Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

УПРАВЛІННЯ ПРОХОДЖЕННЯМ ТЕСТІВ У CI/CD ПРОЦЕСІ

Текстова частина до курсової роботи за спеціальністю „Інженерія Програмного Забезпечення” 121

Керівник курсової роботи

асистент Яремко С.А.

_____

(підпис)

“____” _____ 2022р.

Виконав студент Накитняк В.І.

“____” _____ 2022р.

Київ 2022

Тема: Управління проходженням тестів у CI/CD процесі

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	13.10.2021	
2.	Вивчення предметної області та огляд технічної літератури за темою роботи	04.12.2021	
3.	Написання вступу курсової роботи	15.04.2022	
4.	Написання теоретичної частини курсової роботи	20.04.2022	
5.	Написання застосунку мовою програмування Kotlin	5.05.2022	
6.	Тестування готового застосунку	14.05.2022	
7.	Написання висновку та фінальне коригування текстової частини роботи	15.05.2022	
8.	Надсилання роботи для перевірки на відсутність плагіату	20.05.2022	

Зміст

Вступ

Розробка програмного забезпечення є складним і довготривалим процесом. Від ефективності, часу розробки, а також якості залежить успіх проєкту. Для того, щоб полегшити життя програмістам, існують стандартизовані механізми, які допомагають управляти великими проєктами, і розбивати процеси їх розробки й впровадження на менші задачі.

Безперервна інтеграція і безперервна доставка(**CI/CD**), є найбільш розповсюдженими практиками впровадження та реалізації сучасних проєктів в життя. Вони поєднують всі етапи розробки, і усувають розрив між власне розробкою і кінцевим продуктом, який виходить на ринок. Це досягається завдяки автоматизації всіх етапів розробки, починаючи від написання коду, до розгортання застосунку на сервер і підтримки його життєдіяльності.

Одним з етапів **CI/CD** є тестування. Це один з найважливіших етапів, бо саме на ньому вирішується чи вийде проєкт у реліз. На ньому затверджується працездатність та коректність всіх компонентів в умовах симуляції максимально наближеної до реального середовища.

Постановка задачі

Серед задач, які ставить дана робота:

- Дослідити сучасні засоби управління тестуванням у розробці
- Дослідити основні платформи, які здійснюють **CI/CD**
- Дослідити основні засоби автоматизації тестування в розробці на прикладі сучасних мов програмування
- Провести структурну розробку власного рішення, з детальним описом використаних технологій і засобів
- Розробити та протестувати плагін для зручного управління автоматизованими тестами на основі бібліотеки з тегами

Основна мета

Основною метою роботи є дослідження ефективних методів управління проходженням автоматизованих тестів у процесі **CI/CD**.

1. Аналіз предметної області і наявних проблем і задач

Зазвичай, лише після проходження всіх тестів, або як їх ще називають тест-кейсів, проєкт може переходити на наступний етап розробки. Кодова база проєкту потребує постійної підтримки, бо маленька помилка через неуважність програміста в майбутньому може коштувати годинам, або навіть дням простою проєкту. Для того, щоб уникнути таких ситуацій, кожен компонент покривають тестами.

За стандартами, проєкт має бути покритий тестами як мінімум на половину. [1] Наприклад в **Google** дотримуються наступних показників тестового покриття. Вони оцінюють 60% як «прийнятне», 75% як «похвальне» та 90% як «зразкове» покриття. Але це в першу чергу стосується так званих модульних тестів, які перевіряють загальну коректність коду та зменшують до нуля можливість помилки через людський фактор. Існує навіть підхід до розробки, **TDD** (**test driven development**), який в першу чергу націлений на написання тестів, а вже потім, коду самої програми під них.

Варто пам’ятати, що у великих корпоративних рішеннях зазвичай велика к-сть коду, а кількість тестів пропорційна кількості написаного коду. Через це, перевірка всіх автоматизованих тестів може затягуватись на декілька хвилин, а в деяких ситуаціях навіть годин, якщо це складні наскрізні тести, які перевіряють роботу одночасно всіх компонентів системи.

Є декілька етапів тестування проєкту. Перший етап, це власне компіляція проєкту та його збірка. Зазвичай в цей етап якраз входять модульні тести. У випадку, якщо хоча б один з тестів провалиться, то подальша збірка скасовується. Це свідчить про наявність помилки у коді програми. Лише після того, як розробник усуне проблему і проходження тестів покаже зелене світло, збірку можна продовжувати.

Другий етап, вже після того, як ми зібрали наш проєкт. На цьому етапі ми явно викликаємо певні набори тестів, які перевіряють коректність бізнес-логіки нашого застосунку. В цей етап входять в основному інтеграційні тести

І останній етап, коли наш застосунок вже остаточно розгорнутий на тестовому середовищі. Коли ми розгортаємо наш застосунок на продакшн, цей етап зазвичай опускають, бо попередньо вже було проведено наскрізне тестування.

В залежності від кінцевого середовища, куди ми розгортаємо наш проєкт, залежать і тести, які ми запускаємо. Через це, ми іноді хочемо знехтувати деякими тестами, і надати перевагу іншим, більш важливим на даному етапі.

Наприклад, коли ми розгортаємо на тестове оточення, часто не потрібно валити збірку, якщо всього 5% відсотків тестів з тегом **@LowPriority** впали. Щоб обходити цю проблему, створюють блокувальні й деблокувальні тести. Блокувальні тести проганяються перед розгортанням. Якщо вони впали, процес не відбувається. Деблокувальні тести запускають уже після деплою.

Прикладом таких тестів є наскрізні тести, які зазвичай витрачають дуже багато часу, та залежать від багатьох чинників. Ці тести найбільш вразливі до найменших коливань у комунікаціях різних частин систему. Якщо під час прогону таких тестів трапляється випадковий збій в базі, чи будь-де, це можу коштувати годинам часу. Саме тому, іноді розробники опускають наскрізні тести.

Також це особливо актуально на етапі розробки, або тестуванні **QA/QC** командою(не плутати з етапом **CI/CD**). Коли в програмі наявна деяка кількість помилок, ми зазвичай не маємо змоги полагодити все й одразу. А це призводить до того, що багато тестів одночасно падають, і заважають продовжувати роботу (блокують). На час розробки одного багу, ми хочемо сконцентруватись тільки на ньому, тому такі непотрібні тести можна просто відключити. Очевидно, що закоментувати код не завжди ефективно. Краще підійшов би засіб, який би

дозволяв помічати певні тести тегами, і потім, в залежності від налаштувань, пропускати ці тести, чи ні.

Надалі ми розглянемо засоби, які дозволяють вирішувати цю та інші проблеми, пов’язані з виконанням тестів.

2. Теоретичні засади розробки програмного забезпечення у CI/CD процесі з допомогою автоматизованого тестування

Я поділив опис теоретичних аспектів роботи на декілька частин:

- Опис різних видів тестів та автоматизованого тестування
- Опис основних бібліотек та фреймворків на **Java**, які полегшують написання тестів
- Опис основних засобів управління проходженням тестів
- Опис практик **CI/CD** та основних платформ його впровадження

Пропоную розпочати з найнижчого рівня, а саме написання коду та покриття його тестами.

2.1. Опис автоматизованого тестування та різних видів тестів

Недостатньо просто написати програмний продукт, його також потрібно протестувати. Тестування буває ручне та автоматизоване.

Ручне тестування зазвичай виконується окремою командою контролю якості, які здійснюють перевірку роботи програми. Цей тип тестування відбувається на спеціально відведеному для цього тестовому середовищі, і його основна мета, виявити помилки в бізнес-логіці або відображенні інтерфейсу програми.

Автоматизоване тестування відбувається до того, як проєкт потрапить на тестове середовище, і виконується автоматично. [2] Для кращого розуміння автоматизованих тестів, існує «тестова піраміда».

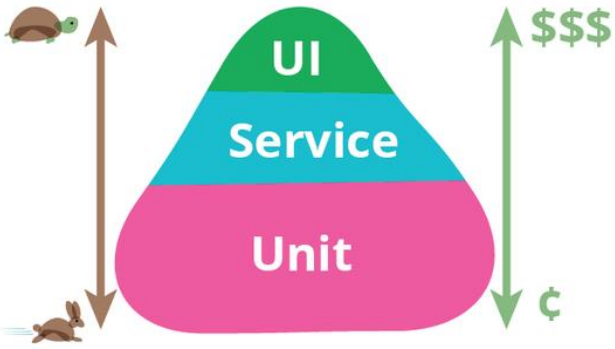

Рисунок 1 Піраміда автоматизованого тестування

Піраміда тестів – це спосіб мислення про те, як різні види автоматизованих тестів мають використовуватися для створення збалансованого покриття. Її основна мета полягає у тому, що у вас повинно бути набагато більше низькорівневих модульних тестів, ніж наскрізних тестів високого рівня. Тому тестів не завжди має бути багато, бо підтримка тестів теж вартує часу.

Окрім цього, тестування програм також поділяють на дві типи за призначенням: функціональне і нефункціональне.

2.1.1. *Функціональне тестування*

Функціональне тестування включає тестування функціональних аспектів програмного додатка. Коли ви виконуєте функціональні тести, ви повинні перевірити кожну функціональність. Вам потрібно побачити, чи отримуєте ви бажані результати чи ні.

[3] Існує кілька типів функціонального тестування:

- Модульне тестування
- Інтеграційне тестування
- Наскрізне тестування
- Димове тестування
- Санітарне тестування
- Регресійне тестування
- Приймальне тестування
- Тестування інтерфейсу

Функціональні тести виконуються як вручну, так і за допомогою засобів автоматизації. Для такого типу тестування провести ручне тестування легко, але краще використовувати інструменти для автоматизації. Детальніше зупинимось на описі деяких видів функціональних тестів.

Модульне тестування

Модульне тестування перевіряє функціонування окремих блоків коду програми. Зазвичай один модульний тест покриває один конкретний метод класу або функцію. Основна мета – ізолювати кожен блок системи для виявлення,

аналізу та усунення дефектів. Модульні тести охоплюють найбільшу частину всіх тестів. Вони знаходяться у фундаменті так званої «піраміди тестування». Написання цих тестів дуже важливе для того, щоб впевненість у правильності роботи програми, та в тому, щоб мінімізувати помилки, зроблені самим розробником через людський фактор.

Інтеграційне тестування

Інтеграційні тести перевіряють взаємодію між різними програмними компонентами. Під час написання цих тестів, інженери програмного забезпечення абстрагуються з точки зору нижнього рівня і концентруються на перспективах рівня, на якому інтегруються ці компоненти. Наприклад, це може бути перевірка інтеграції репозиторію з базою даних, або контролера із сервісом. Для цих тестів зазвичай створюють окрему базу даних. Також, для цих тестів дуже часто використовують технологію **docker**-контейнерів. Для того, щоб не засмічувати дані в основній базі, для кожного окремого тесту, або для всіх одночасно, підіймається окремий контейнер, на якому запущена потрібна база.

Наскрізне тестування

Наскрізне тестування — це функціональне тестування всієї програмної системи. Такі тести покривають весь проєкт від фронтенду до бекенду. Зазвичай, це відбувається завдяки імітації дій користувача на сайті, і перевірці правильної реакції на ці дії. Ці тести також перевіряють правильність бізнес-логіки програми, а також валідацію правильної обробки некоректного вводу користувача.

Димове тестування

Димове тестування орієнтується на поверхневу перевірку роботи важливих компонентів програми, без коректної роботи яких, подальше тестування не має сенсу. Основна ідея цього виду тестування – виявити помилки на ранньому етапі, щоб не переходити до складніших і довгих тестів. Якщо димове тестування впало, це означає, що ми можемо одразу відхилити збірку. Цим ми економимо купу часу

на те, щоб розібратись з проблемою, перед тим як тестувати наступні компоненти програми.

Регресійне тестування

Регресійне тестування виконується для того, щоб підтвердити, чи не вплинула нещодавня зміна програми або додавання нового коду на вже наявні функції.

Приймальне тестування

Приймальне тестування — це фінальний етап тестування програми перед публічним запуском. Як тільки прийнято рішення, що продукт повністю готовий до використання — приймальне тестування має підтвердити це. Приймальне тестування виконується на підставі набору типових тестових випадків і сценаріїв, розроблених на підставі вимог до даного додатка.

Санітарне тестування

Це дуже коротка поглиблена перевірка певної функціональності комп'ютерної програми, системи, вебсайту, розрахунків і так далі, яка проводиться з метою переконатися, що система працює належним чином. Цей вид тестування часто виконується до початку повного циклу регресійного тестування, але після проведення димового тестування.

Тестування інтерфейсу

Тестування інтерфейсу заточено під тестування саме передньої частини програми, з якою буде взаємодіяти кінцевий користувач. Різниця між наскрізним тестуванням і тестуванням інтерфейсу в тому, що під час тестування інтерфейсу, ми концентруємося тільки на інтерфейс програми, а не на її бізнес-логіку. Сюди також входить перевірка коректного відображення всіх компонентів та сторінок програми, і сумісність з різними версіями різних браузерів.

2.1.2. *Нефункціональне тестування*

Нефункціональне тестування — це тестування нефункціональних аспектів програми, таких як продуктивність, надійність, зручність використання, безпека тощо. Нефункціональні тести проводяться лише після всіх функціональних.

За допомогою нефункціонального тестування можна значно покращити якість програмного забезпечення. Цей вид тестування не стосується того, працює програмне забезпечення чи ні. Йдеться про те, наскільки добре працює програмне забезпечення та багато інших речей.

Нефункціональні тести, як правило, не виконуються вручну. Насправді виконати такі тести вручну важко. Тому ці тести зазвичай виконуються за допомогою сторонніх інструментів.

2.1.3. *Блокувальне тестування*

Також, тести іноді поділяють на блокувальні та на деблокувальні. Блокувальних тестів переважна більшість. Це саме ті тести, які перешкоджають наш проєкт від переходу на наступну стадію, якщо хоча б один такий тест впав. На відміну від них, деблокувальні тести не перешкоджають конвеєру **CI/CD**. Навіть якщо такі тести провалюються, дані про них заносяться в логи, але програма продовжує збірку. Це потрібно для того, щоб розрізняти тести за важливістю на різних етапах, тому ми хочемо використовувати деблокувальні тести переважно на етапі розробки чи тестуванні застосунку.

2.2. Опис основних бібліотек та фреймворків для тестування на прикладі мов з екосистеми JVM: Java, Groovy, Kotlin.

Для покриття проєкту різними типами тестів існує безліч вже готових рішень, які полегшують їх написання та автоматизацію. Зупинимось на основних тестових фреймворках для мови **Java**, **Kotlin** та **Groovy**.

2.2.1. *Selenium*

[4] Напевно найбільш відомий фреймворк для автоматизування наскрізних тестів. Це цілий набір з різних інструментів та бібліотек, які підтримують найпопулярніші мови програмування, серед яких **Java, C#, JavaScript, Python**, **Ruby**. Вони дозволяють автоматизувати браузер та вебсторінки, що значно полегшує наскрізне тестування клієнт-серверних додатків.

2.2.2. *Cucumber*

[5] Ще один доволі популярний фреймворк для наскрізного тестування. Не дивлячись на явну схожість з **Selenium**, **Cucumber** має свої переваги. **Cucumber** — інструмент автоматизації розробки типу **BDD (behavious driven development)**. Такий підхід до розробки в основному використовується, коли програма документується та одночасно розробляється на основі очікуваної поведінки при взаємодії з нею користувача. **Selenium** переважно виконує тестування інтерфейсу, а **Cucumber** — приймальне тестування. Сценарії в **Cucumber** пишуться зрозумілою мовою, тому навіть нетехнічний персонал може писати такі тести.

2.2.3. *JUnit4/JUnit5*

Найпопулярніший фреймворк для тестування програм написаних мовою програмування **Java**. Це також платформа на базі якої побудована велика к-сть інших бібліотек (напр. **Kotest**). Першочергова задача **JUnit**, як зрозуміло з назви, автоматизація модульних тестів. Але лише цим вона не обмежується, оскільки дозволяє писати також інтеграційні, димові, регресійні, наскрізні тести.

2.2.4. *TestNG*

[6] Ще один відкритий тестовий фреймворк для Java, який надихався **JUnit**. Різниця з **JUnit** у більш зрозумілих анотаціях, можливості зручно групувати тести, а також можливість паралельного тестування.

2.2.5. *Spock*

Spock — це ще одна платформа для тестування та специфікації для програм на **Java** та **Groovy**. Його основна перевага в тому, що тести в **Spock** більше

структуровані, ще в ньому є вбудована система імітації класів(англ. **mock**), а також він більш орієнтований на **BDD**.

2.3. Опис основних плагінів для управлінням проходження тестів та їх налаштування

Не менш важливим при тестуванні програми, є використання спеціальних плагінів, програмних додатків, які дозволяються автоматизувати виклики тестів на різних етапах **CI/CD**. Зупинюсь на найбільш відомих плагінах для засобу автоматизації роботи з проєктами **Maven**, який заточений під мову програмування **Java**.

2.3.1. *Maven-surefire-plugin*

[7] Плагін **Surefire** належить до набору основних плагінів **Maven**. Він використовується на етапі тестування життєвого циклу збірки для виконання модульних тестів програми. У разі будь-яких збоїв тестування збірка завершується, і жодні подальші фази не виконуються. Додатково, він створює звіти у двох різних форматах: звичайні текстові файли (.txt) та **XML** (.xml).

Серед основних переваг цього плагіну:

- Підтримка фреймворків тестування **JUnit4/5**, **NUnit**, **Spock**
- Можливість налаштувати запуск тестів у **docker**-контейнері
- Можливість паралельного запуску тестів
- Можливість пропускати всі тести певного проєкту, або лише деякі за допомогою налаштувань

Додатково наведено перелік деяких властивостей, які можна конфігурувати та настроювати:

skipTests – чи потрібно пропускати деякі тести

include/exclude – вказує які тести потрібно включати, а які ні

rerunFailingTestsCount – скільки разів потрібно перезапускати тест, якщо він падає

parallel – включає розпаралелювання тестів

2.3.2. *Maven-failsafe-plugin*

[8] Схожий на **Surefire** плагін, який заточений на виконання інтеграційних тестів. **Failsafe** запускає всі інтеграційні тести, але якщо якісь тести зазнають невдачі під час фази інтеграційного тесту, плагін не завершує збірку негайно. Натомість **Maven** продовжує наступні критично важливі етапи збірки. Тому ми все ще можемо виконувати будь-яке очищення та демонтаж середовища в рамках етапу тестування після інтеграції. На наступній фазі перевірки процесу збирання повідомляється про будь-які збої тестування.

По властивостях він ідентичний до плагіну **Surefire**, але додає 4 додаткові етапи для запуску інтеграційних тестів:

- **pre-integration-test** для налаштування середовища інтеграційного середовища (це може бути створення **docker**-контейнерів, підняття локального сервера тощо)
- **integration-test** для запуску інтеграційних тестів
- **post-integration-test** для очищення середовища тестування
- **verify** для перевірки результатів інтеграційного тестування

2.3.3. *Jacoco-maven-plugin*

[9] Цей плагін не несе ніякої зміни до процесу запуску тестів, а лише служить для того, щоб створювати звіти на основі покриття тестами проєкту. Для інтерпретації цих звітів можна скористатись сторонніми програмами, наприклад **SonarQube**. Додатково, ми можемо вказати ціль **jacoco:report** при запуску плагіну, що на виході згенерує звіти про покриття коду, які можна читати, у кількох форматах, таких як **HTML**, **CSV** та **XML**.

2.4. Опис практик CI/CD та основних платформ його впровадження

Після написання всіх тестів та конфігураційних файлів збірки проєкту, потрібно їх десь запустити. Для цього дуже важливим є реплікація максимально наближеного середовища до продакшена. В цьому на допомогу до нас приходять платформи **CI/CD**. [10] Як відомо, всього 8 основних фаз: планування,

програмування, збірка, тестування, реліз, деплой, оперування та моніторинг. Всі ці фази разом створюють нескінченний цикл розробки.

1. Фаза планування включає одну з моделей опису життєвого циклу програмного забезпечення (**SDLC**). Це може бути **Scrum**, **Agile**, **Waterfall** тощо.
2. Фаза програмування зосереджена на основних завданнях написання проєкту в інтегрованому середовищі розробки.
3. Фаза збірки запускається, коли написаний розробником код потрапляє у репозиторій у системі контролю версій. За допомогою засобу автоматизації роботи з проєктами код компілюється з усіма потрібними залежностями у готову збірку.
4. Фаза тестування зосереджена на автоматизованій перевірці коректності роботи програми. Тестування іноді включається як частина етапу збірки й зазвичай поширюється певним чином на всі фази процесу **CI/CD**, щоб забезпечити безперервний зворотний зв’язок та покращення.
5. Фаза випуску зосереджена на підготовці артефакту збірки для розгортання та документації змін.
6. Фаза розгортання — це процес доставки артефакту до середовища, де воно працюватиме як готовий застосунок.
7. Фаза оперування відбувається після того, як проєкт запущений, і складається з моніторингу та оркестрування роботи системи у режимі реального часу.
8. Етап моніторингу та оптимізації відбувається паралельно з етапом оперування і складається зі збору даних, аналізу та зворотного зв’язку до початку наступного конвеєра, якщо це необхідно.

Для того, щоб побудувати ефективний конвеєр доставки нашого продукту, існують багато різних платформ. Я зупинюсь на описі деяких з них, а саме **Jenkins** та **Github Actions**. Також розглянемо основні переваги і недоліки кожної з систем.

2.4.1. *Jenkins*

[11] Найбільш відомий сервер автоматизації з відкритою кодовою базою, яка розробляється небайдужими програмістами. Це автономна програма на основі мови програмування **Java**, тому доступна на більшості платформ(**Windows**, **macOS**, **Unix**-подібних). Завдяки підтримці спільноті, має сотні доступних

плагінів і підтримує розгортання та автоматизацію проєктів розробки програмного забезпечення.

Переваги:

- Одна з переваг використанням **Jenkins** є те, що він має відкриту кодову базу і поширюється безплатно.
- Велика кількість плагінів, які підтримуються небайдужими розробниками
- Не залежить від якогось конкретного репозиторію коду, як, наприклад **Gitlab Ci/Cd**, або **Github Actions**.

Недоліки:

- В той час, як багато плагінів підтримуються розробниками, є такі, що вже давно застарілі. Тому важливо відбирати плагіни більш менш нові й стабільні
- Занадто висока залежність від плагінів
- Підтримувати та управляти інфраструктурою **Jenkins** потрібно вручну

2.4.2. *Github Actions*

Github Actions — це керований подіями інструмент безперервної інтеграції та безперервної доставки, який дозволяє автоматизувати процес створення, тестування та розгортання ваших програм у різних середовищах. Під керованими подіями ми маємо на увазі робочий процес дій **Github**, який запускається у відповідь на подію, яка відбувається у вашому сховищі **Github**, наприклад створення **pull-request**, запит на **pull** або **push** до гілки тощо. Основним його продуктом є вебменеджер сховищ **Git** з такими функціями, як відстеження проблем, аналітика та **Wiki**. **Github** дозволяє запускати збірки, запускати тести та розгортати код з кожним фіксацією або натисканням. Ви можете створювати завдання у віртуальній машині, контейнері **Docker** або на іншому сервері. На відміну від **Jenkins, Github Actions** працює в рамках вашого репозиторію **Github**. Це можливо, тому що **Github** надає вам всю необхідну інфраструктуру для запуску, а це сервери, що належать **Gitlab**, які можуть бути серверами **Linux**, **Mac** або **Windows**. Ці сервери надають середовище для створення, тестування та

розгортання ваших програм. Файл робочого процесу **Github** написаний на мові розмітки **YAML**, на відміну від **Jenkins**, який використовує **Groovy**.

Переваги:

- Простота у використанні. **Github Action**s доволі дружній для початківців, на відміну від **Jenkins** і простий в управлінні. Саме тому він ідеально підходить для невеликих стартапів
- **Github Actions** одразу надає всю необхідну інфраструктуру для **CI/CD** операцій

Недоліки:

- Прив’язка до репозиторію **Github**
- Відносна новизна, а тому він, ще не настільки перевірений часом, як той самий **Jenkins**
- Має платну підписку для розширення функціоналу і покращення ефективності

3. Реалізація практичної частини

Завданням практичної частини роботи становило розробити плагін, який дозволятиме помічати окремі тести тегами, і в залежності від налаштувань, зупиняти збірку або ні.

3.1. Опис застосунку

Застосунок складається з двох частин: сам плагін і додаткова бібліотека тегів. Плагін може працювати поверх цієї бібліотеки тегів або окремо з тегами визначеними самим користувачем.

3.1.1. Основний функціонал

- Можливість помічати тестові класи, а також конкретні тест-кейсти тегами, з додатково створеної бібліотеки тегів, або визначеними користувачем.
- Можливість детальної конфігурації плагіну, з указанням процентного співвідношення різних тегів, або окремих тест-кейсів, як ті що проходять, або зупиняють збірку.
- Можливість динамічно додавати нові **JUnit** теги, а також їх налаштовувати у конфігурації плагіну.
- Можливість окремого запуску плагіну в командній стрічці, з указанням додаткових параметрів.

3.1.2. Особливості застосунку

- Простий в налаштуванні. Для того, щоб почати користуватись плагіном, достатньо в конфігураційному файлові збирача проєкту, в цьому випадку **Gradle**, вказати репозиторій, звідки буде стягуватись плагін та додаткова бібліотека. Потім додати плагін у відповідне поле за його назвою та версією. Після цього, він вже готовий до використання.
- Вивід детальної інформації про проходження тестів у консоль. В консоль виводиться інформація по кожному окремому тесту, його результат та

час. В кінці тестування, виводиться загальна інформація по всіх тестах, скільки тегованих тестів було провалено, пропущено або скільки завершились успішно.

- Інтеграція плагіну в наявний **Task “test”** в збирачі проєктів Gradle.

3.1.3. Стек технологій

Щодо технологій реалізації, для написання плагіну я обрав мову програмування **Kotlin**. Тестовий проєкт і додаткова бібліотека тегів написані на **Java**. В якості збирача проєктів було обрано **Gradle**, бо саме на нього і орієнтований кінцевий плагін.

3.2. Обгрунтування використаних інструментів

Для реалізації плагіну для **Gradle** підійде будь-яка мова, яка може бути скомпільована у байт-код **JVM**. Серед кандидатів були **Java**, **Groovy** і **Kotlin**. Я обрав **Kotlin**, бо він новіший і сучасніший за **Java**, і на відміну від **Groovy** статично типізований. Для написання конфігураційного файлу збірки я теж використовує **Kotlin** (класичний варіант це **Groovy** скрипт).

Було вирішено розробити плагін для **Gradle**. **Gradle** - це інструмент автоматизації створення багатомовного програмного забезпечення. Всього на платформі **JVM** існує 3 основних збирачі проєктів: **Ant**, **Maven** та **Gradle**. **Ant** вже давно застарілий і майже ніде не використовується. Щодо **Maven** і **Gradle**, я обрав саме **Gradle** через його значні переваги й те, що він дозволяє кастомізувати збірку не тільки під мову програмування **Java**, але й інші.

Для кастомізації **Gradle** є два способи. Написати плагін, або ж можна дописати потрібну логіку напряму в скрипт конфігурації. Maven не володіє такою гнучкістю, бо для його кастомізації підходять тільки плагіни.

3.2.1. Gradle Vs Maven

Feature	**Maven**	**Gradle**
Підхід	Декларативний	Імперативний

Підтримка проєктів	Переважно Java	Java, Kotlin, Groovy та будь-яка інша мова програмування
Написання плагінів	+	+
Швидкість збірки	Стандартна швидкість	На 40% швидше за Maven [12]
Інкрементальна збірка	-	+
Паралельна збірка	+	+ (Завдання можуть виконуватись паралельно)
Скриптова мова	XML розмітка	Groovy, Kotlin
Можливість кастомізації	Можлива тільки за умови написання нових плагінів	Можна напряму додавати нові завдання у скрипт, або виносити їх в окремі плагіни

3.2.2. JUnit VS TestNG

Плагін буде працювати поверх вже наявної бібліотеки тегів **JUnit**, я її розширив власними кастомними тегами і додав їх в окрему бібліотеку. Я обрав **JUnit** а не **TestNG**, через наступні причини.

JUnit і **TestNG**, безперечно, є двома найбільш популярними фреймворками для модульного тестування в екосистемі **Java**. Я обрав **JUnit**, а саме **JUnit5**, через наявність вбудованої анотації **@Tag**. Це дозволяє помічати тести як **@Tag(“smoke”)**, а потім за потреби запускати тестування тільки для тестів помічених як **smoke**. В **TestNG** є схожий функціонал у вигляді груп тестів, проте це не зовсім те, що мені потрібно, бо на його основі не дуже зручно писати кастомні теги. У випадку з **@Tag**, можна створити нову анотацію з назвою **@Smoke**, і вже до цієї анотації додати анотацію **@Tag(“smoke”)** з потрібною назвою. Після цього лишається дописати до тесту створену анотацію і все.

3.3. Опис детальної розробки плагіну для Gradle

3.3.1. Загальний підхід до написання плагінів для Gradle

Спочатку варто зупинитись на тому, що являють собою плагіни у **Gradle**. **Gradle**-плагіни - це контейнери, які можуть містити в собі логіку, що настроюється, і різні завдання для сценаріїв складання проєкту. [13] Плагіни повністю автономні та дозволяють зберігати логіку для її повторного використання у кількох проєктах або **Gradle**-модулях. Вони чудово підходять для будь-яких завдань, що вимагають роботу з вихідним кодом. Такими завданнями можуть бути генерація коду або документації, перевірка коду, запуск задач на **CI/CD** та багато іншого.

З точки зору коду, для того, щоб написати плагін, достатньо реалізувати інтерфейс **Plugin** з єдиним методом **apply**:

```
class TestsManagerPlugin : Plugin<Project> {

    companion object {
        val PLUGIN_ACTIONS = listOf(JavaPluginAction)
    }

    override fun apply(project: Project) {
```

*Рисунок 2 Приклад імплементації інтерфейсу **Plugin***

Для передачі параметрів в плагін **Gradle** використовуються **extension**-контейнери, які, по своїй суті, нічим не відрізняються від звичайних класів. У нашому випадку такий контейнер виглядає наступним чином:

```
interface TestsManagerExtension {

    var enabled: Boolean
    var verbose: Boolean
    var parallel: Boolean
    var projectName: String
    var framework: FrameworkMode
    val logging: LoggingExtension

    fun logging(action: Action<in LoggingExtension>)

    fun tagsConfigs(): Map<String?, String?>?

    fun tag(tagConfig: String?)

    fun tag(tagName: String?, tagConfig: String?)
```

*Рисунок 3 Приклад **extension**-контейнера з проєкту*

Контейнер повинен бути абстрактним або відкритим для наслідування класом, оскільки **Gradle** створюватиме його екземпляр за допомогою рефлексії.

Створити та зареєструвати контейнер можна наступним чином:

```
// Register extension DSL
val testReport = extensions.create(
    TestsManagerExtension::class.java,
    TestsManagerExtension.NAME,
    TestsManagerExtensionImpl::class.java,
    project
)
```

*Рисунок 4 Приклад реєстрації **extension**-контейнера з проєкту*

Основна мета плагіну, це додавання нового функціоналу або зміна старого. Для цього в Gradle існують **Task** та **Action**. Створюється відповідний клас, який реалізує інтерфейс **PluginAction** з єдиним методом **execute**. В цей метод додається потрібна логіка:

```
object JavaPluginAction : PluginAction {

    override fun execute(project: Project) {

        val extension = project.extensions.getByType(TestsManagerExtension::class.java)

        project.run { this: Project
            tasks { this: TaskContainerScope
                named<Test>( name: "test") { this: Test
                    ignoreFailures = true
                    val manager = TestsManager( test: this, extension, project)
                    doFirst { this: Task
                        manager.apply()
                    }
                }
            }
        }
    }

    override fun getPluginClass(): KClass<out Plugin<out Project>> = JavaPlugin::class
}
```

*Рисунок 5 Приклад імплементації інтерфейсу **PluginAction** з проєкту*

В такому випадку, модифікується **Task** “**test**”, який відповідає за запуск і перевірку тестів. Звідси **Task** можна конфігурувати та додавати нові виклики. Наприклад викликом **doFirst**, кожен раз перед початком роботи **Task** буде спрацьовувати певна логіка, а виклик **doLast** додасть ту саму логіку до кінця **Task**. В прикладі ми поміщаємо застосування основного класу плагіну, а саме **TestsManeger** у лямбду **doFirst**, для того, щоб налаштувати всі потрібні хуки пере початком тестування:

```
internal class TestsManager(
    private val test: Test,
    private val config: TestsManagerExtension,
    private val project: Project
) {

    private val out = FormattedOutput(config.logging, test.logger)
    private val logBuffer: MutableMap<String, MutableList<String>> = ConcurrentHashMap()
    private val classLoader: URLClassLoader = getClassLoader(project)

    fun apply() {
        with(test) { this: Test
            if (config.enabled) {
                registerTestReport()
            }
        }
    }
```

Рисунок 6 Основний клас, який відповідає за логіку плагіну

Хуки в цьому випадку відіграють роль заглушок, або методів, які викликаються на певних стадіях тестування. Основна робота плагіну якраз і базується на перевизначенні хуків, таких як **beforeSuit**(викликається перед набором тестів), **afterSuite**(викликається в кінці тестування набору тестів), **beforeTest**(викликається кожен раз перед викликом окремого тестового методу), **afterTest**(викликається після закінчення окремого тестового методу). В методі **registerTestReport** відбувається налаштування цих хуків:

```
private fun Test.registerTestReport() {
    testLogging {  this: TestLoggingContainer
        logging.captureStandardOutput(LogLevel.LIFECYCLE)
        exceptionFormat = config.logging.exceptionFormat
    }
    // technology specific
    applyFrameworkSpecificSettings(config.framework)
    // events
    if (!config.parallel) {
        beforeSuiteReport()
        beforeTestReport()
        afterTestReport()
        afterSuiteReport()
    } else {
        afterParallelTestReport()
    }
```

Рисунок 7 Налаштування хуків автоматизованого тестування

Розглянемо виклик хуку на прикладі **beforeTest**:

```
private fun Test.beforeTestReport() {
    beforeTest(KotlinClosure1<DecoratingTestDescriptor, Unit>({  this: DecoratingTestDescriptor
        activeTestClasses.getOrPut(className.toString()) { AnnotatedTestClass.create(classLoader.loadClass(className)) }
        val name = if (displayName.endsWith( suffix: "()")) displayName.substringBefore( delimiter: "()") else displayName
        out.display( str: ":sand_watch: $name",  indentLevel: 5, LogLevel.LIFECYCLE)
    }))
}
```

*Рисунок 8 Приклад логіки з хуку **beforeTest***

В тіло хуку передається **DecoratingTestDescriptor**, що є контейнером інформації про поточний тест. Цей контейнер використовується для виведення інформації про запуск поточного тесту в консоль, а також для завантаження тестового класу за допомогою завантажувача класів.

3.3.2. Основна логіка програми

Для реалізації поставленої задачі, а саме управління проходженням тестів, найкраще підходить рефлексивне програмування. Для того, щоб динамічно вирішувати які тести та якими анотаціями анотовані викликаються, я скористався перевагами **Java Reflection API**.

[14] **Java Reflection API** – це основна бібліотека мови програмування **Java**, яка також доступна на **Kotlin**. Вона дозволяє в ході роботи програми змінювати сигнатуру методів, отримувати інформацію про методи класу, його анотації тощо. Рефлексія дозволяє нам перевіряти та змінювати атрибути класів, інтерфейсів, полів і методів під час виконання. Це стає в пригоді, коли ми не знаємо їхніх імен під час компіляції. Крім того, ми можемо створювати екземпляри нових об’єктів, викликати методи та отримувати або встановлювати значення полів за допомогою рефлексії.

[15] Основним інструментом для цього є клас **ClassLoader**. **ClassLoader** відповідає за динамічне завантаження класів **Java** у **JVM** під час виконання. Кожен клас в **Java** завантажується за допомогою **ClassLoader**, і в кожного класу є метод **getClassLoader()**, який дозволяє отримати екземпляр завантажувача, яким був завантажений конкретний клас.

В нашому випадку, потрібен завантажувач класів, який буде вантажити класи з директорії **target** клієнтського проєкту(тобто того, до якого ми прикріпимо плагін), а також класи всіх бібліотек і залежностей. Для цього створимо власний екземпляр **URLClassLoader**. Нам потрібно отримати шляхи всіх потрібних директорій. Цього можна досягти за допомогою **API** самого **Gradle**:

```
private fun getClassLoader(project: Project): URLClassLoader {
    val dirTest = File( pathname: project.buildDir.toString() + "/classes/java/test")
    val dirMain = File( pathname: project.buildDir.toString() + "/classes/java/main")
    val compileClasspathUrls = project.configurations.getByName( name: "compileClasspath").resolve().map { it: File!
        it.toURI().toURL()
    }
    val runtimeClasspathUrls = project.configurations.getByName( name: "runtimeClasspath").resolve().map { it: File!
        it.toURI().toURL()
    }
    val testCompileClasspathUrls = project.configurations.getByName( name: "testCompileClasspath").resolve().map { it: File!
        it.toURI().toURL()
    }
    val testRuntimeClasspathUrls = project.configurations.getByName( name: "testRuntimeClasspath").resolve().map { it: File!
        it.toURI().toURL()
    }
    val urls = HashSet<URL>()
    urls.addAll(compileClasspathUrls)
    urls.addAll(runtimeClasspathUrls)
    urls.addAll(testCompileClasspathUrls)
    urls.addAll(testRuntimeClasspathUrls)
    urls.add(dirTest.toURI().toURL())
    urls.add(dirMain.toURI().toURL())
    return URLClassLoader.newInstance(urls.toArray(arrayOf()))
}
```

Рисунок 9 Метод з проєкту для створення ***URLClassLoader***

Клас **Project** є своєрідним фасадом, яким містить всю важливу інформацію про проєкт, до якого прикріплено плагін. Він містить інформацію про всі шляхи, в цьому випадку нам потрібні шляхи **buildDir**, для основного коду і для тестів, а також посилання на всі **Classpath**, з якими взаємодіє проєкт. Створюємо екземпляр **URLClassLoader** за допомогою статичного методу і передаємо туди масив згенерованих посилань.

Я використовую **URLClassLoader** для того, щоб завантажувати тестовий клас перед початком конкретного тесту:

```
private fun Test.afterTestReport() {
    afterTest(KotlinClosure2<DecoratingTestDescriptor, TestResult, Unit>({ desc, result ->
        val name = with(desc) {  this: DecoratingTestDescriptor
            if (displayName.endsWith( suffix: "()")) displayName.substringBefore( delimiter: "()") else displayName
        }

        val testClass = activeTestClasses.getOrPut(desc.className.toString()) {
            AnnotatedTestClass.create(
                classLoader.loadClass(desc.className)
            )
        }

        val testMethod = testClass.createAnnotatedMethod(name)

        incrementTestCount(result)
        testMethod.getTags().forEach {  it: TestAnnotationTagWrapper
            tagIncrement(it.tagValue(), result)
        }

        val testResult = evalTestResult(result)
        val elapsed = evalElapsed(result)

        out.display( str: "$testResult $name $elapsed",  indentLevel: 5)
        logFromBufferIfRequired(desc, result)
    }))
}
```

Рисунок 10 Логіка хуку, який спрацьовує після кожного окремого тесту

З цього завантаженого класу, після завершення тесту, ми отримуємо інформацію про викликаний метод, та анотації цього методу. Відповідно до наявних анотацій, заносимо цю інформацію у геш-таблицю. Пізніше ця інформація знадобиться для того, щоб на основі заданих користувачем конфігурацій, та інформації про проходження тестів помічених різними анотаціями, можна було прийняти рішення чи збігається процентне співвідношення пройдених тестів до заданих конфігурацій.

Щодо самих анотацій, то я виніс їх в окрему бібліотеку. Вона включає найбільш поширені тестові теги:

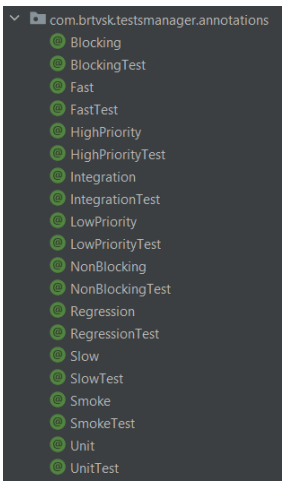

Рисунок 11 Теги з кастомної бібліотеки тегів

Різниця у тегів з закінченням **-Test**, в тому, що вони додатково проанотовані **JUnit** анотацією **@Test**.

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("highPriority")
public @interface HighPriority {
}
```

*Рисунок 12 Приклад анотації з тегом **highPriority***

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HighPriority
@Test
public @interface HighPriorityTest {
}
```

*Рисунок 13 Приклад анотації з тегом **highPriority** і **@Test***

В клієнтському проєкті проанотований тест виглядає наступним чином:

```
@HighPriorityTest
void shouldCreateTodo() {
    Mockito.when(repository.save(any(Todo.class))).thenReturn(expectedTodo);

    TodoResponse createdTodoDto = service.create(todoCreationDto, USER);

    assertThat(createdTodoDto.getId()).isEqualTo(expectedTodoDto.getId());
    assertThat(createdTodoDto.getTitle()).isEqualTo(expectedTodoDto.getTitle());
    assertThat(createdTodoDto.getDescription()).isEqualTo(expectedTodoDto.getDescription());
    assertThat(createdTodoDto.getTags()).containsAll(expectedTodoDto.getTags());
    assertThat(createdTodoDto.getCreationTime()).isEqualTo(expectedTodoDto.getCreationTime());
}
```

*Рисунок 14 Приклад тесту проанотованого тегом **highPriority***

3.4. Публікація плагіну

Для того, щоб можна було скористатись плагіном в середовищі розробки, існує декілька способів. Перший спосіб, підходить лише для локального

тестування і налагодження. Це є компіляція та запуск плагіну як сторонній проєкт до основного.

[16] Другий підхід - публікація плагіну в локальний **Maven**-репозиторій. Це дозволяє використовувати плагін всім проєктам на машині, проте цього недостатньо, якщо ми хочем розгорнути наш проєкт на сервер.

Для цього існують приватні та публічні репозиторії. Найбільш відомим в колі **Java** є офіційний **Maven**-репозиторій. Для того, щоб залити туди бібліотеку чи плагін, потрібно пройти серйозну перевірку. Також, для плагінів **Gradle** окремо існує **Gradle Registry**, але умови публікації там теж доволі суворі. Єдиним можливим варіантом для мене лишається скористатись приватним репозитарієм.

Приватні репозиторії в основному платні. Є цілі платформи які надають хмарну інфраструктуру, наприклад **JFrog**. Проте зазвичай ціна за автоматичне налаштування і підтримку такої платформи занадто висока. Розглянемо більш бюджетні варіанти.

Є можливість безплатно викласти бібліотеку на **Github Registry**, але це дає можливість використовувати її лише у власних репозиторіях без публічного доступу. В принципі цей підхід працює, але є одне але. Протестувавши його, було виявлено, що він не підтримує плагіни **Gradle**. Я зміг без проблем викласти туди бібліотеку тегів, проте мій проєкт відмовлявся бачити викладений там плагін.

Альтернативним рішенням стало створення власного репозитарію на основі хмарної платформи **GCP**. [17] Це можливо завдяки відкритому рішенню **appengine-maven-repository**. Це вже готовий проєкт **Maven**-репозиторію з аутентифікацією, який потрібно лише розгорнути на віртуальну машину **GCP**. Ось як виглядає готовий репозиторій з публічним доступом:

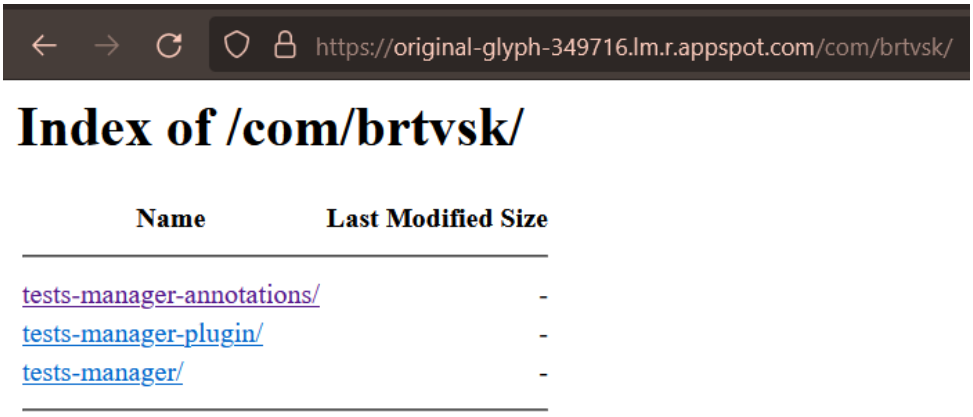

*Рисунок 15 Сторінка приватного **Maven**-репозиторію, розгорнутого на **GCP***

Для того, щоб опублікувати туди плагін та бібліотеку достатньо прописати наступні налаштування у скриптовому файлі **Gradle**:

```
// Publish to my private repo
publishing {
    repositories {
        repositories {
            maven {
                name = "MyPrivateGCPRepo"
                url = uri("https://original-glyph-349716.lm.r.appspot.com/")
                credentials {
                    username = project.property("gcp.repo.username").toString()
                    password = project.property("gcp.repo.password").toString()
                }
            }
        }
    }
    publications {
        create<MavenPublication>("maven") {
            groupId = project.group.toString()
            artifactId = "tests-manager"
            version = project.version.toString()

            from(components["java"])
        }
    }
}
```

*Рисунок 16 Приклад конфігурації для публікації плагіну в **Maven**-репозиторій*

Для того, щоб скористатись вже викладеним плагіном, в цільовому проєкті потрібно прописати наступне налаштування у файлі **settings.gradle**:

```
pluginManagement {
    repositories {
        maven {
            name = "MyPrivateGCPRepo"
            url = uri("https://original-glyph-349716.lm.r.appspot.com/")
        }
        gradlePluginPortal()
    }
}
```

*Рисунок 17 Приклад налаштування для використання приватного **Maven**-репозиторію*

3.5. Тестування застосунку

Для тестування застосунку, я створив невеликий вебпроєкт на фреймворку **SpringBoot**, покрив його тестами й інтегрував у **CI/CD** пайплайн **Github Actions**. Як платформу для **CI/CD** я вирішив обрати саме **Github Actions**, через його простоту у використанні і пряму інтеграцію з кодовою базою **Github**.

З додаткових особливостей, до пайплайну також додано кроки для статичного аналізу коду і кешування залежностей збірки. Тестування проєкту було винесено в окремі кроки, для модульного і інтеграційного тестування відповідно (**ci.yml** Див. Додаток А.).

Ось як виглядає пайплайн **GitHub Actions** у дії:

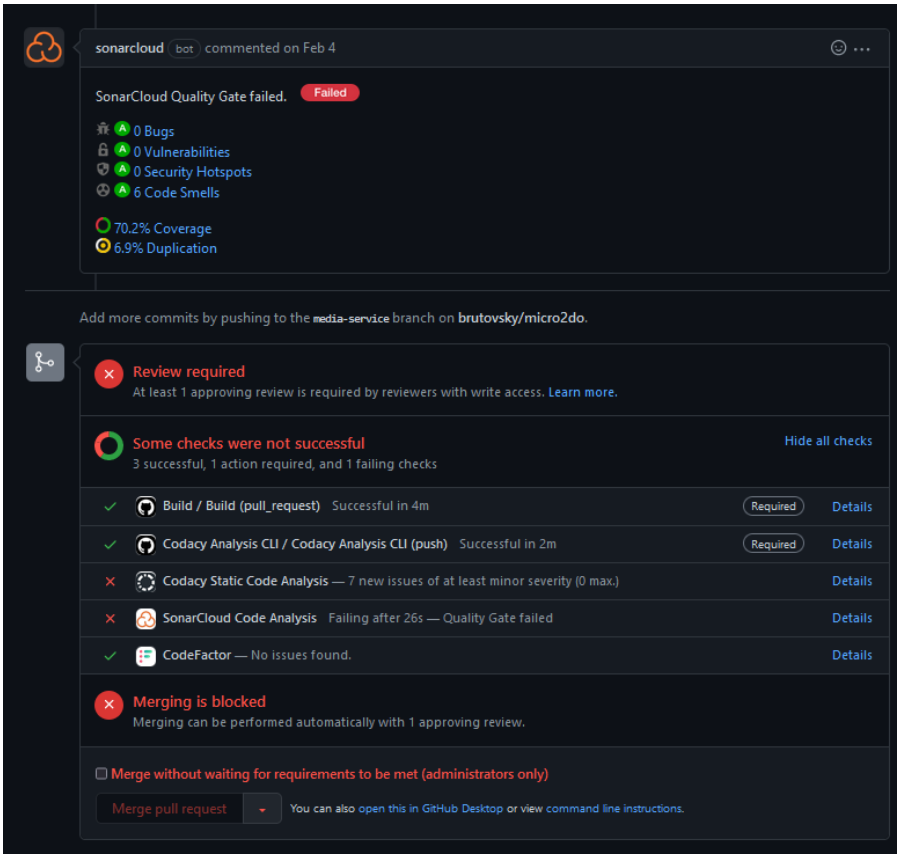

*Рисунок 18 Приклад успішних/неуспішних етапів **CI/CD** з **Github Actions***

Перейдемо до плагіну. Перш за все потрібно його підключити до проєкту та налаштувати. Ось як виглядає конфігурація плагіну у **build.gradle** файлі:

```
testsManagerConfig {  this: TestsManagerExtension
    enabled = true
    tag( tagConfig: "100%")
    tag( tagName: "integration",  tagConfig: "80%")
    tag( tagName: "slow",  tagConfig: "90%")
    tag( tagName: "lowPriority",  tagConfig: "80")
}
```

Рисунок 19 Приклад конфігурації створеного плагіну

Всі тести без анотацій за замовчуванням помічаються як **default. tag(“100%”)** – конфігурація для **default** тестів. Процент означає мінімальну кількість тестів для успішного проходження тестування. Відповідно для **lowPriority** тестів виставлено ліміт у 80% успішно пройдених тестів.

Для запуску тестів достатньо викликати команду **“./gradlew test**”. Ось як виглядає успішне застосування плагіну в дії:

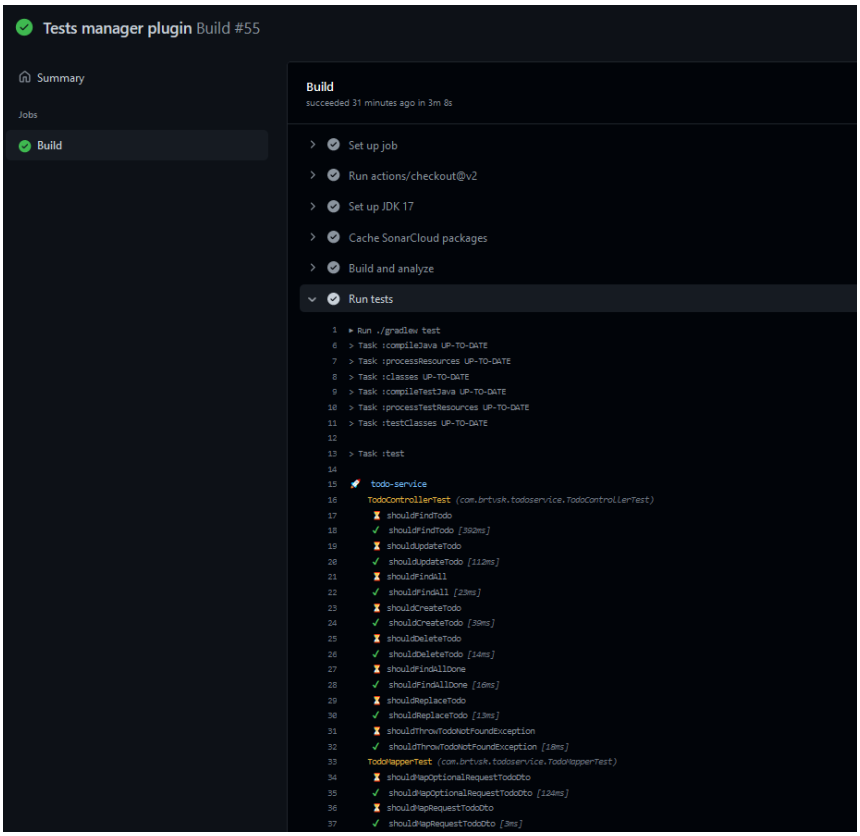

*Рисунок 20 Приклад успішно зібраного проєкту на **Github Actions***

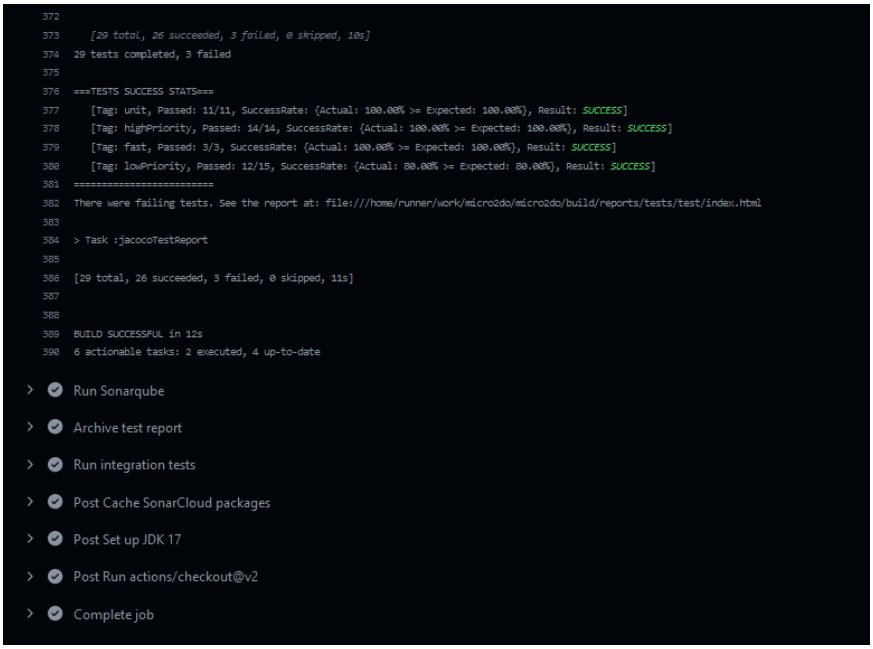

Рисунок 21 Приклад успішного тестування програми з допомогою створеного плагіну

З зображення видно, що лише 12 з 15 тестів помічених як **lowPriority** пройшли, проте збірка не впала. Це через те, що очікуваний тестовий рейтинг проходження збігається зі сконфігурованим. У випадку, якщо впаде на один тест більше, виникне помилка:

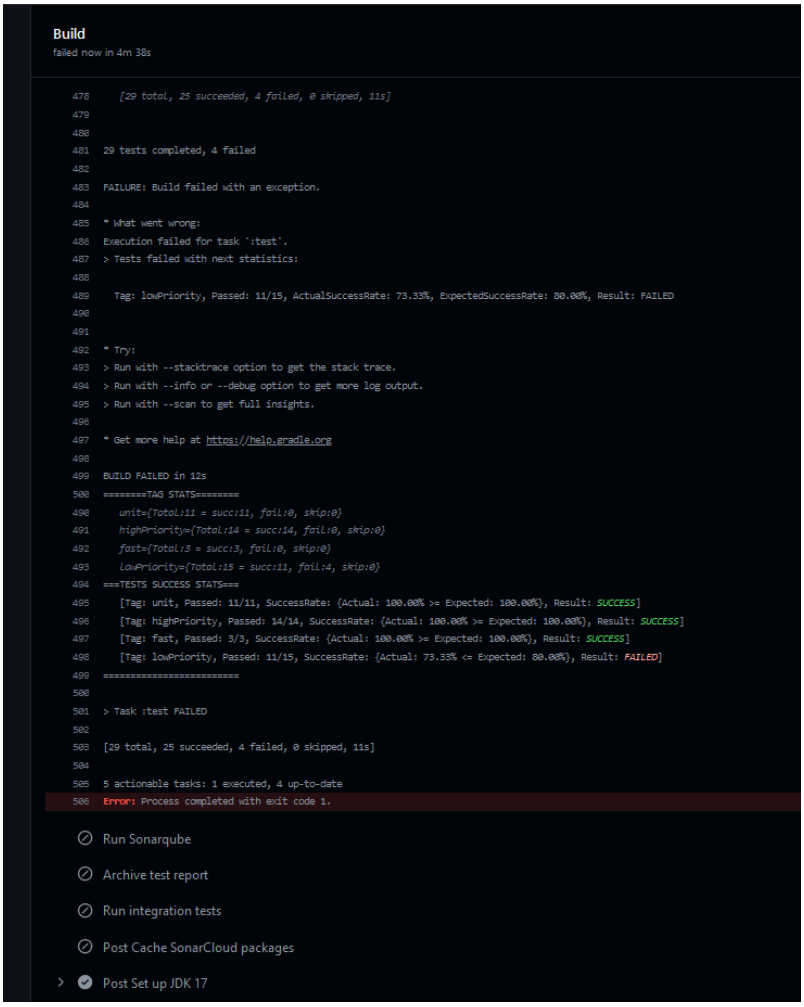

Рисунок 22 Приклад невдалого тестування програми з допомогою створеного плагіну

В помилці виконання вказане, які саме тести спричинили збій. В цьому випадку, до команди також було додано параметр **-PtestsManager.verbose**, який відповідає за показ повної статистики по всім анотаціям.

Можливі покращення

- Реалізація аналогічного плагіну під **Maven**
- Додати підтримку тестових бібліотек окрім **JUnit5**
- Додати підтримку паралельного виконання тестів
- Публікація плагіну в офіційний **Gradle**-репозиторій плагінів

Висновок

В ході роботи було проведено дослідження способів управління проходженням тестів у **CI/CD** процесі. Було розглянуто проблеми, які виникають при тестуванні програм, різні типи автоматизованих тестів, бібліотеки та фреймворки для написання тестів, плагіни для покращення управлянням проходженням тестів та платформи для впровадження **CI/CD** і та здійснено порівняльний аналіз

На основі цих досліджень було спроєктовано і розроблено власний **Gradle**-плагін. Плагін було створено мовою програмування **Kotlin**, додаткову бібліотеку тегів на **Java**, а також веб-проєкт для тестування плагіну на фреймворку **SpringBoot**.

Для ведення розробки було використано систему контролю версій **Git**, платформу **GitHub**, і середовище інтегрованої розробки **Intellij IDEA**.

Було зроблено висновок, що тестування є одним з найбільш важливих етапів у **CI/CD** процесі, і від якості його імплементації залежить успіх майбутнього продукту.

Список використаної літератури

[1] — Code coverage best practices. *Google Testing Blog*. URL: https://testing.googleblog.com/2020/08/code-coverage-best-practices.html (дата звернення: 18.05.2022).

[2] — TestPyramid. *martinfowler.com*. URL: https://martinfowler.com/bliki/TestPyramid.html (дата звернення: 18.05.2022).

[3] — Types of software testing: different testing types with details. *Hackr.io*. URL: https://hackr.io/blog/types-of-software-testing (дата звернення: 18.05.2022).

[4] — 13 best test automation frameworks: the 2021 list. *LambdaTest*. URL: https://www.lambdatest.com/blog/best-test-automation-frameworks-2021/ (дата звернення: 18.05.2022).

[5] — Selenium Vs. Cucumber Explained: Quick Guide to Key Differences | by Perforce. *Perfecto by Perforce*. URL: https://www.perfecto.io/blog/cucumber-vs-selenium (дата звернення: 18.05.2022).

[6] — TestNG Tutorial: What is Annotations & Framework in Selenium. *Guru99*. URL: https://www.guru99.com/all-about-testng-and-selenium.html#3 (дата звернення: 18.05.2022).

[7] — Maven Surefire Plugin — Introduction. *Maven — Welcome to Apache Maven*. URL: https://maven.apache.org/surefire/maven-surefire-plugin/ (дата звернення: 18.05.2022).

[8] — Maven Failsafe Plugin — Introduction. *Maven — Welcome to Apache Maven*. URL: https://maven.apache.org/surefire/maven-failsafe-plugin/ (дата звернення: 18.05.2022).

[9] — Intro to JaCoCo | Baeldung. *Baeldung*. URL: https://www.baeldung.com/jacoco (дата звернення: 18.05.2022).

[10] — The 8 Phases of Continuous Integration & Continuous Delivery | Free Guide. *Densify*. URL: https://www.densify.com/resources/continuous-integration-delivery-phases (дата звернення: 18.05.2022).

[11] — FAUN Publication. *Medium.com*. URL: https://faun.pub/github-actions-vs-jenkins-which-should-you-use-d7ba6800e8bb (дата звернення: 18.05.2022).

[12] — Maven vs. Gradle in-depth comparison. *Tom Gregory*. URL: https://tomgregory.com/maven-vs-gradle-comparison/ (дата звернення: 18.05.2022).

[13] — Gradle Documentation. *Gradle User Manual*. URL: https://docs.gradle.org/current/userguide/plugins.html (дата звернення: 18.05.2022).

[14] — Guide to Java Reflection | Baeldung. *Baeldung*. URL: https://www.baeldung.com/java-reflection (дата звернення: 18.05.2022).

[15] — Class Loaders in Java | Baeldung. *Baeldung*. URL: https://www.baeldung.com/java-classloaders (дата звернення: 18.05.2022).

[16] — Maven — Introduction to Repositories. *Maven — Welcome to Apache Maven*. URL: https://maven.apache.org/guides/introduction/introduction-to-repositories.html (дата звернення: 18.05.2022).

[17] — GitHub - renaudcerrato/appengine-maven-repository: Free Private Maven repositories hosted on Google App-Engine, backed by Google Cloud Storage and deployed in less than 5 minutes. GitHub. URL: https://github.com/renaudcerrato/appengine-maven-repository (дата звернення: 18.05.2022).

Додаток А

(обов`язковий)

```yaml
name: Build
on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build:
    name: Build
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0  # Shallow clones should be disabled for a better relevancy of analysis
      - name: Set up JDK 17
        uses: actions/setup-java@v1
        with:
          java-version: 17
      - name: Cache SonarCloud packages
        uses: actions/cache@v1
        with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar
      - name: Cache Maven packages
        uses: actions/cache@v1
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: mvn clean org.jacoco:jacoco-maven-plugin:prepare-agent verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=brutovsky_micro2do
      - name: Run integration tests
        run: mvn failsafe:integration-test -DskipITs=false
```