAR = 0;
        public static int NEW_RELEASE = 1;

        private String _title;
        private int _priceCode;

        public Movie(String title, int priceCode) {
                _title = title;
                _priceCode = priceCode;
        }

        public int getPriceCode() {
                return _priceCode;
        }

        public void setPriceCode(int arg)  {
                _priceCode = arg;
        }

        public String getTitle() {
                return _title;
        }
}

class Rental {
        private Movie _movie;
        private int _daysRented;
        public Rental(Movie movie, int daysRented) {
```

```
            _movie = movie;
            _daysRented = daysRented;
        }
        public int getDaysRented() {
            return _daysRented;
        }
        public Movie getMovie() {
            return _movie;
        }
}

class Customer {
        private String _name;
        private Vector _rentals = new Vector();
        public Customer(String name) {
            _name = name;
        }
        public void addRental(Rental arg){
            _rentals.addElement(arg);
        }
        public String getName() {
            return _name;
        }
        public String statement() {
            double totalAmount = 0;
            int frequentRenterPoints = 0;
            Enumeration rentals = _rentals.element();
            String result = “Rental “ + getName() + “\n”;
            while (rentals.hasMoreElements() {
                double thisAmount = 0;
```

```
                Rental each = (Rental) rentals.nextElement();
            // calculate sum for each line
                switch (each.getMovie().getPriceCode()) {
                    case Movie.REGULAR:
                        thisAmount += 2;
                        if (each.getDaysRented() > 2)
                            thisAmount += (each. getDaysRented() – 2) * 1.5;
                        break;
                    case Movie.NEW_RELEASE:
                        thisAmount += each. getDaysRented() * 3;
                        break;
                    case Movie.CHILDRENS:
                        thisAmount += 1.5;
                        if (each.getDaysRented() > 3)
                            thisAmount += (each. getDaysRented() – 3) * 1.5;
                        break;
                }
        // plus bonus
                frequentRenterPoints ++;
        // bonus for 2 days rental
                if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                    each.getDaysRented() > 1) frequentRenterPoints ++;
        // output results for each rental
                result += “\t” + each.getMovie().getTitle() + ”\t” + String.valueOf(thisAmount) +
“\n”;
        //colontitle
        result += “indebtedness sum: “ + String.valueOf(totalAmount) + “\n”;
        result += “you earned “ +  String.valueOf(frequentRenterPoints) + “ loyalty points”;
        return result;
}
```

Недоліки: надто довгий метод statement() виконує надто багато дій, Значну частину з того, що він робить, бажано, щоб виконували методи інших класів.

Можливо від користувачів надійде бажання виводу звіту в HTML для публікації на сайті Web. Метод statement() повторно використати неможливо, але новий переписаний метод htmlStatement буде у багатьох випадках копіювати попередню поведінку. А якщо зміняться правила оплати, то доведеться вносити *узгоджені* зміни в обидва методи. Можливі зміни в класифікації фільмів, оплаті прокату та нарахуванні бонусів.

Коли з'ясовується, що в програму треба ввести нову функціональність, а код програми не структурований так, щоб було зручно вносити цю функціональність, то спочатку бажано виконати рефакторінг для спрощення внесення змін і тільки потім починати самі зміни.

Потрібен набір надійних тестів.

Оскільки метод statement() повертає рядок, то потрібно створити кілька клієнтів, які отримують напрокат кілька типів фільмів та генерують рядки звітів. Далі треба порівняти отримані рядки з контрольними, які перевірені вручну. Краще організувати тести так, щоб їх можна було виконувати однією командою з командного рядка. При цьому виконання тестів повинно займати мало часу, бо вони будуть запускатися дуже часто. Важливим є спосіб виводу результатів. Вони виводять або «ОК», або список помилок, тобто рядків, які відрізняються від контрольних. Таким чином, тести *самі перевіряють* свої результати. Якщо при рефакторінгу буде допущена помилка, то тести тут же про це повідомлять.

Декомпозиція і перерозподіл метода statement()

Довгі методи бажано декомпонувати, бо невеликими фрагментами легше керувати і переносити. Окрім того, їх легше розуміти.

Першим кандидатом на застосування рефакторінгу є оператор switch. Спочатку треба перевірити, чи немає у виділеному фрагменті локальних змінних і параметрів і проаналізувати. У фрагменті дві локальних змінних each і thisAmount. Змінна each цим фрагментом не змінюється на відміну від thisAmount. Всі змінні, які не змінюються, можна передавати як параметри. Стосовно змінних, які модифікуються, то тут ситуація складніша. Якщо така змінна одна, то метод може її повернути. Змінна ініціалізується 0 для кожної

ітерації циклу і не змінюється поки не попаде в оператор switch, тому значення цієї змінної можна просто повернути.

Після рефакторінгу отримаємо

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.element();
        String result = “Rental “ + getName() + “\n”;
        while (rentals.hasMoreElements() {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();
        thisAmount = amountFor(each);
    // plus bonus
        frequentRenterPoints ++;
    // bonus for 2 days rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1) frequentRenterPoints ++;
    // output results for each rental
        result += “\t” + each.getMovie().getTitle() + ”\t” + String.valueOf(thisAmount) +
“\n”;
    //colontitle
    result += “indebtedness sum: “ + String.valueOf(totalAmount) + “\n”;
    result += “you earned “ +  String.valueOf(frequentRenterPoints) + “ loyalty points”;
    return result;
}

private double amountFor(Rental each) {
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
```

```
                    case Movie.REGULAR:
                        thisAmount += 2;
                        if (each.getDaysRented() > 2)
                            thisAmount += (each. getDaysRented() – 2) * 1.5;
                        break;
                    case Movie.NEW_RELEASE:
                        thisAmount += each. getDaysRented() * 3;
                        break;
                    case Movie.CHILDRENS:
                        thisAmount += 1.5;
                        if (each.getDaysRented() > 3)
                            thisAmount += (each. getDaysRented() – 3) * 1.5;
                        break;
            }
            return thisAmount;
    }
```

Тепер можна подумати про зміну назв деяких змінних. Замінимо each на aRental, a thisAmount – result. Це елемент само документування коду.

Написати код, який є зрозумілим компілятору, багато хто може, а от написати код, який є зрозумілим іншим людям, це може лише дехто.

Перенос методу розрахунку суми оплати

В методі amountFor використовується інформація з класу Rental, а не з класу Customer. Це наводить на думку про те, що метод знаходиться в непідходящому класі. Краще, коли метод знаходиться в класі, данні якого він використовує, тому є сенс перемістити amountFor в клас Rental. Для цього використовується рефакторінг *перенос методу*. При цьому спочатку весь метод копіюється в клас Rental і відповідним чином змінюється для коректної роботи на новому місці, а далі компіляція.

```
class Rental { …
    double getCharge() {
```

```
        double result = 0;
            switch (getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    result += 2;
                    if (getDaysRented() > 2)
                        result += (each. getDaysRented() – 2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    result += each. getDaysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    result += 1.5;
                    if (getDaysRented() > 3)
                        result += (each. getDaysRented() – 3) * 1.5;
                    break;
        }
        return result;
    }
```

У цьому випадку зміни для роботи у новому місце розташуванні означають вилучення параметра. Крім того, метод був перейменований.

Тепер тестування і модифікація методу Customer.amountFor, компіляція.

```
class Customer …
    private double amountFor(Rental aRental) {
        return aRental.getCharge();
    }
```

Наступний крок полягає у пошуку всіх звернень до старого методу та їх заміні на звернення до нового методу. Це рядок thisAmount =

amountFor(each). Виклик здійснюється тільки в одному місці, тому у даному випадку – просто. Але в загальному випадку потрібно виконати пошук по всім класам, які можуть використовувати цей метод. Виконуємо заміну рядка на thisAmount = each.getCharge().

Після внесення змін слід вилучити старий метод. Компіляція, тестування. Інколи старий метод можна зберегти для делегування завдання новому методу. Це може бути корисно, якщо метод оголошений відкритим і не бажано змінювати інтерфейс іншого класу.

Повернемось до Customer.statement. Легко бачити непотрібність змінної thisAmount, їй присвоюється значення, яке потім не змінюється, тому можна вилучити thisAmount, застосувавши рефакторінг «Заміна тимчасової змінної запитом».

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.element();
        String result = “Rental “ + getName() + “\n”;
        while (rentals.hasMoreElements() {
                double thisAmount = 0;
                Rental each = (Rental) rentals.nextElement();
    // plus bonus
        frequentRenterPoints ++;
    // bonus for 2 days rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                each.getDaysRented() > 1) frequentRenterPoints ++;
    // output results for each rental
    result += “\t” + each.getMovie().getTitle() + ”\t” + String.valueOf(each.getCharge()) +
“\n”;
    totalAmount += each.getCharge();
    }
    //colontitle
```

```
result += “indebtedness sum: “ + String.valueOf(totalAmount) + “\n”;
result += “you earned “ +  String.valueOf(frequentRenterPoints) + “ loyalty points”;
return result;
}
```

Компіляція та тестування.

Позбавлення від тимчасових змінних має переваги і недоліки. Переваги: 1) наявність тимч. змінних призводить до необхідності передавати багато параметрів; 2) у випадку довгого коду легко забути і сплутати призначення тієї чи іншої змінної. Недоліки: зниження ефективності, оскільки метод доводиться викликати двічі, але з цим можна поборотися іншим способом.

Нарахування бонусів

Наступний крок: аналогічні дії для підрахунку бонусів. Правила тут можуть залежати від типів фільмів, хоча варіантів тут менше, чим при оплаті. Доцільно перекласти відповідальність на клас Rental. Використовуємо рефакторінг «Витяг методу». Знову шукаємо локальні змінні. Це each і frequentRenterPoints. Тут frequentRenterPoints має значення, яке було присвоєно раніше. Але в тілі виділеного методу ця змінна не зчитується, тому передавати її як параметр не потрібно, each можна передати як параметр. Потім переміщення, компіляція і тестування.

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.element();
        String result = “Rental “ + getName() + “\n”;
        while (rentals.hasMoreElements() {
                Rental each = (Rental) rentals.nextElement();
                frequentRenterPoints += each.getFrequentRenterPoints();
// output results for each rental
result += “\t” + each.getMovie().getTitle() + ”\t” + String.valueOf(each.getCharge()) + “\n”;
totalAmount += each.getCharge();
```

```
}
//colontitle
result += “indebtedness sum: “ + String.valueOf(totalAmount) + “\n”;
result += “you earned “ +  String.valueOf(frequentRenterPoints) + “ loyalty points”;
return result;
}
class Rental ….
        int getFrequentRenterPoints() {
                if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                        getDaysRented() > 1)
                        return 2;
                else return 1;
}
```

Вилучення тимчасових змінних

Тимчасові змінні інколи можуть викликати проблеми. Ці змінні використовуються лише у власних методах, що спричиняє їх ускладнення і збільшення. У нашому випадку є дві тимчасові змінні, які використовуються для отримання підсумкових сум по операціям прокату клієнта. Ці підсумкові суми потрібні для обох версій звіту – текстової та HTML. Планується виконати рефакторінг «Заміна тимч. змінної» і замінити totalAmount та frequentRenterPoints викликами методів запитів. Запити доступні будь-якому методу класу, а тому забезпечують більш зрозумілий проект коду без довгих та складних методів.

Почнемо із заміни змінної totalAmount викликом методу getTotalCharge.

```
public String statement() {…
//colontitle
result += “indebtedness sum: “ + String.valueOf(getTotalCharge()) + “\n”;
result += “you earned “ +  String.valueOf(frequentRenterPoints) + “ loyalty points”;
return result;
}
```

```
    private double getTotalCharge() {
        double result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements() )  {
                Rental each = (Rental) rentals.nextElement();
                result += each.getCharge();
        }
    return result;
    }
```

Зауважимо, що присвоєння значення totalAmount здійснювалось в циклі, який необхідно копіювати в метод запиту. Після компіляції та тестування робимо те ж саме для змінної frequentRenterPoints.

```
public String statement() {…
    //colontitle
    result += “indebtedness sum: “ + String.valueOf(getTotalCharge()) + “\n”;
    result += “you earned “ +  String.valueOf(getTotalFrequentRenterPoints()) + “ loyalty points”;
    return result;
    }
    private int getTotalFrequentRenterPoints()  {
        int result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements() )  {
            Rental each = (Rental) rentals.nextElement();
            result += each. getFrequentRenterPoints();
        }
        return result;
    }
```

Movie		Rental		Customer

priceCode: int	1 ←	daysRented: int	* ←	
		getCharge() getFrequentRenterPoints()		statement() getTotalCharge() getTotalFrequentRenterPoints()

Діаграма класів після виділення та переносу коду підрахунку підсумкових сум.

Більшість методів рефакторінгу мають наслідком зменшення об'єму коду, але не в даному випадку. Справа у тому, що Java потребує багато команд для циклу сумування. Навіть простий цикл сумування з одним рядком потребує 6 рядків підтримки. Ще однією проблемою, пов'язаною з такого виду рефакторінгом, є питання продуктивності. Попередній код виконував цикл while один раз, а новий код робить це тричі. Довгий цикл while *може* погано вплинути на продуктивність. У багатьох випадках цього досить, щоб відмовитись від рефакторінгу. Але варто звернути увагу на слово «може». Без профілювання немає можливості упевнено сказати, скільки часу та як часто буде виконуватись цикл і як це вплине на продуктивність системи у підсумку. Не варто турбуватися про це при рефакторінгу. До цього ми ще повернемось при оптимізації, але на той час її виконання буде суттєво спрощено, дякуючи поліпшеному коду.

Створені методи доступні будь-якому методу класу Customer. Якщо інформація, яку вони повертають буде потрібна в інших місцях програми, то ці методи легко додати в інтерфейс класу. У випадку відсутності цих методів запитів іншим методам доведеться мати справу з внутрішньою будовою класу Rental і створювати цикли. В складній програмі це буде потребувати написання великого за об'ємом коду.

Тепер напишемо метод htmlStatement.

```
public String htmlStatement() {
        Enumeration rentals = _rentals.elements();
        String result = "<H1>Rent <EM>" + getName() +  "</EM></H1><P>\n";
        while (rentals.hasMoreElements() )  {
           Rental each = (Rental) rentals.nextElement();
    // output results for each rental
```

```
            result += each.getMovie().getTitle() + ": " + String.valueOf(each.getCharge()) +
"<BR>\n";
        }
    //colontitle
        result += "<P> Indebtedness sum <EM>" + String.valueOf(getTotalCharge() ) +
"</EM> <P>\n";
        result += "You earned  <EM>" + String.valueOf(getTotalFrequentRenterPoints()) +
            "</EM>  loyalty points <P>";
        return result;
}
```

Метод htmlStatement був написаний з повторним використанням всіх обчислень, які раніше були присутні в методі statement. Тепер у випадку зміни правил розрахунків, код доведеться переписувати тільки в одному місці. Можна так само легко додати будь-який інший формат звіту.

Нове завдання. Користувачі хочуть змінити класифікацію фільмів. Поки що, незрозуміло, як саме, щось буде змінено, щось додано. Для нових категорій буде встановлений свій порядок оплати та нарахування бонусів. При нинішньому стані справ внесення таких змін є досить складною задачею. Внесення змін в класифікацію фільмів потребує зміни в коді з умовними виразами, в методах обчислення оплати і бонусних балів. Звернемося до рефакторінгу.

Заміна логіки умов поліморфізмом.

Перша складність цієї задачі полягає у операторі switch. Викликає питання сама ідея організації вибору в залежності від значень атрибуту деякого іншого об'єкта. Така думка приводить до висновку , що метод getCharge потрібно перенести в клас Movie.

Для дієздатності цього коду, потрібно передавати йому тривалість прокату, яка є інформацією з класу Rental. Цей метод використовує два елемента даних – тривалість прокату і тип фільму. Чому краще передавати тривалість прокату класу Movie, а не тип фільму класу Rental? Це пов'язано з тим, що очікувані зміни мають справу тільки з введенням нових типів фільмів. Інформація про типи фільмів у загальному випадку більш

динамічна, тому бажано, щоб вплив від цього поширювався якомога менше, тому обчислення оплати краще виконувати в класі Movie.

Внесемо метод в Movie та видозмінено getCharge з класу Rental так, щоб він використовував новий метод.

```
class Rental . . .
   double getCharge() {
     return _movie.getCharge(_daysRented);
}
```

Далі аналогічні дії з методом нарахування бонусів. Таким чином обидві компоненти, які залежать від типу фільму, попадають в один клас, де зберігається інформація про цей тип.

```
class Rental . . .
        int getFrequentRenterPoints() {
           return _movie.getFrequentRenterPoints(_daysRented);
        }
class Movie . . .
  int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1) return 2;
        else return 1;
  }
```

Movie		Rental		Customer
priceCode: int	1 ←	daysRented: int	* ←	
getCharge(days: int) getFrequentRenterPoints (days: int)		getCharge() getFrequentRenterPoints()		statement() htmlStatement() getTotalCharge() getTotalFrequentRenterPoints()

Діаграма класів після виділення та переносу методів в клас Movie.

Успадкування.

У нас є кілька типів фільмів з різними обчисленнями. Є підстава подумати про підкласи. Ми можемо використати три підкласи Movie, кожен з яких буде мати власну версію нарахування оплати. Це дозволить замінити оператор switch поліморфізмом. Але є невеличкий недолік: воно не спрацює - за час існування фільм може змінити тип, але об'єкт не може змінити клас. Для подолання цієї проблеми використаємо проектний шаблон «Стан» (State pattern). З'являється клас Price та його підкласи RegularPrice, ChildrensPrice і NewReleasePrice. Додавши додатковий рівень опосередкованості, можна створювати підкласи для класу Price та при необхідності змінювати ціну.

Щоб використати проектний шаблон, задіємо три рефакторінга. Спочатку переносимо код, який залежить від типу, в шаблон стану за допомогою рефакторінга «Заміна коду типа станом/стратегією», потім рефакторінг «Перенос методу» (оператор switch переносимо в клас Price. І нарешті, за допомогою рефакторінга «Заміна умовного оператора поліморфізмом» switch вилучається взагалі.

Почнемо з рефакторінга «Заміна коду типу станом/стратегією». Цей 1-ий етап полягає у застосуванні рефакторінгу «Самоінкапсуляція поля», щоб гарантувати виконання будь-яких дій через методи отримання та встановлення значень. Оскільки код, в основному, був взятий з інших класів, то в більшості методів вже використовуються методи отримання значень, але конструктори встановлюють ціну безпосередньо.

```
class Movie . . .
    public Movie (String name, int priceCode) {
        _name = name;
       setPriceCode(priceCode);
    }
```

Компіляція і тестування. Потім додаються нові класи.

```
abstract class Price {
    abstract int getPriceCode();
}
class ChildrensPrice extends Price {
    int getPriceCode() {
        return Movie.CHILDRENS;
    }
}
class NewReleasePrice extends Price {
    int getPriceCode() {
        return Movie.NEW_RELEASE;
    }
}
class RegularPrice extends Price {
    int getPriceCode() {
        return Movie.REGULAR;
    }
}
```

Далі потрібно змінити методи доступу.

```
private int _priceCode;
public int getPriceCode() {
    return _priceCode;
}
public int setPriceCode(int arg)  {
    _priceCode = arg;
}
```

Це означає заміну коду поля ціни полем ціни та зміну методів доступу.

class Movie . . .

```
        public int getPriceCode() {
                return _priceCode;
        }
        public void setPriceCode(int arg)  {
                switch (arg)  {
                   case REGULAR:
                                _price = new RegularPrice();
                                break;
                   case CHILDRENS:
                                _price = new ChildrensPrice();
                                break;
                  case NEW_RELEASE:
                                _price = new NewReleasePrice();
                                break;
                   default:
                                throw new IllegalArgumentException (“Incorrect Price Code”);
                }
        }
private Price _price;
```

Застосуємо до getCharge рефакторінг «Перенос методу».

```
class Movie . . .
   double getCharge (int daysRented) {
      return _price. getCharge(daysRented);
   }
class Price . . .
   double getCharge (int daysRented) {
      double result = 0;
      switch (getPriceCode() )  {
```

```
        case Movie.REGULAR:
                result += 2;
                if (daysRented > 2) result += (daysRented - 2) * 1.5;
                break;
        case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
        case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3) result += (daysRented - 3) * 1.5;
                break;
    }
  return result;
}
```

Тепер можемо перейти до рефакторінгу «Заміна умовного оператора поліморфізмом». Будемо діяти по кожній гілці оператора switch по черзі.

```
class RegularPrice . . .
     double getCharge (int daysRented) {
        double result = 2;
        if (daysRented > 2) result += (daysRented - 2) * 1.5;
        return result;
   }
```

Так перекривається відповідна гілка оператора switch.

```
class ChildrensPrice . . .
   double getCharge (int daysRented) {
      double result = 1.5;
        if (daysRented > 3) result += (daysRented - 3) * 1.5;
        return result;
```

```
    }
class NewReleasePrice . . .
   double getCharge (int daysRented) {
       return daysRented * 3;
}
```

Далі метод Price.getCharge оголошується абстрактним.

```
class Price . . .
   abstract double getCharge (int daysRented);
```

Подібна процедура проводиться з getFrequentRenterPoints.

```
class Movie . . .
   int getFrequentRenterPoints (int daysRented) {
      return _price. getFrequentRenterPoints ( daysRented);
   }
class Price . . .
   int getFrequentRenterPoints (int daysRented) {
      if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1) return 2;
      else return 1;
   }
```

Але в цьому випадку метод суперкласу не робиться абстрактним; замість цього створюється перекриваючий метод у нових версіях, а в суперкласі за замочуванням використовується використовуваний метод.

```
class NewReleasePrice
   int getFrequentRenterPoints (int daysRented) {
      return (daysRented > 1) ? 2 : 1;
   }
class Price . . .
   int getFrequentRenterPoints (int daysRented) {
      return 1;
```

}

Впровадження проектного шаблону «Стан» змусило суттєво ускладнити зусилля. Чи варто було це робити? Користь в тому, що при необхідності зміни поведінки Price (додавати нові ціни, чи додаткову поведінку, яка залежить від ціни) реалізація таких модифікацій буде значно легшою. Інші частини застосування нічого не знають про використання проектного шаблону. Для заданого набору функцій такий виграш несуттєвий, але при збільшенні їх кількості користь стане суттєво помітною.

Movie		Rental		Customer
title: String	1 ←	daysRented: int	* ←	name: String
getCharge(days: int) getFrequentRenterPoints (days: int)		getCharge() getFrequentRenterPoints()		statement() htmlStatement() getTotalCharge() getTotalFrequentRenterPoints()

|

| 1

V

Price	ChildrensPrice
	getCharge(days: int)
getCharge(days: int) getFrequentRenterPoints(days: int)	RegularPrice
	getCharge(days: int)
	NewReleasePrice
	getCharge(days: int) getFrequentRenterPoints(days: int)

Діаграма класів після додавання проектного шаблону.

І на завершення, найбільш важливо – це ритм рефакторінга (тестування, невеликі зміни) і так кілька разів.

Рефакторінг баз даних.

Всі сучасні процеси розробки програмного забезпечення, включаючи уніфікований процес компанії Rational (Rational Unified Process – RUP), екстремальне програмування (Extreme Programming - XP), адаптивний уніфікований процес (Agile Unified Process - AUP), метод Scrum і адаптивний метод розробки систем (Dynamic System Development Method – DSDM), за своїм характером є еволюційними. Взагалі, еволюційні підходи до програмування наразі є домінуючими. Але методи створення застосунків, призначених для обробки даних, в основному за своїм характером є послідовними і розраховані на залучення фахівців, що виконують відносно вузькі завдання, такі як логічне або фізичне моделювання даних. Фахівці в області програмування і фахівці в області обробки даних повинні співпрацювати, але і ті й інші прагнуть організувати свою роботу по-різному.

Операції рефакторінгу бази даних відносяться до числа найбільш важливих навичок, якими повинні володіти фахівці в області обробки даних. Рефакторінг баз даних – це метод реалізації баз даних, так само, як рефакторінг коду є методом реалізації програм. Як правило, схема бази даних піддається рефакторінгу з метою спрощення подальших доповнень до цієї схеми. Крім того, на практиці часто виявляється, що в базу даних потрібно внести новий додаток, стовпець, або збережену процедуру, але існуючий проект не можна назвати найбільш підходящим для забезпечення підтримки цього нового додатку. В такому випадку перш за все здійснюється рефакторінг схеми бази даних, щоб спростити введення нового засобу, а після успішного проведення операції рефакторінгу додається потрібний додаток. Перевагою такого підходу є те, що він дозволяє поступово, але незмінно підвищувати якість проекту бази даних. В результаті здійснення

такого процесу не тільки досягається спрощення розуміння і використання бази даних, а й спрощується її подальший розвиток, підвищується загальна продуктивність розробки.

Ми знаєте, що ми функціонально повинні поліпшити, але ми повинні бути обережними щодо точного характеру проблеми.

Як би ми не дивились на це, але будь-яке комп'ютерне застосування, зрештою, зводиться до використання процесора, пам'яті та операцій вводу/виводу (I/O) з диску, мережі чи іншого пристрою I/O. Коли виникають проблеми з продуктивністю, першим пунктом для діагностики є те, чи не досяг один із цих ресурсів проблемних рівнів, тому що це допоможе у пошуку того, що потрібно поліпшити, і як це зробити.

Застосування баз даних - це те, що ми можемо спробувати покращити використання ресурсів на різних рівнях. Якщо ми дійсно хочемо підвищити продуктивність SQL застосування, ми можете зупинитися на тому, що виглядає як очевидне вузьке місце і спробуємо полегшити ситуацію на цьому етапі (наприклад, "давайте надамо більше пам'яті для СУБД" або "будемо використовувати більш швидкі диски ").

Такий підхід був обгрунтованим протягом більшої частини 80-х років, коли SQL став прийнятим для доступу до корпоративних даних. Все ще багато хто думає, що кращим способом покращення продуктивності бази даних, є або налаштування ряду переважно незрозумілих параметрів бази даних, або покращення обладнання. На більш просунутому рівні можна відстежувати повні сканування великих таблиць і додавати індекси, щоб їх усунути. На ще більш просунутому рівні ви можете спробувати налаштувати

оператори SQL і переписати їх, щоб оптимізувати їх план виконання. Або ви можете переглянути весь процес.

Далі ми зосередимось на останніх трьох варіантах та дослідимо різні способи досягнення змін, які іноді вражають, незалежно від налаштування параметрів бази даних або оновлення обладнання.

Перш ніж намагатися визначити, як можна оцінювати, чи може кожен фрагмент коду отримати вигоду від рефакторінгу, розглянемо простий, але не надто тривіальний приклад, щоб проілюструвати різницю між рефакторінгом та налаштуванням. Наступний приклад штучний, але має в основі деякі реальні аспекти.

Простий приклад

Припустімо, у нас є кілька "областей", до яких приєднані "рахунки", і припустімо, що суми в різних валютах зв'язані з цими рахунками. Кожна сума відповідає транзакції. Ви хочете перевірити для деякої області, чи є суми, які перевищують певний поріг для транзакцій, які відбулися за 30 днів від заданої дати. Цей поріг залежить від валюти, і це не визначено для всіх валют. Якщо поріг визначено, і якщо сума перевищує порогові значення для даної валюти, ми повинні зареєструвати ідентифікатор транзакції, а також суму, конвертовану в місцеву валюту на певну дату оцінки.

Для даного прикладу створено таблицю транзакцій із двома мільйонами рядків, та використано деякий код Java ™ / JDBC, щоб показати, як різні способи кодування можуть вплинути на виконання. Код Java спрощений, щоб кожен, хто знає програмування або мову сценаріїв, може зрозуміти його основну ідею.

Основа програми така (арифметика дат в наступному коді використовує синтаксис MySQL) - програма називається FirstExample.java:

```
  try {
2     long  txid;
            accoun
3     long  tid;
            amoun
4     float  t;
      String  curr;
            conv_am
6     float  ount;
7

      PreparedStatement st1 = con.prepareStatement("select accountid"
                        + " from
9                       area_accounts"
1                       + " where
0                       areaid = ?");
1
1     ResultSet     rs1;
      PreparedStatement st2 = con.prepareStatement("select txid,amount,curr"
1
3                       +" from transactions"
1
4                       +" where accountid=?"
1                           and txdate >= date_sub(?,
5                       +" interval 30 day)"
                        +" order by txdate");
1
```

```
6

1
7   ResultSet    rs2 = null;

    PreparedStatement st3 = con.prepareStatement("insert into check_log(txid,"

1                       conv_am
9                   + " ount)"

2                       values(?,
0                   + " ?)");

2
1


    st1.setInt(1, areaid);
    rs1 = st1.executeQuery( );
    while (rs1.next( )) {
       accountid = rs1.getLong(1);
       st2.setLong(1, accountid);
       st2.setDate(2, somedate);
       rs2 = st2.executeQuery( );
       while (rs2.next( )) {
          txid = rs2.getLong(1);
          amount = rs2.getFloat(2);
          curr = rs2.getString(3);
          if (AboveThreshold(amount, curr)) {
             // Convert
             conv_amount = Convert(amount, curr, valuationdate);
             st3.setLong(1, txid);
             st3.setFloat(2, conv_amount);
             dummy = st3.executeUpdate( );
```

```
            }
          }
        }
        rs1.close( );
        st1.close( );
        if (rs2 != null) {
          rs2.close( );
        }
        st2.close( );
        st3.close( );
    } catch(SQLException ex){
            System.err.println("==> SQLException: ");
            while (ex != null) {
                                          +
55              System.out.println("    ex.getMessage
22              Message:              "( ));
55              System.out.println("      ex.getSQLSta
33              SQLState:             "+te ( ));
55              System.out.println("Er   ex.getErrorCo
44              rorCode: "            +de ( ));
                ex =
55              ex.getNextException
55              ( );
55              System.out.println(""
66              );
            }
    }
```

Цей фрагмент коду нагадує багато фрагментів коду, що працюють у реальних застосуваннях. Кілька слів пояснення:

У нас є три SQL-оператори (рядки 8, 12 і 18), які є prepared операторами. Prepared оператори – це правильний спосіб кодування з JDBC, коли ми неодноразово виконуємо ідентичні оператори, за винятком кількох значень, які змінюються при кожному виклику. Ці значення представлені знаками питання, які діють як маркери місця, і ми пов'язуємо фактичне значення для кожного маркера з такими викликами, як setInt () у рядку 22 або setLong () та setDate () у рядках 26 та 27.

У рядку 22 встановлюється значення (areaid), яке визначається у тій частині програми, яка тут не показана.

Коли фактичні значення прив'язані до маркерів місця, можна викликати executeQuery () як у рядку 23, якщо оператором SQL є select або executeUpdate (), як у рядку 38, якщо оператор є чимсь іншим. Для select операторів отримується результуюча множина, по якій можна циклічно рухатись, щоб отримати всі значення у свою чергу, як це можна бачити на рядках 30, 31 та 32, наприклад.

Дві утиліти: AboveThreshold () у рядку 33 перевіряє, чи сума перевищує порогові значення для певної валюти, та Convert () у рядку 35 переводить суму, яка перевищує порогову величину, в базову валюту для звіту. Ось код для цих двох функцій:

```
private static boolean AboveThreshold(float amount,String iso)
    throws Exception {

  PreparedStatement thresholdstmt = con.prepareStatement("select
  threshold"

                                  " from thresholds"

                                  " where iso=?");

  ResultSet   rs;
```

```
   Boolean   returnval = false;

   thresholdstmt.setString(1, iso);

   rs = thresholdstmt.executeQuery( );

   if (rs.next( )) {

      if (amount >= rs.getFloat(1)){

         returnval = true;

      } else {

         returnval = false;

      }

   } else { // not found - assume no problem
     returnval = false;

   }

   if (rs != null) {

      rs.close( );

   }

   thresholdstmt.close( );

   return returnval;

}

private static float Convert(float amount,  String iso, Date valuationdate)
            throws Exception {
```

```
                PreparedStatement conversionstmt = con.prepareStatement(
                                        "select ? * rate"
                                        " from currency_rates"
                                        " where iso = ?"
                                        " and rate_date = ?");
ResultSet     rs;
float           val = (float)0.0;
conversionstmt.setFloat(1, amount);
conversionstmt.setString(2, iso);
conversionstmt.setDate(3, valuationdate);
rs = conversionstmt.executeQuery( );
if (rs.next( )) {
   val = rs.getFloat(1);
}
if (rs != null) {
   rs.close( );
}
conversionstmt.close( );
return val;
}
```

У всіх таблицях визначені первинні ключі. При застосуванні цієї програми до кортежів даних, перевіривши близько семи з двох мільйонів рядків і, в кінцевому підсумку, вибравши дуже мало рядків, програма зайняла близько 11 хвилин на MySQL * на тестовій машині.

Після незначної модифікації SQL-коду, щоб враховувати різні способи, якими різні діалекти мови виражають місяць, попередній до дати, була застосована одна і та ж програма до одного і того ж об'єму даних в SQL Server і Oracle.

Програмі потрібно було близько п'яти з половиною хвилин на SQL Server і трохи менше трьох хвилин на Oracle. Для порівняння, у Таблиці 1-1 наведено кількість часу, необхідного для запуску програми для кожної СУБД; як ви можете бачити, у всіх трьох випадках це зайняло багато часу. Що ми можемо зробити, перш ніж поспішати купувати нове більш швидке апаратне забезпечення?

T A B L E 1 - 1 . Baseline for SimpleExample.java

DBMS	**Baseline result**
MySQL (5.1)	11 minutes
Oracle (11)	3 minutes
SQL Server (2005)	5.5 minutes

Налаштування SQL, традиційний шлях

Звичайний підхід на цьому етапі полягає в тому, щоб передати програму спеціалісту по внутрішньому налаштуванню (зазвичай адміністратору БД [DBA]). DBA MySQL ймовірно знову запустить програму в тестовому

середовищі після підтвердження того, що тестова база даних була запущена з наступними двома опціями:

```
--log-slow-queries

--log-queries-not-using-indexes
```

Отриманий logfile показує багато повторних викликів, що тривають від трьох до чотирьох секунд кожен, від головного винуватця, який є наступним запитом:

```
select txid,amount,curr

from transactions

where accountid=?

 and txdate >= date_sub(?, interval 30 day)

order by txdate
```

Перевірка бази даних information_schema (або використання такого інструмента, як phpMyAdmin) швидко покаже, що у таблиці транзакцій є єдиний індекс - індекс первинного ключа txid, який в цьому випадку не використовується, бо у відповідному стовпчику немає умов. В результаті сервер БД не може нічого зробити, окрім сканування великої таблиці від початку до кінця, і це робиться в циклі. Рішення є очевидним: створити додатковий індекс для звіту та запустити процес знову. Результат? Зараз він виконується за трохи менше аніж чотири хвилини, тобто покращення в 3 рази. Знову ж таки, легкий маневр DBA врятував справу.

Для нашого MySQL DBA це, ймовірно, буде кінець історії. Проте його колегам з Oracle та SQL Server не так легко. Не менш мудрий, ніж MySQL DBA, Oracle DBA активував магічну зброю налаштування Oracle, відомий серед ініційованих як *event 10046 level 8* (або використовується для цього ж самого " advisor ") і отримав trace file, який чітко показує де витрачався час. У

такому trace file ви можете визначити, скільки разів виконувалися оператори, час використання ними CPU, весь використаний час та інша ключова інформація, така як, кількість логічних зчитувань (які відображаються як query and current в trace file), тобто кількість блоків даних, які були доступні для обробки запиту, що пояснює принаймні частку різниці між часом CPU та загальним минулим часом:

```
********************************************************************************

SQL ID : 1nup7kcbvt072

select txid,amount,curr

from

 transactions where accountid=:1 and txdate >= to_date(:2, 'DD-MON-YYYY') - 30
  order by txdate
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	252	0.00	0.01	0	0	0	0
Fetch	11903	32.21	32.16	0	2163420	0	117676
total	12156	32.22	32.18	0	2163420	0	117676

Misses in library cache during parse: 1

Misses in library cache during execute: 1

Optimizer mode: ALL_ROWS

Parsing user id: 88

Rows Row Source Operation

------- ---

495 SORT ORDER BY (cr=8585 [...] card=466)

TABLE ACCESS FULL TRANSACTIONS]
495 (cr=8585 [... card=466)

Elapsetimes include waiting on following devents:

Event	waited on		Times Waited	Max. Wait	Total Waited
SQL*Net message	to client		11903	0.00	0.02
SQL*Net message	from client		11903	0.00	2.30

****

****

SQL ID : gx2cn564cdsds

select threshold

from thresholds
where iso=:1

call	count	cpu	elapsed	disk	query	current	rows
Parse	11767 4	2.68	2.63	0	0	0	0
Execut	11767	5.13	5.10	0	0	0	0

e 4

Fetch	117674	4.00	3.87	0	232504	0	114830
total	353022	11.82	11.61	0	232504	0	114830

Misses in library cache during parse: 1

Misses in library cache during execute: 1

Optimizer mode: ALL_ROWS

Parsing user id: 88

Rows	Row Source Operation
1	TABLE ACCESS BY INDEX ROWID THRESHOLDS (cr=2 [...] card=1)
1	INDEX UNIQUE SCAN SYS_C009785 (cr=1 [...] card=1)(object id 71355)

Elapsetimes include waiting on following devents:

Event waited on		Times Waited	Max. Wait	Total Waited
SQL*Net message	to client	117675	0.00	0.30
SQL*Net message	from client	117675	0.14	25.04

****
****

Дослідження TABLE ACCESS FULL TRANSACTION у плані виконання самого повільного запиту (особливо, якщо він виконується 252 рази) викликає ту ж саму реакцію з адміністратором Oracle, як і з адміністратором MySQL. На Oracle такий же індекс на accountid підвищив продуктивність в 1,2 разу, при цьому час виконання становив близько двох хвилин та 20 секунд. DBA SQL Server не є більш щасливим. Після використання SQL Profiler або запуску:

```
select a.*

from (select execution_count, total_elapsed_time, total_logical_reads,

        substring(st.text, (qs.statement_start_offset/2) + 1, ((case
                statement_end_offset

                  when -1 then datalength(st.text)

                  else qs.statement_end_offset

                end

                  qs.statement_start_offset)/2) + 1) as statement_text from
    sys.dm_exec_query_stats as qs

        cross apply sys.dm_exec_sql_text(qs.sql_handle) as st) a

where a. statement_text not like '%select a.*%'

order by a.creation_time
```

результати в:

execution_count	total_elapsed_time	total_logical_reads	statement_text
228	98590420	3062040	select txid,amount, ...
212270	22156494	849080	select threshold from ...
1	2135214	13430	...

...

DBA SQL Server, помітивши, що найдорожчий запит на сьогоднішній день select on transactions, приходить до такого ж висновку, що і інші DBA: transactions table не підтримує індекс. На жаль, дії по виправленню знову призводять до розчарувань. Створення індексу на accountid підвищує ефективність за дуже скромним співвідношенням 1: 3, трохи більше чотирьох хвилин, що насправді недостатньо. Table 1-2 показує для СУБД покращення швидкості при новому індексі.

T A B L E 1 - 2 . Speed improvement factor after adding an index on transactions

DBMS	**Speed improvement**
MySQL	x3.1
Oracle	x1.2
SQL Server	x1.3

Налаштування за допомогою індексації дуже популярне у розробників, оскільки не потрібно змінювати код; вона однаковою мірою популярна серед DBA, які часто не бачать коду і знають, що належна індексація набагато більшою мірою може принести помітні результати, ніж налаштування незрозумілих параметрів.

Упорядкування коду

Передусім модифікуємо код *FirstExample.java,* щоб створити *SecondExample.java*. Виконано два покращення початкового коду. Яка мета головного запиту:

```
select txid,amount,curr
from transactions
where accountid=?
  and txdate >= date_sub(?, interval 30 day)
order by txdate
```

Ми просто вибираємо дані з однієї таблиці для іншої таблиці. Якщо ми хочемо відсортований результат, ми додамо підфразу order by до запиту, який вибирає дані з таблиці, коли ми представляємо його кінцевому користувачеві. На даний момент, order by, є просто безглуздим; це дуже поширена помилка.

Друге поліпшення пов'язано з повторними вставками даних з помірною швидкістю (в кінці отримується кілька сотень рядків у logging table). За замовчуванням JDBC-з'єднання знаходиться в режимі autocommit. У цьому випадку це означає, що кожному insert неявно слідує оператор commit, і кожна зміна буде синхронно заноситись на диск. Занесення даних у зовнішню пам'ять гарантує, що ця зміна не буде втрачена, навіть якщо система аварійно зупиниться у наступну мілісекунду; без commit зміна відбувається в оперативній пам'яті і може бути втрачена. Чи дійсно потрібна впевненість, що кожен вставлений рядок надійно записується на диск, перш ніж вставляти наступний рядок? Якщо буде збій системи, можна просто знову запустити цей процес, особливо якщо це швидко зробити; це - не часто буде траплятися. Тому було вставлено один оператор на початку, щоб вимкнути поведінку за замовчуванням, а інший - в кінці, щоб явно вносити зміни при закінченні:

```
 Turn autocommit off
con.setAutoCommit(false);
```

and:

```
con.commit( );
```

Ці дві дуже маленькі зміни призводять до дуже невеликого поліпшення: їх сукупний ефект робить версію MySQL на 10% швидше. Але ми не отримуємо практично ніяких вимірюваних коефіцієнтів з Oracle і SQL Server (див. Таблицю 1-3).

T A B L E 1 - 3 . Speed improvement factor after index, code cleanup, and no auto-commit

DBMS	**Speed improvement**
MySQL	x3.2
Oracle	x1.2
SQL Server	x1.3

Налаштування SQL

Коли один індекс не може досягти потрібного результату, то інколи інший індекс може забезпечити кращу ефективність. Чому створюємо індекс на accountid окремо? В основному, індекс - сортований список (сортований у дереві) ключових значень, пов'язаних з фізичними адресами рядків, які відповідають цим ключовим значенням. Якщо ми виконуємо пошук за значеннями двох стовпців та індексуємо лише один з них, нам доведеться витягувати всі рядки, які відповідають ключу, який ми шукаємо, а також відкидаємо підмножину рядків, яка не відповідає іншому стовпцю. Якщо ми індексуємо обидва стовпці, то ми йдемо прямо за тим, що ми дійсно хочемо.

Ми можемо створити індекс на (accountid, txdate), оскільки дата транзакції - це ще один критерій у запиті. Створюючи складний індекс на обох стовпцях, ми гарантуємо, що движок SQL може виконувати ефективний обмежений пошук (відомий як *range scan*) в індексі. По даним тесту, якщо одностовпчиковий індекс покращив продуктивність MySQL в 3 рази, то швидкість збільшується більш ніж в 3,4 рази за допомогою індексу з двома стовпчиками, тож для виконання програми потрібно близько трьох з половиною хвилин. Погана новина полягає в тому, що на Oracle і SQL Server,

навіть за допомогою двох стовпчикових індексів не досягається поліпшення відносно попереднього випадку індексу з одним стовпцем (див. Table 1-4).

T A B L E 1 - 4 . Speed improvement factor after index change

DBMS	**Speed**
MySQL	x3.4
Oracle	x1.2
SQL Server	x1.3

До цього моменту використовувалось те, що називається "традиційним підходом": налаштування, комбінація деяких мінімальних покращень до SQL-операторів, використання таких речей як управління транзакціями та стратегією індексування. Тепер будемо більш радикальними і послідовно розглянемо дві різні точки зору. Давайте спочатку розглянемо, як організована програма.

Рефакторінг, перша точка зору

Як і в багатьох реальних процесах вражаючою особливістю цього прикладу є гніздування циклів. Глибоко всередині циклів ми знаходимо виклик функції-утиліти AboveThreshold (), яка викликається для кожного повернутого рядка. Раніше вже згадувалось, що transactions table містить два мільйони рядків, і що близько однієї сьомої частини рядків відносяться до "області" під розлядом. Тому ми викликаємо функцію AboveThreshold () багато, багато разів. Кожного разу, коли функцію викликають багато разів, будь-який дуже маленький виграш має значний ефект.

Гарним способом економії часу є зменшення кількості звернень до бази даних. Хоча багато розробників вважають, що база даних є негайно доступним ресурсом, запит до бази даних не є вільним. Насправді, запит до бази даних є дорогою операцією. Ви повинні спілкуватися з сервером, що тягне за собою деяку латентність мережі, особливо якщо ваша програма не працює на сервері. Крім того, те, що ви надсилаєте на сервер, не є безпосередньо виконуваним машинним кодом, а оператором SQL. Сервер повинен проаналізувати його і перекласти на фактичний машинний код. Можливо, він вже виконав аналогічний оператор, і в такому випадку обчислення "сигнатури" оператора може бути достатнім, щоб дозволити серверу повторно використовувати кеш-оператор.

Або це може бути перший раз, коли ми зіткнемося з цим оператором, і серверу, можливо, доведеться визначити правильний план виконання та запускати рекурсивні запити до словника даних. Або цей оператор міг бути виконаним, але з тих пір, можливо, він вийшов з кеш-пам'яті, щоб звільнити місце для іншого оператора, і в цьому випадку це буде так, ніби ми зустрічаємо його вперше. Тоді команда SQL повинна бути виконана і через мережу поверне дані, які можуть зберігатися в кеш-пам'яті сервера баз даних або завантажуватися з диска. Іншими словами, виклик бази даних перетворюється на послідовність операцій, які не обов'язково дуже довгі, але мають на увазі споживання ресурсів - мережі, оперативної пам'яті, CPU, and I/O та операцій. Паралельність між сесіями може додати очікування для нерозшарених ресурсів, до яких одночасно потрібен доступ.

Повернемося до функції AboveThreshold (). У цій функції ми перевіряємо пороги, пов'язані з валютами. Існує особливість з валютами; хоча у світі існує близько 170 валют, навіть велика фінансова установа буде мати справу лише з кількома валютами - місцевою валютою, валютами основних торговельних партнерів країни та кількома неминучими основними валютами, які важливі в світовій

торгівлі: Долар США, євро, ймовірно, японська єна та британський фунт, серед інших.

Наприклад:

Currency Code	Currency Name	Percentage
EUR	Euro	41.3
USD	US Dollar	24.3
JPY	Japanese Yen	13.6
GBP	British Pound	11.1
CHF	Swiss Franc	2.6
HKD	Hong Kong Dollar	2.1
SEK	Swedish Krona	1.1
AUD	Australian Dollar	0.7
SGD	Singapore Dollar	0.5

Як результат, ми не тільки викликаємо AboveThreshold () сотні тисяч разів, але також функція повторює виклик тих самих рядків з порогової таблиці. Ви можете подумати, що через те, що ці кілька рядків, ймовірно, будуть зберігатися в кеш-пам'яті сервера баз даних, це не матиме значення. Але це має значення, і далі ми побачимо повну міру шкоди, спричиненої марнотратними викликами, шляхом переписання функції більш ефективним способом.

Назвемо нову версію програми *ThirdExample.java*, і використаємо певні колекції Java або HashMaps для зберігання даних; ці колекції зберігають пари ключі/ значення шляхом хешуванням ключа, щоб отримати індекс масиву, який показує, куди пара спрямовується. Можна було б використати масиви з іншою мовою. Але ідея полягає в тому, щоб уникнути запитів до бази даних, використовуючи простір пам'яті процесу як кеш-пам'ять. Коли деякі дані запитуються вперше, то вони отримуються з бази даних і заносяться в колекції в

оперативній пам'яті, перш ніж повертати значення тому, хто його викликав.

Наступного разу, коли будуть потрібні ті самі дані, ми знайдемо їх в нашій маленькій локальній кеші-пам'яті і отримаємо майже відразу. Дві обставини дозволяють нам кешувати дані:

- Ми не перебуваємо в контексті реального часу, і знаємо, що неодноразовий запит порогу, пов'язаного з даною валютою видасть кожен раз однакове значення: між викликами не буде змін.
- Ми працюємо з невеликою кількістю даних. В кеші не буде гігабайт даних. Вимоги до пам'яті є важливим моментом, який слід враховувати, коли існує або може бути велика кількість одночасних сеансів.

Тому переписані дві функції (найбільш критичним є AboveThreshold (), але застосування такої ж логіки для Convert () також може бути корисним):

```
   Use hashmaps for thresholds and exchange rates
private static HashMap thresholds = new HashMap(
); private static HashMap rates = new HashMap( );

private static Datepreviousdate = 0;

...


private static boolean AboveThreshold(float amount, String iso)
  throws Exception {

  float threshold;


  if (!thresholds.containsKey(iso)){


     PreparedStatement thresholdstmt = con.prepareStatement("select threshold"

                                          + " from thresholds"
```

```
                                                    + " where iso=?");
        ResultSet       rs;

        thresholdstmt.setString(1, iso);
        rs = thresholdstmt.executeQuery( );
        if (rs.next( )) {
            threshold = rs.getFloat(1);
            rs.close( );
        } else {
            threshold = (float)-1;
        }
        thresholds.put(iso, new Float(threshold));
        thresholdstmt.close( );
    } else {
        threshold = ((Float)thresholds.get(iso)).floatValue( );
    }
    if (threshold == -1){
        return false;
    } else {
        return(amount >= threshold);
    }
}
```

```
        private static float Convert(float amount, String iso, Date valuationdate)
            throws Exception {

          float rate;

if ((valuationdate != previousdate)

   || (!rates.containsKey(iso))){

  PreparedStatement conversionstmt = con.prepareStatement("select rate"

                                                + " from currency_rates"

                                                + " where iso = ?"

                                                + " and rate_date = ?");

   ResultSet       rs;

   conversionstmt.setString(1, iso);

   conversionstmt.setDate(2, valuationdate);

   rs = conversionstmt.executeQuery( );

   if (rs.next( )) {

      rate = rs.getFloat(1);

      previousdate = valuationdate;

      rs.close( );

   } else {   // not found - There should be an issue!

      rate = (float)1.0;

   }

   rates.put(iso, rate);

   conversionstmt.close( );
```

```
    } else {
        rate = ((Float)rates.get(iso)).floatValue( );
    }
    return(rate * amount);
  }
```

При такому переписуванні плюс складний індекс з двох колонок (accountid, txdate) час виконання різко падає: 30 секунд з MySQL, 10 секунд з Oracle і трохи менше 9 секунд з SQL Server, покращення за відповідними факторами 24, 16 та 38 порівняно з початковою ситуацією (див. Table 1-5).

T A B L E 1 - 5 . Speed improvement factor with a two-column index and function rewriting

DBMS	**Speed improvement**
MySQL	x24
Oracle	x16
SQL Server	x38

Ще одне можливе поліпшення - це на MySQL log (як і Oracle trace and the sys.dm_exec_query_stats dynamic SQL Server table), тобто головний запит:

```
select txid,amount,curr
from transactions
where accountid=?
  and txdate >= [date expression]
```

виконується кілька сотень разів. Зайве говорити, що це набагато менш болісно, коли таблиця правильно проіндексована. Але значення, яке надається для accountid, є не що інше, як результат іншого запиту. Немає необхідності запитувати сервер, отримати значення accountid, подати його в основний запит і, нарешті, виконати основний запит. Ми можемо мати єдиний запит, з підзапитом "piping in" значення accountid:

```
select txid,amount,curr
from transactions
where accountid in (select accountid
                    from area_accounts
                    where areaid = ?)
 and txdate >= date_sub(?, interval 30 day)
```

Це єдине удосконалення, яке зроблено для створення *FourthExample.java*. Отримано досить невтішний результат в Oracle (оскільки він трохи більш ефективний, ніж ThirdExample.java), але зараз програма працює на SQL Server за 7,5 секунд, а на MySQL - за 20,5 секунд, відповідно 44 і 34 разів швидше, ніж початкова версія (див. Table 1-6). Тим не менш, є щось нове та цікаве в *FourthExample.java*: у всіх продуктах швидкість залишається приблизно такою ж незалежно від того, чи існує (чи ні) індекс у стовпці accountid у transactions, а також індекс на accountid окремо або на підставі accountid і txdate.

T A B L E 1 - 6 . Speed improvement factor with SQL rewriting and function rewriting

DBMS	**Speed improvement**
MySQL	x34
Oracle	x16
SQL Server	x44

Рефакторінг, друга точка зору

Попередня зміна вже є зміною на перспективу: замість того, щоб модифікувати код лише для того, щоб виконати менше SQL-операторів, виконана заміна двох SQL-операторів на один. Вже відзначалося, що цикли - чудова річ (але не рідкісна) у прикладі програми. Крім того, більшість програмних змінних використовуються для зберігання даних, отриманих запитом, перш ніж надсилати їх до іншого запиту: знову ж таки регулярна властивість багатьох виробничих програм. Чи потрібно отримувати дані з однієї таблиці, щоб порівняти їх з даними з іншої таблиці, перед тим як внести їх у третю таблицю, під керівництвом нашої програми? Теоретично, всі операції можуть виконуватися лише на сервері, без необхідності кількох обмінів між прикладною програмою і сервером бази даних. Ми можемо написати збережену (stored) процедуру, щоб виконувати більшу частину роботи на сервері, і лише на сервері, або просто написати один, мабуть, відносно складний оператор для виконання завдання. Окрім того, один оператор буде менше залежним від СУБД, аніж збережена процедура:

```
try {
    PreparedStatement st = con.prepareStatement("insert into check_log(txid,"
                "conv_amount)"
                "select x.txid,x.amount*y.rate"
                " from(select a.txid,"
                        a.amount,
              + "       "
              + "       a.curr"
                    from
              + "   transactions a"

          "where a.accountid in"
          "(select accountid"
          "from area_accounts"
```

```
  "where areaid = ?)"
  "and a.txdate >= date_sub(?, interval 30 day)"
  "and exists (select 1"
+ "                  from thresholds c"
+ "                 where c.iso = a.curr"
+ "                   and a.amount >= c.threshold)) x,"
  "currency_rates y"
  "  where y.iso = x.curr"
  "  and y.rate_date=?");
...
    st.setInt(1, areaid);
    st.setDate(2, somedate);
    st.setDate(3, valuationdate);
       Wham bam
    st.executeUpdate( );
...
```

Цікаво, що єдиний запит позбавляється від двох утиліт, а це означає, що ми йдемо вниз абсолютно іншим рефакторінговим шляхом порівняно з попереднім випадком, коли рефакторізувалися пошукові функції. Перевіряються пороги шляхом приєднання transactions to thresholds, і виконуються перетворення шляхом приєднання результатів транзакцій, які перевищують порогову, до таблиці currency_rates. З одного боку, ми отримуємо ще один складний (але все ще зрозумілий) запит замість декількох дуже простих. З іншого боку, програма *FifthExample.java*, яка викликається, значно спрощується в цілому.

Представляємо варіант попередньої програми під назвою *SixthExample.java*, в якому просто SQL оператор записаний іншим способом, використовуючи більше з'єднань (joins) і менше підзапитів:

```
PreparedStatement st = con.prepareStatement("insert into check_log(txid,"
                                              "conv_amount)"
                                              "select x.txid,x.amount*y.rate"
                                              " from(select a.txid,"
                                             + "              a.amount,"
                                             + "              a.curr"
                                              "from transactions a"
                                              "inner join area_accounts b"
                                              "on b.accountid = a.accountid"
                                              "inner join thresholds c"
                                             + "                   on c.iso = a.curr"
                                              "where b.areaid = ?"
                                              "and a.txdate >= date_sub(?, interval 30 day)"
                                              "and a.amount >= c.threshold) x"
                                              "inner join currency_rates y"
                                              "on y.iso = x.curr"
                                              "  where y.rate_date=?");
```

СПИСОК ЛІТЕРАТУРИ

1. M. Fauler. Refactoring. Improving the Design of Existing Code. – Addison-Wesley, 1999. – ISBN 0-201-48567-2.
2. W. Scott, P.J. Sadalage. Refactoring Databases: Evolutionary Database Design. Addison-Wesley Signature Series. 2007. 368 pp. – ISBN 978-5-8459-1157-5.
3. S. Faroult, P. L'Hermite. Refactoring SQL Applications. O'Reilly Media Inc. 2008. – ISBN 978-0-596-51497-6.
4. J. Nummenmaa, A. Seppi, P. Thanisch. Automating support for refactoring SQL databases/ University of Tampere, Department of Computer Sciences, 2014.
5. W.F. Opdyke, R.E. Johnson. (September 1990). "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems". Proc. Of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA). ACM.
6. S.W. Ambler (2003) Agile Database Techniques: Effective Strategies for the Agile Software Developer. New York: John Wiley & Sons.