Міністерство освіти і науки України
Сумський державний університет

4561 Методичні вказівки
до практичних занять із дисципліни
«Реінжиніринг і верифікація програмного забезпечення»
на тему «Пошук недоліків коду й використання
інструментів рефакторингу»
для студентів спеціальності 122 *«Комп'ютерні науки»*
всіх форм навчання

Суми
Сумський державний університет
2019

Методичні вказівки до практичних занять із дисципліни «Реінжиніринг і верифікація програмного забезпечення» на тему «Пошук недоліків коду й використання інструментів рефакторингу» для студентів спеціальності 122 «Комп’ютерні науки» всіх форм навчання / укладач В. В. Москаленко. – Суми : Сумський державний університет, 2019. – 37 с.

Кафедра комп’ютерних наук

ЗМІСТ

ВСТУП

Поняття «реінжиніринг» програмного забезпечення досі не є загальноприйнятим. Здебільшого під реінжинірингом (англ. software reengineering) розуміють повторну реалізацію успадкованої програмної системи для підвищення зручності її використання, супроводження, розширення функціонала чи впровадження оновлених технологій.

Виконання реінжинірингу часто неможливе без виконання масштабних рефакторингів (англ. refactoring) – послідовностей невеликих еквівалентних (тобто зберігаючих поведінку) перетворень. Більш формалізовано рефакторинг можна визначити як процес змінювання внутрішньої структури програми, що не зачіпає її зовнішньої поведінки, для полегшення розуміння її роботи й функціонального масштабування.

Виконання процесів рефакторингу й реінжинірингу без допущення жодних помилок – справа надзвичайно непроста. Тому на практиці застосовують методи валідації й верифікації. Вони необхідні для перевірки програмного забезпечення чи його частин на коректність реалізації поставленого завдання порівнянням із затребуваними властивостями. Але варто зазначити, що в процесі верифікації перевіряється правильність роботи функціонала програмного продукту, а в процесі валідації – відповідність функціонала програмного забезпечення поведінці, на яку очікував чи мав на увазі користувач або замовник. Тому інженери-розробники програмного забезпечення більше зосереджуться на завданнях верифікації.

Основним етапом верифікації є тестування, тобто процес виконання програми для виявлення помилки. Водночас найбільш поширеним видом тестування систем малого й середнього розмірів є модульне тестування, виконуване для кожного незалежного програмного модуля.

Метою практичної роботи є ознайомлення з основами рефакторингу та модульного тестування як процесів, що передують і супроводжують реінжиніринг програмного забезпечення. У результаті виконання практичної роботи студент повинен удосконалити знання з об’єктно-орієнтованої мови java, набути навичок пошуку недоліків програмного коду, використання основних методів рефакторингу й розроблення модульних тестів.

1 КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Мета роботи

Мета роботи – у набутті навичок пошуку недоліків коду, ознайомлення з бібліотекою модульного тестування JUnit та інструментами рефакторингу в середовищі розроблення Intellij IDEA.

1.2 Причини виникнення недоліків коду

Під час розроблення будь-який ідеальний код, що ще називають чистим, може погіршитися, після чого його вважають брудним.

Однією з причин забруднення коду є тиск зі сторони бізнесу, що примушує прискорювати розроблення. Це може призводити до виникнення заплаток і розпорок, що приховуватимуть недороблені частини проекту.

Інколи проект набуває монолітного вигляду, замість структури з окремих пов’язаних модулів. Водночас зміни однієї частини проекту зачіпатимуть інші, що ускладнює командне розроблення.

Відсутність тестового коду призводить до втрати негайного зворотного зв’язку й мотивує до швидкого, але ризикованого правлення коду.

Також джерелом забруднення коду може бути постійна зміна вимог до проекту, що призводить до застаріння частини коду, нагромадження окремих частин коду.

Злиття змін воєдино після тривалого розроблення декількох гілок проекту також нерідко призводить до забруднення коду, тому, що зміни робили ізольовано.

На чистоту коду істотно впливають наявність документації, взаємодія між членами команди, контроль за додержанням стандартів кодування й компетентність.

1.3 Види недоліків коду

1.3.1 Основними недоліками коду, не пов'язаними з архітектурою модуля, є недоліки форматування, іменування сутностей, документування та локального використання змінних.

Програмний код описує певні програмні структури ієрархії. Для полегшення роботи з кодом текст програми потрібно відформатувати так, щоб підкреслити структуру програми. Існують різні стилі форматування, проте в команді розробників зазвичай домовляються про використання певного загального для всіх стилю. Це полегшую спільну роботу над програмним кодом. Виправлення форматування здебільшого можна зробити автоматичними засобами без змін у роботі програми. У java під час форматування коду прийнято користуватися рекомендаціями, зазначеними в документі Java Code Conventions.

Програмний код завжди містить певні компоненти, що мають імена. Іменування є творчою дією, але імена необхідно створювати з використанням термінів проблемної області й згідно з правилами форматування коду. Тобто програмні елементи потрібно іменувати, базуючись на тих сутностях проблемної області, що вони описують. Наприклад, змінні повинні мати імена не за своїм типом (число, масив, дані, список), а за значенням, що вони зберігають у термінах проблеми (заробітна плата, кількість робітників, студенти, податкові ставки). Загальні принципи форматування імен методів і змінних, прийняті в багатьох об'єктноорієнтованих мовах програмування, грунтуються на так званій «верблюжій нотації» (англ. Camel Case):

– назви з одного слова необхідно писати малими літерами;

– назви, з двох і більше слів, потрібно писати разом малими літерами, за винятком букв на межі слів (їх пишуть великими літерами);

– під час іменування змінних використовують іменники;

– під час іменування масивів і колекцій – іменник у

множині;

– ім’я методу починається з дієслова, за винятком конструктора, ім’я якого пишуть із великої літери.

У багатьох стилях форматування програмного коду поширеною практикою є використання знаку підкреслення:

– назва констант складається з великих літер, де слова чи їх скорочення розділяються знаком підкреслення;

– імена полів класу починаються зі знаку підкреслення.

Під час написання документації програмного забезпечення необхідно знаходити баланс між відсутністю документації (що робить роботу з продуктом дуже важкою) та передокументованістю (за якої важливі моменти ховаються між очевидними, що також не полегшує роботи). Тому документація повинна вказувати на важливі моменти: призначення елемента, нестандартні підходи, вимоги та обмеження, описання метафори (порівняння роботи модуля із чимось знайомим користувачеві). Документацію необхідно починати писати до написання програмного компоненту.

Програмні системи є доволі складними й чим більше зв’язків між певними контекстом і зовнішнім середовищем, тим простіше зрозуміти та внести корективи в програму. Проте одним із простих прикладів зайвих зв’язків є змінні, наявні в ширшому контексті, ніж їх використовують. Наприклад глобальні змінні, що пов’язують функції для роботи з ними з глобальним контекстом, таким чином ускладнюючи їх. Змінні необхідно описувати якомога ближче до того місця, в якому їх використовують.

Для оцінювання необхідності проведення рефакторингу потрібно оцінити якість коду, для цього можна визначити певні випадки, що мають «поганий присмак». Використання таких підходів може й не бути приводом для рефакторингу, але чим більше таких моментів у коді, тим гірше «присмак» і тим вірогідніша необхідність проведення рефакторингу. Розглянемо основні приклади коду з «присмаком».

1.3.2 Дублювання коду – використання скопійованого коду в декількох місцях. Найпростішим прикладом дублювання коду є наявність одного й того самого виразу в декількох методах одного класу. Рефакторинг полягатиме у виділенні нового методу (англ. Extract Method). Іншим прикладом є наявність одного й того самого виразу в методах двох підкласів одного рівня. У такому разі, крім виділення методу, застосовують підняття поля (англ. Pull Up Field). Якщо фрагменти коду подібні, але не збігаються повністю, використовують формування шаблону метода (англ. Form Template Method). Якщо фрагменти коду виконують одну й ту саму функцію, але реалізуються різними алгоритмами, то обирають один алгоритм і використовують заміщення алгоритму (англ. Substitute Algorithm). У разі дублювання коду у двох різних класах використовують виділення класу (англ. Extract Class).

1.3.3 Довгий список параметрів методу (більший ніж чотирьох-п’яти). В об’єктно-орієнтованих програмах списки параметрів значно коротші, ніж у процедурних програмах, тому що методу часто передають не все необхідне, а лише стільки, щоб метод міг одержати доступ до всіх потрібних йому даних. Довгий список параметрів часто усувають заміною параметра викликом методу (англ. Replace Parameter with Method) у вже відомого об’єкта, що вже відомий і надає доступ до необхідних даних. Цей об’єкт може бути полем класу (англ. Preserve Whole Object). Інколи елементи даних не мають логічного об’єкта, тому вводять об’єкт-параметр (англ. Introduce Parameter Object).

1.3.4 Довгий метод – занадто великий метод (функція, процедура). Ознаками довгого методу можуть бути коментарі, що пояснюють його окремі частини, та значна семантична відстань між тим, що робить метод, і тим, як він це робить. Рефакторинг довгих методів – це виділення методів, ім’я яких може базуватися на коменаріях. Проте велика кількість параметрів і тимчасових змінних ускладнює проведення ефективного рефакторингу. Усунення тимчасових параметрів рекомендовано здійснювати за допомогою заміни тимчасової

змінної викликом метода (англ. Replace Temp with Query). Зменшення довгих списків параметрів було розглянуто вище. У складних зразках виконують заміну методу об'єктом методу (англ. Replace Method with Method Object). Інколи ознаками довгого методу є умовні оператори, до яких потрібно застосувати декомпозицію (англ. Decompose Conditional).

1.3.5 Великий клас –клас, що містить занадто багато функцій та атрибутів. За необхідності виділяють клас, підклас. Під час з'ясування, які клієнти використовують клас, можна застосувати й виділення інтерфейсу.

1.3.6 Модифікації, що розходяться – наслідок унесення в клас частих змін із різних причин. Кожен клас повинен мати свою чітку зону відповідальності. Його потрібно змінювати відповідно до змін у такій зоні. Інколи в класі можна знайти місця, в які постійно вносяться модифікації внаслідок зміни умов у певній зоні відповідальності. Цю частину класу варто виділити в окремий клас (англ. Extract Class).

1.3.7 Постріли дробиною – приклад, коли під час унесення однієї зміни необхідно модифікувати багато класів. У такому разі застосування переміщення методів (англ. Move Method) і переміщення полів (англ. Move Field) можуть звести всі зміни в один клас. Якщо серед наявних класів немає підходящого, необхідно створити новий клас. Цей прийом називають убудовуванням класу (англ. Inline Class).

1.3.8 «Заздрісні» методи – методи, що частіше використовують члени інших класів, ніж члени класу, якому вони належать. Предметом «заздрості» найчастіше є дані. «Заздрісні» методи переміщують у клас, дані якого він більше використовує. Іноді «заздрість» властива лише частині методу (її рекомендовано виділити в окремий метод), а іноді різні частини методу заздрять різним класам (ці частини варто виділити в окремі методи й перемістити в різні класи).

1.3.9 Групи даних – три, чотири й більше елементів даних, що разом (групами) з'являються в багатьох місцях: поля в парі класів, параметри в декількох сигнатурах методів. Такі групи

даних варто перетворити на самостійний клас, під час передавання групи даних у сигнатуру метода використовувати об'єкт-параметр (англ. Introduce Parameter Object), або зберегти об'єкт групи даних у полі класу (англ. Preserve Whole Object). В утворені класи можуть бути переміщеними методи з інших класів, у яких вони є «заздрісними».

1.3.10 Одержимість елементарними типами – часте використання простих типів там, де більше підходять записи або об'єкти. За необхідності заміняють значення об'єктом (англ. Replace Data Value with Object). Під час використання елементарного типу для подання коду типу, що не змінює поведінки програми, використовують заміну коду типу класом (англ. Replace Type Code with Class). У разі умовних операторів, що залежать від коду типа, можна використати заміну коду типу підкласами (англ. Replace Type Code with Subclasses) чи станом/стратегією (англ. Replace Type Code with State/Strategy). Групи примітивних полів чи групи параметрів примітивного типу, як розглянуто вище, можна виділити в окремий клас.

1.3.11 Оператор типу switch – використання switch-конструкцій для виконання різних дій в залежності від типу чогось. Часто один і той самий блок switch беребуває в різних місцях програми й тому виникає необхідність модифікації кожного з них під час додавання нового варіанта. В об'єктноорієнтованих програмах заміна блоків switch-case на поліморфізм, здебільшого, дозволяє покращити організацію коду. Якщо switch перемикає режим виконання програми кодом типу, то варто виконати заміну кодування типу підкласами чи заміну кодування типу станом/стратегією.

1.3.12 Паралельні ієрархії наслідування – випадок, коли під час створення нового підкласу однієї ієрархії необхідно створити підклас іншої. Ознакою паралельних ієрархій є збігання префіксів імен класів у двох ієрархіях класів. Усунення дублювання, спричиненого цим, полягає в реалізації посилання екземплярів однієї ієрархії на екземпляри іншої переміщенням методів і поля.

1.3.13 Ледачий клас – клас, що має занадто мало функцій або майже не використовуваний. Функціонал ледачого класу варто перенести в інший клас. У разі підкласів із недостатнім функціоналом варто згорнути ієрархію.

1.3.14 Теоретична загальність – наявність конструкцій, що не є необхідними зараз, але можливо знадобляться в майбутньому. Параметри не використовуваного методу потрібно видаляти (англ. Remove Parameter). Методи з дивними й абстрактними іменами варто перейменовувати (англ. Rename Method). Малокорисних абстрактних класів необхідно позбуватися згортанням ієрархії (англ. Collapse Hierarhy).

1.3.15 Тимчасове поле – поле (або глобальна змінна), не використовувана весь час життя об'єкта, а лише в деякий період або за деяких обставинах. Тимчасове поле й весь код, що з ними функціонує, можна помістити у свій власний клас за допомогою виділення класу. Видалити код перевірки наявності значень у тимчасових полях можна за допомогою введення Null-об'єкта для створення альтернативного компонента.

1.3.16 Ланцюжки повідомлень – конструкції, в яких клієнт бере в одного об'єкта інший, у того ще один і так далі, поки не одержить дійсно потрібного йому об'єкта. Будь-які зміни проміжних зв'язків означають необхідність модифікації клієнта. Рефакторинг полягає в переміщенні фрагмента коду, що використовує потрібний об'єкт, вниз за ланцюжком виділенням і переміщенням методу.

1.3.17 Посередник – клас, що здебільшого передає повідомлення від клієнта до іншого об'єкта й більше нічого не робить. Посередників потрібно позбавлятися, наприклад додаванням необхідних методів безпосередньо в програму клієнта. Проте не варто видаляти посередників, введених для позбавлення небажаної залежності між класами або в інших спеціально передбачених ситуаціях.

1.3.18 Недоречна близькість – пара (або більше) класів, що занадто часто звертаються до приватних даних один одного. Таку близькість варто усувати переміщенням полів і методів. У

разі наявності в класів спільних інтересів можна виділити окремий клас, у який переміститься спільна частина інтересів.

1.3.19 Альтернативні класи з різними інтерфейсами – класи, що виконують подібні функції, але мають різні інтерфейси. Рефакторинг полягатиме в переміщенні й перейменовуванні методів доки інтерфейси не стануть однаковими. Якщо лише частина функціональності класів ідентична, варто виділити цю частину як суперклас.

1.3.20 Неповнота бібліотечного класу – бібліотеки з часом перестають задовольняти вимоги користувачів, проте внесення змін до бібліотеки недоступне. Додавання методів до бібліотечного класу реалізується введенням зовнішнього методу (англ. Introduce Foreign Method). Значні зміни поведінки бібліотечних класів виконуються введенням локального розширення (англ. Introduce Local Extension).

1.3.21 Клас даних – класи, що містять лише публічні поля. До публічних полів варто застосувати інкапсуляцію і дозволити доступ тільки через гетери та сетери. Рефакторинг передбачає аналіз клієнтського коду з метою виявлення функціональності, більш властивої цьому класу. Таку функціональність варто перемістити шляхом виділення методів та їх переміщенням.

1.3.22 Відмова від спадкування – підклас використовує лише малу частину наслідуваних методів і властивостей суперкласу, що є ознакою неправильної ієрархії. При цьому непотрібні методи або не використовуються, або перевизначені й викидають виключення. У цьому разі варто позбавлятися від відношення наслідування між цими класами, застосувавши заміну наслідування делегацією.

1.3.23 Коментарі – коментарі не є поганою практикою, але поганий код часто має надлишок коментарів для компенсації, тому вони можуть бути знаком «поганого» коду.

1.4 Рефакторинг

1.4.1 Рефакторинг – це, насамперед перетворення брудного коду на чистий. Чистий код характеризується більшою

очевидністю для інших програмістів, ніж брудний код. Також чистий код не містить дублювання. Чистий код звжди містить мінімум класів та інших рухомих частин.

1.4.2 Рефакторинг покращує нефункціональні атрибути програмного забезпечення. У будь-якому великому програмному проекті настає час, коли потрібно рефакторити ваш код. Часто корисно рефакторити ваш код багато разів під час розроблення проекту. Під час вашого прогресу ви думаєте про кращі способи структурування вашого коду, а також, можливо, кращі алгоритми або структури даних, використовуваних для підтримання функціональності та швидкості. У цій роботі ми зосередимося на техніках рефакторингу. Наявність тестового коду полегшує рефакторинг, оскільки ліквідує страх щось зламати, тому тести є ознакою добре продуманого проекту. Якщо ви, як інженер-програміст, боїтеся переробити свій код, це, мабуть, тому, що не написано достатньо тестів, щоб відчувати себе комфортно, змінюючи код вашого проекту.

У кінцевому підсумку рефакторинг дозволяє прискорити процес розробки коду, оскільки більша частина часу припадає не на додавання нових функцій, а на пошук та виправлення помилок, пошук дублювання коду і т. п.

2 ЗАВДАННЯ ДО ПРАКТИЧНОЇ РОБОТИ

2.1 Підготовча робота

Завантажте сім java-файлів для їх подальшого рефакторингу (рис. 2.1). Скопіюйте їх потім в папку проекту src, що містить вихідний код проекту (рис. 2.2).

Швидше за все ви отримаєте негайні помилки, які свідчать про те, що JUNIT-бібліотека не була знайдена. Щоб виправити це: наприклад, відкрийте один із двох тестових класів "VideoGameRentalTest.java". Ви побачите червоний виділений код поряд із "extends" над словом TestCase, клацніть правою кнопкою миші на червоній лампочці та виберіть "Find JAR on

web" (рис. 2.3). Зі списку виберіть junit 4.7.jar. Розмістіть jar у папці проекту та натисніть "OK".

Тепер давайте запустимо ці тести, щоб переконатися, що наш проект працює як очікується. Перейдіть до VideoGameRentalTest.java, відкрийте файл і клацніть правою кнопкою миші будь-де на панелі коду.

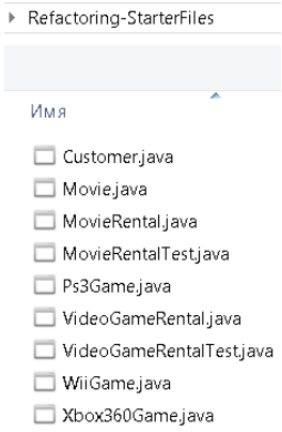

Рисунок 2.1 – Каталог із файлами проекту

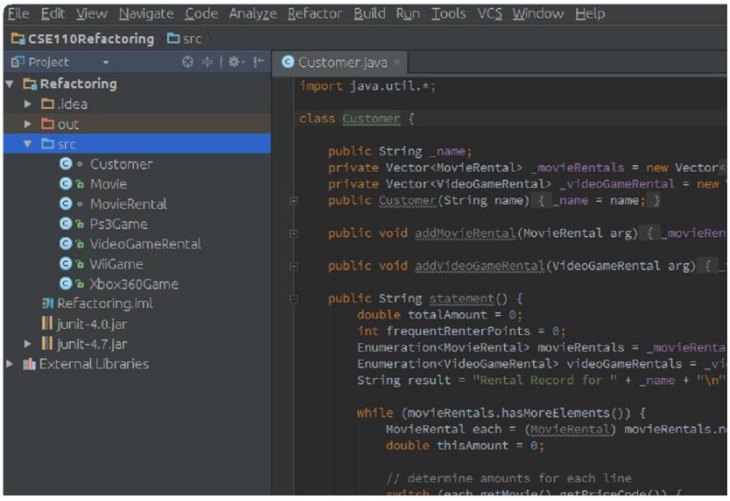

Рисунок 2.2 – Вигляд проекту після додавання файлів

Ви побачите опцію Run MovieRentalTest, і вона повинна виконатися без помилок (рис. 2.4). Повторіть це для файла під назвою VideoGameRentalTest.java.

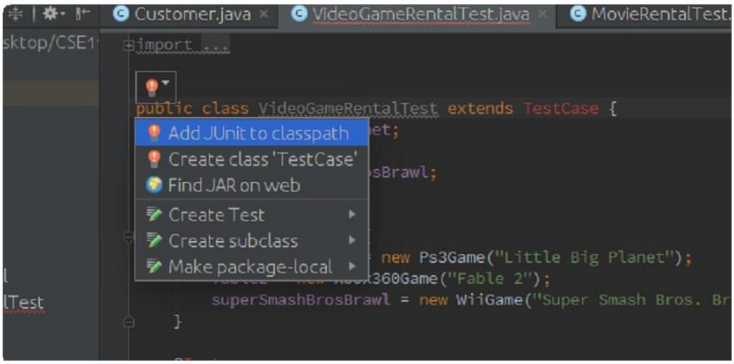

Рисунок 2.3 – Додавання бібліотеки Junit до системного шляху

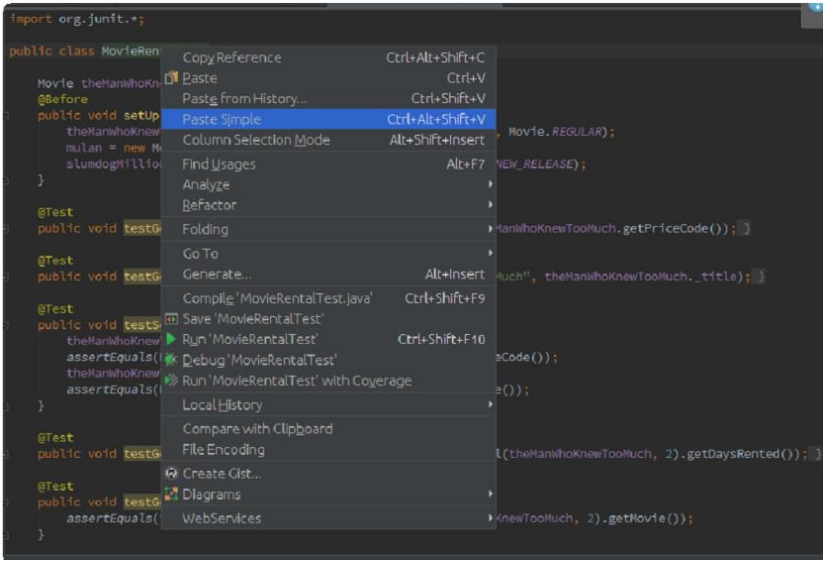

Рисунок 2.4 – Вигляд контекстного меню тестового класу для запуску його на виконання

2.2 Використання JUnit-тестів

Рефакторинг не повинен змінювати будь-яку функціональність коду, ви просто повинні очистити кодову базу. Таким чином, життєво важливо, щоб у вас був набір одиничних тестів, які перевіряють, що ваш рефакторинг нічого не порушив. Якщо ви подивитеся в файл MovieRentalTest.java, то побачите декілька вже написаних тестів і декілька заголовків. Коли клієнт замовляє фільм, вони вказують, скільки днів орендують цей фільм. Залежно від типу фільму (REGULAR, CHILDRENS або NEW_RELEASE) вони сплачують іншу ціну та одержують різну кількість балів за часту оренду (табл. 2.1).

а

Таблиця 2.1 – Ціна і кількість балів за аренду фільму

	Ціна	Бал за часту оренду
REGULAR	2$ для перших 2 днів і додатково 1.5$ за кожен з наступних днів	1 бал/аренда
CHILDRENS	1,5$ для перших 3 днів і додатково 1.25$ за кожен з наступних днів	1 бал/ аренда
NEW_RELEASEE	3$/день	1 бал/аренда за 1 день; 2 бали – за 2 і більше днів

Зверніть увагу на три оголошення методів: testStatementRegularMovieOnly(), testStatementChildrensMovieOnly(), і testStatementNewReleaseOnly(). Ви повинні заповнити ці методи тестами, які перевірятимуть, що відбувається, коли клієнт орендує зазначений тип фільму на різні дні. Підказка: для перевірки на коректність додавання кількості клієнтів та кількості балів за часту оренду за допомогою методу statement () використовуйте метод assertEquals (), наприклад:

```
Customer janeDoe = new Customer( "Jane Doe" );
janeDoe. addMovieRental( new MovieRental( slumdogMillionaire, 1) ) ;
assertEquals( " Rental Record for Jane Doe\n" +
       " \tSlumdog Millionaire\t3. 0\n" +
       " Amount owed is 3. 0\n" +
       " You earned 1 frequent renter points" ,
       janeDoe.statement());
janeDoe.addMovieRental( new MovieRental( slumdogMillionaire, 2) );
assertEquals( " Rental Record for Jane Doe\n" +
       //the first movie rental above
       " \tSlumdog Millionaire\t3. 0\n" +
       //the current movie rental
       " \tSlumdog Millionaire\t6. 0\n" +
       " Amount owed is 9. 0\n" +
       //notice it is the 1 point from before and 2
       //points for 2 or more days for the current movie.
       " You earned 3 frequent renter points" ,
       janeDoe.statement() );
```

Ви повинні помітити в цьому коді те, що другий виклик statement () показує як перший, так і другий фільм, оскільки перший фільм усе ще перебуває в оренді в janeDoe's Movie Rentals. Обов'язково враховуйте це під час написання інструкцій щодо виведення у своїх тестах! Ви повинні мати щонайменше 3 орендовані одиниці в кожному тестовому варіанті, щоб забезпечити покриття різних випадків (кількість днів) для кожного типу оренди.

2.3 Використання автоматичних інструментів рефакторингу

IntelliJ являє собою безліч відмінних автоматичних інструментів рефакторингу, що займають більшу частину періоду рефакторингу. Зазвичай, жоден із цих інструментів не є досконалим, і вам усе одно доведеться виконати деякий рефакторинг вручну, але вони можуть скоротити загальний обсяг часу.

2.3.1 Перейдіть до методу statement () у класі Customer.java і зверніть увагу, що для кожного фільму у векторі ми запускаємо оператор switch. Щоб IntelliJ автоматично здійснював екстракцію методу, виділіть блок коду, починаючи з double thisAmount = 0, і всього оператора switch (рис. 2.5).

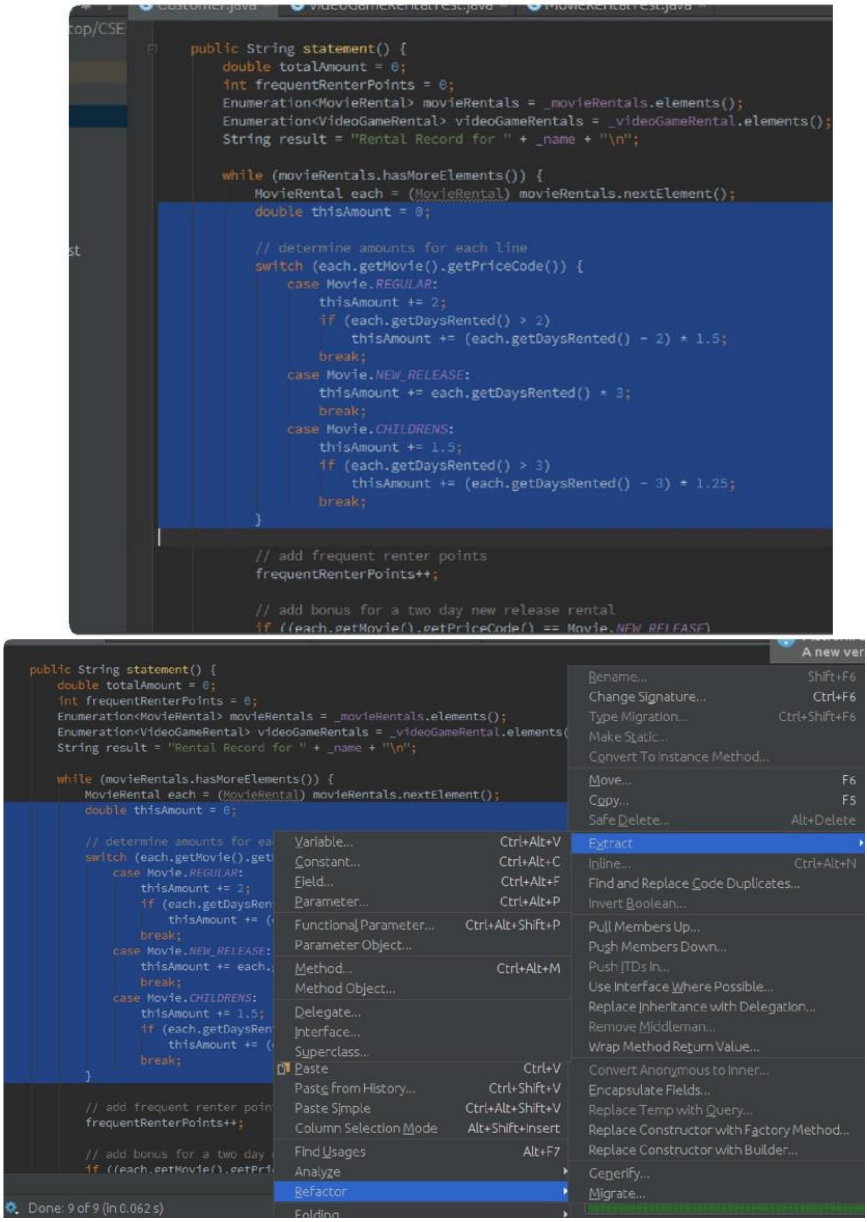

Рисунок 2.5 – Ілюстрація до рефакторингу Exctract Method

Тепер виберіть Refactor => Extract => Method… і для «Method» уведіть amountFor (рис. 2.6). Зверніть увагу, що IntelliJ автоматично створила цей метод у кінці файла.

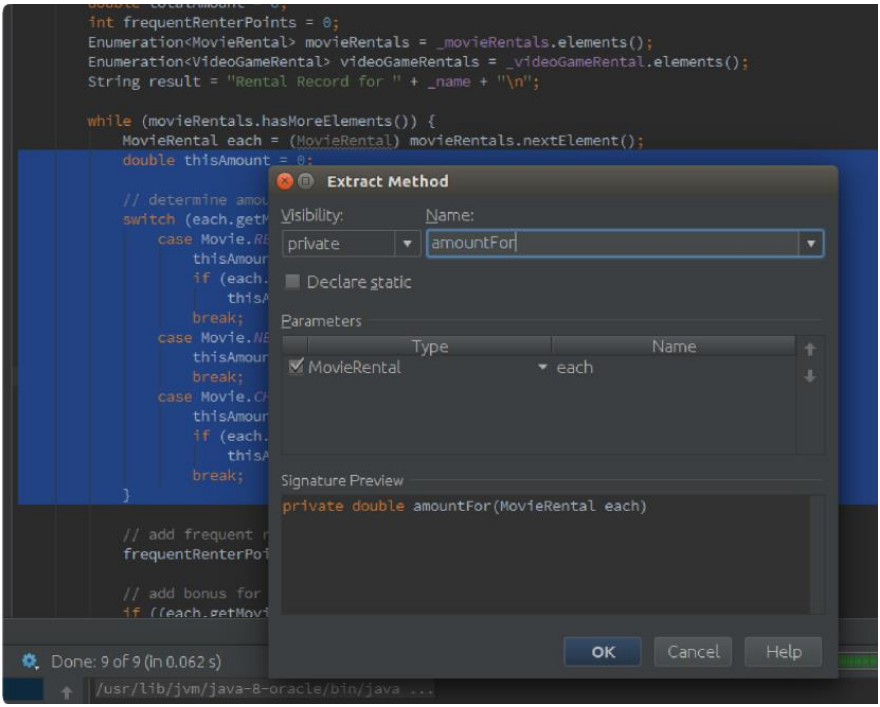

Рисунок 2.6 – Ілюстрація іменування виділеного методу

Тепер ви повинні побачити код, замінений викликом методу (рис. 2.7).

2.3.2 Імена змінних усередині нашого нового методу повинні бути перейменовані, оскільки вони не містять достатньої інформації про те, що вона являють собою. Для цього виділіть each змінну всередині методу amountFor і перейдіть до Refactor => Rename… та введіть нове ім’я aRental (рис. 2.8).

Повторіть це і для змінної thisAmount. IntelliJ буде перейменовувати екземпляри обох змінних у межах цього блоку.

```
Enumeration<VideoGameRental> videoGameRentals = _videoGameRental.elements();
String result = "Rental Record for " + _name + "\n";

while (movieRentals.hasMoreElements()) {
    MovieRental each = (MovieRental) movieRentals.nextElement();
    double thisAmount = amountFor(each);

    // add frequent renter points
    frequentRenterPoints++;

    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
            && each.getDaysRented() > 1) frequentRenterPoints++;
```

Рисунок 2.7 – Ілюстрація коду з використанням виділеного методу

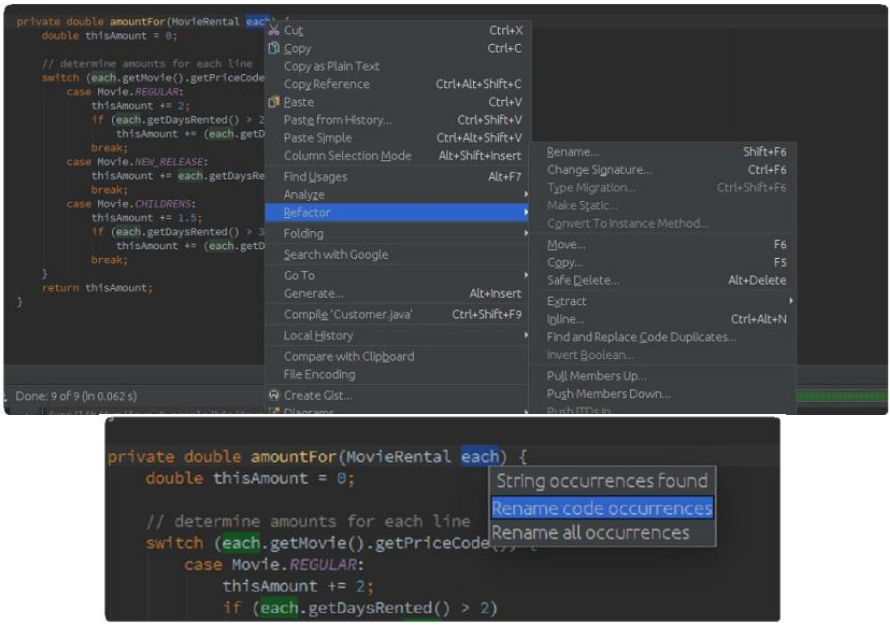

Рисунок 2.8 – Ілюстрація кроків рефакторингу Rename

2.3.3 Оскільки метод amountFor () не використовує нічого в класі Customer і використовує методи всередині класу MovieRental, ми повинні перемістити цей метод. Виділіть метод

amountFor і перейдіть до Refactor => Move... (рис. 2.9). IntelliJ досить розумна, щоб автоматично визначити, що цей метод необхідно перемістити до класу MovieRental, тому не потрібно виконувати жодної роботи, крім зміни назви методу на щось більш читабельне. Давайте назвемо цей метод getCharge і натиснемо OK та Refactor.

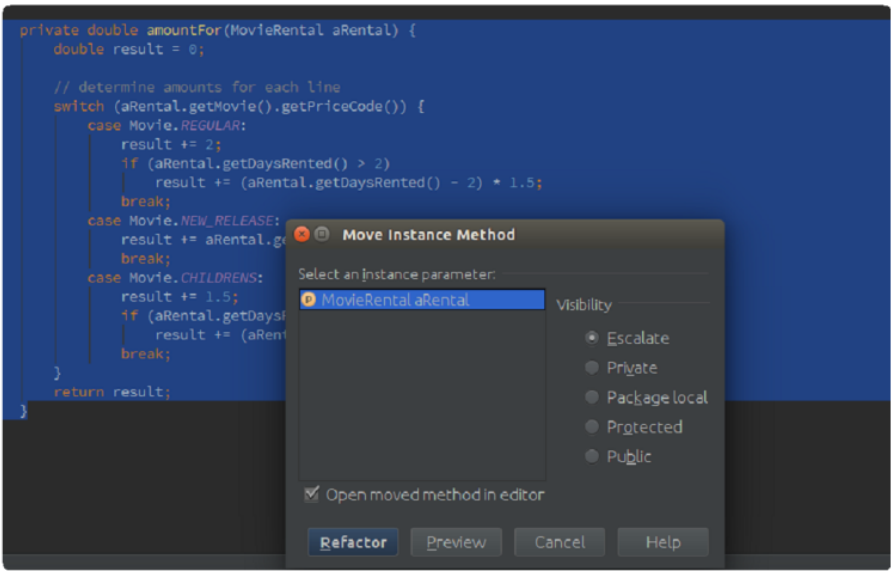

Рисунок 2.9 – Ілюстрація переміщення методу

2.3.4 Зверніть увагу, що в класі Customer ми безпосередньо одержимо доступ до поля _title фільму, що не є гарною інкапсуляцією, тому давайте продовжимо і змінимо це. Виділіть поле _title та перейдіть до Movie.java, виберіть поле _title Refactor => Encapsulate Field ... (рис. 2.10). Збережіть усе в командному рядку та натисніть Refactor. Це замінить усі посилання на поле _title в усьому нашому коді, і відпаде необхідність шукати всі посилання.

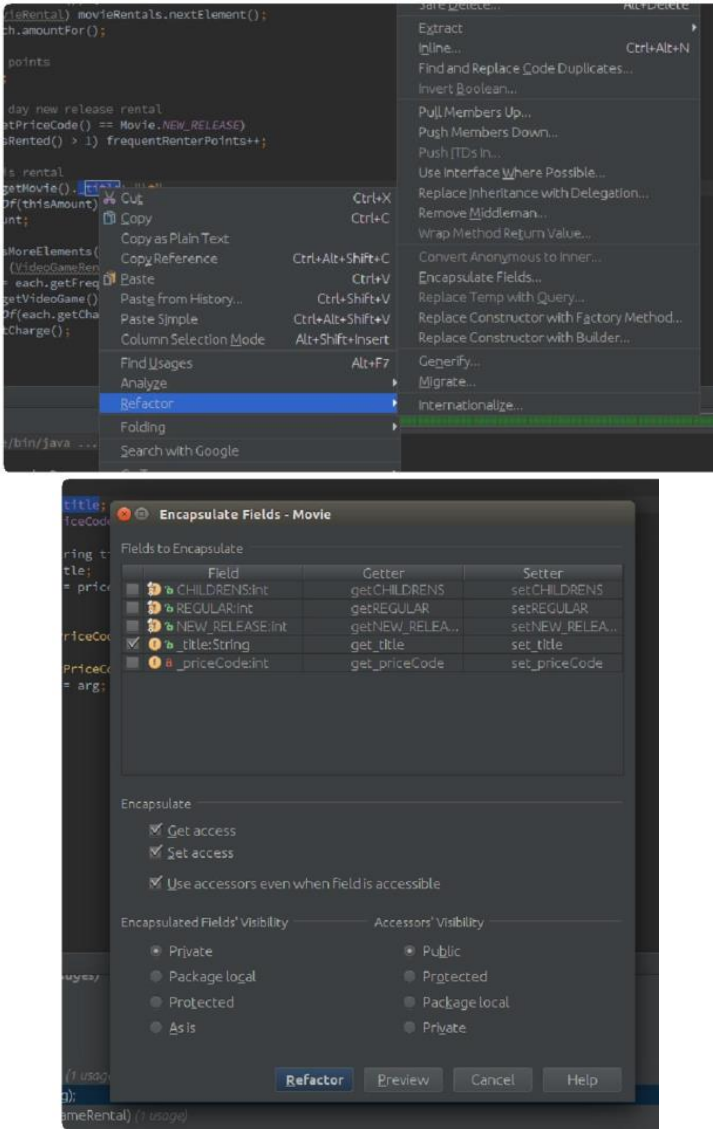

Рисунок 2.10 – Ілюстрація внесення інкапсуляції поля

2.3.5 Змінна thisAmount, здається, використовується лише в 2 місцях коду, і нам було б краще не оголошувати змінну взагалі. Коли ми вставляємо змінну, ми обмінюємося всіма

посиланнями на цю змінну з її значенням. У цьому разі ми будемо замінювати всі екземпляри thisAmount за допомогою each.getCharge (). Find та replace роблять те саме, але інструмент рефакторингу більш обізнаний із синтаксисом Java і замінює лише посилання на thisAmount у заголовку методу. Виділіть будь-який thisAmount (де завгодно) і перейдіть до Refactor => Inline... та виберіть «Добре» (рис. 2.11).

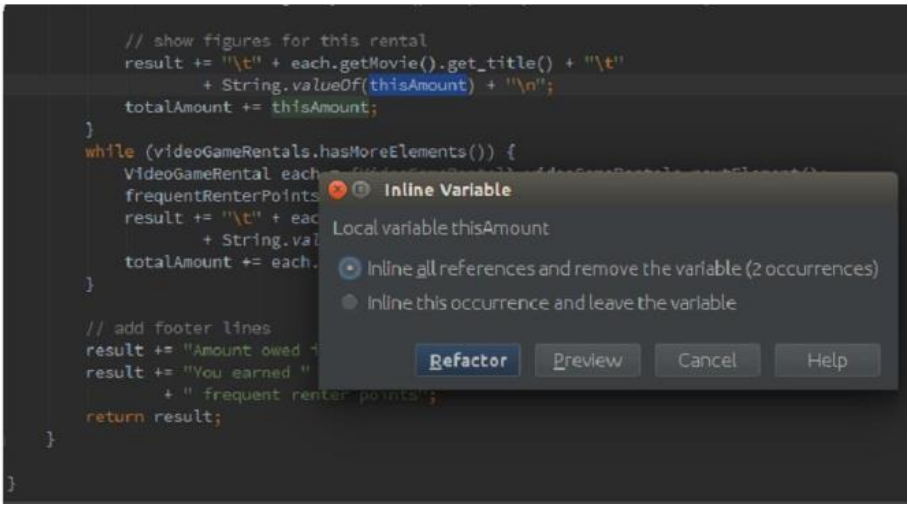

Рисунок 2.11 – Ілюстрація рефакторингу Inline Variable

2.3.6 Давайте тепер витягнемо розрахунок для frequentRenterPoints у свій власний метод. Зробіть це, вибравши команду frequentRenterPoints ++ і якщо statement if після «//add bonus…». Потім виконайте Refactor => Extract Method… знову рисунок 2.12. На цей раз назвемо метод getFrequentRenterPoints.

2.3.7 Давайте дещо очистимо цей метод, тому що ми не повинні дійсно переходити до frequentRenterPoints. Оскільки ми вже зараз повертаємо нове значення frequentRenterPoints, то також можемо просто змінити метод, щоб повернути значення точок, одержаних лише в цьому фільмі, а замість цього додати

повернене значення до поточного значення змінної frequentRenterPoints замість змінної.

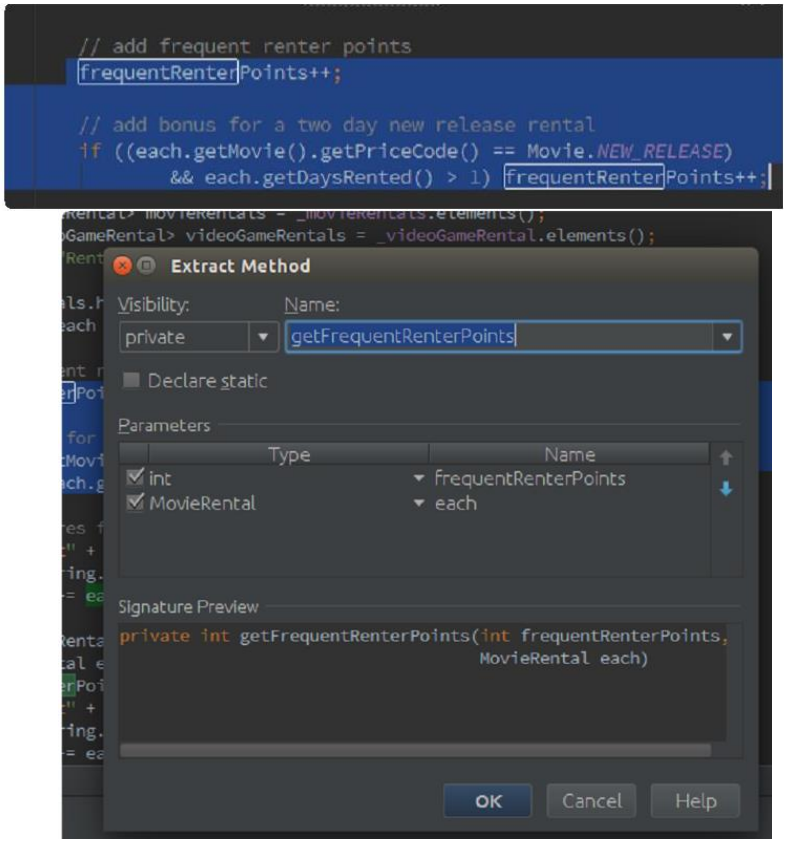

Рисунок 2.12 – Ілюстрація екстракції та перейменування методу

Тепер давайте видалимо параметр frequentRenterPoints із методу getFrequentRenterPoints. Для цього виберіть метод getFrequentRenterPoints і перейдіть до Refactor => Change Signature. Виділіть параметр frequentRenterPoints та натисніть знак мінуса. Зверніть увагу, все, що ми створили, стало червоним? Можна просто виправити, замінивши перший рядок (frequentRenterPoints++) у методі на int frequentRenterPoints = 1.

2.3.8 Тепер потрібно перевірити наші нові методи, щоб ми нічого не порушили. Клацніть правою кнопкою миші MovieRentalTest і натисніть run. Внизу з'явиться повідомлення про помилку. Наш рефакторинг щось зламав! Натисніть на помилку, щоб одержати нижній екран (рис. 2.13).

Рисунок 2.13 – Ілюстрація про помилку після рефакторингу

2.3.9 Зважаючи на описання помилки, здається, ми неправильно накопичуємо бали за часту оренду. Існує одна необхідна зміна символа (frequentRenterPoints = getFrequentRenterpoints (кожен);) залежно від того, як ми робили виділення до цього. Виконайте зміну одного символа. Підказка: перечитайте кроки, пов'язані з нарахуванням балів за часту оренду. Ось чому нам потрібні модульні тести. Невеликі помилки можуть порушити функціональність вашої програми, і ви, ймовірно, ніколи не побачите цього.

Якщо уважніше подивитися на getFrequentRenterPoints, то здається, що цей метод має набагато більше сенсу в класі MovieRental. Тому давайте перемістимо цей метод так, як це було зроблено для amountFor / getCharge.

2.4 Підкласи і константи

Використана конструкція switch із часом стане довшою внаслідок розширення специфікації фільмів. Тому давайте замінимо змінну priceCode успадкуванням, щоб її позбутися. Зверніть увагу, що ця заміна дещо змінить використання наших класів. Функціональність залишиться там, але нам більше не буде потрібно передавати priceCode у фільм. Оскільки доводиться редагувати наші тести, це технічно не справжній

рефакторинг, але ми продовжимо цю реалізацію, оскільки вона покаже деякі корисніші функції IntelliJ.

2.4.1 Спочатку потрібно перемістити методи getCharge () і getFrequentRenterPoints() у клас Movie. Не використовуйте метод move середовища IntelliJ. Щоб заощадити ваш час, код, необхідний вам для додавання до класу фільму, наведено тут:

```
public double getCharge( int _daysRented) {
    double result = 0;
    // determine amounts for each line
    switch ( _priceCode) {
        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. 25;
            break;
    }
 return result;
}

public int getFrequentRenterPoints( int _daysRented) {
    int frequentRenterPoints = 1;
    // add bonus for a two day new release rental
    if ( ( getPriceCode( ) == Movie. NEW_RELEASE) && _daysRented > 1)
    frequentRenterPoints += frequentRenterPoints;
    return frequentRenterPoints;
}
```

2.4.2 Тепер змініть метод getCharge () і getFrequentRenterPoints() у класі MovieRental:

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

public int getFrequentRenterPoints() {
    return _movie.getFrequentRenterPoints( _daysRented) ;
}
```

2.4.3 Давайте використаємо деякі принципи об’єктноорієнтованого проектування і створимо 3 підкласи

класу Movie. Клацніть правою кнопкою миші на src folder => new => Java class Name і дайте назву першому класу RegularMovie, а потім відредагуйте клас для наслідування від Movie (рис. 2.14).

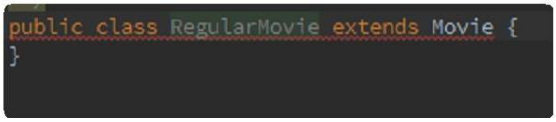

Рисунок 2.14 – Ілюстрація підкласу класу Movie

Ви можете побачити, що клас підкреслено червоним кольором. Наведіть курсор миші на червоне підкреслення та натисніть лампочку ліворуч і виберіть Create constructor matching super. Унаслідок цього буде створено конструктор для роботи з оригінальним конструктором Movie за допомогою super-методу (рис. 2.15).

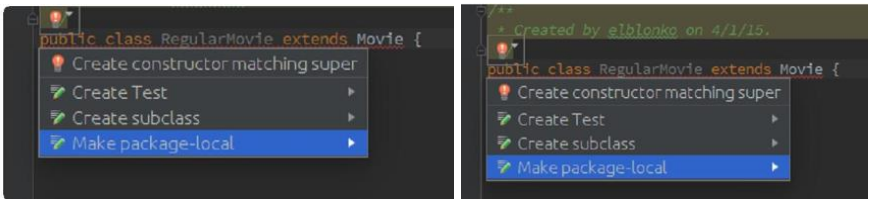

Рисунок 2.15 – Ілюстрація до процесу створення конструктора

Повторіть ці кроки під час створення 2 класів ChildrensMovie, NewReleaseMovie.

2.4.4 Оскільки клас Movie тепер має класи, що його наслідують, ми можемо просувати методи getCharge (int) і getFrequentRenterPoints (int) в його спадкоємців. Для того, щоб фактична реалізація методів getCharge (int) і getFrequentRenterPoints (int) здійснювалася в підкласах, ми повинні перемістити їх із Movie вниз за ієрархією. Виберіть метод getCharge (int) і getFrequentRenterPoints (int) і перейдіть до

Refactor => Push Member Down ... Виберіть два методи, які потрібно просунути, і натисніть праву кнопку (рис. 2.16).

У кожного з 3-х підкласів видаліть оператор switch у getCharge. Він повинен повертати вартість для відповідного типу фільму.

2.4.5 Поки ви перебуваєте в цих 3 класах, необхідно виконати видалення всіх магічних цифр. Рядки, подібні до «thisAmount += 1.5;», магічно додають якусь amount, для редагування якої нам доведеться пройти через весь код, виконуючи зміни. Найпростіше рішення полягає у їх заміні константами, які роблять набагато простішим читання. У класі ChildrensMovie виділіть число 3 і перейдіть до Refactor => Extract => Constant ...; назвіть його NUM_DAYS_BASE_CONST і переконайтеся, що встановлено прапорець Replace all occurrences для вибраного виразу з посиланнями на константу (рис. 2.17–2.18). Потім замініть залишені цифри. У кінці ви повинні мати 4 константи:

– FREQUENT_RENTER_POINTS;
– NUM_DAYS_BASE_CONST;
– LONG_TERM_RENTAL_COST;
– BASE_RENTAL_COST.

Для класу ChildrensMovie ці значення матимуть значення 1, 3, 1.25 та 1.5 відповідно. Зробіть те ж саме в RegularMovie та NewReleaseMovie.

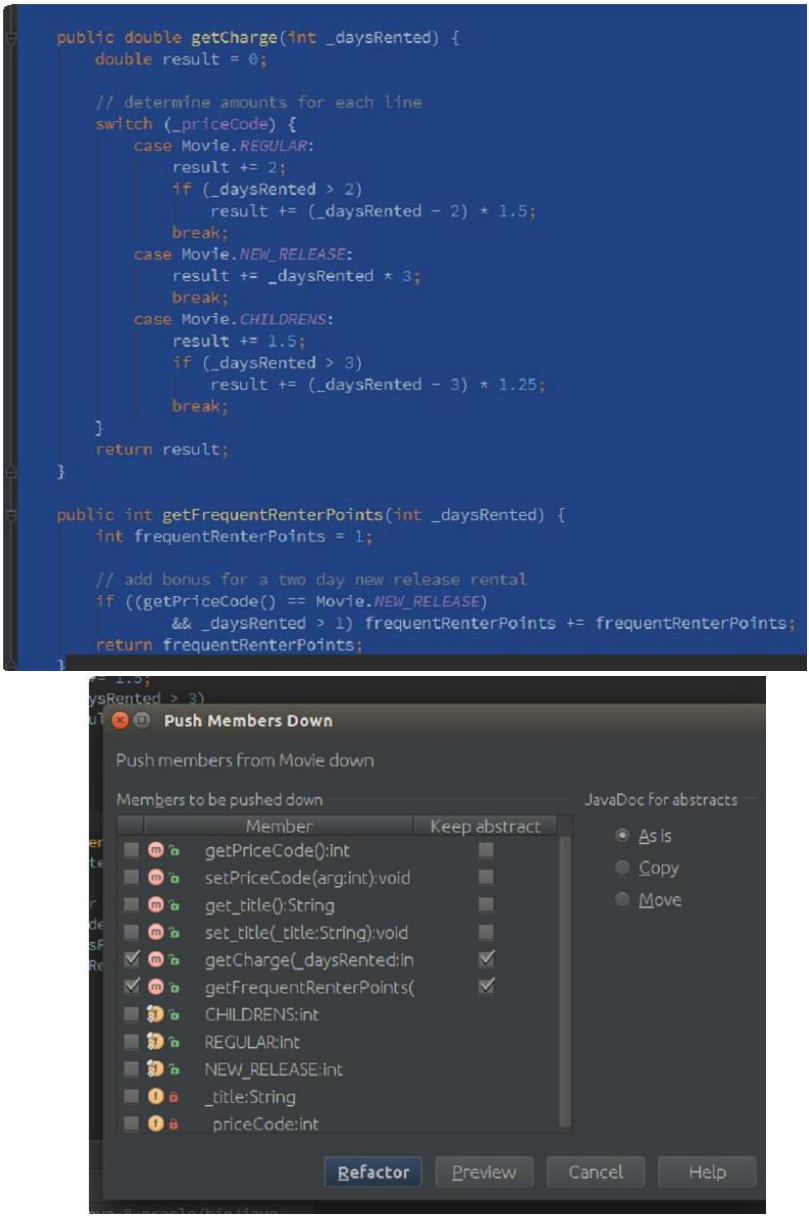

Рисунок 2.16 – Ілюстрація до рефакторингу Push Member Down

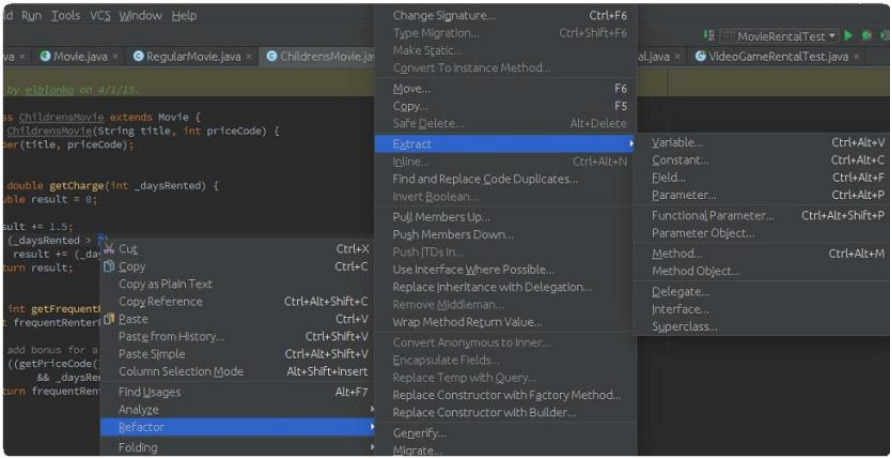

Рисунок 2.17 – Ілюстрація вибору опції Extract Constant

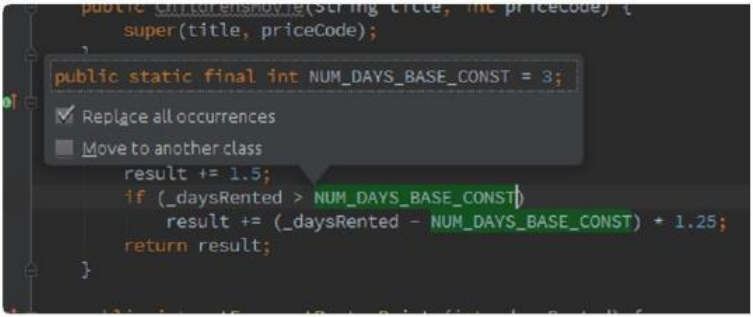

Рисунок 2.18 – Ілюстрація до рефакторингу Extract Constant

2.4.6 Оскільки getCharge тепер повертає вартість для кожного підкласу, параметр priceCode не підтримується. Видаліть всі посилання на параметр _priceCode в конструкторів усіх 3 підкласів. Видаліть всі 3 статичних константи в класі Movie, будь-яке посилання на _priceCode в конструкторі, а також the getter and setter методи. Для цього натисніть cntrl + shift + f, щоб відкрити поле пошуку всіх необхідних файлів (рис. 2.19–2.20). Вони будуть перелічені нижче.

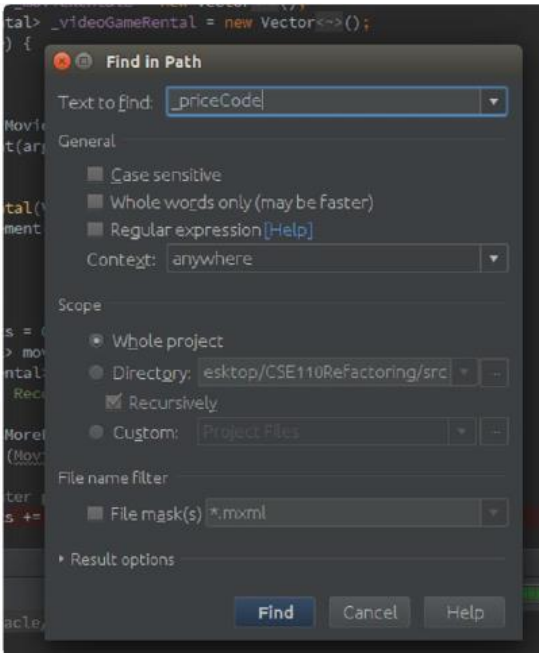

Рисунок 2.19 – Ілюстрація поля пошуку

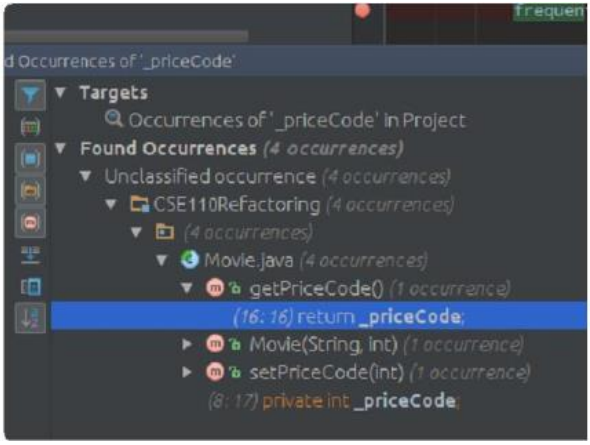

Рисунок 2.20 – Ілюстрація переліку знайдених включень

Оскільки ми тепер принципово змінили функціональність нашого коду, тести потрібно трохи переписати. Замініть setUp-метод у тестовому класі таким кодом:

```
public void setUp() {
    theManWhoKnewTooMuch =
        new RegularMovie( " The Man Who Knew Too Much" );
    mulan = new ChildrensMovie( " Mulan" );
    slumdogMillionaire =
        new NewReleaseMovie( "Slumdog Millionaire" );
}
```

2.4.7 Тепер видаліть всі тести, пов'язані з priceCode.

Перезапустіть свої тести, щоб переконатися, що всі вони виконуються без помилок. Код, який ви вставили, виправляє проблему з frequent renter, щоб вони відповідали описанню, наданому вам під час запуску. Будь ласка, змініть свої тести відповідно. Наприклад, якщо ви взяли в оренду 3 дитячих фільми, ви одержите 3 renter points (по одному на кожну оренду).

Зверніть увагу, що тести спочатку пройдуть із помилками і вказуватимуть на методи, які потрібно змінити для підтримки вашої нової реалізації, основаної на наслідуванні. Якщо ви розумієте кроки, які щойно виконали, виправлення цих помилок має бути тривіальним.

2.5 Виділення суперкласу

Поки що ми лише звернули увагу на клас Movie та все, що з ним пов'язано. Тепер звернемо увагу на 3 класи відеоігор: Ps3Game, Xbox360Game та WiiGame. Якщо поглянути на ці класи, то здається, що вони мають дуже схожі методи (насправді здебільшого дублюються), однак вони використовуються спотворено в класі VideoGameRental. Щоб виправити це, ми будемо створювати суперклас (батьківський клас) під назвою VideoGame.

2.5.1 У класі Ps3Game виберіть «Ps3Game» відразу після public class і перейдіть до Refactor => Extract => Superclass ...

Обов’язково назвіть новий клас VideoGame. Виберіть параметри, показані на зображенні нижче, щоб створити свій суперклас (рис. 2.21). Тепер змініть інші класи відеоігор xbox360 і Wii для наслідування від нового суперкласу VideoGame.

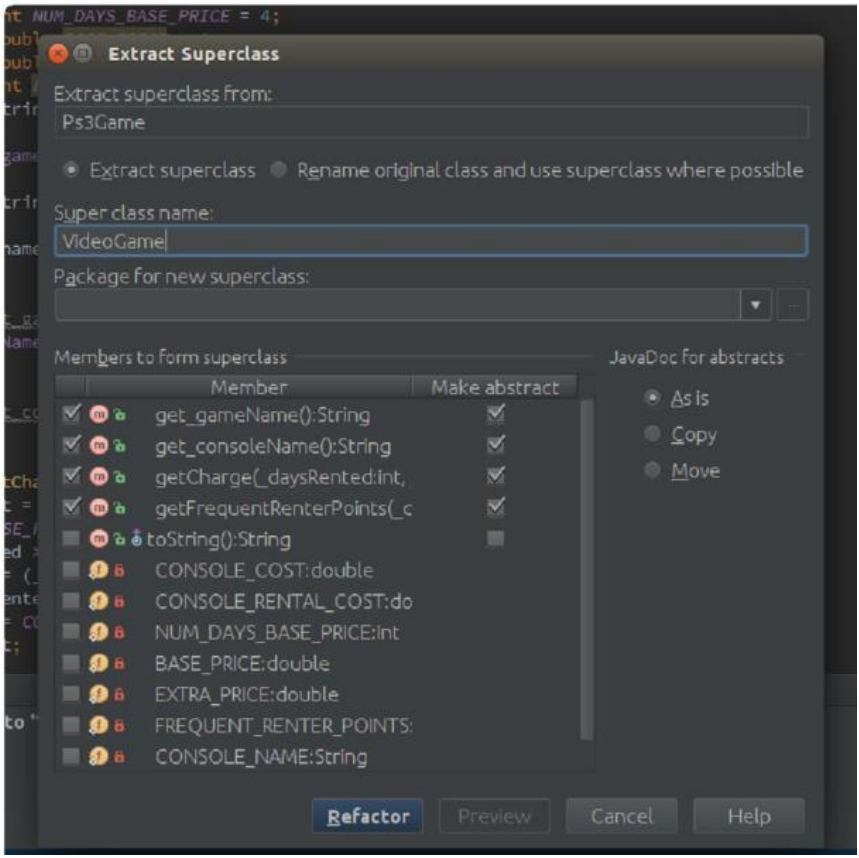

Рисунок 2.21 – Ілюстрація до рефакторингу Extract Superclass

Тепер нам потрібно змінити реалізацію класу VideoGameRental. Зараз він використовує поле типу Object (дуже погана практика). Замініть клас:

```
public class VideoGameRental {
    private VideoGame _videoGame;
    private int _daysRented;
    private boolean _consoleRented;
    public VideoGameRental( VideoGame videoGame,
            int daysRented, boolean consoleRented) {
        _videoGame = videoGame;
        _daysRented = daysRented;
        _consoleRented = consoleRented;
    }

    public int getDaysRented(){
       return _daysRented;
    }

    public VideoGame getVideoGame() {
       return _videoGame;
    }

    public double getCharge() {
       return _videoGame. getCharge( _daysRented, _consoleRented);
    }

    public int getFrequentRenterPoints() {
        return _videoGame.getFrequentRenterPoints( _daysRented, _consoleRented);
    }
}
```

2.6 Формування звіту

Виконайте свої тести та переконайтеся, що вони проходять! Щоб одержати перевірку, будь-ласка, покажіть наступне:

– усі ваші випробування;

–код для тестів, які ви написали;

– ваш суперклас VideoGrame.

СПИСОК ЛІТЕРАТУРИ

1. URL: https://refactoring.guru/uk.
2. URL: https://sourcemaking.com.
3. URL: https://www.jetbrains.com/help/idea/2016.2/refactoring-source-code.html.

Навчальне видання

Методичні вказівки
до практичних занять із дисципліни
«Реінжиніринг і верифікація програмного забезпечення»
на тему «Пошук недоліків коду й використання
інструментів рефакторингу»
для студентів спеціальності 122 «Комп'ютерні науки»
всіх форм навчання

Відповідальний за випуск А. С. Довбиш
Редактори : Н. З. Клочко, Н. М. Мажуга
Комп'ютерне верстання В. В. Москаленко

Формат 60x84/16. Ум.-друк. арк. 2,32. Обл.-вид. арк. 2,29.

Видавець і виготовлювач
Сумський державний університет,
вул. Римського-Корсакова, 2, м. Суми, 40007
Свідоцтво суб'єкта видавничої справи ДК № 3062 від 17.12.2007.