

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
(повна назва інституту/факультету)

Кафедра інформатики та програмної інженерії
(повна назва кафедри)

«До захисту допущено»

Завідувач кафедри

_____ Едуард ЖАРІКОВ
(підпис) (ім'я прізвище)

“ ” _____ 2025 р.

Дипломний проєкт
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення
інформаційних систем»
спеціальності «121 Інженерія програмного забезпечення»

на тему: ІоС контейнер для створення об'єктів на основі складних умов

Виконав студент IV курсу, групи _____ ІП-12
(шифр групи)

_____ Горобець Олексій Сергійович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник ст. викл., д-р філософії, Сарнацький В. В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант ст. викл., д-р філософії, Головченко М. М. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент доцент, к.т.н., доцент, Жданова О. Г. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті
немає запозичень з праць інших авторів без
відповідних посилань.

Студент _____
(підпис)

Київ – 2025

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 Інженерія програмного забезпечення

Освітньо-професійна програма – Інженерія програмного забезпечення
інформаційних систем

ЗАТВЕРДЖУЮ

Завідувач кафедри

(підпис) Едуард ЖАРІКОВ
(ім'я прізвище)

“ ____ ” _____ 2025 р.

ЗАВДАННЯ
на дипломний проєкт студенту

Горобцю Олексію Сергійовичу

(прізвище, ім'я, по батькові)

1. Тема проєкту ІоС контейнер для створення об'єктів на основі складних умов

керівник проєкту Сарнацький Владислав Віталійович, д-р філософії, ст. викл.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «23» травня 2025 р. №1705-с

2. Термін подання студентом проєкту «16» червня 2025 року

3. Вихідні дані до проєкту: технічне завдання

4. Зміст пояснювальної записки

1) Передпроєктне обстеження предметної області: аналіз предметної області та існуючих рішень, аналіз відомих програмних продуктів, алгоритмічних та технічних рішень, постановка завдання.

2) Розроблення вимог до програмного забезпечення: варіанти використання ПЗ, розроблення функціональних, нефункціональних та системних вимог, аналіз економічних показників ПЗ.

3) Конструювання та розроблення програмного забезпечення: архітектура ПЗ, архітектурні рішення та обґрунтування вибору засобів розробки, конструювання програмного забезпечення, аналіз безпеки даних.

4) Аналіз якості та тестування програмного забезпечення: аналіз якості ПЗ, опис процесів тестування, опис контрольного прикладу.

5) Розгортання та супровід програмного забезпечення: розгортання програмного забезпечення, супровід програмного забезпечення.

5. Перелік графічного матеріалу

1) Схема структурна варіантів використань _____

2) Схема структурна компонентів програмного забезпечення _____

3) Схема бізнес процесу «Ініціалізація ІоС контейнеру» _____

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання «15» березня 2025 року _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Вивчення рекомендованої літератури	15.03.2025	
2	Аналіз існуючих методів розв'язання задачі	22.03.2025	
3	Постановка та формалізація задачі	29.03.2025	
4	Розробка інформаційного забезпечення	05.04.2025	
5	Алгоритмізація задачі	12.04.2025	
6	Обґрунтування вибору використаних технічних засобів	19.04.2025	
7	Розробка програмного забезпечення	26.04.2025	
8	Налагодження програми	03.05.2025	
9	Виконання графічних документів	10.05.2025	
10	Оформлення пояснювальної записки	25.05.2025	
11	Подання ДП на попередній захист	02.06.2025	
12	Подання ДП рецензенту	10.06.2025	
13	Подання ДП на основний захист	16.06.2025	

Студент

(підпис)

Олексій ГОРОБЕЦЬ

(ініціали, прізвище)

Керівник

(підпис)

Владислав САРНАЦЬКИЙ

(ініціали, прізвище)

АНОТАЦІЯ

Пояснювальна записка дипломного проєкту складається з п'яти розділів, містить 31 таблицю, 36 рисунків та 36 джерел – загалом 83 сторінки.

Дипломний проєкт присвячений розробці ІоС контейнера для створення об'єктів на основі складних умов.

Метою розробки є спрощення реалізації функціоналу та тестування програмних компонентів шляхом автоматизації управління життєвим циклом об'єктів на основі ІоС підходу.

В першому розділі проведено аналіз предметної області та існуючих рішень, аналіз відомих програмних продуктів, алгоритмічних та технічних рішень, виконано постановку завдання.

Другий розділ присвячений розробці вимог до програмного забезпечення. Розроблено варіанти використання ПЗ, функціональні, нефункціональні та системні вимоги, проаналізовано економічні показники ПЗ.

В третьому розділі виділено та описано архітектурні рішення та обґрунтовано вибір засобів розробки, конструювання програмного забезпечення.

В четвертому розділі проведено аналіз якості та тестування фреймворку Contextor, описано процес тестування та контрольний приклад.

У п'ятому розділі розглянуто процес розгортання та супроводу ІоС контейнера Contextor. Фреймворк опубліковано в репозиторії Sonatype, що значно спростило інтеграцію з іншими проєктами.

Програмне забезпечення Contextor опубліковано у відкритому репозиторії Sonatype. Таке розміщення надало простий та зручний спосіб інтеграції фреймворку як у нові, так і в існуючі проєкти.

КЛЮЧОВІ СЛОВА: ІОС КОНТЕЙНЕР, ФРЕЙМВОРК, ІН'ЄКЦІЯ ЗАЛЕЖНОСТЕЙ, УМОВНЕ СТВОРЕННЯ ОБ'ЄКТІВ, MAVEN, РЕФЛЕКСІЯ.

ABSTRACT

The explanatory note of the diploma project consists of five sections, contains 31 tables, 36 figures and 36 sources a total of 83 pages.

The diploma project is dedicated to the development of an IoC container for creating objects based on complex conditions.

The goal of the development is to simplify the implementation of functionality and testing of software components by automating the life cycle management of objects based on the IoC approach.

In the first section, an analysis of the subject area and existing solutions, an analysis of known software products, algorithmic and technical solutions, and a task statement are performed.

The second section is dedicated to the development of software requirements. Software usage options, functional, non-functional and system requirements are developed, and economic indicators of the software are analyzed.

In the third section, architectural solutions are highlighted and described and the choice of software development and construction tools is justified.

In the fourth section, a quality analysis and testing of the Contextor framework is conducted, the testing process and a control case are described.

The fifth section discusses the process of deploying and maintaining the IoC of the Contextor container. The framework is published in the Sonatype repository, which greatly simplifies integration with other projects.

The Contextor software is published in the Sonatype open repository. This publication provides a simple and convenient way to integrate the framework into both new and existing projects.

KEYWORDS: IOC CONTAINER, FRAMEWORK, DEPENDENCY INJECTION, CONDITIONAL OBJECT CREATION, MAVEN, REFLECTION.

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ____ ” _____ 2025 р.

**ІОС КОНТЕЙНЕР ДЛЯ СТВОРЕННЯ ОБ’ЄКТІВ НА ОСНОВІ
СКЛАДНИХ УМОВ**

Технічне завдання

КПІ.ПІ-1208.045200.01.91

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Владислав САРНАЦЬКИЙ

Нормоконтроль:

_____ Максим ГОЛОВЧЕНКО

Виконавець:

_____ Олексій ГОРОБЕЦЬ

Київ – 2025

ЗМІСТ

1	НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ.....	4
2	ПІДСТАВА ДЛЯ РОЗРОБКИ.....	5
3	ПРИЗНАЧЕННЯ РОЗРОБКИ.....	6
4	ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	7
4.1	Вимоги до функціональних характеристик	7
4.1.1	Виявлення компонентів	7
4.1.2	Створення об'єктів	7
4.1.3	Вирішення конфліктів.....	7
4.1.4	Створювати об'єкти на основі умов	7
4.1.5	Обробка помилок.....	8
4.1.6	Доступ до об'єктів в контексті.....	8
4.1.7	Логування процесів, що відбувається	8
4.2	Вимоги до надійності	8
4.3	Умови експлуатації.....	8
4.3.1	Вид обслуговування	8
4.3.2	Обслуговуючий персонал	8
4.4	Вимоги до складу і параметрів технічних засобів	9
4.5	Вимоги до інформаційної та програмної сумісності	9
4.5.1	Вимоги до вхідних даних.....	9
4.5.2	Вимоги до вихідних даних	9
4.5.3	Вимоги до мови розробки.....	9
4.5.4	Вимоги до середовища розробки	9
4.5.5	Вимоги до представленню вихідних кодів	9
4.6	Вимоги до маркування та пакування.....	10
4.7	Вимоги до транспортування та зберігання	10
4.8	Спеціальні вимоги	10
5	ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ	11
5.1	Попередній склад програмної документації	11
5.2	Спеціальні вимоги до програмної документації	11

6	СТАДІЇ І ЕТАПИ РОЗРОБКИ.....	12
7	ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ	13

1 НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: ІоС контейнер для створення об'єктів на основі складних умов.

Галузь застосування:

Наведене технічне завдання поширюється на розробку програмного забезпечення Contextor, яке використовується для автоматизації розв'язання залежностей між об'єктами та контролю їх життєвого циклу і призначена для розробників комплексного десктопного та серверного програмного забезпечення.

2 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки Програмного забезпечення Contextor є завдання на дипломне проєктування, затверджене кафедрою інформатики та програмної інженерії Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для автоматизації життєвого циклу об'єктів та полегшення механізмів контролю над умовами їх створення за допомогою ІоС контейнера.

Метою розробки є спрощення реалізації функціоналу та тестування програмних компонентів шляхом автоматизації управління життєвим циклом об'єктів на основі ІоС підходу.

4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вимоги до функціональних характеристик

Програмне забезпечення повинно забезпечувати виконання наступних основних функцій:

4.1.1 Виявлення компонентів

- Контейнер повинен автоматично виявляти та реєструвати об'єкти, позначені спеціальною анотацією;
- Проводить сканування класів на основі вказаного пакету.

4.1.2 Створення об'єктів

- Контейнер повинен підтримувати ін'єкцію залежностей на основі конструктора.
- Можливість задавати власну назву об'єкту.

4.1.3 Вирішення конфліктів

- Можливість вказати конкретну імплементацію класу, який необхідно додати як залежність в об'єкт.
- Повинен вирішувати залежності рекурсивно. Тобто, якщо А залежить від В, а В залежить від С, всі об'єкти повинні бути створені коректно.

4.1.4 Створювати об'єкти на основі умов

- Можливість вирішувати чи буде створено об'єкт на основі системних параметрів
- Можливість задавати часові рамки за яких можливо створити об'єкт
- Створення об'єкта лише за доступності веб сервісу
- Створювати об'єкт лише за умова, що заданий файл існує

4.1.5 Обробка помилок

- Оброблення помилок при ініціалізації об'єкта за відсутності необхідних залежностей
- Оброблення помилок при помилці опрацювання умови створення об'єкта

4.1.6 Доступ до об'єктів в контексті

- Можливо отримання об'єкта за його класом
- Отримання об'єкта за його унікальним ім'ям

4.1.7 Логування процесів, що відбувається

- Логується інформація, про створення об'єктів
- Інформація, про умови, що були перевірені перед створенням об'єктів
- Можливість налаштувати рівень логування

4.2 Вимоги до надійності

Особливих вимог по надійності не висувається, відповідальність за некоректне використання програмного забезпечення покладається на користувача.

4.3 Умови експлуатації

Умови експлуатації згідно СанПін 2.2.2.542 – 96.

4.3.1 Вид обслуговування

Вимоги до виду обслуговування не висуваються.

4.3.2 Обслуговуючий персонал

Вимоги до обслуговуючого персоналу не висуваються

4.4 Вимоги до складу і параметрів технічних засобів

Мінімальна конфігурація технічних засобів:

- тип процесору: Intel Core i5;
- об'єм ОЗП: 4 GB;
- вільне місце на диску 1GB

Рекомендована конфігурація технічних засобів:

- тип процесору: Intel Core i7;
- об'єм ОЗП: 16 GB;
- вільне місце на диску 10GB

4.5 Вимоги до інформаційної та програмної сумісності

Середовище розробки IntelliJ IDEA, доступ до Maven репозиторію.

4.5.1 Вимоги до вхідних даних

Вимоги до спеціального форматування вхідних даних не висуваються.

4.5.2 Вимоги до вихідних даних

Вимоги до спеціального форматування вихідних даних не висуваються

4.5.3 Вимоги до мови розробки

Розробку виконати на мові програмування Java. Версія повинна бути 17 чи вище.

4.5.4 Вимоги до середовища розробки

Розробку виконати на платформі IntelliJ IDEA.

4.5.5 Вимоги до представлення вихідних кодів

Вихідний код програми має бути представлений у вигляді бібліотеки .jar файлу, який включає в себе необхідні ресурсні файли.

4.6 Вимоги до маркування та пакування

Вимоги до маркування та пакування не висуваються.

4.7 Вимоги до транспортування та зберігання

Вимоги до транспортування та зберігання не висуваються.

4.8 Спеціальні вимоги

Зібрати та розмістити створену бібліотеку в хмарному публічному репозиторії, для вільного доступу до неї зі збірника проєктів Maven.

5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

5.1 Попередній склад програмної документації

У склад супроводжувальної документації повинні входити наступні документи на аркушах формату А4:

- пояснювальна записка;
- технічне завдання;
- текст програми;
- програма та методика тестування;
- керівництво користувача.

Графічна частина повинна бути виконана на аркушах формату А3 та містити наступні документи:

- схема структурна варіантів використання;
- схема структурна компонентів програмного забезпечення;
- схема бізнес процесу «Ініціалізація ІоС контейнеру».

5.2 Спеціальні вимоги до програмної документації

Програмні модулі, котрі розробляються, повинні бути задокументовані, тобто тексти програм повинні містити всі необхідні коментарі.

6 СТАДІЇ І ЕТАПИ РОЗРОБКИ

№	Назва етапу	Строк	Звітність
1.	Вивчення рекомендованої літератури	15.03	
2.	Аналіз існуючих методів розв'язання задачі	22.03	
3.	Постановка та формалізація задачі	29.03	Технічне завдання
4.	Розробка інформаційного забезпечення	05.04	Специфікації програмного забезпечення
5.	Алгоритмізація задачі	12.04	Схема структурна програмного забезпечення та специфікація компонентів (діаграма класів, схема алгоритму)
6.	Обґрунтування вибору використаних технічних засобів	19.04	
7.	Розробка програмного забезпечення	26.04	Тексти програмного забезпечення
8.	Налагодження програми	03.05	Тести, результати тестування
9.	Виконання графічних документів	10.05	Графічний матеріал проєкту
10.	Оформлення пояснювальної записки	25.05	Пояснювальна записка

7 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

Тестування розробленого програмного продукту виконується відповідно до “Програми та методики тестування”.

**Пояснювальна записка
до дипломного проєкту**

на тему: ІоС контейнер для створення об'єктів на основі складних умов

КПІ.ПІ-1208.045200.02.81

Київ – 2025

ЗМІСТ

ВСТУП	5
1 ПЕРЕДПРОЄКТНЕ ОБСТЕЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ	6
1.1 Постановка завдання дипломного проєктування	6
1.2 Аналіз предметної області	6
1.3 Аналіз існуючих рішень.....	11
1.3.1 Аналіз відомих програмних продуктів.....	11
1.3.2 Аналіз відомих алгоритмічних та технічних рішень.....	16
Висновки до розділу	20
2 РОЗРОБЛЕННЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	21
2.1 Варіанти використання програмного забезпечення.....	21
2.2 Розроблення функціональних вимог	26
2.3 Розроблення нефункціональних вимог	33
2.4 Аналіз системних вимог.....	34
2.5 Аналіз економічних показників програмного забезпечення.....	34
2.6 Постановка завдання на розробку програмного забезпечення.....	37
Висновки до розділу	38
3 КОНСТРУЮВАННЯ ТА РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	39
3.1 Архітектура програмного забезпечення.....	39
3.2 Архітектурні рішення та обґрунтування вибору засобів розробки.....	42
3.3 Конструювання програмного забезпечення.....	51
3.4 Аналіз безпеки даних	53
Висновки до розділу	53
4 АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	55
4.1 Аналіз якості ПЗ.....	55
4.2 Опис процесів тестування.....	59
4.3 Опис контрольного прикладу.....	63
Висновки до розділу	69

5 РОЗГОРТАННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	71
5.1 Розгортання програмного забезпечення.....	71
5.2 Супровід програмного забезпечення.....	74
Висновки до розділу	76
ВИСНОВКИ.....	77
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	80
ДОДАТОК А ЗВІТ ПОДІБНОСТІ.....	83

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

IoC	– Inversion of control – інверсія контролю.
IDE	– Integrated Development Environment – інтегроване середовище розробки.
API	– Application programming interface, прикладний програмний інтерфейс.
JDK	– Java development kit.
ОС	– Операційна система.
ПЗ	– Програмне забезпечення.
ООП	– Об’єктно-орієнтоване програмування.
ІТ	– Information Technologies – інформаційні технології.

ВСТУП

Сучасний світ стрімко цифровізується, і разом з цим зростають вимоги до програмного забезпечення. Якщо раніше більшість систем мали локальний характер і вирішували порівняно прості задачі, то сьогодні програмні продукти постійно ускладнюються та до них висуваються більш жорсткі вимоги. Наприклад, сьогодні програмні застосунки повинні забезпечувати роботу в розподіленому середовищі, витримувати одночасну велику кількість запитів, інтегруватися з зовнішніми сервісами та відповідати усім вимогам безпеки, надійності та масштабованості.

Змінюється й підхід до вибору та побудови архітектури. Застосунки з часом еволюціонували від монолітних до мікросервісних, іншими словами від ізольованих до таких, що працюють разом у рамках складних цифрових екосистем. Це означає, що навіть відносно прості за задумом системи з часом можуть перерости в повноцінні ентєрпрайз-рішення, які вимагають структурованої архітектури, автоматизованого управління залежностями та високого рівня організації коду.

Саме тому розробка програмного забезпечення стає все більш складним процесом. Для спрощення та здешевлення процесу розробки було створено фреймворки високого рівня абстракції, які допомагають ефективно розробляти програмне забезпечення, та більше зосереджуючись на бізнес-логіці, ніж на інфраструктурі.

Основною ідеєю, яка лежить в основі фреймворку, є інверсія контролю (*англ.* inversion of control, IoC). Саме завдяки їй можливо віддати контроль над ходом виконання програми фреймворку, тим самим зменшивши навантаження на розробника. Сучасну ентєрпрайз-розробку неможливо уявити без використання IoC фреймворків. Отже, тема дипломного проєкту є досить актуальною.

Метою розробки є спрощення реалізації функціоналу та тестування програмних компонентів шляхом автоматизації управління життєвим циклом об'єктів на основі IoC підходу.

1 ПЕРЕДПРОЄКТНЕ ОБСТЕЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка завдання дипломного проєктування

Для успішного виконання дипломного проєктування необхідно виконати наступні завдання:

- здійснити аналіз предметної області та описати бізнес-процеси, які є ключовими для функціонування ІоС контейнеру;
- визначити загальне завдання розробки, яка буде виконуватися в рамках дипломного проєктування;
- проаналізувати вже існуючі алгоритмічні та технічні рішення для реалізації ІоС контейнеру та програмні продукти, представлені на ринку;
- виконати аналіз та моделювання бізнес-процесів;
- визначити та розробити вимоги до програмного забезпечення, зокрема функціональні, нефункціональні та системні;
- здійснити аналіз якості та тестування програмного забезпечення;
- розробити програмне забезпечення та обґрунтувати вибір засобів для його розробки;
- сконструювати та розробити програмне забезпечення;
- створити супроводжувальну документацію до розробленого програмного забезпечення.

1.2 Аналіз предметної області

Упродовж всієї історії програмної інженерії, з початку її зародження до сьогодення підходи та методики створення програм постійно змінюються та вдосконалюються. Протягом останніх десятиліть індустрія програмного забезпечення зазнала значних змін як у технологіях, що використовуються, так і в концептуальних, архітектурних рішеннях. Деякі практики, які раніше вважалися стандартами індустрії, поступово відходять на другий план і замінюються новими кращими практиками та підходами.

В пошуках спрощення та прискорення процесу розробки нового програмного забезпечення інженери почали виділяти окремі функції для вирішення чітких специфічних задач у бібліотеки. Бібліотека – це набір заздалегідь написаного коду, який розробники використовують для вирішення певних завдань [1].

Бібліотеки значно прискорюють процес розробки та підвищують надійність коду, адже їхні функції та методи зазвичай ретельно протестовані та перевірені як компаніями-розробниками, так і спільнотами програмістів у випадку проєктів з відкритим кодом. Водночас вони не усувають одну з основних проблем – відсутність стандартизованих підходів до вирішення архітектурних завдань.

З часом, коли програмне забезпечення почало все більше ускладнюватись, інженери дійшли висновку, що необхідно виявляти та виділяти патерни та архітектурні рішення, які повторювалися з проєкту в проєкт. Таким чином, за допомогою еволюційного підходу, було виділено групи типових задач та їх типових рішень в окремі програмні продукти – фреймворки.

Отже, фреймворк – це інтегрований набір програмних артефактів, таких як класи, об'єкти та компоненти, які співпрацюють, щоб забезпечити архітектуру, яку можна повторно використовувати, для сімейства пов'язаних програм [2].

Масовий перехід на використання фреймворків став надзвичайно важливим етапом у розвитку індустрії програмного забезпечення. Такий підхід має чимало переваг. Їх можна розділити на соціальні та інженерні. Розглянемо деякі соціальні переваги.

По-перше, єдиний структурований підхід дозволяє розробникам швидко зорієнтуватися в нових проєктах. Саме уніфіковані архітектурні рішення скорочують час онбордингу розробника в компанії, та прискорюють момент, коли він почне приносити бізнесу користь. Відбувається це завдяки тому, що розробник не витрачає час на вивчення унікальних архітектурних рішень,

властивих суто для цього проєкту, а в першу чергу може зосередитися на аналізі бізнес-процесів та їх реалізації.

Ще однією суттєвою перевагою є формування спільного набору термінів, що формують «словник», який вживають розробники, що працюють з одним фреймворком. Це значно спрощує комунікацію як всередині команди, так і на більш глобальному рівні в рамках спільнот розробників.

Також можна додати, що навколо фреймворків природньо формуються спільноти фахівців. У таких спільнотах можна знайти відповіді на актуальні питання та обмінюватися досвідом для вирішення конкретних задач, використовуючи всім відомі спільні інструменти. Це надає значну перевагу порівняно з використанням власних рішень, які були створені та підтримуються в закритому вигляді в межах однієї компанії.

Тепер розглянемо, які технічні переваги надає використання фреймворків. Зазвичай, фреймворки задають модульну архітектуру проєкту. Ефективність такого рішення перевірена на багатьох проєктах, що вже використовували такий підхід. Це значно полегшує інтеграцію нового коду та розширення функціональності без значних змін у структурі проєкту.

Крім цього, фреймворки містять вбудовані рішення для багатьох стандартних технічних задач. Таким чином, розробникам, що використовують фреймворк, не потрібно щоразу «винаходити колесо» для розв'язання типових задач. Такими задачами, зазвичай, є інфраструктурні аспекти, що не пов'язані з бізнес-логікою застосунку. Наприклад, більшість фреймворків мають вбудовану логіку та інструменти для відловлювання помилок, логування, збору метрик та тестування.

Більшість із вище зазначених переваг можливі завдяки тому, що фреймворки користуються принципом інверсії контролю (Inversion of control IoC). Саме інверсія контролю – це те, що відрізняє фреймворк від бібліотеки. На відміну від бібліотек, де розробники просто викликають певні функції, функція виконує роботу і повертає контроль над ходом виконання клієнтському коду, фреймворк бере контроль над ходом виконання коду на

себе. Ця відмінність між бібліотеками та фреймворками чітко описана в статті «Designing Reusable Classes» відомого автора книги «Design Patterns: Elements of Reusable Object-Oriented Software» [3].

У своїй статті він зазначає, що однією з важливих характеристик фреймворку є те, що методи, які розробник визначає для його налаштування, дуже часто викликаються саме із самого фреймворку, а не з клієнтського коду. Тобто фреймворк часто відіграє роль основної програми, яка координує та визначає послідовності дій, що виконуються у застосунку. Така інверсія контролю надає фреймворкам можливість виступати у ролі розширюваних скелетами. Методи, які визначає користувач, просто адаптують та налаштовують загальні алгоритми та бізнес логіку, яка вже визначена у фреймворку для конкретно взятого застосунку [4].

Отже, дамо визначення інверсії контролю. IoC – це загальний принцип, який означає передачу контролю над певними процесами, що відбуваються в програмі, від самого застосунку до зовнішнього фреймворку або середовища виконання.

Інверсію контролю можна реалізувати різними способами. Однією з найбільш популярних реалізацій є Dependency Injection (DI). Це підхід, що втілює принцип інверсії контролю, основна ідея якого полягає в передачі відповідальності за пошук, створення та визначення залежностей спеціалізованому контейнеру. Іншими словами, клас займається виключно бізнес-логікою, а вирішення залежностей цього класу повністю делегується на зовні.

Сучасну ентерпрайз-розробку неможливо уявити без використання фреймворків та DI зокрема. Застосування DI забезпечує цілу низку архітектурних переваг.

Так, із використанням DI контейнеру компоненти програми стають незалежними від конкретної реалізації, що дозволяє легко замінювати, налаштовувати або розширювати функціональність без модифікації вже

існуючого коду. Це призводить до підвищення гнучкості архітектури застосунку.

За допомогою DI можна організувати централізоване керування життєвим циклом об'єктів та розв'язування залежностей. Такий підхід дозволяє ефективно контролювати споживання ресурсів та налаштовувати поведінку застосунку до різних варіантів використання. Завдяки слабкому зв'язку між модулями та можливості декларативного опису залежностей, код набагато легше підтримувати та модифікувати відповідно до змін у вимогах та бізнес-логіки.

Оскільки залежності передаються ззовні, з'являється можливість замінювати реалізації залежностей. Така функціональність необхідна для якісного покриття коду автотестами, оскільки дозволяє використовувати mock-об'єкти при тестуванні [5]. Це значно спрощує процес покриття коду unit-тестами, а також сприяє впровадженню сучасних практик тестування, таких як Test Driven Development [6]. Також, завдяки зменшенню зв'язності між компонентами, окремі частини системи можна ізолювати та уникнути складних конфігураційних налаштувань при написанні тестів.

Використання IoC контейнерів та фреймворків має також певні недоліки. Напевно, одним із основних недоліків є велике накопичення шарів абстракції, що може ускладнити процес відладки коду та дебагу. Також варто визнати, що фреймворки використовують більше ресурсів для роботи застосунку, особливо під час запуску програми.

В середовищах, де швидкість старту застосунку критична, потрібно приділяти більше уваги до нефункціональних вимог, зокрема, тестування продуктивності та швидкодії застосунку. В разі незадовільних результатів тестування потрібно взагалі розглянути можливість відмови від використання IoC контейнерів, тим самим ускладнивши процес розробки, але вигравши в продуктивності застосунку. Проте, для більшості серверних застосунків ця особливість не є критичною, оскільки не має суттєвого впливу на кінцевий користувацький досвід. Це обумовлено тим, що на реальному продакшн

середовищі застосовуються поступові оновлення, які дозволяють уникнути час простою системи.

Однією із стратегій покращення існуючих систем може бути зменшення рівня абстракції. Під цим розуміється, що в сучасних фреймворках іноді закладено занадто багато функцій та шарів абстракції, більшість із яких залишаються не використаними. Тому для створення легкого та простого у використанні ІоС контейнера має сенс виокремити лише ключові функції та прибрати зайві компоненти. Водночас з цим, багато сучасних фреймворків не мають розширених вбудованих механізмів, які б дозволяли керувати умовами створення об'єктів.

В рамках дипломного проєкту було вирішено в процесі розробки програмного застосунку дотримуватись оптимального рівня абстракції – достатньо простого для легкого налагодження, але водночас не надто деталізованого, щоб зберегти простоту розробки та зменшити криву навчання. Також буде розроблено розширений набір інструментів, який дозволить користувачу більш зручно задавати умови для створення об'єктів.

1.3 Аналіз існуючих рішень

Проаналізуємо відоме на сьогодні алгоритмічне забезпечення в даній області та технічні рішення, що допоможуть у реалізації ІоС контейнеру Contextor. Далі будуть розглянуті готові програмні рішення, допоміжні програмні засоби та засоби розробки.

1.3.1 Аналіз відомих програмних продуктів

На ринку представлено чимало фреймворків із підтримкою ІоС контейнерів. Для порівняння було обрано проєкти, написані як на Java, так і на інших мовах програмування, щоб отримати більш широкую картину доступних рішень.

Dagger – це фреймворк для реалізації принципу ін'єкції залежності для Java та Kotlin, створений компанією Square, а зараз його підтримує та

розробляє Google. Головною особливістю цього фреймворку є використання підходу зі статичною генерацією коду на базі анотацій, які розставляє розробник. Dagger широко використовується в розробці під мобільні пристрої, зокрема під операційну систему Android [7].

Guice – фреймворк, написаний мовою Java, який реалізує принцип інверсії контролю, а саме Dependency Injection. Він розроблений компанією Google. Зазвичай використовується в нескладних серверних застосунках, дозволяє імплементувати управління залежностями та принцип «слабкого зв'язування» (англ. «loose-coupling») [8, 9].

SmartIoC – це простий фреймворк для ін'єкції залежностей, написаний на мові Ruby. Для реєстрації та пошуку компонентів цей фреймворк використовує сканування пакетів чи вказаних вхідних файлів. Набір функцій, які надає SmartIoC, доволі обмежений порівняно з більш просунутими контейнерами IoC, такими як Spring. Йому не вистачає розвинутого управління життєвим циклом і створення об'єктів на основі вказаних умов. SmartIoC підходить лише для невеликих проєктів, де кількість залежностей, які необхідно обробляти, незначна [10].

Spring – це популярний фреймворк для розробки складних серверних застосунків мовою Java. Перша версія Spring була розроблена у 2003 р. австралійським розробником Родом Джонсоном та згодом була викуплена компанією VMware. Фреймворк Spring вирізняється тим, що надає продвинутий функціонал не тільки для контролю над життєвим циклом об'єктів, а й в цілому для розробки веб-застосунків. Крім того, істотною перевагою цього фреймворку є наявність великої спільноти розробників, які активно підтримують і розвивають його. Однак, хоча Spring і пропонує широкий набір функцій, витрати часу на його налаштування можуть бути невиправдано великими для простих проєктів. [11]

InversifyJS – це доволі простий контейнер інверсії контролю для програм, написаних мовою JavaScript і TypeScript. Головною ідеєю цього фреймворку є сприяння модульності проєкту та стимулювання написання

чистого коду з дотриманням принципів SOLID. InversifyJS будує логіку своєї роботи на декораторах та додатково додає метадані для спрощення керування залежностями. Додатково варто мати на увазі, що InversifyJS несе за собою накладні витрати по часу виконання програми, через залежність від відображення та обробки метаданих об'єктів [12].

Порівняльний аналіз проєкту з аналогами наведено в табл. 1.1.

Таблиця 1.1 – Порівняння Contextor з аналогами

Функціонал	Contextor	Dagger	Guice	Smart IoC	Spring	InversifyJS	Пояснення
Виявлення компонентів	На основі анотацій, сканування пакету	Лише на основі анотацій	На основі анотацій, сканування пакету	Сканування пакету	На основі анотацій, сканування пакету	На основі анотацій, але вимагає мануального зв'язування	Для наповнення контексту об'єктами спочатку потрібно визначити, які саме об'єкти потрібно створити. Для цього проводиться пошук в дереві проєкту.
Ін'єкція залежностей через конструктор	Підтримує	Підтримує	Підтримує	Не підтримує, ін'єкція через окремий inject метод	Підтримує	Частково підтримує	Ін'єкція через конструктор – це основний спосіб для встановлення залежностей в об'єкт

Продовження таблиці 1.1

Механізми логування	Підтримує логування та вивід детальної інформації про умови створення об'єктів	Базове логу- вання	Базове логу- вання	Базове логу- вання	Підтри- мує, прод- винуте логу- вання	Базове логування	Продвину- тий механізм логування, при створенні об'єктів є дуже важливим для відстежен- ня компонен- тів, що створюють- ся, та налагод- ження програми
Обробка помилки	Підтримує обробку помилки заванта- ження контексту з детальним повідом- ленням про причину помилки	Не має вбудова- ного механіз- му обробки помилки, викори- стовує стандар- тні Java виключення	Підтри- мує, обгор- тає помил- ки в свої власні класи	Обме- жено підтри- мує	Підтри- мує продви- нуту обробку помилки	Відсутнє, помилки видають- ся одразу, обробка поклада- ється на користу- вача	Механізм обробки помилки гарантує, що помилки, які можуть виникнути під час виконання програми, будуть оброблені належним чином. Дозволяють надати ко- ристувачу додатковий контекст виникнення помилки для її подальшого усунення

Продовження таблиці 1.1

Знаходження залежності за ідентифікатором	Підтримує	Не підтримує	Підтримує	Не підтримує	Підтримує	Не підтримує	Дозволяє знаходити об'єкт в контексті застосунку за певним ідентифікатором
Створення об'єктів на основі часових рамок	Підтримує	Не підтримує	Не підтримує	Не підтримує	Не підтримує	Не підтримує	Функціональність, яка дозволяє встановлювати умови для створення об'єктів, задаючи певні часові рамки. Наприклад, створення об'єктів лише вдень або вночі
Мова програмування	Java	Java, Kotlin	Java, Kotlin	Ruby	Java, Kotlin	TypeScript	Вибір мови програмування впливає на швидкодію, підхід до використання фреймворку

Продовження таблиці 1.1

Наявність підтримуваної документації	+	+	+		+	+	Актуалізована документація важлива для полегшення адаптації нових розробників та зменшення ризику непорозумінь і помилок під час використання проекту
--------------------------------------	---	---	---	--	---	---	---

1.3.2 Аналіз відомих алгоритмічних та технічних рішень

Існує два основних підходи до реалізації ІоС контейнерів. Перший підхід полягає у визначенні компонентів та залежностей в процесі роботи застосунку (runtime-based). Другий підхід використовує перевірки залежностей під час компіляції, а для виконання створення об'єктів застосовується статична генерація коду. Хоча обидва підходи призначені для створення об'єктів і вирішення залежностей, вони досить суттєво відрізняються у своєму внутрішньому устрою. Проаналізуємо кожен з підходів та визначимо їх переваги та недоліки.

Контейнери, що використовують runtime-based підхід, здатні динамічно вирішувати залежності об'єктів під час виконання програми, тобто компоненти скануються, створюються та встановлюються в залежність лише після компіляції та запуску програми. Популярними прикладами фреймворків, що реалізують такий підхід, є Spring та Guice.

Перевагами даного підходу є гнучкість. Оскільки застосунок вже запущено, для фреймворку відкрита можливість будувати більш складні процеси, такі як перевірки умов створення об'єктів, отримання конфігурації з різних джерел, встановлення зв'язку із зовнішніми сервісами тощо. Також такий підхід дозволяє реалізувати механізми обробки помилок, перехоплюючи та оброблюючи винятки, що виникають. Це дозволяє надати розробнику більш детальний контекст про причину виникнення тієї чи іншої проблеми, що суттєво спростить процес її вирішення.

Статичні IoC контейнери, такі як Dagger, покладаються на попередньо згенерований вихідний код для вирішення залежностей під час компіляції. В результаті ми отримуємо жорстко зв'язані залежності, а створення об'єктів відбувається за фіксованою структурою, яка була попередньо згенерована.

Однозначною перевагою такого підходу є можливість впевнитись, що усі необхідні залежності підключені ще на етапі компіляції програми. Також варто зазначити, що час розгортання таких застосунків буде меншим, оскільки при запуску виконуються заздалегідь згенеровані інструкції, без будь-яких перевірок. Ще однією перевагою є простий процес налагодження програми, зазвичай, статичні контейнери обгортають клас в один, два проксі класи, в яких легко орієнтуватись та знаходити необхідні функції.

Для порівняння, якщо використовувати runtime-based контейнер, класи зазвичай проксіюються та обгортаються в велику кількість шарів, вносячи цим неочевидну логіку та «побічні ефекти», тому проводити процес дебагу такого коду іноді буває доволі складно. Не зважаючи на перераховані переваги, статична генерація коду має суттєві обмеження щодо виконання операцій під час ініціалізації контейнеру. Це суттєво скорочує можливості по виконанню перевірок умов для створення об'єктів. Також варто зазначити, що статичні генератори коду зазвичай підтримують лише прості механізми обробки помилок та логування, покладаючись на стандартні не деталізовані помилки, створені компілятором. Тому таким повідомленням бракує контекстної інформації, яку може надати runtime-based IoC контейнер.

Враховуючи цілі проєктування та передбачувані варіанти використання фреймворку, що розробляється, було обрано підхід runtime-based як базове алгоритмічне та архітектурне рішення. Такий вибір обумовлений необхідністю створення об'єктів на основі заданих умов, динамічного вирішення залежностей в процесі виконання програми і необхідності забезпечувати чітку та прозору інформацію про хід роботи контейнера.

Contextor використовує перевагу runtime-based підходу, надаючи розширений API для задання умов, за яких будуть створюватися об'єкти. Це включає такі параметри, як стан системи, поточний час системи, наявність файлу в системі або режим, в якому була запущена програма. Цю інформацію можливо отримати лише після запуску програми. Додатково архітектура застосунку, заснована на runtime конфігураціях, здатна забезпечувати більш розширене керування життєвим циклом об'єктів і дозволяє тонко налаштовувати політику інстанціювання об'єктів.

Аналіз та моделювання бізнес-процесів

Наведемо основні бізнес процеси, які повинен підтримувати сучасний ІоС контейнер. Для моделювання бізнес процесів використаємо BPMN моделі (рис. 1.1) [13]. Діаграма бізнес процесу «Ініціалізація ІоС контейнеру» розміщена у графічних матеріалах.

Опис моделі бізнес-процесу ініціалізації ІоС контейнера:

- Користувач підключає фреймворк Contextor до проєкту за допомогою автоматичного інструменту для збірки проєктів;
- Користувач запускає контейнер;
- Contextor запускає завантаження конфігурації;
- Contextor виконує запит до зовнішнього ресурсу для зчитування конфігураційного файлу;
- Якщо конфігураційний файл завантажено успішно, контроль повертається до Contextor, інакше виводиться повідомлення про помилку, далі виконання завершується;

- Contextor застосовує отриману конфігурацію;
- Contextor розпочинає сканування пакетів на наявність класів, які необхідно створити. Для цього звертається до зовнішньої системи;
- Зовнішня система надає доступ до пакетів, що будуть скануватися;
- Contextor виявляє компоненти в заданих пакетах;
- Якщо компоненти не знайдені, вивести попередження користувачу, інакше продовжити роботу;
- Contextor перевіряє, чи виконується умова для створення об'єкту;
- Якщо умова виконується, створює об'єкт, інакше виводить попередження користувачу, що умова для створення об'єкту була не виконана;
- Зберігає створений об'єкт в контексті IoC контейнера;
- Завершення ініціалізації IoC контейнера.

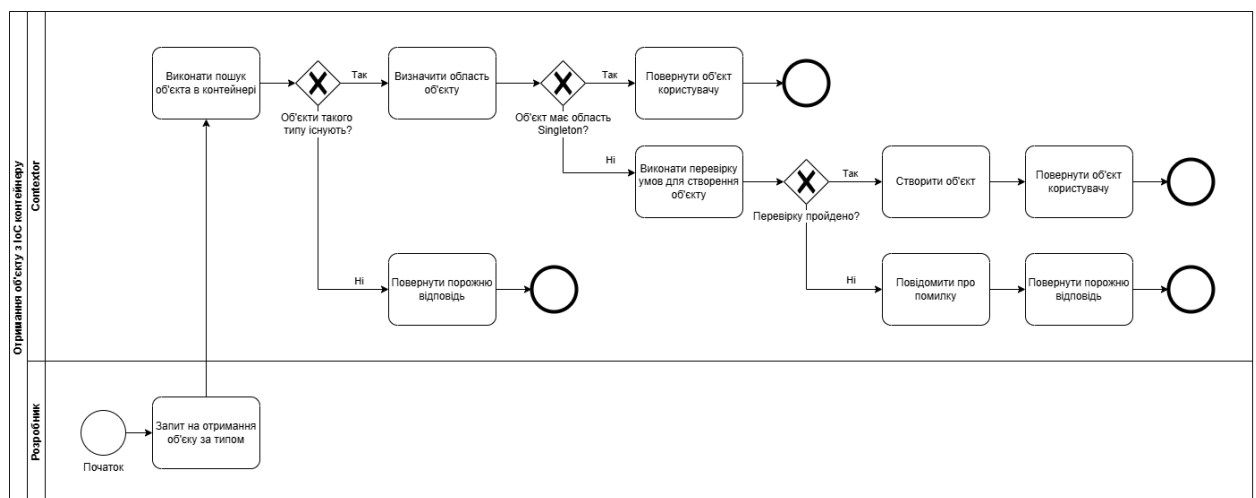


Рисунок 1.1 – Модель бізнес-процесу «Отримання об'єкту з IoC контейнера»

Опис моделі бізнес-процесу отримання об'єкту з контексту IoC контейнера:

- Користувач виконує запит на отримання об'єкту за певним типом;
- Contextor виконує пошук об'єкту в контейнері;
- Якщо об'єкт такого типу існує, продовжити роботу, інакше повернути порожню відповідь та завершити виконання пошуку;
- Contextor визначає область об'єкту;

- Якщо область об'єкту «Singleton», повернути об'єкт користувачу та завершити виконання пошуку, інакше продовжити виконання;
- Виконати перевірку умов для створення об'єкту;
- Якщо умови виконуються, створити об'єкт, інакше повідомити користувача про помилку та повернути порожню відповідь;
- Повернути створений об'єкт користувачу.

Висновки до розділу

Отже, в першому розділі було проведено аналіз сучасного стану у сфері ІоС контейнерів та індустрії в цілому. Спираючись на досліджену літературу, описано базові поняття предметної області. Виокремлено основні фактори, які необхідно враховувати при виборі фреймворку, що реалізує ІоС контейнер.

Проаналізовано відомі технічні та алгоритмічні рішення, порівняно два основних підходи до написання ІоС контейнерів, а саме статичну генерацію коду та runtime-based підхід. Вказано недоліки та переваги кожного з них та аргументовано вибір підходу для реалізації власного ІоС контейнера.

Виконано опис та аналіз відомих програмних рішень, які представлені на ринку. Проведено порівняльний аналіз існуючих фреймворків та продукту, що розробляється. Зокрема проаналізовано існуючі фреймворки з точки зору їхньої підтримки умовного створення об'єктів. Виявлено, що ця функціональність є не достатньо розвиненою у більшості програмних рішень. На основі проведеного аналізу було зроблено висновок про доцільність розробки сучасного ІоС контейнера з розширеним набором інструментів для управління умовами створення об'єктів.

За допомогою BPMN діаграм проведено детальне моделювання основних бізнес-процесів, які повинен підтримувати сучасний ІоС контейнер, а саме: процес ініціалізації та заповнення контексту контейнера та процес отримання об'єкта з ІоС контейнера.

2 РОЗРОБЛЕННЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Варіанти використання програмного забезпечення

Для виявлення вимог до програмного забезпечення, що розробляється використаємо діаграму варіантів використання з ключовими акторами та основними функціями. Діаграма варіантів використання знаходиться у графічних матеріалах.

В таблицях 2.1–2.9 наведено опис варіантів використання програмного забезпечення Contextor.

Таблиця 2.1 – Варіант використання UC-01

Use case name	Виявити компоненти.
Use case ID	UC-01
Goals	Автоматично виявляти та реєструвати всі анотовані компоненти в заданому пакеті.
Actors	Користувач
Trigger	Контейнер ініціалізовано або йому надано новий пакет для сканування.
Pre-conditions	Надано шлях до цільового пакета.
Flow of Events	Користувач запускає контейнер Contextor. Контейнер зчитує вказаний шлях до пакета. Контейнер сканує пакет на наявність класів. Далі для кожного класу контейнер перевіряє наявність анотації @Element. Коли клас знайдено, контейнер реєструє клас у своєму внутрішньому реєстрі. Контейнер реєструє кожен виявлений компонент. Виявлення завершено, і контейнер повідомляє про кількість зареєстрованих компонентів.
Extension	Якщо шлях до пакета не знайдено, то контейнер видає помилку PackageNotFoundException. Якщо файл класу неможливо завантажити, то контейнер реєструє помилку ClassLoadError та продовжує сканування.
Post-Condition	Усі анотовані класи в пакеті зареєстровані в IoC контейнері Contextor.

Таблиця 2.2 – Варіант використання UC-02

Use case name	Створити об'єкт.
Use case ID	UC-02
Goals	Створити екземпляр класу.
Actors	Користувач.
Trigger	Ініціалізація контейнера.
Pre-conditions	Цільовий клас зареєстровано в контексті. Існують усі необхідні залежності.
Flow of Events	Користувач запускає контейнер Contextor. Контейнер зчитує, які залежності необхідно надати екземпляру класу. Для цього перевіряє сигнатуру конструктора та визначає залежності. Контейнер рекурсивно створює всі необхідні залежності. Далі викликає конструктор класу, який намагається ініціалізувати користувач. Реєструє об'єкт відповідно до області, в якій цей об'єкт був створений (сінглтон, прототип). Контейнер повертає повністю побудований об'єкт користувачу.
Extension	Якщо залежність відсутня, то контейнер викидає <code>DependencyNotFoundException</code> потік завершується з помилкою. Якщо виявлено циклічну залежність, то контейнер викидає <code>CircularDependencyException</code> .
Post-Condition	Створюється екземпляр класу з усіма залежностями конструктора.

Таблиця 2.3 – Варіант використання UC-03

Use case name	Вирішити конфлікт залежностей.
Use case ID	UC-03
Goals	Дозволити користувачу вказувати, яку реалізацію залежності необхідно передати у випадку, коли існує кілька кандидатів
Actors	Користувач.
Trigger	Ініціалізація контейнера.
Pre-conditions	Зареєстровано більше однієї реалізацій інтерфейсу чи класу. За допомогою кваліфікатора вказано, яку саме залежність необхідно встановити.

Продовження таблиці 2.3

Flow of Events	Контейнер починає пошук об'єктів, що реалізують інтерфейс залежності. Контейнер знаходить дві чи більше реалізацій. Contextor перевіряє наявність кваліфікатора, який задав розробник при конфігурації. Далі контейнер обирає, яку саме реалізацію необхідно встановити в залежність, на основі кваліфікатора або пріоритету. Контейнер знаходить та встановлює обрану реалізацію в залежність об'єкта, що створюється. Контейнер продовжує процес створення об'єкта.
Extension	Якщо кваліфікатор не знайдено, тоді контейнер викидає <code>AmbiguousImplementationException</code> . Якщо кваліфікатор вказує на неіснуючу реалізацію, тоді контейнер викидає виняток <code>InvalidQualifierException</code> .
Post-Condition	В об'єкт встановлена правильна, вказана розробником імплементація.

Таблиця 2.4 – Варіант використання UC-04

Use case name	Створити об'єкт при виконанні заданих умов.
Use case ID	UC-04
Goals	Створювати об'єкти лише за умови виконання умов, заданих розробником.
Actors	Користувач.
Trigger	Ініціалізація контейнера.
Pre-conditions	Компоненти позначаються спеціальними анотаціями з умовою.
Flow of Events	Користувач хоче отримати об'єкт лише за умови виконання умов, які він вказав при конфігурації об'єкта. Контейнер ідентифікує компоненти, які позначені спеціальними анотаціями з вказаною умовою. Контейнер обробляє кожну умову на об'єкті та перевіряє її. Якщо всі умови виконуються, контейнер переходить до створення об'єкту. Якщо якась умова не виконується, контейнер логує проблему і причину виникнення, та не створює екземпляр класу.

Продовження таблиці 2.4

Extension	Якщо перевірка умови викидає неочікувану помилку, тоді контейнер реєструє <code>ConditionEvaluationFailedException</code> та вважає, що умова не була виконана.
Post-Condition	Створюються лише ті об'єкти, в яких умови для їх створення виконані

Таблиця 2.5 – Варіант використання UC-05

Use case name	Обробити помилки під час ініціалізації.
Use case ID	UC-05
Goals	Відловлювати та коректно обробляти помилки під час ініціалізації IoC контейнера.
Actors	Користувач.
Trigger	Ініціалізація контейнера.
Pre-conditions	Виникає помилка при запуску контейнера.
Flow of Events	Контейнер намагається створити компонент. В процесі ініціалізації об'єкта відбувається помилка. Контейнер відловлює цю помилку. Додає до повідомлення про помилку усю необхідну інформацію про причину виникнення помилки. Далі, залежно від конфігурації, або продовжує роботу, або завершує її.
Extension	
Post-Condition	Помилка оброблена, необхідна інформація надана.

Таблиця 2.6 – Варіант використання UC-06

Use case name	Отримати доступ до об'єктів у контексті.
Use case ID	UC-06
Goals	Мати можливість отримувати об'єкти з контексту.
Actors	Користувач.
Trigger	Користувач викликає функцію отримання об'єкту з IoC контейнера.
Pre-conditions	IoC контейнер ініціалізовано, помилки відсутні.

Продовження таблиці 2.6

Flow of Events	Користувач викликає відповідний метод пошуку в API контейнера Contextor. Контейнер виконує пошук об'єкта в своєму контексті. Контейнер повертає ініціалізований об'єкт з усіма залежностями користувачу.
Extension	Якщо бажаний елемент не знайдено, контейнер викидає помилку <code>ElementNotFoundException</code> .
Post-Condition	Користувач отримує коректно сформований екземпляр класу, пошук якого здійснювався.

Таблиця 2.7 – Варіант використання UC-07

Use case name	Налаштувати рівень деталізації логування.
Use case ID	UC-07
Goals	Мати можливість налаштовувати глибину логування.
Actors	Користувач.
Trigger	Запуск контейнера.
Pre-conditions	
Flow of Events	Користувач задає бажаний рівень деталізації логів. Contextor зчитує відповідну конфігурацію та застосовує вказаний рівень глибини логування для всіх операцій, що виконуються.
Extension	
Post-Condition	Користувач отримує можливість конфігурувати рівень деталізації логів.

Таблиця 2.8 – Варіант використання UC-08

Use case name	Зчитати конфігураційний файл.
Use case ID	UC-08
Goals	Мати можливість задати конфігурацію контейнера через зовнішній конфігураційний файл.
Actors	Користувач.
Trigger	Запуск контейнера.
Pre-conditions	Користувач створює коректний файл конфігурації та вказує коректний шлях до нього.

Продовження таблиці 2.8

Flow of Events	Контейнер запускається та перевіряє наявність файлу конфігурації. Зчитує параметри, задані в конфігураційному файлі. Застосовує параметри конфігурації до IoC контейнера.
Extension	В разі відсутності файлу конфігурації про це повідомляється користувачу та застосовуються налаштування за замовчуванням.
Post-Condition	Конфігурація зчитана коректно та застосована до контейнера Contextor.

Таблиця 2.9 – Варіант використання UC-09

Use case name	Інтегрувати Contextor у проєкт.
Use case ID	UC-09
Goals	Мати можливість легко підключати Contextor до проєктів.
Actors	Користувач.
Trigger	Розробник включає Contextor як залежність у конфігурацію проєкту.
Pre-conditions	Фреймворк Contextor опубліковано у репозиторії. Інструмент збірки проєкту коректно налаштований та має доступ до репозиторію.
Flow of Events	Інструмент збірки отримує бібліотеку Contextor із зазначеного репозиторію. Contextor завантажується та додається до Classpath проєкту. Користувач підключає необхідну базову конфігурацію для запуску IoC контейнера.
Extension	
Post-Condition	Проєкт успішно зібрано, залежність Contextor викачана з репозиторію та доступна до використання в проєкті.

2.2 Розроблення функціональних вимог

Програмне забезпечення розділене на модулі. Кожен модуль має свій певний набір функцій. У таблиці 2.10 наведено загальну модель з переліком функціональних вимог до програмного забезпечення, що розробляється.

Таблиця 2.10 – Загальна модель вимог

№	Назва	ID Вимоги	Пріоритети	Ризики
1	Виявлення компонентів	FR-1	Високий	Низький
1.1	Сканування на основі анотацій	FR-2	Високий	Низький
1.2	Сканування на основі пакетів	FR-3	Високий	Низький
2	Створення об'єктів	FR-4	Високий	Середній
2.1	Впровадження на основі конструктора	FR-5	Високий	Середній
2.3	Рекурсивне вирішення залежностей	FR-6	Високий	Високий
3	Вирішення конфліктів	FR-7	Високий	Середній
3.1	Прив'язка до конкретної реалізації	FR-8	Високий	Середній
3.2	Динамічний вибір реалізації	FR-9	Середній	Середній
4	Умовне створення об'єктів	FR-10	Високий	Високий
4.1	Умовне створення за часом	FR-11	Середній	Середній
4.2	Умовне створення за змінною у середовищі	FR-12	Високий	Низький
4.3	Умовне створення за власною умовою	FR-13	Середній	Високий
4.4	Умовне створення за наявності заданого файлу	FR-14	Низький	Низький
4.5	Умовне створення об'єктів за наявністю зв'язку із зовнішнім сервісом	FR-15	Середній	Низький
5	Обробка помилок	FR-16	Високий	Середній
5.1	Помилка відсутньої залежності	FR-17	Високий	Середній
5.2	Помилка оцінки умови	FR-18	Високий	Середній
6	Доступ до об'єктів у контексті	FR-19	Високий	Низький
6.1	Доступ за класом	FR-20	Високий	Низький
6.2	Доступ за іменем	FR-21	Середній	Низький
7	Підтримка логування	FR-22	Середній	Низький
7.1	Логування процесу створення об'єктів	FR-23	Середній	Низький
7.2	Логування результатів умовних перевірок	FR-24	Середній	Низький
7.3	Конфігурація рівня логування	FR-25	Середній	Низький
8	Зчитування конфігурації	FR-26	Високий	Середній

Продовження таблиці 2.10

8.1	Зчитування конфігураційного файлу	FR-27	Високий	Середній
8.2	Зчитування змінних середовища	FR-28	Середній	Середній
9	Інтеграція застосунку з фреймворком	FR-29	Високий	Середній
9.1	Завантаження залежностей	FR-30	Високий	Низький
9.2	Ініціалізація проєкту	FR-31	Високий	Середній
9.3	Налаштування конфігурації	FR-32	Високий	Середній
10	Покращена звітність про помилки	FR-33	Середній	Середній
10.1	Логування стектрейсу помилок	FR-34	Середній	Низький
10.2	Аналіз винятків	FR-35	Середній	Середній
11	Користувацькі анотації	FR-36	Високий	Середній
11.1	Конфігурація на основі анотацій	FR-37	Високий	Середній
11.2	Умовна логіка на основі анотацій	FR-38	Високий	Середній

Далі в таблицях 2.11 – 2.21 наведений опис ключових функціональних вимог до програмного забезпечення.

Таблиця 2.11 – Функціональна вимога FR-1

Назва	Опис
FR-1	Контейнер здатний автоматично виявляти компоненти в програмі шляхом сканування заданих пакетів. Він реєструє ці компоненти на основі попередньо визначених анотацій, що дозволяє безперешкодно інтегрувати класи без ручного налаштування. Ця функція зменшує обсяг шаблонного коду та покращує модульність, централізовано керуючи реєстрацією компонентів.

Таблиця 2.12 – Функціональна вимога FR-4

Назва	Опис
FR-4	Контейнер керує життєвим циклом та створенням екземплярів об'єктів, включаючи вирішення залежностей та застосування логіки ініціалізації. Це гарантує, що залежності вводяться в правильному порядку, а об'єкти будуть завжди доступні за потреби. Для цього Contextor обробляє всі залежності об'єкту та гарантує, що усі об'єкти будуть створені коректно.

Таблиця 2.13 – Функціональна вимога FR-7

Назва	Опис
FR-7	Contextor здатен вирішувати конфлікти, що виникають при ініціалізації об'єктів. Якщо доступно кілька імплементацій залежності, фреймворк повинен вирішити, яку саме з них використовувати. Contextor дозволяє розробникам вказувати бажану реалізацію або віддати вирішення на бік контейнера. Такий підхід особливо корисний, коли різні реалізації одного інтерфейсу потрібні за різних обставин. Якщо конфлікт вирішити неможливо, контейнер видає помилку з детальним описом проблеми.

Таблиця 2.14 – Функціональна вимога FR-10

Назва	Опис
FR-10	Умовне створення об'єктів контролює створення екземплярів об'єктів на основі певних умов. Contextor надає розробнику розширений набір інструментів для вказування умов ініціалізації об'єкту, зокрема, умов за часом, змінними середовища, наявності файлу, доступності зовнішнього вебсервісу тощо. Така функціональність дозволяє здійснювати детальний контроль над життєвими циклами об'єктів та швидко задавати умови, за яких той чи інший об'єкт повинен бути створений.

Таблиця 2.15 – Функціональна вимога FR-16

Назва	Опис
FR-16	Фреймворк Contextor повинен вміти обробляти помилки, що виникають під час його роботи. Більшість помилок буде зустрічатися при ініціалізації контейнеру. Типовими помилками є відсутні необхідні залежності, циклічні залежності, помилки при перевірці умов створення об'єкту. Contextor здатен перехоплювати ці помилки, додавати необхідну інформацію щодо причин виникнення проблеми та відображати їх користувачу. Таким чином, контейнер надає структуровані повідомлення про помилки та логує їх для допомоги розробнику в налагодженні та усуненні цих помилок.

Таблиця 2.16 – Функціональна вимога FR-19

Назва	Опис
FR-19	Доступ до об'єктів у контексті дозволяє розробникам отримувати об'єкти, якими керує контейнер, за допомогою різних методів доступу, таких як тип класу або унікальне ім'я. Також фреймворк Contextor дозволяє враховувати випадки, коли існує кілька екземплярів одного класу, повертаючи список таких об'єктів. Варто зазначити, що для забезпечення інкапсуляції доступ до об'єктів надається лише через визначені інтерфейси.

Таблиця 2.17 – Функціональна вимога FR-22

Назва	Опис
FR-22	Контейнер логує всі події, що відбуваються при ініціалізації контейнеру та його подальшій роботі. Відстеження того, що контейнер робить під капотом, є важливим для можливості аналізу подій, що відбуваються. Contextor реєструє ключові операції, такі як створення об'єктів, результати перевірок умов при створенні об'єктів, конфігураційні налаштування. Такий підхід дає змогу розробнику легко розуміти потік команд, що виконуються. Також Contextor дозволяє розробникам вказувати бажану глибину деталізації логування.

Таблиця 2.18 – Функціональна вимога FR-26

Назва	Опис
FR-26	Contextor повинен вміти зчитувати параметри конфігурації під час запуску, застосовуючи їх до налаштувань програми. Такий підхід забезпечує узгоджену, більш гнучку поведінку додатку, дозволяючи гнучкі налаштування на основі середовища розгортання або конфігураційних файлів. Contextor підтримує один із найпопулярніших на сьогодні конфігураційних форматів YAML.

Таблиця 2.19 – Функціональна вимога FR-29

Назва	Опис
FR-29	Необхідно мати можливість швидко підключати фреймворк Contextor як до існуючих проєктів, так і до новостворених. Для цього передбачено легкий процес інтеграції Contextor до застосунку за допомогою автоматизованого інструменту для збірки проєктів. Від розробника вимагається мінімально зусиль, щоб забезпечити безшовну міграцію з мінімумом налаштувань та реструктуризацій існуючої кодової бази.

Таблиця 2.20 – Функціональна вимога FR-33

Назва	Опис
FR-33	Коли щось йде не так, Contextor не просто видає загальне повідомлення про помилку. Він реєструє повний стек, показуючи, де саме та чому виникла проблема. Він також може аналізувати винятки та пропонувати можливі виправлення, що полегшує розробникам процес усунення помилок і вирішення проблем.

Таблиця 2.21 – Функціональна вимога FR-36

Назва	Опис
FR-36	У фреймворк Contextor повинна бути закладена можливість розширення функціоналу. Саме тому розробникам надається можливість створювати власні анотації для обробки об'єктів за необхідною їм бізнес логікою. Така розширюваність дозволить фреймворку задовільнити специфічні випадки використання, не передбачені початково.

У таблиці 2.22 наведено матрицю трасування вимог.

Таблиця 2.22 – Матриця трасування вимог

	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8	UC-9
FR-1	+		+						
FR-2	+		+						
FR-3	+		+						
FR-4		+							
FR-5		+							
FR-6		+							
FR-7			+						
FR-8			+						
FR-9			+						
FR-10				+					
FR-11				+					

Продовження таблиці 2.22

FR-12				+					
FR-13				+					
FR-14				+					
FR-15				+					
FR-16					+				
FR-17					+				
FR-18					+				
FR-19						+			
FR-20						+			
FR-21						+			
FR-22							+		
FR-23							+		
FR-24							+		
FR-25							+		
FR-26								+	
FR-27								+	
FR-28								+	
FR-29									+
FR-30									+
FR-31									+
FR-32									+
FR-33					+		+		
FR-34					+		+		
FR-35					+		+		
FR-36		+		+					
FR-37		+		+					
FR-38		+		+					

2.3 Розроблення нефункціональних вимог

Програмне забезпечення, що розробляється, повинно:

- Мати зручний інтуїтивно зрозумілий API для взаємодії;
- Мати можливість бути інтегрованим як до нового, так і до вже існуючого проєкту;

- Мати інформативні повідомлення під час процесу ініціалізації та роботи IoC контейнеру;
- Підтримувати зберігання більше 50 об'єктів у контексті одночасно;
- Повністю реалізовувати базовий функціонал IoC (DI) контейнера.

2.4 Аналіз системних вимог

Проведемо аналіз системних вимог для програмного забезпечення, що розробляється. За основу було прийнято рішення взяти системні вимоги одного з найпопулярніших Java дистрибутивів OpenJDK 17, оскільки це сучасна LTS версія мови програмування Java, на якій буде виконано розробку фреймворку Contextor.

Перерахуємо операційні системи, які підтримують роботу з OpenJDK 17.

Підтримується на Windows Server, Windows 11 та Windows 10, усі з архітектурою x64.

Підтримується різними дистрибутивами Linux, включаючи Oracle Linux 7+, Ubuntu Linux, Red Hat Enterprise Linux 7+ та Suse Linux Enterprise Server 12. Архітектури включають x64 та AArch64, причому в більшості випадків підтримується лише режим X.org. Підтримка Wayland обмежена або взагалі не підтримується в більшості дистрибутивів. Проте, оскільки застосунок, що розробляється, не містить графічного інтерфейсу, відсутність підтримки цього протоколу ніяк не впливає на працездатність фреймворку Contextor.

Підтримується на macOS 13 та новішому, на архітектурі як x64, так і для AArch64.

Також підтримується віртуалізація за допомогою Oracle VM, Docker та Microsoft Hyper-V Server.

2.5 Аналіз економічних показників програмного забезпечення

Для аналізу економічних показників застосуємо методику оцінки COSOMO 2.0 [14].

Для розрахунку метрики *Effort* скористаємося формулою, запропонованою у методиці оцінки COCOMO 2.0:

$$Effort = 2.94 * EAF * (KSLOC)^E, \quad (2.1)$$

де *Effort* – це оціночна кількість людино-місяців, тобто кількість місяців, які необхідні одній людині для завершення роботи;

2.94 – мультиплікативна константа, яка описана в моделі COCOMO 2.0;

EAF – коефіцієнт коригування зусиль. Цей фактор враховує так звані «фактори витрат», які впливають на зусилля для виконання проєкту. *EAF* є продуктом мультиплікації усіх цих зусиль;

KSLOC – орієнтовна кількість тисяч строк вихідного коду;

E – коефіцієнт масштабування, який для проєктів середньої складності становить 1.05.

Розрахуємо коефіцієнт коригування зусиль, для чого занесемо основні фактори в таблицю.

Таблиця 2.23 – Мультиплікатори зусиль

Фактор	Коефіцієнт (М)	Деталі
Необхідна надійність	1.10	Висока завдяки ретельній обробці помилок та оцінці умов
Складність	1.15	Складна рекурсивна логіка DI
Досвід	0.95	Кваліфікований розробник
Платформа	1.00	Стандартна розробка на Java
Обмеження графіка	1.05	Помірні терміни

Користуючись таблицею 2.23, розрахуємо *EAF* за формулою:

$$EAF = M1 * M2 * M3 * M4 * M5, \quad (2.2)$$

$$EAF = 1.10 * 1.15 * 0.95 * 1.00 * 1.05 \approx 1.26$$

Далі, в таблиці 2.24 опишемо основні компоненти та оцінимо кількість строк вихідного коду кожного з них.

Таблиця 2.24 – Оцінка строк коду основних компонентів

Компонент	Оціночна SLOC	Деталі
Виявлення компонента	500	Пошук класів з певною анотацією по файловому дереву.
Створення об'єктів	350	Рекурсивна обробка залежностей.
Вирішення конфліктів	300	Виконує пошук об'єктів в контексті для вирішення конфліктних ситуацій.
Обробка циклічних залежностей	250	Знаходить та повідомляє про циклічні залежності між об'єктами.
Умовне створення об'єктів	400	Оцінює умови створення об'єктів.
Обробка помилок	250	Обробляє помилки, надаючи користувачу інформативні повідомлення та можливі шляхи виправлення.
Доступ об'єктів у контексті	300	Надає користувачу доступ до створених ІоС контейнером об'єктів.
Логування	150	Логує інформацію про хід виконання програми.
Зчитування конфігурації	200	Зчитує конфігурацію проєкта із зовнішнього джерела.

Отже, з таблиці 2.24 розраховуємо кількість *KLOC* за формулою:

$$KLOC = \frac{\sum SLOCK_i}{1000} \quad (2.3)$$

$$KLOC = \frac{2700}{1000} = 2.7$$

Тепер, маючи усі необхідні параметри, розрахуємо значення *Effort* за формулою 2.1

$$Effort = 2.94 * 1.26 * (2.7)^{1.05} \approx 10.5$$

Отримали значення 10.5 людино-місяців на виконання розробки.

2.6 Постановка завдання на розробку програмного забезпечення

В рамках розробки програмного забезпечення Contextor необхідно розробити ІоС фреймворк. Необхідно розробити легко розширювану архітектуру застосунку, враховуючи кращі практики програмування та дотримуючись принципів ООП та SOLID. Застосунок повинен підтримувати конфігураційні налаштування через механізм анотацій. Для цього він повинен використовувати спеціальний модуль, який дозволить зчитувати анотації та метадані, які вони містять.

Застосунок повинен підтримувати усі базові функціональні вимоги, що висуваються до сучасних ІоС контейнерів, та додатково надавати розширений функціонал для керування умовами створення об'єктів.

Розробка призначена для автоматизації життєвого циклу об'єктів та полегшення механізмів контролю над умовами їх створення за допомогою ІоС контейнера.

Метою розробки є спрощення реалізації функціоналу та тестування програмних компонентів шляхом автоматизації управління життєвим циклом об'єктів на основі ІоС підходу.

Мета досягається шляхом реалізації наступних функціональних задач:

- Застосунок повинен мати функціональність для виявлення компонентів, які необхідно створити та додати в контекст ІоС контейнеру;

- Contextor повинен створювати об'єкти, вирішуючи при цьому проблеми із залежностями;
- Застосунок повинен підтримувати функціонал для запобігання конфліктів, що можуть виникати при вирішенні залежностей;
- Застосунок повинен надавати користувачу набір інструментів для задання умов, за яких необхідно створювати екземпляр класу;
- Contextor обробляє помилки та надає користувачу інформативні повідомлення про причину проблеми;
- Застосунок повинен мати можливість зчитати та застосувати зовнішню конфігурацію.

Висновки до розділу

В другому розділі дипломної роботи було виділено та розроблено вимоги до програмного забезпечення, що розробляється, зокрема:

- створено діаграму варіантів використання ІоС контейнера, за допомогою якої охоплено всі ключові аспекти взаємодії користувача з системою – від виявлення компонентів і створення об'єктів до обробки помилок, конфігурування та інтеграції контейнера у зовнішні проєкти;
- розроблено функціональні вимоги, які сучасний ІоС контейнер повинен підтримувати, що дозволяє структурувати систему за модулями, врахувати пріоритети реалізації та ризики, пов'язані з кожною вимогою;
- розроблено нефункціональні вимоги до програмного забезпечення. Вказано важливі вимоги, які необхідно реалізувати;
- додатково проаналізовано системні вимоги для коректної роботи ІоС контейнера.

Також здійснено аналіз економічних показників ПЗ, що розробляється, для чого була застосована методика оцінки СОСОМО 2.0. Згідно з даною методикою було розраховано, що на виконання розробки необхідно витратити 10.5 людино-місяців. Ці показники є прийнятними та цілком досяжними.

За результатами розділу сформовано технічне завдання на розробку ПЗ.

3 КОНСТРУЮВАННЯ ТА РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Архітектура програмного забезпечення

Для побудови архітектури ІоС контейнера було визначено основні модулі, які повинні бути реалізовані. Для більшої зручності зобразимо ці модулі за допомогою нотації С4, зокрема, розглянемо три перших рівня цієї моделі.

Почнемо з першого рівня моделі С4 – контекстної діаграми системи, наведеною на рисунку 3.1. Діаграма системного контексту є чудовим способом почати документування системи програмного забезпечення, оскільки дозволяє побачити загальну картину архітектури [15].

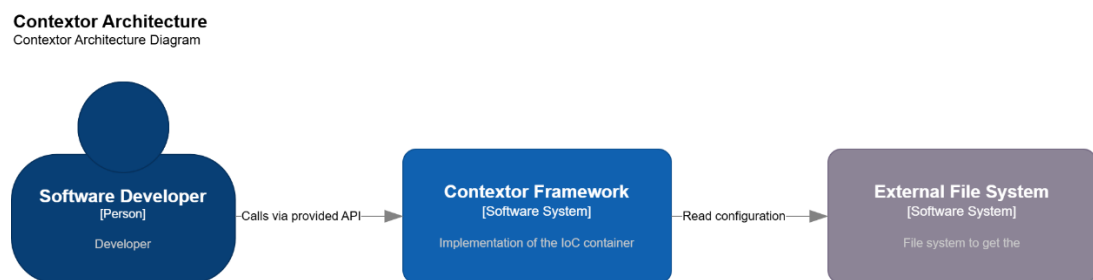


Рисунок 3.1 – Діаграма системного контексту фреймворку Contextor

З діаграми можна отримати загальне розуміння про системи, що взаємодіють між собою, а саме:

- актором виступає розробник програмного забезпечення;
- система Contextor фреймворк реалізує та виконує функції ІоС контейнера;
- зовнішня файлова система використовується для зчитування конфігурації та перевірок умов створення об’єктів, наприклад, перевірки наявності певного файлу в дереві файлів.

Далі перейдемо до другого рівня діаграми С4 – діаграми контейнерів. Контейнерна діаграма дозволяє зобразити високорівневу форму архітектури

- Configuration Reader – модуль, який відповідає за зчитування конфігурації з різних джерел, таких як змінні середовища та конфігураційний файл;
- Component Scanner – модуль, що проводить сканування проєкту для пошуку класів, які необхідно створити. Використовує рекурсійний підхід для обходу дерева файлів;
- Dependency Resolver – модуль, який відповідає за вирішення залежностей об'єктів, які необхідно створити. Здатен виявляти, які саме залежності потрібно створити для того чи іншого класу, будувати граф залежностей та виявляти циклічні залежності;
- Condition Evaluator – відповідає за перевірку умов при створенні об'єкта. Зчитує відповідні умовні анотації та їх метадані і здатен проводити оцінку виконання умов;
- Logger – модуль, який відповідає за запис логів у консоль та лог файл. Підтримує різні рівні деталізації логування;
- Object Factory – фабрика для створення об'єктів, саме цей модуль виконує створення об'єкту та фасилітує виклики до інших модулів, які надають необхідні дані для успішного створення нового об'єкту.

Процес того, як саме відбувається взаємодія між модулями, детально зображено на рисунку 3.2.

Далі більш детально розглянемо архітектуру та взаємодію модулів для створення об'єктів, вирішення залежностей та перевірки виконання умов при створенні об'єкту. Для цього використаємо третій рівень моделі рівень моделі C4 – діаграму компонентів. Ця діаграма корисна тим, що дозволяє збільшити деталізацію опису конкретного модуля, чи в даному випадку групи модулів, при цьому вказуючи основні структурні блоки та взаємодію між ними. Діаграма компонентів фреймворку Contextor винесена в графічні матеріали.

Опишемо компоненти більш детально.

- Object Factory – компонент, який займається створенням об'єктів. Взаємодіє з іншими компонентами для отримання необхідної інформації про

об'єкт. Додатково налагоджена взаємодія із компонентом Object Scope Manager для отримання області, в якій необхідно створити об'єкт;

- Object Scope Manager – компонент, який визначає область, в якій буде створено об'єкт. Підтримує дві області створення, а саме Singleton – екземпляр об'єкту створюється лише один раз під час ініціалізації контейнера та зберігається у контексті. Ще однією областю об'єкта є Prototype – створюється новий екземпляр кожного разу при спробі отримати цей об'єкт з контексту;

- Dependency Resolver API – компонент, який надає зручний API для роботи із залежностями об'єкта. Виконує виклики сервісів, таких як Cyclic Dependency Checker, Injectable Constructor Processor та Dependency graph Builder;

- Injectable Constructor Processor – компонент, що визначає, який конструктор необхідно використати для інстанціювання об'єкту. Визначивши, конструктор збирає список усіх необхідних залежностей, які потрібно передати в нього;

- Dependency Graph Builder – компонент, який відповідає за побудову графу залежностей;

- Cyclic Dependency Checker – компонент, основною задачею якого є визначення наявності циклічних залежностей в побудованому графі;

- Conditional Evaluator API – надає зручний API для перевірки умов при створенні об'єктів. Включає в себе взаємодію з компонентами Condition Collector та Condition Evaluator.

3.2 Архітектурні рішення та обґрунтування вибору засобів розробки

Розглянемо загальні підходи та патерни, які були використані при проєктуванні системи.

Основні задачі, які вирішує IoC контейнер, можна поділити на дві категорії:

Перший тип – інфраструктурні, наприклад, розв’язання залежностей, зчитування конфігурації і т. д. Такі задачі зазвичай вирішуються за допомогою стандартних механізмів мови програмування.

Другий тип – додаткова бізнес-логіка, пов’язана з об’єктами, які описує користувач. Тут яскравим прикладом може слугувати механізм умовного створення об’єктів. Тобто для реалізації такого функціоналу необхідно додавати певну логіку до об’єктів.

При розробці ІоС контейнера постало питання, як саме додавати функціональності об’єктам, при цьому не змінюючи логіку, яка була закладена в них розробником.

Основним способом, за допомогою якого можна легко зробити таке розширення логіки для об’єкта, є застосування патерну проксі. Це один із найбільш відомих та популярних шаблонів проєктування, описаних у книзі «Design Patterns: Elements of Reusable Object-Oriented Software» [3]. Отже, проксі – це структурний дизайн патерн, який дозволяє створювати спеціальні об’єкти-замінники та підставляти їх замість оригінальних об’єктів [15].

Зобразимо типову реалізацію цього шаблону проєктування за допомогою UML діаграми класів (рисунок 3.3).

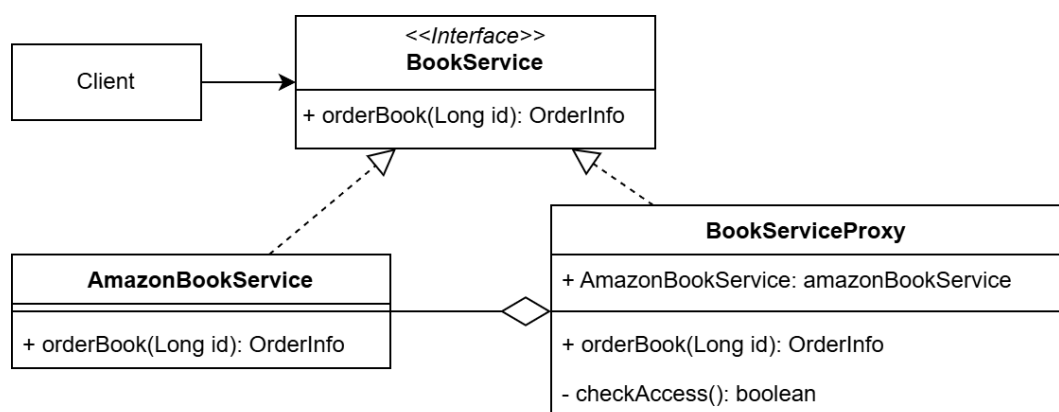


Рисунок 3.3 – Діаграма класів структурного шаблону проєктування проксі

Розглянемо, які переваги надає реалізація такого патерну на прикладі, зображеному на рисунку 3.3. Отже, перевагами такого підходу є:

- контроль над об'єктом користувача здійснюється непомітно та не впливає на клієнтський код. Це основна та надзвичайно важлива особливість цього патерну, який можна використати для побудови IoC контейнера. В нашому прикладі, користувач знає лише про інтерфейс BookService та його імплементацію AmazonBookService. Всі деталі реалізації проксі класу інкапсульовані від клієнтського коду;

- дозволяє контролювати життєвий цикл службового об'єкту. Наприклад, можна організувати ліниву ініціалізацію об'єкту, який ми проксіємо. В прикладі, що розглядаємо, можливо, не потрібно створювати об'єкт AmazonBookService, поки до нього не було звернень;

- дозволяє огортати оригінальні методи класу, додаючи функціональності. У наведеному прикладі за допомогою BookServiceProху можливо реалізувати перевірку доступу до сервісу. Такий підхід часто використовується у веб застосунках в разі необхідності перевіряти ролі та доступи користувача до певних операцій.

У фреймворку Contextor такий підхід зокрема використовується для моніторингу та пошуку «вузьких місць» (bottleneck) у коді, які можуть бути вдосконалені. Для цього фреймворк надає можливість створити проксі клас та огорнути методи з додаванням бізнес-логіки із заміром часу виконання методів, для подальшого визначення необхідності їх оптимізації.

Ще одним важливим патерном проєктування, що застосовується при розробці IoC контейнера Contextor, є ланцюжок обов'язків (англ chain of responsibility). Це шаблон проєктування, який належить до поведінкових. Загальна ідея цього патерну полягає в передачі певного запиту на обробку через ланцюг обробників.

Кожен обробник має специфічну, закладену в нього бізнес-логіку, та знає лише, як обробити запит, що до нього прийшов, та як передати його далі. В класичному варіанті цей патерн здебільшого використовується для уникнення написання складних за структурою умовних операторів. Особливо

це проявляється, коли запит необхідно обробити одразу декількома, а не одним процесором.

У фреймворку Contextor найбільш наочним прикладом застосування такого патерну є процес перевірки умов при створенні об'єктів. Для більшого розуміння цю задачу можна розбити на декілька окремих завдань, а саме:

Надати клієнту можливість обирати режим перевірки умов. Перший – режим «короткого замикання», тобто, якщо хоч одна умова не виконується, перевірки завершуються. Другий режим – без «короткого замикання», тобто перевірки продовжуються навіть після першої невдалої. Такі режими корисні для різних сценаріїв. Наприклад, якщо клієнт хоче мати можливість переглядати результати перевірок за всіма умовами, йому необхідно задати другий режим.

Зберегти інформацію про результати перевірок.

Надати користувачу можливість додавати власні умови створення об'єктів та їх обробники.

Тепер наведемо UML діаграму класів, за допомогою яких було вирішено наведені вище задачі для перевірки умов при створенні об'єктів (рисунк 3.4).

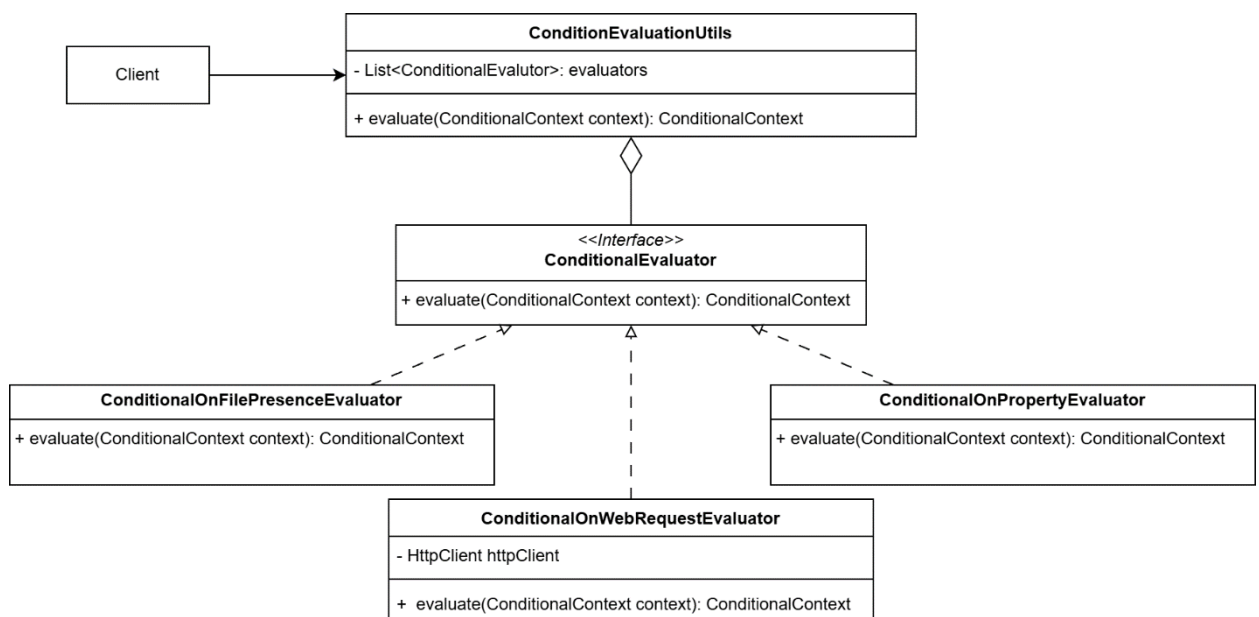


Рисунок 3.4 – Діаграма класів для перевірки умов створення об'єкту

Опишемо елементи, наведені на рисунку, більш детально:

`ConditionalEvaluator` – спільний інтерфейс для обробників, що описує контракт, який повинен реалізовувати кожен процесор умов. Містить єдиний метод *evaluate*, який в свою чергу приймає `ConditionalContext` як параметр. Зупинимось на контексті, який передається між обробниками, більш детально.

`ConditionalContext` – клас, який містить у собі всю необхідну інформацію щодо умовних перевірок. Саме цей клас передається між обробниками та дозволяє їм використовувати уніфікований вхідний та вихідний формати. Опишемо структуру класу `ConditionalContext`:

```
public class ConditionalContext {
    // Клас, який підлягає умовній перевірці
    Class<?> elementClass;
    // Прапорець, який вказує, чи увімкнено "переривання ланцюга"
    boolean isCircuitBreaker;
    // Прапорець, який вказує, чи пройшла умовна перевірка
    boolean conditionalCheckPassed;
    // Список результатів умовних перевірок
    List<String> conditionalCheckResults;
}
```

Конкретні реалізації інтерфейсу `ConditionalEvaluator`. На рис. 3.4 наведено декілька реалізацій процесорів, що виконують перевірку умов. Розглянемо логіку, за якою працюють ці обробники на прикладі `ConditionalOnFilePresenceEvaluator`. Для спрощення розуміння принципу роботи наведемо псевдокод даного процесора.

```
CLASS ConditionalOnFilePresenceEvaluator IMPLEMENTS ConditionalEvaluator

METHOD evaluate(context)
    element = context.getElementClass()

    IF element DOES NOT HAVE annotation "ConditionalOnFilePresence" THEN
        RETURN context
    ENDIF
```

```

filePaths = get "filePaths" from annotation metadata OR empty list

FOR EACH path IN filePaths DO
  IF path NOT found in classpath AND NOT found in file system THEN
    context.setCheckPassed(false)
    context.addResult("File " + path + " does not exist")
  RETURN context
ENDIF
ENDFOR

RETURN context
ENDMETHOD

ENDCLASS

```

Як бачимо, логіка обробки доволі проста та описується в декілька кроків.

1. Перевіряємо, чи присутня на класі з контексту анотація, яку обробляє наш процесор.
2. Якщо анотація відсутня пропускаємо обробку, передаємо контекст іншому процесору.
3. Інакше виконуємо обробку та перевіряємо задані умови. Записуємо результат перевірки в контекст та передаємо його далі.

Тепер, наведемо принцип роботи класу `ConditionEvaluationUtils`, який фасилітує та запускає ланцюжок з обробників. Отже, цей клас виконує декілька важливих функцій, а саме:

1. Зберігає ланцюжок з обробників для перевірки умов, які зазначені на об'єкті.
2. Ініціалізує початковий контекст перевірки, який буде передаватись між обробниками
3. Запускає процес перевірки об'єктів та слідкує за ходом його виконання.

Реалізація цих трьох задач наведена на рисунку 3.5.

```

public class ConditionEvaluationUtils {

    private static final List<ConditionalEvaluator> conditionalEvaluator =
List.of(
    new ConditionalOnFilePresenceEvaluator(),
    new ConditionalOnPropertyEvaluator(),
    new ConditionalOnLinuxEvaluator(),
    new ConditionalOnWindowsEvaluator(),
    new ConditionalOnWebRequestEvaluator()
    );

    public static ConditionalContext evaluate(Class<?> elementType) {
        var context = initializeConditionalContext(elementType);

        for (ConditionalEvaluator evaluator : conditionalEvaluator) {
            context = evaluator.evaluate(context);
            if (!context.isConditionalCheckPassed() && context.isCircuitBreaker())
            {
                log.warn("Circuit breaker is on and evalutaion failed. Skipping rest
{} evaluations", elementType.getSimpleName());
                return context;
            }
        }

        return context;
    }
}

```

Рисунок 3.5 – Лістинг класу ConditionEvaluationUtils

Окрім вже перерахованих переваг, найбільшою та найважливішою перевагою такого підходу в контексті розробки IoC контейнера є можливість надати користувачу способи розширення функціоналу. Тобто, користувач може створити свій обробник умовних анотацій відповідно до контракту ConditionalEvaluator. Далі достатньо просто додати створений обробник до списку вже існуючих обробників, а все інше виконає інфраструктура фреймворку Contextor. Це дозволяє фреймворку залишатися актуальним та вирішувати будь-які, навіть найскладніші задачі, пов'язані з умовним створенням об'єктів, які можуть виникнути при розробці. Саме тому Contextor надає розробникам можливість створювати власні анотації та обробники, не обмежуючись рішеннями, які йдуть від фреймворку «з коробки».

Далі обґрунтуємо вибір мови програмування, за допомогою якої буде реалізовано IoC контейнер.

Основними варіантами при виборі були мови програмування C++, Python, Rust, Typescript та Java. Надамо короткий опис кожної з мов.

C++ – це статично типізована мова програмування загального призначення, яка підтримує всі популярні на сьогоднішній день парадигми програмування, включаючи процедурне, об'єктно-орієнтоване програмування [16]. C++ дуже широко використовується в системному програмуванні, розробці складних ігор, вбудованих систем і програм, в яких швидкодія є критичною, наприклад, в системах фінансової торгівлі. Проте, хоча C++ і пропонує широкий набір об'єктно-орієнтованих можливостей, ця мова програмування не є найкращим вибором для реалізації ІоС контейнера. Їй бракує механізмів рефлексії та обробки анотацій, які є критичними для автоматичного виявлення та керування залежностями в контейнері. Реалізація таких функцій вимагала б інтенсивного використання шаблонного метапрограмування та складних макросів, що з часом призвело б до складного та важко підтримуваного коду.

Python – це динамічно типізована мультипарадигмена мова програмування, яка підтримує об'єктно-орієнтоване, процедурне та функціональне програмування [17]. Через свою простоту, невисокий поріг входу та велику кількість доступних бібліотек ця мова програмування протягом останніх 10 років швидко набирає популярності. Сьогодні Python широко використовується для аналізу даних, машинного навчання, веб-розробки та автоматизації процесів. Python надає можливості динамічної типізації та метапрограмування, які можуть бути корисними при розробці ІоС контейнера, але відсутність інтерфейсів і строгої типізації може призвести до менш структурованого та менш передбачуваного керування залежностями.

Rust – це статично типізована мультипарадигмена мова програмування з фокусом на безпеці пам'яті, паралелізмі та швидкодії [18]. Хоча це не суто об'єктно-орієнтована мова, вона підтримує принципи ООП через схожі інструменти. Rust зазвичай використовується в системному програмуванні, веб засосунках, розробці блокчейнів і високопродуктивних програмах, де безпека пам'яті та паралелізм є критичними. Основний недолік Rust при реалізації ІоС контейнера – це відсутність вбудованої підтримки анотацій та

механізмів рефлексії, які можна було б використовувати під час виконання застосунку. Це робить автоматичне виявлення компонентів та розв'язання залежностей занадто складним і громіздким процесом.

TypeScript – це статично типізований наднабір JavaScript, який надає чітку типізацію, інтерфейси та дозволяє писати код в парадигмі об'єктно-орієнтованого програмування [19]. Також варто зазначити, що TypeScript підтримує функціональне та процедурне програмування. Сьогодні TypeScript широко використовується здебільшого у фронтенд та бекенд розробці. Доволі поширена у великих корпоративних застосунках, де строга типізація та легкість підтримки коду є критичними. Проте, хоча деякі фреймворки написані на TypeScript, наприклад, NestJS [20] і забезпечують механізми, схожі на IoC, вони зазвичай покладаються на обробку метаданих на основі декоратора. Цей підхід менш надійний та важчий у підтримці, ніж можливості анотацій та рефлексії у мові Java.

Java – це статично типізована об'єктно-орієнтована мова програмування [21]. Java робить акцент саме на підтримці принципів ООП, таких як наслідування, поліморфізм, інкапсуляція та абстракція. При цьому, починаючи з Java 8, також підтримує функціональне програмування за допомогою лямбда-виразів та стандартної бібліотеки Stream API. Java дуже широко використовується в корпоративних застосунках, розробці серверних додатків, розробці під Android, фінансових та банківських системах і великомасштабних розподілених системах. Екосистема Java за роки існування накопичила величезну кількість бібліотек і фреймворків, які охоплюють майже всі домени та потреби користувачів.

Саме мова програмування Java найкраще підходить для реалізації контейнера IoC завдяки своєму розвиненому Reflection API, продвинутим механізмам для обробки анотацій та отримання метаданих з них і стандартизованим механізмам завантаження класів [22]. Ці функції полегшують автоматичне сканування компонентів, впровадження

залежностей і керування життєвим циклом об'єктів під час виконання програми.

Крім того, системи для збірки сміття (*англ.* garbage collector) і опрацювання помилок у Java застосунках забезпечують надійну основу для організації керування життєвим циклом об'єктів та обробки помилок, що можуть виникати в процесі роботи застосунку [23].

Отже, Java є найкращим варіантом для реалізації поставленої задачі створення IoC контейнера.

3.3 Конструювання програмного забезпечення

Виконаємо опис утиліт, бібліотек та іншого стороннього програмного забезпечення, яке використано у розробці IoC контейнера Contextor та наведено в таблиці 3.1.

Таблиця 3.1 – Опис утиліт, які були застосовані при розробці Contextor

№ п/п	Назва утиліти	Опис застосування
1	IntelliJ IDEA [24]	Основне середовище розробки, яке використовувалося при створенні фреймворку Contextor. Найпопулярніша та найрозвинутіша IDE для розробки застосунків мовою програмування Java. Безкоштовно надає широкий функціонал для розробки, відлагодження та профайлінгу застосунків.
2	Maven [25]	Інструмент для автоматизації збірки проекту. Виконує компіляцію вихідного коду, збірку проекту в Jar файли, запускає тести. Також задає стандартизовану структуру проекту. Використовується для підключення сторонніх бібліотек та плагінів.

Продовження таблиці 3.1

3	Logback Project [26]	Сучасний інструмент для логування, який імплементує найпопулярніший на сьогодні SL4J API. Ключовими особливостями даного проєкту є можливість тонко налаштувати його конфігурацію під специфічні задачі.
4	Project Lombok [27]	Бібліотека для автоматичної генерації так званого «шаблонного коду». Дозволяє генерувати конструктори класів, гетери та сетери, швидко підключати логер та багато інших корисних можливостей.
5	Reflections Library [28]	Бібліотека, яка надає розширений API над стандартним Reflections API у Java. Дозволяє легко зчитувати анотації та їх метадані, визначити, які інтерфейси реалізує конкретний клас та інші функції.
6	Apache Commons [29]	Надає додаткові утилітні функції для вирішення типових задач. Прикладами таких задач є операції зі строками, перевірки об'єктів на «null» значення і т.д
7	Junit 5 [30]	Найпопулярніший фреймворк для покриття коду unit-тестами. Широко застосовується в корпоративних застосунках. Надає простий та інтуїтивний інструментарій для створення тест кейсів. Додатково дозволяє підключати зовнішні модулі, для розширення

Продовження таблиці 3.1

8	Mockito [31]	Бібліотека, яка дозволяє створювати так звані «мок-об'єкти». Використовуються ці об'єкти при написанні тестів, та дозволяють задавати бажану поведінку об'єкту при цьому ізолюючи компонент який тестується від реальних залежностей. Цей фреймворк чудово інтегрований із Junit 5, що ще більше спрощує роботу з ним.
9	AssertJ [32]	Надає читабельний API для написання тверджень (англ. assertions). Робить тести більш природними та зрозумілими.

Тексти програмного коду наведені в окремому документі «Текст програми».

3.4 Аналіз безпеки даних

Спеціальних вимог до безпеки даних не висувається, однак варто зазначити, що всі функції та методи, які реалізовані у IoC контейнері, не містять «побічних ефектів» та реагують на дії користувача коректно.

Особливих вимог по надійності не висувається, відповідальність за некоректне використання програмного забезпечення Contextor покладається на користувача.

Висновки до розділу

В третьому розділі дипломної роботи було виділено та описано основні архітектурні рішення та підходи до конструювання IoC контейнера, зокрема:

Детально описано архітектуру фреймворку Contextor за допомогою моделі С4. Побудовано діаграми перших трьох рівнів: контекстну діаграму системи, діаграму модулів та діаграму компонентів.

Розглянуто архітектурні рішення, що були втілені при розробці ІоС контейнера. При цьому проведено детальний огляд основних патернів, що використовувалися для побудови масштабованої та легко підтримуваної архітектури фреймворку Contextor.

Проведено визначення сильних та слабких сторін сучасних мов програмування та обґрунтовано вибір мови програмування, що буде використана для реалізації поставленої задачі створення ІоС контейнера. Аргументовано, що найкращим варіантом, на якому можна виконати реалізацію проєкту Contextor, є одна з найпопулярніших ООП мов програмування Java.

Наведено та описано утиліти, бібліотеки та інструменти, що використовувалися при написанні програми. Вказано призначення та спосіб використання кожного з інструментів, що були задіяні в розробці.

4 АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз якості ПЗ

Для аналізу якості програмного забезпечення було вирішено застосувати статичний аналізатор коду. На ринку існує доволі багато продуктів, які здатні провести аналіз коду. Одним з найбільш популярних інструментів для мови програмування Java є платформа SonarCloud [33].

Метриками для оцінки якості ПЗ обрано наступні:

- безпека коду;
- надійність коду;
- можливість підтримки коду;
- цикломатична складність;
- покриття коду тестами.

Безпека коду – це метрика, яка надає можливість проаналізувати та виявити можливі слабкі місця, що можуть бути вразливими до хакерських атак. Результат статичного аналізу метрики «Безпека коду» наведений на рисунку 4.1.

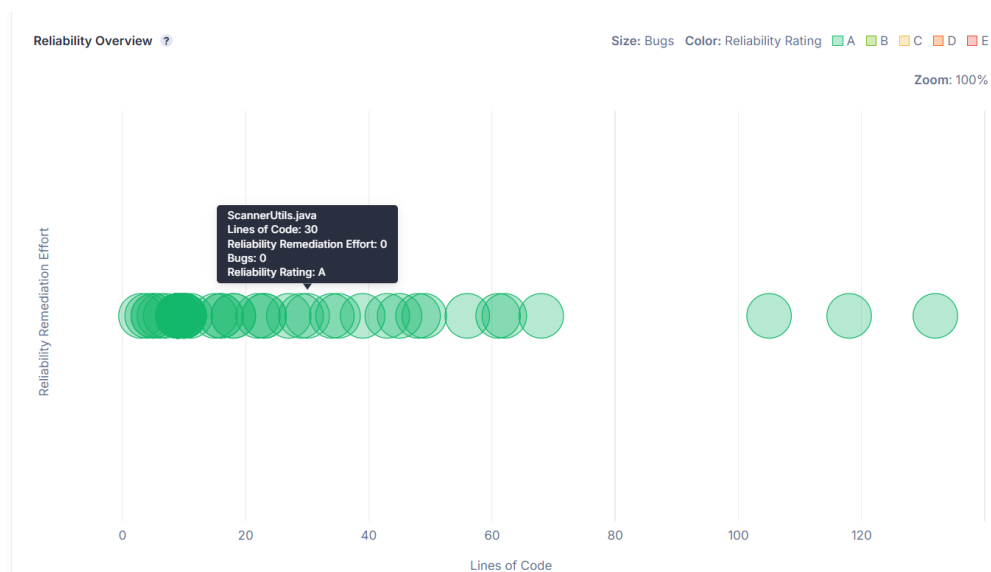


Рисунок 4.1 – Графічне зображення результату статичного аналізу коду за метрикою «Безпека коду»

Необхідно зазначити, що на рисунку 4.1 та на інших графіках кожен круг репрезентує конкретний клас у програмі. Чим більш зелений колір має круг, тим краще значення має метрика, графік якої розглядається. По вісі абсцис вказана кількість строк коду, за віссю ординат вказаний рівень зусиль, який необхідний для покращення метрики. Розмір конкретного круга вказує на складність вирішення проблеми та серйозність проблеми. Чим більший розмір, тим більша проблема.

Метрика «Надійність коду» використовується для оцінки надійності та стабільності програмного коду з часом. Використовуючи цю метрику, можна оцінити ймовірність того, що код буде працювати без збоїв за визначених умов протягом певного періоду часу. Як видно з результатів, наведених на рисунку 4.2, отримано найвищу оцінку А.

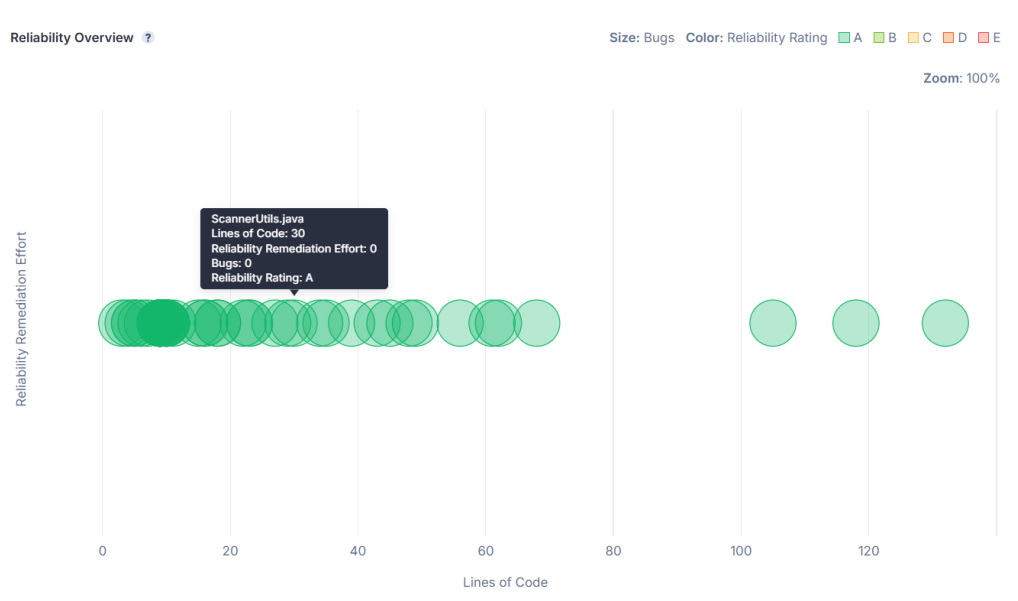


Рисунок 4.2 – Графічне зображення результату статичного аналізу коду за метрикою «Надійність коду»

Метрика «Можливість підтримки коду» використовується для оцінки того, наскільки легко буде розібратися в коді, змінити певну бізнес-логіку, виправити баг та в цілому розширити кодову базу. Чим легше підтримувати код, тим менше потрібно витратити зусиль та часу на внесення змін в існуючий

функціонал чи на додавання нового. Як видно з рисунку 4.3, метрика «Можливість підтримки коду» оцінена, як дуже висока.

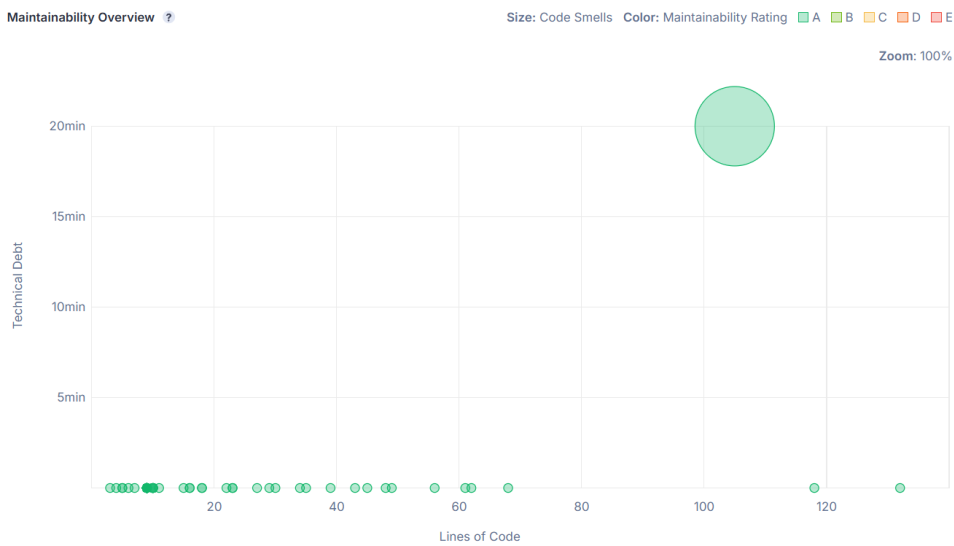


Рисунок 4.3 – Графічне зображення результату статичного аналізу коду за метрикою «Можливість підтримки коду»

Оцінку цикломатичної складності також було виконано за допомогою SonarCloud. Загальна цикломатична складність дорівнює 172. Детальні результати зазначені на рисунку 4.4.

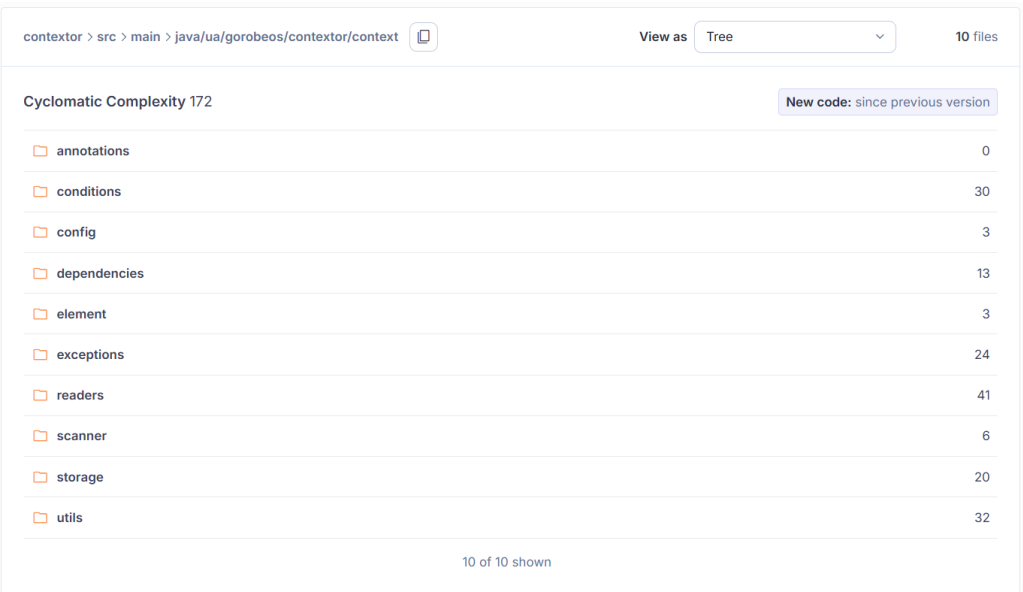


Рисунок 4.4 – Цикломатична складність пакетів фреймворку Contextor

Також код був покритий юніт та інтеграційними тестами. Всього тестовий набір налічує 62 тест-кейси. Результат виконання тестів можна побачити на рисунку 4.5.

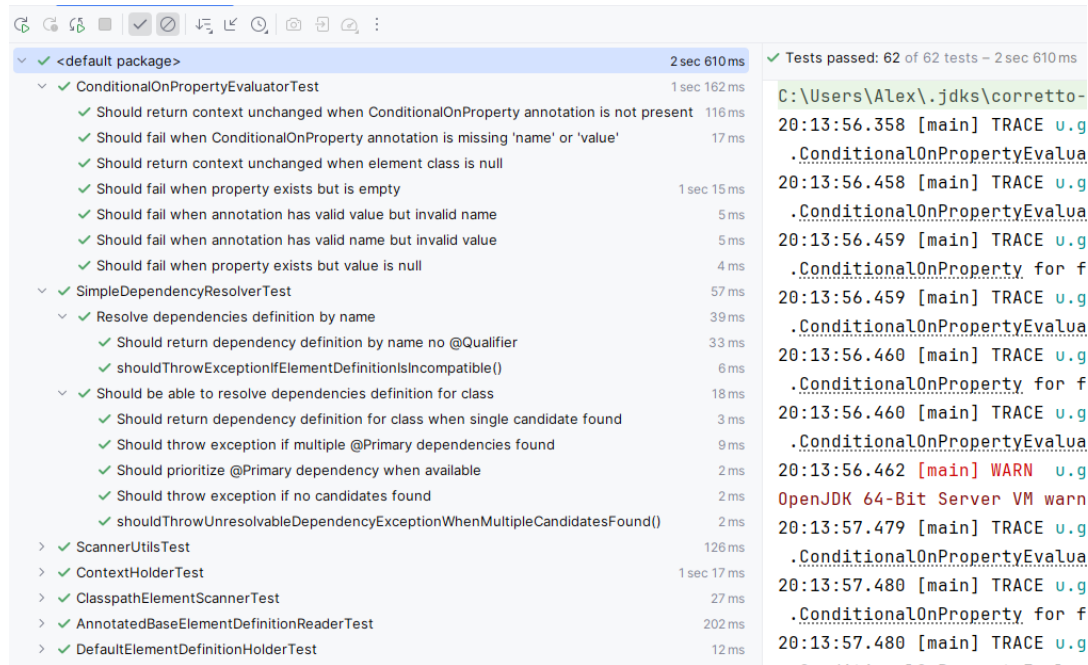


Рисунок 4.5 – Результат виконання юніт тестів

Додатково було виконано аналіз покриття кодової бази тестами. Було визначено індивідуальний відсоток покриття тестами строк коду для кожного модуля та класу, а також загальний відсоток такого покриття. Крім цього, визначено покриття усіх можливих розгалужень ходу виконання програми.

Оцінювання проводилося за допомогою плагіну JaCoCo, який дозволяє генерувати зручні інтуїтивно зрозумілі звіти [34]. Результат аналізу наведено на рисунку 4.6.

contextor

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ua.gorobeos.contextor.context.storage		76%		81%	3	23	19	98	1	15	0	2
ua.gorobeos.contextor.context.readers		89%		84%	5	36	10	111	1	23	0	4
ua.gorobeos.contextor.context.conditions evaluators		90%		75%	10	34	9	92	1	16	0	5
ua.gorobeos.contextor.context.readers.impl		81%		50%	3	12	6	27	1	10	0	1
ua.gorobeos.contextor.context.utils		95%		86%	3	35	4	80	0	24	0	2
ua.gorobeos.contextor.context.scanner		91%		100%	1	7	0	21	1	6	0	1
ua.gorobeos.contextor.context.conditions		90%		83%	1	6	2	16	0	3	0	1
ua.gorobeos.contextor.context.config		80%		n/a	0	4	2	12	0	4	0	1
ua.gorobeos.contextor.context.dependencies		100%		100%	0	10	0	36	0	5	0	1
ua.gorobeos.contextor.context.dependencies.checker		100%		100%	0	5	0	15	0	4	0	1
ua.gorobeos.contextor.context.element		100%		n/a	0	3	0	9	0	3	0	3
ua.gorobeos.contextor.context.annotations.conditions		100%		n/a	0	1	0	6	0	1	0	1
Total	246 of 2,312	89%	22 of 124	82%	26	176	52	523	5	114	0	23

Рисунок 4.6 – Покриття коду тестами, звіт JaCoCo

4.2 Опис процесів тестування

Тестування виконується згідно документу «Програма та методика тестування». Як вже зазначалося раніше, код було покрито набором з юніт та інтеграційних тестів. Наведемо декілька прикладів юніт та інтеграційних тестів.

Розглянемо приклад юніт тесту, який перевіряє правильність сканування пакетів. В якості вхідних даних сканеру передається конфігураційний клас, з метаінформацією про розташування пакетів. Очікуваним результатом є знайдені п'ять класів, що розміщені за вказаним шляхом. Реалізацію цього тесту зображено на рисунку 4.7.

```

16  class ClasspathElementScannerTest {
17
18      @Test
19      @DisplayName("Test scanning of classpath elements with multiple packages")
20      void shouldScanPackagesCorrectly() {
21          var scannedClasses : Collection<Class<...>> = ClasspathElementScanner.scanPackages(InitClass.class);
22
23          assertThat(scannedClasses).isNotNull()
24              .hasSize( expected: 5)
25              .containsExactlyInAnyOrder(ClassA.class,
26                  ClassB.class,
27                  ClassC.class,
28                  ClassD.class,
29                  ClassE.class
30              );
31      }

```

Рисунок 4.7 – Приклад юніт тесту, що перевіряє роботу сканеру пакетів

Тепер розглянемо інтеграційний тест. В цьому тесті перевіряємо коректність роботи механізму області елементів. Хід виконання тесту наступний. Спочатку ініціюємо контекст, надаючи йому конфігураційний клас з посиланням на модулі, які необхідно просканувати. Після ініціалізації дістаємо об'єкт з областю *PROTOTYPE* двічі. Далі перевіряємо, що об'єкт дійсно нам повернувся та що посилання на ці об'єкти різні, чого і вимагає така область об'єкту. Реалізація цього тесту наведена на рисунку 4.8.

```

38     @Test
39     void shouldReturnNewObjectForPrototype() {
40         var context : ContextHolder = ContextHolder.initializeContext(ContextWithConfigClass.class);
41
42         context.getElementDefinitionHolder().getElementDefinitions();
43         var prototypeElementFirstCall : Optional<Object> = context.getElement( name: "singleDependencyExternalElement");
44         var prototypeElementSecondCall : Optional<Object> = context.getElement( name: "singleDependencyExternalElement");
45
46         assertThat(prototypeElementFirstCall)
47             .isNotEmpty();
48         assertThat(prototypeElementSecondCall) OptionalAssert<Object>
49             .isNotEmpty()
50             .get() AbstractObjectAssert<capture of ?, Object>
51             .isNotSameAs(prototypeElementFirstCall);
52     }

```

Рисунок 4.8 – Приклад інтеграційного тесту

Проте, інтеграційний тест вимагає більше зусиль по налаштуванню конфігурації та підготовки тестових даних. Тому підтримка таких тестів є більш складним та трудомістким процесом. На рисунках 4.9 та 4.10 наведено додаткові налаштування, які необхідно було виконати для створення інтеграційного тесту.

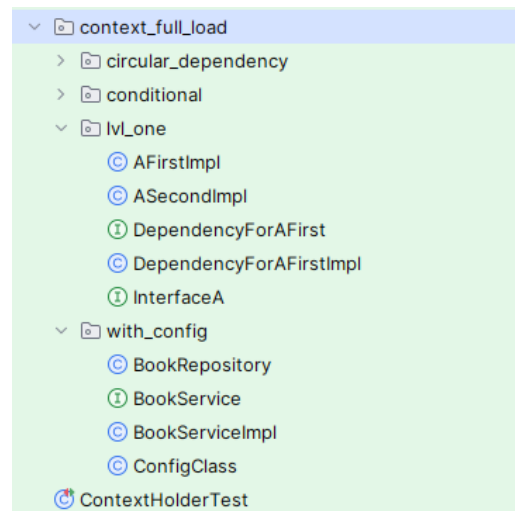


Рисунок 4.9 – Модулі та класи, підготовлені для інтеграційного тесту

```

171
172     @ElementScan(
173         basePackages = "ua.gorobeos.contextor.context.storage.context_full_load.with_config"
174     )
175     private static class ContextWithConfigClass {
176
177     }

```

Рисунок 4.10 – Конфігураційний клас, який слугує точкою входу та зберігає метадані про пакети, які необхідно просканувати

Додатково було проведене мануальне тестування фреймворку Contextor. Як вже зазначалося вище, вдалося покрити код автоматизованими тестами більше, ніж на 80%, тому мануальне тестування проводилося в обмеженому обсягу. Основні тест кейси наведено в таблицях 4.1 – 4.5.

Таблиця 4.1 – Тест 1.1. Сканування та створення об'єктів із рекурсивними залежностями

Початковий стан системи	Немає жодних попередньо створених об'єктів у контексті. Створено декілька класів зі складною ієрархією залежностей ($A \rightarrow B \rightarrow C$)
Вхідні дані	Класи з анотаціями @Element, @Inject. Конфігурація для сканування пакету з цими класами.
Опис проведення тесту	Ініціалізувати Contextor, вказати пакет для сканування. Запросити об'єкт класу А через API ContextHolder. Після цього отримати усі об'єкти з контексту.
Очікуваний результат	Всі об'єкти створені успішно з урахуванням рекурсивних залежностей. У логах видно кроки створення кожного об'єкта. Немає циклічних залежностей.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.2 – Тест 1.2. Виявлення відсутньої залежності

Початковий стан системи	Існує клас А, який має залежність на клас В, який в свою чергу не помічений для додавання в контекст.
Вхідні дані	Сканування пакету з класом А.
Опис проведення тесту	Ініціалізувати Contextor, спробувати отримати об'єкт А.

Продовження таблиці 4.2

Очікуваний результат	Отримуємо помилку <code>ElementInitializationException</code> . У логах виведено ім'я класу, причину виникнення помилки та інформацію про залежність, яку не знайдено. Повідомлення зрозуміле для розробника.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.3 – Тест 1.3. Масове збереження понад 50 об'єктів у контексті

Початковий стан системи	Згенеровано 100 класів з анотацією <code>@Element</code> у різних пакетах.
Вхідні дані	Запуск <code>Contextor</code> на основі конфігурації, яка охоплює всі пакети з розташованими в них класами
Опис проведення тесту	Ініціалізувати <code>Contextor</code> , просканувати всі пакети. Спробувати отримати кожен об'єкт через API <code>ContextHolder</code> .
Очікуваний результат	Усі 100 об'єктів створені й доступні через метод <code>getElement()</code> без помилок. Продуктивність знижується не суттєво. Контекст зберігає всі об'єкти, API працює без обмежень.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.4 – Тест 1.4. Обробка винятку при помилці оцінки умови

Початковий стан системи	Є клас з анотацією <code>@ConditionalOnWebRequest</code> із запитом, який завжди видає помилку.
Вхідні дані	Запуск <code>Contextor</code> на основі конфігурації, яка охоплює всі пакети з розташованими в них класами.

Продовження таблиці 4.4

Опис проведення тесту	Ініціалізувати Contextor і просканувати клас. Перевірити логи і результат.
Очікуваний результат	Об'єкт не створюється. У логах виведена зрозуміла інформація про виняток. Робота системи не завершується, а решта компонентів ініціалізуються нормально.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.5 – Тест 1.5. Некоректна прив'язка до реалізації

Початковий стан системи	Є два класи, які реалізують один інтерфейс, обидва позначені як @Element.
Вхідні дані	Клас-клієнт має залежність на інтерфейс, але без уточнення реалізації.
Опис проведення тесту	Ініціалізувати контейнер. Спробувати створити об'єкт клієнта без уточнення реалізації.
Очікуваний результат	Виникає помилка про неможливість вирішення, яку саме реалізацію застосувати в якості залежності. Розробнику надаються інформативні логи з поясненням, що пішло не так.
Фактичний результат	Збігається з очікуванням.

4.3 Опис контрольного прикладу

В якості контрольного прикладу використаємо один з основних бізнес-процесів, який підтримує IoC контейнер Contextor, а саме умовне створення об'єктів.

Поставимо задачу наступним чином: «Мені, як розробнику, необхідно створити наступні класи»:

1. BookRepository – контракт, який повинні реалізовувати усі репозиторії книжок. Лістинг коду наведено на рисунку 4.11.

```

6 ④ public interface BookRepository {
7
8 ④   List<String> getBooks();
9   }

```

Рисунок 4.11 – Лістинг коду BookRepository

2. AmazonS3BookRepository – конкретна реалізація репозиторію книжок. Здійснює пошук доступних книжок, використовуючи хмарне сховище Amazon S3. Проте цей клас повинен створюватися лише за умови, що в файлах проєкту присутній конфігураційний файл "amazon.config", а також у властивостях проєкту задано параметр "amazon.enabled=true". Лістинг коду наведено на рисунку 4.12.

```

9   @Element
10  @ConditionalOnFilePresence(filePaths = {"amazon.config"})
11  @ConditionalOnProperty(
12      name = "amazon.s3.enabled",
13      value = "true"
14  )
15  public class AmazonS3BookRepository implements BookRepository {
16
17      @Override
18  ④ public List<String> getBooks() {
19          return List.of(
20              "Amazon Book 1",
21              "Amazon Book 2",
22              "Amazon Book 3"
23          );
24      }
25  }

```

Рисунок 4.12 – Лістинг коду AmazonS3BookRepository

Опишемо конфігураційні анотації, що застосовані в цьому класі.

`@Element` – маркер, який використовується для вказування IoC контейнеру, що необхідно створити екземпляр цього класу.

`@ConditionalOnFilePresence` – вказує, що необхідною умовою для створення об'єкту є наявність у проєкті певного файлу. В цьому випадку файлу *"amazon.properties"*.

`@ConditionalOnProperty` – вказує, що необхідною умовою для створення об'єкту є наявність у відповідних властивостях проєкту параметру із вказаним значенням. В цьому випадку параметру *"amazon.s3.enabled"* зі значенням *"true"*.

3. `PostgresBookRepository` – реалізація інтерфейсу `BookRepository`, яка використовує базу даних PostgreSQL для отримання інформації про книжки. Повинна створюватися за замовчуванням, спеціальні умови створення не передбачені. Лістинг коду наведено на рисунку 4.13.

```

8      @Element
9      @Primary
10     public class PostgresBookRepository implements BookRepository {
11
12         @Override
13         public List<String> getBooks() {
14             return List.of(
15                 "Postgres Book 1",
16                 "Postgres Book 2",
17                 "Postgres Book 3"
18             );
19         }
20     }

```

Рисунок 4.13 – Лістинг коду `PostgresBookRepository`

Розглянемо конфігураційні анотації, які ще не були описані.

`@Primary` – вказує, що даний клас потрібно пріоритезувати при вирішенні конфліктів залежностей. Тобто, якщо існує дві реалізації інтерфейсу, який необхідний для створення класу, то в залежність буде встановлено саме клас, який має помітку `@Primary`.

4. BookService – інтерфейс, що описує методи, які повинен містити кожен книжковий сервіс. А саме метод для отримання книжок getBooks(). Лістинг коду наведено на рисунку 4.14.

```

5  public interface BookService {
6
7      List<String> getBooks();
8
9  }

```

Рисунок 4.14 – Лістинг коду BookService

5. BookServiceImpl – імплементація книжкового сервісу, яка використовує BookRepository для отримання інформації про книжки. В ході контрольного прикладу спробуємо змінювати конкретну реалізацію, яку потребує цей сервіс. Початкову конфігурацію зображено на рисунку 4.15.

```

10  @Element("superBookService")
11  public class BookServiceImpl implements BookService {
12
13      private final BookRepository bookRepository;
14
15      @Injected
16      public BookServiceImpl(@Qualifier("amazonS3BookRepository") BookRepository bookRepository) {
17          this.bookRepository = bookRepository;
18      }
19
20      @Override
21      public List<String> getBooks() { return bookRepository.getBooks(); }
24  }

```

Рисунок 4.15 – Лістинг коду BookServiceImpl

Зупинимось на конфігурації, яка була задана за допомогою анотацій в класі BookServiceImpl. Розберемо детально кожен з них:

@Element – крім зазначеного вище використання, також може використовуватися для вказування імені елемента, який буде збережено у контексті.

`@Injected` – вказує, який саме конструктор використовувати для встановлення залежностей.

`@Qualifier` – використовується для зазначення, яку саме імплементацію необхідно встановити в даний клас.

Діаграму класів, які були описані вище, зображено на рисунку 4.16.

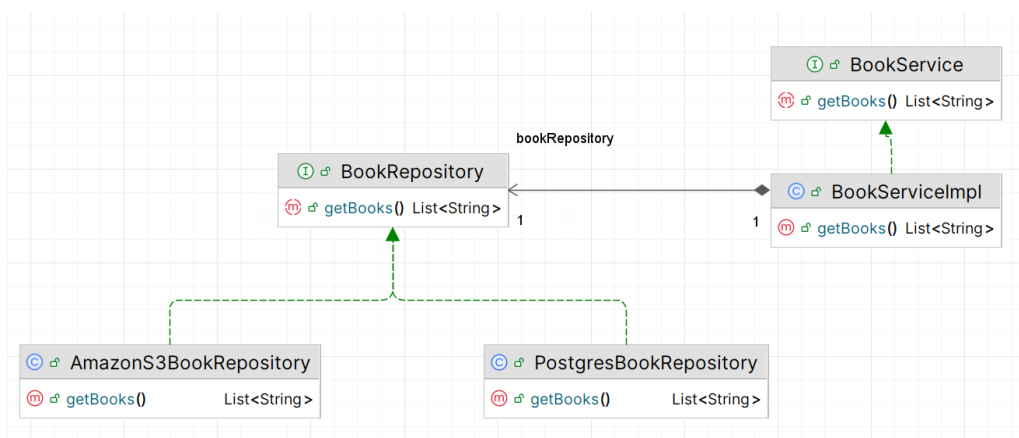


Рисунок 4.16 – Діаграма класів контрольного прикладу

Далі розглянемо файли та параметри, які присутні у проєкті. Ці дані наведені на рисунках 4.17 та 4.18 відповідно.

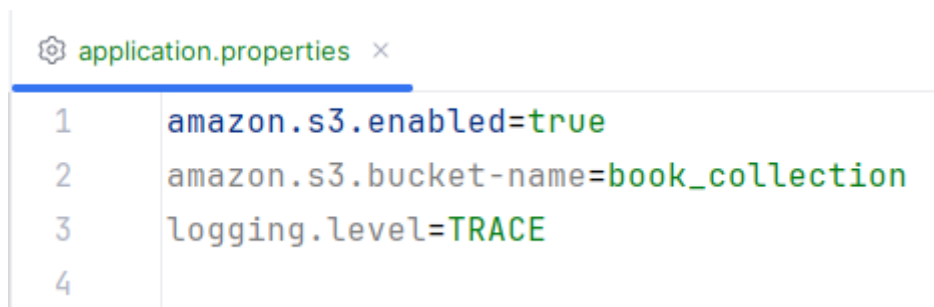


Рисунок 4.17 – Параметри застосунку

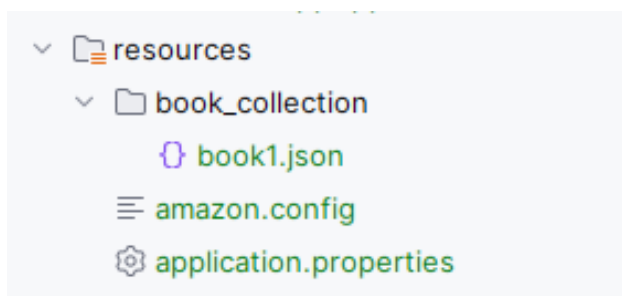


Рисунок 4.18 – Файли, які присутні в проєкті

Ініціалізуємо ІоС контейнер та спробуємо переглянути, які класи були створені. Для цього створимо клас з метайнформацією про пакети, які необхідно просканувати, та запустимо контейнер (рисунок 4.19). На рисунку 4.20 можемо бачити приклад детальних логів, які надаються користувачу для забезпечення максимальної прозорості процесів, що відбуваються всередині фреймворку.

```

9      @ElementScan(basePackages = "ua.gorobeos.demoapp")
10  public class DemoAppApplication {
11
12      private static final Logger log = LoggerFactory.getLogger(DemoAppApplication.class);
13
14      public static void main(String[] args) {
15          ContextHolder contextHolder = ContextHolder.initializeContext(DemoAppApplication.class);
16          contextHolder.getNameToElementMap();
17          log.info("Context initialized successfully!");
18      }
19
20  }

```

Рисунок 4.19 – Ініціалізація контейнера ІоС

```

23:52:48.537 [main] INFO u.g.c.context.storage.ContextHolder -> Initializing context with class: ua.gorobeos.demoapp.DemoAppApplication
23:52:48.542 [main] INFO u.g.c.c.s.classpath.ElementScanner -> Initializing context... Init class: DemoAppApplication
23:52:48.542 [main] DEBUG u.g.c.c.s.classpath.ElementScanner -> Resolving packages for class: ua.gorobeos.demoapp.DemoAppApplication
23:52:48.544 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting value from annotation: ua.gorobeos.contextor.context.annotations.ElementScan for field: basePackages in class: ua.gorobeos.demoapp.DemoAppApplication
23:52:48.549 [main] DEBUG u.g.c.c.s.classpath.ElementScanner -> Resolved packages: [ua.gorobeos.demoapp]
23:52:48.551 [main] INFO u.g.c.context.utils.ScannerUtils -> Scanning classes in package: ua.gorobeos.demoapp
23:52:48.634 [main] INFO org.reflections.Reflections -> Reflections took 53 ms to scan 1 urls, producing 3 keys and 9 values
23:52:48.644 [main] TRACE u.g.c.context.utils.ScannerUtils -> Found 6 classes in package: ua.gorobeos.demoapp
23:52:48.645 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.DemoAppApplication
23:52:48.645 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.book.repository.impl.PostgresBookRepository
23:52:48.648 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.book.repository.impl.BookRepository
23:52:48.649 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.book.service.impl.BookService
23:52:48.649 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl
23:52:48.670 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.book.repository.impl.AmazonS3BookRepository
23:52:48.677 [main] DEBUG u.g.c.c.s.classpath.ElementScanner -> Classes found in packages: [class ua.gorobeos.demoapp.book.service.impl.BookServiceImpl, class ua.gorobeos.demoapp.book.repository.impl.AmazonS3BookRepository, class ua.gorobeos.demoapp.book.repository.impl.PostgresBookRepository]
23:52:48.678 [main] DEBUG u.g.c.context.storage.ContextHolder -> Classes found during scanning: [class ua.gorobeos.demoapp.book.service.impl.BookServiceImpl, class ua.gorobeos.demoapp.book.repository.impl.AmazonS3BookRepository, class ua.gorobeos.demoapp.book.repository.impl.PostgresBookRepository]
23:52:48.678 [main] INFO u.g.c.context.storage.ContextHolder -> Registering class definitions in context holder
23:52:48.687 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl
23:52:48.688 [main] DEBUG u.g.c.c.r.ElementDefinitionReaderFacadeImpl -> Using AnnotatedBaseElementDefinitionReader reader for class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl
23:52:48.688 [main] DEBUG u.g.c.c.r.AnnotatedBaseElementDefinitionReader -> Reading element definition for class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl
23:52:48.688 [main] DEBUG u.g.c.c.r.AnnotatedBaseElementDefinitionReader -> Resolving element name for class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl
23:52:48.688 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting value from annotation: ua.gorobeos.contextor.context.annotations.Element for field: value in class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl
23:52:48.688 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl for annotation: ua.gorobeos.contextor.context.annotations.Element
23:52:48.694 [main] DEBUG u.g.c.c.r.BaseElementDefinitionReader -> Resolved element name: superBookService
23:52:48.695 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting value from annotation: ua.gorobeos.contextor.context.annotations.Scope for field: value in class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl
23:52:48.695 [main] TRACE u.g.c.context.utils.ReflectionUtils -> Getting annotations from class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl for annotation: ua.gorobeos.contextor.context.annotations.Scope
23:52:48.695 [main] DEBUG u.g.c.c.r.AnnotatedBaseElementDefinitionReader -> No specific scope provided for class: class ua.gorobeos.demoapp.book.service.impl.BookServiceImpl. Using default: singleton
23:52:48.695 [main] DEBUG u.g.c.c.r.BaseElementDefinitionReader -> Resolved element scope: singleton
23:52:48.696 [main] TRACE u.g.c.c.r.BaseElementDefinitionReader -> Single constructor [public ua.gorobeos.demoapp.book.service.impl.BookServiceImpl(ua.gorobeos.demoapp.book.repository.impl.BookRepository)] found for class: ua.gorobeos.demoapp.book.service.impl.BookServiceImpl

```

Рисунок 4.20 – Логи в процесі ініціалізації контейнера ІоС

Після запуску контейнера для зручності скористаємося дебагером та виконаємо перевірку, які саме об'єкти було створено та додано в ІоС контейнер. Результат наведено на рисунку 4.21.

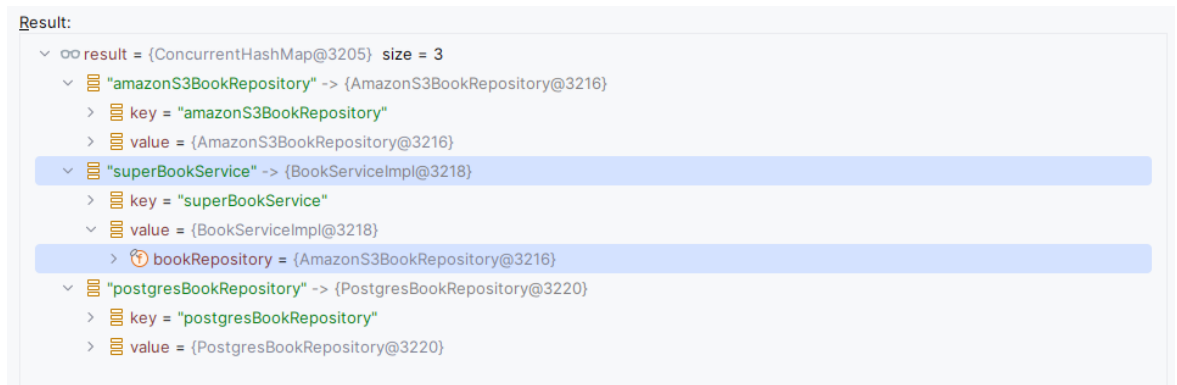


Рисунок 4.21 – Класи, додані в контекст ІоС контейнера

Як бачимо, отримано очікуваний результат, усі необхідні класи було створено. Вказана залежність встановлена коректно.

Висновки до розділу

У рамках цього розділу було виконано комплексний аналіз якості програмного забезпечення фреймворкуContextor. Аналіз охоплював оцінку коду за ключовими метриками, проведення автоматизованого й мануального тестування, а також включав перевірку функціональності на контрольному прикладі.

Для оцінювання якості ПЗ було обрано інструмент для статичного аналізу коду SonarCloud. З його допомогою було здійснено повну перевірку коду за наступними метриками: безпека, надійність, легкість підтримки, цикломатична складність та рівень тестового покриття. За результатами аналізу визначено, що код оцінено на оцінку «А», тобто, відмінно. Зокрема зазначена відсутність проблем та можливих уразливих місць у безпеці, отримано максимальний бал за надійністю коду, також була надана висока оцінка підтримуваності коду. Цикломатична складність є прийнятною з урахуванням складності фреймворку та задач, які він вирішує.

Для тестування правильності бізнес-логіки використовувалися різноманітні підходи. Кодова база була покрита юніт та інтеграційними тестами. Додатково було виконано мануальну перевірку специфічних сценаріїв. Під час автоматизованого тестування було створено більше 60-ти

тест-кейсів, що охоплюють як окремі компоненти, так і взаємодію між ними. Інтеграційні тести перевірили коректність роботи механізмів ініціалізації, створення об'єктів з різними типами областей, механізми обробки помилок та умовного створення об'єктів.

Мануальне тестування дозволило підтвердити стійкість системи в складних сценаріях, наприклад, при роботі з рекурсивними залежностями, помилках конфігурації та конфліктах при вирішенні залежностей. Усі результати збіглися з очікуваними.

Контрольний приклад продемонстрував можливість умовного створення об'єктів на основі зовнішніх конфігурацій, тобто одну з ключових особливостей фреймворку Contextor. Це підтвердило коректність роботи створеного ПЗ.

Отже, проведений аналіз і тестування дозволили не лише підтвердити відповідність розробленого фреймворку сучасним вимогам до якості ПЗ, а й продемонстрували ефективність реалізованих рішень у типових і нестандартних сценаріях використання.

5 РОЗГОРТАННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Розгортання програмного забезпечення

Легкість інтеграції фреймворку до проекту грає одну з провідних ролей у зручності використання Contextor. Для спрощення такої інтеграції було вирішено завантажити проєкт до глобального репозиторію Sonatype [35]. Це дозволило надати усім розробникам швидкий доступ до підключення Contextor в будь-який власний проєкт.

Розпишемо кроки, які необхідно було виконати, задля розгортання артефакту Contextor у глобальний репозиторій.

По-перше, було сконфігуровано відповідні плагіни – для генерації коду, документації та ресурсів, що необхідні для фреймворку. Частина конфігурації зображена на рисунку 5.1.

```

148     <plugin>
149       <groupId>org.apache.maven.plugins</groupId>
150       <artifactId>maven-source-plugin</artifactId>
151       <version>2.2.1</version>
152       <executions>
153         <execution>
154           <id>attach-sources</id>
155           <goals>
156             <goal>jar-no-fork</goal>
157           </goals>
158         </execution>
159       </executions>
160     </plugin>
161     <plugin>
162       <groupId>org.apache.maven.plugins</groupId>
163       <artifactId>maven-javadoc-plugin</artifactId>
164       <version>3.6.2</version>
165       <executions>
166         <execution>
167           <id>attach-javadocs</id>
168           <goals>
169             <goal>jar</goal>
170           </goals>
171         </execution>
172       </executions>
173       <configuration>
174         <tags>
175           <tag>
176             <name>apiNote</name>
177             <placement>a</placement>
178             <head>API Note:</head>
179           </tag>
180         </tags>
181       </configuration>
182     </plugin>

```

Рисунок 5.1 – Частина конфігураційного файлу pom.xml

Також, згідно з вимогами безпеки необхідно було провести шифрування файлів, для запобігання підміни оригінальних файлів. Шифрування проводилося за допомогою інструмента GPG [36]. В результаті кодування було отримано наступні файли, зображені на рисунку 5.2.

Усі файли, що мають розширення ".asc" є файлами, зашифрованими за допомогою приватного ключа.

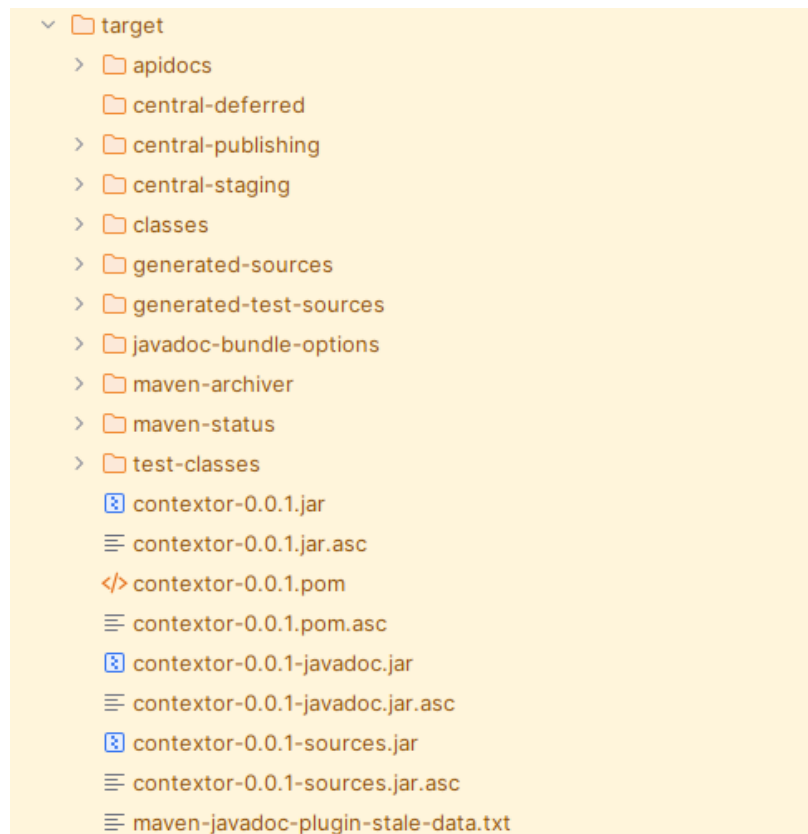


Рисунок 5.2 – Файли, отримані після збірки артефакту Contextor

Далі було створено простір імен на репозиторії Sonatype. Для цього необхідно було створити організацію та пройти верифікацію. Створений простір імен зображено на рисунку 5.3.

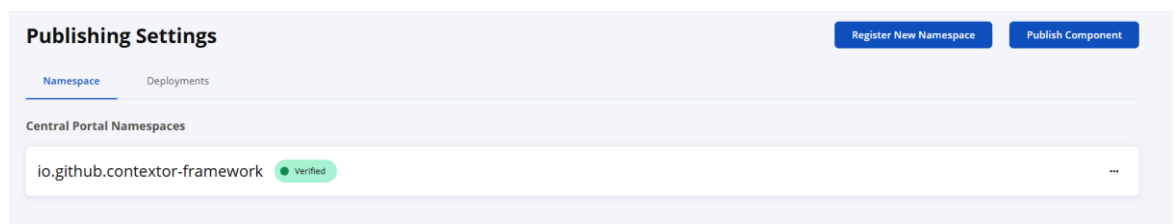


Рисунок 5.3 – Простір імен, зарезервований під фреймворк Contextor

Далі, було зібрано та опубліковано артефакт Contextor у зазначений вище простір імен. Результат публікації наведено на рисунках 5.4 та 5.5.

```
[INFO] Created bundle successfully C:\Study\University\Diploma\contextor-app\contextor-new\target\central-staging\central-bundle.zip
[INFO] Going to upload C:\Study\University\Diploma\contextor-app\contextor-new\target\central-publishing\central-bundle.zip
[INFO] Uploaded bundle successfully, deployment name: Deployment, deploymentId: 40a439ff-c9fb-4e57-bec6-d547c5272f27. Deployment will require manual publishing
[INFO] Waiting until Deployment 40a439ff-c9fb-4e57-bec6-d547c5272f27 is validated
[INFO] Deployment 40a439ff-c9fb-4e57-bec6-d547c5272f27 has been validated. To finish publishing visit https://central.sonatype.com/publishing/deployments
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 13.303 s
[INFO] Finished at: 2025-06-03T21:10:08+03:00
[INFO] -----
```

Рисунок 5.4 – Успішне завантаження Contextor до Sonatype репозиторію

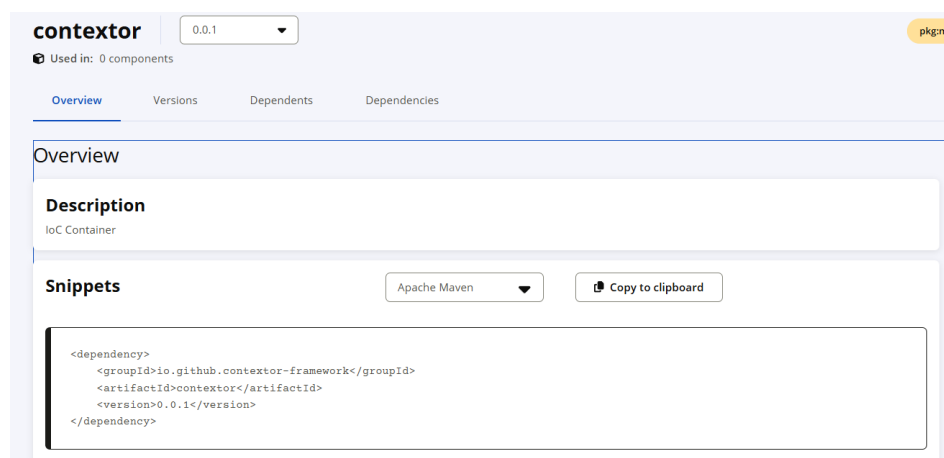


Рисунок 5.5 – Сторінка опублікованого фреймворку Contextor

Після публікації фреймворк Contextor став доступним до завантаження з інших пристроїв. Процес підключення фреймворку до проєкту є стандартним та не потребує ніяких спеціальних доналаштувань. Достатньо вказати необхідні параметри залежності у збірнику проєкту та виконати команду завантаження залежностей. Приклад підключення Contextor до проєкту, який базується на Maven, зображено на рисунках 5.6 та 5.7.

```
26 <dependencies>
27   <dependency>
28     <groupId>io.github.contextor-framework</groupId>
29     <artifactId>contextor</artifactId>
30     <version>0.0.1</version>
31   </dependency>
```

Рисунок 5.6 – Підключення залежності Contextor до проєкту

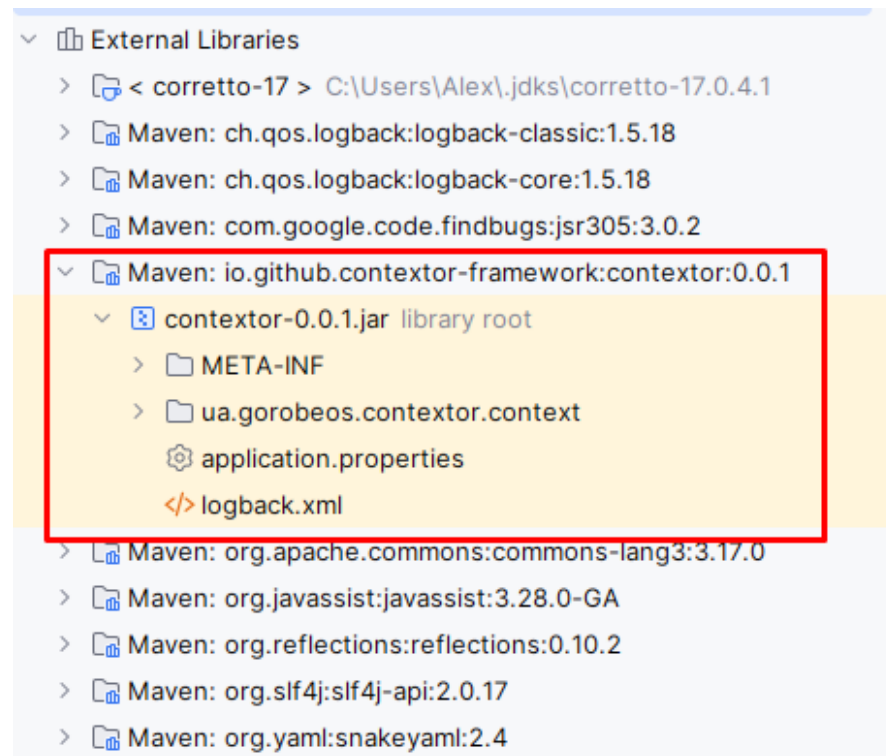


Рисунок 5.7 – Фреймворк Contextor серед залежностей проєкту

Отже, підключення виконано успішно та легко.

5.2 Супровід програмного забезпечення

Інструкція користувача програміста наведена в окремому документі «Керівництво користувача».

Оскільки планується продовжувати підтримувати та розвивати ІоС контейнер Contextor, то необхідно надати користувачам зручний спосіб для завантаження нових версій застосунку. Для цього застосовується система версіонування. Кожний реліз фреймворку Contextor його версія зростає. Користувач, який зацікавлений в оновленні застосунку, повинен внести зміни у версію в збірнику проєкту, який він використовує, та перезавантажити його.

Наведемо приклад оновлення Contextor у проєкті, що використовує збірник Apache Maven. На рисунку 5.8 виділено поле з версією, яку необхідно оновити на більш нову.

```

26     <dependencies>
27     <dependency>
28         <groupId>io.github.contextor-framework</groupId>
29         <artifactId>contextor</artifactId>
30         <version>0.0.2</version>
31     </dependency>
32 </dependencies>

```

Рисунок 5.8 – Вказана нова версія фреймворку Contextor

Після оновлення необхідно виконати команду перезбирання залежностей проєкту. Для Maven це *"mvn dependency:resolve"*. Результат виконання команди зображено на рисунку 5.9.

```

[INFO] -----< ua.gorobeos:demo-app >-----
[INFO] Building demo-app 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- dependency:3.7.0:resolve (default-cli) @ demo-app ---
[INFO]
[INFO] The following files have been resolved:
[INFO]   io.github.contextor-framework:contextor:jar:0.0.2:compile -- module contextor (auto)
[INFO]   org.yaml:snakeyaml:jar:2.4:compile -- module org.yaml.snakeyaml
[INFO]   ch.qos.logback:logback-classic:jar:1.5.18:compile -- module ch.qos.logback.classic
[INFO]   ch.qos.logback:logback-core:jar:1.5.18:compile -- module ch.qos.logback.core
[INFO]   org.slf4j:slf4j-api:jar:2.0.17:compile -- module org.slf4j
[INFO]   org.reflections:reflections:jar:0.10.2:compile -- module org.reflections [auto]
[INFO]   org.javassist:javassist:jar:3.28.0-GA:compile -- module javassist (auto)
[INFO]   com.google.code.findbugs:jsr305:jar:3.0.2:compile -- module jsr305 (auto)
[INFO]   org.apache.commons:commons-lang3:jar:3.17.0:compile -- module org.apache.commons.lang3
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  4.855 s
[INFO] Finished at: 2025-06-03T22:36:47+03:00
[INFO] -----

```

Рисунок 5.9 – Результат оновлення версії фреймворку Contextor

Як бачимо, проєкт успішно оновлено. Процес оновлення Contextor зручний та інтуїтивно зрозумілий для будь-якого розробника, що вміє користуватися збірником проєктів.

Висновки до розділу

У п'ятому розділі дипломної роботи було детально розглянуто процес розгортання та супроводу програмного забезпечення – ІоС контейнера Contextor. Завдяки публікації артефакту у глобальній репозиторії Sonatype, фреймворк став легко доступним для підключення до будь-якого зовнішнього проєкту. Це значно спрощує інтеграцію та сприяє зручному поширенню фреймворку серед розробників.

Додатково було описано кроки, необхідні для забезпечення безпеки розгортання, а саме шифрування файлів проєкту та проходження верифікації для створення простору імен у репозиторії Sonatype. Крім того, розглянуто механізм оновлення фреймворку в існуючих проєктах, який реалізовано за допомогою стандартної системи версіонування. Це дозволяє користувачам швидко отримувати актуальні оновлення без зайвих труднощів.

Таким чином, фреймворк Contextor не лише зручно інтегрується у проєкти, а й забезпечує прозорий та зрозумілий механізм його супроводу, що є важливим фактором у контексті підтримки та розвитку сучасного програмного забезпечення.

ВИСНОВКИ

В ході виконання дипломного проєктування було розроблено IoC фреймворк Contextor, який підтримує всі базові функціональні вимоги, що висуваються до сучасних IoC контейнерів, та додатково надає розширений функціонал для керування умовами створення об'єктів. Розробка призначена для автоматизації життєвого циклу об'єктів та полегшення механізмів контролю над умовами їх створення за допомогою IoC контейнера.

В процесі виконання дипломного проєкту проаналізовано сучасний стан у сфері IoC контейнерів та індустрії в цілому; проаналізовано відомі технічні та алгоритмічні рішення, порівняно два основних підходи до написання IoC контейнерів, а саме статичну генерацію коду та runtime-based підхід. Вказано недоліки та переваги кожного з них та аргументовано вибір підходу для реалізації власного IoC контейнера.

Виконано опис та аналіз відомих програмних рішень, які представлені на ринку. Проведено порівняльний аналіз існуючих фреймворків та продукту, що розробляється, на основі якого зроблено висновок про доцільність розробки сучасного IoC контейнеру з розширеним набором інструментів для управління умовами створення об'єктів. За допомогою BPMN діаграм проведено детальне моделювання основних бізнес-процесів, які повинен підтримувати IoC контейнер, а саме: процес ініціалізації та заповнення контексту контейнера та процес отримання об'єкта з IoC контейнера.

В ході виконання проєкту розроблено функціональні, нефункціональні та системні вимоги до ПЗ, що розробляється. Виділено та описано основні архітектурні рішення та підходи до конструювання IoC контейнера, зокрема: побудовано діаграми перших трьох рівнів моделі С4; розглянуто архітектурні рішення та проведено детальний огляд основних патернів, що використовувалися для побудови масштабованої та легко підтримуваної архітектури фреймворку Contextor.

Проведено аналіз сучасних мов програмування та обґрунтовано вибір мови програмування Java для створення IoC контейнера. Наведено та описано

утиліти, бібліотеки та інструменти, що використовувалися при розробці програми. Вказано призначення та спосіб використання кожного з інструментів, що були задіяні в розробці.

Проведено аналіз якості фреймворку Contextor, що включав статичний аналіз коду, автоматизоване й мануальне тестування, а також перевірку коректності роботи на контрольному прикладі. Для статичного аналізу коду використано популярну платформу SonarCloud, яка надала найвищу оцінку проєкту за критеріями: безпека, надійність, підтримуваність, цикломатична складність і тестове покриття. Це свідчить про відсутність уразливостей, високу стабільність і легкість підтримки. Цикломатична складність визнана прийнятною з огляду на призначення фреймворку.

Тестування включало в себе написання юніт та інтеграційних тестів, що перевіряли ключові процеси – ініціалізацію контексту, створення об'єктів з різними конфігураціями, обробку помилок і умовне створення компонентів. Створено понад 60 тест-кейсів. Додатково виконано ручне тестування складних сценаріїв, зокрема з рекурсивними залежностями та помилками конфігурації. Результати відповідали очікуванам.

Контрольний приклад підтвердив правильність реалізації однієї з головних функцій Contextor – створення об'єктів на основі заданих умов.

Розглянуто процес розгортання та супроводу IoC контейнера Contextor. Фреймворк опубліковано в репозиторії Sonatype, що значно спростило інтеграцію з іншими проєктами. Описано заходи безпеки, що були вжиті для запобігання підміни оригінального артефакту. Виконано шифрування файлів, створення та верифікацію окремого простору імен. Надано інформацію про механізм оновлень, який відповідає сучасним стандартам та легко інтегрується у збірники проєктів.

Проєкт Contextor планується активно розвивати й надалі, адаптуючи його до потреб сучасної розробки. З огляду на постійні зміни та вдосконалення в ІТ-сфері, зростає актуальність гнучких і ефективних архітектурних рішень, тож попит на подібні інструменти лише посилюватиметься. Основу для

спрощеного розвитку та підтримки Contextor було закладено ще на етапі проєктування – через продуману архітектуру, систему версіонування та зручний механізм розповсюдження. Завдяки цим попереднім рішенням забезпечено прозорий і зручний процес супроводу та подальшого вдосконалення фреймворку.

Отже, завдання дипломного проєктування виконані, мети повністю досягнуто.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Goodness W. The difference between libraries and frameworks. URL: <https://www.red-gate.com/simple-talk/development/other-development/the-difference-between-libraries-and-frameworks/> (date of access: 25.05.2025).
2. Schmidt D.C., Gokhale A., Natarajan B. Frameworks: Why They Are Important and How to Apply Them Effectively. URL: <https://www.dre.vanderbilt.edu/~schmidt/PDF/Queue-04.pdf> (date of access: 25.05.2025).
3. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. URL: <https://www.amazon.com/Design-Patterns-Object-Oriented-Addison-Wesley-Professional-ebook/dp/B000SEIBB8> (date of access: 25.05.2025).
4. Johnson R. E., Foote B. Designing Reusable Classes. URL: <https://www.cse.msu.edu/~cse870/Input/SS2002/MiniProject/Sources/DRC.pdf> (date of access: 25.05.2025).
5. Meszaros G. xUnit Test Patterns: Refactoring Test Code. URL: <https://www.amazon.com/xUnit-Test-Patterns-Refactoring-Code/dp/0131495054> (date of access: 25.05.2025).
6. Beck K. Test Driven Development: By Example. URL: <https://www.amazon.com/Test-Driven-Development-Kent-Beck/dp/0321146530> (date of access: 25.05.2025).
7. Dagger Repository. URL: <https://github.com/google/dagger>. (date of access: 25.05.2025).
8. Guice Repository. URL: <https://github.com/google/guice> (date of access: 25.05.2025).
9. Loose Coupling Definition. URL: <https://www.techtarget.com/searchnetworking/definition/loose-coupling> (date of access: 25.05.2025).
10. Smart IoC Repository. URL: https://github.com/droidlabs/smart_ioc (date of access: 25.05.2025).

11. Spring Framework Documentation. URL: <https://docs.spring.io/spring-framework/reference/index.html> (date of access: 25.05.2025).
12. InversifyJS Repository. URL: <https://github.com/inversify/InversifyJS> (date of access: 25.05.2025).
13. BPMN 2.0 Specification. URL: <https://www.omg.org/spec/BPMN/2.0/> (date of access: 25.05.2025).
14. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0 <https://staff.emu.edu.tr/alexanderchefranov/Documents/CMPE412/Boehm1995%20COCOMO%20%20.pdf> (date of access: 25.05.2025).
15. Proxy Design Pattern. URL: <https://refactoring.guru/design-patterns/proxy> (date of access: 25.05.2025).
16. C++ Documentation. URL: <https://devdocs.io/cpp/> (date of access: 25.05.2025).
17. Python 3 Documentation. URL: <https://docs.python.org/3/> (date of access: 25.05.2025).
18. Rust Documentation. URL: <https://doc.rust-lang.org/stable/> (date of access: 25.05.2025).
19. TypeScript Documentation. URL: <https://www.typescriptlang.org/docs/> (date of access: 25.05.2025).
20. NestJS Documentation. URL: <https://docs.nestjs.com/> (date of access: 25.05.2025).
21. Java SE 17 API Documentation. URL: <https://docs.oracle.com/en/java/javase/17/docs/api/index.html> (date of access: 25.05.2025).
22. Java Reflection Guide. URL: <https://dev.java/learn/reflection/intro/> (date of access: 25.05.2025).
23. JVM Garbage Collectors. URL: <https://www.baeldung.com/jvm-garbage-collectors> (date of access: 25.05.2025).
24. IntelliJ IDEA Official Site. URL: <https://www.jetbrains.com/idea/> (date of access: 25.05.2025).

25. Maven Guides. URL: <https://maven.apache.org/guides/index.html> (date of access: 25.05.2025).
26. Logback Project. URL: <https://logback.qos.ch/> (date of access: 25.05.2025).
27. Project Lombok. URL: <https://projectlombok.org/> (date of access: 25.05.2025).
28. Reflections Library. URL: <https://github.com/ronmamo/reflections> (date of access: 25.05.2025).
29. Apache Commons Lang. URL: <https://commons.apache.org/proper/commons-lang/> (date of access: 25.05.2025).
30. JUnit 5 Official Site. URL: <https://junit.org/junit5/> (date of access: 25.05.2025).
31. Mockito Official Site. URL: <https://site.mockito.org/> (date of access: 25.05.2025).
32. AssertJ Documentation. URL: <https://assertj.github.io/doc/> (date of access: 25.05.2025).
33. SonarQube Cloud Documentation. URL: <https://docs.sonarsource.com/sonarqube-cloud/> (date of access: 25.05.2025).
34. Java Code Coverage for Eclipse. URL: <https://www.eclemma.org/index.html> (date of access: 25.05.2025).
35. Maven central repository. Documentation. <https://central.sonatype.org/register/central-portal/> (date of access: 25.05.2025).
36. GnuPG. Documentation. URL: <https://www.gnupg.org/documentation/index.html> (date of access: 25.05.2025).

ДОДАТОК А ЗВІТ ПОДІБНОСТІ



Дата звіту 6/6/2025
Дата редагування 6/6/2025

Документ прийнятий

Звіт подібності

метадані

Назва організації
National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute
Заголовок
ІП-12_Горобець_ПЗ
Автор
Науковий керівник / Експерт
ІП-12_ГоробецьСарнацький В.В.
підрозділ
ФІОТ, К-а інформатики та програмної інженерії

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



10
Довжина фрази для коефіцієнта подібності 2

12940
Кількість слів

98925
Кількість символів

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ____ ” _____ 2025 р.

**ІОС КОНТЕЙНЕР ДЛЯ СТВОРЕННЯ ОБ’ЄКТІВ НА ОСНОВІ
СКЛАДНИХ УМОВ**

Текст програми

КПІ.ПІ-1208.045200.03.12

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Владислав САРНАЦЬКИЙ

Нормоконтроль:

_____ Максим ГОЛОВЧЕНКО

Виконавець:

_____ Олексій ГОРОБЕЦЬ

Посилання на репозиторій з повним текстом програмного коду

<https://github.com/OleksiiHorobets/contextor>

Файл ContextHolder.java

Реалізація функціональної задачі ініціалізації контексту контейнера IoC

```
import static
ua.gorobeos.contextor.context.element.ElementDefinition.PROTOTYPE_SCOPE;
import static
ua.gorobeos.contextor.context.element.ElementDefinition.SINGLETON_SCOPE;

import ua.gorobeos.contextor.context.conditions.ConditionEvaluationUtils;
import ua.gorobeos.contextor.context.dependencies.DependencyResolver;
import ua.gorobeos.contextor.context.dependencies.SimpleDependencyResolver;
import
ua.gorobeos.contextor.context.dependencies.checker.DependencyCircularChecker;
import
ua.gorobeos.contextor.context.dependencies.checker.DfsDependencyCircularCheck
er;
import ua.gorobeos.contextor.context.element.ElementDefinition;
import ua.gorobeos.contextor.context.element.MethodDefinedElementDefinition;
import ua.gorobeos.contextor.context.exceptions.ElementCreationException;
import ua.gorobeos.contextor.context.readers.ElementDefinitionReaderFacade;
import
ua.gorobeos.contextor.context.readers.ElementDefinitionReaderFacadeImpl;
import ua.gorobeos.contextor.context.scanner.ClasspathElementScanner;

public class ContextHolder {

    Map<String, Object> nameToElementMap = new ConcurrentHashMap<>();
    ElementDefinitionHolder elementDefinitionHolder;
    ElementDefinitionReaderFacade elementDefinitionReaderFacade;
    DependencyResolver dependencyResolver;

    public static ContextHolder initializeContext(Class<?> initClazz) {
        log.info("Initializing context with class: {}", initClazz.getName());
        var classesFound = ClasspathElementScanner.scanPackages(initClazz);
        log.debug("Classes found during scanning: {}", classesFound);
        log.info("Registering class definitions in context holder");

        ElementDefinitionHolder elementDefinitionHolder = new
DefaultElementDefinitionHolder();
        ElementDefinitionReaderFacade elementDefinitionReaderFacade = new
ElementDefinitionReaderFacadeImpl(elementDefinitionHolder);
        DependencyResolver dependencyResolver = new
SimpleDependencyResolver(elementDefinitionHolder);
        DependencyCircularChecker dependencyCircularChecker = new
DfsDependencyCircularChecker(dependencyResolver);

        classesFound.
            forEach(elementDefinitionReaderFacade::addElementDefinition);

        elementDefinitionHolder.getElementDefinitions()
            .forEach(dependencyCircularChecker::checkForCircularDependencies);

        log.info("Context initialized with {} element definitions",
elementDefinitionHolder.getElementDefinitions().size());

        ContextHolder contextHolder = new ContextHolder(elementDefinitionHolder,
elementDefinitionReaderFacade, dependencyResolver);
```

```

        elementDefinitionHolder.getElementDefinitions()
            .stream()
            .map(ElementDefinition::getName)
            .forEach(contextHolder::getElement);

        return contextHolder;
    }

    public Optional<Object> getElement(String name) {
        if (nameToElementMap.containsKey(name)) {
            return Optional.ofNullable(nameToElementMap.get(name));
        }
        var elementDefinitionOptional =
            elementDefinitionHolder.getElementDefinition(name);
        if (elementDefinitionOptional.isEmpty()) {
            log.error("Element definition for name '{}' not found", name);
            return Optional.empty();
        }
        var elementDefinition = elementDefinitionOptional.get();

        log.debug("Creating new instance for element definition: {}",
            elementDefinition);

        return Optional.ofNullable(createElementInstance(elementDefinition));
    }

    private Object createElementInstance(ElementDefinition elementDefinition) {
        var dependenciesDefinitions = elementDefinition.getDependencies()
            .stream()
            .map(dependencyResolver::retrieveDependency)
            .map(ElementDefinition::getName)
            .map(this::getElement)
            .flatMap(Optional::stream)
            .toArray();

        log.debug("Creating instance of element: {} with dependencies: {}",
            elementDefinition.getName(), dependenciesDefinitions);

        Object createdElement;
        if (elementDefinition instanceof MethodDefinedElementDefinition
            methodDefined) {
            log.debug("Creating element using method defined element definition:
            {}", methodDefined);
            var configElement =
                getElement(StringUtil.lowercaseFirstLetter(methodDefined.getConfigClass().getSimpleName())
                    .orElseThrow(() -> new NoSuchElementException(
                        "Config element '%s' not found for method defined element
                        '%s'".formatted(
                            StringUtil.lowercaseFirstLetter(methodDefined.getConfigClass().getSimpleName()),
                            methodDefined.getName()))));
            try {
                log.debug("-----Invoking init method '{}' on config element: {}",
                    methodDefined.getInitMethod().getName(), configElement);
                createdElement = methodDefined.getInitMethod().invoke(configElement,
                    dependenciesDefinitions);
            } catch (IllegalAccessException | InvocationTargetException e) {
                throw new ElementCreationException(
                    "Failed to invoke init method '%s' on config element '%s' with
                    dependencies: %s".formatted(

```

```

        methodDefined.getInitMethod().getName(), configElement,
        dependenciesDefinitions), e);
    }
    } else {
        checkConditionsIfPrototype(elementDefinition);
        createdElement = initElementByConstructor(elementDefinition,
        dependenciesDefinitions);
    }

    log.info("Element '{}' created successfully with dependencies: {}",
    elementDefinition.getName(), dependenciesDefinitions);
    if (SINGLETON_SCOPE.equals(elementDefinition.getScope())) {
        log.debug("Registering element '{}' in context holder",
        elementDefinition.getName());
        nameToElementMap.put(elementDefinition.getName(), createdElement);
        return createdElement;
    } else {
        log.debug("Element '{}' is prototype, not registering in context
        holder", elementDefinition.getName());
    }
    return createdElement;
}

private void checkConditionsIfPrototype(ElementDefinition
elementDefinition) {
    if (!PROTOTYPE_SCOPE.equals(elementDefinition.getScope())) {
        return;
    }

    var evaluationResult =
    ConditionEvaluationUtils.evaluate(elementDefinition.getType());
    if (!evaluationResult.isConditionalCheckPassed()) {
        log.warn("Conditional check failed for element: {}",
        elementDefinition.getName());
        log.warn("Failed evaluation list: {}",
        evaluationResult.getConditionalCheckResults());
        throw new ElementCreationException(
            "Conditional check failed for element '%s' with dependencies:
            %s".formatted(
                elementDefinition.getName(),
                elementDefinition.getDependencies()));
    }
}

private static Object initElementByConstructor(ElementDefinition
elementDefinition, Object[] dependenciesDefinitions) {
    var initConstructor = elementDefinition.getInitConstructor();
    initConstructor.setAccessible(true);
    try {
        return initConstructor.newInstance(dependenciesDefinitions);
    } catch (InstantiationException | IllegalAccessException |
    InvocationTargetException e) {
        throw new ElementCreationException(
            "Failed to create instance of element '%s' using constructor '%s'
            with dependencies: %s".formatted(
                elementDefinition.getName(), initConstructor,
                dependenciesDefinitions), e);
    }
}
}

```

Файл ConditionEvaluationUtils.java

Реалізація функціональної задачі перевірки умови створення об'єкту

```
package ua.gorobeos.contextor.context.conditions;

import
ua.gorobeos.contextor.context.conditions.evaluators.ConditionalOnFilePresence
Evaluator;
import
ua.gorobeos.contextor.context.conditions.evaluators.ConditionalOnLinuxEvaluat
or;
import
ua.gorobeos.contextor.context.conditions.evaluators.ConditionalOnPropertyEval
uator;
import
ua.gorobeos.contextor.context.conditions.evaluators.ConditionalOnWebRequestEv
aluator;
import
ua.gorobeos.contextor.context.conditions.evaluators.ConditionalOnWindowsEvalu
ator;
import ua.gorobeos.contextor.context.config.ConfigurationReader;

public class ConditionEvaluationUtils {

    private static final List<ConditionalEvaluator> conditionalEvaluator = new
ArrayList<>(List.of(
        new ConditionalOnFilePresenceEvaluator(),
        new ConditionalOnPropertyEvaluator(),
        new ConditionalOnLinuxEvaluator(),
        new ConditionalOnWindowsEvaluator(),
        new ConditionalOnWebRequestEvaluator()
    ));

    public static ConditionalContext evaluate(Class<?> elementType) {
        var context = initializeConditionalContext(elementType);

        for (ConditionalEvaluator evaluator : conditionalEvaluator) {
            context = evaluator.evaluate(context);
            if (!context.isConditionalCheckPassed() && context.isCircuitBreaker())
            {
                log.warn("Circuit breaker is on and evalutaion failed. Skipping rest
{} evaluations", elementType.getSimpleName());
                return context;
            }
        }

        return context;
    }

    private static ConditionalContext initializeConditionalContext(Class<?>
elementClass) {
        return ConditionalContext.builder()
            .elementClass(elementClass)

            .isCircuitBreaker(ConfigurationReader.getOrDefault("conditional.checks.circui
t.breaker", false, Boolean::parseBoolean))
            .conditionalCheckResults(new ArrayList<>())
            .conditionalCheckPassed(true)
            .build();
    }
}
```

Файл ConditionalContext.java

Реалізація функціональної задачі для зберігання інформації про умови створення об'єкту

```
public class ConditionalContext {
    private Class<?> elementClass;
    private boolean isCircuitBreaker;
    private boolean conditionalCheckPassed;
    List<String> conditionalCheckResults;
    Map<Class<? extends Annotation>, List<Annotation>> processedConditions;
}
```

Файл BaseElementDefinitionReader.java

Реалізація функціональної задачі для зчитування мета інформації про об'єкт

```
public abstract class BaseElementDefinitionReader implements
ElementDefinitionReader {

    protected static final Set<String> VALID_SCOPES = Set.of(SINGLETON_SCOPE,
PROTOTYPE_SCOPE);

    public ElementDefinition readElementDefinition(Class<?> clazz) {
        if (clazz == null) {
            log.error("Cannot read element definition for null class");
            throw new ElementDefinitionCreationException("Class cannot be null");
        }
        log.debug("Reading element definition for class: {}", clazz.getName());
        String elementName = resolveElementName(clazz);
        log.debug("Resolved element name: {}", elementName);

        String elementScope = resolveScope(clazz);
        log.debug("Resolved element scope: {}", elementScope);

        Constructor<?> initConstructor = resolveElementInitConstructor(clazz);
        log.debug("Resolved init constructor: {}", initConstructor);

        Collection<DependencyDefinition> dependencyDefinitions =
resolveDependencies(initConstructor);

        ElementDefinition elementDefinition =
getElementDefinitionContainerWithSpecifics(clazz);
        elementDefinition.setName(elementName);
        elementDefinition.setType(clazz);
        elementDefinition.setInitConstructor(initConstructor);
        elementDefinition.setDependencies(dependencyDefinitions);
        elementDefinition.setScope(elementScope);
        return elementDefinition;
    }
}
```

Файл AnnotatedBaseElementDefinitionReader.java

Реалізація функціональної задачі для зчитування мета інформації про об'єкт на основі анотацій

```
import ua.gorobeos.contextor.context.element.AnnotationElementDefinition;
import ua.gorobeos.contextor.context.element.ElementDefinition;
import ua.gorobeos.contextor.context.utils.ReflectionUtils;
```

```

public class AnnotatedBaseElementDefinitionReader extends
BaseElementDefinitionReader {

    @Override
    public String resolveScope(Class<?> clazz) {
        var scope = ReflectionUtils.getValueFromAnnotation(clazz, Scope.class,
"value", String.class)
            .orElseGet(() -> {
                log.debug("No specific scope provided for class: {}. Using default:
{}", clazz, ElementDefinition.SINGLETON_SCOPE);
                return ElementDefinition.SINGLETON_SCOPE;
            });
        checkIfValidScopeProvided(scope);
        return scope;
    }

    @Override
    public String resolveElementName(Class<?> clazz) {
        log.debug("Resolving element name for class: {}", clazz.getName());
        return resolveNameFromAnnotations(clazz);
    }

    @Override
    public ElementDefinition readElementDefinition(Class<?> clazz) {
        var elementDefinition = super.readElementDefinition(clazz);
        elementDefinition.setIsPrimary(resolveIfIsPrimary(clazz));
        log.debug("Element definition created: {}", elementDefinition);
        return elementDefinition;
    }

    @Override
    public ElementDefinition
getElementDefinitionContainerWithSpecifics(Class<?> clazz) {
        return new AnnotationElementDefinition();
    }

    private String resolveNameFromAnnotations(Class<?> clazz) {
        return ReflectionUtils.getValueFromAnnotation(clazz,
ua.gorobeos.contextor.context.annotations.Element.class, "value",
String.class)
            .filter(StringUtils::isNotBlank)
            .orElseGet(() -> {
                log.debug("No specific name in @Element annotation found on class:
{}", clazz.getName());
                return getDefaultElementName(clazz);
            });
    }

    private boolean resolveIfIsPrimary(Class<?> clazz) {
        return ReflectionUtils.isAnnotationPresentFullCheck(clazz,
Primary.class);
    }
}

```

Файл ElementDefinitionReaderFacadeImpl.java

Реалізує фасад, який інкапсулює функції для процесу зчитування мета інформації про об'єкти

```

package ua.gorobeos.contextor.context.readers;

import static
    ua.gorobeos.contextor.context.element.ElementDefinition.SINGLETON_SCOPE;

import java.util.Collection;
import java.util.Map;
import lombok.AccessLevel;
import ua.gorobeos.contextor.context.annotations.ContextConfig;
import ua.gorobeos.contextor.context.conditions.ConditionEvaluationUtils;
import ua.gorobeos.contextor.context.element.ConfigElementDefinition;
import ua.gorobeos.contextor.context.element.ElementDefinition;
import
    ua.gorobeos.contextor.context.readers.impl.ConfigElementDefinitionReader;
import ua.gorobeos.contextor.context.storage.ElementDefinitionHolder;
import ua.gorobeos.contextor.context.utils.ReflectionUtils;

public class ElementDefinitionReaderFacadeImpl implements
    ElementDefinitionReaderFacade {

    private enum DefinitionType {
        CONFIGURATION, ELEMENT
    }

    Map<DefinitionType, ElementDefinitionReader> elementDefinitionReader =
    Map.of(
        DefinitionType.CONFIGURATION, new ConfigElementDefinitionReader(),
        DefinitionType.ELEMENT, new AnnotatedBaseElementDefinitionReader()
    );
    ElementDefinitionHolder elementDefinitionHolder;

    @Override
    public void addElementDefinition(Class<?> clazz) {
        ElementDefinitionReader definitionReader = checkWhatReaderToUse(clazz);

        var elementDefinition = definitionReader.readElementDefinition(clazz);

        if
            (!checkIfElementConditionsArePassingOnlyIfSingleton(elementDefinition)) {
                return;
            }

        if (elementDefinition instanceof ConfigElementDefinition configDef) {
            configDef.getMethodDefinedElements()
                .stream()
                .filter(this::checkIfElementConditionsArePassingOnlyIfSingleton)
                .forEach(elementDefinitionHolder::addElementDefinition);
        }
        elementDefinitionHolder.addElementDefinition(elementDefinition);
    }

    private boolean
    checkIfElementConditionsArePassingOnlyIfSingleton(ElementDefinition
    elementDefinition) {
        if (!SINGLETON_SCOPE.equals(elementDefinition.getScope())) {
            log.debug("Element definition '{}' has scope '{}', so skipping
            conditional checks on definition registration", elementDefinition.getName(),
            elementDefinition.getScope());
            return true;
        }
    }

    var context =
    ConditionEvaluationUtils.evaluate(elementDefinition.getType());
    if (!context.isConditionalCheckPassed()) {

```

```

        log.warn("Conditional check failed for element: {}. Scope is
'Singleton', so skipping saving element definition",
elementDefinition.getName());
        log.warn("Conditional check results: {} ",
context.getConditionalCheckResults());
        return false;
    }
    return true;
}

private ElementDefinitionReader checkWhatReaderToUse(Class<?> clazz) {
    DefinitionType definitionType =
ReflectionUtils.isAnnotationPresentFullCheck(clazz, ContextConfig.class)
    ? DefinitionType.CONFIGURATION
    : DefinitionType.ELEMENT;
    ElementDefinitionReader reader =
elementDefinitionReader.get(definitionType);
    log.debug("Using {} reader for class: {}",
reader.getClass().getSimpleName(), clazz.getName());
    return reader;
}

@Override
public Collection<ElementDefinition> getElementDefinitions() {
    return elementDefinitionHolder.getElementDefinitions();
}
}

```

Файл ConditionalOnLinuxEvaluator.java

Реалізація функціональної задачі для перевірки системи на якій виконано запуск застосунку.

```

package ua.gorobeos.contextor.context.conditions.evaluators;

import lombok.extern.slf4j.Slf4j;
import ua.gorobeos.contextor.context.annotations.conditions.ConditionalOnLinux;
import ua.gorobeos.contextor.context.conditions.ConditionalContext;
import ua.gorobeos.contextor.context.conditions.ConditionalEvaluator;
import ua.gorobeos.contextor.context.utils.ReflectionUtils;

@Slf4j
public class ConditionalOnLinuxEvaluator implements ConditionalEvaluator {

    @Override
    public ConditionalContext evaluate(ConditionalContext context) {

        Class<?> element = context.getElementClass();
        if (ReflectionUtils.isAnnotationPresentFullCheck(element,
ConditionalOnLinux.class)) {
            log.debug("Evaluating {} for element {}",
ConditionalOnLinuxEvaluator.class.getSimpleName(), element.getName());
        } else {
            log.debug("{} is not present on element {}, skipping evaluation",
ConditionalOnLinux.class.getSimpleName(), element.getName());
            return context;
        }

        String osName = System.getProperty("os.name").toLowerCase();
        if (!osName.contains("linux")) {

```

```

        log.warn("Current OS is not Linux, {} evaluation failed",
ConditionalOnLinuxEvaluator.class.getSimpleName());
        context.setConditionalCheckPassed(false);
        context.getConditionalCheckResults().add(
            String.format("Current OS '%s' is not Linux, evaluation failed",
osName));
    }
    return context;
}
}

```

Файл ConditionalOnLinuxEvaluator.java

Реалізація функціональної задачі для перевірки системи на якій виконано запуск застосунку.

```

package ua.gorobeos.contextor.context.conditions.evaluators;

import
ua.gorobeos.contextor.context.annotations.conditions.ConditionalOnLinux;
import ua.gorobeos.contextor.context.conditions.ConditionalContext;
import ua.gorobeos.contextor.context.conditions.ConditionalEvaluator;
import ua.gorobeos.contextor.context.utils.ReflectionUtils;

public class ConditionalOnLinuxEvaluator implements ConditionalEvaluator {

    @Override
    public ConditionalContext evaluate(ConditionalContext context) {

        Class<?> element = context.getElementClass();
        if (ReflectionUtils.isAnnotationPresentFullCheck(element,
ConditionalOnLinux.class)) {
            log.debug("Evaluating {} for element {}",
ConditionalOnLinuxEvaluator.class.getSimpleName(), element.getName());
        } else {
            log.debug("{} is not present on element {}, skipping evaluation",
ConditionalOnLinux.class.getSimpleName(), element.getName());
            return context;
        }

        String osName = System.getProperty("os.name").toLowerCase();
        if (!osName.contains("linux")) {
            log.warn("Current OS is not Linux, {} evaluation failed",
ConditionalOnLinuxEvaluator.class.getSimpleName());
            context.setConditionalCheckPassed(false);
            context.getConditionalCheckResults().add(
                String.format("Current OS '%s' is not Linux, evaluation failed",
osName));
        }
        return context;
    }
}

```

Файл ConditionalOnPropertyEvaluator.java

Реалізація функціональної задачі для перевірки умови при створенні об'єкту на основі властивостей системи.

```

package ua.gorobeos.contextor.context.conditions.evaluators;

import static
ua.gorobeos.contextor.context.utils.ReflectionUtils.getValueFromAnnotation;
import static
ua.gorobeos.contextor.context.utils.ReflectionUtils.isAnnotationPresentFullCheck;

import
ua.gorobeos.contextor.context.annotations.conditions.ConditionalOnProperty;
import ua.gorobeos.contextor.context.conditions.ConditionalContext;
import ua.gorobeos.contextor.context.conditions.ConditionalEvaluator;
import ua.gorobeos.contextor.context.config.ConfigurationReader;

public class ConditionalOnPropertyEvaluator implements ConditionalEvaluator {

    @Override
    public ConditionalContext evaluate(ConditionalContext context) {
        Class<?> element = context.getElementClass();
        if (element == null) {
            return context;
        }
        if (!isAnnotationPresentFullCheck(element, ConditionalOnProperty.class))
        {
            return context;
        }

        var name = getValueFromAnnotation(element, ConditionalOnProperty.class,
"name", String.class);
        var value = getValueFromAnnotation(element, ConditionalOnProperty.class,
"value", String.class);
        if (name.isEmpty() || value.isEmpty()) {
            log.warn("ConditionalOnProperty annotation is missing 'name' or 'value'
attribute in class: {}", element.getName());
            context.setConditionalCheckPassed(false);
            context.getConditionalCheckResults().add(
                String.format("ConditionalOnProperty annotation is missing 'name'
or 'value' attribute in class: %s", element.getName()));
            return context;
        }
        var propertyValue = ConfigurationReader.getOrDefault(name.get(), null);
        if (propertyValue == null || !propertyValue.equals(value.get())) {
            log.warn("Property '{}' with value '{}' does not exist, {} evaluation
failed", name, value,
                ConditionalOnProperty.class.getSimpleName());
            context.setConditionalCheckPassed(false);
            context.getConditionalCheckResults().add(
                String.format("Property '%s' with value '%s' does not exist,
evaluation failed", name, value));
        }
        return context;
    }
}

```

Файл ConditionalOnWindowsEvaluator.java

Реалізація функціональної задачі для перевірки системи на якій виконано запуск застосунку.

```

package ua.gorobeos.contextor.context.conditions.evaluators;

import
ua.gorobeos.contextor.context.annotations.conditions.ConditionalOnWindows;
import ua.gorobeos.contextor.context.conditions.ConditionalContext;
import ua.gorobeos.contextor.context.conditions.ConditionalEvaluator;
import ua.gorobeos.contextor.context.utils.ReflectionUtils;

public class ConditionalOnWindowsEvaluator implements ConditionalEvaluator {

    @Override
    public ConditionalContext evaluate(ConditionalContext context) {
        if (!isAnnotationPresentFullCheck(context.getElementClass(),
ConditionalOnWindows.class)) {
            return context;
        }

        boolean isWindows =
System.getProperty("os.name").toLowerCase().contains("win");
        context.setConditionalCheckPassed(isWindows);
        if (!isWindows) {
            context.getConditionalCheckResults().add(
                String.format("Current OS is not Windows, %s evaluation failed",
ConditionalOnWindows.class.getSimpleName()));
            log.warn("Current OS is not Windows, {} evaluation failed",
ConditionalOnWindows.class.getSimpleName());
        }
        return context;
    }
}

```

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ____ ” _____ 2025 р.

**ІОС КОНТЕЙНЕР ДЛЯ СТВОРЕННЯ ОБ’ЄКТІВ НА ОСНОВІ
СКЛАДНИХ УМОВ**

Програма та методика тестування

КПІ.ПІ-1208.045200.04.51

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Владислав САРНАЦЬКИЙ

Нормоконтроль:

_____ Максим ГОЛОВЧЕНКО

Виконавець:

_____ Олексій ГОРОБЕЦЬ

Київ – 2025

ЗМІСТ

1	ОБ’ЄКТ ВИПРОБУВАНЬ.....	3
2	МЕТА ТЕСТУВАННЯ	4
3	МЕТОДИ ТЕСТУВАННЯ.....	5
4	ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ	6

1 ОБ'ЄКТ ВИПРОБУВАНЬ

Об'єктом випробування є ІоС фреймворк Contextor, перевірка проводиться, використовуючи сучасний SDK для Java OpenJDK 17 та операційну систему Windows 10.

2 МЕТА ТЕСТУВАННЯ

Метою тестування є наступне:

- перевірка правильності роботи програмного забезпечення відповідно до функціональних вимог, зокрема, функціонування контейнера ІоС, обробки анотацій, умовного створення об'єктів та управління життєвим циклом об'єктів;
- перевірка збереження даних у контейнері, їх кешування та доступності при повторному зверненні;
- перевірка коректної роботи механізму умовного створення об'єктів залежно від заданих умов;
- перевірка виявлення циклічних залежностей та коректного повідомлення про них;
- знаходження проблем, помилок і недоліків з метою їх усунення;
- перевірка рівня покриття коду тестами;
- перевірка відповідності проєкту принципам SOLID і виявлення надмірно складних методів чи дублювання логіки.

3 МЕТОДИ ТЕСТУВАННЯ

Для тестування програмного забезпечення використовуються такі методи:

- Статичне тестування використовується для аналізу коду на предмет дотримання стандартів, виявлення дублювання коду, потенційно небезпечних конструкцій, високої когнітивної складності та інших параметрів.
- Динамічне тестування проведене, застосовуючи автоматичні юніт та інтеграційні тести з використанням відповідних платформ для тестування та перегляду результатів виконання, а також аналіз відсоткового покриття тестами вихідного коду.
- Функціональне тестування застосовувалося для перевірки того, чи правильно контейнер обробляє функціональні вимоги: створення об'єктів, обробка залежностей, кешування, умовна логіка.
- Мануальне тестування було використано для тестування окремих тест кейсів, які складно змодельовати автоматично.
- Тестування «чорної скриньки» використовувалося при функціональному тестуванні та інтеграційному тестуванні, оскільки перевірялися тільки вхідні та вихідні дані без заглиблення у внутрішню реалізацію.
- Тестування «білої скриньки» використовувалося при unit-тестуванні, здійснювалася перевірка конкретної логіки методів, умов, взаємодій між компонентами.
- Тестування «сірої скриньки» проводилося для перевірки внутрішніх алгоритмів, таких як сканування класів, побудова дерева залежностей, кешування, коли частково була відома внутрішня структура, але перевірка велася переважно через API.

4 ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Під час проведення тестування будуть використовуватись наступні допоміжні засоби:

- JUnit 5 – буде використовуватись для написання юніт та інтеграційних тестів. Він забезпечить інфраструктуру для створення та виконання тестових випадків.
- Mockito – буде застосовуватись для створення моків залежностей під час unit-тестування, що дозволить ізолювати окремі компоненти системи.
- JaCoCo – буде використовуватись для аналізу покриття тестами вихідного коду. Він дозволить оцінити покриття класів, методів та рядків.
- SonarCloud – буде застосовано для проведення статичного аналізу коду з метою виявлення потенційних помилок, дублювань, порушень стилю та складних ділянок коду.
- IntelliJ IDEA – буде використовуватись як основне середовище розробки та запуску тестів. Воно забезпечить зручний інтерфейс для керування тестами та перегляду результатів.

Порядок проведення тестування буде наступним:

- Буде проведено статичний аналіз коду для перевірки відповідності стандартам, виявлення дублікатів, помилок та потенційно небезпечних конструкцій.
- Буде проведено юніт-тестування для перевірки правильності роботи окремих методів та компонентів.
- Буде проведено інтеграційне тестування для перевірки взаємодії між модулями контейнера.
- Буде виконано аналіз покриття тестами за допомогою утиліти JaCoCo.
- Буде проведено мануальне тестування для перевірки кейсів, які складно автоматизувати.

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ” _____ 2025 р.

**ІОС КОНТЕЙНЕР ДЛЯ СТВОРЕННЯ ОБ’ЄКТІВ НА ОСНОВІ
СКЛАДНИХ УМОВ**

Керівництво користувача

КПІ.ПІ-1208.045200.05.34

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Владислав САРНАЦЬКИЙ

Нормоконтроль:

_____ Максим ГОЛОВЧЕНКО

Виконавець:

_____ Олексій ГОРОБЕЦЬ

Київ – 2025

ЗМІСТ

1	ПРИЗНАЧЕННЯ ПРОГРАМИ	3
2	ПІДГОТОВКА ДО РОБОТИ З ПРОГРАМНИМ ЗАБЕЗПЕЧЕННЯМ.....	4
2.1	Системні вимоги для коректної роботи.....	4
2.2	Завантаження застосунку	5
2.3	Перевірка коректної роботи.....	6
3	ВИКОНАННЯ ПРОГРАМИ	7

1 ПРИЗНАЧЕННЯ ПРОГРАМИ

Програмне забезпечення Contextor є власною розробкою ІоС контейнера, призначеного для автоматизації створення, управління життєвим циклом і контролю залежностей об'єктів у Java-додатках. Контейнер дозволяє створювати об'єкти не лише за стандартними правилами ін'єкції залежностей, але й на основі складних умов, зокрема поточного часу, наявності файлів або доступності зовнішніх веб-сервісів.

Кінцева збірка програмного забезпечення представляє собою .jar бібліотеку, яка може бути легко підключена до проєкту через систему керування залежностями, наприклад, Maven. Бібліотека підтримує автоматичне сканування пакетів, обробку умов створення об'єктів, логування, а також включає API для взаємодії з контекстом контейнера.

Інсталяційна версія Contextor опублікована у публічному Maven-репозиторії та доступна для використання через конфігураційні файли систем збірки (наприклад, pom.xml).

2 ПІДГОТОВКА ДО РОБОТИ З ПРОГРАМНИМ ЗАБЕЗПЕЧЕННЯМ

2.1 Системні вимоги для коректної роботи

Для успішної роботи даного застосунку необхідне виконання наступних вимог.

Мінімальні системні вимоги

- Операційна система: Windows 10 / Linux / macOS;
- Процесор: Intel Core i5 або еквівалентний;
- Оперативна пам'ять (RAM): 4 ГБ;
- Вільне місце на жорсткому диску: 1 ГБ;
- Встановлене середовище розробки: IntelliJ IDEA або інше сумісне з JDK;
- Java Development Kit (JDK): версія 17 або вище;
- Доступ до Maven-репозиторію (локального або віддаленого).

Рекомендовані системні вимоги:

- Операційна система: Windows 10 / сучасний дистрибутив Linux / macOS 13 або новіше;
- Процесор: Intel Core i7 або AMD Ryzen 7;
- Оперативна пам'ять (RAM): 16 ГБ;
- Вільне місце на диску: 10 ГБ;
- Стабільне підключення до Інтернету для доступу до репозиторіїв залежностей;
- Оновлене середовище IntelliJ IDEA (версія 2024.1 або новіша);
- Підтримка Java Virtual Machine з доступом до параметрів середовища виконання.

2.2 Завантаження застосунку

Для інтеграції програмного забезпечення Contextor необхідно додати бібліотеку до системи збірки проєкту. Застосунок доступний через публічне сховище Maven Central у вигляді jar-бібліотеки.

Підключення до Maven-проєкту

Для інтеграції необхідно додати наступну залежність до секції `<dependencies>` у файлі `pom.xml`. Лістинг коду:

```
<dependency>
  <groupId>io.github.contextor-framework</groupId>
  <artifactId>contextor</artifactId>
  <version>0.0.1</version>
</dependency>
```

Додаткові репозиторії зазначати не потрібно, оскільки бібліотека розміщується у Maven Central.

Підключення до Gradle-проєкту

Для інтеграції у проєкт, який використовує Gradle, потрібно додати такий запис до секції `dependencies` у файлі `build.gradle`. Лістинг коду:

```
dependencies {
  implementation 'io.github.contextor-framework:contextor:0.0.1'
}
```

При використанні Gradle Kotlin DSL (`build.gradle.kts`) застосовується наступний запис:

```
dependencies {
  implementation("io.github.contextor-framework:contextor:0.0.1")
}
```

Альтернативне завантаження

Якщо використання систем керування залежностями не передбачається, `.jar` файл можна завантажити вручну за посиланням:

<https://search.maven.org/artifact/io.github.contextor-framework/contextor/0.0.1/jar>

Після завантаження необхідно скопіювати файл у каталог проєкту та додати його до Classpath вручну відповідно до використовуваного середовища розробки.

2.3 Перевірка коректної роботи

Необхідно переконатися, що збірка проєкту проходить успішно після додавання залежності Contextor до системи збірки Maven або Gradle.

Для Maven потрібно виконати команду *«mvn clean install»*.

Для Gradle потрібно виконати команду *«./gradlew build»*.

За потреби варто виконати очищення кешу збірки та повторити компіляцію. Це дозволяє переконатися у відсутності конфліктів між залежностями та у правильності підключення бібліотеки.

3 ВИКОНАННЯ ПРОГРАМИ

Розглянемо основні принципи роботи із фреймворком Contextor.

Основні концепції

1. Анотація @Element

Анотація @Element є основою фреймворку Contextor. Вона використовується для позначення класу як компонента, яким має керувати фреймворк. Класи, анотовані за допомогою @Element, автоматично виявляються та реєструються в контексті програми.

Анотація @Element містить додаткове поле для явного задання імені об'єкту. За замовчуванням в якості ім'я використовується назва класу з маленької букви.

Приклад використання наведено на рисунку 3.1.

```

14 // Фінальне ім'я -> defaultNameClass
15 @Element
16 class DefaultNameClass {
17
18 }
19
20 // Фінальне ім'я -> superName
21 @Element("superName")
22 class SpecifiedNameClass {
23
24 }
```

Рисунок 3.1 – Позначення класу для створення фреймворком Contextor

2. Анотація @Scope

Анотація @Scope у фреймворку Contextor використовується для визначення області дії компонента. Вона дозволяє вказати, як і коли створюються та керуються екземпляри компонента в контейнері.

Існують дві області, які можна задати елементу:

1. Singleton – це область видимості за замовчуванням. Об'єкт, визначений як Singleton, створюється та ініціалізується лише один раз при

запуску проекту або під час першої ініціалізації контейнера. Після створення єдиний екземпляр цього об'єкта зберігається в контексті контейнера. Кожний наступний запит на отримання цього об'єкта з контейнера буде повертати посилання на той самий кешований екземпляр. Такий підхід забезпечує ефективне використання ресурсів і консистентність стану об'єкта впродовж усього життєвого циклу застосунку.

2. Prototype (прототип) – це область видимості, яка передбачає створення нового екземпляра об'єкта щоразу, коли до нього звертаються через контейнер. На відміну від Singleton, об'єкти з областю видимості Prototype не зберігаються в кеші контейнера. Кожен запит до контейнера призводить до повторного конструювання та ініціалізації об'єкта, що робить його ідеальним для об'єктів, які не є потокобезпечними або мають унікальний стан для кожного використання.

Приклад створення об'єктів з різними областями видимості наведено на рисунку 3.2.

```

29 // Область видимості -> prototype
30 @Element
31 @Scope(PROTOTYPE_SCOPE)
32 class PrototypeClass {
33
34 }
35
36 // Область видимості -> singleton
37 @Element
38 @Scope(SINGLETON_SCOPE)
39 class SingletonClass {
40
41 }
42
43 // Область видимості -> singleton
44 @Element
45 class AnotherSingletonClass {
46
47 }

```

Рисунок 3.2 – Задання області видимості класам

3. Анотація @Primary

Анотація @Primary використовується для позначення того, що певному компоненту слід надавати пріоритет, коли в контексті програми присутні кілька елементів одного типу. Це допомагає вирішити неоднозначність під час впровадження залежностей, позначаючи один елемент, як вибір за замовчуванням.

Приклад використання наведено на рисунку 3.3.

```

53 // Клас з анотацією Primary, який буде використовуватися як основний сервіс
54 @Element
55 @Primary
56 class PrimaryService {
57     public String getMessage() {
58         return "This is the primary service.";
59     }
60 }
61
62
63 @Element
64 class SecondaryService {
65     public String getMessage() {
66         return "This is the secondary service.";
67     }
68 }

```

Рисунок 3.3 – Задання @Primary пріоритету

4. Анотація @Qualifier

Анотація @Qualifier використовується для вирішення неоднозначностей, коли в контексті програми існує кілька компонентів одного типу. Вона дозволяє вказати, який саме елемент слід впровадити, надаючи унікальне ім'я або ідентифікатор. Застосовується до параметрів конструктора.

Наведемо приклад використання анотації @Qualifier. Для цього задамо наступну структуру класів: контракт GreetingService з визначеним методом «greet()» та дві його реалізації EnglishGreetingService та SpanishGreetingService (рисунок 3.4). Лістинг коду наведено на рисунку 3.5.

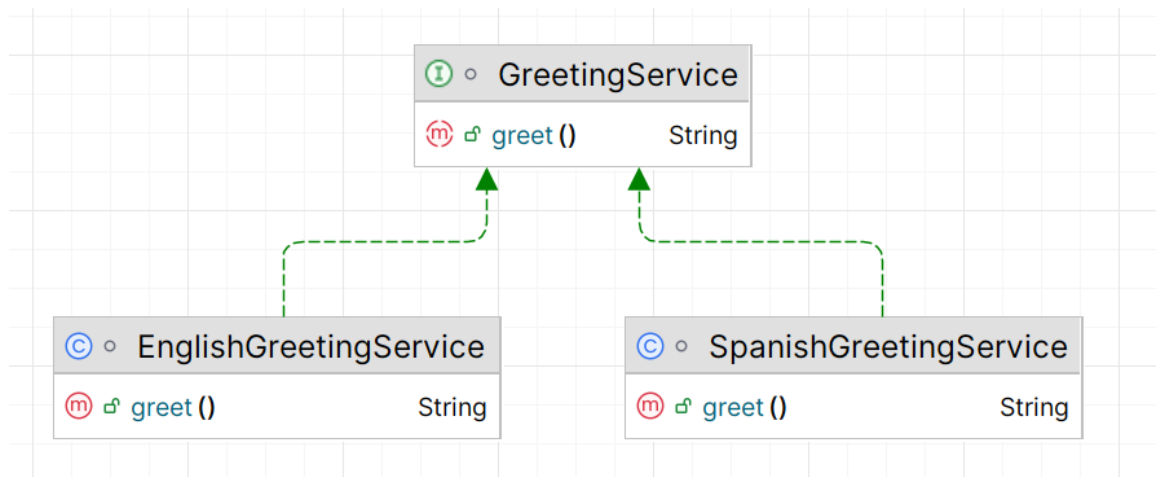


Рисунок 3.4 – Схема створених класів

```

73      @Element
74      @interface GreetingService {
75          @interface String greet();
76      }
77
78
79      @Element("englishService")
80      class EnglishGreetingService implements GreetingService {
81          public String greet() {
82              return "Hello!";
83          }
84      }
85
86      @Element("spanishService")
87      class SpanishGreetingService implements GreetingService {
88          public String greet() {
89              return ";Hola!";
90          }
91      }
92  
```

Рисунок 3.5 – Лістинг коду створених класів

Далі наведемо створимо ще один клас, який буде залежати від контракту `GreetingService`. Оскільки в контексті існує два об'єкти, які реалізують цей контракт, без додаткових уточнень контейнер не зможе вирішити яку саме імплементацію використати. Саме в таких випадках і можна використати анотацію `@Qualifier`, щоб зазначити конкретну реалізацію (рисунок 3.6).

```

93     @Element
94     class GreetingClient {
95         GreetingService greetingService;
96
97         public GreetingClient(@Qualifier("englishService") GreetingService greetingService) {
98             this.greetingService = greetingService;
99         }
100     }

```

Рисунок 3.6 – Лістинг коду класу GreetingClient

Як видно з рисунку 3.6 було зазначено за допомогою анотації `@Qualifier`, що потрібно встановити залежність "englishService".

5. Аннотація `@Injected`

Анотація `@Injected` використовується для позначення конструктора в класі, який слід використовувати для створення об'єктів у рамках фреймворку Contextor. Це особливо корисно, коли клас має кілька конструкторів, і користувачу необхідно явно вказати, який з них слід використовувати для впровадження залежностей.

Приклад застосування наведено на рисунку 3.7.

```

107     @Element
108     class MultipleConstructorsClass {
109
110         final String message;
111
112         //Цей конструктор буде використано
113         @Injected
114         public MultipleConstructorsClass(String message) {
115             this.message = message;
116         }
117
118         //Конструктор проігноровано
119         public MultipleConstructorsClass() {
120             this.message = "Default constructor called";
121         }
122         //Конструктор проігноровано
123     @ public MultipleConstructorsClass(GreetingClient greetingClient) {
124         this.message = greetingClient.greetingService.greet();
125     }
126 }

```

Рисунок 3.7 – Приклад використання анотації `@Injected`

6. Анотація @ElementScan

Анотація @ElementScan використовується для визначення базових пакетів, які слід сканувати на наявність класів, які необхідно додати у контекст фреймворку Contextor. Вона дозволяє фреймворку виявляти та реєструвати класи, анотовані за допомогою @Element, у вказаних пакетах рекурсивно.

Приклад використання цієї анотації наведено на рисунку 3.8.

```

133 // Вказаний пакет для сканування елементів
134 @ElementScan(basePackages = {"ua.gorobeos.contextor.context"})
135 class Main {
136     public static void main(String[] args) {
137         ContextHolder context = ContextHolder.initializeContext(Main.class);
138     }
139 }
140

```

Рисунок 3.8 – Приклад використання анотації @ElementScan

7. Анотація @ContextConfig та @ExternalElement

Анотація @ContextConfig використовується для позначення класу як класу конфігурації у фреймворку Contextor. Класи, анотовані за допомогою @ContextConfig, можуть визначати методи, анотовані за допомогою @ExternalElement, для оголошення та налаштування елементів, якими керуватиме фреймворк.

Методи, анотовані за допомогою @ExternalElement у класі @ContextConfig, розглядаються як фабричні методи для створення та керування елемент.

Оскільки @ContextConfig сам анотований за допомогою @Element, клас конфігурації автоматично реєструється як компонент у контексті. Приклад використання

Приклад використання наведено на рисунку 3.9.

```

146  @ContextConfig
147  class GreetingServiceConfig {
148
149      @ExternalElement
150      public SpanishGreetingService spanishGreetingService() {
151          return new SpanishGreetingService();
152      }
153
154      @ExternalElement
155      public EnglishGreetingService englishGreetingService() {
156          return new EnglishGreetingService();
157      }
158
159      @ExternalElement
160      public GreetingClient greetingClient(@Qualifier("englishService") GreetingService greetingService) {
161          return new GreetingClient(greetingService);
162      }
163  }

```

Рисунок 3.9 – Приклад задання створення елементів через методи та конфігураційний клас

8. Умовні анотації

`@ConditionalOnFilePresence` ця анотація використовується для умовного ввімкнення компонента або конфігурації, якщо певний файл присутній у файловій системі. Вона перевіряє існування файлу під час виконання.

`@ConditionalOnLinux` анотація активує компонент або конфігурацію, лише якщо програма працює в операційній системі на базі Linux. Зазвичай вона перевіряє властивість назви ОС.

`@ConditionalOnProperty` анотація вмикає компонент або конфігурацію на основі наявності та значення певної властивості в конфігурації програми у `application.properties` файлі.

`@ConditionalOnWebRequest` ця анотація активує компонент або конфігурацію, якщо заданий веб запит повернув успішну відповідь.

`@ConditionalOnWindows` створює компонент лише, якщо програма працює в операційній системі Windows. Вона перевіряє властивість назви ОС, щоб визначити платформу.

Приклад використання цих анотацій наведений на рисунку 3.10.

```

171     @Element
172     @ConditionalOnFilePresence(filePaths = "/path/to/specific/file.txt")
173     class FileBasedBean {
174
175     }
176
177     @Element
178     @ConditionalOnLinux
179     class LinuxBean {
180
181     }
182
183     @Element
184     @ConditionalOnWindows
185     class WindowsBean {
186
187     }
188
189     @Element
190     @ConditionalOnProperty(name = "app.feature.enabled", value = "true")
191     class PropertyBasedBean {
192
193     }
194
195     @Element
196     @ConditionalOnWebRequest(method = GET, url = "/api/data")
197     class WebRequestBean {
198
199     }

```

Рисунок 3.10 – Приклад створення компонентів за заданими умовами

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ____ ” _____ 2025 р.

**ІОС КОНТЕЙНЕР ДЛЯ СТВОРЕННЯ ОБ’ЄКТІВ НА ОСНОВІ
СКЛАДНИХ УМОВ**

Графічний матеріал

КПІ.ПІ-1208.045200.06.99

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Владислав САРНАЦЬКИЙ

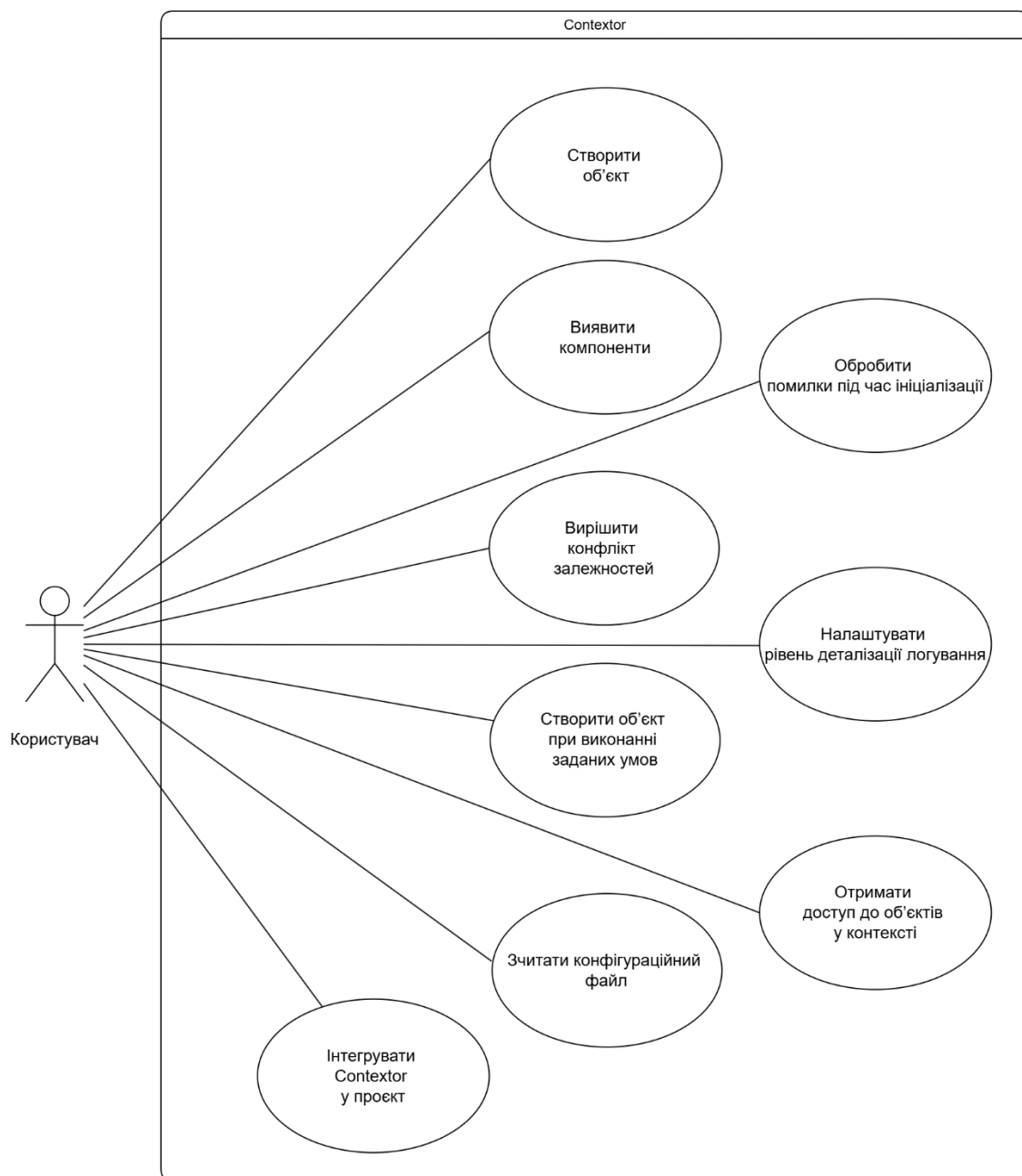
Нормоконтроль:

_____ Максим ГОЛОВЧЕНКО

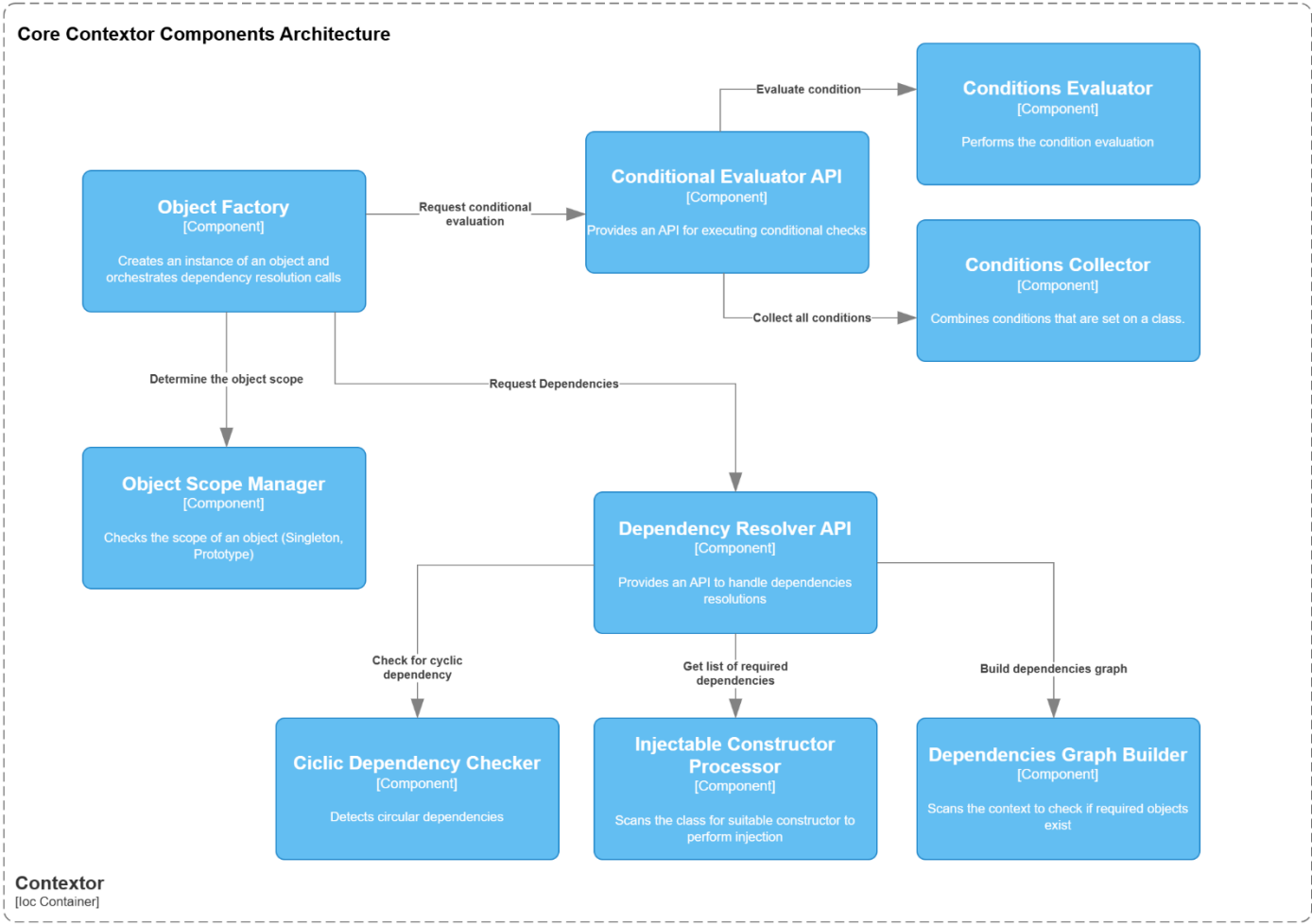
Виконавець:

_____ Олексій ГОРОБЕЦЬ

Київ – 2025

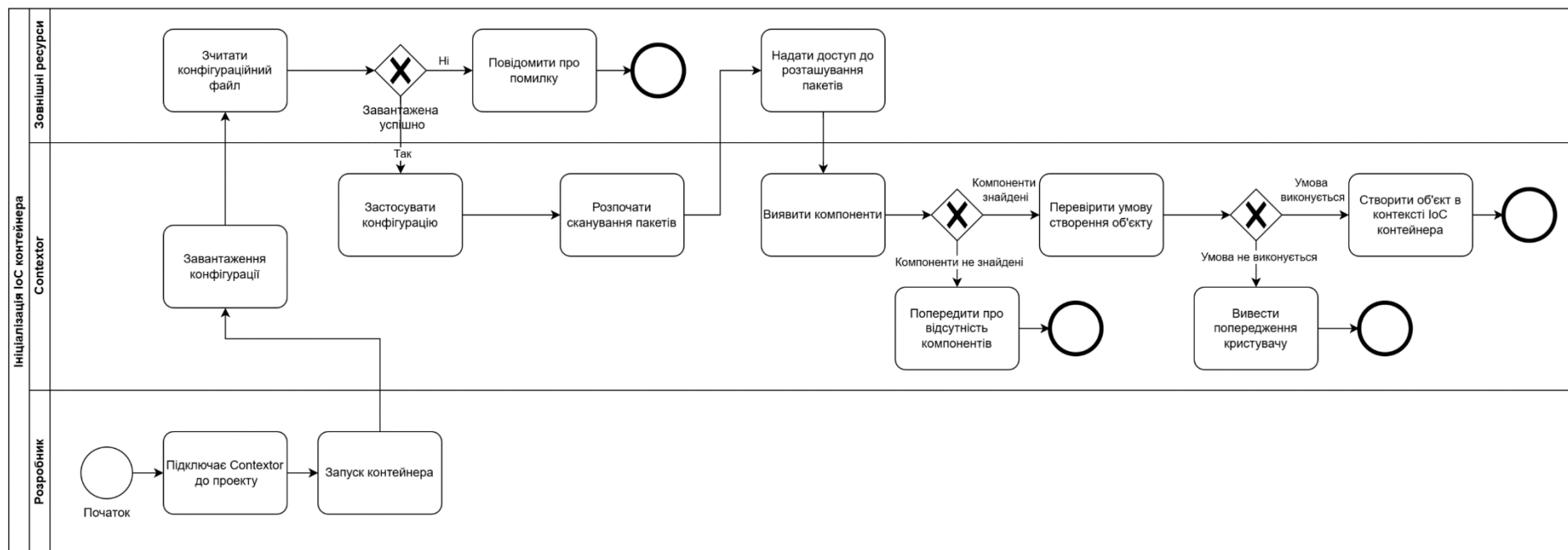


					КПІ.ІП-1208.045200.06.99.CCB				
					Діаграма варіантів використань	Лит.		Маса	Масштаб
Зм.	Арк.	№ докум.	Підп.	Дата					
Розробив	Горобець О. С.								
Перевірив	Сарнацький В. В.								
Т. контр.									
						Аркуш 1		Аркушів 1	
Н. контр.	Головченко М. М.				РОЗРОБКА ІОС КОНТЕЙНЕРА ДЛЯ СТВОРЕННЯ ОБ'ЄКТІВ НА ОСНОВІ СКЛАДНИХ УМОВ	КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-12			
Затвердив	Жаріков Е.В.								



					КПІ.ІП-1208.045200.06.99.CCM							
					Діаграма компонентів	Літера		Маса		Масштаб		
Зм.	Арк.	№ документа	Підпис	Дата								
Розробив		Горобець О. С.										
Перевірив		Сарнацький В. В.										
Т. контр.							Аркуш 1		Аркушів 1			
Н. контр.		Головченко М. М.			РОЗРОБКА ІОС КОНТЕЙНЕРА ДЛЯ СТВОРЕННЯ ОБ'ЄКТІВ НА ОСНОВІ					КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-12		
Затвердив		Жаріков Е.В.			СКЛАДНИХ УМОВ							

ДІАГРАМА БІЗНЕС ПРОЦЕСУ «ІНІЦІАЛІЗАЦІЯ ІОС КОНТЕЙНЕРУ»



Демонстраційний плакат до дипломного проекту

Розробка ІоС контейнера для створення об'єктів на основі складних умов

Виконав студент гр. ІІ-12 Горобець О. С.

Керівник Сарнацький В. В.