Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Деменкова Сергія Олександровича*
	(ПІБ)
академічної групи	*121М-223-1*
	(шифр)
спеціальності	*121 Інженерія програмного забезпечення*
	(код і назва спеціальності)
освітньої програми	*«121 Інженерія програмного забезпечення»*
	(назва освітньої програми)
на тему:	*Розробка та дослідження ефективності реалізації автоматичної генерації коду для ORM-системи ActiveJDBC з використанням технології Annotation Processing*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	*проф. Мещеряков Л.І.*			

Рецензент				

Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_____ М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » _____ 20 23 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *121 Інженерія програмного забезпечення*

студента *121М-223-1* (група) *Деменкова Сергія Олександровича* (прізвище та ініціали)

Тема кваліфікаційної роботи *Розробка та дослідження ефективності реалізації автоматичної генерації коду для ORM-системи ActiveJDBC з використанням технологій Annotation Processing*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1228 -с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень: процес розробки обробника анотацій для ORM системи ActiveJDBC.

Предмет досліджень: методи генерації коду на базі Annotation Processing у мові програмування Java

Мета роботи: розробка та оптимізація процесора анотацій для автоматичної генерації обгорток для ActiveJDBC об'єктів, які надаватимуть можливість працювати з ActiveJDBC як з POJO об'єктами, створювати нові об'єкти ActiveJDBC за допомогою породжувального патерна Будівельник, та порівнювати обгорнуті об'єкти ActiveJDBC за допомогою автоматично згенерованих методів equals та hashCode

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна отриманих результатів полягає у тому, що вперше було розроблено програмне забезпечення, яке полегшує взаємодію з ActiveJDBC компонентами, та дозволяє додавати функціонал, який не був закладений у ActiveJDBC його розробником.

Практична цінність полягає у створенні інструменту, який дозволяє не писати шаблонний код, зберігає час програміста та підвищує читабельність та підтримуваність кінцевого продукту.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати дослідження мають бути подані у вигляді, що дозволяє безпосередньо побачити та оцінити використання створеної бібліотеки для генерації коду.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок –кінець)**
Аналіз предметної галузі та постановка завдання	05.09.2023-24.09.2023
Дослідження методів генерації коду	25.09.2023-11.10.2023
Проектування та розробка програмного продукту	12.10.2023-07.12.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки розробці програмного забезпечення для автоматизації генерації коду для ORM-системи ActiveJDBC, що дозволить вивільнити ресурси програміста від написання та підтримки шаблонного коду.

Завдання видав _____ *Мещеряков Л.І.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Деменков С.О.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *05.09.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *09.12.2023*

РЕФЕРАТ

Пояснювальна записка: 123 стор., 52 рис., 4 додатка, 35 джерел.

Об'єкт дослідження: процес розробки обробника анотацій для ORM системи ActiveJDBC.

Предмет дослідження: методи генерації коду на базі Annotation Processing у мові програмування Java.

Мета роботи: розробка та оптимізація процесора анотацій для автоматичної генерації обгорток для ActiveJDBC об'єктів, які надаватимуть можливість працювати з ActiveJDBC як з POJO об'єктами, створювати нові об'єкти ActiveJDBC за допомогою породжувального патерна Будівельник, та порівнювати обгорнуті об'єкти ActiveJDBC за допомогою автоматично згенерованих методів equals та hashCode.

Методи дослідження: для вирішення поставлених задач були використані наступні методи: порівняльний аналіз існуючих методів генерації коду, теоретичні основи генерації програмного коду, об’єктно орієнтоване програмування.

Наукова новизна отриманих результатів полягає у тому, що вперше було розроблено програмне забезпечення, яке полегшує взаємодію з ActiveJDBC компонентами, та дозволяє додавати функціонал, який не був закладений у ActiveJDBC його розробником.

Практична цінність полягає у створенні інструменту, який дозволяє згенерувати шаблонний програмний код, що підвищує читабельність та підтримуваність кінцевого продукту.

Список ключових слів: ActiveJDBC, ORM, генерація коду, Java, Dynamic Proxy, Annotation Processor.

ABSTRACT

Explanatory note: 123 pages, 52 figures, 4 appendices, 35 references.

Object of research: the process of developing an annotation processor for the ORM system ActiveJDBC.

Subject of research: code generation methods based on Annotation Processing in the Java programming language.

Purpose of Master's thesis: development and optimization of an annotation processor for the automatic generation of wrappers for ActiveJDBC objects, enabling the usage of ActiveJDBC as POJO objects, creating new ActiveJDBC objects using the Builder design pattern, and comparing wrapped ActiveJDBC objects through automatically generated equals and hashCode methods.

Research Methods: to address the set objectives, the following methods were employed: comparative analysis of existing code generation methods, theoretical foundations of code generation, and object-oriented programming.

The scientific novelty of the obtained results lies in the development of software that facilitates interaction with ActiveJDBC components, allowing the addition of functionality that was not originally embedded in ActiveJDBC by its developer.

Practical value is derived from the creation of a tool capable of generating boilerplate code, enhancing the readability and maintainability of the end product.

Keywords: ActiveJDBC, ORM, code generation, Java, Annotation Processor.

СПИСОК УТМОНИХ ПОЗНАЧЕНЬ

AST – Abstract Source Tree
SQL – Structured Query Language
NoSQL – Not only SQL
JPA – Java Persistence API
ORM – Object-Relational Mapping
JDBC – Java Database Connectivity
DTO - Data Transfer Object
POJO – Plain Old Java Object
JRE - Java Runtime Environment
DI – Dependency Injection
API – Application Programming Interface

ЗМІСТ

ВСТУП

Успіх і зростання будь-якого IT проекту залежить від багатьох факторів, які можна розглядати на різних рівнях: від стратегічного до оперативного. На стратегічному рівні успіх проекту залежить від того, наскільки він узгоджується з цілями та потребами організації, яка його замовляє або реалізує. На оперативному – від того, яким чином виконуються його основні процеси: планування, організація, контроль та закриття, а також наскільки правильно були обрані інструменти, і наскільки зручно ними користуватися для досягнення оперативного та стратегічного успіху проекту.

Часто буває так, що завдяки не дуже вдалому рішенню при виборі інструмента, розробка програмного забезпечення потребує написання великої кількості шаблонного коду. Це призводить до того, що програміст витрачає велику частину робочого часу на обслуговування самого інструмента, який використовується для вирішення поставленої задачі, а не на саме вирішення задачі, що в свою чергу призводить до таких наслідків як:

1. Витрачаються кошти компанії не на вирішення поставлених задач, а на написання шаблонного, часто повторюваного коду.
2. Підвищується вірогідність появ помилок, багів та вразливостей, бо шаблонний код може містити зайві, застарілі або некоректні частини, які можуть спричинити конфлікти, несумісності або небезпечну поведінку.
3. Збільшується обсяг та складність кодової бази, оскільки шаблонний код може займати багато місця та ресурсів, що може призвести до погіршення продуктивності, швидкості та надійності програмного забезпечення, а також значно ускладнити читання такого коду.

Саме тому, щоб запобігти появі шаблонного коду, часто використовуються інструменти, мета яких ще на етапі компіляції проекту, або вже під час виконання коду програмного забезпечення, згенерувати код, який би в іншому випадку програміст був би вимушений написати власноруч.

Одним з прикладів інструменту, використання якого може призвести до появи шаблонного коду є доволі розповсюджена за рахунок своєї легковажності ORM-система ActiveJDBC, для якої і буде створено відповідне рішення.

Данна магістерська робота поділена на три розділи.

У першому розділі проаналізовано предметну галузь, проведено ознайомлення з існуючими інструментами що використовуються для генерації коду взагалі, та для запобігання появі шаблонного коду зокрема. Розглянуто ORM-систему ActiveJDBC, її недоліки та переваги, виявлено її потенційно слабкі місця, та сформульовано вимоги щодо функціональних характеристик для розроблюваного рішення. У другому розділі розглянуто існуючі підходи для автоматичної генерації коду та обґрунтовано вибір одного з них. У третій частині детально описано розроблюваний продукт: використану архітектуру, технічні рішення та, відповідно, код програми.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Загальні відомості з предметної галузі

1.1.1. Автоматична генерація коду у сьогоденні

Автоматична генерація коду є дуже розповсюдженим явищем у нашому сьогоденні. Стрімкий розвиток галузі IT та жорсткі вимоги до строків, бюджетів та якості коду призводять до того, що з’явилось безліч інструментів, що створені для того, щоб перекласти рутинну роботу програміста на комп’ютер. Наразі існують інструменти, які можуть навіть створити програму майже зовсім без написання коду, у якості прикладу таких інструментів можна навести Acceleo від Eclipse Foundation, що окрім іншого, дозволяє створювати класи на основі UML [1], або appmysite, що дозволяє за лічені хвилини створити IOS та Android застосунок на базі вже існуючої веб сторінки [2].

Згідно дослідження проведеного М. Ольтрогге [3] видно, що з більше ніж двох мільйонів безкоштовних застосунків доступних на Google Play, 11.1% було створено за допомогою онлайн сервісів. Все ширше використання штучного інтелекту, дозволяє створювати інструменти, які можуть генерувати код оснований на текстовому опису того, що він повинен робити [4].

1.1.2. Існуючі рішення для запобігання появи шаблонного коду

Але все ж таки на теренах інтернету є багато різних рішень, які використовуються для вирішення цієї проблеми. Коротко пройдемося по деяких з них, що використовуються у зв’язку з мовою програмування Java.

Project Lombok

Рис. 1.1. Логотип Project Lombok

Історія проекту Lombok почалася в 2009 році, коли два нідерландські розробники Роель Спілкер та Рейнер Цвітсерлоот [5] вирішили створити інструмент, який би полегшив їм роботу з Java, саме тому проект Lombok було створено з метою зробити так, щоб зменшити необхідність в ручному написанні рутинного та шаблонного коду, такого як геттери, сеттери, конструктори, та методи toString, equals та hashCode. [6]

Традиційно в Java для розробки класів доводилось написати багато додаткового коду, який не вносив жодного нового функціоналу в клас, що призводило до зайвої роботи та збільшення обсягу коду. Розробники Lombok вирішили цю проблему, надаючи анотації та інструменти, що автоматизують створення цього коду без втручання програміста.

Сам по собі, Lombok представляє собою обробник анотацій, але працює він дещо інакше. Якщо задача звичайного обробника анотацій полягає у тому, щоб зробити якісь перевірки або згенерувати новий клас, то Lombok може вносити зміни у AST [7], тим самим він може додавати нові методи у вже існуючий клас, або навіть створювати нові внутрішні об’єкти в середині самого класу.

Таким чином, Lombok допомагає зробити код більш чистим, компактним і читабельним, сприяє розробці та підтримці програмного забезпечення та

дозволяє розробникам приділяти більше уваги безпосередньо бізнес-логікі свого проекту, а не шаблонному коду.

Spring Data

Рис. 1.2. Логотип фреймворку Spring

Spring Data – це частина проекту Spring Framework, яка надає багато різних інструментів для спрощення взаємодії між додатками на побудованими за використанням мови програмування Java і різними системами зберігання даних, такими як SQL бази даних, NoSQL-сховища та інші джерела даних [8]. Spring Data розширює можливості Spring Framework і допомагає програмістам працювати з даними ефективніше та більш абстраговано.

Цей проект включає різні підпроекти, кожен із яких призначений для взаємодії з певним видом джерел даних. Найпопулярніші підпроекти Spring Data включають:

- Spring Data JPA: Цей підпроект надає абстракції для роботи з JPA та спрощує взаємодію з реляційними базами даних через ORM;
- Spring Data MongoDB: Даний підпроект спрощує взаємодію з базою даних MongoDB, нереляційною системою зберігання документів;
- Spring Data Redis: Цей підпроект дозволяє взаємодіяти з системою зберігання даних Redis, яка використовується для кешування та як сховище даних;

- Spring Data JDBC: Він надає спосіб взаємодії з базами даних за допомогою JDBC, пропонуючи абстракції та спрощені підходи до виконання запитів до баз даних і збереження даних;

Тобто, Spring Data робить роботу з базами даних та іншими джерелами даних більш зручною та продуктивною, надаючи анотації та API для роботи з даними. Він також пропонує можливості пошуку, сортування, пагінації і автоматизованої генерації запитів, що допомагає розробникам швидше створювати додатки, що взаємодіють з різними джерелами даних.

На відміну від проекту Lombok, що є обробником анотацій, Spring Data працює з анотаціями не на етапі компіляції, а безпосередньо під час виконання програмного коду.

Зазвичай, коли потрібно додати у якийсь метод нову функціональність, і цей метод прийшов з якогось інтерфейсу, то у цьому випадку Spring буде використовувати Dynamic Proxies для імплементації цього інтерфейсу під час підняття контексту і в реальності, код програми буде оперувати замісником оригінального об’єкта, методи якого вже будуть обгорнуті декількома додатковими шарами логіки, що будуть контролювати доступ до цього об’єкту. Якщо ж інтерфейсу не існує, і метод який потрібно викликати у заміснику є власним методом класу, то у цьому випадку Spring буде використовувати бібліотеку CGLIB [9], що посеред іншого, дозволяє створювати нащадків класу, для якого необхідно створити замісника.

MapStruct

Рис. 1.3. Логотип фреймворку MapStruct

MapStruct – це ще один обробник анотацій, що дозволяє автоматизувати процес перетворення даних між об'єктами різних класів або типів без

необхідності писати цей код власноруч. MapStruct спрощує розробку та підтримку коду, який відповідає за перетворення даних, і допомагає потенційно зменшити кількість помилок, пов'язаних з неправильним маппінгом. [10]

Основні особливості MapStruct включають:

- MapStruct автоматично генерує код маппінгу на основі інтерфейсів та анотацій, що надаються розробником;
- MapStruct підтримує різні типи маппінгу, що дозволяє мапити дані між об'єктами різних класів, інтерфейсами та типами даних;
- MapStruct підтримує аннотації, які дозволяють налаштовувати генерацію коду маппінгу, включаючи анотації для кастомних мапперів, ігнорування деяких полів тощо;
- MapStruct можна доволі легко інтегрувати з іншими технологіями та фреймворками, такими як Spring, Hibernate, і багатьма іншими;
- Існують різні розширення для MapStruct, які надають додатковий функціонал та можливості;

Record у Java

Рис. 1.4. Логотип Java

Розробники Java також не стоять осторонь і розуміють, що проблема шаблонного коду частково створена самим синтаксисом цієї мови програмування, тому з чотирнадцятої версії Java у нас з’являється новий тип даних record. [11] Record – це спеціальний тип класу, призначений для представлення даних або моделювання об'єктів, які мають фіксований набір

полів, значення яких не може змінюватися після створення об'єкта. В записах автоматично створюються конструктори, методи доступу до полів (геттери), методи порівняння equals() та hashCode(), і метод toString(), що у свою чергу спрощує створення і роботу з об'єктами даних та покращує читабельність коду.

Record зазвичай використовується для представлення даних в Java, таких як DTO, моделі бази даних, відповіді API, тощо. [12] Він допомагає скоротити кількість шаблонного коду та полегшує роботу з незмінними об'єктами даних.

1.2. ORM-система ActiveJDBC

У цій магістерській роботі створено рішення для генерації коду для ORM-системи ActiveJDBC, тому буде доцільно з нею ознайомитись, щоб зрозуміти її недоліки та переваги.

ActiveJDBC – це Java-фреймворк, який надає ORM для роботи з реляційними базами даних. Він реалізує патерн Активний Запис (Active Record) і дозволяє розробникам працювати з базою даних, використовуючи об'єкти Java, без необхідності писати SQL-запити власноруч.

Основні особливості і характеристики ActiveJDBC включають:

- Замість традиційного підходу до ORM, де щоб виконати маніпуляцію з таблицями бази даних потрібно використати додаткові інструменти, ActiveJDBC використовує підхід "активного запису", завдяки якому, кожен клас моделі включає в себе логіку для взаємодії з базою даних;
- ActiveJDBC використовує анотації для вказівки маппінгу класу на таблицю бази даних, вказівки схеми бази даних, у якій зберігається ця таблиця, та поле первинного ключа;
- фреймворк дозволяє визначати відносини між моделями, такі як багато-до-багатьох, багато-до-одного, один-до-багатьох;
- ActiveJDBC дозволяє виконувати складні запити SQL, якщо це необхідно, і в той же час надає простий і зручний спосіб виконання базових операцій, таких як вставка, оновлення і видалення записів;

- Цей фреймворк підтримує міграції бази даних, що окрім іншого дозволяє керувати структурою бази даних та версіями схеми;

ActiveJDBC розвивається як відкрите програмне забезпечення і є популярним варіантом для розробки додатків на Java, які взаємодіють з реляційними базами даних. Він може бути особливо корисним для проектів, де розробники віддають перевагу підходу "активного запису" і потребують зручного і простого використання ORM для спільної роботи з базою даних.

Основними принципами, що сповідує ActiveJDBC є [13]:

- він виводить метадані з бази даних;
- конфігурація на основі домовленостей;
- немає потреби вивчати іншу мову запитів. SQL буде досить;
- код часто читається як англійський текст;
- немає сесій, немає прикріплення до сесій, немає повторного приєднання до сесій;
- немає менеджерів збереження;
- моделі є легкими, простими POJO;
- немає Proxy. Те, що ви пишете, те і отримуєте;
- немає геттерів і сеттерів. Ви можете їх писати, якщо бажаєте;
- немає DAO і DTO;
- немає анемічної моделі домену;

Як можна побачити з вищенаведеного списку принципів, у ActiveJDBC відсутні звичні геттери та сеттери, тому якщо потрібно отримати якісь данні з об’єкту, то це потребує виклику get та set методів, у які треба передавати назву атрибута з яким потрібно взаємодіяти. Приклад коду наведено на рисунку 1.5.

```
// шукаємо у базі даних об'єкт з індексом 12
FancyTable fancyTable = FancyTable.findById(12);
// отримаємо зміст атрибуту FIRST_NAME як строку
String firstName = fancyTable.getString( attributeName: "FIRST_NAME");
// отримаємо зміст атрибуту BIRTH_DATE як java.sql.Date
Date birthDate = fancyTable.getDate( attributeName: "BIRTH_DATE");
// отримаємо зміст атрибуту ANOTHER_DATE як java.sql.Timestamp
Timestamp anotherDate = fancyTable.getTimestamp( attributeName: "ANOTHER_DATE");
// присвоюємо булевому атрибуту LOCKED значення true
fancyTable.setBoolean( attributeName: "LOCKED",  value: true);
// зберігаємо об'єкт у базу даних
fancyTable.save();
```

Рис. 1.5. Приклад коду взаємодії з ActiveJDBC об’єктом

Як видно з наведеного приклада, звертання до атрибутів таблиці не є дуже зручним, тому що програміст вимушений кожного разу власноруч вказувати назву цього атрибуту при кожній маніпуляції з ActiveJDBC об’єктом, що призводить до того, що програміст повинен пам’ятати, які саме поля є у тій чи іншій таблиці, та легко може припустити помилку по неуважності.

У той же час, ActiveJDBC не забороняє створювати власні гетери та сетери, тому нижче буде наведено приклад такого коду.

```
@Table("FANCY_TABLE")
@IdName("FANCY_TABLE_ID")
public class FancyTable extends Model {
    2 usages
    private static final String BIRTH_DATE = "BIRTH_DATE";
    no usages
    public Date getBirthDate() { return this.getDate(BIRTH_DATE); }
    no usages
    public void setBirthDate(Date date) {this.setDate(BIRTH_DATE, date);}
}
```

Рис. 1.6. Приклад створення гетерів та сетерів власноруч

Як видно з рисунку 1.6, клас став доволі великим і не дуже зручним у підтримці, незважаючи на те, що геттери та сеттери були створені не для всіх полів, для заощадження місця. Все ще можливо доволі легко припуститися помилки при створення геттера або сеттера, помилково використавши хибне ім’я поля.

Також з наведеного приклада видно, що при роботі з ActiveJDBC потрібно працювати з датами як з java.sql.Date та java.sql.Timestamp, а не як з більш сучасними java.time.LocalDate та java.time.LocalDateTime, бо ActiveJDBC не підтримує ці типи даних, тому програміст вимушений привести їх до типу, який підтримує ActiveJDBC.

В результаті вищенаведених особливостей використання буде отримано такий код.

```
2 usages
public LocalDate getBirthDate() {
    // вимушені власноруч перевіряти значення поля на null
    // та робити перетворювання
    return Optional.ofNullable(this.getDate(BIRTH_DATE)) Optional<Date>
            .map(Date::toLocalDate) Optional<LocalDate>
            .orElse( other: null);
}

2 usages
public LocalDateTime getAnotherDate() {
    // вимушені власноруч перевіряти значення поля на null
    // та робити перетворювання
    return Optional.ofNullable(this.getTimestamp(ANOTHER_DATE)) Optional<Timestamp>
            .map(Timestamp::toLocalDateTime) Optional<LocalDateTime>
            .orElse( other: null);
}

no usages
public void setAnotherDate(LocalDateTime anotherDate) {
    // вимушені власноруч перевіряти значення поля на null
    // та робити перетворювання
    this.setTimestamp(ANOTHER_DATE, Optional.ofNullable(anotherDate) Optional<LocalDateTime>
            .map(Timestamp::valueOf) Optional<Timestamp>
            .orElse( other: null));
}
```

Рис. 1.7. Приклад написання гетерів та сетерів з java.time.LocalDate та java.time.LocalDateTime

Як видно з рисунку 1.7, в результаті коду стало ще більше, і так як будь які сет методи ActiveJDBC об’єкту приймають в якості значення тип java.lang.Object, то треба постійно контролювати відповідність типів даних, бо середовище розробки, на превеликий жаль, не підкаже, що програміст припустився помилки, і є можливість дізнатися про неї тільки при виконанні коду, і добре, якщо ця ситуація трапиться у тестовому середовищі, а не у продакшені.

Наступною особливістю є те, що базовий абстрактний клас для ActiveJDBC об’єктів Model не імплементує методи equals() та hashCode(), що унеможливлює порівняння двох ActiveJDBC об’єктів між собою, що у свою чергу призводить то того, що такі об’єкти неможливо використовувати у якості ключів для хеш колекцій. Щоб отримати доступ до цих методів, треба реалізувати відповідний код власноруч. Приклад отриманого коду наведено на рисунку 1.8.

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    FancyTable that = (FancyTable) o;
    return Objects.equals(this.getFirstName(), that.getFirstName()) &&
            Objects.equals(this.getBirthDate(), that.getBirthDate()) &&
            Objects.equals(this.getAnotherDate(), that.getAnotherDate());
}

@Override
public int hashCode() {
    return Objects.hash(
            this.getFirstName(),
            this.getBirthDate(),
            this.getAnotherDate());
}
```

Рис. 1.8. Перевизначені методі equals() та hashCode()

Ще одним потенційним обмеженням є те, що метод toString() також не перевизначено, що іноді може бути не зручно, наприклад, коли є потреба вивести якусь інформацію у лог. Так як автоматично перевизначити цей метод неможливо, то знову ж таки програміст вимушений це зробити власноруч, у результаті чого буде отримано метод наведений на рисунку 1.9.

```
@Override
public String toString() {
    return "{firstName = " + (this.getFirstName() == null
            ? null
            : "'" + this.getFirstName() + "'")
            + ", birthDate = " + (this.getBirthDate() == null
            ? null
            : "'" + this.getBirthDate() + "'")
            + ", anotherDate = " + (this.getAnotherDate() == null
            ? null
            : "'" + this.getAnotherDate() + "'")
            + "}";
}
```

Рис. 1.9. Перевизначений метод toString()

Також потрібно зазначити що ще одним недоліком цього інструмента являється відсутність можливості використовувати породжувальний патерн Будівельник для створення нових об’єктів. На жаль, програмування цієї можливості власноруч потребує написання великої кількості коду.

Отже, для того щоб отримати всі необхідні методи для таблиці всього з трьох полів, потрібно було написати дуже багато коду. А якщо полів буде ще більше, то й код ставатиме все більш складним у написанні та підтримці.

1.3. Призначення розробки та галузь її застосування

Розробка та дослідження ефективності інструмента генерації коду для ORM-системи ActiveJDBC.

Наразі було з’ясовано, що даний фреймворк має деякі потенційні недоліки, серед яких можна зазначити:

- нема вбудованої можливості користуватися звичними гетерами та сетерами;
- нема вбудованої можливості порівняти два об’єкта між собою;
- немає вбудованої підтримки сучасних типів дат;
- немає вбудованої можливості привести об’єкт ActiveJDBC до строки зі збереженням інформації з цього об’єкту;
- немає вбудованої можливості використання породжувального патерна Будівельник;
- існують обмеження щодо використання ActiveJDBC об’єктів у хеш колекціях;

Розроблений продукт потенційно має використовуватися програмістами, що працюють фреймворком ActiveJDBC для:

- зменшення кількості шаблонного коду;
- підвищення швидкості написання коду;
- підвищення читабельності отриманого коду;
- підвищення безпеки роботи з ActiveJDBC об’єктами;
- полегшення підтримки отриманого коду;
- полегшення створення нових ActiveJDBC об’єктів;

1.4. Підстави для розробки

Згідно з навчальним планом та відповідно до навчальної програми та графіка навчального процесу, наприкінці навчання студентами виконується магістерська робота.

Тема проекту є узгодженою з його керівником, кафедрою, а також затверджена наказом ректора.

Отже, підставами для виконання кваліфікаційної роботи є:

- навчальна програма спеціальності 121 «Інженерія програмного забезпечення»;
- навчальний план та графік навчального процесу;
- наказ ректора Національного технічного університету «Дніпровська політехніка» від 09.10.2023 р. № 1228 -с;
- завдання на магістерську роботу: «Розробка та дослідження ефективності реалізації автоматичної генерації коду для ORM-системи ActiveJDBC з використанням технологій Annotation Processing»;

1.5. Постановка завдання

В даній кваліфікаційній роботі розглядається розробка та дослідження ефективності реалізації автоматичної генерації коду для ORM-системи ActiveJDBC, що повинна призвести до зменшення кількості шаблонного коду, що може з’являтися при використанні цього фреймворку.

Мета цієї роботи: розробка та оптимізація інструмента для автоматичної генерації коду для ActiveJDBC об'єктів, які надаватимуть можливість працювати з ActiveJDBC як з POJO об'єктами, створювати нові об'єкти ActiveJDBC за допомогою породжувального патерна Будівельник, та порівнювати об'єкти ActiveJDBC за допомогою автоматично згенерованих методів equals() та hashCode().

Створення й розробка додатку повинна включати наступні етапи:

- формулювання функціональних та нефункціональних вимог до розроблюваного інструмента;
- аналіз існуючих підходів та технологій з генерації коду для виявлення найбільш оптимального для вирішення поставленої задачі;

- проектування інтерфейсу, через який буде проводитись налаштування розроблюваного інструмента та розробка його архітектури;
- розробка фунціоналу, інтерфейсу налаштування, тестування та відлагодження розробленого інструменту.
- Підтримка, та розвиток, що повинно включати в себе надання допомоги користувачам розробленого інструменту, додавання нових можливостей згідно потреб користувачів.

Кожен з цих етапів є дуже важливим для успішного створення та просування розробленого інструменту, бо без сформульованих вимог, дуже важко зрозуміти, що конкретно потрібно розробити, без аналізу існуючих рішень та підходів, які використовуються для отримання бажаної мети, є великий шанс того, що буде обрано хибне рішення що до реалізації, що потенційно призведе до того, що інструмент потрібно буде розробляти заново. Без підтримки та розвитку інструмент може дуже швидко застаріти, що призведе до зменшення кількості нових користувачів. Особливо тому, що у деяких компаніях є негласне правило не використовувати інструменти, які давно не оновлюються, або у яких слабке та не активна спільнота користувачів.

1.6. Вимоги до програми або програмного виробу

1.6.1. Вимоги до функціональних характеристик

Розроблений інструмент повинен відповідати наступним вимогам:

- повинна бути можливість налаштування розробленого інструменту, щоб користувач мав змогу керувати назвами згенерованих методів та класів;
- інструмент повинен працювати зовсім без додаткових налаштувань, завдяки налаштуванням за замовченням;
- інструмент повинен генерувати геттери та сеттери;
- інструмент повинен генерувати методи equals(), hasCode() та toString();

- інструмент повинен генерувати клас Будівельник, для більш зручного створення нових ActiveJDBC об’єктів;
- інструмент повинен використовувати анотації для того, щоб вказати, для яких саме атрибутів таблиць баз даних потрібно згенерувати відповідні методи, та шаблон для назви цих методів. Якщо шаблон не задано, то інструмент повинен замість нього використовувати назву поля таблиці у стилі CamelCase, і на її основі будувати геттери, сеттери тощо;
- інструмент повинен приховувати існуючі методи ActiveJDBC об’єктів для того, щоб унеможливити некоректне використання згенерованого коду, та не плутати користувача великою кількістю зайвих методів;
- інструмент повинен давати змогу отримувати оригінальний ActiveJDBC об’єкт, для доступу до всіх раніше прихованих методів;
- інструмент повинен використовувати якомога менше залежностей на інші бібліотеки;
- інструмент не повинен значно впливати на використання ресурсів системи користувача;
- робота інструменту не повинна залежати від операційної системи користувача;

1.6.2. Вимоги до складу та параметрів технічних засобів

Інструмент повинен працювати на системі програміста для генерації коду, який потім буде використовуватися на відповідному сервері клієнта, або у тестовому середовищі.

Для нормального використання розробленого інструменту треба, щоб проект у якому він буде використовуватись задовольняв таким вимогам як:

- Проект повинен бути побудований з використанням засобу автоматизації роботи з програмними проектами «Apache Maven»;
- Версія Java повинна бути 8 або вище;

РОЗДІЛ 2
ДОСЛІДЖЕННЯ МЕТОДІВ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ КОДУ

1.1. Різновиди підходів автоматичної генерації коду

Так як розроблюваний інструмент повинен мати можливість згенерувати потрібний код, то спочатку, є потреба з’ясувати, яким чином буде досягнуто цієї мети у мові програмування Java, де саме це буде зроблено, та за допомогою яких інструментів.

Наразі існує декілька етапів, впродовж яких Java дозволяє втручатися у програмний код і робити з ним щось додаткове, що не було явно зазначено програмістом.

1.1.1. Генерація під час виконання програмного коду

Генерація коду під час виконання є доволі розповсюдженим явищем і використовується у багатьох різних фреймворках та бібліотеках. У якості прикладу такого фреймворку можна навести Spring Framework. Відповідно до дослідження проведеного Б. Вермером [14], що провів опитування більш ніж двох тисяч респондентів, Spring використовують 60% з них.

Генерація коду під час виконання може бути умовно поділена на:

- генерація коду під час завантаження класу;
- генерація коду під час виклику відповідних методів-генераторів;

Обидва підходи використовують завантажувач класів (ClassLoader) у якості основного інструмента.

ClassLoader в Java - це компонент JRE, який відповідає за завантаження класів під час роботи. Класи в Java завантажуються динамічно в процесі виконання програми і можуть робити це у різних «класлоадерах». Вони визначають звідки і як класи завантажуються в пам'ять JVM, та виконують такі основні завдання [15]:

- завантаження класів: ClassLoader завантажує класи з різних джерел, таких як файлова система, мережа або навіть інші ресурси, і перетворює їх в об'єкти класів у пам'яті;
- пошук класів: ClassLoader визначає, звідки треба завантажувати класи, і шукає їх в заданих джерелах. Це включає в себе пошук класів в шляхах, зазначених в CLASSPATH або в інших зовнішніх ресурсах;
- визначення батьківського ClassLoader: ClassLoader має посилання на батьківський ClassLoader. Якщо клас не знайдено в поточному ClassLoader, він передає запит батьківському ClassLoader для подальшого пошуку, що створює ієрархію ClassLoader-ів.
- Кешування класів: ClassLoader може кешувати завантажені класи для подальшого використання, щоб уникнути дублювання завантаження.

ClassLoader включає в себе кілька вбудованих реалізацій:

- Bootstrap ClassLoader: це перший ClassLoader, який є суперкласом до Extension ClassLoader. Він завантажує файл «rt.jar», який містить всі класові файли Java Standard Edition, такі як класи пакету java.lang, класи пакету java.net, класи пакету java.util, класи пакету java.io та класи пакету java.sql;
- Extension ClassLoader: це дочірній завантажувач класів відносно до Bootstrap і батьківський завантажувач класів відносно до System ClassLoader. Він відповідає за завантаження файлів «jar», розташованих в JDK Extension library;
- System/Application ClassLoader: цей завантажувач класів визнається дочірнім до Extension ClassLoader, та відповідає за завантаження файлів-класів з Classpath, у тому числі і класи самого застосунку;

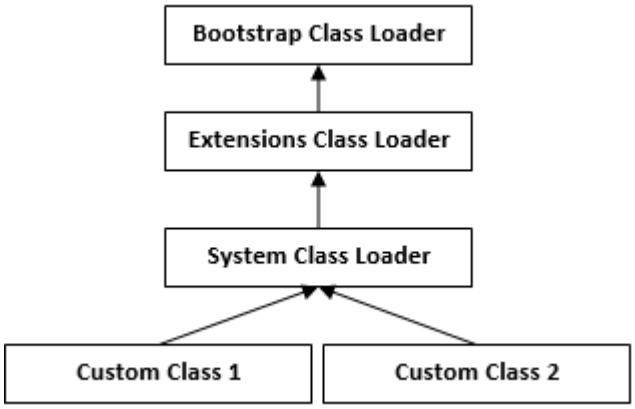

Рис 2.1. Ієрархія завантажувачів класів

Крім цих вбудованих завантажувачів класів, є можливість створити власні реалізації ClassLoader-ів для специфічних потреб. ClassLoader є важливою частиною механізму Java, який дозволяє динамічно завантажувати і використовувати класи в Java додатках, а також обмежувати «зону видимості» завантажених класів, що дозволяє завантажувати більш ніж одну версію одного і того ж самого класа, та використовувати їх незалежно, що дозволяє запобігти неочікуваній поведінці.

Якщо є необхідність взаємодіяти з кодом під час завантаження класу ClassLoader-ом, то необхідно втрутитися у цей процес, для чого можна використати Java Instrumentation API [16], що за допомогою створення та використання java agent дозволяє виконати якісь дії до того, як main метод буде виконано, так як на цьому етапі класи застосунку ще не завантажені, і ще можливо втрутитися у цей процес завдяки додаванню власної імплементації ClassFileTransformer-а, що буде вносити відповідні зміни у клас. Приклад Java агенту наведено на рисунку 2.1.

```
public class Premain {

    erioh
    public static void premain(String agentArgs, Instrumentation instrumentation) {
        instrumentation.addTransformer(new PongClassFileTransformer());
    }

    1 usage   erioh *
    public static class PongClassFileTransformer implements ClassFileTransformer {
        2 usages
        private static final String CLASS_NAME = "common/FinalUnderTestImpl";
        erioh *
        @Override
        public byte[] transform(ClassLoader loader,
                                String className,
                                Class<?> classBeingRedefined,
                                ProtectionDomain protectionDomain,
                                byte[] classfileBuffer) {
            try {
                if (className != null && className.equals(CLASS_NAME)) {
                    ClassPool classPool = ClassPool.getDefault();
                    CtClass ctClass = classPool.get(CLASS_NAME.replace( target: "/",  replacement: "."));
                    CtMethod ping = ctClass.getDeclaredMethod( name: "ping");
                    ping.setBody("{return \"pong\";}");
                    return ctClass.toBytecode();
                }
            } catch (Throwable throwable) {
                return classfileBuffer;
            }
            return classfileBuffer;
        }
    }
}
```

Рис. 2.1. Приклад модифікації класа на етапі завантаження

Додавання цього агенту у командну строку запуску застосунку через параметр «-javaagent» призводить до того, що метод ping() класу FinalUnderTestImpl буде перезаписаний при завантажені самого класу. При чому є потреба зазначити, що буде змінено саме FinalUnderTestImpl, без створення класів нащадків.

Генерація коду під час виклику «методів-генераторів» відбувається за рахунок або створення нового класу завдяки імплементації відповідного інтерфейсу, для чого використовується підхід динамічних проксі, або створення нового класу за використання розширення існуючого класа завдяки технологіям

CGLIB, JavaAssist або ByteBuddy, що є найпоширеніші інструменти для взаємодії з байткодом в індустрії.

Коротко пройдемося по цим інструментам.

Динамічні проксі, це механізм в мові програмування Java, який надає можливість створювати об'єкти-посередники для інших об'єктів, що будуть використані для перехоплення та обробки викликів методів до оригінальних об'єктів, або повністю імплементувани абстрактні методи бажаних інтерфейсів.

Основні компоненти динамічних проксі включають [17]:

- InvocationHandler: Це інтерфейс, який містить метод invoke(Object proxy, Method method, Object[] args). Реалізація цього інтерфейсу визначає, яку логіку слід виконувати перед, після, або замість виклику методу оригінального об'єкта. Обробник виклику приймає три аргументи: сам проксі-об'єкт, викликаний метод і аргументи методу;
- Proxy-клас: Динамічний проксі створюється за допомогою виклика методу Proxy.newProxyInstance(). Цей метод приймає три аргументи: ClassLoader, масив інтерфейсів і об'єкт InvocationHandler. Він створює новий об'єкт, який реалізує всі інтерфейси, передані в аргументах, і перенаправляє виклики методів до обробника виклику.

У якості прикладу використання динамічних проксі можна навести імплементацію метода ping() інтерфейса InterfaceUnderTest зображеного на рисунку 2.3.

```
String result = "Hello There Obi-Wan Kenobi.";
InterfaceUnderTest helloWorld = (InterfaceUnderTest) Proxy.newProxyInstance(
        this.getClass().getClassLoader(), new Class<?>[]{InterfaceUnderTest.class},
        (proxy, method, args) -> {
            if (method.getName().equals("ping")) {
                return result;
            }
            throw new OperationNotSupportedException();
        });
```

Рис. 2.3. Реалізація метода ping() інтерфейса InterfaceUnderTest за допомогою динамічних проксі

На жаль, технологія динамічних проксі вимагає від програміста користуватися інтерфейсами, і якщо їх не існує, то використати цей підхід не є можливим.

У цьому випадку на допомогу приходять більш потужні інструменти, як, наприклад CGLIB, що, посеред іншого, дозволяє створювати нові об’єкти-нащадки від існуючого класа за допомогою генерації байт-коду при виконанні програмного коду програми, у якій він використовується. CGLIB надає дуже багато різних інструментів [18], але у розрізі цієї роботи є необхідність зосередити увагу тільки на деякі з них:

- Enhancer – цей клас дозволяє створювати проксі як від інтерфейсів, так і від неінтерфейсних типів;
- FixedValue – інтерфейс, імплементація якого використовується у якості зворотного виклику будь якого методу об’єкта;
- InvocationHandler – інтерфейс, імплементація якого також буде використовуватися у якості зворотного виклику, але він вже має доступ до огорнутого у проксі об’єкту, методу, що викликається та до аргументів, з якими цей метод викликався;
- MethodInterceptor – ще більш потужний інтерфейс, де посеред іншого, можливо отримати доступ до самого об’єкта проксі та до огорнутого у проксі методу;

У якості прикладу використання можна навести операцію перевизначення методу ping() об’єкта типу UnderTestImpl, щоб він повертав значення PONG, що наведено на рисунку 2.4.

```
UnderTestImpl realObject = new UnderTestImpl();
UnderTestImpl proxyObject = (UnderTestImpl) Enhancer.create(
        UnderTestImpl.class,
        (MethodInterceptor) (object, method, args, methodProxy) -> {
            if (method.getDeclaringClass() != Object.class
                    && method.getName().equals("ping")) {
                return "PONG";
            }
            return methodProxy.invokeSuper(object, args);
        });
```

Рис. 2.4. Перевизначення методу ping() об’єкта типу UnderTestImpl

Хоча CGLIB є потужним інструментом, важливо користуватися ним обережно і враховувати його вплив на продуктивність, оскільки генерація байт-коду може бути витратною операцією, а також він може не працювати з сучасними версіями Java. [19]

1.1.2. Генерація під час компіляції

Генерація коду під час компіляції може виконуватись завдяки Maven плагінам, але зазвичай використовується підхід обробки анотацій, як найбільш універсального та не залежного від засобу автоматизації роботи з програмними проектами «Apache Maven».

Обробник анотацій (Annotation Processor) – це інструмент в середовищі розробки Java, який дозволяє розробникам створювати власні анотації та визначати логіку, пов'язану з ними. Він грає важливу роль у сучасній Java розробці, дозволяючи автоматизувати багато рутинних завдань і забезпечувати додатковий функціонал для написаного коду.

Основні завдання процесора анотацій включають:

- Аналіз анотацій. Процесор анотацій аналізує вихідний код написаної програми, виявляючи та обробляючи анотації, які використані у коді, що дозволяє прикріплювати анотації до класів, методів, полів, параметрів та інших елементів коду. Після виявлення анотацій, обробник анотацій може

виконувати певну логіку, пов'язану з ними. Ця логіка може включати в себе перевірку умов, валідацію коду, створення метаданих, автоматичне створення та публікування документації та багато іншого.

- Генерація коду. Одним з найпоширеніших використань обробників анотацій є генерація коду. При використанні цього підходу потрібно визначити власні анотації та використовувати процесори анотацій для автоматичної генерації коду на їх основі. Це дозволяє спростити багато завдань, такі як рутинна бізнес-логіка, серіалізація та десеріалізація даних, створення CRUD-операцій для баз даних і багато інших задач.

Існує цикл, якому слідкує компілятор javac, що дозволяє кожному обробнику анотацій генерувати код для відповідних анотацій, які відкриті саме для нього. Ось порядок цього процесу: [20]

1. Позначаємо елемент анотацією, яка відкрита для відповідного обробника анотацій.
2. javac розпочинає компіляцію класів проекту. Він вже обізнаний з усіма наявними процесорами анотацій, оскільки вони вже були включені у classpath.
3. javac створює файли з розширенням «.class» під час першого проходу обробки. Оскільки javac вже обробив файли з розширенням «.java», то він вже знає про анотації, які були використані. Після цього він передає керування відповідному процесору анотацій.
4. На цьому кроці процесор анотацій починає виконувати свою роботу. Тут він може генерувати додаткові файли з розширенням «.java», або робити будь яку іншу роботу.
5. На цей момент всі початкові файли «.java» були скомпільовані, але тільки-но згенеровані «.java» файли ще не пройшли обробку, але javac виявить це і розпочне інший раунд обробки з першого кроку.

Цей цикл буде продовжуватися до тих пір, поки не буде більше файлів «.java», що не були виявлені і опрацьовані компілятором javac.

Важливою рисою цього циклу є те, що він дозволяє обробникам анотацій спільно взаємодіяти та генерувати код для анотацій, які можуть бути використані іншими обробниками анотацій. Це важливо для автоматизації рутинних завдань та полегшення роботи з анотаціями в Java.

Схематичне зображення роботи обробника анотацій зображено на рисунку 2.5.

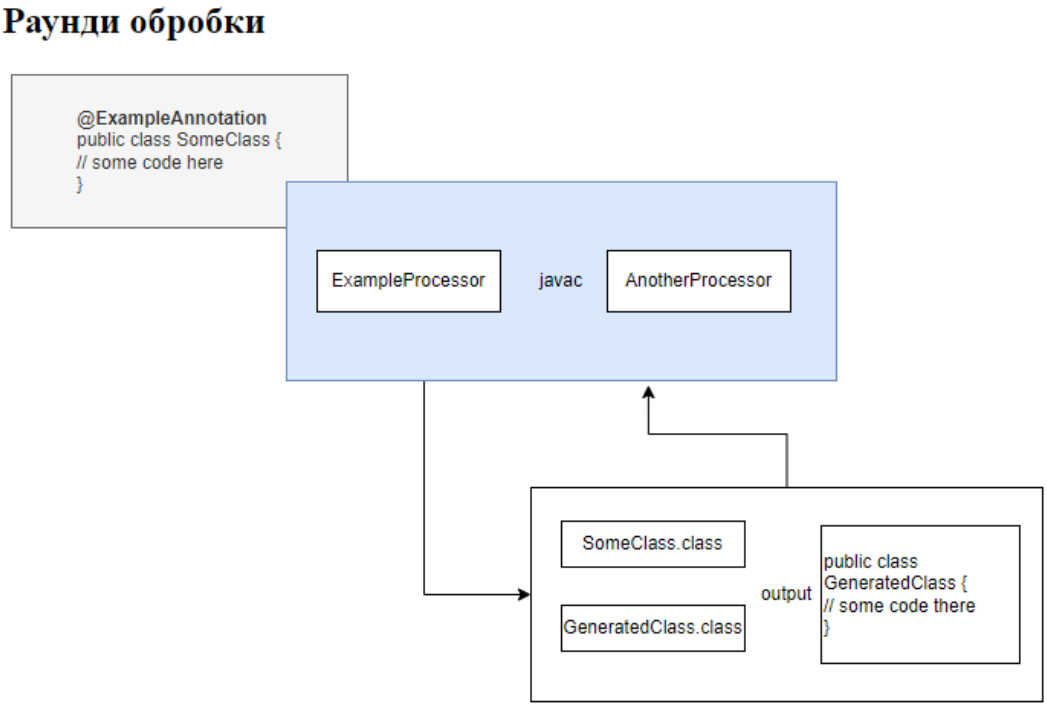

Рис. 2.5. Схематичне зображення роботи обробника анотацій

Почати працювати з опрацюванням анотацій умовно легко. Для того щоб створити власний обробник анотацій достатньо реалізувати інтерфейс Processor, або розширити клас AbstractProcessor з пакету javax.annotation.processing. Приклад коду наведено на рисунку 2.6.

```
public class TestProcessor extends AbstractProcessor {
    @Override
    public boolean process(
            Set<? extends TypeElement> annotations,
            RoundEnvironment roundEnv) {
        for (TypeElement annotation : annotations) {
            Set<? extends Element> elementsAnnotatedWith = roundEnv
                    .getElementsAnnotatedWith(annotation);
            for (Element element : elementsAnnotatedWith) {
                processingEnv.getMessager()
                        .printMessage(Diagnostic.Kind.NOTE, msg: "Анотацію знайдено!");
            }
        }
        return false;
    }
}
```

Рис. 2.6. Приклад обробника анотацій

Обробники анотацій використовуються в багатьох відомих фреймворках та бібліотеках, таких як Lombok, MapStruct і багатьох інших. Вони допомагають зробити код більш ефективним, припускатися меншої кількості помилок та дозволяють створювати частину функціональності ще на етапі компіляції проекту.

1.2. Обґрунтування обраного підходу

Після того, як було проведено ознайомлення з наявними підходами, які використовуються для генерації коду, є доцільним зробити їх порівняльну характеристику та обрати рішення яке задовольняє поставленому завданню, для чого розглянемо порівняльну таблицю характеристик методів генерації коду.

Таблиця 2.1

Таблиця порівняльних характеристик методів генерації коду

Характеристика	Генерація коду від час виконання	Генерація коду під час компіляції
Створює додаткове навантаження на процес компіляції	Ні	Так
Створює додаткове навантаження на систему при виконанні коду	Так	Ні
Дозволяє перевизначити метод equals()	Так	Так
Дозволяє перевизначити метод hashCode()	Так	Так
Дозволяє перевизначити метод toString()	Так	Так
Дозволяє згенерувати нові методи	Так, але цей код неможливо використати програмістом при написані нового коду	Так
Дозволяє згенерувати класи-будівельники	Так, але цей код неможливо використати програмістом при написані нового коду	Так

Ґрунтуючись на даних з вищенаведеної таблиці робимо висновок, що для досягнення нашої мети найоптимальнішим рішенням буде використовувати підхід опрацювання анотацій.

РОЗДІЛ 3
ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

1.3. Опис використаної архітектури та шаблонів проектування

Так як однією з вимог до інструмента що розробляється є максимальна самостійність, то використовувати фреймворк Spring не є можливим, бо хоч він і доволі розповсюджений, але все ж таки, існує багато проектів, які його не використовують. Саме тому для керування залежностями створюваних об’єктів буде використано патерн Внедрення Залежностей.

У програмній інженерії внедрення залежностей (DI) - це техніка програмування, при якій об'єкт або функція отримують інші об'єкти або функції, які їм потрібні, ззовні, замість того, щоб створювати їх всередині свого об’єкту. Внедрення залежностей спрямоване на розділення відповідальності конструювання об'єктів і їх використання, що призводить до зменшення зчеплення між об’єктами [21].

Цей патерн забезпечує те, що об'єкт або функція, які хочуть використовувати певний об'єкт-сервіс, не повинні знати, як його створювати, які у нього є свої залежності, та остаточний тип об’єкта-сервіса. Замість цього об'єкт або функція, які приймають залежності, отримують їх від зовнішнього коду (ін'єктора), про якого вони не повинні нічого знати.

Внедрення залежностей реалізує ідею «інвертування контролю над реалізацією залежностей», тому іноді цей концепт називають «інверсією управління» [22].

Внедрення залежностей може приймати такі форми як [22]:

Внедрення залежностей через конструктор.

При цьому варіанті внедрення залежностей використовується конструктор з параметрами, куди передається вже створений і скорнігурований

об’єкт-залежність. Цей варіант внедрення залежностей є найрозповсюдженішим, бо є вельми простим для розуміння і не потребує додаткових технічних методів.

Перевагою цього підходу є те, що одразу після створення об’єкта всі потрібні залежності будуть явним чином у нього передані.

Потенційним же недоліком цього підходу можна навести те, що замінити об’єкт-залежність після створення об’єкту-носія не вийде.

Приклад імплементації внедрення залежності через конструктор наведено на рисунку 3.1.

```
public class Application {
    public static void main(String[] args) {
        Printer printer = new Printer();
        RunMe runMe = new RunMe(printer);
        runMe.run();
    }
    2 usages
    private static class RunMe implements Runnable {
        2 usages
        private final Printer printer;
        1 usage
        private RunMe(Printer printer) {
            this.printer = printer;
        }
        @Override
        public void run() {
            printer.print();
        }
    }
    4 usages
    private static class Printer {
        1 usage
        public void print() {
            System.out.println("print.");
        }
    }
}
```

Рис. 3.1. Приклад імплементації внедрення залежностей через конструктор

Внедрення залежностей через метод сеттер.

При цьому варіанті внедрення залежностей використовуються специфічні методи-сеттери, викликаючи які можна підкласти потрібну залежність у об’єкт-носій.

Перевагою цього методу є те, що можливо викликати цей сеттер будь яку кількість разів і заміняти залежність на іншу, якщо це потрібно.

Недоліком є те, що потрібно мати цей шаблонний метод-сетер, який є сугубо технічним і він ніяк не пов’язаний з бізнес-логікою створюваного об’єкта, що сильно забруднює код, особливо коли залежностей дуже багато. Також є необхідність зазначити такий недолік, що на відміну від внедрення залежностей через конструктор, об’єкт у який залежності підкладаються сеттером не буде готовий до використання одразу після створення, тому треба бути дуже обачним при імплементації цього підходу.

Приклад використання внедрення залежностей за допомогою методів сетерів наведено на рисунку 3.2.

```
public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
    Printer printer = new Printer();
    RunMe runMe = new RunMe();
    runMe.setPrinter(printer);
    runMe.run();
}
2 usages
private static class RunMe implements Runnable {
    2 usages
    private Printer printer;

    @Override
    public void run() { printer.print(); }
    1 usage
    public void setPrinter(Printer printer) { this.printer = printer; }
}
```

Рис. 3.2. Приклад імплементації внедрення залежностей через сетер

Також є потреба виявити такий спосіб внедрення залежностей як внедрення через reflection. Його можна розглядати як варіант внедрення

залежностей через сеттер, але з тією різницею, що сеттери у цьому випадку не використовуються.

Перевагою цього підходу є те, що у нас зникають сеттери, що до цього забруднювали код. Також у нас залишається можливість підмінити залежність іншою.

Недоліками цього підходу можна зазначити складність імплементації і те, що одразу після створення, наш об’єкт не готовий до використання і потребує додаткового налаштування. Приклад імплементації внедрення залежностей через reflection наведено на рисунку 3.3.

```
public class Application {
    public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
        Printer printer = new Printer();
        RunMe runMe = new RunMe();
        Field declaredField = runMe.getClass().getDeclaredField( name: "printer");
        declaredField.setAccessible(true);
        declaredField.set(runMe, printer);
        declaredField.setAccessible(false);
        runMe.run();
    }
    2 usages
    private static class RunMe implements Runnable {
        2 usages
        private Printer printer;

        @Override
        public void run() {
            printer.print();
        }
    }
    3 usages
    private static class Printer {
        1 usage
        public void print() {
            System.out.println("print.");
        }
    }
}
```

Рис. 3.3. Приклад імплементації внедрення залежностей через reflection

Так як є наявна потреба реалізовувати породжувальний патерн Будівельник, то доцільно ознайомитись з ним також.

Породжувальний патерн Будівельник – це патерн що створено для вирішення проблеми побудови великих та складних об’єктів, які зазвичай або мають дуже великі та складні конструктори, або велику кількість сетерів, ціль яких покласти у створений об’єкт всі необхідні дані.

На жаль, такими об’єктами дуже незручно, а іноді і небезпечно, користуватися, бо якщо існує конструктор з десятьма вхідними параметрами, що мають повторювані типи даних, то вірогідність того, що вхідні параметри будуть випадково передані у хибному порядку є доволі високою, що потенційно може привести до того, що об’єкт буде створено невірно. Приклад хибної передачі параметрів наведено на рисунку 3.4.

```
public class BuilderIsRequired {
    2 usages
    private final String name;
    2 usages
    private final String city;

    1 usage
    public BuilderIsRequired(String name, String city) {
        this.name = name;
        this.city = city;
    }

    1 usage
    public void whoAmI() {
        System.out.printf("My name is %s and I live in %s%n", name, city);
    };

    public static void main(String[] args) {
        new BuilderIsRequired( name: "Dnipro",  city: "Serhii").whoAmI();
    }
}
```

```
BuilderIsRequired
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
My name is Dnipro and I live in Serhii
```

Рис. 3.4. Хибна передача даних при створені об’єкту.

А якщо використовувати сеттери, то є вірогідність того, що неналаштований об’єкт буде використано до того, як всі потрібні сеттери будуть викликані. Приклад використання некоректно налаштованого об’єкта наведено на рисунку 3.5.

```
public class BuilderIsRequired {
    2 usages
    private String name;
    2 usages
    private String city;

    1 usage
    public void setName(String name) {
        this.name = name;
    }

    1 usage
    public void setCity(String city) {
        this.city = city;
    }

    1 usage
    public void whoAmI() {
        System.out.printf("My name is %s and I live in %s%n", name, city);
    };

    public static void main(String[] args) {
        BuilderIsRequired builderIsRequired = new BuilderIsRequired();
        builderIsRequired.setCity("Dnipro");
        builderIsRequired.whoAmI();
        builderIsRequired.setName("Serhii");
    }
}
```

```
BuilderIsRequired
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
My name is null and I live in Dnipro
```

Рис. 3.5. Об’єкт використано до того, як він був повністю налаштований.

Для того щоб вирішити цю проблему патерн Будівельник делегує логіку побудови об’єкта у інший клас [23], і надає можливість взаємодіяти виключно з

класом будівельником до тих пір, поки не буде створено повністю налаштований об’єкт. Приклад класа будівельника можна побачити на рисунку 3.6, а його використання на рисунку 3.7.

```
public static class  BuilderIsRequiredBuilder {
    2 usages
    private String name;
    2 usages
    private String city;
    1 usage
    private BuilderIsRequiredBuilder() {}
    1 usage
    public static BuilderIsRequiredBuilder builder() {
        return new BuilderIsRequiredBuilder();
    }
    1 usage
    public BuilderIsRequiredBuilder name(String name) {
        this.name = name;
        return this;
    }
    1 usage
    public BuilderIsRequiredBuilder city(String city) {
        this.city = city;
        return this;
    }
    1 usage
    public BuilderIsRequired build() {
        return new BuilderIsRequired(name, city);
    }
}
```

Рис. 3.6. Клас будівельник

```
public static void main(String[] args) {
    BuilderIsRequiredBuilder.builder() Bui
            .name("Serhii")
            .city("Dnipro")
            .build() BuilderIsRequired
            .whoAmI();
}
```

Рис. 3.7. Використання класа будівельника

Так як при побудові інструмента для генерації коду буде дуже часто використовувати поведінковий патерн Стратегія, то є доречним розглянути його окремо.

Поведінковий патерн Стратегія визначає сімейство алгоритмів, укладає кожен з них у окремий клас і робить їх взаємозамінними. Таким чином, цей патерн дозволяє об'єктам змінювати свою поведінку відокремленою від клієнтів, які його використовують [23].

Побудова патерна Стратегія у тому чи іншому вигляді потребує використання таких учасників як:

- Контекст – це клас, який має посилання на об'єкт класу-стратегії. Контекст взаємодіє зі стратегією через спільний інтерфейс.
- Стратегія – це інтерфейс або абстрактний клас, який описує специфікацію для всіх конкретних стратегій. Конкретні стратегії реалізують цей інтерфейс або успадковуються від абстрактного класу стратегії.
- Конкретна стратегія – це клас, який реалізує інтерфейс стратегії, і що тримає у собі реалізацію відповідного алгоритму.

Основна ідея патерна Стратегія полягає в тому, що контекст об'єкта може змінювати свою стратегію під час виконання без необхідності змінювати свій власний код, що і дозволяє змінювати поведінку об'єкта під час виконання програми.

1.4. Опис використаних технологій та мов програмування

Розроблюваний інструмент генерації коду було створено за допомогою таких програмних інструментів як:

- мова програмування Java;
- засіб автоматизації роботи з програмними проектами Apache Maven;
- ORM-система ActiveJDBC;

Для автоматизації збірки проекту та для подальшої публікації отриманого інструменту використовувалися такі плагіни для Apache Maven як:

- maven-compiler-plugin
- nexus-staging-maven-plugin
- maven-source-plugin
- maven-javadoc-plugin
- maven-gpg-plugin

Для тестування отриманого продукту використовувались такі інструменти як:

- junit
- assertj-core
- mockito-core
- compile-testing

Java

Java – це високорівнева, об'єктно-орієнтована, мова програмування та платформа розробки, що була створена компанією Sun Microsystems. Вона вперше була випущена в 1995 році та здобула велику популярність завдяки своїй кросплатформенності, безпеці, надійності та потужній підтримці спільноти [24]. З 2009-го року розробкою та підтримкою мови програмування Java займається Oracle Corporation.

Основними характеристиками мови програмування Java є:

- Об'єктно-орієнтований підхід: Java побудована на об'єктах, що сприяє легкості створення, моделювання та управління складними системами.
- Платформонезалежність: Java використовує віртуальну машину Java, яка дозволяє виконувати Java-код на різних операційних системах без змін завдяки тому, що програмний код компілюється не під відповідну архітектуру або операційну систему, а під саму віртуальну машину.
- Надійність: завдяки тому, що у Java використовується строга типизація, що передбачає, що всі змінні мають бути використані лише у відповідних типах

даних і не допускається автоматичного перетворення одного типу в інший без явної вказівки програміста.

- Оптимізація генерації байткоду: компілятор виконує аналіз та оптимізацію коду програми для досягнення більшої продуктивності.
- Автоматична збірка сміття: мова програмування Java надає інструменти автоматичного видалення об’єктів з пам’яті, які більше не мають посилань з інших об’єктів, а також класи, які дозволяють вказати збірнику сміття, що об’єкт можна видалити, якщо на нього існують тільки «слабкі» посилання.
- Можливість багатопотокового програмування: Java підтримує створення та управління багатьма потоками в одному програмному коді, що корисно для розробки програм, які вимагають виконання якихось дій паралельно та незалежно.
- Велика спільнота програмістів: Java має активну та велику спільноту програмістів, яка створила багато бібліотек, фреймворків та інструментів для розробки, що значно пришвидшує власну розробку, бо для багатьох операцій вже є стабільні рішення.

Apache Maven

Apache Maven – це інструмент для управління збіркою та залежностями в проектах розробки програмного забезпечення. Наразі, завдяки плагінам, він може використовуватися у тандемі з різними мовами програмування, такими як C/C++, Ruby, Scala, PHP, але найширше використання він зазнав саме з мовою програмування Java. [25]

Однією з важливих особливостей цього фреймворку є декларативний підхід до опису проекту. Декларативний підхід дозволяє зручно та систематично описувати всі аспекти проекту, а Maven автоматично виконує збірку та управління залежностями відповідно до опису в файлі pom.xml, що значно полегшує розробку, робить її більш передбачуваною та спрощує управління проектом.

Основним інструментом для декларативного опису проекту в Maven є файл pom.xml, назва якого є абревіатурою від Project Object Model. У ньому є можливість встановити ідентифікатори проекту, такі як groupId, artifactId та version, що дозволяє унікально ідентифікувати проект та використати його в якості залежності де інде.

Також, він дозволяє декларувати пакети, від яких залежить наш проект, та вказати точну версію потрібного пакета, який буде автоматично скачано з репозиторію при збірці проекту. Він надає інструменти для вирішення конфліктів між залежностями, які можуть трапитись у випадку, коли різні залежності проекту використовують одну й ту саму бібліотеку, але різних версій, що може призвести, наприклад, до MethodNotFoundException.

Також Maven дозволяє використовувати плагіни для більш гнучкого налаштування процесу збірки проекту, та автоматизувати такі дії як: включення або виключення ресурсів у пакеті, публікація отриманого проекту, генерація документації, генерація коду, виконання зовнішнього коду тощо.

Додатково треба зазначити, що коректна робота Maven не залежить від операційної системи, на якій він буде використовуватися, всі налаштування та команди будуть ідентичними.

Структура проекту Maven включає в себе директорію головного проекту, в якій можуть знаходитись такі директорії як:

- src/main/java – директорія де зберігаються «.java» файли самого проекту, що розробляється;
- src/main/resources – директорія, де будуть зберігатися ресурси проекту, які будуть використовуватися під час виконання програмного коду. Вони можуть включати в себе: файли налаштувань, такі як «.properties», «.xml», «.yml» або «.json» файли; ресурси для веб-додатків, у якості прикладів яких можна навести HTML, CSS, JavaScript, зображення, шрифти тощо; файли перекладу, або ресурси для інтернаціоналізації та локалізації; SQL-скрипти для створення та оновлення бази даних; Будь-які інші ресурси, які не є вихідним кодом, але використовуються в проекті;

- src/test/java – директорія для зберігання програмного коду, що представляє собою тести для розробленого застосунку;
- src/test/resources – директорія, яка зберігає ресурси, які будуть доступні тільки при виконанні тестів.
- Також, будь-який Maven проект може мати в собі модулі, що є самостійними Maven підпроектами, зі своїм власним «pom.xml» файлом, зі своїми власними залежностями, налаштуваннями та ресурсами. Використання модулів дозволяє організувати і структурувати код програми у більш логічні та окремі частини, що в свою чергу полегшує розробку, управління та підтримку проекту.

Будь-яка з наведених вище директорій не є обов'язковою і може як існувати, так і не існувати у проекті Maven, що дозволяє не створювати та не тримати у структурі проекту зайві пусті директорії. Приклад структури Maven проекту наведено на рисунку 3.8.

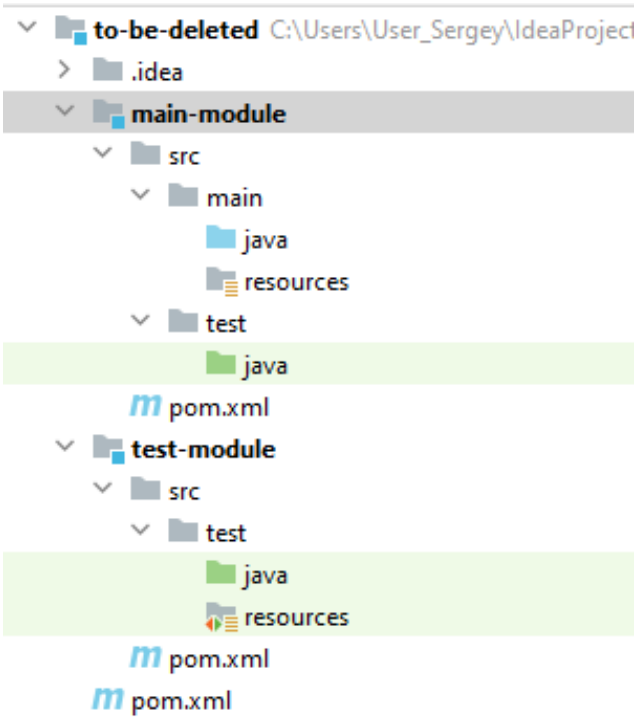

Рис. 3.8. Структура проекту Maven

maven-compiler-plugin

maven-compiler-plugin – це один з плагінів Apache Maven, який використовується для налаштування та керування компіляцією проектів написаних мовою програмування Java. Завдяки цьому плагіну можливо визначати параметри компіляції, такі як версія Java, джерело коду та директорію, куди будуть складатися скомпільовані класи, він дозволяє встановити діапазон використання оперативної пам’яті, яка буде використовуватися при компіляції проекту, передавати додаткові аргументи компілятору, налаштувати використання компілятора відмінного від javac, та багато іншого. [26]

Приклад використання наведено на рисунку 3.9.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <source>8</source>
        <target>8</target>
    </configuration>
</plugin>
```

Рис. 3.9. Налаштований maven-compiler-plugin

maven-source-plugin

maven-source-plugin – це ще один плагін від Apache для Maven, який забезпечує доступ до написаного коду поточного проекту іншим програмістам завдяки створенню архівів jar, що містять всі вихідні файли проекту. [27]

Завдяки ньому можливо вказати ім’я майбутнього архіва, вказувати фільтри та обмеження щодо того, які файли та класи потраплять у архів, вказувати, куди саме створений пакет буде покладено та багато іншого. Приклад використання наведено на рисунку 3.10.

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.2.1</version>
    <executions>
        <execution>
            <id>attach-sources</id>
            <goals>
                <goal>jar-no-fork</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Рис. 3.10. Налаштований maven-source-plugin

maven-javadoc-plugin

maven-javadoc-plugin – це доволі розповсюджений плагін для Maven, який використовується для автоматичної генерації документації до написаного коду. Цей плагін автоматично збирає всі javadoc коментарі, та генерує HTML-документацію, яка допомагає іншим розробникам швидко зрозуміти, як використовувати написаний код.

Для того, щоб використовувати maven-javadoc-plugin, його достатньо додати в pom.xml файл створеного проекту, але він також підтримує різні налаштування, такі як шлях до директорії, в якій буде збережена документація, або виключення певних класів або пакетів з процесу генерації документації. [28]. Приклад використання наведено на рисунку 3.11.

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-javadoc-plugin</artifactId>
    <version>2.9.1</version>
    <executions>
        <execution>
            <id>attach-javadocs</id>
            <goals>
                <goal>jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Рис. 3.11. Налаштований maven-javadoc-plugin

nexus-staging-maven-plugin

nexus-staging-maven-plugin - це плагін для Maven, який дозволяє автоматизувати проходження по крокам робочого процесу стадіювання Nexus завдяки виконанню відповідних команд Maven. [29] Приклад використання наведено на рисунку 3.12.

```
<plugin>
    <groupId>org.sonatype.plugins</groupId>
    <artifactId>nexus-staging-maven-plugin</artifactId>
    <version>1.6.7</version>
    <extensions>true</extensions>
    <configuration>
        <serverId>ossrh</serverId>
        <nexusUrl>https://s01.oss.sonatype.org/</nexusUrl>
        <autoReleaseAfterClose>true</autoReleaseAfterClose>
    </configuration>
</plugin>
```

Рис. 3.12. Налаштований nexus-staging-maven-plugin

Цей плагін зазвичай використовується програмістами, які розробляють бібліотеки або пакети, які в подальшому планується публікувати в центральному Maven-репозиторії. Він значно спрощує процес публікації, забезпечує контроль версій та безпеку артефактів, а також допомагає гарантувати доступність створеного артефакту іншим користувачам Maven.

maven-gpg-plugin

maven-gpg-plugin – популярний плагін для створення підпису артефактів, що в подальшому планується публікувати у Maven репозиторіях. Для створення електронного підпису цей плагін використовує відкритий криптографічний інструмент GnuPG [30]. Приклад використання наведено на рисунку 3.13.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-gpg-plugin</artifactId>
    <version>1.5</version>
    <executions>
        <execution>
            <id>sign-artifacts</id>
            <phase>verify</phase>
            <goals>
                <goal>sign</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Рис. 3.13. Налаштований maven-gpg-plugin

JUnit

Рис. 3.14. Логотип Junit 5

JUnit – популярний фреймворк для тестування коду написаного мовою програмування Java.

JUnit надає такі інструменти як: [31]

- анотації для позначення методів як тестових;
- анотації для позначення методів, які повинні виконуватись перед тестом або після нього;
- анотації для групування тестів в набори тестів;
- деякий набір методів для перевірки очікуваних результатів;

Незважаючи на те, що JUnit і надає деякий набір методів для перевірки очікуваного результату, і у п’ятій версії цього інструмента він став значно ширшим ніж це було раніше, але все ж таки цей функціонал і досі є не дуже

зручним для використання і іноді потребує написання шаблонного коду для того, щоб зробити певні перевірки. Наприклад, якщо у нас метод що тестується повертає CompletableFuture, то потрібно власноруч його завершити та тільки після цього перевіряти отримані результати.

Якщо ж є потреба мати більш зручний інструмент для здійснення наших перевірок, то доцільним буде використати AssertJ.

AssertJ

AssertJ – це бібліотека для тестування програмного коду написаного на Java, яка надає багатий набір методів для зручної і експресивної верифікації поведінки написаного коду.

Основними перевагами AssertJ можу зазначити: [32]

- AssertJ має велику кількість методів для перевірки різних умов, і окрім загальних перевірок, як то перевірка на еквівалентність та перевірка на null або валідація типів, є можливість порівнювати колекції не зважаючи на порядок елементів в них, перевіряти результат виконання CompletableFuture, порівнювати об’єкти за виключенням деяких полів тощо.
- AssertJ надає Fluent API, що дозволяє ланцюжком методів виконувати різні перевірки для одного й того ж самого об’єкту, без необхідності явно передавати його у наступний метод перевірки, що призводить до покращення читабельності написаного коду.
- AssertJ має можливість інтуїтивно легко використовувати Soft Assertions, що дозволяє тестовому методу не позначити тестовий випадок як хибний, після першої невдалої перевірки, а спочатку зробити всі «м’які» перевірки, і тільки потім, якщо якісь з перевірок виявили хибний результат, позначити тестовий випадок як не пройдений і вивести всі результати тестування;

```
String value = "Hi";
CompletionStage<String> actual = CompletableFuture.completedFuture(value);
assertThat(actual) CompletableFutureAssert<String>
        .isInstanceOf(CompletionStage.class)
        .succeedsWithin(Duration.ofSeconds(1)) ObjectAssert<String>
        .isInstanceOf(String.class)
        .isEqualTo(value);
```

Рис. 3.15. Приклад використання AssertJ

Mockito

Mockito, це фреймворк, що дозволяє створювати тестові об’єкти, які імітують роботу реальних об’єктів, що дозволяє програмістам створювати юніт тести, без використання справжніх об’єктів-залежностей і, відповідно, не залежати від логіки роботи залежностей і зосередитись на тестуванні класа, що тестується. [33]

compile-testing

Для тестування обробника анотацій є потреба у використанні інструмента, який би дозволив виконувати код обробника анотацій та перевіряти отриманий результат. Такою бібліотекою є compile-testing від Google, Основна ідея якої полягає в тому, що тестовий код, буде компілюватися під час виконання тесту.

Приклад тесту наведено на рисунку 3.16

```
@Test
public void should_create_new_class_with_custom_column() {
    // given
    JavaFileObject javaFileObject =
            JavaFileObjects.forResource( resourceName: "FancyTableWithCustomColumn.java");
    // when
    Compilation compilation =
            javac()
                    .withProcessors(
                            new ActiveJdbcRequiredPropertyProcessor()
                    )
                    .compile(javaFileObject);
    // then
    assertThat(compilation)
            .succeededWithoutWarnings();
    assertThat(compilation) CompilationSubject
            .generatedSourceFile(
                    qualifiedName: "FancyTableWithCustomColumnWrapper"
            ) JavaFileObjectSubject
            .hasSourceEquivalentTo(
                    JavaFileObjects
                            .forResource(
                                    resourceName: "FancyTableWithCustomColumnWrapper.java"
                            )
            );
}
```

Рис. 3.16. Тест написаний за використанням compile-testing

1.5. Опис структури програми та алгоритмів її функціонування

Алгоритм функціонування програми що розробляється полягає у тому, що вона буде створювати нові класи-обгортки для ActiveJDBC класів, які у свою чергу повинні мати всі необхідні очікувані методи для взаємодії з оригінальним об’єктом ActiveJDBC. Також, цей клас-обгортка має мати метод «getActivejdbcObject()», який буде надавати доступ до оригінального об’єкту.

Структура програми поділена на декілька шарів зображених на рисунку 3.17, і тут є потреба зупинитися, та розглянути їх детальніше.

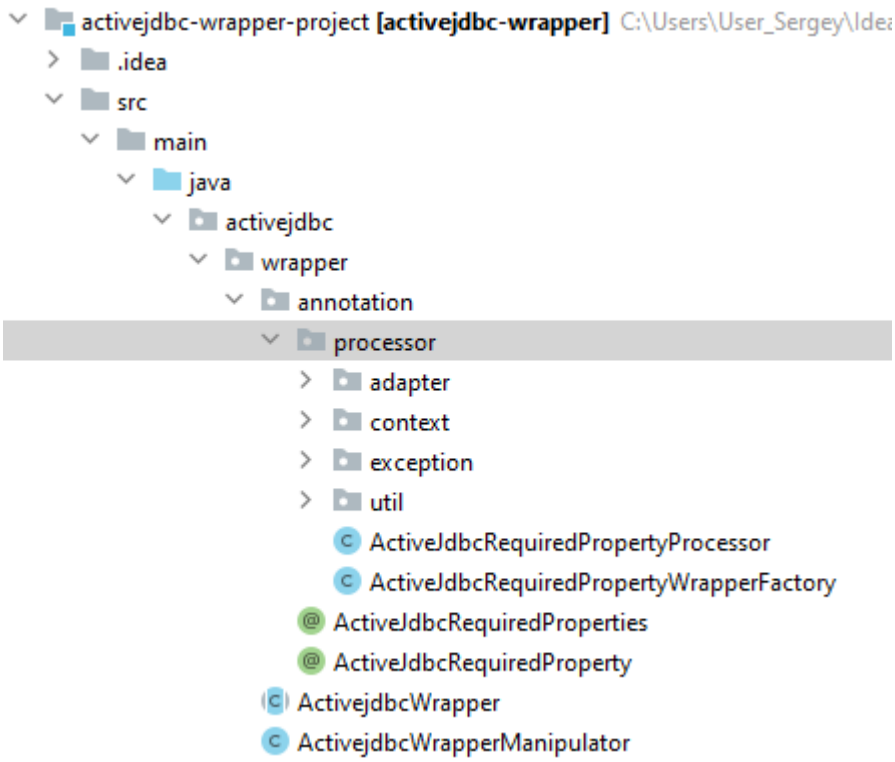

Рис. 3.17. Структура процесора анотацій ActiveJDBC Wrapper

activejdbc.wrapper – у цьому пакеті зберігаються базовий клас для будь якого класа-обгортки ActivejdbcWrapper, та клас ActivejdbcWrapperManipulator.

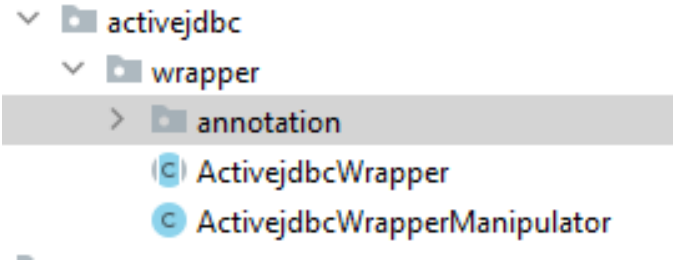

Рис. 3.18. Структура пакета activejdbc.wrapper

```
package activejdbc.wrapper;

import org.javalite.activejdbc.Model;

1 inheritor  erioh
public abstract class ActivejdbcWrapper<T extends Model> {
    1 usage  1 implementation  erioh
    protected abstract T getActivejdbcObject();
}
```

Рис. 3.19. Структура класу ActivejdbcWrapper

Як було зазначено вище, клас ActivejdbcWrapper буде використовуватися у якості базового класа для будь якого згенерованого класа-обгортки і єдине що він буде надавати, це метод «getActivejdbcObject()», який повинен повернути оригінальний ActiveJDBC об’єкт. Також є потреба звернути увагу на те, що цей метод має область бачення «protected», тому програміст який буде використовувати згенерований клас-обгортку не буде мати можливості безпосередньо його викликати. Це зроблено для того, щоб програміст користувався згенерованим об’єктом, як звичайним POJO, і якщо йому потрібно буде отримати доступ до оригінального об’єкту, то треба буде скористатися інструментом зображеним на рисунку 3.20.

```
public class ActivejdbcWrapperManipulator {
    1 usage   erioh *
    public <T extends Model, E extends ActivejdbcWrapper<T>> T getActivejdbcObject(
            E wrapper) {
        return wrapper.getActivejdbcObject();
    }
}
```

Рис. 3.20. Структура класу ActivejdbcWrapperManipulator

Мета існування класу ActivejdbcWrapperManipulator як раз і полягає у тому, щоб отримати доступ до оригінального ActiveJDBC об’єкту. Так як він лежить у тому ж самому пакеті, що і ActivejdbcWrapper, то він відповідно має доступ до метода «getActivejdbcObject()» і може його відповідно викликати.

activejdbc.wrapper.annotation – у цьому пакеті зберігаються анотації, якими буде користуватися програміст, для вказування того, які поля він очікує отримати, та деяких налаштувань. Структура пакету зображена на рисунку 3.21.

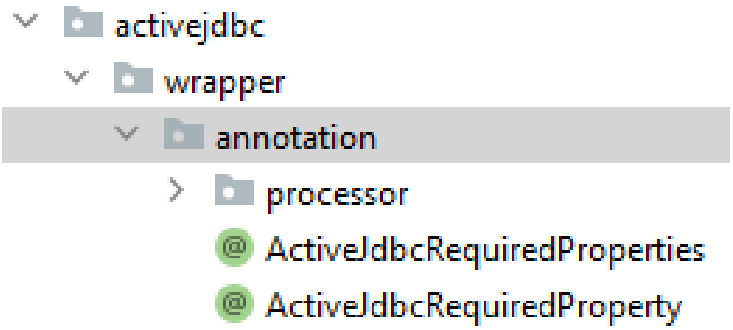

Рис. 3.21. Структура пакета activejdbc.wrapper.annotation

Анотація ActiveJdbcRequiredProperties, що зображена на рисунку 3.22, є анотацією-збирачем.

```
package activejdbc.wrapper.annotation;

import ...

3 usages  erioh
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ActiveJdbcRequiredProperties {
    no usages  erioh
    ActiveJdbcRequiredProperty[] value();
}
```

Рис. 3.22. Анотація ActiveJdbcRequiredProperties

Так як у мові програмування Java є обмеження що до використання анотацій, а саме, що неможливо встановити більш ніж одну анотацію одного типу на один елемент, тому є потреба у створенні анотації-збирача, яка дозволить обійти це обмеження. Java під час компіляції проекту буде замість того, щоб напряму використовувати анотації ActiveJdbcRequiredProperty, буде їх складати у ActiveJdbcRequiredProperties, і використовувати її.

Анотація ActiveJdbcRequiredProperty, що зображена на рисунку 3.23, дозволяє вказати, яке поле очікується отримати з таблиці бази даних, який тип

даних очікується на виході, а також бажаний шаблон для генерації назв методів, у тому випадку, коли автоматично згенерована назва не влаштовує користувача.

```
package activejdbc.wrapper.annotation;

import java.lang.annotation.*;

25 usages  erioh
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
@Repeatable(value = ActiveJdbcRequiredProperties.class)
public @interface ActiveJdbcRequiredProperty {
    erioh
    String columnName();

    18 usages  erioh
    Class<?> clazz();

    erioh
    String desiredFieldName() default "";
}
```

Рис. 3.23. Анотація ActiveJdbcRequiredProperty

Треба зазначити, що всі наведені анотації повинні мати анотації «@Target(ElementType.TYPE)», бо є потреба, щоб вони використовуватись виключно на самому класі, та «@Retention(RetentionPolicy.SOURCE)», таким чином буде вказано, що ці анотації повинні бути доступні лише на етапі компіляції.

activejdbc.wrapper.annotation.processor – цей пакет зберігає сам код процесора анотацій ActiveJdbcRequiredPropertyProcessor та клас-фабрику ActiveJdbcRequiredPropertyWrapperFactory, який і буде відповідальний за побудову нового класу. Структура пакету зображена на рисунку 3.24.

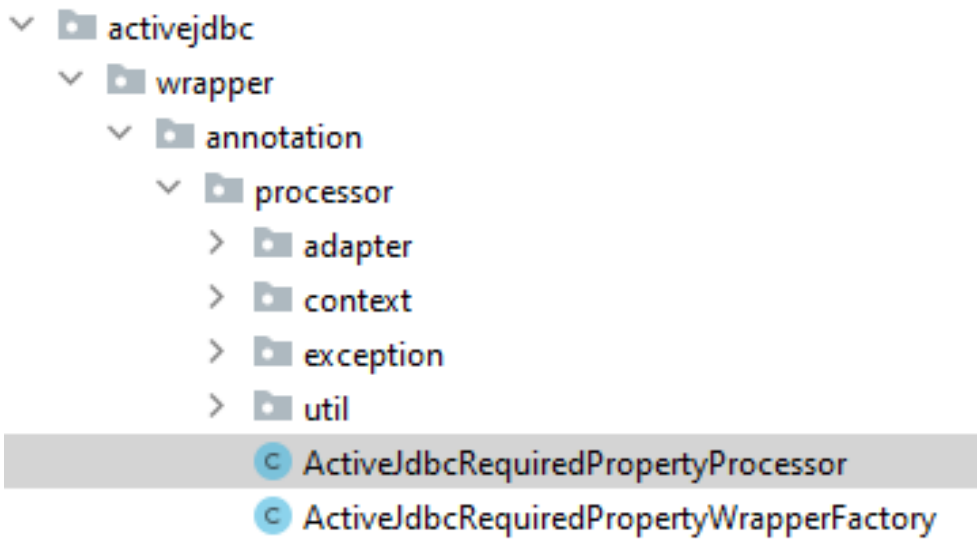

Рис. 3.24. Структура пакета activejdbc.wrapper.annotation.processor

Як було зазначено вище, клас ActiveJdbcRequiredPropertyProcessor є вхідною точкою у процес обробки анотації. Структура класу зображена на рисунку 3.25.

Рис. 3.25. Структура класу ActiveJdbcRequiredPropertyProcessor

У нього є такі методи як: «init()», що створює контекст для процесу та будує налаштовану фабрику, яка буде в подальшому використовуватись для генерації нових класів; «getSupportedSourceVersion()», що буде повертати поточну версію Java; «getWrapperSuffix», що буде повертати бажаний суфікс для згенерованих класів; «filterNeededAnnotationMirrors», що надає доступ до об’єктів AnnotationMirror, які зберігають дані з наведених вище анотацій; «getBuilderMethodPrefix», який буде повертати бажаний суфікс для методів класа

будівельника; головний метод «process», який безпосередньо і буде викликатися під час компіляції коду.

У свою чергу, клас ActiveJdbcRequiredPropertyWrapperFactory відповідає за головний робочий процес генерації потрібних методів та класів, який можна побачити на рисунку 3.26.

```
public String build(String packageName,
                    String className,
                    List<? extends AnnotationMirror> annotationMirrors) {
    List<ColumnContext> columnContexts = annotationMirrors.stream()
            .map(this::getColumnContext)
            .collect(Collectors.toList());
    WrapperClassBuilder wrapperClassBuilder = new WrapperClassBuilder(
            packageName,
            className,
            columnContexts,
            annotationProcessorContext);
    // add setters
    return wrapperClassBuilder.withSetters()
            // add getters
            .withGetters()
            // add get activejdbc object method
            .withMethodGetActivejdbcObject()
            // add toString (using getters)
            .withToString()
            // add equals
            .withEquals()
            // and hashcode (using getters)
            .withHashCode()
            // add builder method with builder class
            .withBuilder()
            .buildClassBody();
}
```

Рис. 3.26. Головний потік генерації нового класу

activejdbc.wrapper.annotation.processor.util – пакет де зберігаються різноманітні класи загального використання, що допомагають створювати

текстове відображення згенерованого класу. Структура пакету зображена на рисунку 3.27.

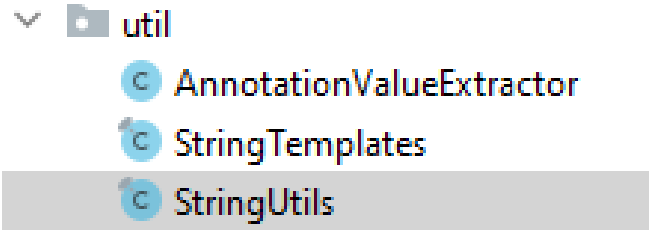

Рис.3.27. Структура пакета activejdbc.wrapper.annotation.processor.util

activejdbc.wrapper.annotation.processor.context – пакет де зберігаються класи контекстів. Саме тут відбувається налаштування всіх об’єктів, та зберігаються екземпляри всіх наявних стратегій. Структура пакету наведена на рисунку 3.28.

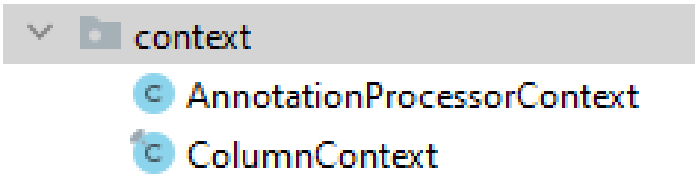

Рис. 3.28. Структура пакета context

activejdbc.wrapper.annotation.processor.adapter.builder– пакет, що зберігає у собі всі наявні стратегії, які будуть використовуватися при побудові нового класу-обгортки. Структура цього пакету наведена на рисунку 3.29.

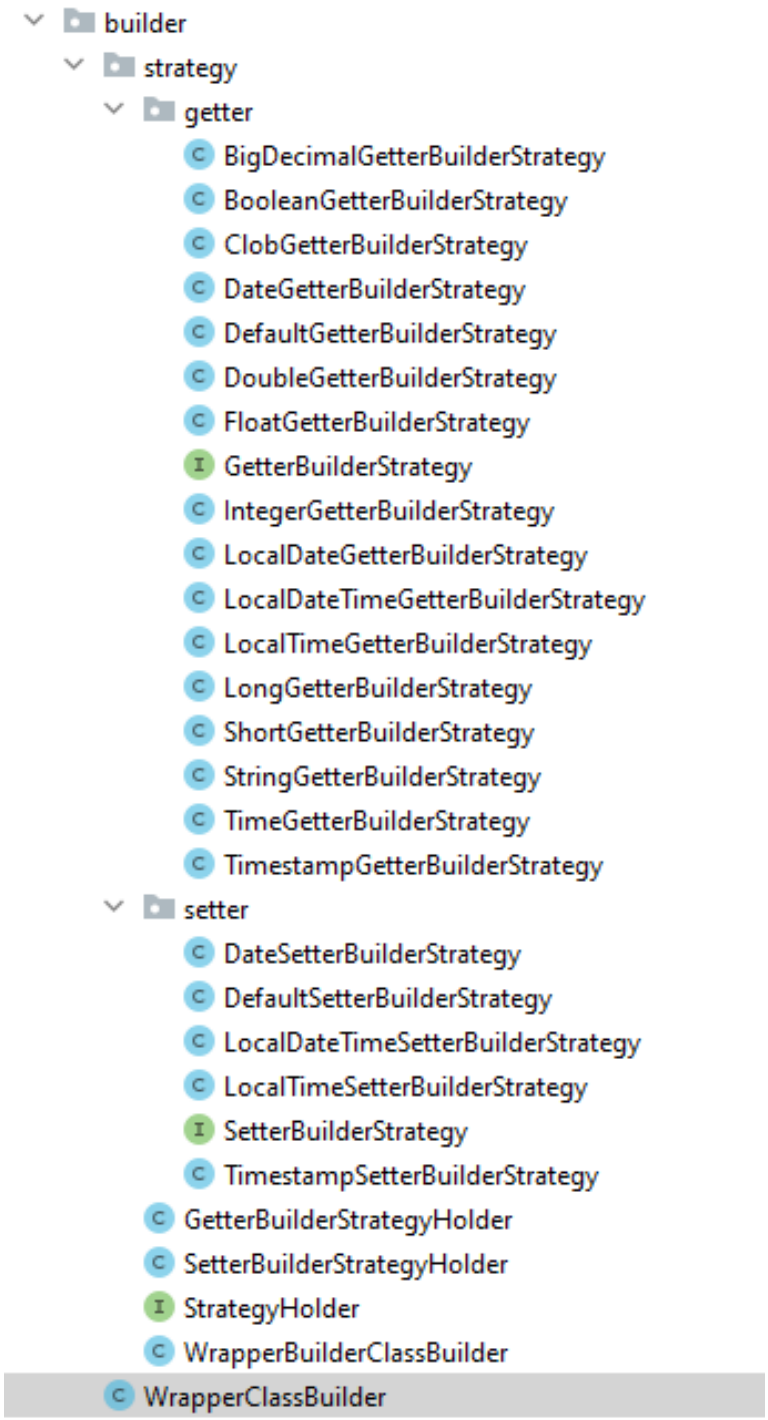

Рис. 3.29. Структура пакетів builder

1.6. Організація вхідних та вихідних даних програми

Виходячи з вимог до програмного продукту видно, що вхідними даними програми є класи позначені анотаціями ActiveJdbcRequiredProperty, у той час як вихідним результати роботи нашої програми буде згенерований програмний код,

яким в подальшому буде користуватися програміст для написання власного коду.

1.7. Опис розробленого програмного продукту

1.7.1. Використані технічні та програмні засоби

В процесі написання та тестування інструменту генерації коду для ORM системи ActiveJDBC були використані наступні технічні засоби:

Тестовий комп’ютер №1:

- операційна система Windows 11;
- Java 8
- оперативна пам’ять обсягом 16 гб;
- процесор Intel Core i7-8700 3.20 GHz;
- накопичувач на жорсткому диску обсягом 256 гб;
- відеоадаптер NVIDIA GeForce GTX 1660;
- клавіатура та маніпулятор типу «миш».

Тестовий комп’ютер №2:

- операційна система MacOS Ventura 13.5;
- Java 17
- оперативна пам’ять обсягом 16 гб;
- процесор Apple M1 Pro;
- накопичувач на жорсткому диску обсягом 512 гб;
- клавіатура та маніпулятор типу «трекпед».

Тестовий комп’ютер №3:

- операційна система Linux Ubuntu 20.04;
- Java 8, 11, 17
- оперативна пам’ять обсягом 32 гб;
- процесор Intel Core i7-3770 3.40 GHz;

- накопичувач на жорсткому диску обсягом 256 гб;
- клавіатура та маніпулятор типу «миш».

У той же час, додатково треба зазначити, що вказані вище технічні засоби не є обов’язковими для використання.

У якості програмних засобів, що використовувалися для розробки даного програмного продукту можна зазначити середовище розробки IntelliJ Idea та систему управління версіями Git.

IntelliJ IDEA

IntelliJ IDEA - це інтегроване середовище розробки, що було розроблено компанією JetBrains, та перша версія якої вийшла ще у 2001-му році. Це середовище розробки дуже розповсюджено використовується для програмування на таких мовах програмування як Java, Kotlin, Groovy, Scala та багато інших, для підтримки яких існують відповідні плагіни. [34]

IntelliJ IDEA надає багато різноманітного функціоналу, що значно полегшує процес розробки програмного забезпечення, серед яких можна зазначити:

- широкий набір автодоповнень, що допомагають розробнику швидше та точніше писати код;
- система аналізу коду, яка допомагає виявляти помилки, неправильне використання API та надає рекомендації щодо поліпшення коду;
- інструменти для рефакторінгу, такі як: можливість пошуку та вилучення дублікатів коду; винесення значень у змінні та константи; автоматичне створення методів з частини коду тощо;
- інтеграція з різними системами контролю версій, такими як Git, SVN, Mercurial та іншими;
- потужна система плагінів, що дозволяє сильно розширити функціонал самого середовища розробки;

Git

Git – це розподілена система контролю версій, яка використовується для ведення історії змін в програмному коді та спільної роботи багатьох програмістів над проектами. Ця система була створена іконою у світі програмного забезпечення Лінусом Торвальдсом для того, щоб керувати розробкою ядра Linux [35], але завдяки своєю зручністю та потужному функціоналу ця система контролю версій стала дуже поширено використовуватись при розробці програмного забезпечення.

1.7.2. Опис інтерфейсу користувача та приклад використання

Для подальшого використання розробленого інструмента, програміст повинен додати її у якості залежності у свій Maven проект, що зображено на рисунку 3.30.

```
<dependency>
    <groupId>io.github.erioh</groupId>
    <artifactId>activejdbc-wrapper</artifactId>
    <version>1.0.5-SNAPSHOT</version>
</dependency>
```

Рис. 3.30. Залежність додано у pom.xml файл

Після чого програмісту достатньо буде тільки додати анотації ActiveJdbcRequiredProperty до існуючого ActiveJDBC класу та скомпілювати свій проект. Приклад використання анотацій наведено на рисунку 3.31.

```
@Table("PERSON")
@IdName("PERSON_ID")
@ActiveJdbcRequiredProperty(columnName = "PERSON_ID", clazz = Long.class)
@ActiveJdbcRequiredProperty(columnName = "FIRST_NAME", clazz = String.class)
@ActiveJdbcRequiredProperty(columnName = "LAST_NAME", clazz = String.class)
@ActiveJdbcRequiredProperty(columnName = "BIRTH_DATE", clazz = LocalDate.class)
@ActiveJdbcRequiredProperty(columnName = "CREATION_DATE", clazz = LocalDateTime.class)
public class Person extends Model {
}
```

Рис. 3.31. анотації ActiveJdbcRequiredProperty додано до класу Person

В результаті буде отримано новий клас з суфіксом Wrapper, який програміст і буде використовувати при написанні власного коду. Цей клас вже має всі потрібні геттери та сеттери, які будуть правильним чином перетворювати вхідні та вихідні параметри у ті, з якими працює ActiveJDBC, має згенерований клас будівельник, та перевизначені методи equals та hashCode. Частина згенерованого класа-обгортки наведена на рисунку 3.32.

```
package com.sdemenkov.dipoma;
import com.sdemenkov.dipoma.Person;
14 usages
public class PersonWrapper extends
        activejdbc.wrapper.ActivejdbcWrapper<Person>{
13 usages
private Person person;
 1 usage
 public PersonWrapper() { this.person = new Person(); }
 2 usages
 public PersonWrapper(Person person) { this.person = person; }

5 usages
public java.lang.String getLastName() { return person.getString( attributeName: "LAST_NAME"); }
```

Рис. 3.32. Частина згенерованого класа-обгортки

В результаті, якщо програмісту потрібно створити новий запис в базі даних, то все що йому потрібно, ще створити новий об’єкт обгортку використавши статичний метод builder(), що призведе до того, що всередині об’єкта-обгортки буде автоматично створено новий екземпляр класу

ActiveJDBC, в який будуть передаватися значення полів використовуючи патерн Будівельник. Після цього, цей об’єкт-обгортку потрібно передати на вхід об’єкту типу ActivejdbcWrapperManipulator, який поверне повністю готовий до використання ActiveJDBC об’єкт. Приклад коду наведено на рисунку 3.33.

```
// Може бути Spring біном, але зараз ми створемо маніпулятор тут
ActivejdbcWrapperManipulator manipulator = new ActivejdbcWrapperManipulator();
// створюємо об'єкт-обгортку, та додаємо у нього потрібні дані
PersonWrapper wPerson = PersonWrapper.builder()
        .personId(1L)
        .firstName("Serhii")
        .birthDate(LocalDate.of(
                year: 1985,
                month: 3,
                dayOfMonth: 23))
        .creationDate(LocalDateTime.of(
                year: 2023,
                month: 11,
                dayOfMonth: 5,
                hour: 22,
                minute: 38,
                second: 12))
        .build();
// отримуємо доступ до оригінального об'єкта
Person activejdbcPerson = manipulator.getActivejdbcObject(wPerson);
// створюємо нову строку у базі даних
activejdbcPerson.insert();
```

Рис. 3.33. Приклад створення нового запису у базі даних

Якщо ж програміст хоче порівняти два об’єкта, то йому потрібно тільки дістати існуючий об’єкт з бази даних та передати його на вхід конструктора об’єкта-обгортки. В результаті він може його легко порівняти з об’єктом, який описує очікуваний результат завдяки тому, що метод equals для об’єктів-обгорток автоматично згенеровано. Приклад використання наведено на рисунку 3.34.

```
long personId = 1L;
// створюємо об'єкт-обгортку з очікуваними результатами.
PersonWrapper expectedPerson = PersonWrapper.builder()
        .personId(personId)
        .firstName("Serhii")
        .birthDate(LocalDate.of(
                year: 1985,
                month: 3,
                dayOfMonth: 23))
        .creationDate(LocalDateTime.of(
                year: 2023,
                month: 11,
                dayOfMonth: 5,
                hour: 22,
                minute: 38,
                second: 12))
        .build();
// отримаємо об'єкт типу Person з бази даних
Person originalPerson = Person.findById(personId);
// створюємо новий об'єкт типу PersonWrapper,
// передаючи у конструктор об'єкт Person
PersonWrapper personWrapper = new PersonWrapper(originalPerson);
// порівнюємо отримані об'єкти
assertThat(personWrapper).isEqualTo(expectedPerson);
```

Рис.3.34. Приклад порівняння двох об’єктів

Модифікацію існуючого запису у базі даних можна легко провести завдяки тому, що у об’єкта-обгортки є згенеровані методи сеттери, що дозволяє встановити потрібні дані не замислюючись над тим, чи підтримує їх ActiveJDBC та які маніпуляції з цими даними потрібно буде зробити, щоб ActiveJDBC їх коректно обробив та зберіг. Приклад коду модифікації існуючої строки у базі даних наведено на рисунку 3.35.

```
// знаходимо існуючий запис у базі даних
Person persistedPerson = Person.findById(1L);
// створюємо для неї об'єкт обгортку
PersonWrapper personWrapper = new PersonWrapper(persistedPerson);
// змінюємо фамілію
personWrapper.setLastName("Demenkov");
// використовуючи об'єкт маніпулятор отримаємо доступ до
// оригінального об'єкту
new ActivejdbcWrapperManipulator()
        .getActivejdbcObject(personWrapper)
        // зберігаємо запис у базу даних
        .saveIt();
```

Рис. 3.35. Модифікація існуючої строки у базі даних

У тому випадку, якщо запропоновані імена методів об’єкту будівельника не задовольняють домовленостям прийнятим на проекті, або програміст хоче використати інший суфікс для автоматично згенерованих класів, то він може передати у обробник анотацій такі параметри як:

- -Aactivejdbc.wrapper.suffix – цей параметр відповідає з суфікс, що буде використовуватись при побудові нового класа-обгортки;
- -Aactivejdbc.wrapper.builder.method.prefix – як видно з назви, цей параметр відповідає за префікс до методів об’єкта будівельника;

Щоб автоматизувати цей процес і не передавати ці параметри власноруч при кожному запуску компілятора, достатньо конфігурувати maven-compiler-plugin, що зображено на рисунку 3.36.

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <compilerArgs>
            <arg>-Aactivejdbc.wrapper.suffix=NewWrapperSuffix</arg>
            <arg>-Aactivejdbc.wrapper.builder.method.prefix=with</arg>
        </compilerArgs>
    </configuration>
</plugin>
```

Рис. 3.36. Конфігурація maven-compiler-plugin

В результаті одразу після компіляції будуть згенеровані нові класи з суфіксом «NewWrapperSuffix», та всі методи об’єкту будівельника будуть мати префікс «with», що і зображено на рисунку 3.37.

```
PersonNewWrapperSuffix personNewWrapperSuffix = PersonNewWrapperSuffix.builder()
        .withPersonId(1L)
        .withFirstName("Serhii")
        .withLastName("Demenkov")
        .build();
```

Рис. 3.37. Результат роботи сконфігурованого процесора анотацій

ВИСНОВКИ

У даній кваліфікаційній роботі був розроблений обробник анотацій, що дозволяє автоматично на етапі компіляції згенерувати код, що буде взаємодіяти з об’єктами ORM системи ActiveJDBC.

Розроблений інструмент дозволяє:

- вивільнити робочий час програміста від написання шаблонного рутинного коду, та зосередитись на вирішенні поставлених перед ним задач, а не на обслуговуванні інструмента взаємодії з базою даних;
- зменшити обсяг коду, що повинен пройти валідацію при проходженні процесу Code Review;
- підвищити читабельність отриманого коду, завдяки використанню звичних геттерів, сеттерів, та можливістю використання патерна Будівельник;
- підвищити стабільність отриманого коду завдяки тому, що інструмент автоматично зробить всі потрібні перевірки та перетворення даних, що будуть отримуватися з бази даних, або навпаки, зберігатися у ній;
- унеможливлює завчасне використання ActiveJDBC об’єкту завдяки використанню патерна Будівельник;

Актуальність поставленої задачі обумовлюється широким застосуванням ORM системи ActiveJDBC та відсутністю інструментів для вирішення наявних проблем та обмежень використання цієї ORM системи. Багато програмістів замість того, щоб писати бізнес-код витрачають свій час на підтримку самого інструмента та тестування написаного для нього коду, що напряму не відноситься до бізнеса замовника. Це у свою чергу уповільнює процес розробки програмного забезпечення, підвищує його вартість, ускладнює процес проходження Code Review та додає додаткові місця, де може виникнути помилка в коді.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Acceleo/Getting Started / URL: https://wiki.eclipse.org/Acceleo/Getting_Started. дата звернення: 27.10.2023.
2. Appmysite / URL: https://www.appmysite.com/. дата звернення: 27.10.2023.
3. Oltrogge M. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators // IEEE Symposium on Security and Privacy. — 2018. — p. 646
4. How to use ChatGPT to write code / URL: https://www.zdnet.com/article/how-to-use-chatgpt-to-write-code/. дата звернення: 27.10.2023.
5. Become a sponsor to Project Lombok / URL: https://github.com/sponsors/projectlombok. дата звернення: 28.10.2023.
6. Projectlombok / URL: https://projectlombok.org/. дата звернення: 28.10.2023.
7. HOW DOES PROJECT LOMBOK WORK? / URL: https://www.freelancinggig.com/blog/2019/04/06/how-does-project-lombok-work/. дата звернення: 28.10.2023.
8. Spring Data / URL: https://spring.io/projects/spring-data. дата звернення: 28.10.2023.
9. Proxying mechanisms / URL: https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch08s06.html. дата звернення: 28.10.2023.
10. Mapstruct / URL: https://mapstruct.org/. дата звернення: 28.10.2023.
11. JDK 14 Release Notes / URL: https://www.oracle.com/java/technologies/javase/14-relnote-issues.html. дата звернення: 28.10.2023.
12. Java Records: When & Why to use them / URL: https://medium.com/javarevisited/java-records-when-why-to-use-them-ebd48de637b6. дата звернення: 28.10.2023.
13. ActiveJDBC / URL: https://javalite.io/activejdbc. дата звернення: 28.10.2023.
14. Spring dominates the Java ecosystem with 60% using it for their main applications / URL: https://snyk.io/blog/spring-dominates-the-java-ecosystem-with-60-using-it-for-their-main-applications/. дата звернення: 31.10.2023.
15. JACY: a JVM-Based Intrusion Detection and Security Analysis System / URL: https://hal.science/hal-03168756v3. дата звернення: 31.10.2023.
16. Package java.lang.instrument / URL: https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/instrument/package-summary.html. дата звернення: 31 10 2023.

17. Dynamic Proxy Classes / URL: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html. дата звернення: 31.10.2023.
18. CGLib: недостающее руководство / URL: https://coderlessons.com/articles/java/cglib-nedostaiushchee-rukovodstvo. дата звернення: 31.10.2023.
19. Cglib / URL: https://github.com/cglib/cglib. дата звернення: 1.11.2023.
20. All About Annotations and Annotation Processors / URL: https://medium.com/swlh/all-about-annotations-and-annotation-processors-4af47159f29d. дата звернення: 1.11.2023.
21. Dependency Injection is Loose Coupling / URL: https://blog.ploeh.dk/2010/04/07/DependencyInjectionisLooseCoupling/. дата звернення: 1.11.2023.
22. Inversion of Control Containers and the Dependency Injection pattern / URL: https://martinfowler.com/articles/injection.html#InversionOfControl. дата звернення: 1.11.2023.
23. Refactoring Guru / URL: https://refactoring.guru/design-patterns. дата звернення: 2.11.2023.
24. History of Java / URL: https://www.javatpoint.com/history-of-java. дата звернення: 2.11.2023.
25. Maven Documentation / URL: https://maven.apache.org/guides/. дата звернення: 2.11.2023.
26. Apache Maven Compiler Plugin / URL: https://maven.apache.org/plugins/maven-compiler-plugin/. дата звернення: 2.11.2023.
27. Apache Maven Source Plugin / URL: https://maven.apache.org/plugins/maven-source-plugin/. дата звернення: 2.11.2023.
28. Apache Maven Javadoc Plugin / URL: https://maven.apache.org/plugins/maven-javadoc-plugin/. дата звернення: 4.11.2023.
29. Nexus Staging Maven Plugin / URL: https://github.com/sonatype/nexus-maven-plugins/tree/main/staging/maven-plugin. дата звернення: 4.11.2023.
30. Apache Maven GPG Plugin / URL: https://maven.apache.org/plugins/maven-gpg-plugin/. дата звернення: 4.11.2023.
31. JUnit 5 / URL: https://junit.org/junit5/. дата звернення: 4.11.2023.
32. AssertJ - fluent assertions java library / URL: https://assertj.github.io/doc/. дата звернення: 4.11.2023.
33. Tasty mocking framework for unit tests in Java / URL: https://site.mockito.org/. дата звернення: 4.11.2023.

34. IntelliJ IDEA – the Leading Java and Kotlin IDE / URL: https://www.jetbrains.com/idea/. дата звернення: 5.11.2023.
35. Chacon S., Straub B. Git Pro // Apress - 2023.

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

ActivejdbcWrapperManipulator.java

```
package activejdbc.wrapper;
import org.javalite.activejdbc.Model;
public class ActivejdbcWrapperManipulator {
    public <T extends Model, E extends ActivejdbcWrapper<T>> T getActivejdbcObject(E wrapper)
{
        return wrapper.getActivejdbcObject();
    }
}
```

ActivejdbcWrapper.java

```
package activejdbc.wrapper;
import org.javalite.activejdbc.Model;
public abstract class ActivejdbcWrapper<T extends Model> {
   protected abstract T getActivejdbcObject();
}
```

ActiveJdbcRequiredProperty.java

```
package activejdbc.wrapper.annotation;
import java.lang.annotation.*;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
@Repeatable(value = ActiveJdbcRequiredProperties.class)
public @interface ActiveJdbcRequiredProperty {
   String columnName();
   Class<?> clazz();
   String desiredFieldName() default "";
}
```

ActiveJdbcRequiredProperties.java

```
package activejdbc.wrapper.annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ActiveJdbcRequiredProperties {
   ActiveJdbcRequiredProperty[] value();
}
```

ActiveJdbcRequiredPropertyWrapperFactory.java

```
package activejdbc.wrapper.annotation.processor;
import activejdbc.wrapper.annotation.processor.adapter.builder.WrapperClassBuilder;
import activejdbc.wrapper.annotation.processor.context.AnnotationProcessorContext;
import activejdbc.wrapper.annotation.processor.context.ColumnContext;
import activejdbc.wrapper.annotation.processor.util.AnnotationValueExtractor;
import activejdbc.wrapper.annotation.processor.util.StringUtils;
import javax.lang.model.element.AnnotationMirror;
import javax.lang.model.element.AnnotationValue;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;
public class ActiveJdbcRequiredPropertyWrapperFactory {
   private final AnnotationProcessorContext annotationProcessorContext;
   public ActiveJdbcRequiredPropertyWrapperFactory(AnnotationProcessorContext
annotationProcessorContext) {
      this.annotationProcessorContext = annotationProcessorContext;
   }

   public String build(String packageName, String className, List<? extends AnnotationMirror>
annotationMirrors) {
```

```
            List<ColumnContext> columnContexts = annotationMirrors.stream()
                    .map(this::getColumnContext)
                    .collect(Collectors.toList());
            WrapperClassBuilder wrapperClassBuilder = new WrapperClassBuilder(packageName,
classNameWithoutPackage, columnContexts, annotationProcessorContext);
            // add setters
            return wrapperClassBuilder.withSetters()
                    // add getters
                    .withGetters()
                    // add get activejdbc object method
                    .withMethodGetActivejdbcObject()
                    // add toString (using getters)
                    .withToString()
                    // add equals
                    .withEquals()
                    // and hashcode (using getters)
                    .withHashCode()
                    // add builder method with builder class
                    .withBuilder()
                    .buildClassBody();
        }
        private ColumnContext getColumnContext(AnnotationMirror annotationMirror) {
            String clazz = extract(annotationMirror, "clazz");
            String columnName = extract(annotationMirror, "columnName");
            String desiredFieldName = extract(annotationMirror, "desiredFieldName");
            return new ColumnContext(clazz, columnName, desiredFieldName);
        }

        private String extract(AnnotationMirror annotationMirror, String property) {
            return AnnotationValueExtractor.extract(annotationMirror.getElementValues(), property)
                    .map(AnnotationValue::getValue)
                    .map(Objects::toString)
                    .orElse(StringUtils.EMPTY_STRING);
        }
    }
```

ActiveJdbcRequiredPropertyProcessor.java

```
package activejdbc.wrapper.annotation.processor;
import activejdbc.wrapper.annotation.ActiveJdbcRequiredProperties;
import activejdbc.wrapper.annotation.ActiveJdbcRequiredProperty;
import activejdbc.wrapper.annotation.processor.context.AnnotationProcessorContext;
import activejdbc.wrapper.annotation.processor.exception.AnnotationProcessorException;
import activejdbc.wrapper.annotation.processor.util.AnnotationValueExtractor;
import activejdbc.wrapper.annotation.processor.util.StringUtils;
import javax.annotation.processing.*;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.*;
import javax.tools.Diagnostic;
import javax.tools.JavaFileObject;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
@SupportedAnnotationTypes({"activejdbc.wrapper.annotation.ActiveJdbcRequiredProperty",
"activejdbc.wrapper.annotation.ActiveJdbcRequiredProperties"})
@SupportedOptions({ActiveJdbcRequiredPropertyProcessor.ACTIVEJDBC_WRAPPER_SUFFI
X,ActiveJdbcRequiredPropertyProcessor.ACTIVEJDBC_BUILDER_METHOD_PREFIX})
public class ActiveJdbcRequiredPropertyProcessor extends AbstractProcessor {
   private static final String DEFAULT_WRAPPER_SUFFIX = "Wrapper";
   public static final String ACTIVEJDBC_WRAPPER_SUFFIX = "activejdbc.wrapper.suffix";
   public static final String ACTIVEJDBC_BUILDER_METHOD_PREFIX =
"activejdbc.wrapper.builder.method.prefix";
   private ActiveJdbcRequiredPropertyWrapperFactory wrapperFactory;
   @Override
   public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv) {
      for (TypeElement annotation : annotations) {
         Set<? extends Element> elementsAnnotatedWith =
roundEnv.getElementsAnnotatedWith(annotation);
         for (Element element : elementsAnnotatedWith) {
            if (element.getKind() != ElementKind.CLASS) {
               throw new IllegalArgumentException("Only classes can be annotated with
ActiveJdbcRequiredProperty");
```

```
                    }
                    processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE, "found
@ActiveJdbcRequiredProperty at " + element);
                    List<? extends AnnotationMirror> annotationMirrors =
filterNeededAnnotationMirrors(element.getAnnotationMirrors());
                    PackageElement packageElement =
processingEnv.getElementUtils().getPackageOf(element);
                    String className = element.getSimpleName().toString();
                    String wrapperClassBody =
wrapperFactory.build(packageElement.getQualifiedName().toString(), className, annotationMirrors);
                    try {
                        JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(className +
getWrapperSuffix());
                        try (PrintWriter out = new PrintWriter(sourceFile.openWriter())) {
                            out.print(wrapperClassBody);
                        }
                    } catch (IOException e) {
                        throw new AnnotationProcessorException(e);
                    }
                }
            }
            return false;
        }
        private List<? extends AnnotationMirror> filterNeededAnnotationMirrors(List<? extends
AnnotationMirror> annotationMirrors) {
            return annotationMirrors.stream()
                    .filter(annotationMirror ->
annotationMirror.getAnnotationType().toString().equals(ActiveJdbcRequiredProperties.class.getName()))
                    .map(AnnotationMirror::getElementValues)
                    .map(elementValues -> AnnotationValueExtractor.extract(elementValues, "value"))
                    .map(annotationValue -> (List<AnnotationMirror>) annotationValue.get().getValue())
                    .findFirst()
                    .orElseGet(() -> annotationMirrors.stream()
                            .filter(annotationMirror ->
annotationMirror.getAnnotationType().toString().equals(ActiveJdbcRequiredProperty.class.getName()))
                            .collect(Collectors.toList()));
        }
        @Override
```

```
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        AnnotationProcessorContext annotationProcessorContext = new
AnnotationProcessorContext(getWrapperSuffix(), getBuilderMethodPrefix());
        this.wrapperFactory = annotationProcessorContext.init();
    }
    @Override
    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.latestSupported();
    }
    private String getWrapperSuffix() {
        String suffix = this.processingEnv.getOptions().get(ACTIVEJDBC_WRAPPER_SUFFIX);
        if (StringUtils.isBlank(suffix)) {
            processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE,
String.format("Custom suffix is not set. Suffix [%s] will be used instead.\r\n",
DEFAULT_WRAPPER_SUFFIX));
            return DEFAULT_WRAPPER_SUFFIX;
        }
        if (!StringUtils.isValid(suffix)) {
            throw new IllegalArgumentException("Custom suffix has illegal characters. Please use only
characters in uppercase and lowercase, underscore, and numbers.");
        }
        return suffix;
    }
    private String getBuilderMethodPrefix() {
        String prefix =
this.processingEnv.getOptions().get(ACTIVEJDBC_BUILDER_METHOD_PREFIX);
        if (StringUtils.isBlank(prefix)) {
            processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE, "Custom builder
method prefix is not set. \r\n");
            return StringUtils.EMPTY_STRING;
        }
        if (!StringUtils.isValid(prefix)) {
            throw new IllegalArgumentException("Custom prefix has illegal characters. Please use
only characters in uppercase and lowercase, underscore, and numbers.");
        }
        return prefix;
    }
```

```
}
```

StringUtils.java

```
package activejdbc.wrapper.annotation.processor.util;
import java.util.Arrays;
import java.util.stream.Collectors;
public final class StringUtils {
   public static final String EMPTY_STRING = "";
   public static boolean isBlank(String string) {
      return string == null || string.trim().isEmpty();
   }
   public static boolean isValid(String string) {
      return string.trim().matches("^[a-zA-Z0-9_]+$");
   }
   public static String lowerCaseFirstCharacter(String string) {
      return Character.toLowerCase(string.charAt(0)) + string.substring(1);
   }
   public static String buildPropertyNameFromColumnName(String columnName) {
      String value = columnName.toLowerCase();
      String[] strings = value.split("_");
      return changeCapitalizationOfTheFirstCharacter(Arrays.stream(strings)
            .filter(string -> !isBlank(string))
            .map(string -> changeCapitalizationOfTheFirstCharacter(string, true))
            .collect(Collectors.joining()), false);
   }
   private static String changeCapitalizationOfTheFirstCharacter(String string, boolean
toUppercase) {
      String firstCharacter = string.substring(0, 1);
      String allOtherCharacters = string.substring(1);
      return toUppercase ? firstCharacter.toUpperCase() + allOtherCharacters :
firstCharacter.toLowerCase() + allOtherCharacters;
   }
   public static String buildMethodName(String propertyName, String prefix) {
      return StringUtils.isBlank(prefix) ?
            propertyName
            : prefix + changeCapitalizationOfTheFirstCharacter(propertyName, true);
```

```
    }
    private StringUtils() {
    }
}
```

StringTemplates.java

```
package activejdbc.wrapper.annotation.processor.util;
public final class StringTemplates {
    public static final String METHOD_GET_OBJECT_TEMPLATE = "protected %s
getActivejdbcObject() {%n" +
            "return %s;%n" +
            "}%n";
    /**
     * 1. toString implementation
     */
    public static final String TO_STRING_METHOD_TEMPLATE = "public java.lang.String
toString() {%n" +
            "return %s;%n" +
            "}%n";
    /**
     * 1. wrapper class name
     * 2. wrapper class name
     * 3. generated checks for equals
     */
    public static final String EQUALS_METHOD_TEMPLATE = "public boolean equals(Object o)
{%n" +
            "if (this == o) return true;%n" +
            "if (o == null || getClass() != o.getClass()) return false;%n" +
            "%s that = (%s) o;%n" +
            "return%s;%n" +
            "}%n";
    /**
     * 1. generated call of getters
     */
    public static final String HASH_CODE_METHOD_TEMPLATE = "public int hashCode()
{%n" +
```

```
            "return java.util.Objects.hash(%s);%n" +
            "}%n";
        /**
         * 1. Wrapper name
         * 2. activejdbc object name
         * 3. activejdbc object class
         */
        public static final String CONSTRUCTOR_WITHOUT_PARAMETERS_TEMPLATE = "
public %s() {%n" +
            "this.%s = new %s();%n" +
            "}%n";
        /**
         * 1. Wrapper name
         * 2. activejdbc object class
         * 2. activejdbc object name
         * 3. activejdbc object name
         * 4. activejdbc object name
         */
        public static final String CONSTRUCTOR_WITH_PARAMETER_TEMPLATE = " public
%s(%s %s) {%n" +
            "this.%s = %s;%n" +
            "}%n";
        /**
         * 1. Wrapper class name
         */
        public static final String BUILDER_METHOD_TEMPLATE = " public static %s builder()
{%n" +
            "return new %s();%n" +
            "}%n";
        /**
         * 1. Builder class name
         * 2. Method name
         * 3. Column type
         * 4. Column name
         * 5. Activejdbc wrapper object name
         * 6. Setters name
         * 7. Column name
         */
```

```
    public static final String BUILDER_SETTER_TEMPLATE = " public %s %s(%s %s) {%n" +
            "this.%s.%s(%s);%n" +
            "return this;%n" +
            "}%n";
}
```

AnnotationValueExtractor.java

```
package activejdbc.wrapper.annotation.processor.util;
import javax.lang.model.element.AnnotationValue;
import javax.lang.model.element.ExecutableElement;
import java.util.Map;
import java.util.Optional;
public class AnnotationValueExtractor {
    public static Optional<AnnotationValue> extract(Map<? extends ExecutableElement, ? extends
AnnotationValue> elementValues, String expectedPropertyName) {
        for (Map.Entry<? extends ExecutableElement, ? extends AnnotationValue> entry :
elementValues.entrySet()) {
            String propertyName = entry.getKey().getSimpleName().toString();
            if (propertyName.equalsIgnoreCase(expectedPropertyName)) {
                return Optional.ofNullable(entry.getValue());
            }
        }
        return Optional.empty();
    }
}
```

AnnotationProcessorException.java

```
public class AnnotationProcessorException extends RuntimeException{
    public AnnotationProcessorException(Throwable cause) {
        super(cause);
    }
}
```

ColumnContext.java

```
package activejdbc.wrapper.annotation.processor.context;
import java.util.Objects;
public final class ColumnContext {
   private final String clazz;
   private final String columnName;
   private final String desiredFieldName;
   public ColumnContext(String clazz, String columnName, String desiredFieldName) {
      this.clazz = clazz;
      this.columnName = columnName;
      this.desiredFieldName = desiredFieldName;
   }
   public String getClazz() {
      return clazz;
   }
   public String getColumnName() {
      return columnName;
   }
   public String getDesiredFieldName() {
      return desiredFieldName;
   }
   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      ColumnContext that = (ColumnContext) o;
      return Objects.equals(getClazz(), that.getClazz()) && Objects.equals(getColumnName(),
that.getColumnName()) && Objects.equals(getDesiredFieldName(), that.getDesiredFieldName());
   }
   @Override
   public int hashCode() {
      return Objects.hash(getClazz(), getColumnName(), getDesiredFieldName());
   }
}
```

AnnotationProcessorContext.java

```
package activejdbc.wrapper.annotation.processor.context;
import activejdbc.wrapper.annotation.processor.ActiveJdbcRequiredPropertyWrapperFactory;
import
activejdbc.wrapper.annotation.processor.adapter.builder.strategy.GetterBuilderStrategyHolder;
import
activejdbc.wrapper.annotation.processor.adapter.builder.strategy.SetterBuilderStrategyHolder;
import activejdbc.wrapper.annotation.processor.adapter.builder.strategy.StrategyHolder;
import activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter.*;
import activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter.*;
import java.math.BigDecimal;
import java.sql.Clob;
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.HashMap;
import java.util.Map;
public class AnnotationProcessorContext {
   private final StrategyHolder<SetterBuilderStrategy> setterBuilderStrategyHolder;
   private final StrategyHolder<GetterBuilderStrategy> getterBuilderStrategyHolder;
   private final String wrapperSuffix;
   private final String builderMethodPrefix;
   public AnnotationProcessorContext(String wrapperSuffix, String builderMethodPrefix) {
      this.wrapperSuffix = wrapperSuffix;
      this.getterBuilderStrategyHolder = initGetterBuilderStrategyHolder();
      this.setterBuilderStrategyHolder = initSetterBuilderStrategyHolder();
      this.builderMethodPrefix = builderMethodPrefix;
   }
   private GetterBuilderStrategyHolder initGetterBuilderStrategyHolder() {
      Map<Class<?>, GetterBuilderStrategy> getterBuilderStrategies = new HashMap<>();
      getterBuilderStrategies.put(BigDecimal.class, new BigDecimalGetterBuilderStrategy());
      getterBuilderStrategies.put(Boolean.class, new BooleanGetterBuilderStrategy());
      getterBuilderStrategies.put(Clob.class, new ClobGetterBuilderStrategy());
      getterBuilderStrategies.put(Date.class, new DateGetterBuilderStrategy());
      getterBuilderStrategies.put(Double.class, new DoubleGetterBuilderStrategy());
      getterBuilderStrategies.put(Float.class, new FloatGetterBuilderStrategy());
```

```
            getterBuilderStrategies.put(Integer.class, new IntegerGetterBuilderStrategy());
            getterBuilderStrategies.put(LocalDate.class, new LocalDateGetterBuilderStrategy());
            getterBuilderStrategies.put(LocalDateTime.class, new
LocalDateTimeGetterBuilderStrategy());
            getterBuilderStrategies.put(LocalTime.class, new LocalTimeGetterBuilderStrategy());
            getterBuilderStrategies.put(Long.class, new LongGetterBuilderStrategy());
            getterBuilderStrategies.put(Short.class, new ShortGetterBuilderStrategy());
            getterBuilderStrategies.put(String.class, new StringGetterBuilderStrategy());
            getterBuilderStrategies.put(Time.class, new TimeGetterBuilderStrategy());
            getterBuilderStrategies.put(Timestamp.class, new TimestampGetterBuilderStrategy());
            return new GetterBuilderStrategyHolder(new DefaultGetterBuilderStrategy(),
getterBuilderStrategies);
        }
        private SetterBuilderStrategyHolder initSetterBuilderStrategyHolder() {
            Map<Class<?>, SetterBuilderStrategy> setterBuilderStrategies = new HashMap<>();
            setterBuilderStrategies.put(Date.class, new DateSetterBuilderStrategy());
            setterBuilderStrategies.put(LocalDate.class, new DateSetterBuilderStrategy());
            setterBuilderStrategies.put(LocalDateTime.class, new
LocalDateTimeSetterBuilderStrategy());
            setterBuilderStrategies.put(LocalTime.class, new LocalTimeSetterBuilderStrategy());
            setterBuilderStrategies.put(Timestamp.class, new TimestampSetterBuilderStrategy());
            return new SetterBuilderStrategyHolder(new DefaultSetterBuilderStrategy(),
setterBuilderStrategies);
        }
        public ActiveJdbcRequiredPropertyWrapperFactory init() {
            return new ActiveJdbcRequiredPropertyWrapperFactory(this);
        }
        public StrategyHolder<SetterBuilderStrategy> getSetterBuilderStrategyHolder() {
            return setterBuilderStrategyHolder;
        }
        public StrategyHolder<GetterBuilderStrategy> getGetterBuilderStrategyHolder() {
            return getterBuilderStrategyHolder;
        }
        public String getWrapperSuffix() {
            return wrapperSuffix;
        }
        public String getBuilderMethodPrefix() {
            return builderMethodPrefix;
```

```
    }
}
```

WrapperClassBuilder.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder;
import activejdbc.wrapper.annotation.processor.adapter.builder.strategy.StrategyHolder;
import
activejdbc.wrapper.annotation.processor.adapter.builder.strategy.WrapperBuilderClassBuilder;
import
activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter.GetterBuilderStrategy;
import
activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter.SetterBuilderStrategy;
import activejdbc.wrapper.annotation.processor.context.AnnotationProcessorContext;
import activejdbc.wrapper.annotation.processor.context.ColumnContext;
import activejdbc.wrapper.annotation.processor.util.StringUtils;
import java.util.*;
import static activejdbc.wrapper.annotation.processor.util.StringTemplates.*;
public class WrapperClassBuilder {
   /**
    * 1. package name
    * 2. import activejbdc class package
    * 3. activejbdc object class for import
    * 4. wrapper class name
    * 5. activejdbc object class
    * 6  activejdbc object class
    * 7. activejdbc object name
    * 8. constructors
    * 9. methods
    * 10. builder method and builder class
    */
   public static final String CLASS_TEMPLATE = "package %s;%n" +
         "import %s.%s;%n" +
         "public class %s extends activejdbc.wrapper.ActivejdbcWrapper<%s>{%n" +
         "private %s %s;%n" +
         "%s%n" +
         "%s%n" +
```

```
                "%s%n" +
                "}";
        private final StrategyHolder<GetterBuilderStrategy> getterBuilderStrategyHolder;
        private final StrategyHolder<SetterBuilderStrategy> setterBuilderStrategyHolder;
        private final String packageName;
        private final String activejdbcObjectClassName;
        private final String wrapperClassName;
        private final String activejdbcObjectName;
        private final Set<String> settersBody = new HashSet<>();
        private final Set<String> gettersBody = new HashSet<>();
        private final List<ColumnContext> columnContexts;
        private final Map<String, String> propertyNamesAndGetters = new HashMap<>();
        private final WrapperBuilderClassBuilder wrapperBuilderClassBuilder;
        private String hashCode = StringUtils.EMPTY_STRING;
        private String equals = StringUtils.EMPTY_STRING;
        private String toString = StringUtils.EMPTY_STRING;
        private String getObject = StringUtils.EMPTY_STRING;
        private String builderAndBuilderClass = StringUtils.EMPTY_STRING;
        public WrapperClassBuilder(String packageName, String activejdbcObjectClassName,
List<ColumnContext> columnContexts, AnnotationProcessorContext annotationProcessorContext) {
            getterBuilderStrategyHolder = annotationProcessorContext.getGetterBuilderStrategyHolder();
            setterBuilderStrategyHolder = annotationProcessorContext.getSetterBuilderStrategyHolder();
            this.packageName = packageName;
            this.activejdbcObjectClassName = activejdbcObjectClassName;
            this.wrapperClassName = activejdbcObjectClassName +
annotationProcessorContext.getWrapperSuffix();
            this.activejdbcObjectName =
StringUtils.lowerCaseFirstCharacter(activejdbcObjectClassName);
            this.columnContexts = columnContexts;
            this.wrapperBuilderClassBuilder = new WrapperBuilderClassBuilder(wrapperClassName,
annotationProcessorContext, columnContexts);
        }
        public String buildClassBody() {
            StringBuilder methods = new StringBuilder();
            gettersBody.forEach(methods::append);
            settersBody.forEach(methods::append);
            methods.append(getObject)
                .append(toString)
```

```
                .append(equals)
                .append(hashCode);
        String constructorWithoutParameters =
String.format(CONSTRUCTOR_WITHOUT_PARAMETERS_TEMPLATE, wrapperClassName,
activejdbcObjectName, activejdbcObjectClassName);
        String constructorWithParameter =
String.format(CONSTRUCTOR_WITH_PARAMETER_TEMPLATE, wrapperClassName,
activejdbcObjectClassName, activejdbcObjectName, activejdbcObjectName, activejdbcObjectName);
        String constructors = constructorWithoutParameters + constructorWithParameter;
        return String.format(CLASS_TEMPLATE, packageName, packageName,
activejdbcObjectClassName,
                wrapperClassName, activejdbcObjectClassName, activejdbcObjectClassName,
activejdbcObjectName, constructors, methods, builderAndBuilderClass);
    }
    public WrapperClassBuilder withHashCode() {
        StringJoiner stringJoiner = new StringJoiner(", ");
        for (String getter : propertyNamesAndGetters.values()) {
            stringJoiner.add(String.format("this.%s()", getter));
        }
        hashCode = String.format(HASH_CODE_METHOD_TEMPLATE, stringJoiner);
        return this;
    }
    public WrapperClassBuilder withEquals() {
        StringJoiner stringJoiner = new StringJoiner(" && ");
        for (String getter : propertyNamesAndGetters.values()) {
            stringJoiner.add(String.format("%njava.util.Objects.equals(this.%s(), that.%s())", getter,
getter));
        }
        equals = String.format(EQUALS_METHOD_TEMPLATE, wrapperClassName,
wrapperClassName, stringJoiner);
        return this;
    }
    public WrapperClassBuilder withToString() {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("")
                .append('{');
        StringJoiner stringJoiner = new StringJoiner(" + \", ");
        propertyNamesAndGetters.forEach((propertyName, getter) ->
```

```
                    stringJoiner.add(String.format("%s = \" + (this.%s() == null ? null : \"\"\" + this.%s() +
\"\"\")%n", propertyName, getter, getter)));
                stringBuilder.append(stringJoiner);
                stringBuilder.append(" + \"}\"");
                toString = String.format(TO_STRING_METHOD_TEMPLATE, stringBuilder);
                return this;
            }
            public WrapperClassBuilder withGetters() {
                this.columnContexts.forEach(columnContext -> {
                    String propertyName = StringUtils.isBlank(columnContext.getDesiredFieldName())
                            ?
StringUtils.buildPropertyNameFromColumnName(columnContext.getColumnName())
                            : columnContext.getDesiredFieldName();
                    String methodName = StringUtils.buildMethodName(propertyName, "get");
                    GetterBuilderStrategy strategy =
getterBuilderStrategyHolder.getStrategy(columnContext.getClazz());
                    gettersBody.add(strategy.buildGetterBody(columnContext, activejdbcObjectName));
                    propertyNamesAndGetters.put(propertyName, methodName);
                });
                return this;
            }
            public WrapperClassBuilder withSetters() {
                this.columnContexts.forEach(columnContext -> {
                    SetterBuilderStrategy strategy =
setterBuilderStrategyHolder.getStrategy(columnContext.getClazz());
                    settersBody.add(strategy.buildSetterBody(columnContext, activejdbcObjectName));
                });
                return this;
            }
            public WrapperClassBuilder withMethodGetActivejdbcObject() {
                getObject = String.format(METHOD_GET_OBJECT_TEMPLATE,
activejdbcObjectClassName, activejdbcObjectName);
                return this;
            }
            public WrapperClassBuilder withBuilder() {
                StringBuilder stringBuilder = new StringBuilder();
                stringBuilder.append(wrapperBuilderClassBuilder.buildBuildMethodName());
                stringBuilder.append(wrapperBuilderClassBuilder.buildClassBody());
```

```
            builderAndBuilderClass = stringBuilder.toString();
            return this;
        }
    }
```

WrapperBuilderClassBuilder.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy;
import activejdbc.wrapper.annotation.processor.context.AnnotationProcessorContext;
import activejdbc.wrapper.annotation.processor.context.ColumnContext;
import activejdbc.wrapper.annotation.processor.util.StringUtils;
import java.util.List;
import static
activejdbc.wrapper.annotation.processor.util.StringTemplates.BUILDER_METHOD_TEMPLATE;
import static
activejdbc.wrapper.annotation.processor.util.StringTemplates.BUILDER_SETTER_TEMPLATE;
public class WrapperBuilderClassBuilder {
    /**
     * 1. Builder class name
     * 2. Wrapper class name
     * 3. Wrapper object name
     * 4. Wrapper class name
     * 5. Builder setters
     * 6. Wrapper class name
     * 7. Wrapper object name
     */
    public static final String CLASS_TEMPLATE = " public static class %s {%n" +
            "private final %s %s = new %s();%n" +
            "%s%n" +
            "public %s build() {%n" +
            "return %s;%n" +
            "}%n" +
            "}";
    private final String wrapperClassName;
    private final String builderClassName;
    private final String methodPrefix;
    private final List<ColumnContext> columnContexts;
```

```
        public WrapperBuilderClassBuilder(String wrapperClassName, AnnotationProcessorContext
annotationProcessorContext, List<ColumnContext> columnContexts) {
            this.wrapperClassName = wrapperClassName;
            this.builderClassName = wrapperClassName + "Builder";
            this.columnContexts = columnContexts;
            this.methodPrefix = annotationProcessorContext.getBuilderMethodPrefix();
        }
        public String buildBuildMethodName() {
            return String.format(BUILDER_METHOD_TEMPLATE, builderClassName,
builderClassName);
        }
        public String buildClassBody() {
            StringBuilder stringBuilder = new StringBuilder();
            String wrapperObjectName = StringUtils.lowerCaseFirstCharacter(wrapperClassName);
            columnContexts.forEach(columnContext -> {
                String propertyName = StringUtils.isBlank(columnContext.getDesiredFieldName())
                        ?
StringUtils.buildPropertyNameFromColumnName(columnContext.getColumnName())
                        : columnContext.getDesiredFieldName();
                String withMethodName = StringUtils.buildMethodName(propertyName, methodPrefix);
                // todo refactor this thing
                String setMethodName = StringUtils.buildMethodName(propertyName, "set");
                stringBuilder.append(String.format(BUILDER_SETTER_TEMPLATE, builderClassName,
withMethodName, columnContext.getClazz(), propertyName, wrapperObjectName, setMethodName,
propertyName));
            });
            return String.format(CLASS_TEMPLATE, builderClassName, wrapperClassName,
wrapperObjectName, wrapperClassName, stringBuilder, wrapperClassName, wrapperObjectName);
        }
    }
```

StrategyHolder.java

```
    package activejdbc.wrapper.annotation.processor.adapter.builder.strategy;
    public interface StrategyHolder<T> {
        T getStrategy(String type);
    }
```

SetterBuilderStrategyHolder.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy;
import
activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter.SetterBuilderStrategy;
import java.util.Map;
public class SetterBuilderStrategyHolder implements StrategyHolder<SetterBuilderStrategy> {
   private final SetterBuilderStrategy defaultStrategy;
   private final Map<Class<?>, SetterBuilderStrategy> setterBuilderStrategies;
   public SetterBuilderStrategyHolder(SetterBuilderStrategy defaultStrategy, Map<Class<?>,
SetterBuilderStrategy> setterBuilderStrategies) {
      this.defaultStrategy = defaultStrategy;
      this.setterBuilderStrategies = setterBuilderStrategies;
   }
   @Override
   public SetterBuilderStrategy getStrategy(String type) {
      return setterBuilderStrategies.getOrDefault(getClass(type), defaultStrategy);
   }
   private Class<?> getClass(String type) {
      try {
         return Class.forName(type);
      } catch (ClassNotFoundException e) {
         throw new IllegalArgumentException("Unknown type " + type, e);
      }
   }
}
```

GetterBuilderStrategyHolder.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy;
import
activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter.GetterBuilderStrategy;
import java.util.Map;
public class GetterBuilderStrategyHolder implements StrategyHolder<GetterBuilderStrategy>{
   private final GetterBuilderStrategy defaultStrategy;
```

```
    private final Map<Class<?>, GetterBuilderStrategy> getterBuilderStrategies;
    public GetterBuilderStrategyHolder(GetterBuilderStrategy defaultStrategy, Map<Class<?>,
GetterBuilderStrategy> getterBuilderStrategies) {
      this.defaultStrategy = defaultStrategy;
      this.getterBuilderStrategies = getterBuilderStrategies;
    }
    @Override
    public GetterBuilderStrategy getStrategy(String type) {
      return getterBuilderStrategies.getOrDefault(getClass(type), defaultStrategy);
    }
    private Class<?> getClass(String type) {
      try {
        return Class.forName(type);
      } catch (ClassNotFoundException e) {
        throw new IllegalArgumentException("Unknown type " + type, e);
      }
    }
  }
```

SetterBuilderStrategy.java

```
  package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter;
  import activejdbc.wrapper.annotation.processor.context.ColumnContext;
  import activejdbc.wrapper.annotation.processor.util.StringUtils;
  import java.util.Set;
  public interface SetterBuilderStrategy {
    default String buildSetterBody(ColumnContext columnContext, String activejdbcObjectName) {
      String propertyName = StringUtils.isBlank(columnContext.getDesiredFieldName())
          ? StringUtils.buildPropertyNameFromColumnName(columnContext.getColumnName())
          : columnContext.getDesiredFieldName();
      String methodName = StringUtils.buildMethodName(propertyName, "set");
      return String.format(getTemplate(), methodName, columnContext.getClazz(), propertyName,
activejdbcObjectName, columnContext.getColumnName(), propertyName);
    }
    String getTemplate();
    Set<Class<?>> typesToApply();
  }
```

TimestampSetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter;
import java.sql.Timestamp;
import java.util.Collections;
import java.util.Set;
public class TimestampSetterBuilderStrategy implements SetterBuilderStrategy {
   /**
    * 1. setters name
    * 2. setters type
    * 3. property name
    * 4. activejdbc object
    * 5. column name
    * 6. property name
    */
   public static final String SETTER_TEMPLATE = "public void %s(%s %s) {%n" +
         "%s.setTimestamp(\"%s\", %s);%n" +
         "}%n";
   @Override
   public String getTemplate() {
      return SETTER_TEMPLATE;
   }
   @Override
   public Set<Class<?>> typesToApply() {
      return Collections.singleton(Timestamp.class);
   }
}
```

LocalTimeSetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter;
import java.time.LocalTime;
import java.util.Collections;
import java.util.Set;
public class LocalTimeSetterBuilderStrategy implements SetterBuilderStrategy {
```

```
    /**
     * 1. setters name
     * 2. setters type
     * 3. property name
     * 4. activejdbc object
     * 5. column name
     * 6. property name
     */
    public static final String SETTER_TEMPLATE = "public void %s(%s %s) {%n" +
            "%s.setTime(\"%s\", java.util.Optional.ofNullable(%s)%n" +
            ".map(java.sql.Time::valueOf)" +
            ".orElse(null));%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return SETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(LocalTime.class);
    }
}
```

LocalDateTimeSetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter;
import java.time.LocalDateTime;
import java.util.Collections;
import java.util.Set;
public class LocalDateTimeSetterBuilderStrategy implements SetterBuilderStrategy {
    /**
     * 1. setters name
     * 2. setters type
     * 3. property name
     * 4. activejdbc object
     * 5. column name
     * 6. property name
```

```
     */
    public static final String SETTER_TEMPLATE = "public void %s(%s %s) {%n" +
            "%s.setTimestamp(\"%s\", java.util.Optional.ofNullable(%s)%n" +
            ".map(java.sql.Timestamp::valueOf)%n" +
            ".orElse(null));%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return SETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(LocalDateTime.class);
    }
}
```

DefaultSetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter;
import java.util.Collections;
import java.util.Set;
public class DefaultSetterBuilderStrategy implements SetterBuilderStrategy {
    /**
     * 1. setters name
     * 2. setters type
     * 3. property name
     * 4. activejdbc object
     * 5. column name
     * 6. property name
     */
    public static final String SETTER_TEMPLATE = "public void %s(%s %s) {%n" +
            "%s.set(\"%s\", %s);%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return SETTER_TEMPLATE;
    }
```

```
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.emptySet();
    }
}
```

DateSetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.setter;
import java.sql.Date;
import java.time.LocalDate;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
public class DateSetterBuilderStrategy implements SetterBuilderStrategy {
    /**
     * 1. setters name
     * 2. setters type
     * 3. property name
     * 4. activejdbc object
     * 5. column name
     * 6. property name
     */
    public static final String SETTER_TEMPLATE = "public void %s(%s %s) {%n" +
            "%s.setDate(\"%s\", %s);%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return SETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return new HashSet<>(Arrays.asList(LocalDate.class, Date.class));
    }
}
```

GetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import activejdbc.wrapper.annotation.processor.context.ColumnContext;
import activejdbc.wrapper.annotation.processor.util.StringUtils;
import java.util.Set;
public interface GetterBuilderStrategy {
    default String buildGetterBody(ColumnContext columnContext, String activejdbcObjectName)
{
        String propertyName = StringUtils.isBlank(columnContext.getDesiredFieldName())
                ? StringUtils.buildPropertyNameFromColumnName(columnContext.getColumnName())
                : columnContext.getDesiredFieldName();
        String methodName = StringUtils.buildMethodName(propertyName, "get");
        return String.format(getTemplate(), columnContext.getClazz(), methodName,
activejdbcObjectName, columnContext.getColumnName());
    }
    String getTemplate();
    Set<Class<?>> typesToApply();
}
```

TimestampGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.sql.Timestamp;
import java.util.Collections;
import java.util.Set;
public class TimestampGetterBuilderStrategy implements GetterBuilderStrategy {
    /**
     * 1. return type
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return %s.getTimestamp(\"%s\");%n" +
            "}%n";
    @Override
    public String getTemplate() {
```

```
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(Timestamp.class);
    }
}
```

TimeGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.sql.Time;
import java.util.Collections;
import java.util.Set;
public class TimeGetterBuilderStrategy implements GetterBuilderStrategy {
    /**
     * 1. return type
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return java.util.Optional.ofNullable(%s.getDate(\"%s\"))%n" +
            ".map(java.sql.Date::getTime)%n" +
            ".map(java.sql.Time::new)%n" +
            ".orElse(null);%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(Time.class);
    }
}
```

StringGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.util.Collections;
import java.util.Set;
public class StringGetterBuilderStrategy implements GetterBuilderStrategy {
   /**
    * 1. return type
    * 2. method name
    * 3. activejdbc object
    * 4. column name
    */
   private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
         "return %s.getString(\"%s\");%n" +
         "}%n";
   @Override
   public String getTemplate() {
      return GETTER_TEMPLATE;
   }
   @Override
   public Set<Class<?>> typesToApply() {
      return Collections.singleton(String.class);
   }
}
```

ShortGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.util.Collections;
import java.util.Set;
public class ShortGetterBuilderStrategy implements GetterBuilderStrategy {
   /**
    * 1. return type
    * 2. method name
    * 3. activejdbc object
    * 4. column name
    */
```

```
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return %s.getShort(\"%s\");%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(Short.class);
    }
}
```

LongGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.util.Collections;
import java.util.Set;
public class LongGetterBuilderStrategy implements GetterBuilderStrategy {
    /**
     * 1. return type
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return %s.getLong(\"%s\");%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(Long.class);
    }
}
```

LocalTimeGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.time.LocalTime;
import java.util.Collections;
import java.util.Set;
public class LocalTimeGetterBuilderStrategy implements GetterBuilderStrategy {
   /**
    * 1. return type
    * 2. method name
    * 3. activejdbc object
    * 4. column name
    */
   private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
         "return java.util.Optional.ofNullable(%s.getTime(\"%s\"))%n" +
         ".map(java.sql.Time::toLocalTime)%n" +
         ".orElse(null);%n" +
         "}%n";
   @Override
   public String getTemplate() {
      return GETTER_TEMPLATE;
   }
   @Override
   public Set<Class<?>> typesToApply() {
      return Collections.singleton(LocalTime.class);
   }
}
```

LocalDateTimeGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.time.LocalDateTime;
import java.util.Collections;
import java.util.Set;
public class LocalDateTimeGetterBuilderStrategy implements GetterBuilderStrategy {
   /**
    * 1. return type
```

```
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return java.util.Optional.ofNullable(%s.getTimestamp(\"%s\"))%n" +
            ".map(java.sql.Timestamp::toLocalDateTime)%n" +
            ".orElse(null);%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(LocalDateTime.class);
    }
}
```

LocalDateGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.time.LocalDate;
import java.util.Collections;
import java.util.Set;
public class LocalDateGetterBuilderStrategy implements GetterBuilderStrategy {
    /**
     * 1. return type
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return java.util.Optional.ofNullable(%s.getDate(\"%s\"))%n" +
            ".map(java.sql.Date::toLocalDate)%n" +
            ".orElse(null);%n" +
            "}%n";
```

```
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(LocalDate.class);
    }
}
```

IntegerGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.util.Collections;
import java.util.Set;
public class IntegerGetterBuilderStrategy implements GetterBuilderStrategy {
    /**
     * 1. return type
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return %s.getInteger(\"%s\");%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(Integer.class);
    }
}
```

FloatGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.util.Collections;
import java.util.Set;
public class FloatGetterBuilderStrategy implements GetterBuilderStrategy {
    /**
     * 1. return type
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return %s.getFloat(\"%s\");%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(Float.class);
    }
}
```

DoubleGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.util.Collections;
import java.util.Set;
public class DoubleGetterBuilderStrategy implements GetterBuilderStrategy{
    /**
     * 1. return type
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
```

```
            "return %s.getDouble(\"%s\");%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(Double.class);
    }
}
```

DefaultGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import activejdbc.wrapper.annotation.processor.context.ColumnContext;
import activejdbc.wrapper.annotation.processor.util.StringUtils;
import java.util.Collections;
import java.util.Set;
public class DefaultGetterBuilderStrategy implements GetterBuilderStrategy {
    /**
     * 1. return type
     * 2. method name
     * 3. class for casting
     * 4. activejdbc object
     * 5. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return (%s) %s.get(\"%s\");%n" +
            "}%n";
    @Override
    public String buildGetterBody(ColumnContext columnContext, String activejdbcObjectName) {
        String propertyName = StringUtils.isBlank(columnContext.getDesiredFieldName())
                ? StringUtils.buildPropertyNameFromColumnName(columnContext.getColumnName())
                : columnContext.getDesiredFieldName();
        String methodName = StringUtils.buildMethodName(propertyName, "get");
```

```
            return String.format(getTemplate(), columnContext.getClazz(), methodName,
columnContext.getClazz(), activejdbcObjectName, columnContext.getColumnName());
        }
        @Override
        public String getTemplate() {
            return GETTER_TEMPLATE;
        }
        @Override
        public Set<Class<?>> typesToApply() {
            return Collections.emptySet();
        }
    }
```

DateGetterBuilderStrategy.java

```
    package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
    import java.sql.Date;
    import java.util.Collections;
    import java.util.Set;
    public class DateGetterBuilderStrategy implements GetterBuilderStrategy {
        /**
         * 1. return type
         * 2. method name
         * 3. activejdbc object
         * 4. column name
         */
        private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
                "return %s.getDate(\"%s\");%n" +
                "}%n";
        @Override
        public String getTemplate() {
            return GETTER_TEMPLATE;
        }
        @Override
        public Set<Class<?>> typesToApply() {
            return Collections.singleton(Date.class);
        }
```

```
}
```

ClobGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.sql.Clob;
import java.util.Collections;
import java.util.Set;
public class ClobGetterBuilderStrategy implements GetterBuilderStrategy {
   /**
    * 1. return type
    * 2. method name
    * 3. activejdbc object
    * 4. column name
    */
   private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
         "return %s.getClob(\"%s\");%n" +
         "}%n";
   @Override
   public String getTemplate() {
      return GETTER_TEMPLATE;
   }
   @Override
   public Set<Class<?>> typesToApply() {
      return Collections.singleton(Clob.class);
   }
}
```

BooleanGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.util.Collections;
import java.util.Set;
public class BooleanGetterBuilderStrategy implements GetterBuilderStrategy{
   /**
    * 1. return type
```

```
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return %s.getBoolean(\"%s\");%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(Boolean.class);
    }
}
```

BigDecimalGetterBuilderStrategy.java

```
package activejdbc.wrapper.annotation.processor.adapter.builder.strategy.getter;
import java.math.BigDecimal;
import java.util.Collections;
import java.util.Set;
public class BigDecimalGetterBuilderStrategy implements GetterBuilderStrategy {
    /**
     * 1. return type
     * 2. method name
     * 3. activejdbc object
     * 4. column name
     */
    private static final String GETTER_TEMPLATE = "public %s %s() {%n" +
            "return %s.getBigDecimal(\"%s\");%n" +
            "}%n";
    @Override
    public String getTemplate() {
        return GETTER_TEMPLATE;
    }
```

```
    @Override
    public Set<Class<?>> typesToApply() {
        return Collections.singleton(BigDecimal.class);
    }
}
```

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>io.github.erioh</groupId>
    <version>1.0.5-SNAPSHOT</version>
    <artifactId>activejdbc-wrapper</artifactId>
    <packaging>jar</packaging>
    <name>ActiveJDBC Wrapper</name>
    <description>Should simplify usage of ActiveJDBC by generation new wrapper
classes</description>
    <url>https://github.com/erioh/activejdbc-wrapper-project</url>
    <properties>
        <junit.version>4.13.1</junit.version>
        <assertj-core.version>3.19.0</assertj-core.version>
        <activejdbc.version>2.2</activejdbc.version>
        <junit-dataprovider.version>1.13.1</junit-dataprovider.version>
        <mockito-core.version>3.9.0</mockito-core.version>
        <compile-testing.version>0.19</compile-testing.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.javalite</groupId>
            <artifactId>activejdbc</artifactId>
            <scope>provided</scope>
            <version>${activejdbc.version}</version>
        </dependency>
```

```
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>${assertj-core.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.tngtech.java</groupId>
      <artifactId>junit-dataprovider</artifactId>
      <version>${junit-dataprovider.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>${mockito-core.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.google.testing.compile</groupId>
      <artifactId>compile-testing</artifactId>
      <version>${compile-testing.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <developers>
    <developer>
      <name>Serhii Demenkov</name>
      <email>sergey dot demenkov7 at gmail dot com</email>
      <organization>io.github.erioh</organization>
      <organizationUrl>https://github.com/erioh</organizationUrl>
    </developer>
```

```
    </developers>
    <licenses>
      <license>
        <name>Apache License 2.0</name>
        <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
        <distribution>repo</distribution>
      </license>
    </licenses>
    <scm>
      <connection>scm:git:git@github.com:erioh/activejdbc-wrapper-project.git</connection>
      <developerConnection>scm:git:git@github.com:erioh/activejdbc-wrapper-
project.git</developerConnection>
      <url>git@github.com:erioh/activejdbc-wrapper-project.git</url>
      <tag>HEAD</tag>
    </scm>
    <distributionManagement>
      <snapshotRepository>
        <id>ossrh</id>
        <url>https://s01.oss.sonatype.org/content/repositories/snapshots</url>
      </snapshotRepository>
      <repository>
        <id>ossrh</id>
        <url>https://s01.oss.sonatype.org/service/local/staging/deploy/maven2/</url>
      </repository>
    </distributionManagement>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.1</version>
          <configuration>
            <source>8</source>
            <target>8</target>
          </configuration>
          <executions>
            <execution>
              <id>default-compile</id>
```

```
            <configuration>
                <compilerArgument>-proc:none</compilerArgument>
                <includes>
                    <include>
activejdbc\wrapper\annotation\processor\ActiveJdbcRequiredPropertyProcessor.java
                    </include>
                </includes>
            </configuration>
        </execution>
        <execution>
            <id>compile-project</id>
            <phase>compile</phase>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.sonatype.plugins</groupId>
    <artifactId>nexus-staging-maven-plugin</artifactId>
    <version>1.6.7</version>
    <extensions>true</extensions>
    <configuration>
        <serverId>ossrh</serverId>
        <nexusUrl>https://s01.oss.sonatype.org/</nexusUrl>
        <autoReleaseAfterClose>true</autoReleaseAfterClose>
    </configuration>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.2.1</version>
    <executions>
        <execution>
            <id>attach-sources</id>
            <goals>
                <goal>jar-no-fork</goal>
```

```
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>2.9.1</version>
        <executions>
          <execution>
            <id>attach-javadocs</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-gpg-plugin</artifactId>
        <version>1.5</version>
        <executions>
          <execution>
            <id>sign-artifacts</id>
            <phase>verify</phase>
            <goals>
              <goal>sign</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build></project>
```

ДОДАТОК Б

ВІДГУК КЕРІВНИКА

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ «ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Наукового керівника Мещерякова Леоніда Івановича, д.т.н., проф. каф. ПЗКС
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

на магістерську роботу
студента Деменкова Сергія Олександровича
(прізвище, ім'я, по батькові)

курсу І групи 121м-21з-1
спеціальності 121 Інженерія програмного забезпечення
освітньої програми «121 Інженерія програмного забезпечення»
на тему Розробка програмного забезпечення для автоматизації зміни стану ресурсів тестування

Актуальність теми Розглянута магістерська кваліфікаційна робота присвячена запобіганні появі шаблонного коду при використанні ORM-системи ActiveJDBC. Дане рішення є перспективним у використанні у корпоративних системах завдяки зменшенню обсягу коду, що потрібно буде підтримувати та полегшенню роботи з вищеназваним фреймворком. На сьогоднішній момент сфера розробки, а разом з нею і сфера застосування ActiveJDBC, розвивається швидкими темпами, таким чином магістерська робота відзначається актуальністю.

Мета досліджень Полягає в розробці та оптимізації процесора анотацій для автоматичної генерації обгорток для ActiveJDBC об'єктів, які надаватимуть можливість працювати з ActiveJDBC як з POJO об'єктами, створювати нові об'єкти ActiveJDBC за допомогою породжувального патерна Будівельник, та порівнювати обгорнуті об'єкти ActiveJDBC за допомогою автоматично згенерованих методів equals та hashCode.

Коротка характеристика розділів роботи Перший розділ роботи складається з аналітичного розбору предметної галузі та існуючими інструментами що використовуються для генерації коду взагалі, та для запобігання появі шаблонного коду зокрема. Розглянуто ORM-систему ActiveJDBC, її недоліки та переваги, виявлено її потенційно слабкі місця, та сформульовано вимоги щодо функціональних характеристик для розроблюваного рішення. Другий розділ містить огляд існуючих підходів що до автоматичної генерації коду та обґрунтовано вибір Обробника Анотацій. Третій розділ присвячено тонкощам програмної реалізації застосунку для автоматичної генерації коду.

Практичне значення роботи Отримані результати роботи є актуальними у проектуванні додатків що працюють з реляційними базами даних за використання ORM-систему ActiveJDBC у корпоративних системах. Також надані результати дослідження є підставою для більш поглибленого вивчення проблем, пов'язаних зі зменшенню кількості шаблонного коду.

Зауваження та недоліки В роботі відсутній більш детальний порівняльний детальний аналіз методів генерації коду.

Висновки та оцінка Магістром було проведено порівняльний аналіз існуючих проблем при використанні ORM-систему ActiveJDBC, проведено порівняння методів генерації коду та реалізовано програмний апарат для вирішення досліджених проблем. Під час виконання магістерської кваліфікаційної роботи студент Деменков С.О. постав кваліфікованим та впевненим спеціалістом, який знаходить оптимальні рішення у складних технічних питаннях. Вважаю, що магістерська кваліфікаційна робота заслуговує оцінку «відмінно», а Деменков С.О. – присвоєння кваліфікації «магістра» по спеціальності комп'ютерних наук.

Науковий керівник Мещеряков Л.І., док. техн. наук, проф., проф. каф. ПЗКС

(прізвище, ім'я, по батькові, посада, місце роботи)

«___»_____20__ р. _____

(підпис)

ДОДАТОК В

РЕЦЕНЗІЯ

на кваліфікаційну роботу

студента *Деменкова Сергія Олександровича*

курсу II групи *121м-22з-1*

кафедри програмного забезпечення комп'ютерних систем

Спеціальності *Інженерія програмного забезпечення*

освітньої програми *«Інженерія програмного забезпечення»*

Тема роботи Розробка та дослідження ефективності реалізації автоматичної генерації коду для ORM-системи ActiveJDBC з використанням технологій Annotation Processing

Стисла характеристика розділів роботи Перший розділ містить аналіз предметної галузі та існуючих рішень що до запобігання появи шаблонного коду взагалі, та генерації коду зокрема. Розглянуто ORM-систему ActiveJDBC та її недоліки з точки зору використання. Другий розділ містить порівняльну характеристику підходів що до генерації коду, що використовуються у мові програмування Java та обґрунтовано вибір обробника анотацій. Третій розділ присвячений розробці системи, та опису інтерфейсу користувача.

Пропозиції, внесені студентом, рівень їх наукового обґрунтування В даній кваліфікаційній роботі студентом надано декілька пропозицій щодо вирішення поставлених задач. Кожна з пропозицій була обґрунтована та підкріплена науковими даними.

Практичне значення роботи Практична цінність полягає в розробці та оптимізації обробника анотацій, що полегшить написання та підтримку коду взаємодії з ORM-системою ActiveJDBC

Якість оформлення роботи робота виконана у відповідності до вимог оформлення кваліфікаційних робіт і відповідає поставленій задачі.

Недоліки в роботі В роботі відсутній опис використання розробленого

інструменту за межами засобу автоматизації роботи з програмними проектами «Apache Maven», проте вказаний недолік не впливає на позитивне враження від роботи.

Загальний висновок В процесі виконання магістерської роботи студентом була створена автоматизована система генерації коду для ORM-системи ActiveJDBC , що дійсно має цінність для ведення бізнесу. Робота поділена на три розділи, що відповідають методичним вказівкам. Студента можна вважати готовим до самостійної роботи як спеціаліста. Деменков С.О. заслуговує присвоєння кваліфікації «магістра» по спеціальності комп'ютерних наук

(підготовленість студента до самостійної роботи як спеціаліста)

Оцінка магістерської роботи Вважаю що магістерська робота заслуговує оцінки 90 «відмінно».

Рецензент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

« » 2023р.

(підпис)

ДОДАТОК Г

ПЕРЕЛІК ФАЙЛІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файлу	**Опис**
Пояснювальні документи	
Диплом_Деменков_С.О.121м-22з-1.docx	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
Диплом_Деменков_С.О.121м-22з-1.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF
Програма	
Demenkov.zip	Архів. Містить коди програми і скомпільовану програму
Презентація	
Деменков_121м-21з-1.pptx	Презентація кваліфікаційної роботи