Хмельницький національний університет
Факультет програмування
та комп'ютерних і телекомунікаційних систем
Кафедра комп'ютерної інженерії та системного програмування

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
Освітній рівень

Фреймворк на основі принципів інверсії управління мовою Java
Назва теми

КвРКІ.170137.17.01.05 ПЗ
Шифр

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія»*
Назва

Виконав: студент IV курсу, група КІ–17–1 К.В. Давидюк
Підпис Ініціали, прізвище

Керівник О. В. Бармак
Підпис, дата Ініціали, прізвище

Нормоконтролер С.М.Лисенко
Підпис, дата Ініціали, прізвище

До захисту допускаю:
Зав. кафедри комп'ютерної
Інженерії та системного
Програмування Т.О. Говорущенко
Підпис Ініціали, прізвище

«7» червня 2021 р.

Хмельницький 2021

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Програмування та комп'ютерних і телекомунікаційних систем

Кафедра Комп'ютерної інженерії та системного програмування

Освітній рівень бакалавр

Галузь знань 12 Інформаційні технології

Спеціальність 123Комп'ютерна інженерія

Освітня програма освітня програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Зав. кафедри Т.О.Говорущенко

“ 11” 012021 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Давидюку Костянтину Васильовичу
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи)Фреймворк на основі принципів інверсії управління мовою Java

Керівник проекту (роботи)Бармак О. В., д.т.н., проф.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 05.02.2021 р. № 11

2. Строк подання студентом проекту (роботи) на кафедру 07.06.2021 р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Дослідженняпредметноїобласті та постановка задачі

Проектуванняпрограмногорішення та вибіртехнологій для розробки

Розробкапрограмногорішення. Тестування та аналізможливостей для покращення

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

UMLдіаграма класів

Діаграма послідовностей

Блок-схема роботи ПЗ

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання при
Нормоконтроль	Лисенко С.М., професор кафедри КІСП		
Антиплагіат	Нічепорук А.О., доцент кафедри КІСП		

7. Дата видачі завдання «_11_» ______01______ 2021 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Приміт
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	11.01.2021	виконан
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначенняоб`єкта та предмета дослідження	01.02.2021	виконан
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2021	виконан
4	Робота над розділом 2 –проєктування програмного-технічного засобу	01.04.2021	виконан
5	Робота над розділом 3 – програмно-апаратна реалізація та тестування програмно-технічного засобу	30.04.2021	виконан
6	Оформленняпояснювальної записки згідновимог	31.05.2021	виконан
7	ПопереднійзахистВКР	02.06.2021	виконан
8	ЗахистВКР на засіданні ЕК	Червень 2021 року	

Студент ____________ К. В. Давидюк
Підпис Ініціали, прізвище

Керівник проекту (роботи) ____________ О. В. Бармак
Підпис Ініціали, прізвище

№ рядка	Формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 170137.17.01.05 ПЗ	Пояснювальна записка	81		
			Графічні матеріали			
2		КвРКІ 170137.17.01.05 Е8	UML діаграма класів	1		
3		КвРКІ 170137.17.01.05 Е8	Діаграма послідовностей	1		
4		КвРКІ 170137.17.01.05 Е8	Блок-схеми роботи програми	1		

КвРКІ 170137.17.01.05 ВП

Зм	Арк	№ докум	Підпис	Дата
Розробив		Давидов К. В		
Перевір.		Бармак О. В		
Н. контр.		Лисенко С. М		
Затв.		Говорущенко Т. О		07.06.21

Відомість проекту

Літера	Аркуш	Аркушів
У	1	1

ХНУ, КІ-17-1

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Фреймворк на основі принципів інверсії управління мовою Java».

Автор роботи: Давидюк Костянтин Васильович.

Керівник роботи: Бармак Олександр Володимирович.

Пояснювальна записка:81 с., 10 рис., 5 дод., 50 джерел.

Графічна частина: 7 презентаційних слайдів.

ФРЕЙМВОРК НА ОСНОВІ ПРИНЦИПІВ ІНВЕРСІЇ УПРАВЛІННЯ МОВОЮ JAVA.

Метою роботи є розробка фреймворка за принципами інверсії управління мовою програмування Java.

У цій роботі розробленофреймворк, що реалізує інверсію управління через впровадження залежностей. Розробка велась на мова програмування Java, що є сучасною цілком об’єктно-орієнтованою мовою, для якої проблема інверсії управління є актуальною. Також розроблено тестову програму на основі фреймворка і протестовані основні особливості програмної реалізації фреймворку.

Підпис студента Дата

07.06.2021р.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 5
ВСТУП........ 6
1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ 8
1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей........ 8
1.2 Аналіз наявного програмно-апаратного забезпечення предметної області 11
1.2.1 Огляд Java фреймворка Spring Framework 11
1.2.2 Метадані налаштувань........ 13
1.2.3 Створення контейнеру........ 14
1.2.4 Розділення XML конфігурацій на декілька файлів 16
1.2.5 Використання контейнера інверсії управління........ 17
1.2.6 Впровадження залежностей 18
1.2.7 Кругові залежності........ 20
1.3 Визначення вимог до системи автоматизації та розробка технічного завдання........ 21
1.4 Висновки 21
2 ПРОЄКТУВАННЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ........ 23
2.1 Функційні вимоги........ 23
2.2 Визначення кроків для початку проектування програмного забезпечення 23
2.3 Вибір рішення для автоматизації збирання та управління проектами...... 25
2.4 Вибір версії та вендора мови програмування Java. 28
2.5 Вибір засобів для автоматичної генерації коду. 32
2.6 Вибір бібліотек для роботи з рефлексією........ 34
2.7 Визначення основних компонентів програмного забезпечення. 35

2.7.1 Фабрика для створення об'єктів 35
2.7.2 Конфігуратор об'єктів 36
2.7.3 Метадані компонентів 36
2.7.4 Кеш об'єктів 36
2.7.5 Компонент для зчитування конфігурацій з конфігураційного файлу 38
2.7.6 Компонент пошуку імплементацій класу 38
2.8 Висновки 39
3 ПРОГРАМНО-АПАРАТНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ 41
3.1 Реалізовані функціональні особливості 41
3.1.1 Концепція Singleton 41
3.1.2 Пост конструктор класу 43
3.1.3 Налаштування об'єктів згідно конфігураційних файлів 45
3.1.4 Підтримка проксі об'єктів 45
3.2 Реалізація програмних компонентів 46
3.2.1 Фабрика для створення об'єктів 46
3.2.2 Конфігуратори об'єктів 46
3.2.3 Компонент пошуку імплементації класу 48
3.2.4 ApplicationContext 48
3.2.5 Компонент для запуску роботи контейнеру інверсії управління ... 50
3.3 Розробка програмного забезпечення на базі реалізованого рішення для інверсії управління 51
3.3.1 Опис тестової програми 51
3.3.2 Результати роботи тестової програми 53
3.4 Порівняння коду з та без використанням контейнеру для інверсії управління. 55
3.5 Способи покращення програмної реалізації 57
3.6 Висновки 58
ВИСНОВКИ 60

Додаток А Лістинг програмного код фреймворку …..……………………………..66
Додаток Б Лістинг коду програми розробленої на базі фреймворку ……………..76
Додаток В Копія креслення «UML діаграма класів»………………………………79
Додаток Г Копія креслення «Діаграма послідовностей»…………………………...80
Додаток Д Копія креслення «Блок-схеми роботи програми»…………………...…81

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ПЗ - програмне забезпечення

БД - база даних

IoC – Inversion of Control(інверсія контролю)

DI – Dependency Injection(впровадження залежностей)

JDK – Java Development Kit

JSF – JavaServer Faces

ВСТУП

Об’єктно-орієнтовані мови програмування впевнено осіли в лідерах за кількістю проектів, що їх використовують. Окрім підтримки існуючих проектів, їх все частіше використовують для написання нових. Це і не дивно, оскільки ООП дає багато тих можливостей, що не можуть дати процедурні мови програмування.

Разом з тим, в світі об’єктно-орієнтованого програмування існують проблеми, що не виникають при використанні процедурних мов. Так, при розробці програмного забезпечення з використанням ООП, розробники постійно стикаються з створенням об’єктів певних класів. Створити об’єкт досить просто, однак важливою також є можливість легко змінювати код в будь-який момент. Для цього розробники намагаються використовувати абстракції при написанні коду, щоб не залежати від кінцевої реалізації певного класу чи інтерфейсу. Це дозволяє не тільки повторно використовувати один і той же код, а і легко змінювати його за необхідності. Зручність полягає в тому, що для зміни реалізації певного класу необхідно лише замінити об’єкт, що створюється на той що необхідно використовувати.

Але навіть з використанням абстракцій, розробка і підтримка програмного забезпечення, що налічує в собі десятки тисяч класів, стає не легким завданням. В подібних системах часто стається так, що код настільки прив’язаний до певних реалізацій, що зміни хоча б одного з класів, тягне за собою зміни в інших і запускає ланцюгову реакцію. До того, ж необхідно не забувати про те, що код повинен мати юніт тести, і їх теж необхідно тримати в актуальному стані. Так навіть невелика зміна в код проекту може стати викликом для розробника.

Як наслідок, розробники все частіше почали винаходити та застосовувати шаблони проектування, що вирішують проблему створення об’єктів. Так, досить популярним шаблоном проектування є «Фабрика». Він передбачає існування певного класу, що відповідає за створення об’єктів в програмі і до якого звертаються інші класи, коли їм необхідний певний об’єкт. Однак, хоч цей шаблон і вирішує певні проблеми, він не є ідеальним рішенням. Так при використанні шаблону «Фабрика» розробникам складніше писати юніт тести

тому, що в тестах не завжди є можливість використовувати справжні реалізації класів, оскільки це може спричинити порушення роботи оригінальної системи, чи навіть втрату даних.

Тому сьогодні популярним рішенням є використання шаблону інверсії управління через впровадження залежностей. Особливістю цього шаблону є те, що відповідальність за створення і управління об’єктами лягає на стороннє програмне забезпечення. Сторону відповідальну за інверсію управління прийнято називати контейнером інверсії управління.

Інверсія управління може застосовуватись в будь-якій об’єктно-орієнтованій мові програмування, і для різних задач, в тому числі в системному програмуванні.

Сьогодні, майже для кожної об’єктно-орієнтованої мови програмування існують готові рішення для інверсії управління. Для середовища Java одним з найпопулярніших таких рішень є Spring Framework.

Метою роботи є дослідження теми інверсії управління та реалізація програмного забезпечення, що надає можливість використання інверсії управління на мові програмування Java.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей

Інверсія контролю – це шаблон проектування в об'єктно-орієнтованому програмуванні, суть якого полягає в передачі управління залежностями об'єкта іншій стороні, що називають контейнером для забезпечення інверсії управління(Inversion of Control container) [15, 27, 28, 30].

Об'єктно-орієнтоване програмування – це парадигма програмування, що описує програмне забезпечення як множину об'єктів класів, що можуть взаємодіяти між собою. Основу ООП складають три основні концепції: інкапсуляція, успадкування, поліморфізм. До переваг ООП можна віднести можливість підвищення модульності програмного забезпечення (сотні функцій процедурної мови, в ООП можна замінити кількома класами зі своїми методами). Ця парадигма з'явилась в 60-х роках, проте вона не мала широкого застосування до 90-х, коли розвиток комп'ютерів та комп'ютерних систем дав змогу писати досить об'ємне та складне програмне забезпечення. Це змусило переглянути популярні тоді підходи до написання програм. Сьогодні багато мов програмування, або підтримують ООП (PHP, Lua), або ж є повністю об'єктно-орієнтованими (зокрема, Java, C#, C++).

Об'єктно-орієнтоване програмування вперше було реалізовано в в 60-х роках в мові програмування Симула[9], саме тоді посилились дискусії про кризу програмного забезпечення. Через різке збільшення розмірів програмного забезпечення та його складності, було дуже важко підтримувати якість коду програм. Об'єктно-орієнтоване програмування частково вирішує цю проблему шляхом розділення програми на модулі(класи), що містять специфічну логіку для конкретного об'єкта. На основі класу можна створити об'єкт, через який може мати певний стан і методи, що реалізують певну логіку.

Кожен об'єкт - це такий собі незалежний юніт зі своїм призначенням та відповідальністю. ООП надає можливість створювати класи, що описують об'єкти з реального життя, такі як рахунок в банку, чи книга в бібліотеці.

Наслідування це принцип ООП, що дає змогу змінювати, або розширювати поведінку існуючого класу та його атрибути. Так Собака може наслідувати клас Тварина. Це означає, що клас наслідник буде мати свою специфічну для цієї імплементації поведінку, але при цьому працювати згідно контракту батьківського класу.

Наступним принципом об'єктно-орієнтованого програмування є інкапсуляція. Інкапсуляція це приховування частини логіки класу від того хто використовує об'єкти цього класу, або наслідує цю логіку. Так більшість мов програмування має принаймні три модифікатори доступу, що можна застосовувати до полів, методів та до самих класів. Це public, protected та private, вони вказують, що поле, або метод буде доступний всім класам, наслідникам, чи лише всередині класу відповідно. Для надання доступу до приватного поля прийнято використовувати так звані гетери(getters) та сетери(setters). Це публічні методи, що повертають значення поля класу, або встановлюють його, відповідно.

Однією з концепцій ООП є поліморфізм. Поліморфізм означає можливість мати різні імплементації одного класу чи інтерфейсу. За рахунок цього в розробників є можливість писати код програми використовуючи абстракції і не вдаватись до деталей реалізації, адже реалізація повинна працювати за певним контрактом, що передбачений абстрактним класом чи інтерфейсом.

В великому програмному забезпеченні, необхідність змінити імплементацію інтерфейсу, що використовує той чи інший об'єкт може стати нетривіальною задачею і потребувати великих змін в існуючий код[7]. Причиною цього є те, що ПЗ, що не застосовує шаблон інверсії контролю має сильно зв'язну архітектуру, що затрудняє внесення змін до коду і, як наслідок, збільшує час і вартість розробки.

Інверсія управління - це термін, що не вказує як саме ця інверсія повинна бути реалізована. Одним з найпоширеніших способів реалізації цього шаблону є Впровадження залежностей (Dependency Injection).

Впровадження - це передача об’єкта залежності (тобто, сервісу) залежному об'єкту (тобто, клієнту). Передавати залежності клієнту замість дозволити клієнту створити сервіс самостійно є фундаментальною вимогою до цього шаблону проектування.

До найбільш поширених форм впровадження залежностей входять:

1) впровадження в конструктор;
2) впровадження в поле класу;
3) впровадження в метод.

Впровадження залежностей — це шаблон проектування, в якому залежності (або сервіси), тобто об’єкти, впроваджуються або передаються по посиланню в залежний об'єкт (клієнт) і стають складовою клієнта. Шаблон відокремлює створення залежностей клієнта від власної логіки клієнта, що дозволяє компонентам бути слабко зв'язаними і дотримуватися принципів інверсії контролю і єдиної відповідальності. Він існує на противагу анти-шаблону service locator, що передбачає можливість клієнтів знати про систему, що використовується для пошуку залежностей.

Переваги:

1. Так як впровадження залежностей не вимагає змін у поведінці коду, цей шабон можна застосувати без додаткових труднощів в уже існуючій системі. В результаті цього клієнти стануть більш незалежними і над ними легше проводити модульне тестування з використанням об’єктів заглушок, які імітують об'єкти, від яких залежить об'єкт, що тестується. Простота тестування дуже часто є першою помітною перевагою використання впровадження залежностей.

2. Використання впровадження залежностей надає змогу клієнту не знати про конкретну реалізацію певної залежності, яку йому потрібно використовувати. Це дозволяє ізолювати клієнт від впливу змін логіки і структури програмного забезпечення. Це сприяє повторному використанню, тестуванню і підтримці коду.

3. Шаблон впровадження залежностей може використовуватися для перенесення інформації про налаштування системи в конфігураційні файли. Це дозволяє змінювати конфігурації програми без перекомпіляції. Деякі

конфігураційні файли можуть бути написані для різних ситуацій чи середовища, що вимагають різних реалізацій компонентів.

4. Впровадження залежностей сприяє паралельній розробці декількома людьми чи командами. Два розробника можуть незалежно створювати класи, які є залежностями один одного, знаючи тільки про інтерфейси чи абстрактні класи, через які класи співпрацюють.

5. Впровадження залежностей знижує ступінь зв'язності між класом і його залежностями.

1.2 Аналіз наявного програмно-апаратного забезпечення предметної області

1.2.1 Огляд Java фреймворка Spring Framework

Сьогодні існує багато фреймворків, що частково, або повністю забезпечують інверсію управління. Для мови програмування Java безумовним лідером останні роки є Spring Framework. Цей фреймворк насправді має дуже широкий і потужний функціонал для різних задач, проте головне місце займає контейнер інверсії управління(Inversion of Control Container)[8, 46, 47]. Згідно документації[1], Spring реалізує інверсію управління через впровадження залежностей. Для цього розробники створили клас Bean, що описує об’єкт Java класу, до якого може застосовуватись впровадження залежностей. Для створення об’єктів класу Bean використовується інтерфейс BeanFactory, що надає просунуті механізми налаштування і управління об’єктами, так званих компонентів(Beans). Після створення всі компоненти зберігаються в ApplicationContext. Створення об’єкта цього класу, як правило, починається при запуску програми. Коли всі компоненти будуть налаштовані і створені, вважається, що ApplicationContext готовий до використання.

Клас BeanFactory це частина реалізації контейнера інверсії управління в Spring, що відповідає за створення та налаштування компонентів.

ApplicationContext це інтерфейс відповідальний за створення, налаштування, та управління іншими компонентами. Контейнер отримує вказівки

щодо об’єктів, які слід створювати, конфігурувати та збирати, читаючи метадані конфігурації. Метадані конфігурації представлені у форматі XML, анотаціях Java або на мові програмування Groovy.

Анотація Java – це спеціальна форма метаданих, що може бути додана в вихідний код програми. Анотації використовуються для аналізу коду, компіляції або виконання.

Анотація виконує наступні функції:

1. Надає інформацію компілятору.
2. Надає інформацію інструментам для генерації іншого коду, певних налаштувань і т. д.
3. Може використовуватись під час виконання коду для отримання даних через рефлексію[21].

Метадані дозволяють виразити об'єкти, що складають вашу програму, і взаємозалежність між такими об'єктами. Кілька реалізацій інтерфейсу ApplicationContext є реалізовані, і зазвичай використовують одну з таких імплементацій. В автономних програмах зазвичай створюється екземпляр ClassPathXmlApplicationContext або FileSystemXmlApplicationContext. Хоча XML був традиційним форматом для визначення метаданих конфігурації, можна доручити контейнеру використовувати анотації або код Java як формат метаданих, надавши невелику кількість конфігурації XML, щоб декларативно увімкнути підтримку цих додаткових форматів метаданих. У більшості випадків ніякий додатковий клієнтський код не потрібен для створення одного або декількох екземплярів контейнера Spring IoC. Наприклад, у випадку веб-додатків, як правило, достатньо простих восьми (або близько того) рядків веб-дескриптора J2EE у файлі web.xml програми. Якщо використовується середовище розробки SpringSource Tool Suite Eclipse або Spring Roo, цю конфігурацію шаблону можна легко створити за допомогою декількох клацань миші або натискання клавіш. На наступній схемі(рис. 1.1) представлено на високому рівні те, як працює Spring Framework. Ваші класи програм поєднуються з метаданими конфігурації, так що після створення та ініціалізації ApplicationContext у вас є повністю налаштована та виконувана система чи програма.

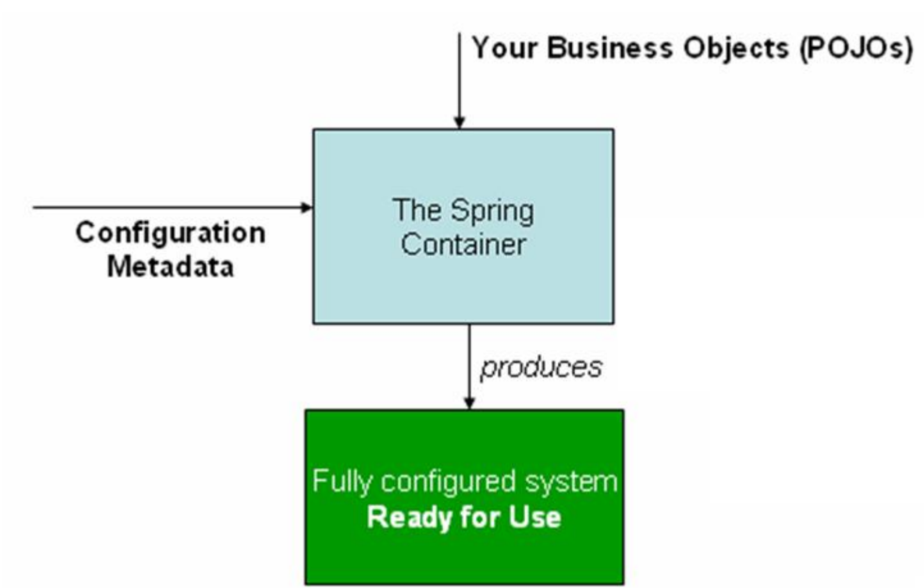


Рисунок 1.1 - Схема, що відображає логіку роботи IoC Spring контейнера.[26]

1.2.2 Метадані налаштувань

Як показано на попередній схемі, контейнер Spring IoC використовує метадані конфігурації; ці метадані конфігурації надаєте ви, як розробник додатків і в такий спосіб вказуєте контейнеру Spring, як створити, налаштувати та зібрати об’єкти у вашому додатку.

Метадані конфігурації традиційно подаються у простому та інтуїтивно зрозумілому форматі XML, що і використовується в більшості розділів для передачі ключових концепцій та особливостей контейнера Spring IoC.

Конфігурація Spring складається щонайменше з одного, як правило, більше одного визначення компонента, яким повинен керувати контейнер. Метадані конфігурації на основі XML відображають ці компоненти, налаштовані як елементи <bean />, всередині елемента <beans /> верхнього рівня.

Ці визначення компонентів відповідають фактичним об’єктам, що складають вашу програму. Зазвичай програміст визначає об'єкти сервісного рівня, об'єкти доступу до даних (DAO), об'єкти презентації, такі як екземпляри дії Struts, об'єкти інфраструктури, такі як Hibernate SessionFactories, черги JMS тощо. Як правило, в контейнері не налаштовуються дрібні об'єкти доменів, оскільки за

створення та завантаження об'єктів домену зазвичай відповідають DAO та бізнес-логіка. Однак можна використовувати інтеграцію Spring з AspectJ для налаштування об'єктів, створених поза контролем IoC-контейнера.

Наступний приклад показує базову структуру метаданих конфігурації на основі XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean id="..." class="...">
  <!-- тут будуть налаштування цього компонента -->
 </bean>

 <bean id="..." class="...">
  <!-- тут будуть налаштування цього компонента -->
 </bean>
 <!-- далі йдуть наступні оголошення компонентів-->
</beans>
```

Атрибут id - це рядок, який використовується для ідентифікації окремого визначення компонента. Атрибут class визначає тип компонента та використовує повну назву класу. Значення атрибута id необхідне для об'єктів, що працюють з цим компонентом. XML код для посилання на об'єкти, що працюють з іншими об’єктами, не показаний у цьому прикладі.

1.2.3 Створення контейнеру

Створення Spring IoC-контейнера є простим[49]. Шлях розташування або шляхи, що надаються конструктору ApplicationContext, насправді є назвами

ресурсів, які дозволяють контейнеру завантажувати метадані конфігурації з різних зовнішніх ресурсів, таких як локальна файлова система, Java Classpath, тощо.

```
ApplicationContext context = new
ClassPathXmlApplicationContext(new String[]
{"services.xml", "daos.xml"});
```

Наступний приклад показує як може виглядати файл “service.xml”

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/b
eans
      http://www.springframework.org/schema/beans/spring-
beans.xsd">

 <!-- сервіси -->

 <bean id="petStore"

class="org.springframework.samples.jpetstore.services.PetSt
oreServiceImpl">
  <property name="accountDao" ref="accountDao"/>
  <property name="itemDao" ref="itemDao"/>
  <!—додаткові налаштування компонента-->
 </bean>
</beans>
Наступний приклад показує можливий вміст файлу “daos.xml”
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/b
eans
      http://www.springframework.org/schema/beans/spring-
beans.xsd">

 <bean id="accountDao"
```

```
class="org.springframework.samples.jpetstore.dao.ibatis.Sql
MapAccountDao">
  <!—додаткові налаштування компонента-->
 </bean>

 <bean id="itemDao"
class="org.springframework.samples.jpetstore.dao.ibatis.Sql
MapItemDao">
  <!—додаткові налаштування компонента-->
 </bean>
</beans>
```

У попередньому прикладі сервісний рівень складається з класу PetStoreServiceImpl і має в собі два об'єкти доступу до даних типу SqlMapAccountDao та SqlMapItemDao, що базуються на фреймворці iBatis. Елемент імені властивості посилається на ім'я властивості JavaBean, а елемент ref - на ім'я іншого визначення компонента. Цей зв'язок між елементами id та ref виражає залежність між об'єктами, що співпрацюють.

1.2.4 Розділення XML конфігурацій на декілька файлів

Може бути корисно, щоб визначення компонента охоплювали декілька файлів XML. Часто кожен окремий файл конфігурації XML представляє логічний рівень або модуль у вашій архітектурі.

Можна використовувати конструктор контексту програми для завантаження визначень компонентів з усіх цих фрагментів XML. Цей конструктор бере декілька розташувань ресурсів, як було показано в попередньому розділі. Крім того, можна використовувати один або декілька елементів <import /> для завантаження визначень компонентів з іншого файлу або файлів. Наприклад:

```
<beans>
  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>
  <bean id="bean1" class="..."/>
```

```
    <bean id="bean2" class="..."/>
</beans>
```

У попередньому прикладі визначення зовнішніх компонентів завантажуються з трьох файлів: services.xml, messageSource.xml та themeSource.xml. Усі шляхи розташування відносяться до файлу визначення, який виконує імпорт, тому services.xml повинен знаходитись у тому ж каталозі або розташуванні шляху до класу, що і файл, що виконує імпорт, тоді як messageSource.xml та themeSource.xml повинні знаходитись у розташуванні ресурсів нижче розташування файлу імпорту. Як бачите, символ / на початку шляху ігнорується, але враховуючи, що ці шляхи відносні, краще не використовувати косу риску взагалі. Вміст файлів, що імпортуються, включаючи елемент <beans /> верхнього рівня, повинен бути дійсними визначеннями компонентів XML відповідно до схеми Spring.

1.2.5 Використання контейнера інверсії управління

ApplicationContext - це інтерфейс для просунутої фабрики, здатної вести реєстр різних компонентів та їх залежностей. За допомогою методу T getBean (ім'я об’єкта, клас <T> requiredType) можна отримувати екземпляри ваших компонентів.

ApplicationContext дозволяє читати визначення компонентів і отримувати до них доступ наступним чином:

```
ApplicationContext context = new
ClassPathXmlApplicationContext(new String[]
{"services.xml", "daos.xml"});
// retrieve configured instance
PetStoreServiceImpl service = context.getBean("petStore",
PetStoreServiceImpl.class);
// use configured instance
List userList = service.getUsernameList();
```

Використовується метод getBean () для отримання екземплярів ваших компонентів. Інтерфейс ApplicationContext має кілька інших методів отримання компонентів, але в ідеалі ваш код програми ніколи не повинен їх використовувати. Дійсно, у вашому коді програми взагалі не повинно бути викликів методу getBean (), а отже, взагалі ніякої залежності від інтерефейсу Spring. Наприклад, інтеграція Spring з веб-фреймворками передбачає введення залежностей для різних класів веб-фреймворку, таких як контролери та компоненти, керовані JSF.

1.2.6 Впровадження залежностей

Введення залежностей (DI) - це процес, за допомогою якого об’єкти визначають свої залежності[12], тобто інші об’єкти, з якими вони працюють, лише за допомогою аргументів конструктора, аргументів до методу класу фабрики або властивостей, які встановлюються в екземплярі об’єкта після його побудови або повернення від методу класу фабрики. Потім контейнер вводить ці залежності, коли створює компонент. Цей процес є принципово зворотнім, звідси і назва Інверсія контролю (Inversion of Control)[15] самого компонента, який самостійно керує створенням екземплярів або розташуванням своїх залежностей, використовуючи пряму побудову класів або шаблон Locator Service[11].

Код є чистішим за принципом DI, і більш ефективним, коли об’єкти отримують свої залежності. Об'єкт не шукає своїх залежностей і не знає розташування або класу залежностей. Таким чином, ваші класи стають простішими для тестування, зокрема, коли в якості залежностей використовуються інтерфейси або абстрактні базові класах, які дозволяють підміняти імплементації в юніт тестах.

DI існує у двох основних варіантах: введення залежностей на основі конструктора та введення залежностей на основі сеттера.

DI на основі конструктора виконується за допомогою контейнера, що викликає конструктор із кількістю аргументів, кожен з яких представляє залежність. Виклик статичного фабричного методу з конкретними аргументами

для побудови компонента майже рівнозначний, і це обговорення аналогічно розглядає аргументи конструктора та статичного фабричного методу. Наступний приклад показує клас, який можна вводити залежно лише за допомогою введення конструктора. Зверніть увагу, що в цьому класі немає нічого особливого, це звичайний Java клас, який не має залежностей від конкретних інтерфейсів контейнера, базових класів чи анотацій.

```
    public class SimpleMovieLister {

     private MovieFinder movieFinder;

     // конструктор, через який Spring може передати
об'єкт MovieFinder

     public SimpleMovieLister(MovieFinder movieFinder) {
       this.movieFinder = movieFinder;
     }
```

DI на основі сеттера виконується контейнером, що викликає методи setter у ваших компонентах, після виклику конструктора без аргументів для створення екземпляра вашого компонента.

Наступний приклад показує клас, якому можна вводити залежності лише за допомогою setter методу. Це такий самий звичайний Java клас, що не має залежностей від конкретних інтерфейсів контейнера, базових класів або анотацій.

```
public class SimpleMovieLister {
 private MovieFinder movieFinder;
 // сеттер метод, за допомогою якого Spring може передати
об'жкт класу MovieFinder
 public void setMovieFinder(MovieFinder movieFinder) {
   this.movieFinder = movieFinder;
 }
}
```

ApplicationContext підтримує DI на основі конструктора та сеттера для керованих компонентів. Він також підтримує впровадження залежностей на

основі сеттера після того, як деякі залежності вже вводяться через конструктор. Залежності налаштовуються у вигляді BeanDefinition, який використовуєте з екземплярами PropertyEditor для перетворення властивостей з одного формату в інший. Однак більшість користувачів Spring працюють не з цими класами безпосередньо (програмно), а з файлом визначення XML, який потім внутрішньо перетворюється у екземпляри цих класів і використовується для завантаження цілого екземпляра контейнера Spring IoC.

1.2.7 Кругові залежності

Якщо використовується переважно впровадження залежностей через конструктор, можна створити нерозв'язний сценарій кругової залежності.

Наприклад: Клас A вимагає екземпляра класу B через введення в конструктор, а клас B вимагає екземпляра класу A через введення в конструктор. Якщо програміст налаштує компоненти для класів A і B для введення один в одного, контейнер Spring виявляє це циклічне посилання під час виконання та генерує виключення класу BeanCurrentlyInCreationException.

Кругові залежності в програмному забезпеченні вважається анти-шаблоном, оскільки це затрудняє створення таких об’єктів, тому їх потрібно уникати. Однак, якщо вони все ж існують в коді, то Spring дозволяє створення таких компонентів.

Одним із можливих рішень є редагування вихідного коду деяких класів, щоб використовувати впровадження через сетери, а не конструктори. Іншими словами, хоча це не рекомендується, можна налаштувати кругові залежності за допомогою впровадження через сетер.

На відміну від типового випадку (без кругових залежностей), кругова залежність між компонентом A та компонентом Б змушує один з компонентів передавати в інший до того, як повністю його ініціалізувати.

1.3 Визначення вимог до системи автоматизації та розробка технічного завдання

Основною задачею є проектування і реалізація програмного забезпечення, ціллю якого буде інверсія контролю, за допомогою впровадження залежностей.

У результаті, розроблене програмне забезпечення повинно надавати функціонал з створення та управління програмними компонентам за допомогою Java анотацій. Повинна бути підтримка шаблону Singleton[3, 7, 13], тобто створення лише одного об’єкта певного класу і впровадження його у всі компоненти, яким потрібен цей об’єкт. Це означає, що повинна бути розроблена можливість кешування створених об’єктів. Якщо компонент потребує впровадження іншого компонента, який ще не був створений, то створення компонента повинно бути відкладеним до того, як необхідна залежність буде створена. Якщо необхідний компонент не знайдений після створення всіх інших компонентів, то ініціалізація об’єктів повинна бути призупинена, а контейнер інверсії контролю повинен згенерувати виключення з повідомленням про помилку.

Обов’язковою є підтримка впровадження залежностей об’єкту через поля класу, навіть якщо вони приватні і не мають реалізованих необхідних setter методів.

Також, повинна бути реалізована можливість динамічної генерації класів для забезпечення існування проксі об’єктів. Налаштування особливостей роботи проксі об’єктів та умов генерації проксі класів повинна бути можлива користувачем контейнеру інверсії контролю без змін в вихідний код реалізованого програмного засобу.

1.4 Висновки

Інверсія контролю – це шаблон проектування, що вирішує актуальні проблеми в об’єктно-орієнтованому програмуванні. Сьогодні практично кожна об’єктно-орієнтована мова програмування має свої імплементації цього шаблону.

В Java найпопулярнішим фреймворком, що працює як контейнер інверсії контролю є Spring Framework.

2 ПРОЄКТУВАННЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ

2.1 Функційні вимоги

Створений програмний засіб повинен реалізовувати принципи інверсії контролю за допомогою впровадження залежностей. Для того щоб користуватись розробленим ПЗ було зручно, повинні бути реалізовані можливості просунутих налаштувань об’єктів.

Однією з реалізованих можливостей повинна бути повноцінна підтримка принципу, що реалізований в шаблоні проектування Singleton. Так повинна бути можливість вказати контейнеру інверсії управління на те, що певний клас повинен мати лише один об’єкт на програму.

Для реалізації шаблону Singleton використовують різні способи, одним є створення статичного синхронізованого методу в класі, що самостійно створює свій екземпляр.

Іншою відомою реалізацією даного шаблону є реалізація його через використання типу enum[4, 43], в якому визначено один статичний об’єкт. Так використання enum дає багато переваг, оскільки в Java для цього типу вже вирішені певні проблеми, що потрібно було б вирішувати самостійно[48]. Однак, використання enum не дає такої ж гнучкості як використання класу, оскільки тоді зникають такі можливості ООП як наслідування, чи поліморфізм.

При розробці ПЗ необхідно реалізувати концепції Singleton таким чином, щоб дотримання цього принципу було відповідальністю контейнера інверсії управління, а не класу, що потребує цього.

До функційних вимог також входить підтримка генерації проксі-об’єктів для просунутого налаштування та управління компонентами програми.

2.2 Визначення кроків для початку проектування програмного забезпечення

Проектування нового програмно-апаратного рішення це не проста задача. В першу чергу важливим є уникнути типові помилки, що виникають при розробці

будь-якого програмного забезпечення[16, 17, 18]. Спочатку, для побудови успішного продукту, необхідно чітко визначитись з вимогами, які ставляться до продукту. Для цього потрібно розуміти задачу і проблему яку створене програмне забезпечення повинно вирішити.

Наступним кроком необхідно визначити проблеми, що можуть виникнути на шляху реалізації програмного рішення саме для тієї області, для якої воно призначено. Найкращим способом зробити це є аналіз існуючих рішень, те як вони розроблені, за допомогою яких технологій, а також їх переваги та недоліки.

Кінцевим результатом очікується, що буде розроблене програмне забезпечення, що поєднує в собі усі переваги існуючих рішень, але при цьому не має їх недоліків.

На першому етапі проектування програмного забезпечення важливим є вибір технологій для розробки. Перед вибором технологій варто провести дослідження щодо того чи підходить певний інструмент для вирішення поставленої задачі. Для того щоб це зробити необхідно ознайомитись з офіційною документацією бібліотеки чи фреймворку.

Після ретельного відбору технологій відбувається встановлення всіх необхідних програмних засобів, бібліотек, інструментів та їх залежностей. Проводиться налаштування компонентів відповідно до вимог розробника.

Другим етапом є безпосередньо процес розробки на базі обраного набору технологій. Перед розробником постає вимога практичної реалізації з використанням відповідних методів розробки компонентів програмного забезпечення[38, 39]. Проведення тестування готового результату, можливостей програмного рішення, і якості.

Так метод розробки ПЗ для реалізації інверсії контролю на мові Java складається з:

1. Вибір програмних інструментів.
2. Ознайомлення з документацією.
3. Налаштування вибраних інструментів.
4. Реалізація компонентів продукту.
5. Тестування готового результату.

Важливим моментом у роботі, що існує на протязі всього життєвого циклу продукту є підтримка. Незалежно від програмної архітектури проекту з плином часу код ускладняється, втрачає початковий стиль та читабельність[22, 23, 24]. Коли читабельність порушена, як наслідок розробник, не знайомий з проектом, повинен витрачати зайвий час на розуміння тих чи інших частин функціоналу. Для виправлення такої ситуації існують стандарти оформлення коду. Якщо проект масштабний, то людина не може контролювати дотримання всіх норм стилізації. Для уникнення цієї проблеми був розроблений наступний метод з використанням статичного аналізатору.

Таким чином даний метод складається з наступної кількості кроків:

1. Вибір статичного аналізатора коду.
2. Встановлення аналізатора в проект.
3. Підключення в середовище розробки.
4. Конфігурування файлу налаштувань.

2.3 Вибір рішення для автоматизації збирання та управління проектами.

Для будь-якого програмного забезпечення, написаного на мові Java є сенс використовувати засоби для автоматичного збирання та управління проектами.

Автоматизація збирання – це автоматизація широкого спектра завдань, що виконують розробники під час роботи над проектом. Включає в себе наступні дії:

1) компіляцію вихідного коду в бінарний;
2) збирання бінарного коду;
3) запуск тестів;
4) розгортання програми на сервері;
5) написання документації або опис змін в новій версії.

До переваг використання автоматизації збирання на проекті можна віднести наступне:

1) покращення якості проекту;
2) прискорення компіляції і обробки посилань;
3) вилучення непотрібних етапів збирання проекту;

4) мінімізація збірок проектів з помилками;

5) зберігання історії збірок для виявлення причин помилок;

6) економія часу та грошей через все вище зазначене.

Базові вимоги до системи автоматизації складання:

1. Підтримка керування залежностями вихідного коду (Source Code Dependency Management).

2. Повідомлення при збігові вихідного коду після складання, з наявними бінарними файлами.

3. Прискорення збирання проекту.

4. Звіт про результати компіляції і компонування.

Додаткові вимоги:

1. Створення опису змін (release notes) та іншої супутньої документації.

2. Звіт про статус збирання.

3. Звіт про проходження тестів.

4. Підсумовування змін у кожному новому складанні.

Сьогодні для Java платформи існують наступні основних засобів для збирання:

1. Apache Ant.

2. Apache Maven.

3. Gradle.

Далі детальніше розглянемо кожен з них.

Apache Ant – це java-утиліта для автоматизації збирання програмного продукту. Ant(англ. Ant - мураха) є аналогом UNIX-утиліти make, але краще пристосований для Java проектів. Управління процесом відбувається за допомогою написання XML-сценарію. До того ж Ant є платформонезалежним проектом, оскільки написаний на мові програмування Java. Це дозволяє використовувати цю утиліту як на операційній системі Windows, так і на Unix системах. Остання версія Ant містить в собі близько 150 типів завдань, таких як компіляція коду, копіювання файлів, запуск юніт-тестів і т. д.

Apache Maven — це інструмент для автоматизації роботи з програмними проектами на базі Java платформи[2, 45]. Він використовується для управління та

збирання програм. Maven був створений в 2002 році Джейсоном ван Зилом. За принципом роботи дуже сильно відрізняється від Apache Ant, і має простіший вигляд щодо build-конфігурацій, що надаються в форматі XML. XML-файл описує проєкт, та його зв'язки з іншими компонентами та зовнішніми модулями, порядок збирання компонентів(build), папки і необхідні плагіни. Додаткові модулі та бібліотеками розміщується на серверах. Раніше Maven був частиною Jakarta EE.

Gradle — це система для автоматичного збирання проектів[20], яка продовжує розвивати принципи, закладені в Apache Ant та Apache Maven і використовує предметно-орієнтовану мову (DSL) на основі мови програмування Groovy замість традиційної для Maven та Ant XML-подібної форми представлення конфігурацій проекту. Для визначення порядку етапів збирання Gradle використовує орієнтований ациклічний граф.

На відміну від Apache Maven, що має на основі концепцію життєвого циклу проекту, і Apache Ant, в якому порядок виконання задач визначається відносинами залежностей (depends-on), Gradle використовує спрямований ациклічний граф для визначення порядку виконання завдань.

Gradle було розроблено для побудови великих мультипроектів, що можуть масштабуватись, і підтримувати інкрементальне збирання. Gradle самостійно визначає, які частини було змінено, і виконує тільки ті етапи, що залежать від цих частин.

Задачі в Gradle описуються за допомогою плагінів. В основному, плагіни призначені для розробки і розгортання додатків на мовах Java, Groovy і Scala, але розробляються плагіни і для інших мов програмування.

Отже, у результаті було обрано Maven, як засіб автоматизації збирання проекту, що є сучаснішим за Ant, та використовує XML як основний спосіб написання конфігурацій.

2.4 Вибір версії та вендора мови програмування Java.

Також, важливим є вибір мови програмування. Згідно завдання потрібно використовувати Java для розробки програми, але ця мова програмування не стоїть на місці і постійно розвивається, то тут необхідно визначитись з версією, яку буде обрано. Не дивлячись на те, що розробники платформи Java намагаються зберігати зворотню сумісність[10] з попередніми версіями мови, змінити версію на готовому проекті може стати складною задачею. Здавалося б, найпростішим способом зробити правильний вибір, було б обрати останню версію і мати увесь можливий функціонал. Однак, для будь-яких серйозних проектів рекомендується все ж такий обирати LTS(Long Term Support) версію Java.

Long Time Support – це частина життєвого циклу програмного забезпечення, що означає, що стабільний реліз буде підтримуватись довгий час і буде вважатись стандартною версією для використання.

LTS збільшує період супроводження програмного забезпечення, також зміню тип патчів і частоту їх виходу, так як переслідуються цілі зменшити ризики, вартість і кількість зривів термінів поставки. З початку періоду LTS розробники повинні призупинити випуск нових функцій і зосередитись на виправленні помилок і закриття критичних вразливостей.

Згідно статистики(рис. 2.1)Сьогодні останньою версією Java є Java 16, а останньою LTS версією є Java 11. Однак, згідно статистики, більшість проектів все ще використовують Java 8 як основну версію на проекті.

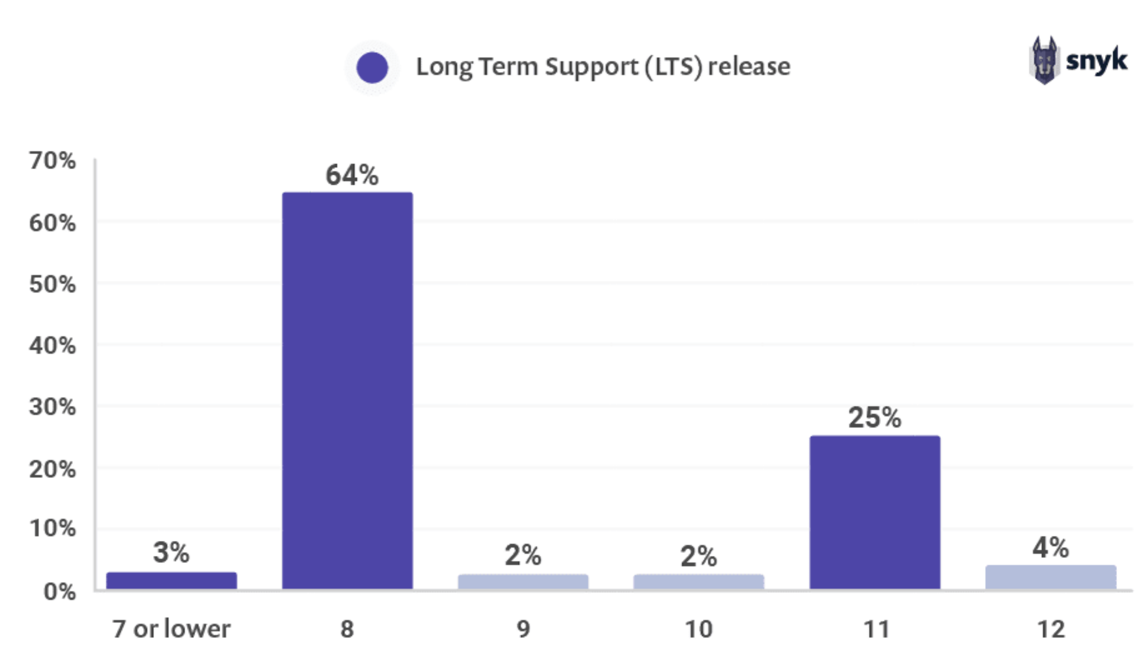


Рисунок 2.1 - Статистика використання версій Java на проектах [6]

Чому ж більшість обирає старішу Java 8, за новішу Java 11, хоча остання теж є версією довгострокової підтримки?

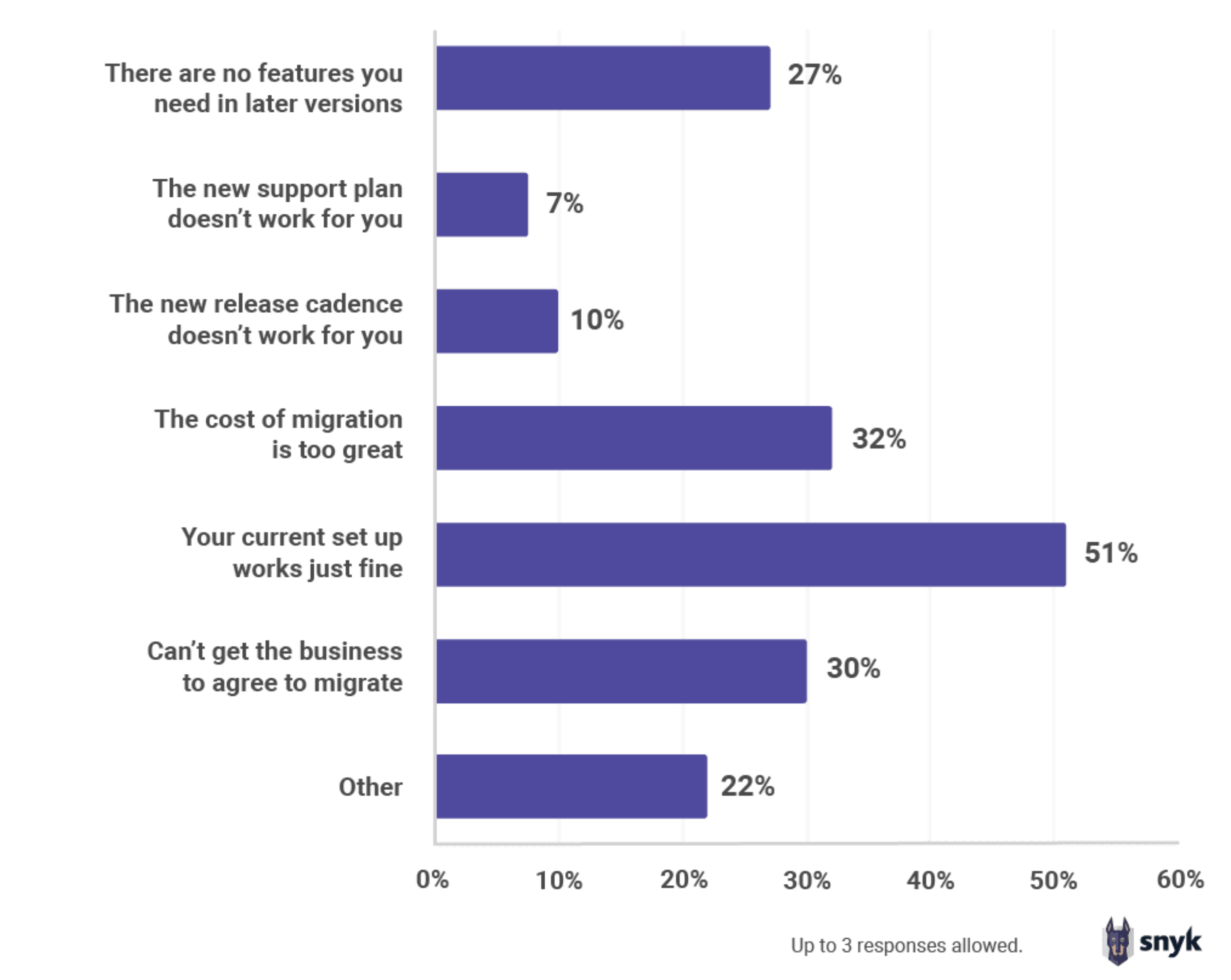


Рисунок 2.2 – Результати опитування по причинах чому розробники віддають перевагу до старіших версій Java.[6]

Згідно опитування(рис. 2.2) більшість розробників(51%) просто не вважають за потрібне робити міграцію проекту на новішу версію, оскільки все працює добре. Також третина вважає, що міграція обійдеться занадто дорого, щоб проводити її. Також 27% вважає, що функції та особливості, що представлені в новіших версіях їм не потрібні. Це не дивно, оскільки після виходу Java 8, в якій було представлено Stream API, що додає деякого функціонального стилю в програмування, не було випущено на стільки ж масштабних оновлень.

До того ж, важливим є вибір вендора Java. Java проголошує певні стандарти, але як вони повинні бути реалізовані, вендор вирішує сам. Вибір вендора впливає на ліцензію під якою можна використовувати надане програмне забезпечення і на

ціну використання. Найбільшими провайдерами сьогодні є Oracle і OpenJDK(з відкритим вихідним кодом).

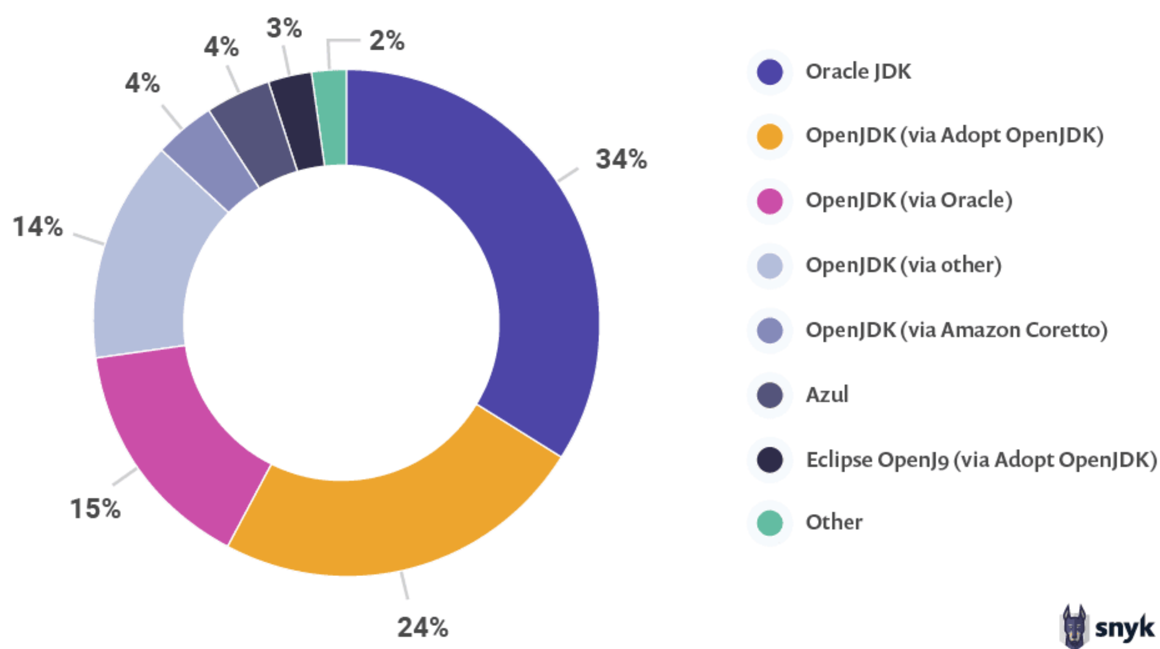


Рисунок 2.3 – Статистика вендорів за 2020 рік, чиї реалізації Java використовуються на проектах[5]

Згідно статистики(рис. 2.3) лише 34% опитаних використовують реалізацію Java від Oracle. Більшість же з різних причини обирає версію з відкритим вихідним кодом. Крім цього, за рік кількість тих, хто віддає перевагу версії від Oracle скоротилась на 36%(рис. 2.4).

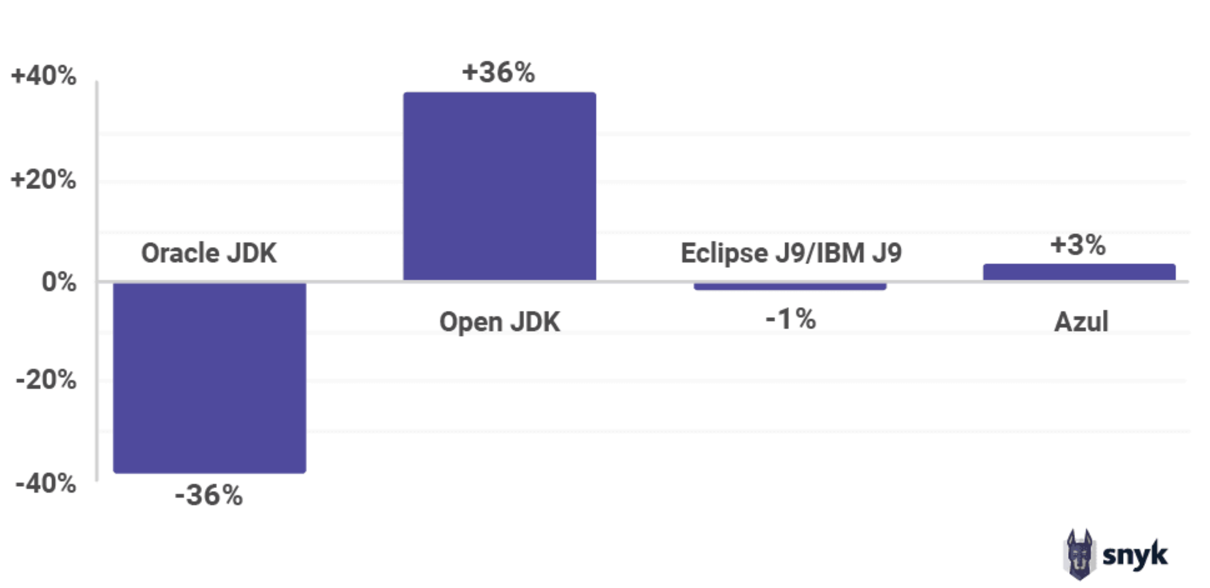


Рисунок 2.4. – Статистика зміни рейтингу вендорів між 2019 і 2020 роком.[5]

Отже, для розробки нового проекту є сенс обрати останню LTS версію Java, а саме Java 11 від OpenJDK.

2.5 Вибір засобів для автоматичної генерації коду.

Також, для розробки часто використовують бібліотеки для автоматичної генерації коду. Одним з таких засобів є Lombok.

Lombok – це бібліотека для автоматичної генерації коду[29], за допомогою анотацій. Під час розробки будь-якого програмного забезпечення, кожен Java розробник повинен писати методи, що необхідні кожному класу. Такими методами можуть бути гетери, сетери а також конструктори. Коли в класі є з десяток полів, а самих класі в проекті можуть нараховуватись тисячі – це може сповільнювати розробку. В сучасних середовищах розробки існують засоби для генерації поширених методів автоматично, але часто це зменшує читабельність коду, оскільки на одне поле, скоріше за все буде приходитись, як мінімум, 2 методи(гетер та сетер). Lombok призначений для того, щоб усунути цю проблему, використовуючи анотації для генерації коду на етапі компіляції.

Окрім звичних гетерів та сетерів, Java розробники повинні перевизначати такі методи як boolean equals(Object obj) та int hashCode(). При чому,

обов’язковим є перевизначення цих методів разом і згідно певних конвенцій. Наприклад, якщо метод equals повертає значення true, то значення, що повертає метод hashcode обох об’єктів повинні бути рівні. Однак, якщо хеш коди об’єктів однакові, це не обов’язково означає, що ці об’єкти рівні, адже при вирахуванні хешу об’єкту можуть виникати колізії. Важливо слідувати цим конвенціям, оскільки інші розробники очікують, що ваше програмне забезпечення слідує ним. Якщо порушиться equals-hashcode контракт, то, наприклад, зберігати ваші об’єкти в Java колекція[4, 40], таких як Map стане неможливим, оскільки є можливість втратити дані. Map – це структура вигляду ключ та значення. Також важливим є перевизначати метод вирахування хеш коду так, щоб хеш код не змінювався, якщо об’єкт не змінився. Така імплементація як HashMap як ключ використовує хеш код ключа, тому якщо хеш код буде змінено, то об’єкт, що зберігається в колекції буде втрачено. Саме тому рекомендується в якості ключа завжди використовувати класи, що є незмінні, наприклад String.

Lombok дозволяє вирішити більшість з цих проблем, і включає в себе:

1) генерацію гетерів та сетерів;
2) генерація методів equals та hashcode;
3) генерацію методу toString();
4) генерація коду для створення об’єкта для логування;
5) генерацію коду, що обгортає код методу в try-catch структуру;
6) Отже, до переваг використання засобу генерації коду Lombok можна віднести:
7) автоматична генерація коду згідно прийнятих стандартів;
8) код виглядає зрозумілішим;
9) простота використання;
10) підтримка і випуск оновлень бібліотеки.

До недоліків Lombok можна віднести:

1) приховування реалізації;
2) необхідність використовувати середовище розробки з встановленим плагіном;

3) труднощі для розробників, що не використовували дану бібліотеку раніше.

У висновку, я вважаю є сенс використовувати бібліотеку Lombok для автоматичної генерації коду в проекті з реалізації інверсії контролю на мові Java.

2.6 Вибір бібліотек для роботи з рефлексією

Разом зі стандартним набором бібліотек в JDK присутні також класи, що надають можливість працювати з рефлексією.

Рефлексія – це механізм дослідження даних програми під час її виконання. Рефлексія дозволяє отримати інформацію про поля, методи та конструктори класів. Механізм рефлексії[37] дозволяє оброблювати класи, що відсутні під час компіляції. Рефлексія надає можливість створювати коректний динамічний код.
До можливостей, що надає Java Reflections API входять:

1) дізнатись клас об'єкта;

2) отримати інформацію про модифікатори класу, полях, методів, констант, конструкторів і суперкласи;

3) вияснити які методи належать до інтерфейсу;

4) створити об'єкт класу, навіть при умові, що назву класу невідомо моменту виконання програми;

5) отримати і встановити значення поля об'єкта за іменем;

6) викликати метод об'єкта за іменем.

Рефлексія використовується практично в більшості сучасних технологій Java.
Наступний приклад показує як можна вивести інформацію про тип на назву полів класу:

```
Class c = obj.getClass();
Field[] publicFields = c.getFields();
for (Field field : publicFields) {
        Class fieldType = field.getType();
        System.out.println("Ім'я: " + field.getName());
        System.out.println("Тип: " + fieldType.getName());
}
```

Окрім стандартних засобів Java для рефлексії також існують спеціальні бібліотеки, що надають просунуті можливості для використання Java Reflections API. Однією з таких бібліотек є Cglib[25].

Cglib(Byte Code Generation Library) – це бібліотека, що надає API високого рівня для генерації та трансформації Java байт коду. Так як класи в Java завантажуються динамічно, під час виконаня програми, Cglib використовує цю особливість для того, щоб зробити можливим додавання нових класів в уже запущену програму. Cglib використовується аспектно-орієнтованому програмування[41, 42], в тестуванні, і фреймворках для роботи з даними для генерації динамічних проксі об'єктів.

Проксі об'єкти - це об'єкти класів, що виступають як надбудова над оригінальним класом і дозволяє, за необхідності, змінити його поведінку.

Так, механізм відкладеної ініціалізації в Java фреймворку Hibernate для роботи з даними працює за допомогою Cglib. При зчитуванні запису з БД, Hibernate може повертати не весь об'єкт, що зберігається в БД, а об'єкт проксі-класу, що завантажить дані тільки коли вони будуть необхідні.

Spring Framework також використовує Cglib, зокрема для того, щоб забезпечити авторизацію для використання певних методів, або проведення транзакцій над БД в рамках певного методу.

Ще однією, менш відомою, та не менш корисною бібліотекою, що містить класи для роботи з рефлексією є Reflections. Її особливістю є те, що вона надає просунуті можливості для зчитування метаданих класів, що є необхідним для роботи контейнера інверсії контролю.

2.7 Визначення основних компонентів програмного забезпечення.

2.7.1 Фабрика для створення об'єктів

Інверсія контролю в першу чергу передбачає передачу відповідальності за створення об'єктів на контейнер інверсії контролю. Відповідно, потрібно передбачити можливість створення об'єктів класів, що були визначені як компоненти, що контролюються контейнером. В даному випадку необхідно

застосувати шаблон проектування під назвою «Фабрика»[50]. Це дозволить написати програмне забезпечення, за принципами, що відомі кожному розробнику. До того ж, це відповідає принципу «Єдиної відповідальності», коли клас наділяється лише однією відповідальністю[32, 33, 34].

2.7.2 Конфігуратор об’єктів

Перед створенням об’єкта важливо знати як саме його необхідно налаштувати для правильної роботи. Для цього в програмі повинні існувати конфігуратори, що відповідальні за налаштування об’єкта, згідно метаданих, вказаних в класі користувача. Важливо зазначити, що об’єкти конфігуратори не повинні створювати об’єкти самостійно, або зберігати в собі інші конфігуратори.

2.7.3 Метадані компонентів

Для того, щоб створити компонент, що керується контейнером і до якого здійснюється впровадження залежностей, спочатку необхідно просканувати пакети задані користувачем, і знайти компоненти для створення. Насамперед потрібно передбачити анотації, що будуть слугувати метаданими які шукає сканер. Якщо клас не містить метаданих, що вказують на те, що об’єкти класу повинні управлятись контейнером, то ці класи просто ігноруються.

2.7.4 Кеш об’єктів

Часто в сучасних програмах написаних на об’єктно-орієнтованих мова програмування застосовується концепція Singleton, коли клас має лише один створений об’єкт на всю програму. Тому якщо під час впровадження залежностей один компонент потребує іншого, що відмічений як клас, що може мати лише один об’єкт, то контейнер повинен передати об’єкт, що був створений раніше. Для цього контейнер повинен кешувати такі об’єкти на етапі створення, згідно блок-схеми(рис. 2.5).

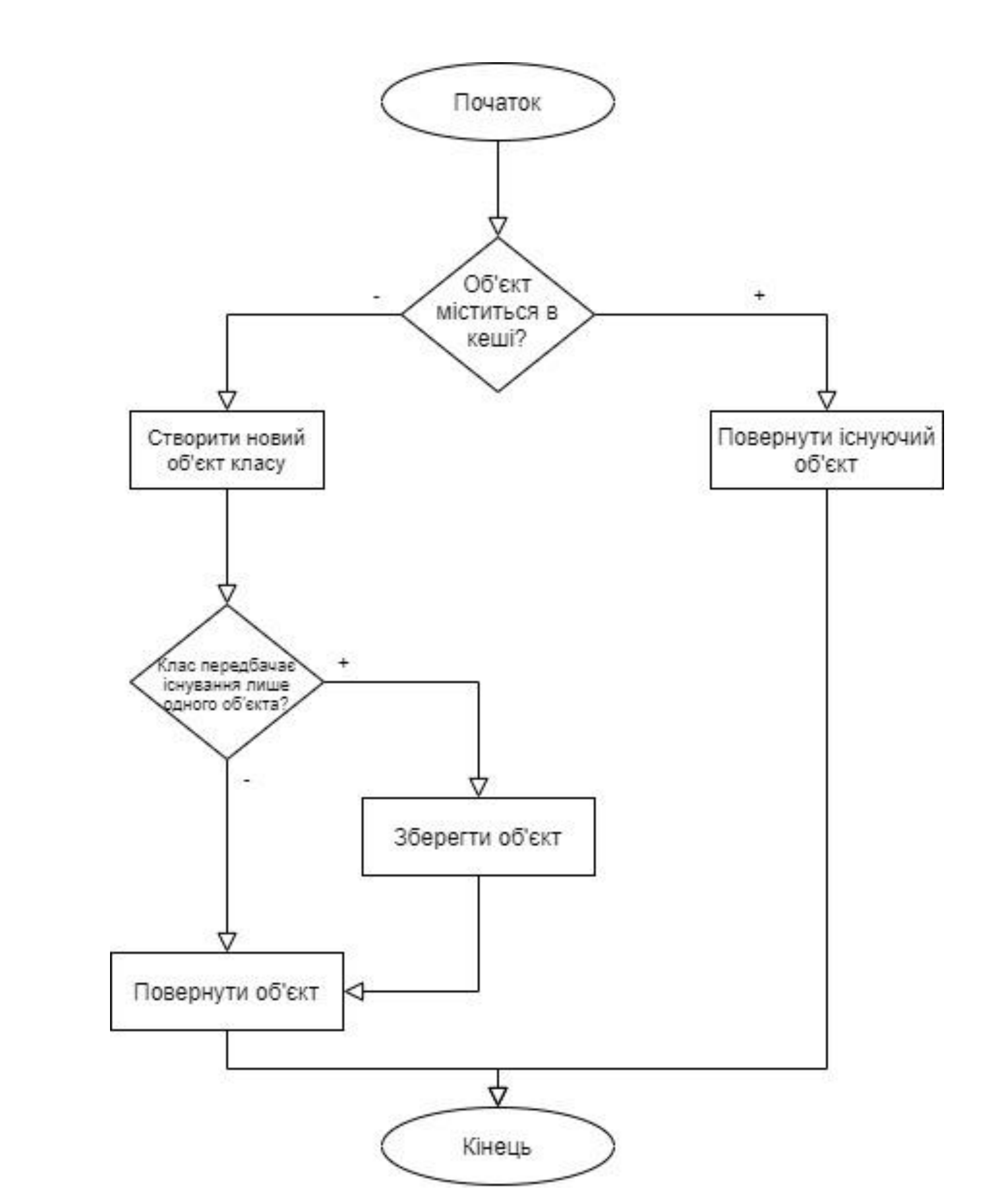


Рисунок 2.5 – Блок-схема кешування об’єктів контейнером інверсії контролю на етапі створення.

2.7.5 Компонент для зчитування конфігурацій з конфігураційного файлу

В сучасних програмах часто є необхідність мати певний конфігураційний файл, що зчитується в програмі і відповідно нього налаштовується середовище. Це додає гнучкості при розробці будь-якого застосунку, адже це дає змогу змінювати поведінку програми в залежності від середовища в якому воно запускається. Так розробники можуть мати необхідність налаштовувати тестове середовище, для того щоб протестувати додаток перед релізом нової версія, адже так можна попередити помилки, що можуть виникнути в новому коді. Для цього розробникам необхідно лише мати правильний конфігураційний файл, без зміни безпосередньо Java коду і перекомпіляції додатку.

2.7.6 Компонент пошуку імплементацій класу

Однією з відповідальностей, що лежить на контейнері інверсії контролю є створення екземплярів класу і їх залежностей. Однак, однією з залежностей класу може бути абстрактний клас, або інтерфейс. Створити екземпляр інтерфейсу без імплементації неможливо, тому необхідно передбачити в програмі компонент, що буде знаходити імплементацію необхідного інтерфейсу. Зазвичай для інтерфейсу існує лише одна імплементація, але це не обов’язково, тому також необхідно передбачити можливість існування декількох імплементацій. Компонент повинен працювати згідно блок-схеми(рис. 2.6).

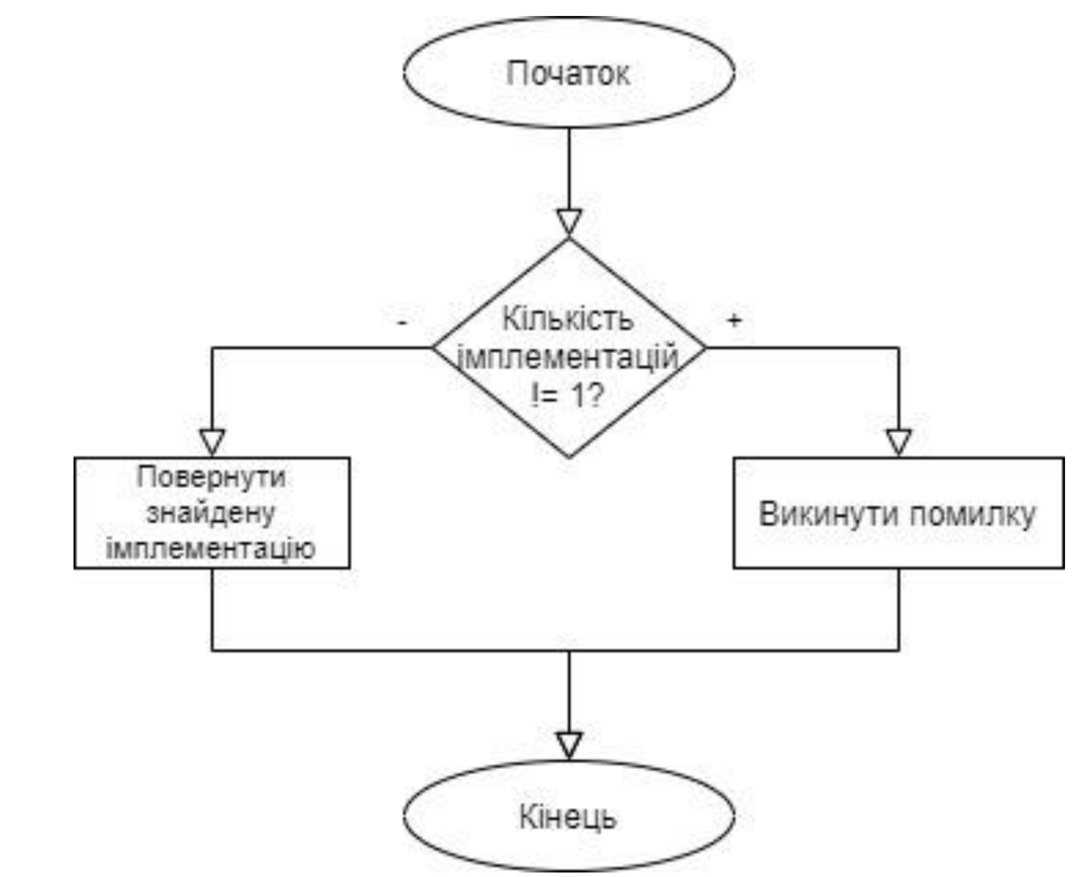


Рисунок 2.6 – Блок-схема роботи компонента з пошуку імплементацій інтерфейсу.

2.8 Висновки

В даному розділі було проаналізовані та визначені технології за допомогою яких буде реалізована практична частина завдання. Досліджено переваги і недоліки програмних засобів, що будуть використані в програмній реалізації, в тому числі автоматизатори для збирання проектів та бібліотеки автоматичної генерації коду.

Не завжди є необхідність використовувати усі можливості середовища для вирішення певної задачі. В сучасних проектах часто на налаштування інфраструктури для застосунку виділяють часу та сил не менше, ніж на безпосередньо програму. Це дозволяє автоматизувати багато речей, що відбуваються на проекті, особливо на етапі активної розробки. Однак, так як програмне забезпечення, що буде розроблене згідно завдання є по собі

інфраструктурним рішенням, в ньому буде використано лише мінімальну кількість необхідних технологій.

Окрім цього, були визначені та описані основні компоненти програми, з детальним поясненням їх задач та зони відповідальності.

3 ПРОГРАМНО-АПАРАТНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ

3.1 Реалізовані функціональні особливості

У результаті проектування було реалізовано програмне забезпечення, що реалізує принципи інверсії управління через впровадження залежностей. Так, контейнер інверсії управління може створювати, налаштовувати та керувати об’єктами. Окрім базових функцій контейнера інверсії управління, були також розроблені додаткові особливості, такі як підтримка концепції Singleton, пост конструктори, чи читання конфігураційних файлів. Окрім використання реалізованих рішень, кінцевий користувач має змогу розширювати функціонал контейнера інверсії управління за допомогою імплементації іинтерфейсів, за допомогою яких працює реалізоване програмне забезпечення.

3.1.1 Концепція Singleton

Однією з можливостей, що надає ПЗ є можливість використовувати концепцію Singleton без реалізації даного шаблону проектування в коді самостійно, що вважається поганою практикою. Для того, щоб це зробити необхідно лише написати анотацію Singleton над класом до якого повинне застосовуватись налаштування. Ось приклад коду, що застосовує концепцію Singleton за допомогою реалізованого ПЗ.

```
 @Singleton
public class RecommendatorImpl implements Recommendator {
      public RecommendatorImpl() {
        System.out.println("recommendator was created");
    }
}
```

До прикладу, реалізація цього шаблону самостійно виглядала б наступним чином:

```
public final class RecommendatorImpl implements
Recommendator {
  private static volatile RecommendatorImpl instance;


  private RecommendatorImpl () {
        System.out.println("recommendator was created");
  }
  public static RecommendatorImpl getInstance() {
    RecommendatorImpl result = instance;
    if (result != null) {
      return result;
    }
    synchronized(RecommendatorImpl.class) {
      if (instance == null) {
        instance = new RecommendatorImpl ();
      }
      return instance;
    }
  }
}
```

Як видно з прикладу вище, реалізація шаблону Singleton є не простою задачею і потребує глибоко розуміння того як працює мова програмування Java, а також багатопоточність[4, 19, 31]. В прикладі вище в класі RecommendatorImpl

немає публічного конструктора, об’єкт цього класу можна отримати тільки за допомогою статичного методу getInstance. Метод самостійно створює об’єкт якщо він ще не був створений, після чого зберігає його і повертає один і той же об’єкт кожного наступного разу. До того ж, щоб уникнути створення декількох об’єктів використовуються методи для синхронізації потоків, тобто частина коду, що створює новий об’єкт може виконуватись лише одним потоком в один момент часу. Разом з цим для зберігання створеного об’єкту використовується ключове слово volatile, що вказує віртуальній машині Java на те, що об’єкт не повинен кешуватись в стеці і доступ до об’єкту повинен відбуватись напряму.

Як вказувалось раніше, окрім складної реалізації, що потрібно реалізувати кожному класу, до якого розробник бажає застосувати концепцію Singleton ця реалізація є анти-шаблоном, оскільки порушує принцип єдиної відповідальності.

3.1.2 Пост конструктор класу

При розробці програмного забезпечення за класичним підходом часто доводиться писати певно логіку в конструкторі. Якщо цьому коду необхідно використовувати певне поле, то воно передається в конструкторі.

Однак, при розробці з використанням контейнера інверсії управління використання полів в конструкторі не завжди є можливим, так як, зазвичай, поля об’єкта ініціалізуються вже після створення об’єкта, а отже після запуску конструктора. Якщо об’єкт не буде ініціалізований до того як до нього буде викликаний, то код згенерує виключення NullPointerException і процес створення об’єкту і роботи контейнеру інверсії управління буде зупинено.

Тому з’явилась необхідність існування такого конструктора, що запускається після того як об’єкт буде повністю налаштований. Тому в нашій реалізації використовується анотація PostConstruct, що постачається як частина JDK. Згідно документації[1], анотація розміщується над методом, що повинен бути викликаний після впровадження залежностей і до того, як об’єкт буде впроваджений в інший компонент. Метод анотований PostConstruct повинен бути викликаний навіть якщо клас не потребує впровадження залежностей. При цьому

тільки один метод класу може мати цю анотацію. Також метод анотований цією анотацією має певні обмеження. Так метод не повинен повертати жодного значення, і бути статичним і не може генерувати checked виключення.

Виключення в Java це спеціальний об'єкт[4], що описує певну виключну ситуацію, що виникнула під час виконання коду. Об'єкт виключення створюється в момент виникнення виключної ситуації. Момент створення такого об'єкту називають генеруванням виключення. Виключення в Java можуть генеруватись автоматично, або вручну. Також виключення поділяються на checked та unchecked, тобто ті що можуть бути визначені на етапі компіляції та ті, що можуть бути визначені лише під час виконання програми, відповідно.

Анотація Java ніякої логіки в собі не має. Тому необхідно написати код, що буде знаходити дану анотацію і викликати метод анотований нею. Таким чином було визначено, що найкраще місце для цього коду є клас ApplicationContext. Причиною по якій цей функціонал не був реалізований у вигляді ще однієї імплементації інтерфейсу ObjectConfigurator є те, що PostConstruct метод повинен запускатись після того, як об'єкт буде налаштований, а всі залежності впровадженні. Код методу, що обробляє анотацію PostConstruct виглядає наступним чином:

```
private <T> void invokeInit(Class<T> implClass, T t) throws
IllegalAccessException, InvocationTargetException {
    for (Method method : implClass.getMethods()) {
        if
    (method.isAnnotationPresent(PostConstruct.class)) {
            method.invoke(t);
        }
    }
}
```

					КвРКІ 170137.17.01.05 ПЗ	Арк.
						44
Зм..	Арк.	№докум.	Підпис	Дата		

Логіка роботи методу є досить простою і схожою до роботи класів, що налаштовують об'єкти. За допомогою рефлексії знаходиться анотований метод і викликається. Метод є приватним і недоступний користувачам контейнера.

3.1.3 Налаштування об'єктів згідно конфігураційних файлів

Можливість мати декілька налаштованих середовищ для роботи програмного забезпечення дає великі можливості для розробників. Важливою є можливість змінювати середовище без зміни програмного коду[35, 36]. Тому в реалізованому програмному забезпечення є можливість впроваджувати залежність класу String зчитану з конфігураційного файлу з назвою «application.properties». Так в проекті можуть бути різні конфігураційні файли для розробників, тестувальників та для використання в робочому середовищі.

Так, не завжди є можливість використовувати одні і ті ж конфігурації в як в основному середовищі, та і в тестовому. Яскравим прикладом є налаштування зв'язку з БД. Використання в тестовому середовищі БД, що використовується як основна БД може спричинити втрату даних, тому можливість змінювати конфігурації є критично важливою.

Для того, щоб впровадити певне значення класу String в залежний об'єкт, достатньо написати анотацію InjectProperty і в ній написати ключ до значення, що повинно бути впроваджено.

3.1.4 Підтримка проксі об'єктів

Генерація проксі об'єктів надає широкі можливості для динамічного програмування певної логіки. Тому, в розробленому контейнері інверсії управління існує можливість створення генерації проксі-об'єктів засобами рефлексії. Компонент програми обгортається в проксі-об'єкт одразу після того як він був створений і повністю налаштований.

Можливість генерації проксі об'єктів надає безмежні можливості для розробників для реалізації та модифікації логіки роботи певних класів. Важливим

є те, що клас, що використовує залежний об'єкт може не знати, що насправді використовує проксі.

В мові програмування Java для того щоб створити проксі[14] для об'єкту певного класу, необхідно знати інтерфейси які він використовує. Таким чином відбувається підміна справжнього об'єкта іншим зі збереженням методів за допомогою яких цей клас працює.

3.2 Реалізація програмних компонентів

3.2.1 Фабрика для створення об'єктів

Під час проектування було визначено, що головною задачею контейнера для інверсії контролю є створення, налаштування та управління об'єктами. При реалізації компонента відповідального за створення об'єктів було застосовано шаблон проектування під назвою «Фабрика». Так було реалізовано клас під назвою ObjectFactory. Окрім безпосередньо створення об'єкта, він викликає методи для налаштування об'єктів, однак, саме налаштування делегується конфігураторам об'єктів. Пошук та створення конфігураторів відбувається у конструкторі класу ObjectFactory. Клас фабрики містить в собі лише один публічний метод з назвою createObject. Він приймає лише один параметр, а саме клас об'єкта, що потрібно створити. Отже, до відповідальностей класу ObjectFactory відноситься створення об'єкта, його налаштування за допомогою об'єктів конфігураторів, а також виклик пост конструктору класу. Окрім цього, перед тим як повернути створений об'єкт, фабрика викликає метод, що обгортає об'єкт в проксі-клас якщо це необхідно.

3.2.2 Конфігуратори об'єктів

Для налаштування об'єктів в програмі передбачений інтерфейс ObjectConfigurator. Інтерфейс має один метод, що приймає об'єкт для

конфігурації, а також об’єкт класу ApplicationContext. Нижче наведений код інтерфейсу.

```
public interface ObjectConfigurator {
    void configure(Object t,ApplicationContext
context);
}
```

Імплементацій даного інтерфейсу може бути безліч, в тому числі реалізовані кінцевим користувачем, що дає гнучкості в роботі з контейнером інверсії управління. Однією з імплементацій є клас InjectByTypeAnnotationObjectConfigurator. Його задача перевірити всі поля класу чи мають вони анотації InjectByType. Якщо клас містить дану анотацію це означає, що до цього класу повинно бути застосоване впровадження залежностей. Таким чином оброблюється анотація InjectByType, що дозволяє впроваджувати залежність, що буде знайдена за типом.

Іншим конфігуратором реалізованим в програмі є клас InjectPropertyAnnotationObjectConfigurator. Він відповідальний за впровадження залежностей, що є об’єктами класу String і зберігаються в конфігураційному файлі. Конфігураційний файл повинен бути побудований у вигляді ключ-значення, що розділені знаком «=». Клас, що використовує впровадження залежності з конфігураційного файлу повинен вказувати в анотації InjectByProperty ключ значення, що повинно бути впроваджене.

Окрім звичайних конфігураторів, в програмі існує інтерфейс ProxyConfigurator з методом з назвою replaceWithProxyIfNeeded, що описує логіку обгортання створених об’єктів в згенеровані проксі-класи за допомогою класів бібліотеки Cglib і виглядає наступним чином:

```
public interface ProxyConfigurator {
    Object replaceWithProxyIfNeeded(Object t, Class
implClass);
}
```

Так, однією з імплементацій цього інтерфейсу є клас DeprecatedHandlerProxyConfigurator, що створює проксі-об’єкт якщо клас

створеного об’єкту містить анотацію Deprecated. Deprecated це анотація, що використовується Java розробниками для того, щоб вказати на те, що метод, поле, або клас не рекомендується для використання і буде видалений в наступних версіях. При виклику будь-якого методу на проксі-об’єктів, в консоль буде виведено попередження про те, що метод застарів.

3.2.3 Компонент пошуку імплементації класу

Для того щоб впровадити певну залежність за типом, потрібно спочатку визначити імплементацію, яку необхідно створити і впровадити. Для цього був реалізований клас під назвою JavaConfig. Цей клас реалізу інтерфейс Config, що має лише 2 методи. Метод getScanner не приймає жодних аргументів і поверта об’єкт класу org.reflections.Reflections. Об’єкт цього класу забезпечує просунуті можливості рефлексії в Java і за допомогою нього можна робити пошук певних класів в пакеті.

Інший метод називається getImplClass. В параметрах він приймає клас, що є інтерфейсом, а повертає імплементацію цього інтерфейсу. Клас JavaConfig, який є реалізацією сканера пакетів, задля економії ресурсів не завжди використовує рефлексії для пошуку реалізацій інтерфейсу, тому що ця операція потребує певного часу. Тому для оптимізації роботи класу було прийнято рішення зберігати знайдені імплементації і робити сканування пакетів тільки якщо реалізація класу не знайдена в збережених об’єктах раніше.

Якщо знайдено більше ніж одна реалізація класу, то сканер згенерує виключення.

3.2.4 ApplicationContext

В попередніх пунктах були описані компоненти, які потребує контейнер інверсії управління для того, щоб мати змогу створювати та налаштовувати об’єкти. Тому є необхідність в класі, що керує всіма об’єктами і по суті є реалізацією контейнера для інверсії управління. Цей клас має назву

ApplicationContext. Він має в собі об’єкт класу ObjectFactory для створення об’єктів, і також об’єкт класу, що реалізує інтерфейс Config. Так він має лише один публічний метод з назвою getObject. В параметрах він приймає клас, і повертає об’єкт цього класу. Якщо це звичайний компонент, то він буде створений і налаштований як новий об’єкт за замовчуванням. Однак, в програмній реалізації передбачена можливість існування анотації Singleton. Ця анотація вказує контейнере на те, що для цього класу повинен існувати лише один об’єкт. Тому ApplicationContext буде кешувати такі об’єкти після створення.

Блок-схема роботи методу getObject розробленого класу ApplicationContext зображено на рисунку 3.1.

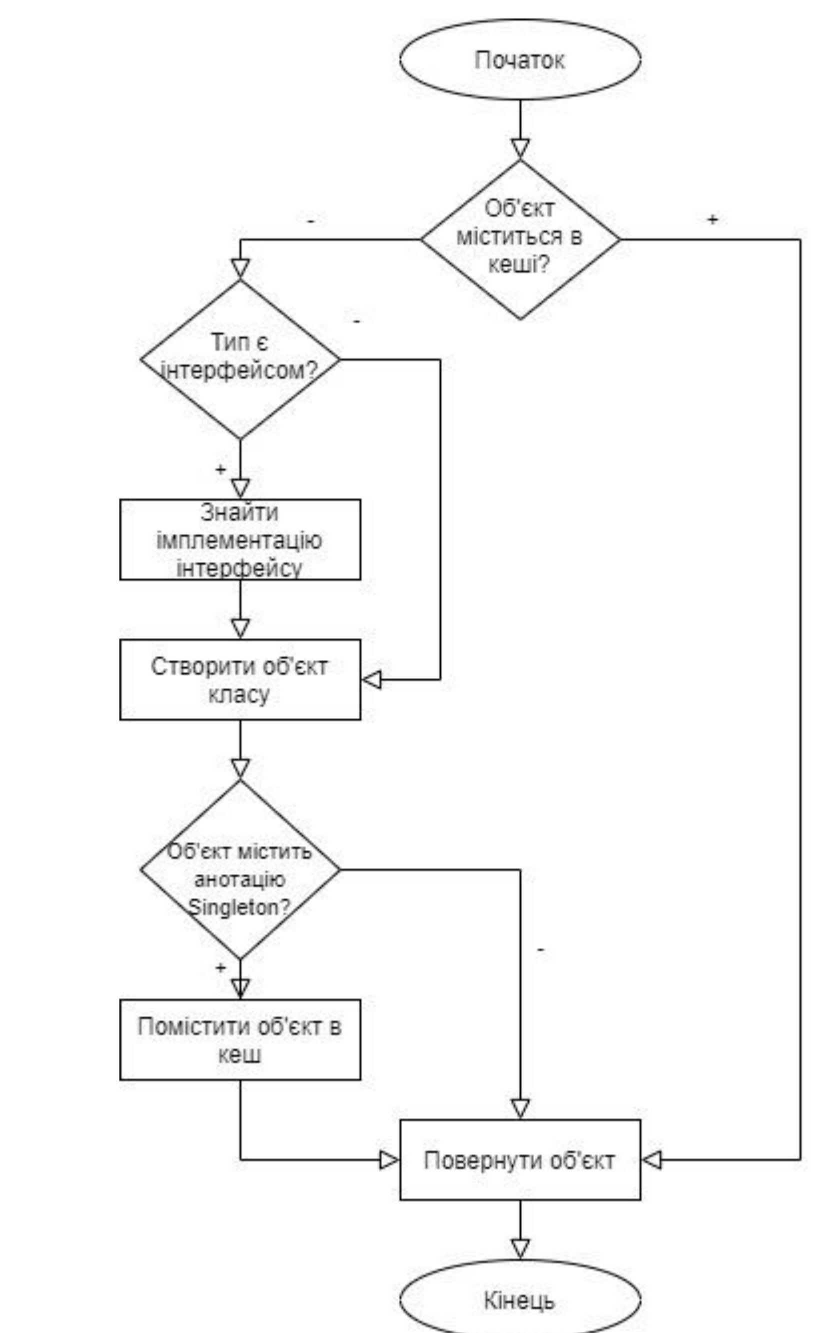


Рисунок 3.1 – Блок-схема роботи методу getObject класу ApplicationContext.

Таким чином, ApplicationContext відповідальний за управління усіма компонентами програми.

3.2.5 Компонент для запуску роботи контейнеру інверсії управління

Як було зазначено раніше, ApplicationContext і є реалізацією контейнера інверсії управління. Тому, задля збереження принципу єдиної відповідальності був створений клас з назвою Application з одним статичним методом run.

Особливість статичних методів в Java в тому, що для їх виклику не потрібно створювати об’єкт класу, вони не використовують не статичні поля чи методи класу і не мають стану, на відміну від об’єктів. Метод run приймає пакет для сканування класів і створює об’єкт класу ApplicationContext, після чого повертає його.

3.3 Розробка програмного забезпечення на базі реалізованого рішення для інверсії управління

3.3.1 Опис тестової програми

В процесі розробки було прийнято рішення протестувати готове рішення та розробити певне програмне забезпечення з використанням інверсії контролю на базі реалізованого програмного рішення.

Так була написана тестова програма, що імітує процес дезінфекції певної кімнати. Для цього був реалізований клас Room, що виступає в ролі кімнати для очистки. Клас Room не має в собі полів чи методів.

Для відображення процесу дезінфекції був реалізований клас з назвою Desinfector. Він має в собі один публічний метод з назвою start, що приймає в параметрах об’єкт типу Room. Також клас Desinfector має поля інтерфейсу Policeman та Announcer. Ці об’єкти цих класів використовуються в методі start. Нижче наведений код класу Desinfector.

```
@Setter
public class Desinfector {

    @InjectByType
    private Announcer announcer;
    @InjectByType
    private Policeman policeman;

    public void start(Room room) {
```

```
            announcer.announce("Starting disinfection, 
leave the room!");
            policeman.makePeopleLeaveRoom();
            desinfect(room);
            announcer.announce("You can go inside");
        }
        private void desinfect(Room room){
            System.out.println("disinfection......");
        }
    }
```

Необхідно звернути увагу, що над полями Policeman та Announcer присутня анотація InjectByType. В даному випадку вона слугує метаданими, що вказують контейнеру інверсії управління на те, що в клас повинна бути впроваджена залежність цього типу. Так обробка цієї анотації лежить на імплементації інтерфейсу ObjectConfigurator з назвою InjectByTypeAnnotationObjectConfigurator.

```
    public class InjectByTypeAnnotationObjectConfigurator 
implements ObjectConfigurator {
        @Override
        @SneakyThrows
        public void configure(Object t, ApplicationContext 
context) {
            for (Field field : 
t.getClass().getDeclaredFields()) {
                if 
(field.isAnnotationPresent(InjectByType.class)) {
                    field.setAccessible(true);
                    Object object = 
context.getObject(field.getType());
                    field.set(t, object);
                }
            }
```

```
        }
    }
```

Як видно з коду наведеного вище, клас InjectByTypeAnnotationObjectConfigurator має один метод, що оголошений в інтерфейсі. Так цей метод приймає певний об’єкт, що потрібно налаштувати. В методі, за допомогою рефлексії, запускається цикл по полям класу і для кожного поля, що містить анотацію InjectByType встановлює об’єкт в це поле, що надає об’єкт класу ApplicationContext.

Такий підхід дозволяє легко створювати інші метадані з певною логікою, з логікою що займається обробкою цих метаданих.

Також, до переваг цієї архітектури можна віднести можливість конфігурації контейнера під себе для кінцевого користувача. Реалізацій інтерфейсу ObjectConfigurator може бути необмежена кількість.

3.3.2 Результати роботи тестової програми

Результати роботи програми виводились в консоль.

```
C:\Users\user\.jdks\adopt-openjdk-11.0.11\bin\java.exe "-javaagent:C:\Program
Recommendator object was created
class com.khnu.RecommendatorImpl
Starting disinfection, leave the room!
To protect from virus take vitamin A
Police. Leave the room!
disinfection......
You can go inside
To protect from virus take vitamin A

Process finished with exit code 0
```

Рисунок 3.2 – Результат роботи тестової програми.

Як видно на скріншоті(рис. 3.2), в першій стрічці пише, що об’єкт, що імплементує інтерфейс Recommendator був створений. Код, що виводить цю стрічку на екран знаходиться в конструкторі класу RecommendatorImpl і виглядає наступним чином:

```
public RecommendatorImpl() {
    System.out.println("Recommendator object was
created");
}
```

Наступна стрічка вказує об’єкт якого саме класу був створений. Цей код знаходиться в класі PolicemanImpl. В методі init, що має анотацію PostConstruct і наведений нижче.

```
@PostConstruct
public void init() {
    System.out.println(recommendator.getClass());
}
```

В цьому випадку використання анотації PostConstruct є дуже показовим, адже якби код наведений в методі init знаходився в конструкторі класу PolicemanImpl, то після запуску програми неодмінно з’явиться помилка під час виконання програми, тому що в момент виклику конструктору, поле recommendator ще не ініціалізовано, тому в цьому випадку необхідно використовувати пост конструктор.

Ще однією деталлю є використання проксі об’єктів в програмі. Так, якщо додати анотацію Deprecated над класом Desinfector, результат роботи(рис. 10) буде відрізнятись від попереднього.

```
C:\Users\user\.jdks\adopt-openjdk-11.0.11\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.2.3\lib\idea_rt.ja
Recommendator object was created
class com.khnu.RecommendatorImpl
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by net.sf.cglib.core.ReflectUtils$1 (file:/C:/Users/user/.m2/repository/cglib/cglib/3.3.0/cglib-
WARNING: Please consider reporting this to the maintainers of net.sf.cglib.core.ReflectUtils$1
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
This method is deprecated!!!
Starting disinfection, leave the room!
To protect from virus take vitamin A
Police. Leave the room!
disinfection......
You can go inside
To protect from virus take vitamin A

Process finished with exit code 0
```

Рисунок 3.3 – Результат роботи програми, що містить анотацію Deprecated.

Так видно, що в консолі з’явились повідомлення від віртуальної машини Java про використання рефлексії за допомогою класу бібліотеки Cglib. Але, окрім цього в виводі програми з’явилось повідомлення «This method is deprecated!!!», якого не було до внесення змін в клас Desinfector. Це сталось через те, що в контейнері для інверсії управління існує об’єкт класу, що створює проксі об’єкти для класів, що містять в собі анотацію Deprecated і виводить повідомлення в консоль кожен раз, коли метод цього об’єкту викликається.

Звичайно, можливості використання проксі-об’єктів є набагато ширші, ніж вивід повідомлення в консоль, однак механізм роботи контейнера інверсії управління від цього не зміниться.

3.4 Порівняння коду з та без використанням контейнеру для інверсії управління.

Раніше було показано як просто реалізовувати відомі шаблони проектування за допомогою інверсії управління через впровадження залежностей. Можливо не кожен код використовує відомі шаблони проектування, але кожен застосунок написаний за допомогою об’єктно-орієнтованої мови програмування повинен створювати об’єкти в програмі. Ось як виглядав би певний код без застосування інверсії управління:

```
public static void main(String[] args){
```

```
    Desinfector desinfector =
ObjectFactory.getInstance().createObject(Desinfector.class)
;
        Announcer announcer =
ObjectFactory.getInstance().createObject(ConsoleAnnouncer.c
lass);
        Policeman policeman =
ObjectFactory.getInstance().createObject(PolicemanImpl.clas
s);
        desinfector.setAnnouncer(announcer);
        desinfector.setPoliceman(policeman);
        desinfector.start(new Room());
      }
```

Так видно, що для того, щоб викликати метод класу Desinfector необхідно створити об'єкт цього класу за допомогою класу-фабрики і всі об'єкти, необхідні для роботи цього класу. Після цього потрібно самостійно передати створені об'єкти і лише тоді можна викликати потрібний метод. Так дуже легко зробити помилку, наприклад, забути проініціалізувати певне поле класу, чи правильно налаштувати певний об'єкт, особливо під час рефакторингу коду. До прикладу, ось код, що робить те саме, але вже з застосуванням реалізованої інверсії управління:

```
      public static void main(String[] args) {
        ApplicationContext context =
Application.run("com.khnu", new
HashMap<>(Map.of(Policeman.class, PolicemanImpl.class)));
        Desinfector desinfector =
context.getObject(Desinfector.class);
        desinfector.start(new Room());
      }
```

Так можна лише за допомогою однієї стрічки коду запустити контейнер інверсії управління і використовувати об'єкти, що він створює і налаштовує. До

того ж, не вказується яку реалізація необхідно, програма визначає це самостійно . Таким чином відповідальність за створення, налаштування і управління об'єктами переноситься на зовнішню бібліотеку. Це дає змогу розробникам не витрачати час на розробку власних інфраструктурних рішень, а більше зосередитись на реалізації бізнес-логіки розроблюваного програмного забезпечення. Отже, за допомогою створеного програмного засобу, є можливість створити слабкозв'язне програмне забезпечення[44], що додає гнучкості та швидкості під час розробки. Коли розміри програми досягають десятків тисяч класів, інверсія контролю показує максимальну користь.

3.5 Способи покращення програмної реалізації

Створене програмне забезпечення реалізоване згідно найкращих практик і з застосуванням відомих шаблонів проектування, тому функціонал програми досить легко і зручно можна розширити.

Я вважаю, що одним з способів покращення реалізованого контейнеру для інверсії контролю є розширення можливостей для налаштування та управління об'єктами. На даному етапі контейнер вміє створювати об'єкти, впроваджувати залежності за типом, створювати проксі об'єкти компонентів контейнеру, а також забезпечує реалізацію концепції Singleton.

На мою думку досить корисним було б мати можливість конфігурувати об'єкти класу, що дотримуються концепції Singleton таким чином, щоб цей об'єкт створювався не відразу, а тільки за умови якщо до нього звернеться інший об'єкт. Це може бути корисно для класів таких об'єктів, що потребують багато часу та ресурсів на створення, хоча можуть не знадобитись на протязі життєвого циклу програми.

Такий підхід, коли створення об'єкта відбувається безпосередньо перед тим як виконати якусь операцію щодо нього, називається Відкладена(лінива) ініціалізація(англ. Lazy initialization).

Цей прийом часто використовують у великих проектах, у випадках, коли існування певного об'єкту є ресурсовитратним. До таких прикладів можна

віднести зв’язок з БД. Об’єкт для створення зв’язку з БД створюється певний час і потребує певного об’єму оперативної пам’яті. Зазвичай в проекті постійно використовують одну БД, але існують випадки, коли проект побудований таким чином, щоб старі дані, що майже не використовуються зберігалися в окремій БД, таким чином підключення до цією БД може не відбутися за час життєвого циклу програми.

Завдяки архітектурі створеної програми, реалізація додаткової опції не повинна бути складною, так як у кінцевого користувача є можливість створити ще одну імплементацію інтерфейсу ProxyConfigurator, що відповідальний за динамічну генерацію класів, яка буде передбачати логіку для відкладеної ініціалізації полів компонента контейнера, що містять певні метадані.

3.6 Висновки

В даному розділі було проведено процес розробки програмного забезпечення, що реалізує принципи інверсії контролю через впровадження залежностей і показані його результати. Описані реалізовані функціональні можливості розробленого програмного забезпечення та їх спосіб роботи.

Була розроблена тестова програма, що написана на реалізованому рішенні для застосування інверсії контролю, показаний результат її роботи, а також порівняно код, що виконує запуск тестової програми за класичним способом програмування та використанням інверсії контролю. У висновку, програма написана з використанням інверсії контролю потребує набагато менше коду і зусиль для запуску, ніж без впровадження залежностей.

Були описана реалізація основних компонентів програми, що були спроектовані в другому розділі, особливості їх роботи, та наведені приклади коду.

Розробка відбувалась в середовищі IntelliJ IDEA, в процесі використовувались наступні технології:

1. Java 11.
2. Maven.
3. Lombok.

4. Byte Code Generation Library.
5. Reflection Library.

ВИСНОВКИ

Після теоретичних та практичних досліджень було визначено одну з проблем, що виникають при розробці програмного забезпечення за допомогою об’єктно-орієнтованих мов програмування та запропоновані шляхи її вирішення.

В першому розділі був зроблений аналіз проблеми, можливі шляхи її вирішення, а також існуючі рішення та принцип їх роботи. Був досліджений принцип інверсії контролю та здійснена постановка задачі.

В другому розділі було визначено вимоги та принципи роботи системи. Було зроблене проектування компонентів програми. Проведено порівняння технологій і визначено набір технологій які потрібні для вирішення поставленої задачі.

В третьому розділі було розроблено програмне забезпечення, що реалізує принципи інверсії контролю через впровадження залежностей. В процесі розробки були використані технології визначені в другому розділі, а також застосовані популярні шаблони проектування.

В розробленому програмному забезпеченні, окрім впровадження залежностей, були реалізовані так можливості, як підтримка шаблону проектування Singleton за допомогою Java анотацій. Важливою є реалізація підтримки пост конструктора класу, що викликається після всіх налаштувань об’єкту, адже при використанні інверсії контролю, створення конструкторів з певною логікою неможливе через особливість реалізації впровадження залежностей об’єкту. Також до реалізованих можливостей відноситься можливість створення проксі об’єктів для будь якого компоненту, до якого застосовується інверсія контролю, що надає широкі можливості для програмування динамічної генерації класів.

Також було проведене порівняння розробки певного коду з використанням розробленого рішення для інверсії контролю через впровадження залежностей та без нього. Разом з тим, було запропоновано шляхи покращення розробленого програмного забезпечення.

В результаті роботи було глибше досліджено особливості об’єктно-орієнтованого програмування та розроблено засіб для реалізації принципу інверсії контролю. Інверсія управління може застосовуватись в різних сферах, наприклад, в системному програмуванні.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Annotation Type PostConstruct. URL: https://docs.oracle.com/javaee/5/api/javax/annotation/PostConstruct.html.

2. Bharathan R. Apache Maven Cookbook. 2015. 272 с.

3. Bipin J. Beginning SOLID Principles and Design Patterns for ASP.NET Developers. 2016. 399 с.

4. Bloch J. Effective Java. 3rd Edition, 2018. 392с.

5. Brian Vermeer. 36% of developers switched from Oracle JDK to an alternate OpenJDK distribution, over the last year. 2020. URL: https://snyk.io/blog/36-of-developers-switched-from-oracle-jdk-to-an-alternate-openjdk-distribution-over-the-last-year/.

6. Brian Vermeer. 64% of developers report that Java 8 remains the most often used release. 2020. URL: https://snyk.io/blog/developers-dont-want-to-leave-java-8-as-64-hold-firm-on-their-preferred-release/.

7. Buschmann F., Henney K., Schmidt D. Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, 2007. 636 с.

8. Cosmina I., Ho C., Harrop R., Schaefer C. Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools. 2017. 849с.

9. Dahl O. The Birth of Object Orientation: the Simula Languages. 2004. 388 с.

10. Darwin I. Java Cookbook: Solutions and Examples for Java Developers 3rd Edition. 2014. 898с.

11. Dependency Injection != using a DI container. 2011. URL: Dependency Injection != using a DI container.

12. Dependency Injection in Spring. URL: https://www.javatpoint.com/dependency-injection-in-spring.

13. Design Patterns: Elements of Reusable Object-Oriented Software / E.Gamma, R. Helm, R. Johnson, J. Vlissides., 1995. – 395 с.

14. Dynamic Proxy Classes. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html.

15. Fowler M. Inversion of Control Containers and the Dependency Injection pattern. 2004. URL: https://martinfowler.com/articles/injection.html.

16. Fowler M. Refactoring: Improving the Design of Existing Code (2nd Edition). 2018. 448 c.

17. Freemann E., Bates B., Sierra K., Robson E. Head First Design Patterns: A Brain-Friendly Guide. 2014. 694 c.

18. Freemann E., Robson E. Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software 2nd Edition. 2020. 672 c.

19. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. 2006. 432 c.

20. Gradle User Manual. URL: https://docs.gradle.org/current/userguide/userguide.html.

21. Guide to Java Reflection. 2021. URL: https://www.baeldung.com/java-reflection.

22. Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. 2003. 736 c.

23. Horstmann Cay S. Core Java Volume I-Fundamentals (10th Edition). 2016. 1008 c.

24. How to write testable code. 2011. URL: https://www.loosecouplings.com/2011/01/how-to-write-testable-code-overview.html.

25. Introduction to cglib. 2019 URL: https://www.baeldung.com/cglib.

26. Introduction to the Spring IoC container and beans. URL: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html.

27. Inversion of Control. URL: https://www.tutorialsteacher.com/ioc/inversion-of-control.

28. IoC Container. URL: https://www.tutorialsteacher.com/ioc/ioc-container.

29. Kimberlin M. Reducing boilerplate code with Project Lombok. 2010. URL: https://objectcomputing.com/resources/publications/sett/january-2010-reducing-boilerplate-code-with-project-lombok.

30. Kumar A. Understanding Inversion of Control (IoC) Principle. 2019. URL: https://medium.com/@amitkma/understanding-inversion-of-control-ioc-principle-163b1dc97454.

31. Loy M., Niemeyer P., Leuck D. Learning Java, 4th Edition. 2020. 520 c.

32. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. 2018. 432 c.

33. Martin R. Clean Book. A Handbook of Agile Software Craftsmanship. 2008. 464c

34. Martin R. Dependency Injection Inversion. 2010. URL: https://sites.google.com/site/unclebobconsultingllc/blogs-by-robert-martin/dependency-injection-inversion.

35. Martin R. The Dependency Inversion principle. https://web.archive.org/web/20041221102842/http://www.objectmentor.com/resources/articles/dip.pdf.

36. Mazzocchi S. On Inversion of Control. 2004. URL: https://web.archive.org/web/20040202120126/http://www.betaversion.org/~stefano/linotype/news/38/.

37. McCluskey G. Using Java Reflection. 1998. URL: https://www.oracle.com/technical-resources/articles/java/javareflection.html.

38. McConnell S. Code Complete. 2004. 960c.

39. McLaughlin B., Pollice G., West D. Head First Object-Oriented Analysis and Design. 2006. 636 c.

40. Naftalin M., Wadler P. Java Generics and Collections. 2006. 294 c.

41. Pawlak R., Seinturier L. Foundations of AOP for J2EE Development. 2005. 328 c.

42. Rubio D. Pro Spring Dynamic Modules for OSGi Service Platforms. 392c.

43. Schild H. Java: The Complete Reference, Tenth Edition. 2017. 1344 c.

44. Seeman M. Dependency Injection is Loose Coupling. 2010. URL: https://blog.ploeh.dk/2010/04/07/DependencyInjectionisLooseCoupling/.

45. Sonatype Company. Maven: The Definitive Guide: The Definitive Guide. 2008. 470 с.

46. Spring – Inversion of Control vs Dependency Injection. URL: https://howtodoinjava.com/spring-core/spring-ioc-vs-di/.

47. Turnquist G. Learning Spring Boot 2.0 - Second Edition: Simplify the development of lightning fast applications based on microservices and reactive programming. 2017. 370 с.

48. Urma R., Fusco M., Mycroft A. Modern Java in Action: Lambdas, streams, functional and reactive programming. 2018. 592 с.

49. Walls C. Spring in Action. 2018. 520 с.

50. Weisfeld M. The Object-Oriented Thought Process. 2013. 336 с.

Додаток А

(обов’язковий)

Лістинг програмного код фреймворку

Файл Application.java

```
package com.khnu;

import java.util.Map;

public class Application {

    public static ApplicationContext run(String packageToScan, Map<Class,
Class> ifc2ImplClass) {

        JavaConfig config = new JavaConfig(packageToScan, ifc2ImplClass);

        ApplicationContext context = new ApplicationContext(config);

        ObjectFactory objectFactory = new ObjectFactory(context);

        context.setFactory(objectFactory);

        return context;

    }

}
```

Файл ApplicationContext.java

```
package com.khnu;

import lombok.Getter;

import lombok.Setter;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

public class ApplicationContext {

    @Setter

    private ObjectFactory factory;

    private Map<Class, Object> cache = new ConcurrentHashMap<>();

    @Getter

    private Config config;


    public ApplicationContext(Config config) {

        this.config = config;
```

```
    }
    public <T> T getObject(Class<T> type) {
        if (cache.containsKey(type)) {
            return (T) cache.get(type);
        }
        Class<? extends T> implClass = type;
        if (type.isInterface()) {
            implClass = config.getImplClass(type);
        }
        T t = factory.createObject(implClass);

        if (implClass.isAnnotationPresent(Singleton.class)) {
            cache.put(type, t);
        }
        return t;
    }
}
```

Клас Config.java

```
package com.khnu;
import org.reflections.Reflections;

public interface Config {
    <T> Class<? extends T> getImplClass(Class<T> ifc);
    Reflections getScanner();
}
```

Файл DeprecatedHandlerProxyConfigurator.java

```
package com.khnu;
import net.sf.cglib.proxy.Enhancer;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
public class DeprecatedHandlerProxyConfigurator implements ProxyConfigurator
{
```

```
    @Override
    public Object replaceWithProxyIfNeeded(Object t, Class implClass) {
        if (implClass.isAnnotationPresent(Deprecated.class)) {


            if (implClass.getInterfaces().length == 0) {
                return Enhancer.create(implClass, new
net.sf.cglib.proxy.InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
                        return getInvocationHandlerLogic(method, args, t);
                    }
                });
            }
            return Proxy.newProxyInstance(implClass.getClassLoader(),
implClass.getInterfaces(), new InvocationHandler() {
                @Override
                public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                    return getInvocationHandlerLogic(method, args, t);
                }
            });
        } else {
            return t;
        }
    }
    private Object getInvocationHandlerLogic(Method method, Object[] args,
Object t) throws IllegalAccessException, InvocationTargetException {
        System.out.println("This method is deprecated!!!");
        return method.invoke(t, args);
    }
}
```

Файл InjectByType.java

```
package com.khnu;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
@Retention(RUNTIME)

public @interface InjectByType {

}
```

Файл InjectByTypeAnnotationObjectConfigurator.java

```
package com.khnu;

import lombok.SneakyThrows;

import java.lang.reflect.Field;

public class InjectByTypeAnnotationObjectConfigurator implements
ObjectConfigurator {

    @Override

    @SneakyThrows

    public void configure(Object t, ApplicationContext context) {

        for (Field field : t.getClass().getDeclaredFields()) {

            if (field.isAnnotationPresent(InjectByType.class)) {

                field.setAccessible(true);

                Object object = context.getObject(field.getType());

                field.set(t, object);

            }

        }

    }

}
```

Клас InjectProperty.java

```
package com.khnu;

import java.lang.annotation.Retention;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Retention(RUNTIME)

public @interface InjectProperty {

    String value() default "";

}
```

Файл InjectPropertyAnnotationObjectConfigurator.java

```
package com.khnu;


import lombok.SneakyThrows;
```

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.lang.reflect.Field;

import java.util.Map;

import java.util.stream.Stream;


import static java.util.stream.Collectors.toMap;


public class InjectPropertyAnnotationObjectConfigurator implements
ObjectConfigurator {


    private Map<String, String> propertiesMap;


    @SneakyThrows

    public InjectPropertyAnnotationObjectConfigurator() {

        String path =
ClassLoader.getSystemClassLoader().getResource("application.properties").getP
ath();

        Stream<String> lines = new BufferedReader(new
FileReader(path)).lines();

        propertiesMap = lines.map(line -> line.split("=")).collect(toMap(arr
-> arr[0], arr -> arr[1]));


    }


    @Override

    @SneakyThrows

    public void configure(Object t,ApplicationContext context) {

        Class<?> implClass = t.getClass();

        for (Field field : implClass.getDeclaredFields()) {

            InjectProperty annotation =
field.getAnnotation(InjectProperty.class);

           if (annotation != null) {

                String value = annotation.value().isEmpty() ?
propertiesMap.get(field.getName()) : propertiesMap.get(annotation.value());

                field.setAccessible(true);

                field.set(t,value);
```

```
            }
        }
    }
}
```

Файл JavaConfig.java

```
package com.khnu;
import lombok.Getter;
import org.reflections.Reflections;
import java.util.Map;
import java.util.Set;
public class JavaConfig implements Config {
    @Getter
    private Reflections scanner;
    private Map<Class, Class> ifc2ImplClass;
    public JavaConfig(String packageToScan, Map<Class, Class> ifc2ImplClass)
{
        this.ifc2ImplClass = ifc2ImplClass;
        this.scanner = new Reflections(packageToScan);
    }
    @Override
    public <T> Class<? extends T> getImplClass(Class<T> ifc) {
        return ifc2ImplClass.computeIfAbsent(ifc, aClass -> {
            Set<Class<? extends T>> classes = scanner.getSubTypesOf(ifc);
            if (classes.size() != 1) {
                throw new RuntimeException(ifc + " has 0 or more than one
impl please update your config");
            }

            return classes.iterator().next();
        });
    }
}
```

Файл ObjectConfigurator.java

```
package com.khnu;
```

```
public interface ObjectConfigurator {
    void configure(Object t,ApplicationContext context);
}
```

Файл ObjectFactory.java

```
package com.khnu;
import lombok.SneakyThrows;
import javax.annotation.PostConstruct;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;
public class ObjectFactory {
    private final ApplicationContext context;
    private List<ObjectConfigurator> configurators = new ArrayList<>();
    private List<ProxyConfigurator> proxyConfigurators = new ArrayList<>();



    @SneakyThrows
    public ObjectFactory(ApplicationContext context) {
        this.context= context;
        for (Class<? extends ObjectConfigurator> aClass :
context.getConfig().getScanner().getSubTypesOf(ObjectConfigurator.class)) {
            configurators.add(aClass.getDeclaredConstructor().newInstance());
        }
        for (Class<? extends ProxyConfigurator> aClass :
context.getConfig().getScanner().getSubTypesOf(ProxyConfigurator.class)) {

proxyConfigurators.add(aClass.getDeclaredConstructor().newInstance());
        }
    }
    @SneakyThrows
    public <T> T createObject(Class<T> implClass) {
        T t = create(implClass);
        configure(t);
```

```
        invokeInit(implClass, t);

        t = wrapWithProxyIfNeeded(implClass, t);

        return t;

    }

    private <T> T wrapWithProxyIfNeeded(Class<T> implClass, T t) {

        for (ProxyConfigurator proxyConfigurator : proxyConfigurators) {

            t = (T) proxyConfigurator.replaceWithProxyIfNeeded(t, implClass);

        }

        return t;

    }

    private <T> void invokeInit(Class<T> implClass, T t) throws
IllegalAccessException, InvocationTargetException {

        for (Method method : implClass.getMethods()) {

            if (method.isAnnotationPresent(PostConstruct.class)) {

                method.invoke(t);

            }

        }

    }


    private <T> void configure(T t) {

        configurators.forEach(objectConfigurator ->
objectConfigurator.configure(t,context));

    }


    private <T> T create(Class<T> implClass) throws InstantiationException,
IllegalAccessException, java.lang.reflect.InvocationTargetException,
NoSuchMethodException {

        return implClass.getDeclaredConstructor().newInstance();

    }

}
```

Файл ProxyConfigurator.java

```
package com.khnu;


public interface ProxyConfigurator {

    Object replaceWithProxyIfNeeded(Object t, Class implClass);

}
```

Файл Singleton.java

```
package com.khnu;

import java.lang.annotation.Retention;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Retention(RUNTIME)

public @interface Singleton {

}
```

Файл 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>com.epam</groupId>

    <artifactId>corona-desinfector-life-demo</artifactId>

    <version>1.0-SNAPSHOT</version>


    <dependencies>


        <dependency>

            <groupId>cglib</groupId>

            <artifactId>cglib</artifactId>

            <version>3.3.0</version>

        </dependency>

        <dependency>

            <groupId>org.reflections</groupId>

            <artifactId>reflections</artifactId>

            <version>0.9.12</version>

        </dependency>

        <dependency>

            <groupId>org.projectlombok</groupId>

            <artifactId>lombok</artifactId>
```

```
            <version>1.18.10</version>
        </dependency>

        <dependency>
            <groupId>javax.annotation</groupId>
            <artifactId>jsr250-api</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
        </plugins>
    </build>


</project>
```

Додаток Б

(обов’язковий)

Лістинг коду програми розробленої на базі фреймворку

Файл AngryPoliceman.java

```
package com.khnu;
public class AngryPoliceman implements Policeman {
    @Override
    public void makePeopleLeaveRoom() {
        System.out.println("Get out from there");
    }
}
```

Файл Announcer.java

```
package com.khnu;
public interface Announcer {
    void announce(String message);
}
```

Файл ConsoleAnnouncer.java

```
package com.khnu;
public class ConsoleAnnouncer implements Announcer {
    @InjectByType
    private Recommendator recommendator;
    @Override
    public void announce(String message) {
        System.out.println(message);
        recommendator.recommend();
    }
}
```

Файл Main.java

```
package com.khnu;
import java.util.HashMap;
import java.util.Map;
```

```java
public class Main {

    public static void main(String[] args) {

        ApplicationContext context = Application.run("com.khnu", new
HashMap<>(Map.of(Policeman.class, PolicemanImpl.class)));

        Desinfector desinfector = context.getObject(Desinfector.class);

        desinfector.start(new Room());

    }

}
```

Файл Policeman.java

```java
package com.khnu;


public interface Policeman {

    void makePeopleLeaveRoom();

}
```

Файл PolicemanImpl.java

```java
package com.khnu;

import javax.annotation.PostConstruct;

public class PolicemanImpl implements Policeman {

    @InjectByType

    private Recommendator recommendator;

    @PostConstruct

    public void init() {

        System.out.println(recommendator.getClass());

    }

    @Override

    public void makePeopleLeaveRoom() {

        System.out.println("Police. Leave the room!");

    }

}
```

Файл Recommendator.java

```java
package com.khnu;

public interface Recommendator {

    void recommend();

}
```

Файл RecommendatorImpl.java

```
package com.khnu;
@Singleton
public class RecommendatorImpl implements Recommendator {
    @InjectProperty("vitamin")
    private String vitamin;

    public RecommendatorImpl() {
        System.out.println("Recommendator object was created");
    }
    @Override
    public void recommend() {
        System.out.println("To protect from virus take vitamin "+ vitamin);
    }
}
```

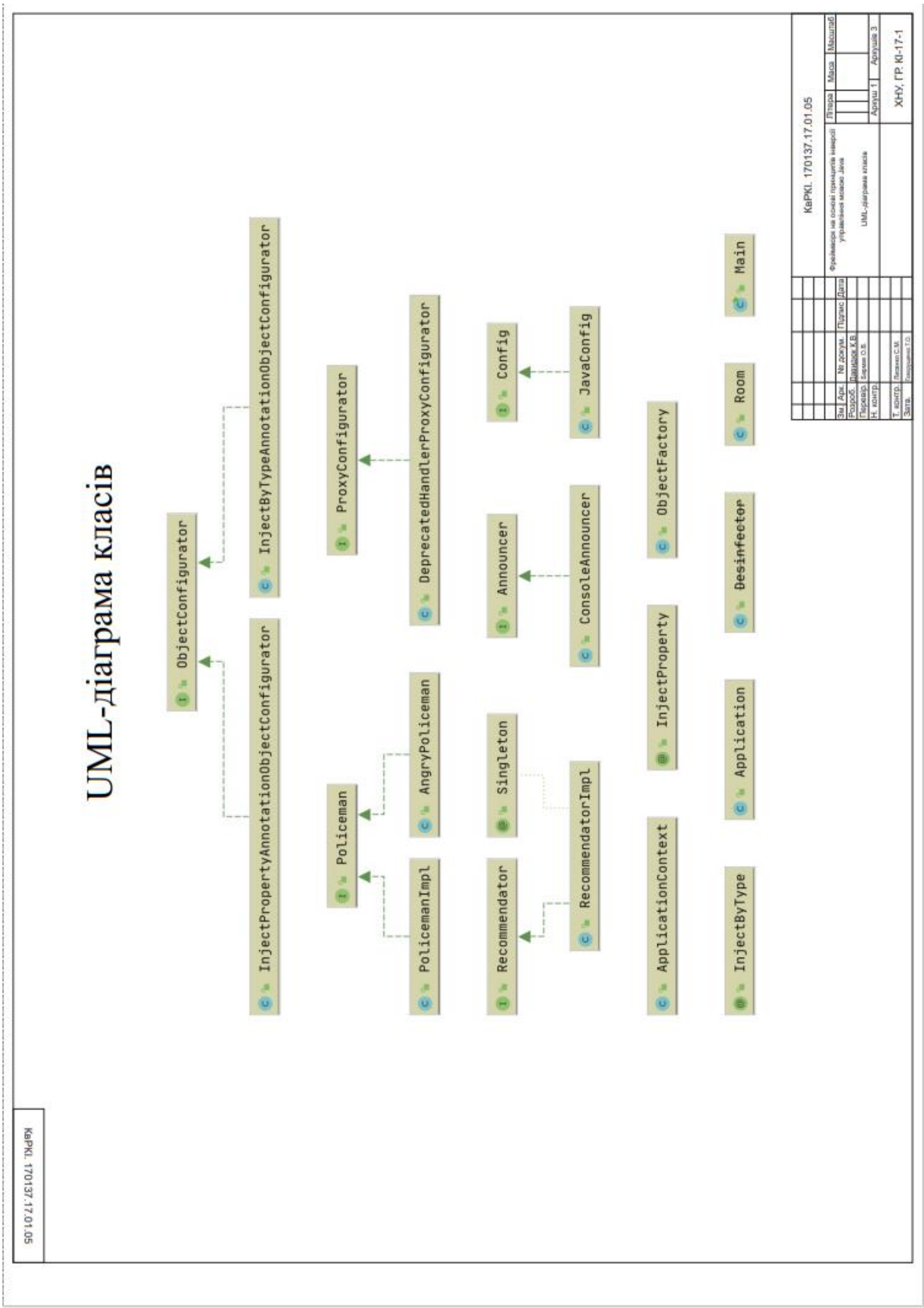
Файл Room.java

```
package com.khnu;
public class Room {
}
```

Додаток В

(обов’язковий)

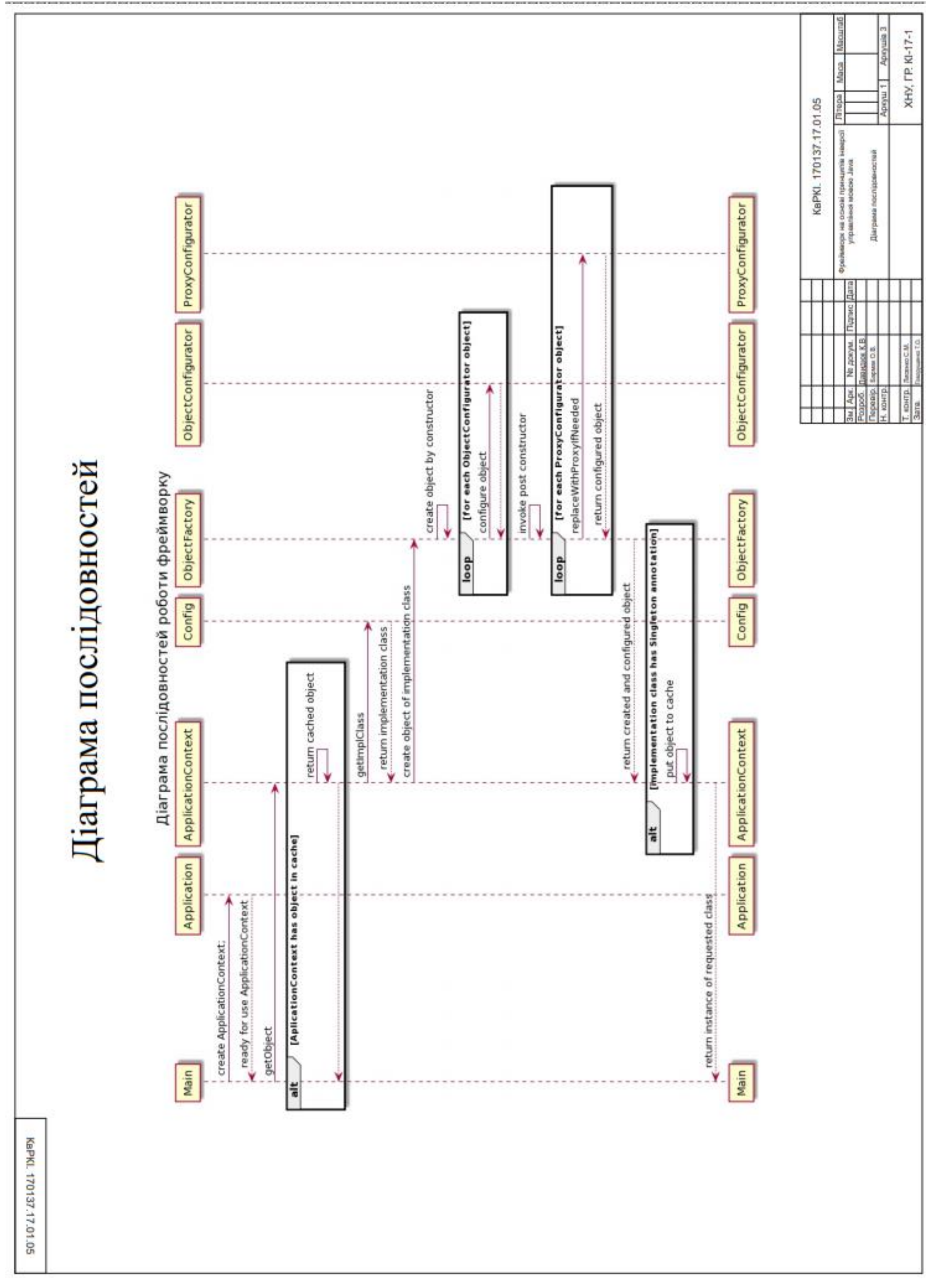
Копія креслення «UML діаграма класів»



Додаток Г

(обов'язковий)

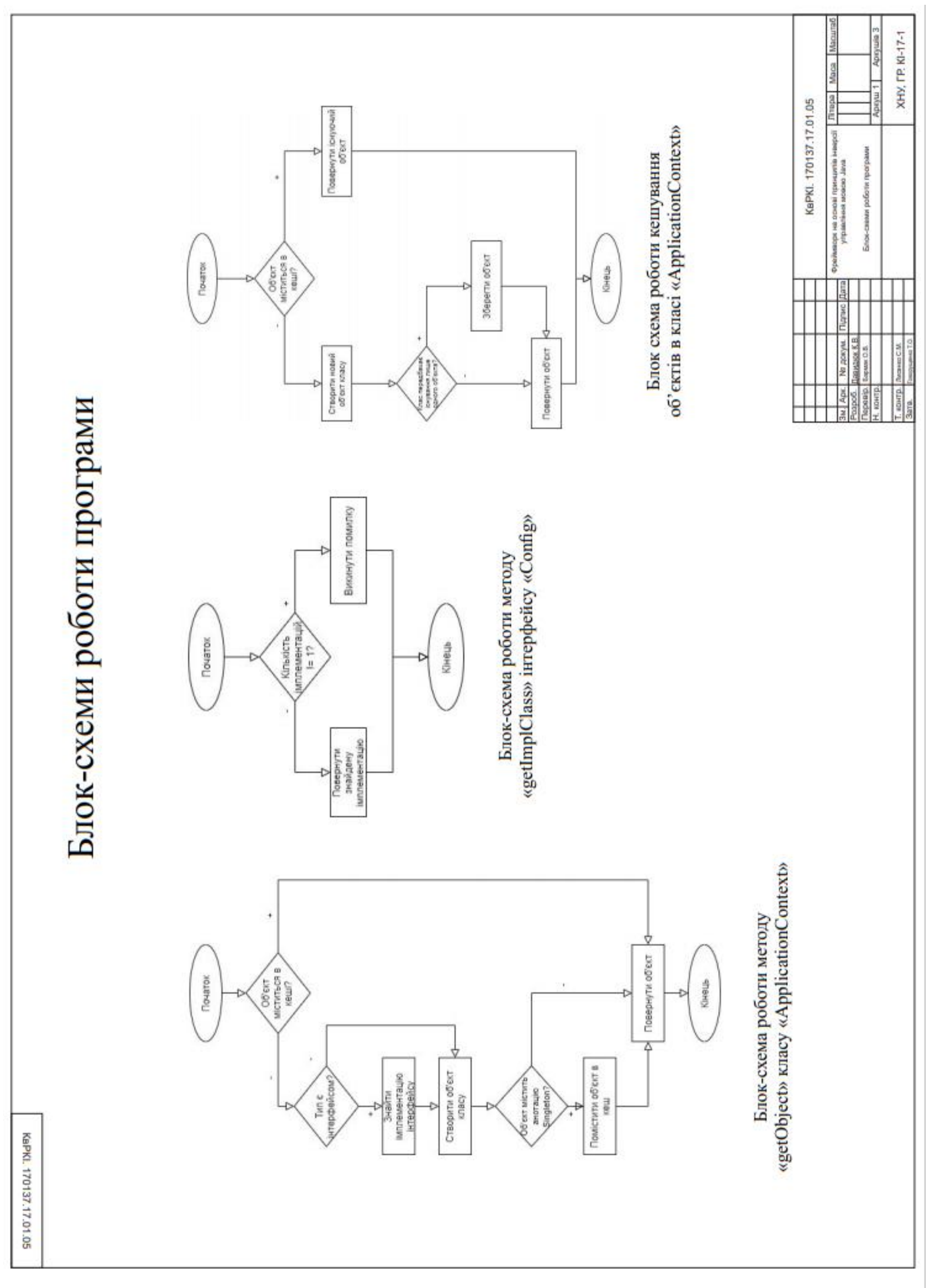
Копія креслення «Діаграма послідовностей»



Додаток Д

(обов’язковий)

Копія креслення «Блок-схеми роботи програми»



UNICHECK

Ім'я користувача
Кафедра КІ

ID перевірки
1008169728

Дата перевірки
04.06.2021 09:26:32 EEST

Тип перевірки
Doc vs Internet + Library

Дата звіту
04.06.2021 09:45:43 EEST

ID користувача
100005591

Назва документа: **Давидюк_Фреймворк на основі принципів інверсії управління мовою Java**

Кількість сторінок: 59 Кількість слів: 11487 Кількість символів: 92803 Розмір файлу: 637.00 KB ID файлу: 100824842

12% Схожість

Найбільша схожість: 7.12% з джерелом з Бібліотеки (ID файлу: 1008214710)

9.51% Джерела з Інтернету | 116 .... Сторінка 61

8.41% Джерела з Бібліотеки | 68 .... Сторінка 61

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0% Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | 1

Anti-Plagiarism v-15.257

Максимальное совпадение с одним документом 1.0%

Словари проверки: en_US, ru_RU, ua_UA. **Ошибок в документах: 14%**

ID: 92259 Название: Фреймворк на основі принципів інверсії управління мовою Java Добавлено в БД: 2021-06-04 Авторы: К. В. Давидюк Руководители: О.В. Бармак Консультанты: Опоненты:	Документ		Суммарное совпадение по Базе Данных	
	Символы	Лексемы	Символы	Лексемы
	73644	650	1773 (2%)	18 (3%)

Источник плагиата

ID	Описание	Наличие плагиата в документе	
		Символы	Лексемы

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Давидюк Костянтин Васильович
Тема: Реалізація фреймворка за принципами інверсії управління на мові Java
Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень ___3___ Кількість сторінок записки ___74___

1. Короткий зміст роботи та прийнятих рішень: метою роботи є розробка мультипроцесорної системи загального призначення на основі топології гіперкуб

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи.

В першому розділі кваліфікаційної роботи проведено дослідження предметної області (принципи об'єктно-орієнтованого проєктування, інверсію управління тощо) та виконано постановку задачі дослідження. В другому розділі кваліфікаційної роботи визначено: функційні вимоги, кроків для проектування програмного забезпечення, проведено вибір рішення для автоматизації збирання та управління проектами, обосновано вибір версії та вендора мови програмування Java, вибрані засоби для автоматичної генерації коду та бібліотеки для роботи з рефлексією. Означені основні компонентів програмного забезпечення: фабрика для створення об'єктів, конфігуратор об'єктів, метадані компонентів, кеш об'єктів, компонент для зчитування конфігурацій з конфігураційного файлу та компонент пошуку імплементацій класу. В третьому розділі кваліфікаційної роботи виконано реалізацію функціональних особливостей: концепція Singleton, пост конструктор класу, налаштування об'єктів згідно конфігураційних файлів, підтримка проксі об'єктів. Також реалізовані програмні компоненти: фабрика для створення об'єктів, конфігуратори об'єктів, компонент пошуку імплементації класу, ApplicationContext, компонент для запуску роботи контейнеру інверсії управління. Проведена розробка програмного забезпечення на базі реалізованого рішення для інверсії управління. Проведено порівняння коду з та без використанням контейнеру для інверсії управління. Розглянуто способи покращення програмної реалізації.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи: недостатньо уваги на валідацію та верифікацію.

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на належному науково-технічному рівні.

8. Інші зауваження: ____________________

9. Оцінка дипломної роботи: відміно

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи)
Мартинюк Валерій Володимирович, зав.каф. АКІТ.ТК, ХНУ д.т.н., професор

"_7_" _червня___ 2021 р. __________ (підпис)

Завідувачу кафедри КІСП
д-ру техн.наук, проф. Говорущенко Т. О.

Давидюка К. В.
ПІБ здобувача вищої освіти

ФПКТС, 4 курсу, групи КІ-17-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність плагіату ознайомлений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

07.06.2021р

дата

підпис

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА СИСТЕМНОГО ПРОГРАМУВАННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Фреймворк на основі принципів інверсії управління мовою Java

Автор: Давидюк Костянтин Васильович

Спеціальність: 123 – Комп'ютерна інженерія

Освітня програма: освітньо-професійна

Науковий керівник: Бармак О.В., д.т.н, професор

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) запозичення розміщені в розділах аналізу існуючих аналогів та прототипів, які не описують безпосередньо авторське дослідження і не стосуються результатів роботи;
2) усі запозичення фрагментарні;
3) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з більш ніж 10 джерелами на один фрагмент речення;
4) серед запозичень знаходяться загальновідомі терміни, скорочення та визначення.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ідентичності/схожості, складає 12 і адресується до 116 першоджерела, що, з урахуванням наведених обгрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи	____________	О.В. Бармак
Гарант ОП	____________	С.М. Лисенко
Завідувач кафедри КІСП	____________	Т. О. Говорущенко