НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Кафедра інформаційних систем та технологій

Тема_____CI server_____

Курсова робота
З дисципліни «Технології розроблення програмного забезпечення»

Керівник
доц. Амонс О.А.
«Допущений до захисту»
_____
(Особистий підпис керівника)
« » _____2025р.
Захищений з оцінкою
_____
(оцінка)

Виконавець
ст. _____Губар Б.О._____
залікова книжка № ___ – _____
гр. ____ІА-31_____
_____
(особистий підпис виконавця)
« »_____2025р.

Члени комісії:

_____ _____
(особистий підпис) (розшифровка підпису)

_____ _____
(особистий підпис) (розшифровка підпису)

Київ – 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

(назва навчального закладу)

Кафедра ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Дисципліна «Технології розроблення програмного забезпечення»

Курс 3 Група ІА-31 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Губара Богдана Олександровича

(прізвище, ім’я, по батькові)

1.Тема роботи CI server

2. Строк здачі студентом закінченої роботи_____

3. Вихідні дані до роботи:

Сервер безперервної інтеграції (CI Server) повинен нагадувати функціонал таких систем, як Jenkins або GitLab CI, з можливостями автоматичного моніторингу репозиторіїв коду (наприклад, Git), запуску налаштовуваних пайплайнів (компіляція, тестування, аналіз коду), збереження артефактів збірки, ведення детальної історії та логів, а також відправки сповіщень (наприклад, email або через месенджер) про успіх чи провал збірки.

4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці)

Проектування системи (огляд існуючих рішень, опис проекту, вимоги до застосунків системи (функціональні/нефункціональні), сценарії використання системи, концептуальна модель системи, вибір бази даних, мови програмування та середовища розробки, проєктування розгортання системи), реалізація компонентів системи (структура бази даних, вибір і обґрунтування патернів реалізації, інструкція користувача).

Додатки:

_____

5.Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

_____

_____

_____

_____

6. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Підбір та вивчення літератури	30.09.2025	
2.	Проектування та написання розділу 1	31.10.2025	
3.	Розробка та написання розділу 2	20.11.2025	
4.	Подання курсової роботи на перевірку	25.11.2025	
5.	Захист курсової роботи	08.12.2025	
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			
15.			
16.			
17.			

Студент _____ _____Богдан ГУБАР_____
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник_____ __Олександр АМОНС__
(підпис) (Ім’я ПРІЗВИЩЕ)

«____»_____ 20___ р.

ЗМІСТ

ВСТУП

Сучасні практики розробки програмного забезпечення стали основою для прискорення інноваційних циклів та підвищення загальної якості цифрових продуктів, що стали важливою частиною бізнесу та повсякденного життя. Однією з основних складових цих практик (часто об'єднаних під терміном DevOps) є сервери безперервної інтеграції (CI), які виконують функцію повної автоматизації процесів збірки, тестування та розгортання програмного коду. Безперервна інтеграція (Continuous Integration) — це інженерна практика, яка вимагає від розробників частої інтеграції змін коду в центральний репозиторій, після чого автоматично запускаються процеси збірки та тестування. Розробка CI-сервера є важливим завданням, яке вимагає глибоких знань в області архітектури програмних систем, автоматизації та принципів управління конфігураціями. У даній курсовій роботі розглядається розробка серверного програмного забезпечення, яке здатне коректно моніторити системи контролю версій (наприклад, Git), запускати визначені користувачем пайплайни (pipelines) та забезпечувати стабільне функціонування в умовах активної паралельної розробки.

Основною метою цієї роботи є створення CI-сервера, який здатний не тільки коректно обробляти тригери (наприклад, webhooks) і формувати чергу завдань, але й виконувати скрипти збірки та тестування в ізольованому середовищі, вести детальну історію та логи всіх запусків, а також забезпечувати обробку завдань у багатопотоковому режимі або за допомогою розподілених агентів. Важливим аспектом є ефективність роботи сервера, що забезпечується використанням відповідних архітектурних рішень та алгоритмів. Одним з найбільш важливих завдань є розпізнавання конфігураційних файлів проекту (на кшталт Jenkinsfile або .gitlab-ci.yml), аналіз залежностей між етапами (stages) та правильне формування відповіді про статус виконання. Для цього необхідно не тільки забезпечити коректне виконання зовнішніх команд, але й реалізувати механізми обробки помилок виконання та формування фінальних звітів і статусів (успіх/провал).

Ще однією ключовою частиною є збір логів та артефактів збірки. Збереження детальної історії є важливою складовою частиною функціонування CI-сервера, оскільки це дозволяє не лише відслідковувати результат кожної інтеграції коду, але й швидко виявляти причини збоїв у тестах або процесі збірки, аналізувати продуктивність та стабільність кодової бази. Обробка завдань у багатопотоковому режимі або через систему черг дозволяє ефективно розподіляти обчислювальні ресурси сервера і забезпечувати високий рівень продуктивності при роботі з великою кількістю розробників та проектів одночасно.

Особливу увагу слід приділити розробці системи управління виконавцями (executors), яка повинна бути гнучкою і масштабованою. Використання пулу потоків або динамічних агентів дозволяє розподіляти навантаження між кількома процесами, що підвищує загальну пропускну здатність сервера. У той же час, асинхронна модель обробки є найбільш ефективною для управління тривалими процесами (як-от повне регресійне тестування), оскільки дозволяє серверу залишатися чутливим до нових тригерів, не блокуючи основний процес.

У результаті розробки CI-сервера в рамках цієї курсової роботи, буде реалізовано ефективне серверне програмне забезпечення, яке здатне працювати в умовах високої частоти комітів, надавати розробникам миттєвий зворотний зв'язок про якість їхніх змін і збирати статистичні дані (логи та звіти), що дозволяють моніторити ефективність усього процесу розробки та приймати відповідні рішення щодо його вдосконалення.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

Сучасний ринок програмного забезпечення для автоматизації процесів розробки (CI/CD) представлений великою кількістю рішень, що задовольняють різноманітні потреби команд. Серед них можна виділити як потужні, універсальні standalone-сервери, так і інтегровані в платформи рішення, що надають CI як частину загальної екосистеми. Огляд цих рішень дозволяє краще зрозуміти сучасні тенденції у сфері DevOps та визначити переваги і недоліки різних підходів.

Jenkins

Jenkins — це один з найбільш популярних і широко використовуваних CI-серверів у світі. Він розроблений з відкритим вихідним кодом і має надзвичайно багатий набір можливостей завдяки величезній екосистемі плагінів (тисячі розширень). Jenkins підтримує інтеграцію практично з будь-якими інструментами та технологіями, забезпечує розподілені збірки (модель master-agent) та гнучке налаштування пайплайнів як через графічний інтерфейс, так і через код (використовуючи Jenkinsfile).

Однак, незважаючи на свою універсальність, Jenkins має певні недоліки. Його налаштування та підтримка можуть бути складними та ресурсозатратними. Велика кількість плагінів може призводити до проблем із сумісністю ("plugin hell"), а його архітектура, особливо в частині основного контролера, може ставати "вузьким місцем" у високонавантажених системах порівняно з новітніми хмарними рішеннями.

GitLab CI/CD

GitLab CI/CD — це ще одне надзвичайно популярне рішення, яке здобуло велику популярність завдяки своїй глибокій інтеграції безпосередньо в платформу управління репозиторіями GitLab. Конфігурація пайплайнів відбувається

декларативно через один файл .gitlab-ci.yml у корені проекту. GitLab CI використовує модель "runner-ів" (виконавців), що дозволяє легко масштабувати ресурси для збірки та забезпечувати високу продуктивність. Завдяки цьому він став вибором для багатьох команд, які шукають єдине "all-in-one" рішення.

Однією з переваг GitLab CI є його здатність виступати як повний DevOps-інструментарій, що включає не лише CI/CD, але й registry для контейнерів, інструменти сканування безпеки (SAST/DAST) та моніторинг. Це дозволяє ефективно побудувати весь життєвий цикл розробки в єдиному середовищі.

Однак, на відміну від Jenkins, GitLab CI тісно пов'язаний з екосистемою GitLab, що може бути обмеженням, якщо компанія використовує інші системи контролю версій (наприклад, GitHub або Bitbucket) як основні.

GitHub Actions

GitHub Actions — це відносно нове, але вже дуже потужне рішення, інтегроване безпосередньо в GitHub. Воно відоме своєю гнучкістю та подієво-орієнтованою моделлю: пайплайни (workflows) можуть бути активовані не лише комітами, але й створенням issue, коментарями, релізами та іншими подіями. GitHub Actions використовує конфігурацію через YAML-файли і має великий Marketplace готових "дій" (actions), що дозволяє швидко будувати складні процеси.

Проте GitHub Actions, подібно до GitLab, є частиною своєї платформи. Хоча він може працювати зі сторонніми інструментами, його найбільша ефективність досягається при роботі в екосистемі GitHub. Керування власними (self-hosted) runner-ами може бути менш гнучким, ніж зріла master/agent архітектура Jenkins.

Кожне з існуючих рішень має свої переваги та обмеження, що робить їх підходящими для різних сценаріїв використання. Для максимальної гнучкості та інтеграції з різнорідними інструментами часто обирають Jenkins. Для команд, що прагнуть отримати єдину, тісно інтегровану платформу, ідеальними є GitLab CI або GitHub Actions. Тому вибір рішення для створення CI-системи значною мірою

залежить від вимог до гнучкості, наявної інфраструктури (зокрема, хостингу репозиторіїв) та специфіки проєкту.

1.2. Загальний опис проєкту

Проєкт має на меті розробку CI-сервера, який здатний автоматично реагувати на зміни в репозиторіях коду (наприклад, через webhooks) та виконувати визначені користувачем пайплайни (pipelines) — послідовності кроків, таких як компіляція, тестування та аналіз коду. Сервер повинен мати здатність запускати ці завдання у відповідь на тригери, керувати чергою завдань, а також вести детальну історію та логи всіх виконань. Окрім цього, важливим елементом є реалізація обробки завдань у багатопотоковому режимі або за допомогою системи черг для забезпечення високої ефективності і масштабованості при обробці великої кількості одночасних збірок від різних проєктів.

Проєкт буде реалізовано як серверне програмне забезпечення, яке може працювати на операційних системах з підтримкою мережевих протоколів. Він забезпечить підтримку циклу розробки шляхом надання розробникам швидкого зворотного зв'язку про якість їхніх змін, а також дозволить ефективно моніторити і аналізувати стабільність кодової бази через збір логів та артефактів збірки.

В рамках проєкту сервер буде реалізований з урахуванням важливих аспектів надійності, гнучкості та ефективності. Одним з основних завдань є коректна реалізація системи виконання пайплайнів, що включає парсинг конфігурації, виконання скриптів у визначеному середовищі та управління залежностями між етапами (stages). Особливу увагу буде приділено обробці помилок виконання (наприклад, провал тестів або помилка компіляції) і забезпеченню правильного зворотного зв'язку для клієнтів (через сповіщення або статуси в системі контролю версій), що дозволяє уникнути непередбачуваних ситуацій та забезпечити високий рівень стабільності процесу розробки.

Важливим компонентом є система логування та зберігання артефактів. Вона дозволить відслідковувати вивід консолі для кожного завдання, час виконання, фінальний статус (успіх/провал) та інші параметри, що є важливими для аналізу причин збоїв, а також для виявлення можливих проблем з продуктивністю тестів. Збережені артефакти (наприклад, скомпільовані бінарні файли або звіти про покриття коду) будуть використовуватися для наступних етапів або для ручного аналізу.

Обробка завдань у багатопотоковому режимі або через систему черг є ще одним важливим аспектом проєкту. Це дозволить ефективно обробляти багато запитів на збірку одночасно, зменшуючи час очікування в черзі і підвищуючи пропускну здатність системи. Вибір моделі обробки завдань буде залежати від вимог до ізоляції середовищ збірки та специфіки задач, що стоять перед сервером.

Загалом, цей проєкт спрямований на створення високопродуктивного і масштабованого CI-сервера, який здатний не тільки автоматизувати процеси збірки і тестування, але й забезпечувати ефективну роботу з високою частотою комітів, що є критично важливим аспектом для сучасних гнучких методологій розробки (Agile, DevOps).

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

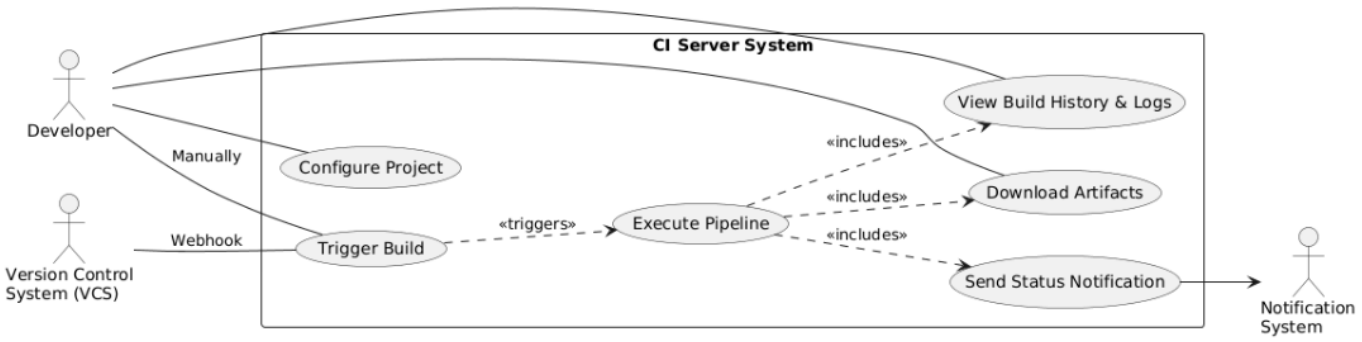

Рис. 1.3.1.1 – Діаграма використання

Функціональні вимоги до системи

- **а) Конфігурація проекту (Configure Project):** Система повинна дозволяти користувачу (Розробнику) визначати конфігурацію для конкретного проекту. Це включає посилання на репозиторій, визначення кроків (stages) та завдань (jobs) пайплайну (наприклад, через YAML-файл або графічний інтерфейс).
- **б) Обробка тригерів (Trigger Build):** Сервер повинен коректно обробляти вхідні тригери для запуску пайплайнів, підтримуючи як автоматичний запуск (наприклад, Webhook від Системи Контролю Версій (VCS) при новому коміті), так і ручний запуск Розробником.
- **в) Виконання пайплайнів (Execute Pipeline):** Система повинна забезпечувати надійне та послідовне виконання визначених у конфігурації завдань (компіляція, тестування, аналіз коду, деплой).
- **г) Керування чергою завдань:** Сервер повинен дозволяти керувати чергою завдань, які очікують на виконання. Наприклад, обробляти завдання в порядку надходження (FIFO) або на основі пріоритетів.
- **д) Налаштування обробки в багатопотоковому або розподіленому форматі:** Система повинна підтримувати обробку завдань у багатопотоковому режимі (кілька виконавців/executors на одній машині) або в розподіленому (використання кількох окремих агентів/runners) для забезпечення ефективної роботи в умовах високого навантаження.
- **е) Отримання історії збірок (View Build History):** Система повинна збирати та надавати доступну історію всіх виконаних пайплайнів. Це включає дані про статус (успіх, провал, скасовано), час запуску, тривалість та тригер.
- **є) Отримання логів (View Logs):** Система повинна підтримувати ведення детальних логів (консольний вивід) для кожного кроку пайплайну, які включають інформацію про виконані команди, помилки та інші операції для діагностики і моніторингу.

- **ж) Зберігання та надання артефактів (Download Artifacts):** Система повинна дозволяти зберігати файли, згенеровані під час виконання пайплайну (наприклад, скомпільовані бінарні файли, звіти тестів), та надавати Розробнику можливість їх завантажити.
- **з) Відправка сповіщень (Send Status Notification):** Система повинна підтримувати відправку сповіщень про статус виконання пайплайну у зовнішні системи (наприклад, Email, Slack, або оновлення статусу коміту в VCS).

1.3.2. Нефункціональні вимоги до системи

Нефункціональні вимоги до системи:

- **а) Продуктивність:**
    1. Сервер повинен забезпечувати мінімальний час затримки між отриманням тригера (напр., коміт) та фактичним початком виконання завдання (час очікування в черзі).
    2. Продуктивність системи повинна підтримувати високе навантаження (обробку великої кількості одночасних пайплайнів).
- **б) Масштабованість:**
    1. Сервер повинен бути здатний **горизонтально масштабуватися** шляхом додавання нових виконавців (агентів/runners) для обробки зростаючого обсягу завдань без погіршення продуктивності.
    2. Має бути можливість розширення архітектури для підтримки росту кількості підключених проектів та розробників.
- **в) Надійність та безпека:**
    1. Сервер повинен бути надійним, з мінімальними перервами в роботі, підтримувати механізми відновлення після збоїв (наприклад, автоматичний перезапуск завдання у разі збою інфраструктури).

2. Система повинна забезпечувати ізоляцію середовищ виконання (наприклад, через контейнеризацію), щоб збірки різних проектів не впливали одна на одну та не мали доступу до чужих даних чи секретів.

- **г) Зручність у використанні:**
    1. Сервер повинен мати простий та декларативний механізм для конфігурації пайплайнів (наприклад, ci.yml файл у репозиторії).
    2. Веб-інтерфейс (якщо передбачений) має бути інтуїтивно зрозумілим для перегляду логів, історії та артефактів.
- **д) Сумісність:**
    1. Сервер повинен працювати на основних операційних системах (Linux, Windows, macOS).
    2. Він має бути сумісним з основними системами контролю версій (насамперед Git) та підтримувати виклик стандартних інструментів збірки (Maven, Gradle, npm, Docker тощо).
- **е) Безперервність роботи:**
    1. Сервер повинен бути спроектований так, щоб забезпечувати високу доступність (High Availability) та мінімізувати час простою.
    2. Важливою є підтримка механізмів розподіленого виконання та балансування навантаження між доступними виконавцями для досягнення високої відмовостійкості.

1.4. Сценарії використання системи

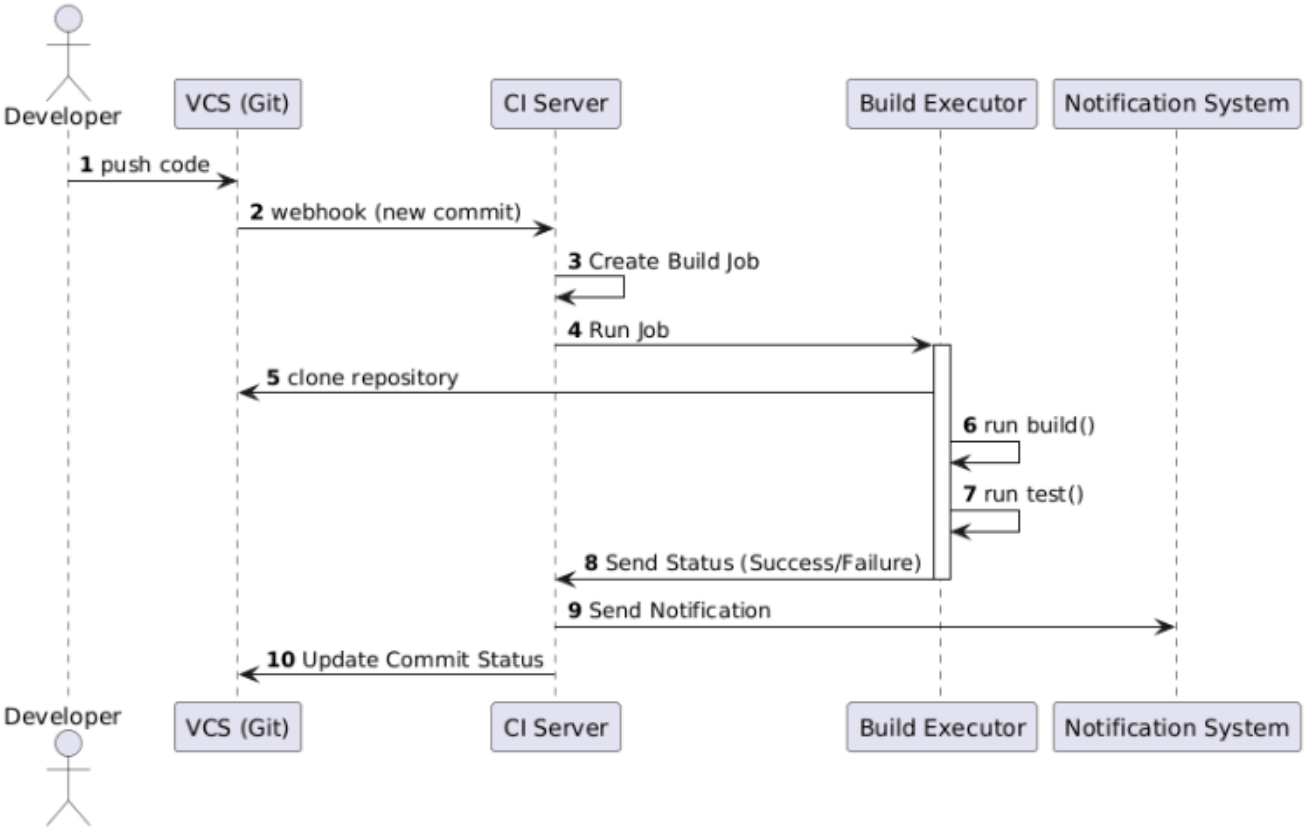

Рис. 1.4.1 – Діаграма класів

Прецедент: Автоматичний запуск збірки (Trigger Build)

Передумови:

1. CI-сервер запущений і готовий приймати запити (webhook).
2. Система контролю версій (VCS) налаштована на відправку webhook на адресу CI-сервера при події push.
3. Розробник має права на відправку (push) коду у віддалений репозиторій.

Постумови:

1. Нові зміни коду успішно завантажені у VCS.
2. CI-сервер успішно отримав webhook та створив нове завдання (Build Job) у черзі.

Взаємодіючі сторони:

1. **Розробник** — ініціює зміни, відправляє код.
2. **VCS (Git)** — отримує код, відправляє webhook.
3. **CI Server** — отримує webhook та створює завдання.

Короткий опис: Розробник надсилає зміни коду до VCS, яка автоматично повідомляє CI-сервер про подію, змушуючи сервер створити нове завдання на збірку.

Основний потік подій:

1. Розробник надсилає зміни коду (push code) до VCS.
2. VCS (Git) отримує зміни та ідентифікує, що для цього репозиторію налаштований webhook.
3. VCS надсилає webhook (HTTP POST-запит) на CI-сервер (крок 2 діаграми).
4. CI-сервер приймає запит.
5. CI-сервер аналізує webhook та створює нове завдання (Create Build Job) (крок 3 діаграми).
6. CI-сервер додає завдання у чергу на виконання.

Прецедент: Виконання пайплайну (Execute Pipeline)

Передумови:

1. CI-сервер має принаймні одне завдання у черзі.
2. Доступний хоча б один вільний Виконавець (Build Executor).
3. У репозиторії існує валідний файл конфігурації пайплайну.

Постумови:

1. Код проекту завантажено, скомпільовано та протестовано.
2. CI-сервер отримав фінальний статус (Success/Failure) та повні логи виконання.

Взаємодіючі сторони:

1. **CI Server** — керує чергою, відправляє завдання.
2. **Build Executor** — виконує реальні команди (збірка, тести).

3. **VCS (Git)** — надає доступ до коду для клонування.

Короткий опис: CI-сервер доручає Виконавцю виконати завдання з черги. Виконавець завантажує код, послідовно виконує кроки (збірка, тести) та звітує про результат.

Основний потік подій:

1. CI-сервер надсилає команду "Run Job" доступному Виконавцю (крок 4 діаграми).
2. Виконавець (Executor) приймає завдання.
3. Виконавець надсилає запит до VCS на клонування репозиторію (clone repository) (крок 5 діаграми).
4. VCS надає код.
5. Виконавець запускає процес збірки (run build()) (крок 6 діаграми).
6. Після успішної збірки, Виконавець запускає процес тестування (run test()) (крок 7 діаграми).
7. Виконавець формує звіт (логи та статус) і відправляє його на CI-сервер (Send Status) (крок 8 діаграми).

Прецедент: Надсилання сповіщень та оновлення статусу

Передумови:

1. CI-сервер отримав фінальний статус (Success/Failure) від Виконавця.
2. У проекті налаштовані інтеграції з системами сповіщень (Slack, Email) та VCS.

Постумови:

1. Сформоване та відправлене сповіщення у зовнішню систему.
2. Статус коміту оновлений у VCS.

Взаємодіючі сторони:

1. **CI Server** — формує та відправляє сповіщення і статус.
2. **Notification System** — отримує та доставляє сповіщення (напр., у Slack).

3. **VCS (Git)** — отримує та відображає статус коміту.

Короткий опис: Сервер на основі отриманого статусу збірки інформує зацікавлені сторони про результат, надсилаючи повідомлення в месенджери та оновлюючи статус у системі контролю версій.

Основний потік подій:

1. CI-сервер отримує статус (Success/Failure) від Виконавця (крок 8 діаграми).
2. Сервер обробляє статус і формує дані для сповіщення.
3. Сервер надсилає сповіщення у зовнішню Систему Сповіщень (Send Notification) (крок 9 діаграми).
4. Сервер формує запит до API системи VCS.
5. Сервер надсилає запит на оновлення статусу коміту (Update Commit Status) до VCS (крок 10 діаграми).
6. Розробник бачить статус (напр., "галочку" або "хрестик") біля свого коміту в VCS.

Прецедент: Конфігурація проекту

Передумови:

1. CI-сервер має функціональність для налаштування проектів.
2. Розробник або адміністратор має доступ до налаштувань сервера.

Постумови:

1. Проект успішно налаштований у CI-сервері.
2. Сервер готовий приймати webhook та запускати пайплайни для цього проекту.

Взаємодіючі сторони:

1. **Розробник (Адміністратор)** — налаштовує проект.
2. **CI Server** — використовує задані налаштування для обробки завдань.

Короткий опис: Розробник налаштовує новий проект у CI-сервері, вказуючи, як отримати код (URL репозиторію) та що з ним робити (конфігурація пайплайну).

Основний потік подій:

1. Розробник додає новий проект у конфігурацію сервера (через UI або додаючи .ci.yml файл у репозиторій).
2. Розробник вказує URL репозиторію.
3. Сервер зберігає конфігурацію у своїй базі даних.
4. Запит (webhook), який відповідає цьому проекту, буде оброблятися відповідним пайплайном.

Прецедент: Отримання логів та історії збірок

Передумови:

1. CI-сервер збирає та зберігає дані про оброблені завдання (логи, статуси, тривалість).
2. Розробник має доступ до веб-інтерфейсу CI-сервера.

Постумови:

1. Користувач отримує актуальну історію збірок та детальні логи виконання.

Взаємодіючі сторони:

1. **Розробник** — запитує історію або логи.
2. **CI Server** — генерує та повертає статистику/логи.

Короткий опис: Розробник надсилає запит (через браузер) на отримання історії збірок або логів конкретного завдання, які сервер збирає під час роботи.

Основний потік подій:

1. Розробник формує запит на отримання історії (відкриває веб-сторінку проекту).
2. CI-сервер отримує запит і звертається до своєї бази даних.
3. Сервер отримує дані про минулі збірки (статус, час, коміт).

4. Сервер формує відповідь (HTML-сторінку) і надсилає її клієнту (браузеру).
5. Розробник отримує історію збірок.

1.5. Концептуальна модель системи

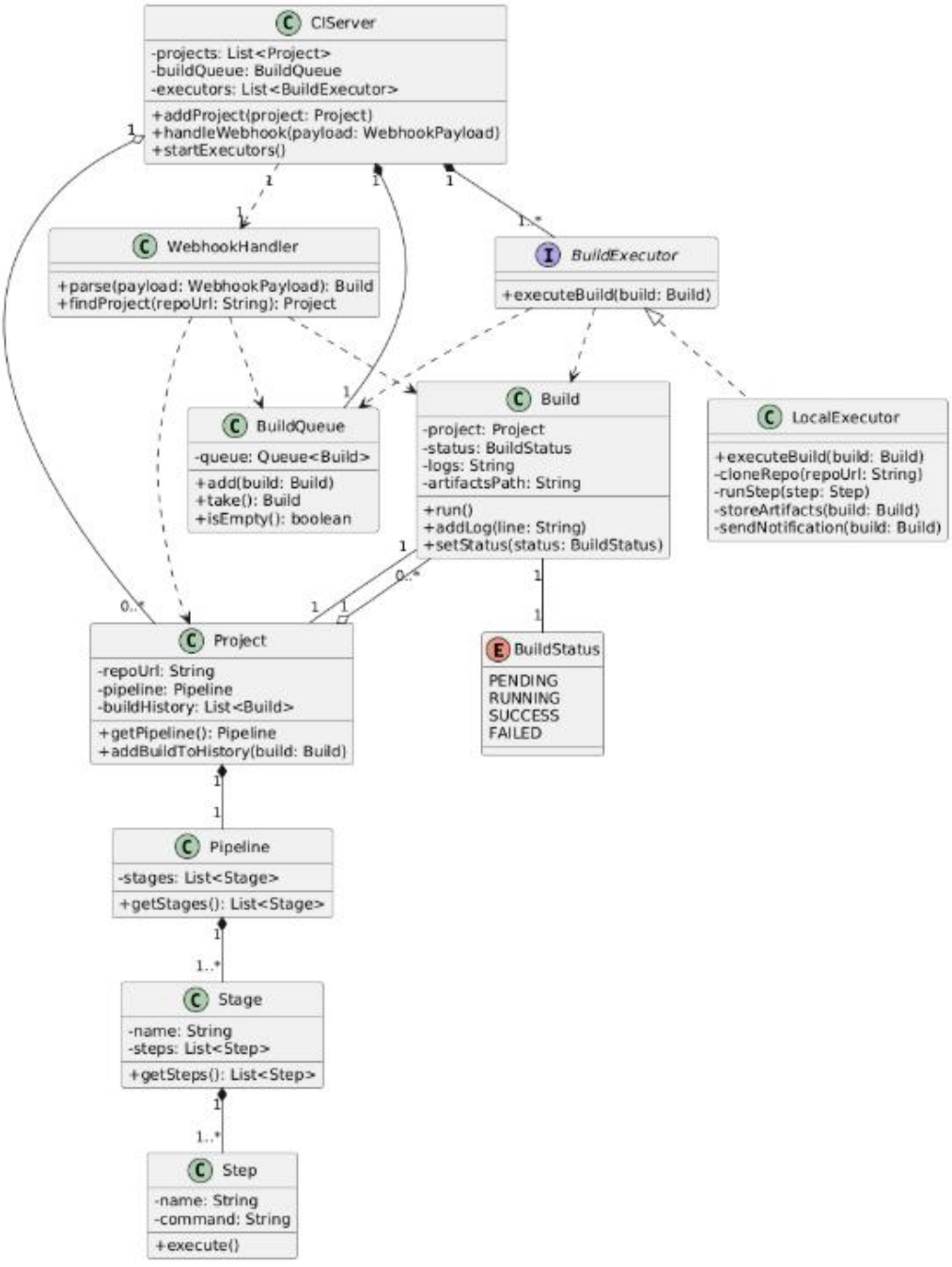

Рис. 1.5.1 – Діаграма класів

На основі діаграми класів центральним елементом системи є клас CIServer, який виконує роль координатора, керуючи списком проектів (Project), чергою збірок (BuildQueue) та пулом виконавців (BuildExecutor). Він опосередковано взаємодіє з WebhookHandler для прийому зовнішніх тригерів, які ініціюють процес збірки.

Обробка тригерів збірки відбувається за участю кількох ключових класів. Клас WebhookHandler є основним обробником, який отримує запити (payloads), виконує їхній аналіз за допомогою методу parse(), знаходить відповідний Project (за repoUrl), створює об’єкт Build для представлення вхідного завдання і додає його до BuildQueue.

Фактичне виконання збірки делеговано інтерфейсу BuildExecutor. Його конкретна реалізація, LocalExecutor, забирає завдання Build з черги та виконує його. LocalExecutor відповідає за клонування репозиторію (cloneRepo), виконання кроків (runStep), збереження артефактів (storeArtifacts) та відправку сповіщень (sendNotification). Сам об'єкт Build акумулює в собі стан збірки, зокрема logs та status (який є переліком BuildStatus: PENDING, RUNNING, SUCCESS, FAILED).

Конфігурація проекту чітко структурована. Кожен Project містить посилання на Pipeline. Клас Pipeline, у свою чергу, складається зі списку Stage (стадій, наприклад, "build", "test"), а кожна Stage містить список Step (кроків), які представляють конкретні команди (command) для виконання.

Система також має модуль історії, реалізований через клас Project, який зберігає список минулих збірок (buildHistory: List<Build>). Це дозволяє відслідковувати статуси та результати попередніх виконань для кожного окремого проекту.

Рівень виконання абстрагований інтерфейсом BuildExecutor та реалізований класом LocalExecutor. Клас BuildQueue керує чергою завдань (за принципом FIFO), реалізуючи асинхронну модель роботи та забезпечуючи обробку збірок у міру звільнення виконавців. Взаємодія між усіма цими компонентами забезпечує високу модульність і чітке розділення обов’язків у системі.

Загалом діаграма класів демонструє чіткі зв’язки між компонентами, що забезпечують модульність, розширюваність і масштабованість системи. Центральна роль CIServer як оркестратора та використання інших компонентів на основі чітких

інтерфейсів (як BuildExecutor) дозволяють легко адаптувати систему до нових вимог (наприклад, додати новий тип виконавця, як-от DockerExecutor). Завдяки розподілу обов’язків між окремими модулями, такими як обробка тригерів (WebhookHandler), конфігурація (Project/Pipeline) та виконання (LocalExecutor), сервер залишається гнучким і зручним у підтримці.

1.6. Вибір бази даних

Для зберігання та обробки даних у рамках цього проєкту було обрано систему управління базами даних **PostgreSQL**. Це рішення обґрунтоване кількома важливими факторами, серед яких висока надійність, підтримка складних типів даних та можливості для ефективного зберігання великих обсягів логів та історії збірок.

Причини вибору PostgreSQL:

- **а) Висока продуктивність:** CI-сервер генерує величезні обсяги даних, особливо текстових **логів збірок** та записів в **історії**. PostgreSQL є однією з найшвидших реляційних баз даних, що дозволяє ефективно працювати з такими великими обсягами. Завдяки потужним механізмам індексації (включаючи індекси для JSONB та повнотекстового пошуку), паралельної обробки запитів та оптимізації, PostgreSQL забезпечує високу швидкість запису (для логів) та читання (для відображення історії збірок) навіть при великих навантаженнях.
- **б) Масштабованість:** PostgreSQL має чудову підтримку горизонтальної (наприклад, через реплікацію) та вертикальної масштабованості. Це критично важливий аспект для CI-сервера, оскільки з часом кількість проектів, розробників, частота комітів, а отже, і обсяг даних (логів, артефактів) буде невпинно збільшуватися.
- **в) Реляційна модель:** PostgreSQL є об'єктно-реляційною базою даних, що дозволяє зручно моделювати дані для CI-системи. Ми маємо чіткі

зв'язки: Project має багато Builds, Build складається з Stages, Stage має Steps. Це дозволяє зберігати інформацію про конфігурації пайплайнів, історію запусків та зв'язки між ними. Використання SQL дає змогу створювати складні запити (наприклад, "показати середній час збірки для гілки 'main' за останній місяць").

- **г) Підтримка транзакцій:** PostgreSQL повністю підтримує **ACID** (Atomicity, Consistency, Isolation, Durability), що гарантує цілісність даних. Це особливо важливо для CI-сервера, де треба забезпечити атомарність операцій: наприклад, коли збірка починається, її статус в базі даних (Build.status) та створення запису для логів (BuildLog) мають відбутися в рамках однієї транзакції.
- **д) Підтримка складних типів даних:** Це одна з ключових переваг PostgreSQL для нашої теми. Він підтримує:
    - **JSONB**: Ідеально підходить для зберігання гнучких даних, таких як **конфігурація пайплайну** (з .yml файлу), параметри запуску, або структуровані метадані артефактів.
    - **TEXT**: Необхідний для зберігання **великих обсягів логів** збірки.
    - **TIMESTAMP**: Критично важливий для ведення точної історії (created_at, started_at, finished_at для кожної збірки та етапу).
- **е) Підтримка розширень:** PostgreSQL дозволяє використовувати численні розширення. Для CI-сервера особливу цінність становить вбудована підтримка **повнотекстового пошуку (FTS)**. Це дозволяє реалізувати потужну функціональність пошуку конкретних помилок або повідомлень у мільйонах рядків логів, що є критичним для діагностики проблем розробниками.
- **є) Відкритий код і спільнота:** PostgreSQL є системою з відкритим вихідним кодом, що дозволяє уникнути "прив'язки" до постачальника (vendor lock-in). Крім того, PostgreSQL має велику і активну спільноту,

що забезпечує безперервний розвиток, чудову документацію та підтримку системи.

- **ж) Безпека:** CI-сервер зберігає надзвичайно чутливі дані: **секрети та ключі** (API-токени, паролі до баз даних, SSH-ключі), доступ до репозиторіїв та бінарні артефакти. PostgreSQL надає широкі можливості для забезпечення безпеки даних, включаючи аутентифікацію, SSL-з'єднання для шифрування даних, а також гранулярне управління доступом на рівні користувачів, ролей та навіть окремих рядків (Row-Level Security).

1.7. Вибір мови програмування та середовища розробки

Для розробки програмного комплексу CI-сервера було обрано мову програмування Java у поєднанні з інтегрованим середовищем розробки (IDE) IntelliJ IDEA. Цей вибір ґрунтується на вимогах до надійності, паралелізму та складності, які притаманні системам безперервної інтеграції.

Обґрунтування вибору Java

Обрана мова програмування визначає фундамент для всієї архітектури. Java є стратегічним вибором для цього проекту з кількох ключових причин:

1. **Зрілість екосистеми для серверних завдань.** CI-сервер — це критична, довготривала інфраструктурна служба. Java має одну з найпотужніших екосистем для створення таких систем. Це надає доступ до перевірених часом бібліотек для вирішення основних завдань нашого сервера:
    - **Робота з Git:** Бібліотеки, як-от JGit, дозволяють реалізувати логіку клонування репозиторіїв безпосередньо в коді BuildExecutor.
    - **Обробка конфігурацій:** Парсери для YAML (наприклад, SnakeYAML) необхідні для читання файлів конфігурації пайплайнів.

- **Мережева взаємодія:** Надійні HTTP-клієнти потрібні для відправки сповіщень та оновлення статусів у VCS.

2. **Потужна вбудована підтримка паралелізму.** Ядро CI-сервера — це його здатність виконувати **кілька збірок одночасно**. Модель багатопоточності Java та, зокрема, пакет java.util.concurrent, надають високорівневі інструменти, які ідеально лягають на нашу архітектуру. Такі класи, як ExecutorService та BlockingQueue, дозволяють елегантно реалізувати BuildQueue (чергу завдань) та пул BuildExecutor (виконавців), забезпечуючи ефективне управління ресурсами та уникнення "гонок" (race conditions).
3. **Платформонезалежність (для сервера та агентів).** Принцип Java "Write Once, Run Anywhere" (WORA) є критично важливим для CI-системи. Сам CIServer (сервер-координатор), скоріш за все, буде розгорнутий на Linux. Однак **агенти** (BuildExecutor) повинні мати можливість запускатися на будь-якій ОС (Windows, macOS, Linux) для того, щоб збирати та тестувати програмне забезпечення для цих конкретних платформ. Java дозволяє створити єдину кодову базу для агентів, яка працюватиме скрізь, де встановлена JVM.
4. **Продуктивність JVM для довготривалих процесів.** Віртуальна машина Java (JVM) оптимізована для серверних додатків, що працюють 24/7. JIT-компілятор (Just-In-Time) та сучасні збирачі сміття (Garbage Collectors) забезпечують стабільно високу продуктивність та ефективне управління пам'яттю, що необхідно для сервера, який постійно обробляє нові завдання.

Обґрунтування вибору IntelliJ IDEA

Ефективність розробки складного програмного забезпечення, як-от CI-сервер, напряму залежить від якості інструментів. IntelliJ IDEA була обрана як основне середовище розробки, оскільки вона надає унікальні переваги для роботи з Java-проектами такої складності.

- **Глибоке розуміння коду та рефакторинг.** Архітектура нашого проекту (як видно з діаграми класів) є модульною та взаємопов'язаною. IntelliJ IDEA має

найкращі в індустрії інструменти для **рефакторингу**. Це дозволяє безпечно змінювати структуру проекту — наприклад, перейменовувати методи в інтерфейсі BuildExecutor або переміщувати класи між модулями — з упевненістю, що IDE оновить усі відповідні посилання.

- **Розширене налагодження багатопотоковості.** Найскладніші помилки в CI-сервері будуть пов'язані з паралелізмом (наприклад, взаємні блокування у BuildQueue або неправильний стан Build). IntelliJ IDEA надає потужний **візуальний налагоджувач (debugger)**, який дозволяє інспектувати стан усіх активних потоків (threads), що є абсолютно незамінним для діагностики та виправлення таких проблем.
- **Інтегрований набір інструментів.** IDEA об'єднує всі необхідні для розробника CI-сервера інструменти в єдиному вікні. Вона включає:
    - **Найкращу інтеграцію з Git** (для роботи з кодом самого CI-сервера).
    - **Вбудований клієнт баз даних** (для прямого підключення до PostgreSQL та перевірки історії збірок).
    - **Підтримку систем збірки** Maven та Gradle (для збірки самого проекту CI-сервера).
    - **Інтеграцію з Docker** (що може бути використано як для запуску БД, так і для тестування агентів у контейнерах).

Ця комбінація (зріла, багатопотокова мова та інтелектуальна IDE) створює надійну основу для успішної реалізації та подальшої підтримки складного проекту CI-сервера.

1.8. Проєктування розгортання системи

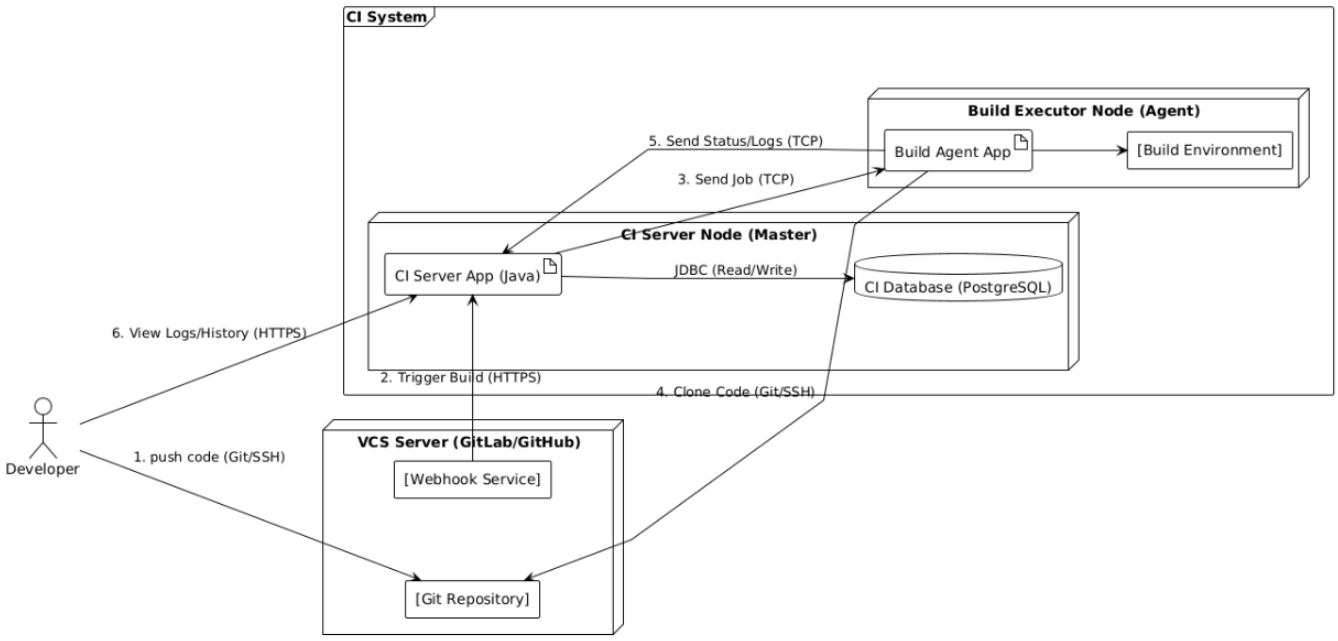

Рис. 1.8.1 – Діаграма розгортання

Проєктування розгортання системи базується на чітко визначеній архітектурі, що включає кілька взаємодіючих вузлів, кожен із яких виконує специфічні функції. Існує дві основні точки взаємодії з системою. По-перше, Розробник (Developer) взаємодіє з VCS Server (наприклад, GitLab/GitHub), відправляючи зміни коду (крок 1, push code). По-друге, Розробник також взаємодіє безпосередньо з CI Server App через веб-браузер (крок 6, View Logs/History), щоб переглядати історію, логи та артефакти збірок, отримуючи у відповідь HTML-сторінки та дані у форматі JSON.

Центральною частиною системи є CI Server Node (Master), який виконує роль основного координатора ("мозку") системи. Він містить CI Server App (Java), що обробляє всі вхідні тригери, зокрема Webhook від VCS Server (крок 2, Trigger Build). Отримавши тригер, CI Server App формує чергу завдань та делегує їхнє виконання, відправляючи команду (Send Job) на доступний Build Executor Node (крок 3). Він також збирає результати (логи та статуси) від виконавців (крок 5, Send Status/Logs) і

забезпечує їх збереження. Завдяки асинхронній моделі та системі черг, сервер здатний ефективно керувати численними одночасними завданнями.

Сервер баз даних, реалізований на основі CI Database (PostgreSQL), забезпечує збереження всієї необхідної інформації для роботи системи. Він знаходиться на тому ж вузлі, що й CI Server App, і взаємодіє з ним через JDBC. База даних зберігає повну історію збірок, детальні логи виконання, метадані про артефакти та конфігурації проектів. PostgreSQL обраний завдяки його надійності, масштабованості та підтримці транзакційності, що гарантує збереження цілісності даних (наприклад, атомарне оновлення статусу збірки).

Вузол Build Executor Node (Agent) виконує найважливішу роль у виконанні завдань, забезпечуючи реалізацію розподіленої моделі. Він отримує команду на запуск (Send Job) від CI Server Node (крок 3) через TCP. Отримавши завдання, Build Agent App взаємодіє безпосередньо з VCS Server, щоб клонувати необхідний код (крок 4, Clone Code). Збірка та тестування відбуваються у ізольованому Build Environment (це може бути, наприклад, Docker-контейнер). Після завершення, агент надсилає повні логи та фінальний статус назад на головний сервер (крок 5).

Загалом, запропоноване проєктування забезпечує ефективну роботу CI-сервера завдяки чіткому розділенню обов'язків. CI Server Node виступає як координатор, а Build Executor Node — як "робоча конячка". Це дозволяє досягти високої продуктивності, гнучкості та стабільності. Така архітектура забезпечує горизонтальне масштабування (для збільшення потужності достатньо додати нові Build Executor Node) та балансування навантаження між виконавцями, гарантуючи надання швидкого та надійного зворотного зв'язку розробникам.

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Структура бази даних

2.2. Архітектура системи

2.2.1. Специфікація системи

2.2.2. Вибір та обґрунтування патернів реалізації

2.3. Інструкція користувача

ВИСНОВКИ

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

ДОДАТКИ