МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
Литвиненко О.Є.
"_____"_____2020 р.

ДИПЛОМНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
"МАГІСТР"

Тема: «Плагін автоматизованого тестування програмної системи у *Jenkins*»

Виконавець: Белозьорова А.С.

Керівник: Коба О.В.

Нормоконтролер: Тупота Є. В.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра комп'ютеризованих систем управління

Спеціальність 123 «Комп'ютерна інженерія»

(шифр, найменування)

Освітньо-професійна програма «Системне програмування»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Литвиненко О.Є.

«_____»_____2020 р.

ЗАВДАННЯ

на виконання дипломної роботи (проекту)

Белозьорова Анастасія Сергіївна

(прізвище, ім'я, по батькові випускника в родовому відмінку)

1.Тема дипломної роботи (проекту) «Плагін автоматизованого тестування програмної системи у *Jenkins*»

затверджена наказом ректора від « 27 » серпня 2020 р. № 1203/ст

2. Термін виконання роботи (проекту): з 5 жовтня 2020 по 13 грудня 2020

3. Вихідні дані до роботи (проекту) інформація про автоматизацію програмного продукту, інформація про тестування, *Jenkins* плагін для автоматизованого тестування.

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

1) Теорія тестування

2) Життєвий цикл розробки програмного забезпечення для автоматизації

3) *Jenkins* плагін для автоматизованого тестування

4) Програма для автоматизованого тестування

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

1) Схема алгоритму процесу розробки в *Jenkins*

2) Розробка релізу за допомогою *Jenkins*

3) Використання плагинів

4) Архітектура використання плагіну

5) Процес безперервної інтеграції в *Jenkins*

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Отримання завдання, написання вступу	05.10-06.10	
2	Аналіз предметної області, написання 1 розділу	07.10-25.10	
3	Аналіз автоматизованого тестуваннння, написання 2 розділу	26.10-01.11	
4	Встановлення програм для написання програми	02.11-03.11	
5	Аналіз роботи та підключення *Jenkins* та *Git*	04.11-06.11	
6	Написання програми плагіну	07.11-15.11	
7	Написання 4 розділу та висновків	16.11-29.11	

7. Дата видачі завдання: “_05_”__жовтня__ 2020 р.

Керівник дипломного проекту _____ Коба О.В.
(підпис керівника)

Завдання прийняв до виконання _____ Белозьорова А.С.
(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи: Плагін автоматизованого тестування програмної системи у *Jenkins*: 90 сторінки, 25 рисунків, 25 використаних джерел.

Ключові слова: автоматизоване тестування, *Jenkins*, *Git*, плагіни у *Jenkins*.

Мета дипломної роботи: розробка та створення плагіну автоматизованого тестування програмної системи у *Jenkins* з додаванням коду в *GitHub*.

Об'єкт дослідження: концепція плагіну у програмній платформі.

Предмет дослідження: плагін автоматизованого тестування програмної системи у *Jenkins*.

Методи дослідження: мова програмування *Java*, *Git*, плагіни у *Jenkins*, автоматизований плагін *Jenkins*, тестування та автоматизоване тестування, написання тестів, життєвий цикл розробки програмного забезпечення.

Наукова значимість: полегшене та прискорене тестування програмних продуктів з застосуванням новітніх інформаційних технологій.

Практична значимість: розробка програмного забезпечення, що планується використовувати при створенні нового та оригінального програмного продукту.

Прогнозні припущення щодо подальшого розвитку матеріалів проекту полягає у збільшені функціоналу який стосується більш детальної інформації та автоматизації більш складних функціональних додатків, та використання даного методу в ІТ компаніях. Застосування новітніх фреймворків.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

XML – Розширювана мова розмітки *(Extensible Markup Languege*)

SOAP – Простий протокол доступу до об’єктів *(Simple Object Access Protocol*)

SDLC – Життєвий цикл розробки програмного забезпечення *(Software Development Life Cycle*)

URI – Уніфікований ідентифікатор ресурсів *(Uniform Resource Identifier*)

URL – Єдиний вказівник на ресурс *(Uniform Resource Locator*)

CI/CD – Методологія розробки програмного забезпечення *(Continuous integration & Continuous delivery*)

API – Прикладний програмний інтерфейс *(Application programming interface*)

HPC – Блочнийий симетричний криптоалгоритм (*Hasty Pudding Cipher*)

HTTP – Протокол передачі даних, що використовується в комп'ютерних мережах *(Hyper Text Transfer Protocol*)

DB – База Даних(*Data Base*)

HTML – стандартна мова розмітки для створення *web*-сторінок і *web*-додатків (*HyperText Markup Language*)

SVN – Безкоштовна система управління версіями з відкритим вихідним кодом (*Subversion*)

CI – Постійна інтеграція(*Continuous Integration*)

SSH – Мережевий протокол рівня додатків *(Secure Shell)*

jar – *Java*-архів *(Java ARchive)*

Jenkins – Сервер автоматизації з відкритим кодом, який можна використовувати для компіляції та тестування вихідного коду

ВСТУП

Темою дипломної роботи є автоматизація тестування програмного продукту за допомогою плагіну в *Jenkins*.

Забезпечення якості – невід’ємна частина циклу розробки програмного продукту. При кожній незначній зміні програмного продукту існують аргументи щодо необхідності виконання його автоматизованого тестування. Проте деякі програмісти вважають, що перевірка функціональності програмного забезпечення повинна здійснюватися лише у найважливіших частинах циклу розробки продукту.

Тестування є життєво важливою частиною процесу розробки програмного забезпечення, і оскільки *web*-додатки стають все більш важливими у нашому житті, надзвичайно важливо, щоб вони були протестовані належним чином. Існує багато різних аспектів, які можна перевірити, і пріоритетом тестування часто є спроба знайти помилки та проблеми безпеки, перевіривши вихідний код на низькому рівні, перевіряючи зв'язок із сервером та базою даних.

Користувацький інтерфейс (*UI*) *web*-додатків раніше був на базовому рівні, тому ретельне тестування не було пріоритетом. Але оскільки *web* -додатки стають більш досконалими та динамічними, тестування функціональності інтерфейсу *web* -додатків набуває більш важливого значення.

Одним із підходів до тестування функціональності інтерфейсу є проведення тестування вручну та за допомогою тестувальників, які використовують інтерфейс, і повідомляють про будь-які проблеми, що часто виникають під час виконання плану тестування. Перевага цього підходу полягає в тому, що *web*-програма тестується щодо функціональності програми на основі реакцій та досвіду фактичного користувача. Недоліком є те, що це може зайняти багато часу і достатньо дорого.

Використання автоматизованого тестування при розробці програмного забезпечення дозволяє повторювані тести, які комп'ютер може виконувати кілька разів, робити менш дорогими та трудомісткими, ніж ручне тестування. Використання автоматизованого тестування *web*-додатків стає все більш

поширеним, але що стосується тестування функціональності інтерфейсу, автоматизація є складною. Більшість *web*-додатків є динамічними, а не статичними, що робить їх складними для автоматизованого тестування, оскільки їх вміст та елементи можуть змінюватися. *Web*-додатки часто неоднорідні, це означає, що вони складаються з компонентів, побудованих з використанням різних мов та методів, що також може ускладнити автоматизоване тестування. Широкий вибір *web*-браузерів, доступних сьогодні, є ще одним аспектом, який ускладнює автоматизованого тестування *web*-додатків, оскільки користувачі очікують однакової продуктивності *web*-додатки незалежно від того, який браузер використовується.

Безперервна інтеграція(*CI*) є важливою практикою в гнучких проектах. Вона може допомогти збільшити частоту релізів, зробити релізи більш керованими і поліпшити якість продукту і коду, що стоїть за ним. Однак це вимагає, того, щоб команда розробників слідувала не тільки практиці *CI,* але і всім її передумовам.

Jenkins – це простий у використанні, що високо настроюється *CI*-сервер з безліччю плагінів для настроювання інструменту для особливих потреб. *Web*-інтерфейс робить його доступним для всіх розробників, необхідна умова *CI.*

Безперервна інтеграція – це метод розробки, при якому розробники часто включають код в повністю інтегровану систему. Автоматизована побудова і автоматизовані тести можуть потім перевірити будь-яку інтеграцію. Автоматичне тестування, як правило, не є суворої частиною *CI*. Одне з головних переваг щоденної інтеграції полягає в тому, що можете легко виявити і швидко ідентифікувати помилки. Оскільки кожне внесене зміна, як правило, невелика, можна швидко визначити конкретні дії тестування, в результаті якого було виявлено дефект. Останнім часом *CI* став таким стандартним протоколом і набором основних елементів для розробки програмного забезпечення.

Метою дипломної роботи є розробка та створення плагіну автоматизованого тестування програмної системи у *Jenkins* з додаванням коду в *GitHub.*

У дипломній роботі будуть представлені та проаналізовані інструментарії та технології, які можуть бути використані для автоматизації тестування програмного

забезпечення. А також буде описано призначення та функціональність програмного продукту, який буде перевірений як частина даної роботи.

Об’єктом дипломної роботи є концепція плагіну у програмній платформі.

Предметом дипломної роботи є плагін автоматизованого тестування програмної системи у *Jenkins* з додаванням коду в *GitHub.*

Завдання, що мають бути виконані в роботі:

1. Дослідити роботу *Jenkins*;
2. З’єднати та підключити сервер *Jenkins*;
3. Створити програму;
4. Провести аналіз результатів;
5. Відобразити результат в графіку побудови білдів.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Тестування програмного забезпечення

Розробка програмного забезпечення є складним і тривалим процесом і він вимагає використання різних типів та інструментів тестування програмного забезпечення. Методи тестування змінюються з часом. Оскільки область тестування програмного забезпечення зростає, вона стає все більш складною. При автоматизованому тестуванні збільшується швидкість розробки, програми скорочується цикл її розробки, економиться час, людські ресурси та гроші.

У наші дні автоматизоване тестування проектів програмної інженерії проводиться у багатьох середовищах розробок.

Тестування використовується в галузі розробки програмного забезпечення як забезпечення якості різних частин програмного проекту. Це практикується не лише на завершальних стадіях розвитку, а й на всіх етапах різними способами. Наприклад, існують стратегії розвитку, засновані на використанні тестів, які визначають вимоги, а потім змінюють програму, поки вона не пройде тести. В інших випадках використовується тестування програмного забезпечення, яке постійно оновлюється та випускається в нових версіях, щоб перевірити, чи все ще працюють ті частини програмного забезпечення, які раніше працювали [1].

Тестування іноді сприймається як демонстрація відсутності помилок, коли зазвичай тестування – це процес, який використовується для підвищення надійності та якості програмного забезпечення шляхом пошуку якомога більшої кількості помилок [1].

1.1.1. Рівні тестування

Однією з фундаментальних частин всієї розробки програмного забезпечення є концепція абстракції. Абстракцію можна описати як спосіб розкладання програми на різні рівні з різним рівнем деталізації. Це дозволяє розробнику ігнорувати певні деталі програмного забезпечення, а натомість зосередитись на

інших деталях. Розглянемо розробку простої гри з базовою графікою. На найнижчому рівні, така гра вимагає величезної роботи, щоб перемішувати дані між апаратними шинами, виконувати доступ до пам'яті та операцій процесора. Використовуючи більш високі рівні абстракції, можна використовувати сторонні фреймворки для малювання графіки на екрані та виявлення зіткнень. Операційна система та мова програмування піклуються про обробку доступу до шини та управління пам’яттю. Це дозволяє розробнику зосередитися на розробці самої ігрової логіки, а не заморочуватись малюванням окремих пікселів або з'ясувати, де в пам'яті зберігати дані [2].

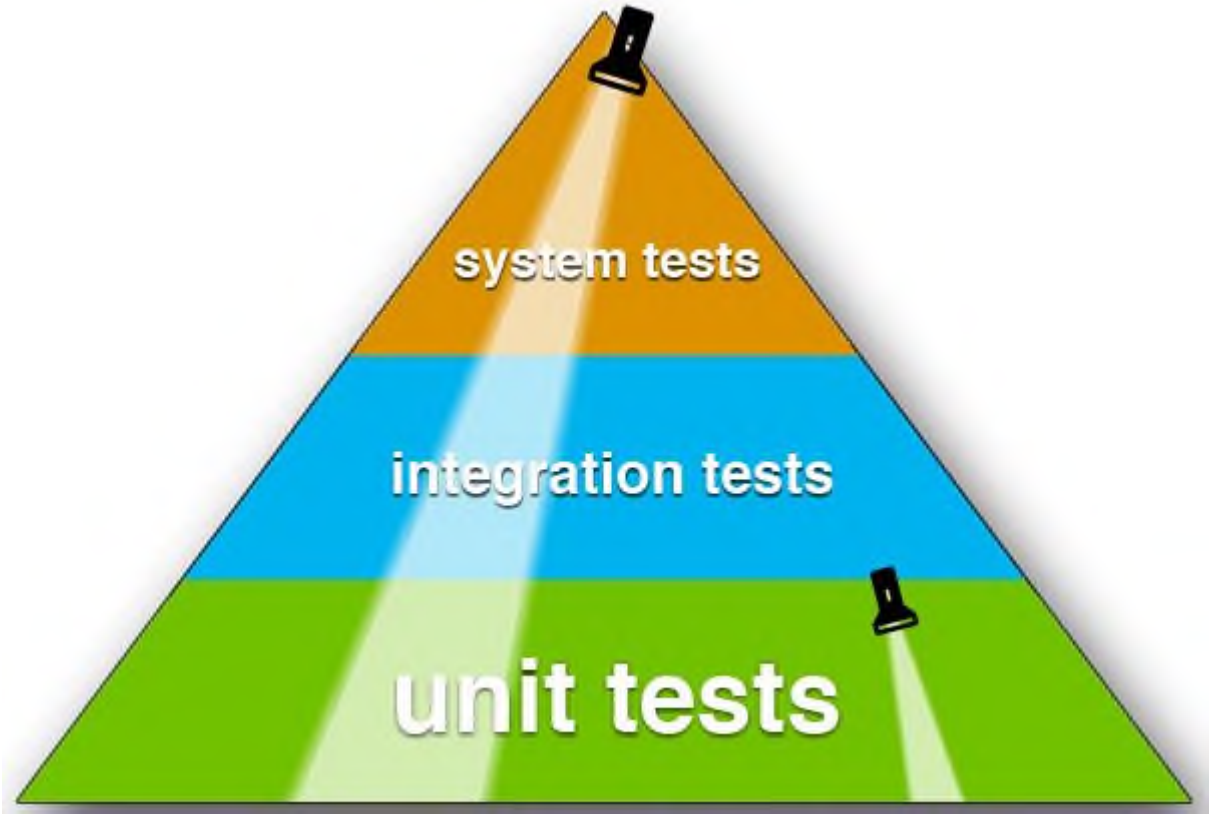

Рис. 1.1. Піраміда тестування програмного забезпечення

На рис 1.1. представлена піраміда тестування програмного забезпечення з двома ліхтариками на різних рівнях, що ілюструють, як рівень тестування впливає на кількість тестованого коду.

Модульне тестування (*Unit testing*) – це процес виконання та перевірки системного компонента з метою перевірки його функціональності. В об'єктно-орієнтованому програмуванні одна одиниця представляє, швидше за все, один клас,

і мінімальний тестовий випадок охоплює перевірку конструктора та деструктора. Модульні тести зазвичай пишуться розробниками під час розробки програмного забезпечення. Модульне тестування призначене для оцінки одиниць, вироблених на етапі впровадження, і є “найнижчим” рівнем тестування. У деяких випадках, наприклад, під час створення бібліотечних модулів загального призначення, модульне тестування проводиться без знання програмного забезпечення, що інкапсулює. Як і при тестуванні модулів, більшість організацій, що займаються розробкою програмного забезпечення, відповідають за програмування модульного тестування.

Мета модульних тестів – перевірити реалізацію, тобто переконатися, що тестований модуль працює коректно [5]. Оскільки кожен модульний тест охоплює лише невелику частину програмного забезпечення, легко знайти причину невдалого тесту. З іншого боку, єдиний модульний тест перевіряє лише те, що невелика частина програми працює належним чином.

Основною частиною модульного тестування є ізоляція кожного блоку, щоб тести провалились через не пов’язані зміни. Одним із способів вирішення залежностей від інших модулів є використання одного об’єкта, який замінює інший модуль під час виконання тесту. Об'єкт заміни має відоме значення, яке визначається тестом, а це означає, що зміни реального об'єкта не вплинуть на результат тесту.

Самостійне написання модульних тестів не дає достатнього охоплення тестом для всієї системи, оскільки модульні тести лише гарантують, що кожен окремий тестований модуль працює належним чином. Оскільки модульний тест гарантує лише те, що одиничний блок працює належним чином, помилки все ще можуть виникати в тому, як модулі працюють разом. Добре перевірена функція перевірки 12-значного ідентифікаційного номера нічого не варта, якщо модуль, який використовує його, передає 10-значний номер як вхід. Метою інтеграційних тестів є тестування декількох окремих блоків разом, щоб побачити, чи працює більша частина програмного забезпечення належним чином.

Інтеграційне тестування (*Integration tests*) охоплює перевірку контракту між окремими модулями, які вже пройшли фазу модульного тестування. Ці модулі можуть бути перевірені ітеративно або всі одночасно. Найважливішою частиною цього тестування є перевірка зв'язку між модулями. Існує ймовірність того, що всі модулі успішно пройдуть модульні тести і здається, все працює правильно, але після підключення всіх модулів до однієї системи можуть виникнути серйозні дефекти, які можуть призвести до повної відмови системи. Загалом, основним завданням інтеграційного тестування є запобігання такому жахливому сценарію.

Замість інтегрованих тестів є інший тип інтеграційних тестів, що називається контрактними та спільними тестами. Метою цих тестів є перевірка інтерфейсу між усіма модулями, що перевіряються модулем, за допомогою макетів для перевірки того, що блок А намагається викликати *xposed* – методи в блоці В. Це називається контрактним тестом. Щоб уникнути помилок через глузування, також необхідні тести, щоб переконатися, що блок дійсно відповідає вимогам, які очікуються, буде виконано блоком А в контрактному тесті. Ідея полягає в тому, щоб за допомогою транзитивності побудувати ланцюг довіри всередині нашого власного програмного забезпечення. Це означає, що якщо блок А і блок В працюють разом, як очікувалося, а блок В і блок С працюють разом, як очікувалося, блок А і блок С також будуть працювати разом, як очікувалося.

Системне тестування (*system testing*) перевіряє всі модулі, які пройшли інтеграційне тестування, і всю саму систему, інтегровану з будь-яким придатним обладнанням. Під відповідним обладнанням це може означати настільний комп'ютер для тестування настільних додатків або сервер для *web*-додатків. Тестування системи призначене для того, щоб визначити, чи відповідає зібрана система її специфікаціям. Він передбачає, що шматки працюють окремо, і запитує, чи працює система в цілому. Цей рівень тестування зазвичай шукає проблеми з дизайном та специфікацією. Це дуже дороге місце для пошуку несправностей нижчого рівня, і зазвичай це роблять не програмісти, а окрема команда тестування.

Найважливішою частиною системного тестування для обсягу цієї дипломної роботи є функціональне тестування. Метою функціонального тестування є

перевірка функціональних вимог програми на найвищому рівні. Іншими словами, потрібно переконатись, що функціонал, який використовується кінцевими користувачами, працює належним чином. Це може бути тоді, коли всі компоненти системи тестуються разом, а також вперше система тестується на декількох платформах. Через це деякі типи дефектів програмного забезпечення можуть інколи не виявлятися, поки не буде проведено тестування системи.

В якості базової ілюстрації прикладу тестування системи може бути використаний такий сценарій:

1. Увійти до системи;
2. Створити замовлення;
3. Створити перевірку вмісту;
4. Редагувати вміст;
5. Видалити вміст;
6. Вийти із системи.

Приймальне тестування (*Acceptance testing*) – це ще більш високий рівень тестування, ніж системне тестування. Його мета – визначити, чи відповідає вся система критеріям, узгодженим із замовником. Цей процес може включати оцінку результатів існуючих системних тестів, а також проведення ручного тестування та переконатися, що існують особливості для певних випадків використання. На відміну від нижчих рівнів тестування, тестування рівня прийнятності не тільки забезпечує роботу системи, а й те, що вона містить правильні функції [6].

Приймальне тестування не виходить за рамки цієї дипломної роботи, тому даний вид тестування включений до теорії.

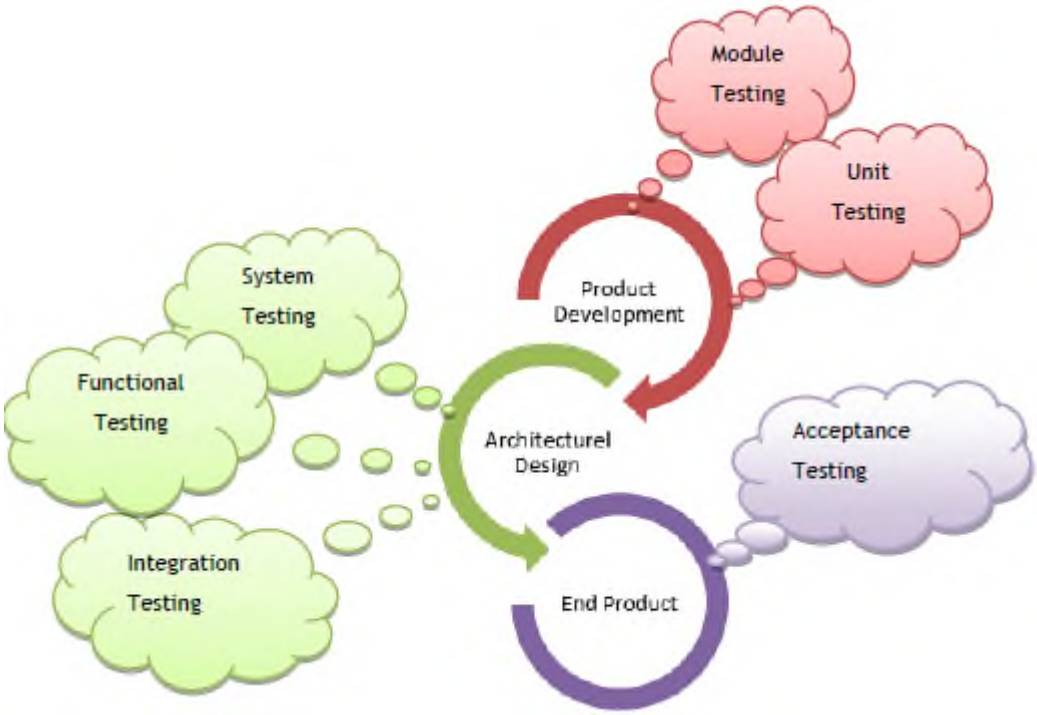

Рис. 1.2. Етапи розробки та необхідні тести

1.1.2. Методи тестування

Тестові кейси розробляються з використанням різних методів тестування для досягнення більш ефективного тестування. Цим забезпечується повнота програмного забезпечення та вибираються умови тестування, які отримують найбільшу ймовірність виявлення помилок. Отже, тестувальники не здогадуються, які тестові кейси обрати, а методи тестування дозволяють систематично розробляти умови тестування. Крім того, якщо комбінувати всі види існуючих методів тестування, можна отримати кращі результати, а якщо використовувати лише одну техніку тестування, можливий пропуск критичних помилок у розробленому програмному забезпеченні.

Програмне забезпечення можна перевірити двома способами. Іншими словами, можна виділити два різні методи:

1. Тестування чорного ящику;

2. Тестування білого ящику.

Тестування білого ящику є високоефективним у виявленні та вирішенні проблем, оскільки помилки часто можна виявити до того, як вони спричинять проблеми. Можна коротко визначити цей метод як тестування програмного

забезпечення зі знанням внутрішньої структури та кодування всередині програми. Тестування білого ящика також називають аналізом білого ящика, тестуванням чистих ящиків або аналізом явного ящика. Це стратегія налагодження програмного забезпечення, в якій тестувальник чудово знає, як взаємодіють компоненти програми. Цей метод може бути використаний для додатків *web* – служб і рідко практичний для налагодження у великих системах та мережах. Крім того, у тестуванні білих ящиків розглядається як тестування безпеки, який може бути використаний для перевірки того, чи відповідає реалізація коду призначеній розробці, перевірити реалізовану функціональність безпеки та виявити вразливі місця, що використовуються [3].

Тестування чорного ящика – це тестування програмного забезпечення на основі вимог до виводу та без знання внутрішньої структури або кодування в програмі. Іншими словами, чорний ящик – це будь-який пристрій, робота якого не зрозуміла або доступна користувачеві. При аналізі даних чорний ящик – це алгоритм, який не дає пояснення того, як це працює. У кіновиробництві чорний ящик – це спеціальний апаратний пристрій: обладнання, яке спеціально використовується для певної функції, але у фінансовому світі це комп'ютеризована торгова система, яка не робить свої правила легкодоступними.

Тестування чорного ящику також називають функціональним тестуванням, що розробляє тестові кейси на основі інформації із специфікації. При тестуванні чорного ящику тестувальник програмного забезпечення не має доступу до самого коду. Тестування чорного ящику не стосується внутрішніх механізмів системи; вони зосереджені виключно на результатах, що генеруються у відповідь на вибрані параметри та умови виконання [4].

В останні роки також розглядається третій метод тестування – тестування сірого ящика. Він визначається як тестування програмного забезпечення, маючи вже певні знання про його базовий код або логіку. Він базується на внутрішніх структурах даних та алгоритмах для проектування тестових випадків більше, ніж тестування чорного ящика, але менше, ніж тестування білого ящика. Цей метод важливий при проведенні інтеграційного тестування між двома модулями коду,

написаними двома різними розробниками, де для тестування доступні лише інтерфейси. Також цей метод може включати зворотне проектування для визначення граничних значень. Тестування сірого ящика є ненав'язливим та неупередженим, оскільки для нього не потрібно мати доступ до вихідного коду.

1.1.3. Види тестування

Під час процесу тестування існує величезна кількість типів тестів, які можна виконати, і кожен з них має свою мету. У цьому підрозділі будуть загалом описані всі значні види тестування.

Alpha – тестування – це перший етап тестування, який проводиться на ранній стадії тестування. Рекомендується тестувати лише програмне забезпечення розробників. Після виявлення та виправлення основних дефектів програмного забезпечення процес тестування може перейти до *Beta* – тестування.

Beta – тестування переходить в *Alpha* – тестування, і його часто називають другим етапом тестування. Цей тип тестування можна розглядати як форму перевірки на відповідність користувача. В основному це передбачає випуск *Beta* – версії в зовнішнє середовище команди розробників програмного забезпечення. Програмне забезпечення доступне для невеликої групи людей, як правило, майбутніх клієнтів. Великим внеском *Beta* – тестування є кількість відгуків, які можна отримати. Слід зазначити, що *Beta* – тестування здебільшого проводиться після закінчення внутрішнього тестування, що означає тестування розробниками та командою забезпечення якості. Але це не правило.

Одночасне тестування (*Concurrent testing*) відстежує продуктивність та забезпечує результат тестованого програмного забезпечення під час його звичайної діяльності. Одночасне тестування виконується одночасно з функціональним тестуванням і його метою є визначення стабільності та продуктивності за очікуваних обставин.

Стрес тестування (*Stress testing*) – намагається змусити програмне забезпечення працювати з помилками або завершити роботу програмного забезпечення. Це тестування іноді називають негативним тестуванням. Таке тестування намагається зробити програму неробочою, надаючи небезпечні тестові

ресурси, такі як дуже довгі тексти, невідомі символи або сценарії. Основна мета стрес тестування – перевірити можливість відновлення системи, що є дуже важливим аспектом надійного програмного забезпечення.

Тестування встановлення (*Installation testing*) – мета такого тестування полягає в тому, щоб визначити, чи правильно встановлено програмне забезпечення та чи воно повністю функціонує на фактичному апаратному забезпеченні замовника. Цей тип тестування має свою протилежність – це тестування на видалення (*uninstallation testing*), і його мета – перевірити, чи правильно було видалено програмне забезпечення.

Поширеною причиною несправності програмного забезпечення є те, що воно тестувалось лише в одному середовищі. У цьому випадку середовище може бути представлене *web*-браузером, операційною системою, можливо, мобільною платформою. Неможливо, щоб у всіх замовників, які купують певний товар, у кожного з них були однакові пристрої. У них може бути спільне одне – використання одного програмного продукту, але не всі вони використовують його в одному середовищі. Отож, оскільки кожен клієнт відчуває потребу знайти відповідність бажаному середовищу, необхідно передбачити всі потреби клієнта. Це підтверджується тестуванням сумісності (*compatibility testing*).

Регресійне тестування (*Regression testing*) визначає, коли модифікації коду призводять до того, що раніше працююча функціональність регресувала або не спрацьовувала, в кінцевому рахунку дозволяючи ловити помилки регресії відразу після їх введення. Більшість організацій перевіряють критичну функціональність один раз, а потім припускають, що вона продовжує працювати, якщо вони навмисно не модифікують її. Однак навіть рутинні та незначні зміни коду можуть мати несподівані побічні ефекти, які можуть порушити раніше перевірену функціональність.

Це вказує на великий попит на інтеграцію регресійних тестів у процес тестування. Хоча ці випробування дуже складні, масштабні і повинні виконуватися постійно. Через ці фактори регресійні тести є найкращим кандидатом на автоматизацію. Це може здатися простим завданням, але з метою подальшої

послідовності та стабільності регресійних тестів повинна бути встановлена чітка політика щодо написання автоматизованих тестових кейсів. В іншому випадку технічна заборгованість коду буде рости все більшою і більшою, поки весь проект не зможе розвалитися.

Димове тестування (*Smoke testing*) передбачає мінімальну кількість випробувань, яку можна провести, щоб переконатись, що система, що перевіряється, готова до подальших випробувань. Це можна вважати захистом для подальшого тестування, якщо система не може виконати якісь мінімальні операції, ви не перейдете до запуску повного набору тестів. Зазвичай це невелика кількість тестів, які перевіряють, чи можна встановити програмне забезпечення, чи працює основна функціональність і чи не з’являються очевидні проблеми.

Існують різні типи тестів, які можна використовувати, і кожен тип надає інформацію про інший аспект програмного забезпечення. Дуже важливо подбати про те, щоб виконати якомога більше тестів, які можна зробити за наявного часу.

1.2. Тестування *WEB* – додатків

Web – програми, як правило, мають декілька властивостей із традиційним програмним забезпеченням, тобто програмним забезпеченням, яке працює як додаток, локально на одному комп'ютері. Багато мов можна використовувати для написання традиційного програмного забезпечення, а також *web* – додатків, і мета тестування програмного забезпечення, як правило, однакова. Однак існують деякі ключові відмінності та особливості, які демонструють *web* – додатків.

Характеристики *web* – додатків :

- програмою може користуватися велика кількість користувачів з різних географічних розташувань;
- середовище виконання є дуже складним і може включати різне обладнання, *web* – сервери, підключення до Інтернету, операційні системи та *web* – браузери;

– саме програмне забезпечення, як правило, складається з різних компонентів, які часто використовують кілька різних технологій. Наприклад, може мати серверний компонент, який використовує *Ruby* та *Ruby on Rails*, і клієнтський компонент, який використовує *HTML, CSS* та *Javascript*;
– деякі компоненти, такі як частини графічного інтерфейсу, можуть генеруватися під час виконання, залежно від вводу користувача та поточного стану програми.

Кожна з цих характеристик сприяє різним проблемам при тестуванні програмного забезпечення. Хоча деякі з цих характеристик створюють проблеми тестування, що виходять за рамки даної дипломної роботи, інші проблеми є надзвичайно актуальними. Наприклад, може бути неможливим використання тієї ж основи тестування для компонентів на стороні сервера, що і для компонентів на стороні клієнта, оскільки компоненти написані різними мовами програмування. Іншим прикладом є складність середовища виконання, що може призвести до дефектів, які виникають в одних середовищах, але не в інших.

Як зазначалося раніше, *web* – додаток може працювати на різних операційних системах та браузерах, що може спричинити дефекти, характерні для кожного середовища. Для того, щоб виявити такі дефекти, потрібно мати можливість протестувати *web* – додаток у кожному підтримуваному середовищі.

Тест браузера – це свого роду тест, де *web* – браузер управляється з метою імітації поведінки реального користувача, натискаючи кнопки та вводячи текст у текстові поля. Цей підхід тестує заявку так само, як і при проведенні ручного тестування, що є перевагою та недоліком. З одного боку, такий спосіб тестування на сьогоднішній день є найбільш реалістичним способом тестування програми. З іншого боку, він часто тестує багато коду, і може знадобитися чимало зусиль, щоб ретельно перевірити речі та знайти причину помилок, коли тести не вдаються.

Сумісність між браузерами (*Cross-browser*) є важливою проблемою для багатьох *web* – додатків. Це в основному означає, що програма повинна працювати однаково незалежно від того, який браузер користувач використовує, принаймні

доти, доки ми вирішили підтримувати відповідний *web*–браузер. Тест браузера часто можна виконати в декількох браузерах і, таким чином, може допомогти знайти дефекти програмного забезпечення, характерні для браузера.

Перевірка браузера є повільним, здебільшого через рівень тестування, а також через накладні витрати на виконання сценаріїв, візуалізацію сторінок та завантаження зображень. Інший мінус полягає в тому, що дане тестування вимагає повноцінного графічного середовища робочого столу і, отже, для налаштування на сервері можуть знадобитися певні зусилля та ресурси. Одним із способів досягти вищої швидкості виконання тесту є використання браузера без голови. Безголовий браузер – це *web*–браузер, де більшість функціональних можливостей сучасних браузерів відсутні, щоб набрати швидкість. Зазвичай він не має жодного графічного інтерфейсу і тому не вимагає графічного середовища робочого столу [7].

Можна стверджувати, що запускати тести в безголовому браузері безглуздо, оскільки жоден реальний користувач не використовує такий браузер. Тому ми не можемо бути впевнені, що перевірена функціональність справді працювала б у реальному браузері лише тому, що вона працює в безголовому браузері. Звичайно, неможливо також знайти специфічні для браузера дефекти при використанні безголового браузера. Можна було б знайти помилки, характерні для самого браузера без голови, але це було б досить безглуздо.

Тести браузера часто мають тенденцію бути прив’язаними до класів *HTML* та *CSS* тестованої сторінки, оскільки нам потрібно знаходити елементи, щоб натискати кнопки або заповнювати форми. Це робить тести тендітними, оскільки зміни в користувальницькому інтерфейсі можуть їх порушити, і якщо ті самі елементи використовуються в багатьох тестах, нам може знадобитися переглянути великі частини нашого набору тестів.

1.3. Методології розробки ПЗ

Моделі розробки – це різні процеси або методології, обрані для розробки проекту відповідно до його мети та завдань. Моделі розробки програмного забезпечення допомагають поліпшити якість програмного забезпечення, а також процес розробки загалом.

Існує кілька моделей життєвого циклу розробки програмного забезпечення, кожна розроблена для певних цілей. *SDLC* – це середовище, яке описує дії, що виконуються на кожному етапі процесу розробки програмного забезпечення. *SDLC* складається з детального плану, який описує, як проводиться розробка, обслуговування та заміна конкретного програмного забезпечення. Це також відомо як процес розробки програмного забезпечення.

Протягом століть розробки комп'ютерів та програмного забезпечення було запропоновано кілька методологій розробки програмного забезпечення. Методологія розробки програмного забезпечення визначає різні види діяльності та моделі, якими можна керуватися під час розробки програмного забезпечення. Така діяльність може бути, наприклад, визначення вимог до програмного забезпечення або написання програмних реалізацій, і методологія, як правило, визначає процес із планом, як і в якому порядку слід виконувати діяльність. Модель водоспаду та модель *V* – два класичні приклади методологій розробки програмного забезпечення.

Тестова розробка (*Test-driven development TDD*) бере свій початок від першого тестового принципу в методології екстремального програмування і вважається однією з найбільш суперечливою та найвпливовішою гнучкою практикою. Слід зауважити, що фраза, керована тестовою розробкою, часто використовується в кількох інших контекстах, де обговорюються загальні практики тестового коду. У цьому розділі ми розглянемо основи *TDD* у його початковому значенні.

Водоспадна модель розробки ПЗ (*Waterfall development model*) – найдавніша модель циклу розвитку. Його назва походить від порівняння послідовності окремих фаз. Основна концепція цієї моделі базується на послідовному підході до кожної з

фаз. На основі моделі попередньою умовою початку нового етапу є те, що попередній етап повинен бути успішно завершений. Ранні фази розробки повинні виконуватися дуже обережно та ретельно, оскільки виявлення дефекту програми на ранній фазі має менший вплив на споживання часу, ніж це могло б мати на наступних фазах.

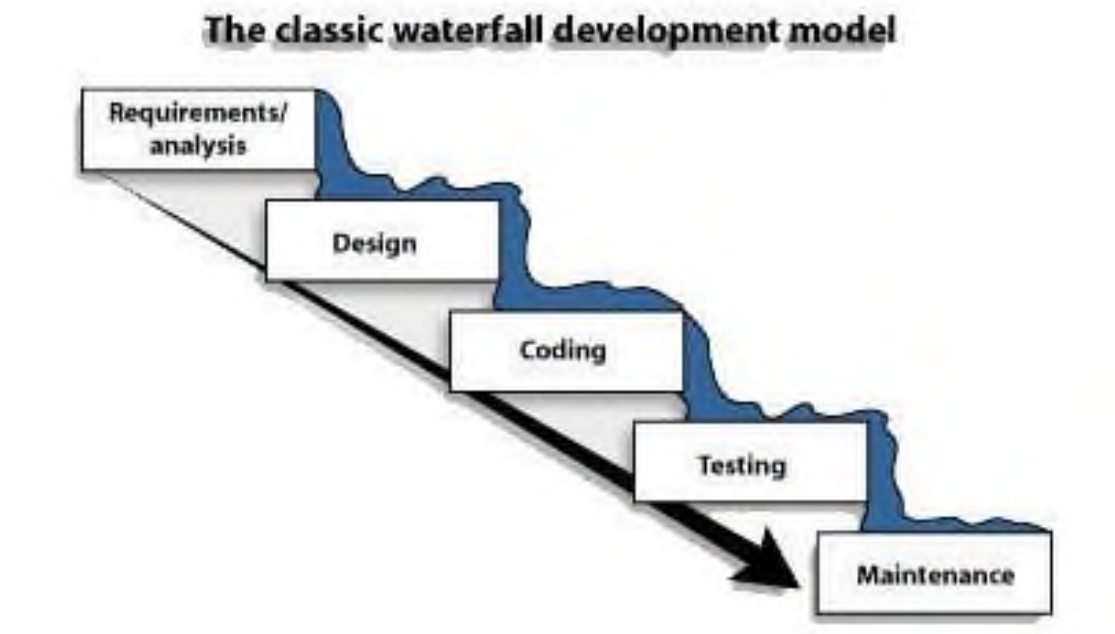

Рис 1.3. *Waterfall* модель розробки ПЗ

Під час розробки масштабних проектів завжди буде потрібно повернутися до раніше завершених фаз. Це може бути спричинено зміною вимог замовника. Найбільшим недоліком моделі водоспаду є те, що повернення до раніше завершених фаз неможливе, тому пристосування до вимог клієнта, що змінюється, може бути складним при використанні цієї моделі розвитку.

Раціональний уніфікований процес (*Rational unified process RUP*) базується на моделі водоспаду та має кращу здатність адаптуватися до мінливих вимог замовника. *RUP* складається з ітерацій, і кожна ітерація складається з усіх фаз, згаданих у моделі розвитку водоспаду. Основна концепція раціонального єдиного процесу буде описана нижче.

У кожній окремій ітерації встановлюється послідовність окремих завдань та її оцінка часу. Найкраща можлива ситуація, коли всі завдання ітерації виконані. Після аналізу створюється специфікація функції. Ця специфікація складається з

нових та існуючих функціональних можливостей. На основі специфікації функції може розпочатися процес розробки, який включає також процес тестування. Після успішного завершення всіх ітерацій виконується одне додаткове тестування всіх ітерацій. Після перевірки всіх ітерацій програмне забезпечення готове пройти прийомне тестування, яке необхідно перед розгортанням нового випуску до виробництва.

Спіральна модель розвитку (*Agile development model*) розробки ПЗ на сьогоднішній час дуже популярна. Це роблять усі круті компанії: *Google, Yahoo, Symantec, Microsoft* і список можна продовжувати. На сьогоднішній день багато компаній, які інтегрують гнучкі методології у свою бізнес-модель. Однак необхідно вказати, як це буде зроблено. Для найкращої роботи цієї методології клієнт повинен співпрацювати протягом усього процесу розробки.

Спіральна методологія – це процес планування та перевірки роботи. Метою методології є вдосконалення організації роботи. Весь процес розробки можна виділити на такі фази:

- перша ітерація – перший аналіз та реалізація базової функціональності;
- аналіз змін – виберіть, що буде впроваджено;
- впровадження;
- презентація клієнту;
- якщо продукт не закінчений, поверніться до кроку 2;
- якщо виріб закінчений, відбувається технічне обслуговування та вдосконалення.

На рис. 1.4 показані фази спіральної моделі.

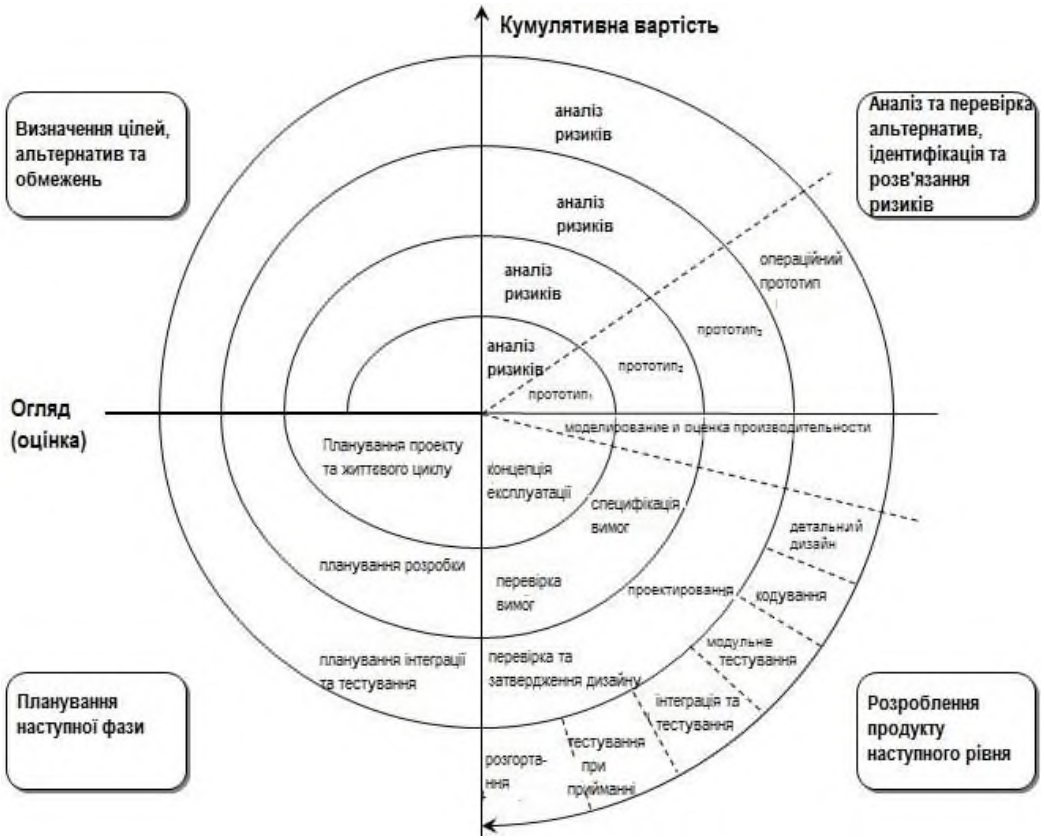

Рис 1.4. Спіральна модель розробки ПЗ

Скрам (*Scrum*) – одна з найвідоміших гнучких методологій і може застосовуватися в команді від чотирьох до п’ятнадцяти людей. Існує дві групи людей, які виступають у розробці. Група складається з власника продукту та майстра сутичок. Власник продукту несе відповідальність за те, що буде впроваджено в майбутньому спринті, і вказує деталі реалізації. Метою *Scrum Master* є керування розробниками. Він повинен задовольняти їх потреби, вирішувати їхні проблеми тощо. Інша група складається із зацікавлених сторін та менеджерів.

Фаза розвитку спринту називається спринтом. Тривалість спринту триває, і це часто становить 14 днів. Перед початком нового спринту завжди планується спринт. При плануванні спринту всі історії користувачів перераховуються, а потім переміщуються до журналу продукту. В рамках планування спринту створюються історії користувачів, вони асоціюються з оцінками, і слід обговорити, яка з історій буде реалізована у майбутньому спринті. Після закінчення кожного спринту вся

команда проводить ретроспективну зустріч, на якій кожен може висловити свої пропозиції або скарги.

З початком спринту приходить і початок розробки, який охоплює реалізацію історій користувачів. Після того, як історія реалізована, вона переходить до перегляду коду. Якщо інші розробники схвалюють огляд, історію користувача можна позначити як виконану та розпочати впровадження іншої історії. Кожного дня вся команда відвідує стенди, де кожен член команди описує обсяг своєї роботи.

Scrum базується на хорошому спілкуванні в цілому колективі, що призводить до кращої ефективності роботи. Усі члени команди повинні співпрацювати та допомагати один одному швидше досягти мети.

1.4. Висновки до розділу

Тестування використовується у галузі розробки програмного забезпечення як забезпечення якості різних частин програмного проекту. Це практикується не лише на останніх стадіях розвитку, а на всіх етапах різними способами. Наприклад, існують стратегії розвитку, засновані на використанні тестів, які визначають вимоги, а потім змінюють програму, поки вона не пройде тести. В інших випадках використовується тестування програмного забезпечення, яке постійно оновлюється та випускається в нових версіях, щоб перевірити, чи все ще працюють ті частини програмного забезпечення, які раніше працювали.

Тестування іноді сприймається як демонстрація відсутності помилок, коли зазвичай тестування – це процес, що використовується для підвищення надійності та якості програмного забезпечення шляхом вказівки якомога більшої кількості помилок.

Модульне тестування використовується для перевірки окремих одиниць у вихідному коді системи.

Коли в програмне забезпечення внесено зміни, використовується регресійне тестування, щоб перевірити, чи працювали раніше працюючі частини програмного

забезпечення. Це особливо корисно для програмного забезпечення, яке часто оновлюється та випускається в нових версіях.

Тестування білого ящика означає тестування програмного забезпечення щодо його внутрішніх компонентів. Прикладами цього є модульне тестування та тестування покриття тестовими сценаріями коду. Ці типи тестів призначені для перевірки того, що система задовольняє свої функціональні вимоги. Тестування білого ящика вимагає знання коду та дизайну системи.

На відміну від тестування білого ящика, тестування чорного ящика передбачає тестування системи, коли її внутрішні компоненти системи залишаються невідомими. Цей тип тестування передбачає виклик системних компонентів через взаємодію з користувальницьким інтерфейсом, а правильна поведінка перевіряється лише переглядом вихідних даних інтерфейсу користувача. Оскільки тестування чорного ящика зазвичай не виявляє всіх дефектів, слід також застосувати інший вид тестування, який називається тестуванням сірого ящика.

Тестування сірого ящика – це поєднання тестування білого та чорного ящиків, яке вимагає часткового знання внутрішньої роботи та використовується тестером для спроби виявити більше дефектів у порівнянні лише з тестуванням білого та чорного ящиків.

РОЗДІЛ 2
АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1. Постановка задачі

Концепція автоматизованого тестування може мати різні визначення залежно від джерела та контексту. Це може стосуватися автоматичної генерації фактичних тестових кейсів або автоматичного запуску тестових кейсів, розроблених вручну. У цьому звіті вибране визначення полягає в тому, що автоматизоване тестування – це тестування на основі розроблених вручну тестових кейсів, які можна повторно запускати сценарієм.

Існує багато причин для використання автоматизованого тестування. Оскільки тестові кейси є повторюваними, і комп'ютер може виконувати їх на високій швидкості, це зменшує час і вартість тестування. При правильному використанні автоматизоване тестування може поліпшити якість програмного забезпечення, оскільки комп'ютер може бути точніше реєструвати дефекти, ніж будь-який людський тестер. Інші позитивні аспекти використання автоматизованого тестування включають збільшення охоплення тестами, заміщення трудомістких завдань та ефективність регресійного тестування.

Тести можуть бути позначені як автоматизовані, якщо це можливо для запуску тестів без взаємодії з боку тестувальника. Автоматизовані тести повинні відповідати цим умовам:

- можливість виконання набору або підмножини тестових кейсів;
- немає необхідності втручатися після початку випробувань;
- важливі параметри тесту встановлюються автоматично;
- результати тесту реєструються автоматично;
- можливість порівняння фактичних результатів випробувань із очікуваними результатами;
- можливість аналізу результатів випробувань та надання результатів.

Сучасні системи автоматизації стають дедалі складнішими завдяки більш потужним програмованим контролерам, польовим пристроям та функціонально багатому програмному забезпеченню, що підтримує людський машинний інтерфейс (*HMI*), історикам тощо. Побудова гнучкої, розподіленої та сумісної системи автоматизації вимагає системного підходу до розробки програмного забезпечення, що, здається, не є нормою в проектах з розробки додатків для автоматизації (*AAD*). У проекті *AAD* беруть участь люди з різних інженерних дисциплін, багато з яких не мають досвіду програмної інженерії; отже, деякі важливі та фундаментальні принципи програмної інженерії часто втрачаються. Наприклад, рамки розробки, широко використовувані в *AAD*, не можуть ефективно вирішувати проблеми, що змінюються пізно, через ранні фази, що залежать від технології, та парадигми, орієнтовані на пристрій, що підтримуються використовуваною структурою *AAD*. Отже, існує потреба впровадити більш гнучкий, але надійний процес розробки програмного забезпечення в рамках проектів *AAD*. З цієї точки зору сучасні методології *AAD* вимагають серйозного контролю.

Більше того, існує необхідність оцінки придатності передових методологій програмної інженерії в процесах *AAD*. У цій роботі представлений критичний огляд нещодавніх спроб впровадження сучасних методів програмної інженерії в життєвому циклі розробки додатків для автоматизації (*AADLC*).

Спочатку ми коротко пояснюємо процес, задіяний у проекті *AAD*, а згодом обговорюємо найсучасніші методи та засоби, що використовуються для прийняття принципів програмної інженерії в проектах *AAD*. *AAD* також зазвичай називають "інженерією управління/автоматизації" або "інженерією додатків". Пояснюваний процес *AAD* ґрунтується на нашій взаємодії практиків та інформації, зібраній із наявної літератури. Отже, можуть бути більш точні відмінності в термінологіях та способі сприйняття *AAD*. Однак ми намагалися з усіх сил дати злагоджений погляд на різні галузі технологічного машинобудування та з різним масштабом.

2.2. Підходи до розробки автоматизованого тестування

Розробка програми автоматизації передбачає співпрацю людей, які беруть участь у проектуванні процесів, проектуванні об'єктів, електричному дизайні, проектуванні приладів та проектуванні автоматизації. Проекти *AAD* створюють програми управління для моніторингу та управління складними системами, такими як електростанції, нафтопереробні заводи тощо. Ми пояснюємо життєвий цикл *AAD* (*AADLC*) чотирма основними фазами:

1. Вимоги та дизайн;
2. Розробка;
3. Тестування;
4. Розгортання та введення в експлуатацію.

На рис. 2.1 показано *AADLC* з точки зору фаз, артефактів (задіяних або створених) у кожній фазі та ролей, відповідальних за керування кожною фазою. Також існує тривала фаза технічного обслуговування після фази введення в експлуатацію, яка не показана на малюнку, оскільки основна увага в цьому документі зосереджена лише на завершенні введення в експлуатацію. Кожна з цих фаз детально пояснюється в наступних розділах. Можуть бути більш точні відмінності у способі сприйняття фаз та генеруванні артефактів у *AADLC* на основі домену процесу.

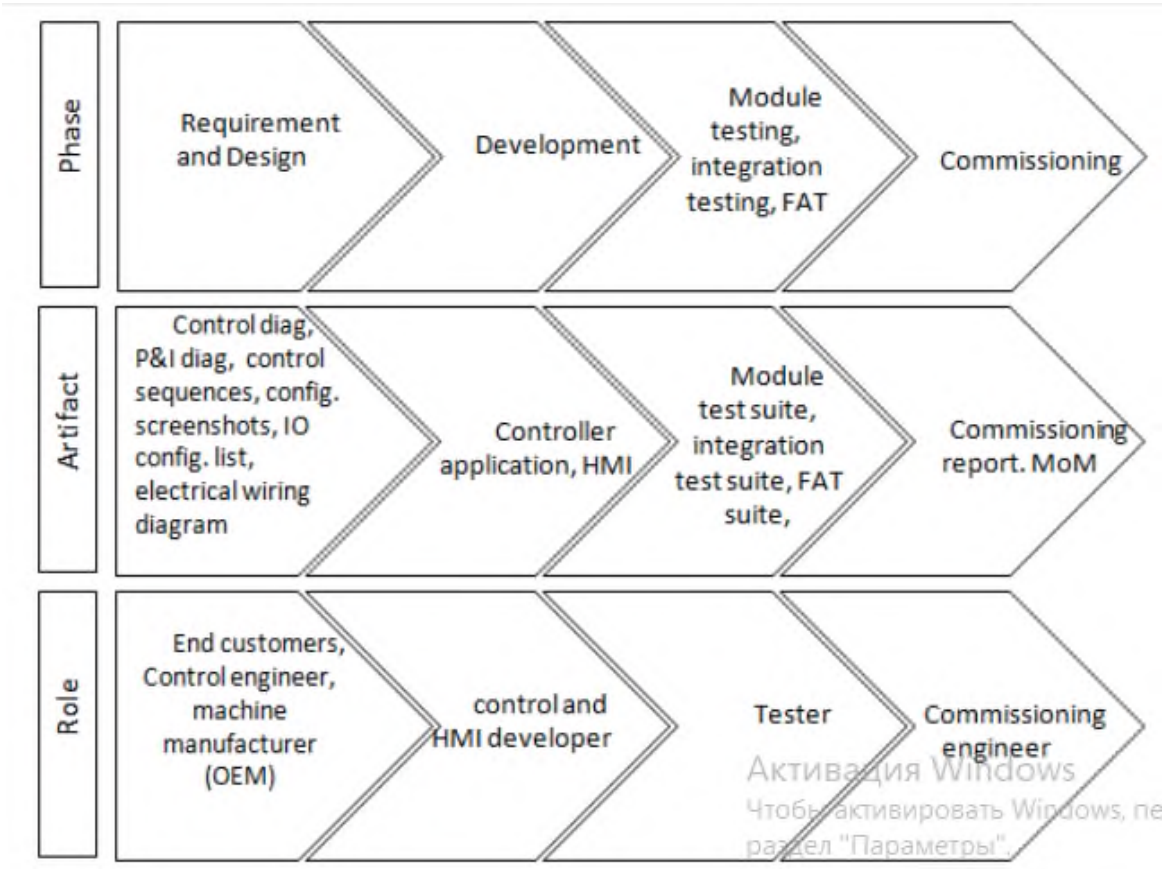

Рис. 2.1. Життєвий цикл розробки додатків для автоматизації

Експлуатаційні аспекти програми збираються під час цієї фази *AADLC*. Цей етап допомагає приймати основні інвестиційні рішення з точки зору вибору та придбання обладнання, такого як вводи-виводи, польові пристрої та контролери. На відміну від процесів програмної інженерії для бізнес-додатків, поділ між збором вимог та проектуванням часто здається розмитим у процесі *AAD*, і їх можна поєднувати та називати діяльністю проектування процесів. Основна причина полягає в тому, що вимоги деталізовані; наприклад, він складається з детальної інформації про кількість польових пристроїв та входів/виходів, які слід придбати, та опису процесу у вигляді схеми технологічного процесу (*PFD*). *PFD* відображає взаємозв'язок між основним обладнанням заводського об'єкта. Проекти генеруються на різних рівнях, починаючи від проекту на високому рівні (*PFD*) і закінчуючи більш детальним дизайном, який називається діаграмами процесів та приладів (*P&I*). Діаграми *P&I* відображають хід процесу разом з незначними деталями, такими як деталі трубопроводів та позначення. Цей етап повторюється,

доки проект не буде доопрацьований між замовниками та командою *AAD* в організації.

Основними формами охоплення вимог є *PFD* та діаграми *P&I*, однак у цих схематичних формах часто спостерігаються специфічні варіації переробної промисловості. Наприклад, різні типи насосів, клапанів тощо існують залежно від типу переробної галузі. Також представництво та термінологія різняться в залежності від організації. Наприклад, організації в Норвегії часто представляють вимоги в схемах системного контролю (*SCD*) [8], яка в основному розроблена для нафтової промисловості.

Додаток взаємодіє в основному з пристроями вводу-виводу в динамічному фізичному середовищі. Отже, призначення вводу-виводу контролерам та їх розташування відіграє важливу роль під час проектування. Хоча стандартна система моделювання вважається корисною для *AAD*, існує кілька важливих питань, які необхідно вирішити, щоб такі підходи були успішними [9]. Сюди входять:

1. Багатство модельної системи; наприклад, підтримка загальних елементів, таких як блокування;
2. Введення настроюваних термінологій, специфічних для доменів, стосовно технології процесів;
3. Визначення інтерфейсу від елементів моделювання до програм управління;
4. Підтримка візуалізації та редагування моделей на різних рівнях.

Наприклад, існує гостра потреба у визначенні додаткових поглядів (таких як вид взаємодії, перегляд матеріального потоку тощо), що мають значення для *AAD* та підтримуючих моделей для вираження цих поглядів. Інструменти для переходу від однієї моделі до еквівалентної сутності в іншій моделі потрібні для поліпшення простежуваності та чіткості вимог [10]. Наприклад, матеріальний потік може бути змодельований як подія в іншому поданні.

На сьогодні доступно багато середовищ прикладного програмування; до них належать як безкоштовні, так і власні інструменти. Часто компанії з автоматизації

використовують власне середовище програмування, яке інтегровано з їх платформою *AAD*.

Revision control system. Apache Subversion (*SVN*) вибрано для системи контролю версій для програми і використовується також для цього проекту. *SVN* – це централізована система управління версіями із відкритим кодом, що надається програмним фондом *Apache*. Для цього проекту *SVN* легко застосувати, оскільки всі використовувані технології та інструментарії мають плагін *SVN* у нашій системі, а тестові кейси та ресурси можна перевірити за допомогою вже використаних команд.

VMware vSphere забезпечує віртуалізацію, яка є абстракційним шаром, що порушує жорсткий зв'язок між фізичним *HW* (обладнання) та ОС (операційна система). Віртуальна машина – це комп'ютер із програмним забезпеченням, який схожий на звичайний фізичний комп'ютер, що працює під управлінням ОС, але має віртуальний *HW*, виділений із сервера *vCenter*. Величезна кількість віртуальних файлів можна легко розгорнути та клонувати залежно від потреби.

ОС *Windows* буде встановлена на віртуальній машині у віртуальній платформі *vSphere* для хмарних обчислень, що надається *VMware*. Клієнт *vSphere* вже використовується у нашому виробництві для розміщення нашого продукту, що працює на розподілених віртуальних машинах. Клієнт *vSphere* також використовується для клонування та створення знімків із продукту системи.

Robot Framework – це загальна система автоматизації тестів для приймального тестування, яка може використовуватися наскрізним тестуванням веб-програми. *Robot Framework* – це фреймворк, керований ключовими словами, який можна розширити за допомогою зовнішніх бібліотек, що містять ключові слова для певних типів тестування.

Jenkins – провідний сервер автоматизації з відкритим кодом, який можна використовувати для побудови та автоматизації будь-якого проекту. *Jenkins* обраний для організації всіх функцій, пов'язаних із *SW*, від компіляції вихідного коду до запуску наскрізної автоматизації тестування проти *SUT*. Візуалізація

результатів тесту буде відповідати основним принципам, які надаватиме *Robot Framework.*

П’ять мов програмування функціональні блоки, структурований текст, сходова логіка, послідовні діаграми функцій та список інструкцій. Хоча програмісти прикладних програм досить часто використовують логічну складову завдяки своєму досвіду насамперед в електротехніці, структурований текст та функціональні блоки набувають популярності серед нових розробників. Існує кілька автономних інструментів для підтримки розробки *HMI*, наприклад, *Wonderware* – одна з популярних платформ розробки *HMI.*

Створення програми проходить багато рівнів випробувань перед введенням в експлуатацію. Сюди входять:

1. Модульне тестування (*MT*);
2. Інтеграційне тестування (*IT*);
3. Приймальне тестування (*FAT*).

Модульне тестування еквівалентно модульному тестуванню, коли розробник пише тестові кейси для перевірки функціональності окремих програм управління. В *IT* перевіряється взаємодія між програмами управління. Тестові кейси пишуться вручну. Наскрізне рішення перевіряється на відповідність вимогам *FAT*. *FAT* проводиться в інженерному центрі, де вся система налаштована так само, як і на заводі. Налаштування *FAT* – це поєднання м’яких тренажерів та власне апаратного забезпечення. Клієнти також є частиною виконання *FAT*, де вони ретельно перевіряють повний набір послідовностей операцій та зовнішній вигляд різних екранів операторів (частина *HMI*). Для переходу до наступного етапу замовник повинен затвердити виконання *FAT*. Після *FAT* апаратне забезпечення відвантажується на місце заводу і починається фаза введення в експлуатацію. *SAT* відбувається на етапі введення в експлуатацію.

Для *MT* розробники пишуть модульні тести на основі власного розуміння логіки програм. Тестові кейси для тестування повної заявки походять від вимог та конструктивних артефактів; такі як контрольні розповіді, *PFD* та порядок старту тощо, які описують випадки використання рішення.

2.3. Засоби автоматизованого тестування

Ресурси *HPC* мають складні та динамічні потреби в програмному забезпеченні управління та обслуговування. Користувачі часто хочуть найновіше програмне забезпечення, доступне для їх досліджень або розробок, що обумовлює необхідність частої установки та оновлення. Оскільки кластери найчастіше є спільним ресурсом зі спеціалізованим обладнанням, програмним забезпеченням повинен централізовано керувати системний інженер, щоб переконатися, що воно було оптимізовано для ресурсу та функцій із наявним програмним забезпеченням. В даний час адміністратори вирішують ці складні потреби в програмному забезпеченні за допомогою таких інструментаріїв, як модулі середовища *Lmod*, *Easybuild*, *Spack* та *Puppet*. Однак навіть за допомогою цих інструментаріїв управління програмним забезпеченням у середовищі *HPC* включає ручні та схильні до помилок кроки. Щойно встановлене програмне забезпечення потрібно протестувати та порівняти, щоб користувачі могли оптимально використовувати ресурс *HPC*. Користувачам, що розробляють програмне забезпечення *HPC*, потрібно створити свій код та провести аналогічні тести на коректність та тести. Адміністраторам та розробникам, що розгортають інші орієнтовані на користувача служби, такі як шлюзи, також потрібно створювати та тестувати ці середовища.

Хоча управління програмним забезпеченням *HPC* є складним процесом, воно поєднує багато однакових проблем із розробленням корпоративного програмного забезпечення, яке вимагає надійного оновлення та перевірки відповідно до складних графіків. Протягом останніх кількох років лідери галузі встановили набір практик, що називаються постійною інтеграцією та безперервна доставка (*CICD*) для вирішення цих проблем. *CICD* складається з двох фаз сучасного конвейєру доставки програмного забезпечення, який повністю автоматизує трансформацію змін коду у випуск життєздатного виробничого програмного забезпечення. У цьому контексті *Continuous* наголошує, що повний конвеєр запускається автоматично після кожного коміту. Цей підхід усуває необхідність у складних процедурах злиття та тестування перед випуском, оскільки кожен коміт негайно

перетворюється на програмне забезпечення, яке можна випустити. Автоматизація є критично важливою для цих фаз, оскільки вона гарантує, що кожен коміт підлягає одній і тій же процедурі для побудови, тестування та випуску. Це виключає схильні до помилок ручні кроки у процесі побудови та перевірки програмного забезпечення. Ще одним основним принципом *CICD* є те, що програмне забезпечення проходить кілька етапів автоматизованих тестових наборів, забезпечуючи готовність всього програмного забезпечення, яке проходить збірку.

Хоча *CICD* традиційно зосереджується на життєвому циклі розробки програмного забезпечення, поява таких практик, як *Everything as Code* та нових рішень, таких як контейнери, розширило *CICD*, щоб забезпечити повне програмне середовище, а не лише окремі програми. Розвинені можливості сучасних практик *CICD* для тестування, управління та розгортання повних середовищ програм мають неоднозначні наслідки для стеку програмного забезпечення *HPC*.

Jenkins використовується кількома групами для полегшення автоматизованого моніторингу продуктивності як наукового програмного, так і апаратного забезпечення. Для автоматизації порівняльного тестування на рівні додатків для визначення впливу змін апаратного та програмного забезпечення на продуктивність коду використовують робочий процес *CI*, що працює від *Jenkins* [12]. Використовують *Jenkins* з метою автоматичного порівняльного аналізу продуктивності суперкомп'ютерів *Stampede* та *Lonestar* [13].

Singularity – це відмінне контейнерне рішення для створення , орієнтованих на користувачів, які виробляє *CICD*. Сама ж інфраструктура не призначена для безпосереднього споживання нашими користувачами, а отже, не успадковує той самий набір проблем безпеки, що спонукав до використання *Singularity*. В результаті використанні даного контейнера отримаємо інфраструктуру за допомогою *Docker*, який є дуже зрілим контейнерним рішенням, розробленим для мікросервісів [14]. Використання *Docker* для інфраструктури дозволяє використовувати велику екосистему інструментарію та документації для оркестрації контейнерів, що допомогає розробити надзвичайно портативне рішення

та збільшило різноманітність завдань збірки та тестування, які *Jenkins* міг підтримати.

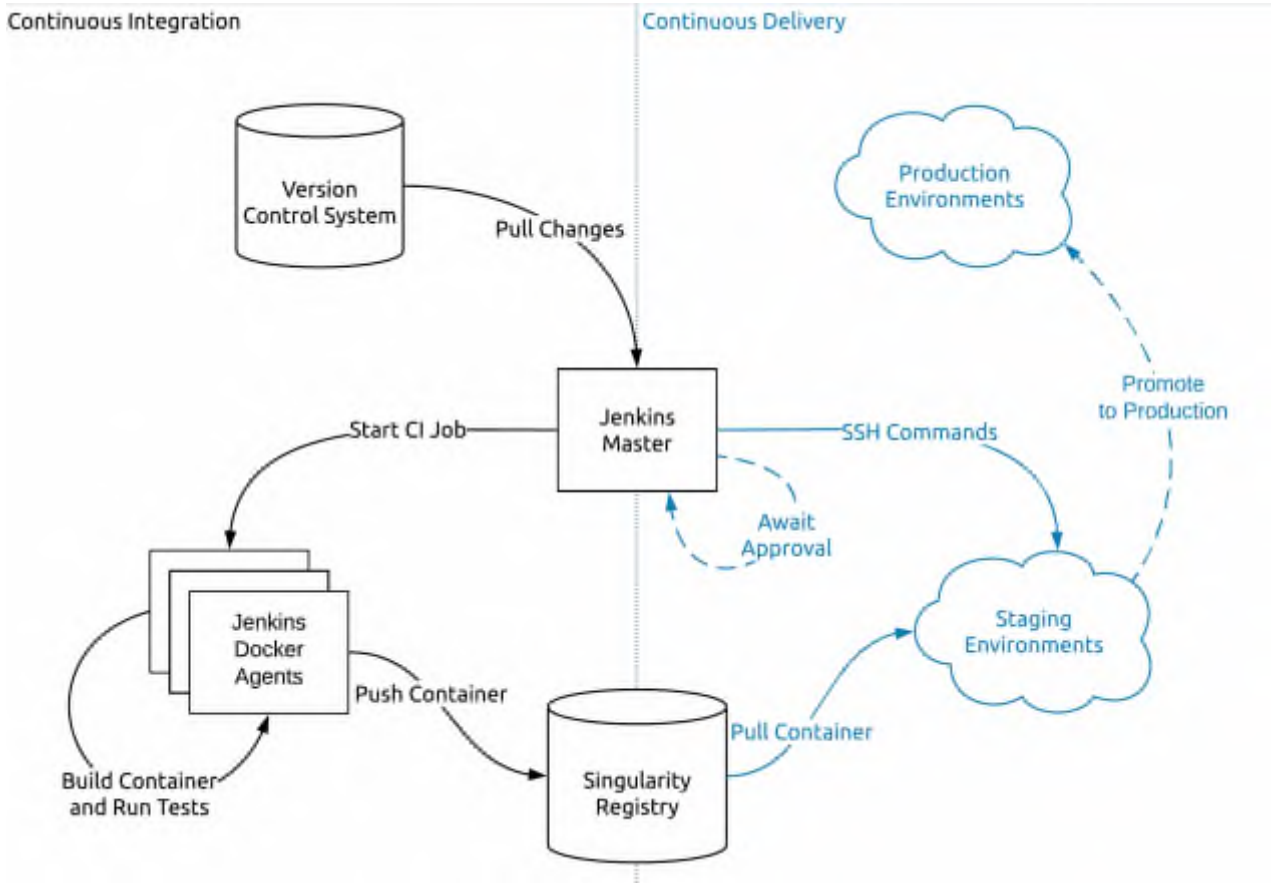

Рис. 2.2. Високорівневий огляд конвеєру, який перетворює зміни коду у версії програмного забезпечення, що розгортаються

Compose – це інструмент оркестрування для багатоконтейнерних програм *Docker*, який визначає конфігурації служби у файлі *YAML* та дозволяє керувати програмами за допомогою простого інструментарію командного рядка. Управління нашою інфраструктурою *CICD* за допомогою *Compose* було цілком природним, оскільки ми вже вкладали в *Docker* програмне середовище майстра та агента *Jenkins*, і це дає можливість можливість легко упаковувати, розповсюджувати та розгортати інфраструктуру в інших місцях ресурсів.

Контейнер *Singularity Linux*, як правило, це ізоляція процесів, яка не завжди допомагає в робочому процесі високопродуктивних обчислень, так як, вона поділяє майже все. Усі процеси, що виконуються всередині *Host*-машини, можуть бачити користувач, який знаходиться всередині контейнера *Singularity*, і всі процеси, що виконуються всередині *Singularity*, такі ж, як і процеси, що виконуються всередині

Host-машини. Але *Singularity* надав деякі параметри конфігурації, щоб у будь-який час забезпечити ізоляцію простору імен, щоб переконатися в його безпеці [18].

2.4. Неперервна інтеграція у *Jenkins*

Jenkins – це платформа для автоматизації з відкритим кодом, яка широко використовується для побудови та розгортання програмного забезпечення. Дану платформу можна розширювати за допомогою плагінів, які забезпечують зручний графічний інтерфейс для створення та редагування процесів збірки і в даний час використовується на декількох сайтах *HPC* як інструментарій автоматизації загального призначення. *Jenkins* підтримує декілька різних стилів конфігурації робочих завдань, і ми обрали метод *Pipeline*, оскільки його легше поставити під контроль версій і зростає популярність, що збільшує кількість доступних плагінів та підтримки. Завдання конвеєра *Jenkins* налаштовуються за допомогою спеціально відформатованого скрипта *Groovy*, який визначає кожен крок процесу побудови та те, де цей крок повинен виконуватися.

Процес клонування сховища коду, що містить рецепт *Singularity*, побудова образу, запуск тестів та публікація через *Singularity Registry* узгоджується між проектами, тому ми розбили процес збірки на спільну бібліотеку. Спільна бібліотека – це механізм виділення компонентів збірки *Groovy* для полегшення повторного використання. Спільні бібліотеки також дозволяють нам редагувати папки побудови *Singularity* незалежно від будь-якого проекту, уникаючи необхідності вставляти файл *Jenkins* у кожен проект, який ми хочемо автоматизувати.

Наприклад, можна скомпілювати свою програму на основі *Java* кожні 20 хвилин або після нового коміту у відповідному сховищі *Git*.

Можливі кроки, виконані *Jenkins*, наприклад, такі:

– виконати збірку програмного забезпечення за допомогою такої системи збірки, як *Apache Maven* або *Gradle*;
– виконати сценарій оболонки;

– архівувати результат побудови;

– запустити тести програмного забезпечення.

Jenkins стежить за виконанням кроків і дозволяє зупинити процес, якщо один із кроків не вдається. *Jenkins* також може надсилати сповіщення у випадку успішної збірки або невдачі.

Jenkins можна розширити за допомогою додаткових плагінів. Наприклад, ви можете встановити плагіни для підтримки побудови та тестування програм *Android* або *iOS*.

CI (*Contentious Integration*) – це набір процесів для автоматизації та інтеграції нових збірок. Усе в *CI* починається від розробників та інших членів команди та інших команд, які вносять зміни у вихідний код програми. Коли команди працюють над тим самим кодом, ризик зламу програмного забезпечення великий.

CI автоматично зв'язує створення пакетів. Постійна інтеграція перевіряє пакети, що підтримує роботу системи як контроль якості. *CI* – процес об'єднання вихідного коду та перевірки того, що програмне забезпечення працює як цілісна одиниця.

На рис. 2.3 представлений процес продуктивності системи *CI* та те, як вони працюють разом.

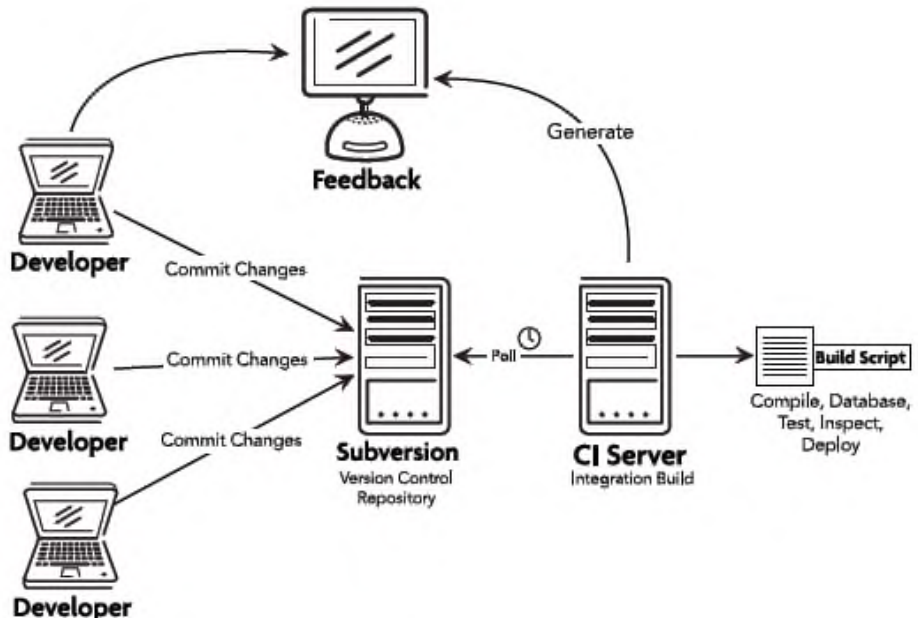

Рис. 2.3. Системні компоненти *CI*

Процес *CI* починається з того, що розробник передає вихідний код до сховища (бази даних для зберігання коду). Після цього сервер *CI* опитує це сховище для змін у машині побудови інтеграції. Час опитування, визначений розробником або тестувальником.

Після фіксації в сховищі сервер *CI* використовує останню копію коду, а потім виконує сценарій збірки. Поки сервер *CI* надсилає результати вказаним членам команди, цю функцію можна також змінити окремо.

CI вимагає чотирьох функцій. *CI*-сервер безперервно опитує зміни у сховищі:

- підключення до сховища контролю версій;
- автоматизований сценарій збірки;
- деякий механізм зворотного зв'язку (наприклад, електронна пошта);
- процес інтеграції змін вихідного коду (вручну / автоматизовано).

Приклад налаштування *Jenkins* для неперервної інтеграції.

Для створення інструментаріїв можна створити власний файл *Docker* із встановленим необхідним інструментами. Наприклад, створити новий образ на основі *Jenkins*:

1. Створити нову директорію;
2. Створити файл під назвою *Testfile* із таким вмістом:

FROM jenkins/jenkins:lts

якщо ми хочемо встановити через *apt*

USER root

RUN apt-get update && apt-get install -y *maven*

ввести назву користувача у *jenkins user*

USER nastia

Завантажити файл *jenkins.war* можна із домашньої сторінки *Jenkins*. З цього файлу можна запустити *Jenkins* безпосередньо через командний рядок за допомогою *java -jar jenkins * .war.*

Щоб запустити образ на сервері *Tomcat*, потрібно скопіювати файл з розширенням *.war* у каталог *webapps*. Після запуску в *Tomcat*, дана установка буде доступна на сторінці в *Jenkins*.

Після встановлення відкрити браузер і підключіться до порта *Jenkins* за замовчуванням: 8080, тому на локальному комп'ютері можна знайти його за такою *URL*-адресою:

http: // localhost: 8080 /

Після виконання та підключення можна встановити плагіни, для цього потрібно натиснути на кнопку «Виберіть/Встановити» та встановити запропоновані плагіни, щоб отримати типову конфігурацію.

Створіти адміністратора можна після встановлення всіх потрібних плагінів.

Управління користувачами у *Jenkins*. Для налаштування користувачів потрібно вибрати «Керувати *Jenkins*», а потім «Налаштування глобальної безпеки», вибрати «Увімкнути безпеку». Найпростіший спосіб – використовувати власну базу даних *Jenkins*. Також можна створити користувача "Анонім" із доступом для читання. Також створіть записи для користувачів, яких ви хочете додати на наступному кроці.

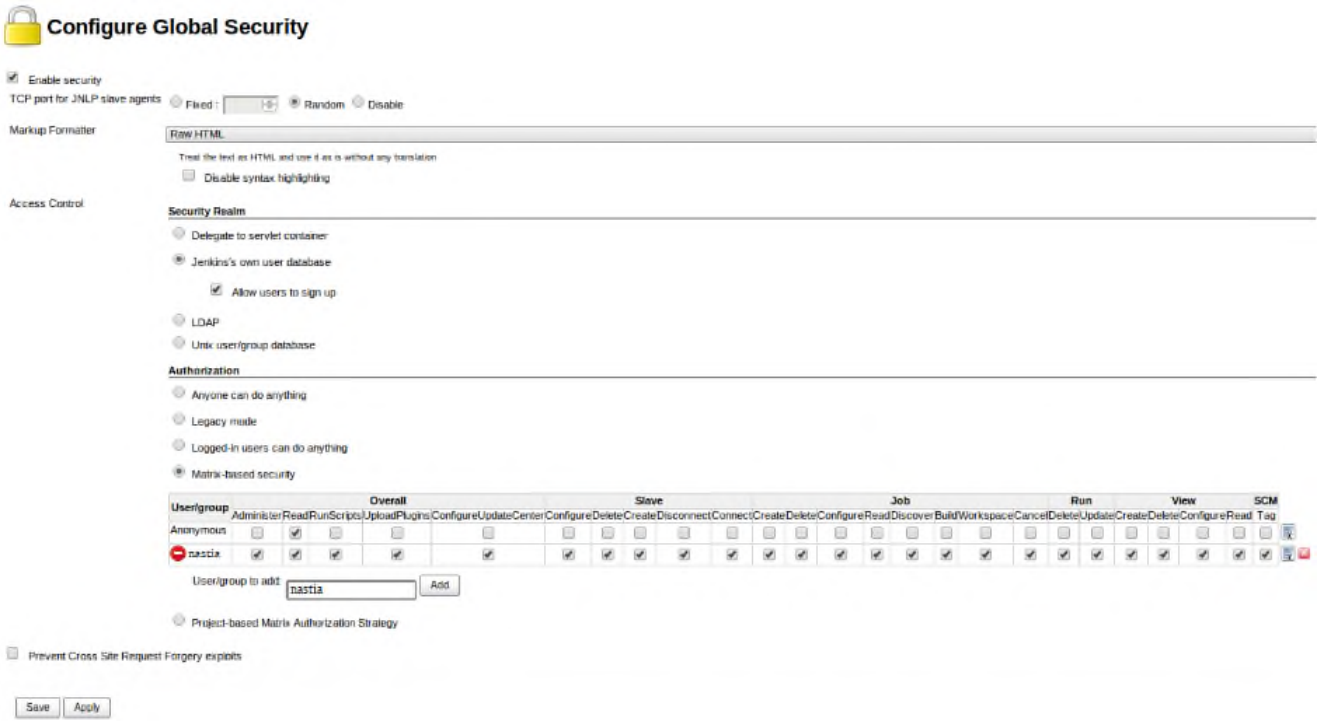

Рис. 2.4. Налаштування безпеки в *Jenkins*

На сторінці входу вибрати «Створити обліковий запис», щоб створити користувачів, яким надали доступ.

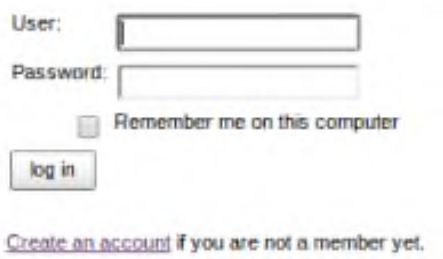

Рис. 2.5. Поля вікна входу в *Jenkins*

Налаштування порта за замовчуванням сервера збірки *Jenkins*. Номер порту за замовчуванням можна змінити у файлі конфігурації за адресою:

sudo vim /etc/default/jenkins

service jenkins restart

Побудова проекту обробляється через робочі місця в *Jenkins*. Виберіть Новий елемент. Потім введіть назву роботи, оберіть проект *Diplom* і натисніть «OK».

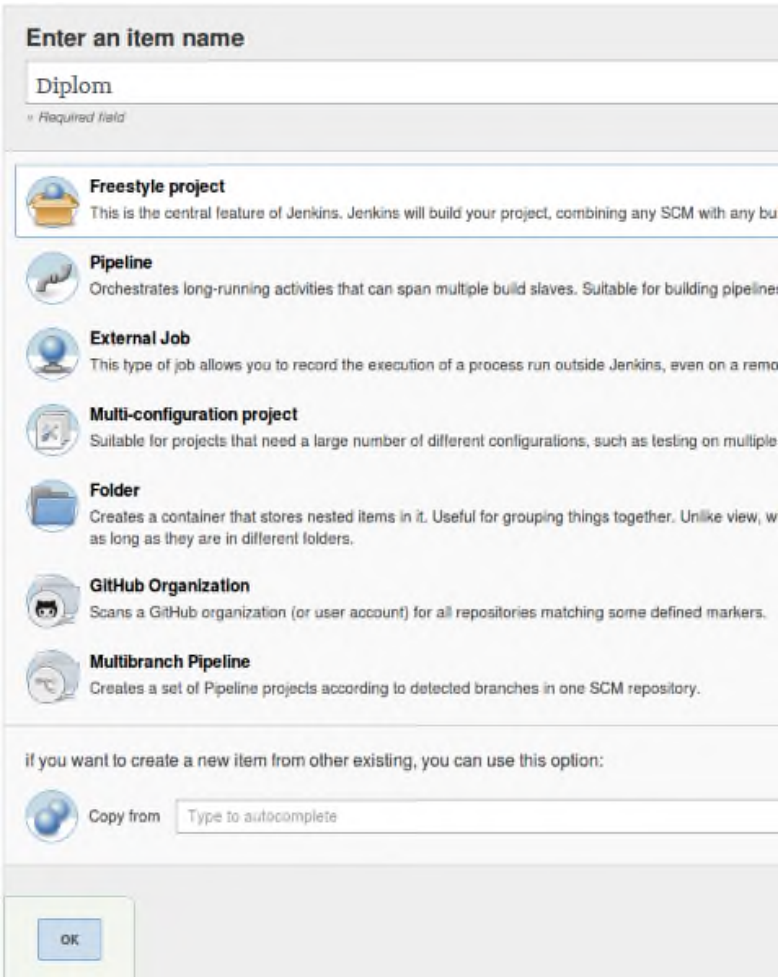

Рис. 2.6. Створення проекту в *Jenkins*

Ввести опис завдання (назва проекту) та налаштувати, скільки збірок слід зберігати та як довго.

Налаштування способів отримання вихідного коду. Якщо використовується, *Git*, ввести *URL*-адресу до сховища *Git*. Якщо сховище не є загальнодоступним, доведеться налаштувати облікові дані. Можна вказати час як часто запускати збірку з сховища *Git*.

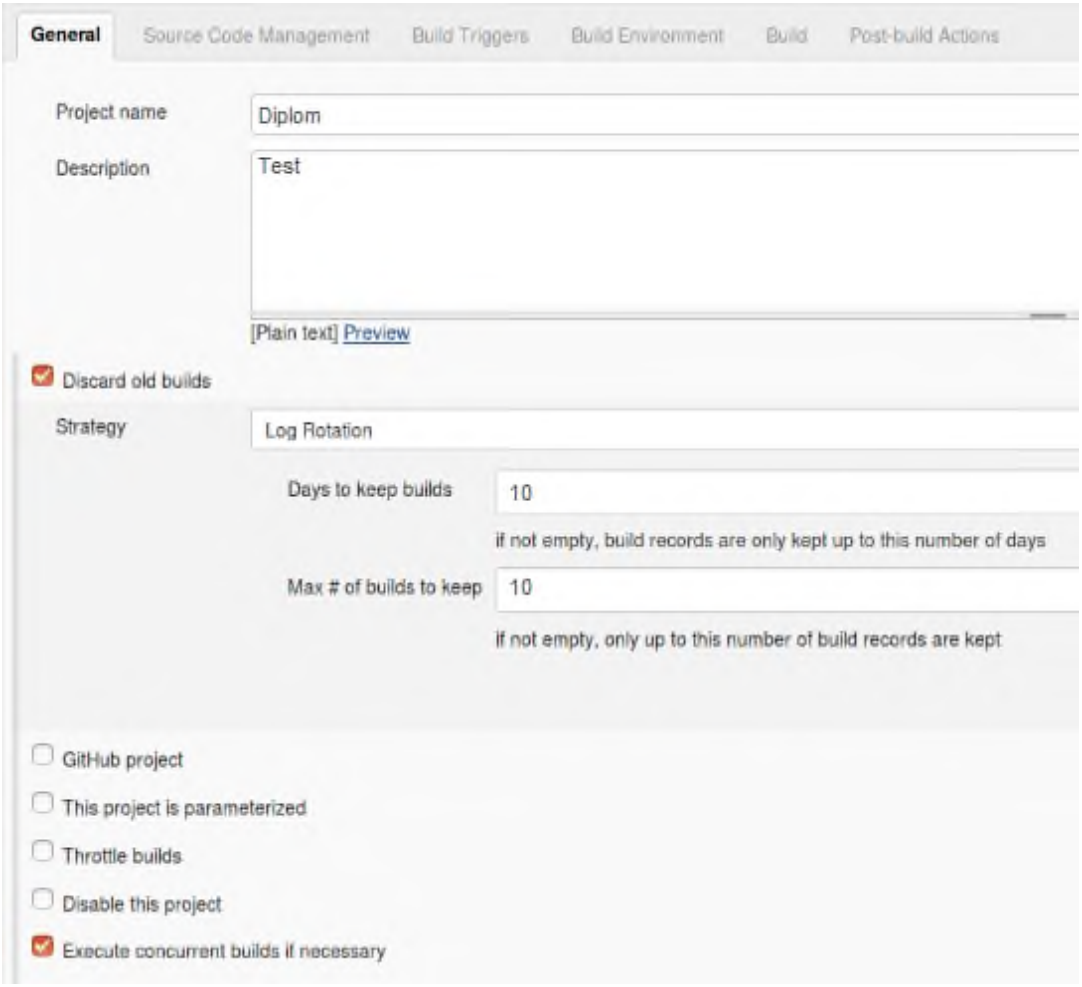

Рис. 2.7. Вікно опису проекту

Після заповнення всіх потрібних полів, натиснути «Зберегти», щоб закінчити визначення завдання. Натиснути «Побудувати зараз» на сторінці завдання, щоб підтвердити, що робота працює належним чином.

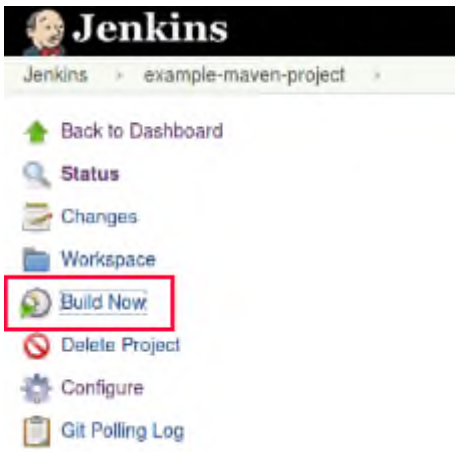

Рис. 2.8. Вікно побудови збірки

Через деякий час робота повинна перейти в зелений або синій колір (залежно від вашої конфігурації), якщо це буде успішно. Клацнувши на завдання, а потім на «Вивід консолі», щоб побачити файл журналу. Тут можна проаналізувати помилки збірки та подивитись з якими параметрами виконувалась програма.

Побудова паралелей. Папки *Jenkins* допомагають узгодити процес побудови проекту. Це робиться шляхом вказівки завдань та порядку їх виконання. Є всі можливі завдання, які паралельно *Jenkins* може зробити. Наприклад, будувати активи, надсилати повідомлення про помилку, надсилати артефакти збірки через *SSH* на сервер додатків тощо.

Jenkins дозволяє визначати папки за допомогою файлу *Jenkins*. Це лише текстовий файл, який містить необхідні дані для виконання *Jenkins*. Він називається *Jenkinsfile* і повинен бути розміщений у кореневій частині проекту.

2.5. Висновки до розділу

Jenkins – це платформа *CI* з відкритим кодом (*OSS*), початковою метою якої є автоматизація процесів побудови та тестування.

Система збірки повністю написана на *Java* і легко розширювана завдяки архітектурі плагінів та точкам розширення, залишеним в об'єктній моделі. Це робить *Jenkins* надзвичайно гнучким інструментом, здатним охопити багато

можливих сценаріїв та вимог завдяки тисячам плагінів, розроблених його величезною спільнотою з відкритим кодом.

Традиційний спосіб управління розробкою програмного забезпечення, відомий як модель *Waterfall*, є логічною та послідовною моделлю, яка має великий недолік, який нелегко застосувати до реальності.

Постійна трансформація в галузі програмного забезпечення, яка вимагає швидкого реагування на зміни, а отже, завдяки командному спілкуванню та співпраці, *Jenkins* розширив свої функціональні можливості, щоб видати роль оркестратора для існуючих та різних ролей, що беруть участь у життєвому циклі продукту. Потреба в управлінні всією конвеєрною системою неперервної доставки вимагала впровадження нових функцій у *Jenkins* з метою сприяння створенню складних робочих процесів, забезпечення простежуваності, скорочення часу виходу на ринок та підвищення продуктивності: плагін сповіщення та механізм робочого процесу плагін, описаний вище, був двома ключовими реакціями *Jenkins* на нові бізнес-очікування.

Однак, хоча цим функціональним можливостям вдається позиціонувати *Jenkins* як концентратор конвеєрів для компакт-дисків, необхідно зробити подальші кроки, щоб зробити *Jenkins* в змозі повністю прийняти революцію *CD*: справді, відсутність найкращих практик для версійних артефактів, що постійно транспортуються, і для пошуку середовище складання створює прогалину, яку потрібно заповнити для подальшого посилення можливостей *Jenkins Continuous Delivery*.

РОЗДІЛ 3
JENKINS – ВІДКРИТИЙ РЕСУРС АВТОМАТИЗОВАНОЇ ПЛАТФОРМИ

3.1. *Jenkins*

Багато програмних додатків було збільшено, а деякі з них оновлені відповідно до вимог бізнесу. Отже, розгортання без автоматизації займе багато часу, і кожного разу, коли ви змінюєте невеликий код, вам слід його розгорнути знову. Тоді як за допомогою *Jenkins* можна автоматизувати процес інтеграції та розгортання одним натисканням кнопки після внесення змін у проект, і це було б корисно в компаніях та університетах, де всі системи підключені за допомогою локальної мережі. Перевагами автоматизації є зменшення схильності до помилок, економія часу, і користувач завжди може знати, як налаштовано програму, якщо він збирається за допомогою автоматизованого процесу. А *Jenkins*, у якому є відкритий код, легко конфігурується, а також підтримує різні операційні системи, тому більша частина роботи з інтеграції виконується автоматично з меншою кількістю помилок. А за допомогою *ansible* можна автоматизувати розгортання додатків, контейнери, інфраструктуру тощо.

Jenkins – це сервер автоматизації з відкритим кодом, який можна використовувати для тестування, побудови та розгортання програмного забезпечення. *Jenkins* виконує завдання з визначеним інтервалом часу, потрібно встановити інтервал часу перед виконанням у *Jenkins*. Результати щодо попередньої збірки можна зберегти, а статус збереженого збірки можна надіслати зацікавленій особі або команді. Результати попередніх збірок за допомогою *Jenkins* також можна зберегти та надіслати статус будь-якому користувачу чи команді.

Саму базову роботу *Jenkins* можна визначити як автоматизацію нелюдської частини життєвого циклу розробки програмного забезпечення. Збірка – це не що інше, як процес перетворення файлів вихідного коду в автономний програмний артефакт, який можна запустити на комп'ютері. Побудова *Jenkins* різними

методами може бути ініційована шляхом визначення механізму або введення в систему контролю версій.

Jenkins підтримує автоматизацію всіх процесів розробки. Також надає розробникам відгуки про те, який останній коміт буде працювати. Це підходить для постійної інтеграції та постійного розвитку.

Jenkins – це серверне програмне забезпечення, встановлене на мережевому комп'ютері та забезпечує *web*-інтерфейс. Адміністратор екземпляра *Jenkins* може встановлювати плагіни для надання певних послуг. Тоді розробники можуть створювати робочі місця, будувати процеси, конвеєри.

Система збірки повністю написана на *Java* і легко розширювана завдяки архітектурі плагінів та точкам розширення, залишеним в об'єктній моделі. Це робить *Jenkins* надзвичайно гнучким інструментом, здатним охопити багато можливих сценаріїв та вимог завдяки тисячам плагінів, розроблених його величезною спільнотою з відкритим кодом.

Конвеєр *Jenkins* підтримує інтеграцію та реалізацію розгортання за допомогою різних плагінів. Тоді плагіни повинні бути встановлені відповідно до програми. У *Jenkins* всі підзавдання написані як етапи сценарію.

Плагін *Jenkins* – це програмне розширення для *Jenkins*, що забезпечує підтримку певного завдання. Спільнота *Jenkins* надає понад 1500 плагінів для автоматизації будь-яких завдань збірки, і вона легко налаштовується [16]. Завдяки всім доступним плагінам, *Jenkins* може будувати проекти, проводити статичний аналіз, розгортати продукти тощо. Плагіни відсортовані за 5 категоріями:

- платформи;
- інтерфейс користувача;
- адміністрація;
- управління вихідним кодом;
- управління збіркою.

Плагін *Platform* забезпечить середовище для тестування вашого проекту. Наприклад, плагін *Android Emulator* пропонує тестові набори для проектів *Android*. Плагін управління вихідним кодом може надати *Jenkins* можливість отримувати

код сховища *GitHub, GitLab* або *Bitbucket*. Нарешті, плагін керування збіркою надасть інструменти для аналізу результатів збірки.

Методологією, якою ми керувались, є безперервна інтеграція за допомогою *Jenkins*, а потім використання інструменту *ansible* для розгортання.

Традиційний спосіб управління розробкою програмного забезпечення, відомий як модель *Waterfall*, є логічною та послідовною моделлю, яка має великий недолік, який нелегко застосувати до реальності: однією з ключових концепцій цієї моделі є визначення заморожених вимог в попередньому та ізольованому стані. Цей крок започаткує водоспади наступних етапів, на припинення яких можуть знадобитися тижні чи місяці, включаючи проектування, впровадження, перевірку та розгортання. Протягом часу, необхідного для проходження цілого водоспаду сходів, вимоги, ймовірно, змінюються внаслідок нових конкурентів або просто до вдосконалення початкового проекту.

Процеси безперервної інтеграції – це практики розробки програмного забезпечення, де кілька зайвих завдань виконуються кілька разів. У робочих процесах *CI* розробники безперервно виробляють вихідний код, висуваючи нові зміни до вихідного сховища (наприклад, *Git, Mercurial* або *SVN*). Потім продукт повинен бути перевірений на декількох рівнях (модульне тестування, інтеграційне тестування) та якості (регресійне тестування, приймальне тестування) перед етапом побудови, створюючи кінцевий продукт.

Номер порту за замовчуванням *Jenkins* буде 8080, і його можна буде змінити за потреби. Отже, після встановлення *Jenkins* починайте створювати новий елемент у *Jenkins*, і ім’я елемента має збігатися з папкою проекту в каталозі робочої області, якщо потрібно інтегрувати та розгорнути з існуючої локальної системи або також можна використовувати сховище *GitHub* куди можна завантажити програму. Після цього написати скрипт у *groovy* для створення папки збірки, автоматично додайте модулі, якщо такі є для програми. Потім після встановлення *jfrog artifactory*, за допомогою якого можна інтегрувати більшість інструментів безперервної інтеграції, щоб надати повне автоматизоване рішення для відстеження артефактів від розробки до виробництва. Потім створити нове сховище в *jfrog artifactory* та

додати новий етап у сценарії *Jenkins*, де папка збірки програми може бути автоматично заархівована та завантажена в *jfrog artifactory*. Визначити змінну з назвою версія у сценарії *Jenkins*, щоб при кожному запуску версія буде створюватись новий *zip*-файл разом зі своєю версією у створеному сховищі.

Після автоматизації побудови та розміщення додатків час, який зайняв, значно скоротився. Існує багато інструментів для побудови, але причина використання *Jenkins* полягає в тому, що він безкоштовний, легко доступний і має безліч плагінів. Це робить більшу частину процесу інтеграції автоматичним. І сам *Jenkins* також автоматично викликає процес розгортання, за допомогою інструменту можна створити розгортання *ansible*. Незабаром після деяких змін, здійснених у програмі, сервер *Jenkins* виявить зміни у вихідному коді програми, і коли відновлення буде виконано знову, він почне нову побудову. Для розгортання *ansible* спочатку потрібно переконатися, що машина або сервер, на яких встановлено *ansible*, повинні бути доступними через *SSH*. Інші інструменти, такі як *Puppet* і *Chef*, можуть бути використані для автоматичного розгортання, але потрібно встановити всі агенти для налаштування, де, як і в *ansible*, можна використовувати *SSH* для просування деяких конфігурацій. Після того, як *Jenkins* починає будувати новий процес, можна бачити кожну стадію, яка є кожною підзадачею, і можно дізнатись, що кожна стадія була успішною.

Після автоматизації в *CI/CD* час на розгортання окремих додатків значно скоротився, що також значно покращить обслуговування програм. Автоматизація також чудово використовується для програм, які дуже часто вносять зміни. З моменту часу команда розробників фіксує код у *Git* або будь-якому іншому сховищі до моменту запуску програми, кожен етап конвеєра автоматизується, що скорочує час і зусилля на перегляд коду, а також може будувати процеси багаторазово після внесення змін до програми.

Завдання *Jenkins* можуть запускатися з різних джерел: подія сховища джерел, за часом, за запитом. Потім, раз на годину, день або тиждень, більші тестові набори та процес побудови можуть бути заплановані для забезпечення моніторингу стану за допомогою можливості сповіщення *Jenkins*.

Хоча *Jenkins* можна використовувати для побудови автоматизації процесів, його також можна використовувати для автоматизації модульних тестів. Наприклад, коміт у вихідному сховищі може викликати тестові завдання *Jenkins* для запуску різних рівнів тестування з різними наборами тестів, щоб надати швидкий звіт у разі виявлення дефекту.

Завдяки декільком доступним плагінам, *Jenkins* може запустити багато наборів тестових модулів для численних тестових середовищ. Однією з потужностей *Jenkins* є його здатність запускати тести для різних середовищ тестування, як різних платформ (*Windows, Linux, MacOS, Android, iOS* та інші). Ще одна хороша сторона автоматизації – це можливість плагінів додавати *Unit Test Generation* (*UTG*) для нещодавно доданого коду, що може заощадити багато часу для тестових команд.

3.2. Плагіни в *Jenkins*

Jenkins має кілька важливих переваг: користувальницький інтерфейс простий, інтуїтивно зрозумілий, візуально привабливий і з дуже низькою кривою навчання; він надзвичайно гнучкий і легко адаптується до різних цілей; і він має кілька доступних плагінів з відкритим кодом, які охоплюють широкий спектр функцій, таких як системи контролю версій, інструменти побудови, показники якості коду, сповіщувачі збірки, інтеграція із зовнішніми системами та налаштування інтерфейсу користувача *Jenkins* дотримується суворого керівництва політичними правилами.

Ідея використання плагінів для модульної функціональності не нова. Однак є небагато задокументованих випадків використання плагінів як методу, що дозволяє набору тестувати доступ до *API*, а також додавати або змінювати *API* без істотних перезаписів. Цей підхід дозволяє розробити основу тестування для багатьох служб, що використовують одну мову або сервер. Що ще важливіше, плагіни також дозволяють розподілити кодування між кількома розробниками, щоб прискорити тестування на платформі.

Під час розробки плагіна кожен автор обирає тип методології, який буде використовувати плагін. Може бути кілька плагінів із різними типами методологій, які перевіряють один *API*. Щоб збільшити корисність плагіна, він також може завантажувати тестові вхідні значення з файлу, що дозволяє швидко завантажувати багато тестів без перезапису плагіна.

Оскільки плагіни незалежні, вони можуть працювати як програми міні-тестування на що завгодно (окремий *API*, певний метод або клас). Однак теоретично плагін може працювати над багатьма *API* в певний час і навіть перевірити їх взаємодію між собою.

Google App Engine дозволяє створювати, розгортати та розміщувати *web*-додатки в інфраструктурі *Google*. Розподіляючи завдання між різними серверами, інфраструктура *Google* забезпечує автоматичне формування трафіку на вимогу та балансування навантаження для певної програми. В результаті багато функцій хмарних обчислень поширюються серед широкого кола користувачів. Різні *API* абстрагують програми від цієї різноманітної інфраструктури, а також дозволяють відстежувати використання ресурсів на платформі. *API* надають доступ до різноманітних служб, включаючи сховище (сховище даних, *blobstore*), управління користувачами, офлайн-обробку (черга завдань, *cron*) та *web*-запити (*URLFetch*). Також був наданий комплект для розробки програмного забезпечення (*SDK*), що складається з інструментів для локальної розробки програм на клієнтських системах користувача *Linux, MacOS* та *Windows*. У *SDK* сервер додатків для розробки імітує підтримувані *API Google App Engine* і дозволяє запускати та налагоджувати розроблені програми перед розгортанням. На момент цього розслідування *Google App Engine* підтримував *API* двома окремими мовами (*Python* та *Java*). Мова *Go* підтримувалася в експериментальній формі.

Основне завдання цього дослідження полягає у застосуванні методів функціонального тестування, а не тестуванні продуктивності чи масштабованості. Отже, сервер додатків для розробки *Java* був обраний як основний засіб як для розробки тесту, так і для виконання тесту. Сервер *Java* забезпечує необмежений час роботи для запитів сторінок, що дозволяє плагінам працювати необмежено

протягом тестування. Однак враховано також тривалість роботи плагінів, щоб додаток *App Engine* міг бути розгорнутий у хмарі, де він також міг протестувати, дотримуючись 30-секундного терміну відповіді на запит. Цей підхід також дозволив порівняти результати тестування між платформою розробки та хмарою, щоб переконатися, що вони ідентичні з точки зору програми клієнта.

Загальна структура фреймворку представлена на рис. 3.1 Структура тестування у цьому дослідженні використовує *Google Web Toolkit* (*GWT*) для створення користувальницького інтерфейсу та функціональної логіки, таких як типи плагінів для запуску. Однак додаток не тестує жодного класу *GWT*, оскільки *GWT* є окремим проектом від платформи *App Engine*. Кожен відображений плагін буде висвітлений в окремому підрозділі.

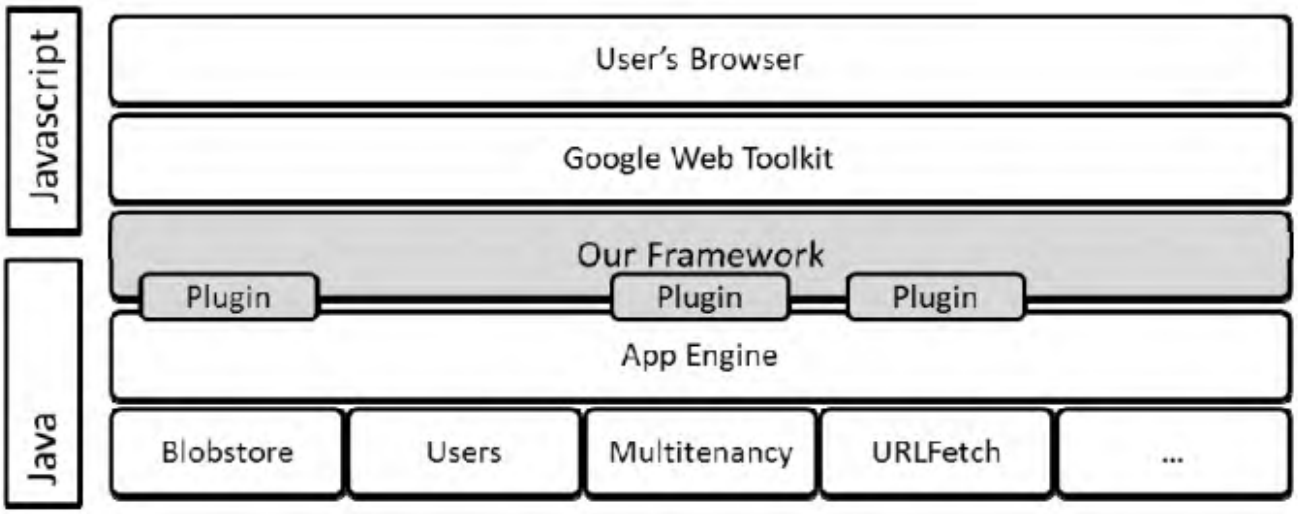

Рис. 3.1. Загальна структура фреймворку для тестування *API*

Плагін за замовчуванням використовується для тестування нашого фреймворку, а не конкретного *API*. Плагін за замовчуванням не виконує жодної реальної функції, за винятком перевірки надійності завантаження, тестового запуску та реєстрації. Враховуючи свою простоту як плагін, плагін за замовчуванням може бути використаний для усунення несправностей плагіна, фреймворку або платформи *API*. Якщо є якісь проблеми, але плагін за замовчуванням завантажується і виконується правильно, тоді проблема може бути як з іншим плагіном, так і з платформою. Пошук джерела проблеми вимагатиме додаткового тестування, але тепер цей фреймворк можна вважати досить надійним.

Плагін *URLFetch* працює, використовуючи ієрархію об’єктів, які необхідно створити для завершення веб-запиту. З цієї ієрархії об’єкт вибирається з початку, середнього та кінця ланцюжка викликів. Метою було виявити будь-які помилки, звужуючи їх позицію, тестуючи об’єкти по всій ієрархії. Цей метод також функціонує як ретельна перевірка, а не покладаючись на об'єкт найвищого рівня, щоб просто отримати правильну відповідь.

URLConnection – це *Java*-клас високого рівня, який надає доступ до параметрів, коли об'єкт *URL*-адреси використовується для отримання даних. Атрибути об’єкта використовуються для формулювання запиту, який в кінцевому підсумку зроблено. Дані повертаються та стають доступними за допомогою додаткових методів від об’єкта. Для об’єкта *URL*-зв’язку вибираються властивості, які потребують попередніх запитів. Потім ці параметри блокуються після надсилання запиту. Тому запит налаштовується по-різному, а не намагається зчитувати дані назад у різних форматах.

URLFetch – це двері, через які повинен проходити весь трафік запитів внизу стека викликів *API URLFetch*. *URLFetch* – це простий клас, який управляє лише кількома властивостями, але це одна точка відмови для *API URLFetch*, і, отже, варта перевірки. З цієї причини було обрано наступні параметри, пов’язані із запитом:

- дозволити/заборонити скорочення – дозволити (чи ні) безшумне скорочення даних над розміром буфера;
- (не) дотримуватися переспрямувань – дозволити або заборонити автоматичне відстеження переадресаційних відповідей;
- *setDeadline* – встановити час у секундах, на який запит повинен відповісти, перш ніж він буде перерваний.

У випадках, коли є небулеве введення, наприклад *setDeadline*, кожному тестовому випадку було дозволено вказати подвійний вхід, який був доданий до поточного системного часу.

Проведено чотири тестові набори. Перші два набори були вбудовані в тестовий плагін *URLFetch*, щоб перевірити наявність активного мережевого

підключення, перш ніж переходити до фактичних тестів. Цей метод також є найближчим до того, що програміст використовував би для простого захоплення *web*-сторінки, тому він допомагає гарантувати, що на найвищих рівнях *API* дані повертаються, як очікувалося.

Для *URLConnection* наданий текстовий документ для побудови масиву, який давав вказівки тестовому плагіну щодо використовуваних параметрів та порядку, в якому вони були представлені. На рис. 3.2 містить перелік восьми параметрів (варіантів) тестування та їх можливих значень. Шість параметрів мають два різних значення, один параметр - три значення, а один параметр - чотири значення. Для небулевих входів, таких як *setIfModifiedSince*, була створена 3-варіантна система, заснована на минулому, теперішньому та майбутньому. Іншими словами, тест би сформулював значення часу, яке відповідало б будь-яким критеріям порівняння і пройшло або не пройшло, як зазначено в документації. Для *SetRequestMethod* існує безліч заголовків *HTTP*, крім *GET* та *POST*; однак для тестування використовувались лише два. Кожен тестовий випадок включає очікуваний результат, який випливає з досвіду роботи з веб-програмами та документацією *Java*. Кожен результат має форму очікуваного коду відповіді *HTTP*, за винятком випадків, коли конфліктна конфігурація може спричинити помилку. У цьому випадку очікується виняток *Java*. Тестування може зайняти багато часу, і перевірити всі ці комбінації нереально. Ми зосереджуємось на створенні меншої кількості кращих тестових кейсів, і замість вичерпного тестування ми використовуємо тестування по парах [17].

	Назва	**Опис**	**Значення**
1	*setAllowUserInteraction*	Запит у користувача облікових данних, якщо потрібно, за допомогою діалогового вікна графічного інтерфейсу	0,1
2	*setDoInput*	Сокету, який буде приймати дані	0,1
3	*setDoOutput*	Сокету, який буде надсилати дані	0,1
4	*SetIfModifiedSince*	Кешована версія, якщо вона доступна та дійсна	За весь час
5	*SetUseCaches*	Кешування документів, які відповідають певним вимогам для швидшого доступу	0,1
6	*SetRequestMethod*	Метод запиту *HTTP* для надсилання даних, кодованих за допомогою форми та *URL* - адреси	GET, POST
7	*Server Response*	Примусова відповідь від сервера за допомогою певного коду відповіді	200,307,404
8	*Download*	Сервер надіслає заголовок вкладення файлу на сервері	0,1

Рис. 3.2. Список тестових параметрів для *URL*-підключення

Багатоканальність більш відома як простори імен. Встановивши простір імен, підтримуючі *API* будуть розділяти дані таким чином, що навіть об'єкти з однаковими первинними ключами будуть зберігатися окремо в різних наданих сховищах даних. Доступ до просторів імен здійснюється через об’єкт *NamespaceManager*, який відповідає за встановлення, отримання та перевірку просторів імен. Цей менеджер є об’єктом, який ми вибрали для тестування. Перевірка відповідності рядка регулярному виразу є складною ситуацією. Ми вирішили завантажити вхідні тестові кейси у файл, щоб можна було спробувати багато різних методів для створення тестових кейсів. Це було зроблено приблизно так само, як і з *URLConnection* та *URLFetch*.

Крім того, навіть незважаючи на те, що плагін розроблений, щоб зосередитись виключно на багатонаціональності, варто зазначити, що процес буде включати також тестування різних *API*. Беручи до уваги, що суттєвою метою багатосторінкової діяльності є вплив на те, як інші *API* зберігають свої дані, важливо включити *API*, що впливають на тестування, щоб забезпечити належне функціонування багатонаціональної компанії.

При тестуванні *Blobstore* дотримується раніше встановленої практики відокремлення тестів від власне коду плагіна. Першим кроком є встановлення умов, яких *Blobstore* повинен дотримуватися. Було вирішено, що *Blobstore* повинен мати можливість приймати завантажені дані та повертати їх без змін. Можливість виявити, чи змінюються подальші завантаження від оригінального завантаженого файлу, буде мірою успіху. З цією метою був створений алфавіт мутацій, визначений плагіном як операція, яка змінює дані, але яка також є оборотною. У цьому випадку використовувалися такі речі, як серіалізація або застосування алгоритму стиснення без втрат.

Ключові операції:

- *heap* створює новий елемент і копіює всі дані;
- *stack* передає значення через рекурсивний виклик;
- *serialize* перетворює поточне сховище даних у рядок, використовуючи байти поточного стану;

– *bytebuffer* перетворює поточний стан у байт-буфер.

Через обмеження в часі ми обмежили процес простими операціями. Однак майбутні реалізації можуть містити більш складні алгоритми стиснення або хешування.

З кожною операцією плагін зберігає копію маніпульованих даних, отримує нову копію, а потім намагається продублювати виконані операції, отримуючи той самий результат. Помилка операції збігатися з попередніми даними на тому самому кроці означає, що дані не були отримані належним чином. Контрольна сума або хеш - це інші варіанти, які було б простіше реалізувати; однак, оскільки загальною метою було знайти помилки будь-яким можливим способом незалежно від *API*, має сенс використовувати додаткові методи, які додають складності процесу під час тестування різних операцій. У плагіні *Blobstore* використовується функція нашого фреймворку, яка називається заготовки. Це запити на додаткову інформацію перед запуском плагіна. У цьому випадку плагін вимагає завантаження додаткового файлу, щоб він міг працювати. Цей файл може бути будь-якого типу файлу на вибір користувача. В даний час ми не тестуємо розмір файлу та не перевіряємо, чи підпадає він під якісь обмеження.

Плагін *Test Generation* спеціально розроблений для аналізу даного класу *Java,* вибору найкращого із зареєстрованих наборів модулів методології розробки тестів та надання користувачеві отриманих наборів тестів. На даний момент плагін тестового покоління входить до складу нашого набору тестів як плагін, але з причин, згаданих пізніше, він повинен стати самостійним додатком.

Плагін починається зі створення списку файлів класів, які він виявляє у пакетах *jar*. Ці пакети *jar* містять усі файли класів для різних *API* та представляють їх фактичну базу коду. Користувачеві представлений цей список через преформу, і звідти користувач може вибрати цілі *API* або файли одного класу для обробки. Потім плагін бере цей список і витягує із файлу лише вибрані файли класів, де потім завантажує класи в *JVM* за допомогою завантажувача спеціальних класів. Крім того, плагін завантажує кожну методологію генерації тестів як модуль. Потім він проходить кожен метод із кожного класу, використовуючи бібліотеку

відображення *Java*, і передає його кожному модулю генерації тестів. Модулі генерації тестів обчислюють кількість різних тестів, які вони б створили для конкретного методу, і повертають це плагіну, який потім вирішує, який модуль виробляє найменшу кількість тестів. Крім того, якщо методологія генерації тестів не підходить для генерування для певного методу через параметри або неймовірно велику кількість тестових випадків, вона може відмовитись від розгляду в плагіні генерації тестів. Якщо вибрано методологію генерації тестів, її знову передають через метод і дозволяють генерувати тестові кейси, які він повертає до плагіна, який форматує тестові набори як вихідні дані, а потім повертає їх до графічного інтерфейсу користувача. Потім користувач відповідає за відокремлення різних тестових випадків та форматування їх для інших інструментів. Слід зазначити, що генерація тестів не дає значення оракула, і їх залишає користувач. Код не може зрозуміти наміри інших кодів без додаткової розмітки, тому неможливо надати оракули, не поєднуючи файли класу з якимсь іншим джерелом вхідних даних, яке декларує намір програміста таким чином, що плагін може обробити та перевірити.

Цей підхід здається досить простим, але є багато причин, чому плагін генерації тестів повинен стати автономним додатком, щоб нормально функціонувати. По-перше, *jar API* для *App Engine* перевищує межу завантаження. По-друге, *JVM* сприймає класи, завантажені різними завантажувачами класів, як абсолютно різні, навіть якщо вони походять з одного файлу класу і є абсолютно подібними. Той факт, що *Test Generation* – це плагін, означає, що він завантажується нашим завантажувачем класів у фреймворк. Тут, коли він намагається завантажити модулі, він робить це за допомогою завантажувача нового класу, а це означає, що він не може використовувати функції оригінального фреймворку без певної складності. По-третє, навіть якби цей плагін був адаптований для належної роботи на платформі *Live App Engine*, він потрапляв би під дію обмеження в 30 секунд, і він не зможе завершити роботу під час обробки великої кількості файлів класів. Нарешті, результати повинні бути об’єднані в один файл журналу, щоб повернути їх користувачеві. Було б набагато краще, якби модуль міг просто записати необхідні для користувача файли на локальний диск.

3.3. Побудова проектів у *Jenkins*

Jenkins надає безліч функцій та конфігурацій. Після нової інсталяції та підключення до сервера відбувається налаштування базових функцій з уже встановленими основними плагінами. В ідеальній установці демон *Jenkins* працював би на окремому обладнанні, але все одно може підключатися до інших вузлів кластера. При запуску *Jenkins* на обчислювальному вузлі і переадресували порт *HTTP* до вузла входу. Потім порт можна тунелювати через *SSH* від вузла входу до локальної машини, щоб отримати доступ до *web*-інтерфейсу в безпечному та зашифрованому вигляді. На сторінці управління *Jenkins* перелічено ряд параметрів конфігурації, починаючи від адміністрування привілеїв користувача до доступу до інтерфейсу командного рядка. Ми завантажили конкретні плагіни, такі як “Рольова стратегія”, “Галерея зображень”, “Копіювати артефакт” та кілька інших для нашого конкретного екземпляра *Jenkins*. Оскільки цей джгут включає велику кількість базових тестів, ми вирішили створювати проекти збірки *Jenkins*, використовуючи багатоконфігураційні та багатопрофільні проекти. Це дозволяє запускати проекти збірки з різними наборами параметрів та групувати окремі проекти збірки разом за повною ієрархією, тим самим запускаючи кілька тестів, будуючи один проект.

Далі потрібно було внести зміни в безпеку. Плагін стратегії, що базується на ролях, надав спосіб дозволити авторизацію *Jenkins* на основі ролей, наданих певним користувачам. Адміністраторам надається повний доступ, тоді як анонімним користувачам просто надається доступ для читання. Ще однією конфігурацією, яку потрібно було зробити, було дозволити *Jenkins* налаштувати *SSH*-ключі для користувачів та ввімкнути автентифікацію *Jenkins-SSH* для виконання команд із командного рядка, використовуючи *SSH*-ключі для автентифікації замість маркерів *API*. Командний рядок зручний для користувачів, які створюють сценарії для автоматизації рутинних завдань або масового оновлення. Наприклад, набір *PAW* має безліч індивідуальних тестів, але кожен проект *Jenkins* по суті однаковий, за винятком виклику іншого виконуваного файлу

PAW. Було написано сценарій *Bash*, який включає командний рядок *Jenkins* для створення всіх різних проектів збірки *PAW* на відміну від натискання веб-графічного інтерфейсу для того самого завдання. Крім того, потрібно було налаштувати параметри протоколу *Content-Security* у *Jenkins*, щоб забезпечити належну роботу вбудованих інструментів візуалізації.

Окрім *Jenkins* в цілому, що потребує конфігурації, окремі проекти збірки вимагали специфікацій для відповідного запуску. Більшість індивідуальних проектів збірки мають однаковий загальний макет, оскільки виконується сценарій оболонки, архівуються файли даних або журналів, а для перегляду *Jenkins* публікуються файли *JUnitXML*. Існує багато інших конфігурацій, таких як тригери збірки, середовище збірки або управління вихідним кодом. Одним з параметрів примітки були параметризовані параметри збірки, які дозволяють автоматично запускати проекти збірки з декількома конфігураціями параметрів, заданих користувачем [25].

Загальні кроки для впровадження типових додатків або тестів у *Jenkins* перелічені нижче. Це була груба процедура, за допомогою якої обговорені раніше апартаменти були пристосовані для використання з *Jenkins*.

Створення сценарію для програм слід спочатку створити сценарій, де процеси побудови та запуску формалізовані у повторюваній та послідовній конфігурації.

Початкова конфігурація проекту збірки як для одиночної, так і для багатозадачної збірки потрібно мати шаблон *XML* конфігурації збірки, адаптований для конкретного випадку. Спочатку це робиться шляхом створення завдання або з *web*-інтерфейсу, або з командного рядка *Jenkins*. Різні конфігурації збірки наведені нижче.

Крок збірки – це серце кожного проекту збірки, де запускається сценарій, створений раніше. У наших випадках ми завжди мали цю оболонку запуску та сценарії *Java* для запуску завдань *Slurm*.

Параметри збірки, архівування, звітування про тести, очищення робочої області тощо. Інші кроки збірки, такі як встановлення параметрів збірки,

архівування результатів, створення звітів про тестування та очищення робочої області, також визначені у сценаріях *XML* конфігурації проекту збірки.

Розширення для додаткових проектів збірки: Після налаштування початкового проекту збірки для будь-якого набору додаткові збірки можуть бути додані шляхом редагування конфігурації завдання *XML*. Конфігураційний *XML* можна завантажити з сервера *Jenkins* через командний рядок, а потім відредагувати та використовувати для створення кількох інших проектів збірки.

Корисна візуалізація: після того, як усі проекти збірки, необхідні для набору, налаштовані як проекти *Jenkins*, їх можна згрупувати для запуску одночасно з багатозадачними проектами верхнього рівня. Багатозадачні проекти також можна згрупувати разом, що дозволяє одному проекту збірки запускати весь пакет. За допомогою графічного інтерфейсу користувача *Jenkins* або командного рядка також можна створювати та налаштовувати подання інформаційної панелі, щоб відображати статус завдання та історичні результати тестів збірки наборів, крім того, що дозволяють вбудовувати власні зображення або графіки.

Модульне тестування є актуальним на всіх рівнях розробки програмного забезпечення, і воно, як правило, розробляється разом із виробничим кодом, але не вбудовується в остаточний програмний продукт. Він перевіряє конкретний об'єкт коду на його продуктивність. Постійне інтеграційне тестування, набір різних інструментів для автоматичного виконання модульного тестування, як тільки розробник додає до нього нові функції. Фреймворк допомагає розробникам уникнути зламаного програмного забезпечення.

Jenkins – це заснований на *Java* відкритий вихідний код, сервер безперервної інтеграції (*CI*), або інструмент безперервного моніторингу збірки для завдань, що виконуються неодноразово, також відомий як засіб тестування автоматизованих тестів. Він постійно відстежує помилки розробки на ранній стадії розробки програмного забезпечення.

Jenkins зосереджується на двох основних завданнях: перше – це тестування, а друге – безперервне будівництво проектів.

Jenkins – одна з найбільших систем побудови *CI* з відкритим кодом. Завдяки своїй гнучкості, сотням плагінів та стійкості до різних типів систем. Це дозволяє різні середовища та працювати з багатьма зацікавленими сторонами процесу.

Поєднана група тестових випадків у журналі ефективності. У властивостях збірки визначені характеристики продуктивності та спеціальні вимоги.

При створенні вбудованого *Jenkins* всі деталі продуктивності повинні бути зазначені в журналі продуктивності. Крім того, ім'я та бібліотека, звідки буде взято вихідний код.

Завантажив останню *Java* та встановив останнє середовище виконання *Java* (*JRE*) з *web*-сайту інсталятора *Java*. *Jenkins* – це *web*-програма на базі *Java*, а *Java Runtime Environment* – це простота для її запуску.

За допомогою сервера додатків *Java Tomcat* відкриє *Jenkins* у *web*-браузері. Встановлено за допомогою офіційного *web*-сайту *tomcat.apache.org*.

Є програма з відкритим кодом *Telnet* і *Secure Shell* (*SSH*), вона використовується для доступу до даних з одного комп'ютера на інший через мережу. Він створює зв’язок між комп’ютерами операційної системи *Windows* та іншими операційними системами, наприклад *Linux*, за допомогою ключа хосту *IP*-адреси. Це вимагає певного підключення до інтернету.

Secure Shell (*SSH*) призначений для захисту вашого комп’ютера від мережевих атак. *Secure Shell* повторно підключає кожен хост-ключ у реєстрі серверів *Windows*, підключеному раніше. Щоразу під час підключення до сервера *SSH* перевіряє, що використовується той самий ключ хосту, якщо не, надсилає сповіщення.

Використовується в цій роботі для легкого доступу до сервера *Jenkins* з робочого комп’ютера, оскільки *Jenkins* працював на офіційних приватних серверах компанії.

Jenkins можна встановити кількома способами – з офіційного *web*-сайту *jenkins.io* та завантажити інсталятор. Файл використовує пряме посилання, або ж використовує *mirrors.jenkins-ci.org/windows/latest* власний пакет *Windows*. Можна

знайти інсталятор *Windows Jenkins*. На рис 3.3 показаний сайт *Jenkins* на якому в правому кутку є кнопка скачати.

Рис. 3.3. Сайт *Jenkins*

3.4. Висновки до розділу

Jenkins, сервер автоматизації з відкритим кодом, використовується у поєднанні з безліччю плагінів для підтримки побудови, розгортання та автоматизації програмних проектів. Цей корисний інструмент має широку підтримку на рівні громади і широко використовується у промисловості та наукових дослідженнях. *Jenkins* в основному використовується в робочих процесах безперервної інтеграції/безперервного розподілу, але також може бути реалізований для різних інших додатків. Оскільки *Jenkins* має високу модульність завдяки своїй системі плагінів, це дозволяє автоматизувати багато автоматизованих процесів. У цьому дослідженні ми зосередимося на використанні *Jenkins* для тестування програмного забезпечення, як на генерації тестових блоків, так і на автоматизації.

Система збірки повністю написана на *Java* і легко розширювана завдяки архітектурі плагінів та точкам розширення, залишеним в об'єктній моделі. Це робить *Jenkins* надзвичайно гнучким та гнучким інструментом, здатним охопити

багато можливих сценаріїв та вимог завдяки тисячам плагінів, розроблених його величезною спільнотою з відкритим кодом.

Jenkins – це програмне забезпечення, яке працює на сервері та виконує різні автоматизовані завдання на програмному забезпеченні. Завдяки своїй гнучкості, *Jenkins* може адаптуватися до багатьох проектів, у процесі обробки *CI/CD*, для побудови, модульного тестування та моніторингу програмних проектів. *Jenkins* широко підтримується спільнотою з відкритим кодом і вдосконалюється завдяки використанню плагінів. Плагіни насправді численні і підходять для різних автоматизацій проектів. Ми представили приклад дослідження *Jenkins*, який ми звикли розуміти та практикувати його конфігурацію. Ми змогли перевірити сховище *Git*, створити файл і відстежувати розвиток цієї роботи [24].

Jenkins постійно використовувався усіма членами для перевірки своїх зобов'язань під час сесій розвитку. У невеликій кількості випадків, коли збірка не вдалася, це було чітко повідомлено решті команди як через монітор зворотного зв'язку, так і голосово розробником, відповідальним за коміт.

Jenkins – чудовий, високо настроюваний *CI*-сервер з безліччю плагінів для адаптації інструменту до конкретних потреб. *Web*-інтерфейс робить його доступним для всіх розробників, що є необхідною умовою *CI*.

Ідея використання плагінів для модульної функціональності не нова. Однак є небагато задокументованих випадків використання плагінів як методу, що дозволяє набору тестувати доступ до *API*, а також додавати або змінювати *API* без істотних перезаписів. Цей підхід дозволяє розробити основу тестування для багатьох служб, що використовують одну мову або сервер. Що ще важливіше, плагіни також дозволяють розподілити кодування між кількома розробниками, щоб прискорити тестування на платформі.

Під час розробки плагіна кожен автор обирає тип методології, який буде використовувати плагін. Може бути кілька плагінів із різними типами методологій, які перевіряють один *API*. Щоб збільшити корисність плагіна, він також може завантажувати тестові вхідні значення з файлу, що дозволяє швидко завантажувати багато тестів без перезапису плагіна.

РОЗДІЛ 4
ПРОГРАМА АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

4.1. Створення програми в *Git*

*Я*дром середовища розробки є *Git*, розподілена система контролю версій з відкритим кодом. *Git* дуже гнучка система і може бути налаштований у різноманітних конфігураціях, щоб відповідати декільком різним робочим процесам. Команди *Git* можна запускати з командного рядка, із виділеного клієнта графічного інтерфейсу користувача (*GUI*) або як набір функцій, вбудованих в інтегроване середовище розробки (*IDE*) або інструмент розробки. *Git* відстежує зміни, внесені у файли, і зберігає їх разом з метаданими як "коміти" у сховищі. *Git* зберігає лише зміни, зроблені у файлах, а не повні копії самих файлів (за винятком двійкових файлів, таких як зображення), що робить використання *Git* дискового простору надзвичайно ефективним. “Гілка” схожа на “версію” сховища. Гілка фактично зберігається як набір комітів, відстежуваних *Git*, доки гілка не буде "вилучена", після чого *Git* застосовує коміти до файлів, оновлюючи їх до останнього коміту у гілці.

Контроль версій – це назва, що використовується для програмного забезпечення, яке може допомогти вам записати зміни, внесені до файлів, у каталог на вашому комп’ютері. Програмне забезпечення та засоби контролю версій (такі як *Git* та *Subversion*/*SVN*) часто асоціюються з розробкою програмного забезпечення, і все частіше вони використовуються для співпраці в наукових та академічних середовищах. Системи контролю версій найкраще працюють з простими текстовими файлами, такими як документи або комп’ютерний код, але сучасні системи контролю версій можуть використовуватися для відстеження змін у файлах будь-якого типу.

На самому базовому рівні програмне забезпечення для контролю версій допомагає нам реєструвати та відстежувати набори змін, внесених до файлів на нашому комп’ютері. Тоді ми можемо міркувати про ці зміни та ділитися ними з

іншими. По мірі накопичення набору змін із часом ми починаємо бачити деякі переваги.

Переваги використання контролю версій:

– контроль версій дозволяє нам визначити формалізовані способи співпраці та обміну кодом. Наприклад, злиття між собою наборів змін від різних сторін дозволяє спільне створення документів та програмного забезпечення для розподілених команд;
– версії – наявність надійного та суворого журналу змін у файлі без перейменування файлів (*v1, v2, final_copy*);
– відкат – контроль версій дозволяє швидко скасувати набір змін. Це може бути корисно, коли нові тексти чи нові доповнення коду створюють проблеми;
– контроль версій може допомогти вам зрозуміти, як з'явився код чи запис, хто написав чи вніс певні частини, і кого ви можете попросити допомогти краще зрозуміти його;
– резервне копіювання – хоча це не передбачається як рішення для резервного копіювання, використання систем контролю версій означає, що ваш код і запис можна зберігати на кількох інших комп'ютерах.

Git – одна з найбільш широко використовуваних систем контролю версій у світі. Це безкоштовний інструмент з відкритим кодом, який можна завантажити на локальну машину та використовувати для реєстрації всіх змін, внесених до групи призначених комп'ютерних файлів. Він може бути використаний для локального керування версіями файлів ви самостійно на своєму комп'ютері, але, мабуть, найпотужніший, коли застосовується для координації одночасної роботи над групою файлів, спільно використовуваних розподіленими групами людей.

Замість того, щоб надсилати документи з відслідковуваними змінами та деякими коментарями та перейменовувати різні версії файлів (*example.txt, exampleV2.txt, exampleV3.txt*), щоб їх диференціювати, ми можемо використовувати *Git* для збереження (або, говорячи *Git*, "зробити") усіх цю інформацію разом із самим документом. Це дозволяє легко отримати огляд усіх змін, внесених до файлу

з часом, переглянувши журнал усіх внесених змін. І всі попередні версії кожного файлу все ще залишаються в оригінальній формі: вони не перезаписуються, якщо ми коли-небудь захочемо зробити *roll-back* до них.

Спочатку *Git* був розроблений, щоб допомогти розробникам програмного забезпечення спільно працювати над програмними проектами, але він може бути і використовується для управління редакціями будь-якого типу файлу в комп'ютерній системі, включаючи текстові документи та електронні таблиці. Після встановлення взаємодія з *Git* здійснюється через командний рядок у *Windows* або термінал на *Mac*/*Linux*. Оскільки документи *Word* містять спеціальне форматування, *Git*, на жаль, не може керувати версіями, а також не може керувати версіями *PDF*-файлів, хоча обидва типи файлів можуть зберігатися у сховищах *Git*.

Для того, щоб в *Git* добавити свій код, потрібно скачати саму програму *Git*, вибравши останню версію програми та операційну систему. Після того, як натиснули на кнопку скачати, загрузка файлу для встановлення додатку почнеться автоматично.

Наступним кроком буде створення свого локального репозиторію в *Git*. Потрібно створити папку на комп’ютері для проекта, зайти в створену папку і добавити в папку локальне сховище *Git*, також можна використати такі команди:

cd Diplom

git init # додає до проекту сховище *Git*

Для початку необхідно пройти реєстрацію на *Github*. Створити проект в *Intellij idea* і назвати проект, наприклад *Diplom*.

Зайти на *GitHub*, створити новий репозиторій, задавший сховищу ім'я та власника, також вибрати доступність проекту(публічний чи приватний), як показано на рис 4.1.

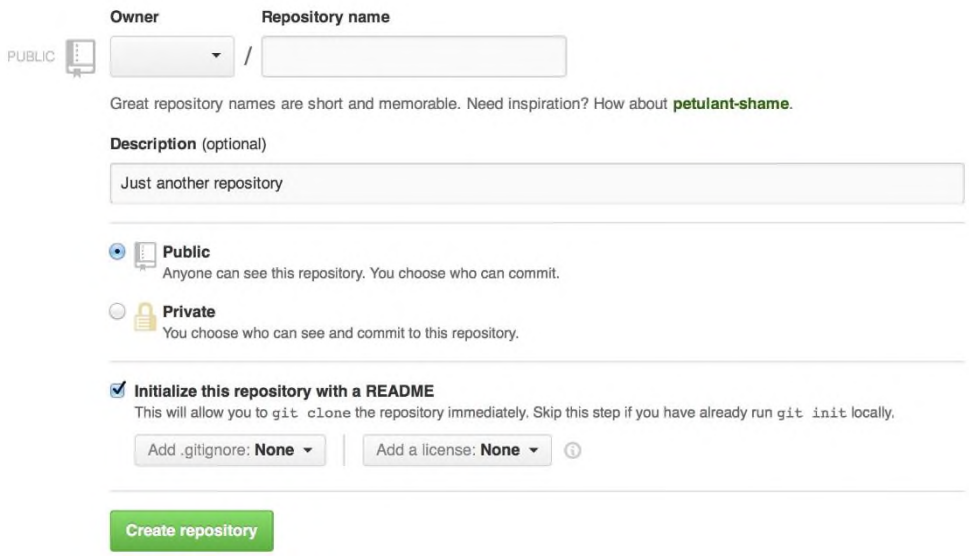

Рис. 4.1. Створення нового репозиторію в *GitHub*

У *Intellij IDEA* натиснути *VCS*, потім вибрати *Import into version control*. Далі *Create Git repository* та вказуємо папку яку створити в *GitHub*.

Потім натискаємо на проект правою клавішею, далі додаємо новий файл з назвою *.gitignore*, в файлі пишемо * *.iml* * *.idea*

Щоб проект зберігався не в локальному репозиторії, а на *Github* в *Intellij idea* натискаючи *VCS-Git-Remotes* натискаємо на кнопку добавити, далі заходимо на *Github* і копіюємо *HTTPS* посилання. Після вставляємо її у вікні *Git Remotes*, задаємо назву гілці і натискаємо зберегти.

Далі натискаємо на кнопку *Commit*, записуємо в поле *Commut Message* повідомлення даного коміта і натискаємо кнопку *Commit and Push* для створення коміта і відправки даних на сервер *Github*. Заходимо на *Github* і дивимося чи з'явився наш проект.

Після того як переконалися, що проект завантажився на сервер, створюємо новий файл в *Github*, записуємо туди дані і зберігаємо. В *Intellij idea* натискаємо на *pull*, після цього файл який був створений на *Github* повинен відобразиться в проекті в програмі *Intellij idea.*

При роботі з *Git* самий використовуваний мережевий протокол для передачі даних - це *SSH*. Причин для цього багато:

1. Можливість *SSH* підключення присутня на більшості серверах;

2. З *SSH* легко працювати і налаштувати;

3. Дає можливість і на запис і на читання;

4. Не потрібно постійно при запитах до центрального серверу вводити логін і пароль;

5. Безпечне з'єднання по 22 порту запобігає можливість включення в сесію і перехоплення даних;

6. Дані передаються в зашифрованому вигляді;

7. Робить файли більш компактними (стискає) перед передачею.

Першим кроком, потрібно згенерувати пару ключів. Це можна зробити командою:

ssh-keygen -t rsa -b 4096 -C "example@gmail.com",

де *example@gmail.com* – електронна пошта користувача, обов'язково потрібно задати ту пошту, яку було вказано при реєстрації на *GitHub*.

Вибрати шлях збереження ключа або ж просто натиснути *Enter*, якщо хочете, щоб ключ зберігся в місці вказаному за замовчуванням, виконавши команду:

generating public/private rsa key pair

Після виконання даної команди, буде запропоновано ввести пароль.

Наступним кроком потрібно додати згенеровані ключі в *SSH*-агент. Для цього потрібно спочатку запустити *SSH*-агент командою:

eval "$(ssh-agent -s)"

Після введення команди, в терміналі на виводі буде показаний *id* запущеного процесу. Далі, як запустився агент, потрібно додати в нього згенеровані ключі командою:

ssh-add ~/.ssh/id_rsa

Ключі додалися, наступним кроком потрібно їх скопіювати і додати в віддалений репозиторій ввести команду в терміналі і скопіювати ключ в буфер обміну:

clip < ~/.ssh/id_rsa.pub.

4.2. Підключення серверів для *Jenkins*

Встановлення *Jenkins* на локальній машині розробки – це перший крок для створення та налаштування автотестів, але встановлення *Jenkins* на належний сервер збірки заслуговує трохи більшої продуманості та планування.

Перше, що потрібно зробити, це підключити сервер збірки. Щоб *Jenkins* працював належним чином потрібні процесор та пам’ять. Сам *Jenkins* є відносно скромним *web*-додатком *Java*. Однак у більшості конфігурацій принаймні деякі збірки будуть виконуватися на основному сервері збірки. Збірки, як правило, є операціями як з пам'яттю, так і з процесором, і *Jenkins* можна налаштувати на паралельне виконання декількох збірок. Залежно від кількості завдань збірки, якими керує власник коду, *Jenkins* також потребуватиме власної пам'яті для власного внутрішнього використання. Обсяг потрібної пам’яті в значній мірі буде залежати від характеру збірок.

Однак, якщо користувач використовує віртуальну машину, переконайтеся, що вона має достатньо пам’яті для підтримки максимальної кількості паралельних збірок. Використання пам'яті сервера *CI* краще всього буде, якщо *Jenkins* створюватиме додатковий *Jvms*. Іншим підходом є створення декількох машин для побудови. *Jenkins* досить легко налаштовує "рабів" на інших машинах, які можуть бути використані для запуску додаткових збірок. Збірки залишаються неактивними до тих пір, поки не буде подано запит на нове завдання збірки – тоді основна установка *Jenkins* відправляє завдання збірки до підлеглого та повідомляє про результати. Також корисною стратегією, якщо певні важкі збірки, як правило, «переймають» основний сервер – просто розмістіть їх на власному виділеному агенті збірки.

Якщо виконувати установку *Jenkins* в середовищі *Windows*, дуже важливо, щоб сама програма працювала як служба *Windows*. Таким чином, *Jenkins* буде автоматично запускатися щоразу, коли сервер перезавантажується, і може управлятися за допомогою стандартних інструментів адміністрування *Windows*.

Одним з переваг запуску *Jenkins* на сервері додатків є те, що, як правило, досить просто налаштувати ці сервери на роботу в якості служби *Windows*.

Jenkins має дуже зручну функцію, розроблену для того, щоб спростити установку *Jenkins* як сервер *Windows*. В даний час немає програми установки з графічним інтерфейсом, яка робила б це за користувача, але отримуємо краще – програму установки з графічним інтерфейсом на базі веб-інтерфейсу.

По-перше, потрібно запустити сервер *Jenkins* на локальній машині. Найпростіший підхід – запустити *Jenkins* за допомогою *Java Web Start*. В якості альтернативи, можна зробити це, завантаживши *Jenkins* і запустивши його з командного рядка.

Ця друга опція корисна, якщо порт за замовчуванням *Jenkins* (8080) вже використовується іншим додатком. Насправді не має значення, який порт використовувати – можна змінити це пізніше.

Як тільки у запуститься *Jenkins*, підключіться до потрібного сервера, потрібно перейдіть на екран *Manage Jenkins*. Тут знайти кнопку "Встановити як сервіс *Windows*". Вона створить на сервері службу *Jenkins*, яка буде автоматично запускати і зупиняти *Jenkins* впорядкованим чином.

Jenkins запросить каталог установки. Це буде домашній каталог *Jenkins*(*JENKINS_HOME*). Стандартний параметр значення за замовчуванням *JENKINS_HOME*: каталог з ім'ям *.jenkins* в домашньому каталозі поточного користувача, як показано на рис 4.2.

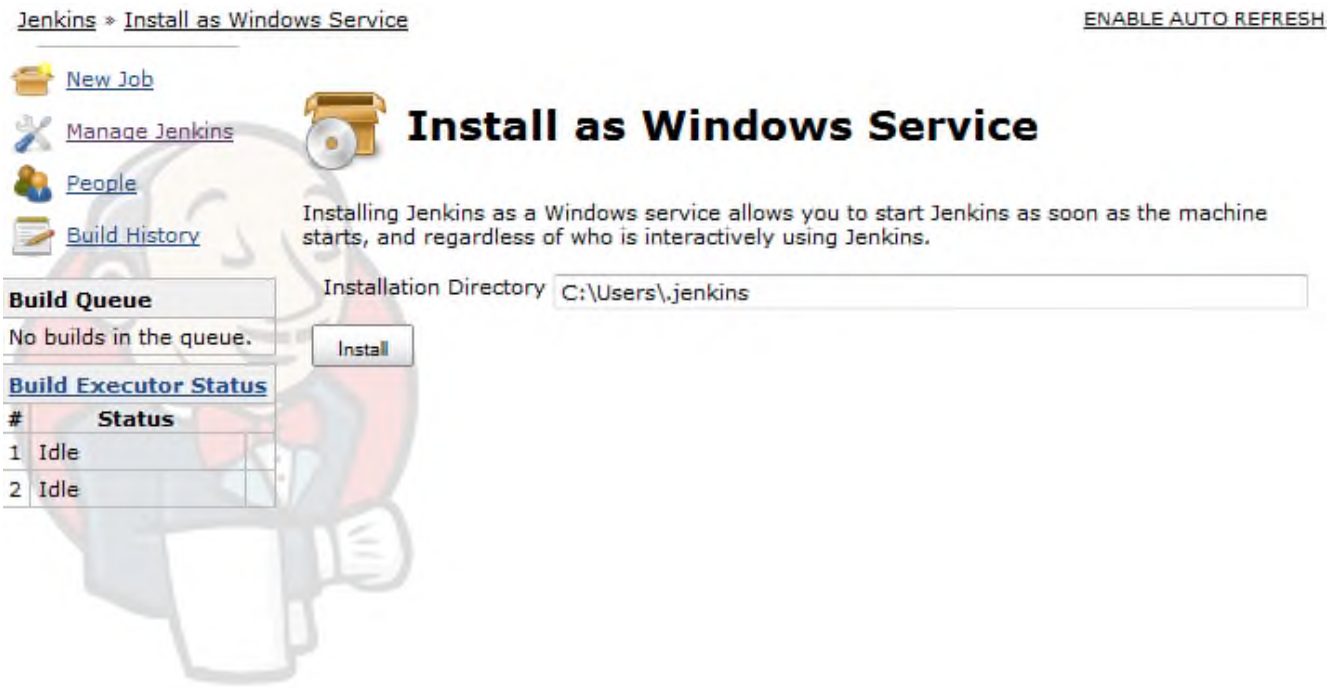

Рис. 4.2. Встановлення *Jenkins* для *Windows Server*

У багатьох версіях *Windows* довжина шляху до файла обмежена приблизно 260 символами. Якщо комбінувати вкладений робочий каталог *Jenkins* і глибокий шлях класу, часто можна перекрити це, що призведе до дуже незрозумілим складальним помилок. Для мінімізації ризиків переповнення шляху до файлів *Windows* обмеження, потрібно перевизначити змінну оточення *JENKINS_HOME*, щоб вказати на більш короткий шлях.

Ця базова установка буде відмінно працювати в простому контексті, але часто вам буде потрібно тонка настройка вашого сервісу. Наприклад, за замовчуванням служба *Jenkins* буде працювати під локальної системної обліковим записом. Однак, якщо ви використовуєте *Maven*, *Jenkins* потрібно каталог *.m2* і файл *settings.xml* в домашньому каталозі. Аналогічно, якщо ви використовуєте *Groovy*, вам може знадобитися каталог *.groovy/lib*. Щоб це було можливо, і щоб спростити тестування установки *Jenkins*, переконайтеся, що запущено ця настройка яка призначена для користувача в обліковому записі з коректно налаштованим середовищем розробки. В якості альтернативи, запустіть додаток від імені системного користувача, але використовуйте систему. Інформаційна сторінка в

Jenkins, щоб перевірити */Users/johnsmart/Projects/Books/jenkins-thedefinitive-guide* каталог, і помістити будь-які файли, які повинні бути.

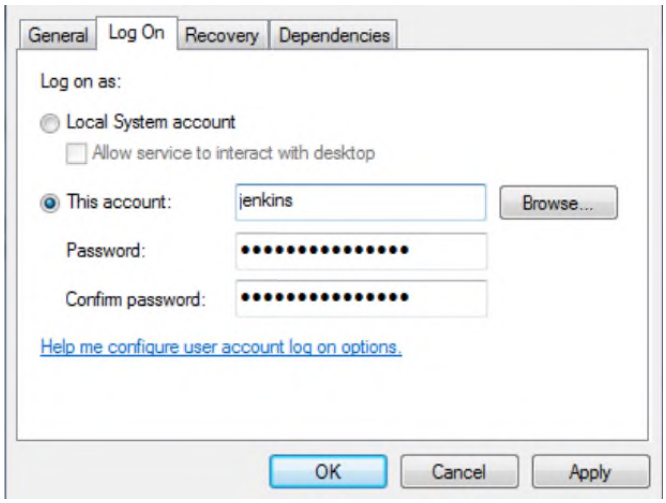

Рис. 4.3. Налаштування *Jenkins* для *Windows Server*

Збереження інших налаштувань деталей служби *Jenkins* буде в файлі під назвою *jenkins.xml*, в тому ж каталозі, що і файл *jenkins.war*. Тут можна налаштувати (або переналаштувати) порти, опції *JVM*, робочу директорію *Jenkins*. В даному прикладі добавляємо для *Jenkins* трохи більше пам'яті і змінюємо його порт на 8080, як показано на рис 4.4.

```
<service>
  <id>jenkins</id>
  <name>Jenkins</name>
  <description>This service runs the Jenkins continuous integration system
    </description>
  <env name="JENKINS_HOME" value="D:\jenkins" />
  <executable>java</executable>
  <arguments>-Xrs -Xmx512m
  -Dhudson.lifecycle=hudson.lifecycle.WindowsServiceLifecycle
  -jar "%BASE%\jenkins.war" --httpPort=8081 --ajp13Port=8010</arguments>
</service>
```

Рис. 4.4. Зміна порта *Jenkins*

4.3. Створення проекту та автотестів в *Jenkins*

При створенні кожного *web*-додатку або програмного забезпечення дуже важливо упевнитися в тому, що функціональність розробленого програмного забезпечення відповідає вимогам зацікавлених сторін. Тестування програмного забезпечення – це процес, який використовується для підвищення надійності та

якості програмного забезпечення шляхом пошуку якомога більшої кількості помилок.

Автоматизоване тестування використовує програмні інструменти, окремо від тестованого програмного забезпечення, для контролю за виконанням тестів і порівняння фактичних результатів з прогнозованими. Процес автоматизації тестування може автоматизувати необхідні завдання в формалізованому процесі тестування, або виконати додаткове тестування, яке було б важко виконати вручну. Автоматизація випробувань має вирішальне значення для безперервного постачання і безперервного тестування.

Існує безліч причин для використання автоматизованого тестування. Оскільки тестові випадки повторюються і комп'ютер може виконувати їх з високою швидкістю, це скорочує час і вартість тестування. При правильному використанні автоматизоване тестування може поліпшити якість програмного забезпечення, тому що комп'ютер може бути більш точним в реєстрації дефектів, ніж будь-який людський тестер. До інших позитивних аспектів використання автоматизованого тестування відносяться збільшення охоплення тестування, заміна трудомістких завдань і регресивний електронне тестування [19].

В курсовій роботі показаний приклад автоматизації *web*-додатку *GitHub* написаний на мові *Java*.

Для того, щоб спочатку створити репозиторію в *GitHub* потрібно зареєструватись або увійти в систему. Тобто першим кроком стоврення автотестів буде написання коду для входу або реєстрації в *web*-додатку. Створити відкритий клас з назвою *LoginPage*, в якому будуть передаватись селектори, які знаходяться на головній сторонці коли користувач хоче увійти в систему *GitHub*.

```
package pages;
import org.openqa.selenium.By;
public class LoginPage {
   public By loginField = By.cssSelector("[id='login']");
   public By passwordField = By.cssSelector("[id='password']");
   public By signInButton = By.cssSelector("[name='commit']");}
```

Зберегти файл в папку де будуть зберігатись всі інші файл, які пізніше перенесемо в *Jenkins* для створення автотестів. Назвати даний файл *LoginInGit.java*

Наступник кроком створенням автоматизації *web*-додатку буде, перехід на головну сторінку в *GitHub.* Для цього створено публічний клас з назвою *MainPage* в якому додані селектори, що описують елементи на головній сторонці нашого *web*-додатку.

public class MainPage {

public By SignedInAsButton = By.cssSelector("[class='css-truncate-target']");

public By OverviewButton = By.cssSelector("[class='UnderlineNav-item mr-0 mr-md-1 mr-lg-3 selected ']");

public By RepositoriesButton = By.xpath("//a[contains(text(),'Repositories')]");

public By ProjectsButton = By.xpath("//a[contains(text(),'Projects')]");

public By StarsButton = By.xpath("//a[contains(text(),'Stars')]");

public By FollowersButton = By.xpath("//a[contains(text(),'Followers')]");

public By FollowingButton = By.xpath("//a[contains(text(),'Following')]");

public By EditProfileButton = By.xpath("//div[@class='d-none d-md-block']//div[@class='hide-sm hide-md']");

public By ShowMoreActivityButton = By.xpath("//button[contains(text(),'Show more activity')]"); }

Зберегти файл з назвою *MainPageGit.java*.

Після написання коду для авоматизованого тестування головної сторінки, можна написати код по створенню нового проекту в *GitHub*. Створити публічний клас з назвою *NewGist*, в якому додані селектори кнопок, створити новий проект, добавити файл, та перехід на крок назад або вперед.

public class NewProject {

public By newProjectButton = By.cssSelector("[data-ga-click='Header, create new project']");

public By projectBoardName = By.cssSelector("[id='project_name']");

public By createProjectButton = By.cssSelector("[class='btn btn-primary flex-auto float-none float-md-left']");}

Зберегти файл з назвою *NewProject.java*.

Також, напишемо код для створення нового репозиторію. Створити публічний клас з назвою *NewRepository*, в якому селектори описують кнопку стоврити нову репзиторію, власник репозиторії, вибору деталей для нового репозиторія і інші елементи на сторінці.

public class NewRepository {

public By newrepositoryButton = By.cssSelector("[data-ga-click='Header, create new repository']");

public By ownerButton = By.cssSelector("[id='repository-owner']");

public By addGitignoreButton = By.cssSelector("[class='details-reset details-overlay select-menu']");

public By addLicenseButton = By.cssSelector("[class='details-reset details-overlay d-inline-block position-relative']");

public By importRepositoryLink = By.cssSelector("[data-ga-click='Create Repository, import repository, location:repo new']"); }

Зберегти файл з назвою *NewRepository.java*.

Після того, як написані програми для автоматизації *web*-сайту потрібно налаштувати *Jenkins* та плагіни для відображення автотестів, позитивних та негативних кейсів.

Плагін *Git* доступний в *Jenkins Plugin Manager*. Плагін передбачає, що *Git* вже встановлено на вашому локальному сервері, тому необхідно переконатися, що це так. Можете зробити це, запустивши таку команду на локальному сервері :

$ git –version

Далі поверніться до *Jenkins*, встановіть відповідний прапорець на сторінці *Jenkins Plugin Manager* і натисніть кнопку *Install*.

Після встановлення плагіну *Git* на сторінці *Manage Jenkins # Configure System*, як представлено на рис 4.5, буде доступний невеликій новий набір конфігураційних опцій. Зокрема, необхідно вказати шлях до виконуваного файлу *Git*. Якщо *Git* вже встановлений в системному шляху, просто поставте "*git*" тут.

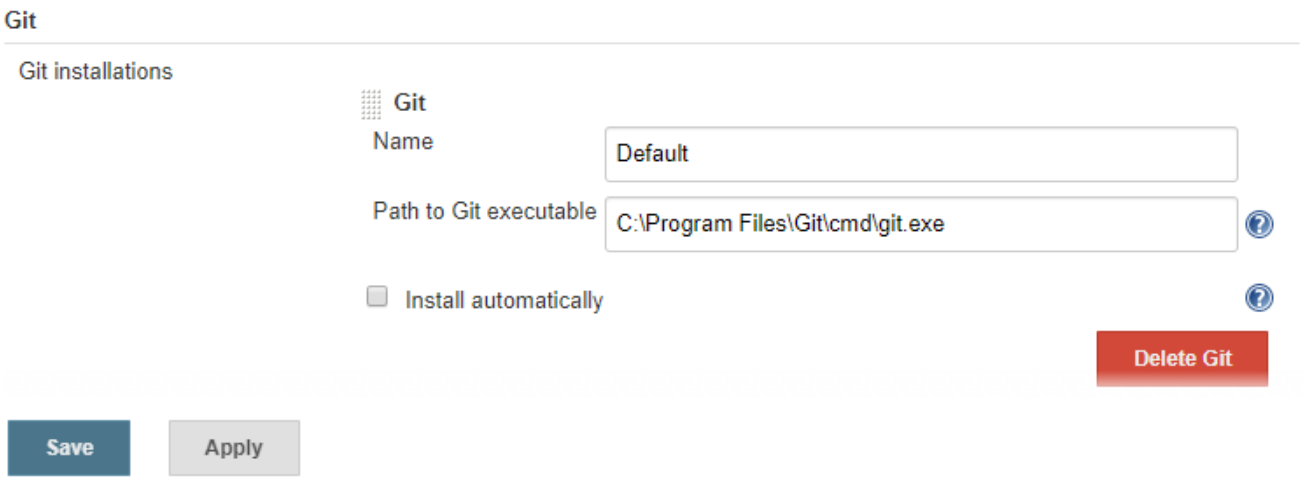

Рис. 4.5. Загальносистемна конфігурація плагіна *Git*

У будь-якому існуючому або новому проекті *Jenkins* буде відображена нова опція управління вихідним кодом для *Git*. Звідси можна налаштувати один або декілька адрес сховища. Одного сховища зазвичай досить для більшості проектів: додавання другого сховища може бути корисно в більш складних випадках, і дозволяє вказати окремі іменовані місця для операцій *pull and push* [23].

Специфікатор гілки – це шаблон підстановки або специфічне ім'я гілки, яке повинно бути побудовано *Jenkins*. Якщо залишити порожнім, усі гілки будуть побудовані. Після першого збереження роботи з порожніми гілками для настройки збірки, вона заповнюється символом *, що означає "побудувати всі гілки". На рис 4.6. показана сторінка для створення проекту.

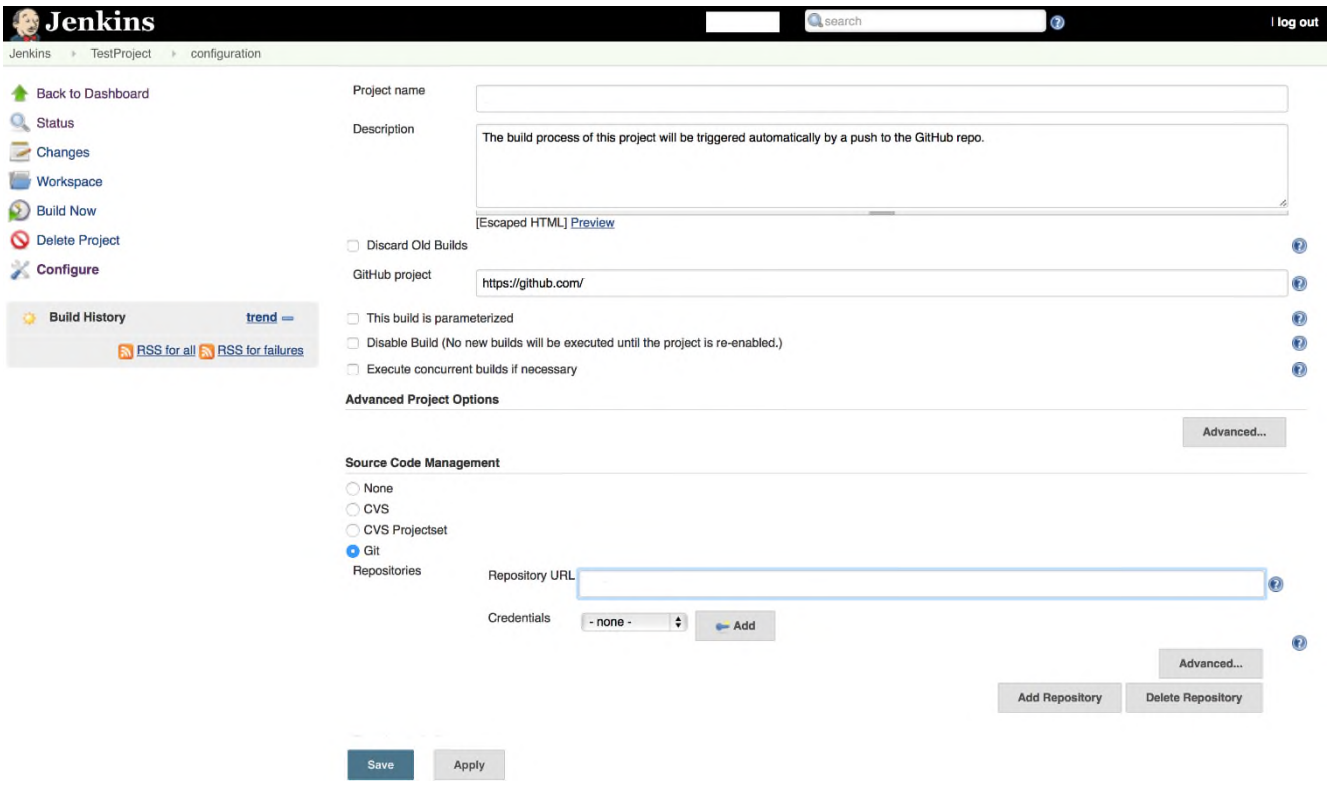

Рис. 4.6. Сторінка *Jenkins* для створення нових проектів

Плагін *Git* також дозволяє ігнорувати певних користувачів, навіть якщо вони вносять зміни в кодову базу, які зазвичай запускають збірку.

Виключені користувачі – це, як правило, автоматизовані користувачі, а не розробники, які випадково мають різні облікові записи з правами фіксації в системі управління вихідним кодом. Ці автоматизовані користувачі часто вносять невеликі числові зміни, такі як збільшення номерів версій в файлі *pom.xml*, замість того, щоб вносити реальні логічні зміни. Якщо потрібно виключити декількох користувачів, просто помістіти на окремі рядки.

Простий процес налаштування *Jenkins* включає в себе кілька плагінів і облікових даних для управління вихідним кодом, а також середовище зборки. Завдяки додатку *React*, плагін *NodeJS* незамінний в якості здатних виконувати *npm*-команди в оболонці виконання *Jenkins*. Плагін можна легко знайти на *Available* закладка на діалозі *Plugin Manager* і перезавантаження після установки не потрібно. Після цього, *NodeJS* розділ в *Global Tool Configuration* буде доступний для додавання інсталятора *NodeJS*, так що тоді люди зможуть вибрати відповідну версію на *nodejs.org* для надання папки *Node* і *npm* в *PATH* збірки *Jenkins*. завдання.

Інший плагін також повинен бути встановлений – це плагін *SSH Agent*, який дозволяє користувачам забезпечувати *SSH* облікові дані для складання за допомогою своєї функції. Що людям потрібно зробити, так це просто завантажити його з *Plugin Manager* і потім з'явиться опція *SSH Agent* в розділі *Build Environment* завдання збірки.

Сервер *Jenkins* повинен бути вже інтегрований з *Git*, і вибір варіанту *Git* в *Source Code Management* дає безліч переваг гнучкості управління робочим простором. Крім того, для цього потрібно глобальна обліковий запис, що містить ім'я користувача та пароль облікового запису, яка доступна в певному репозиторії. Для підключення до *S3*-боксів автор використовував опції секретного тексту в середовищі збірки, яка складається із змінної імені користувача і змінної пароля з обліковим записом *AWS* з коренем або обліковим записом *IAM* з достатнім функціоналом для забезпечення складальної завдання. У той час як у випадку з підключенням примірника *EC2*, прив'язка до його ключового файлу є істотним елементом для надання *SSH*-облікових даних плагіном *SSH Agent*. Правильними вимогами до облікових даних, які повинні бути пов'язані з ім'ям користувача, є дані *EC2*-користувача і дані приватного ключа, які можуть бути введені безпосередньо текстом. Правильна інформація про ключове файлі повинна починатися з "----- *BEGIN RSA PRIVATE KEY* -----" і закінчуватися на "----- *END RSA PRIVATE KEY* -----". Приклад представлений на рис 4.7.

Рис. 4.7. Додавання облікових даних на сервері *Jenkins*

Завдання як для реактивних додатків, так і для *.NET* додатків досить схожі один на одного і переслідують одну і ту ж мету. це завантаження файлів в хмарну середу. Після завершення налаштування для управління вихідним кодом і побудувати середовище, залишилося зробити тільки одне – надати сценарії *Jenkins* для виконання. на виконання оболонки. Фізично існує кілька варіантів виконання скриптів, які залежать від вимог користувача, і можна зробити більше одного кроку виконання, таким чином, користувачі можуть побудувати замовне виконання системи скриптів в їх розпорядженні. Скрипти для завдання *React* прості, включають в себе установку потрібних *npm* пакетів відповідно до файлом *package.json*, потім створіть папку продукту для завантаження в *S*3 скриньки. На 3 іншого боку, скрипти для *.NET* завдання трохи складніше для управління версіями і ведення резервні файли. Крім того, зробити транспортний файл до примірника *EC*2 більш простим.

Один з важливих аспектів це побудовою історії. Кожне завдання на складання споживає додатковий дисковий простір і пам'ять, що може бути виправдано, а може і ні, в залежності від характеру розробленої роботи. Наприклад, для метрики якості коду, що будується за принципом статичний аналіз і метрики покриття коду з плином часу, можливо, захочеться вести запис збірок для тривалість проекту, яке автоматично розгортає додаток для перевірки сервер, збереження історії збірки і артефактів для інших користувачів може бути менш важливим. Приклад відображення даного вікна представлений на рис 4.8.

На рис 4.8 представлений графічний інтерфейс користувача (*GUI*) середовища *Jenkins* після виконання збірки. Завершено кілька збірок. Кожна робота *Jenkins* має стільки збірок, скільки ми робимо для тестів. Після кожної модифікації створюється нова збірка і виконується автоматично, як тільки вихідний код оновлюється.

На рис 4.8 відображені сині кулі та остання червоні. Також відображені погодні знаки дощ, грім, хмарність та сонце. Всі ці погодні знаки описують роботу робочих місць.

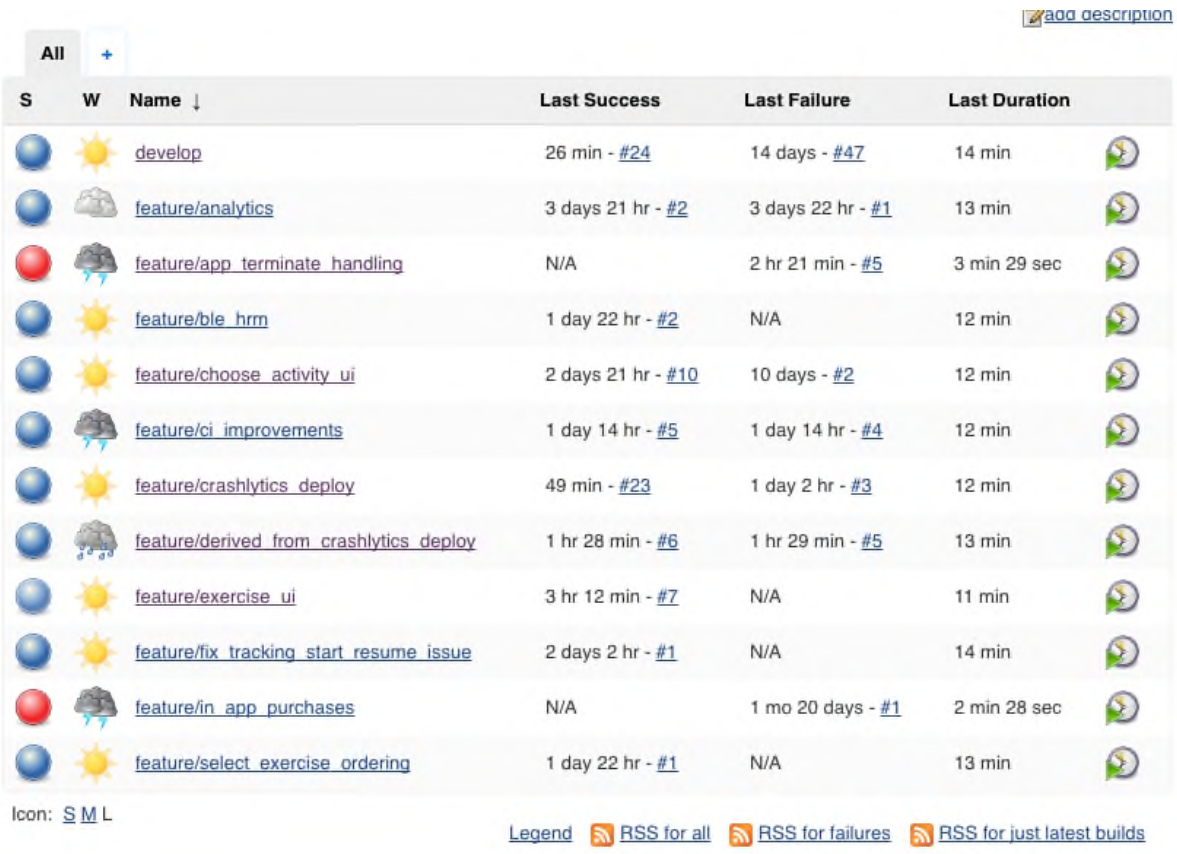

Рис. 4.8. Список збірок у багатогалузевих мережах *Jenkins*

Синій колір означає успішний білд збірки а грім, невдале побудова всередині роботи. Робота все ще успішна, але в ній багато невдалих збірок, це означає, що тест проходить, але є деякі побоювання, що те, що громовий знак демонструє. Символ погоди змінюється з сонячного на хмарний і в кінці грому. Сонячний – успішний сценарій. Хмарно означає, що в роботі мало невдалих виконаних тестів. Грім з'являється щоразу, коли збірка має невдалу побудову всередині тесту.

На рис 4.9 показаний список збірок на створеному дипломному проекті. Перша збірка, яка показана на рис 4.9, це новий запущений білд, який ще виконується. Друга збірка – це вже виконаний білд, у якого є помилки.

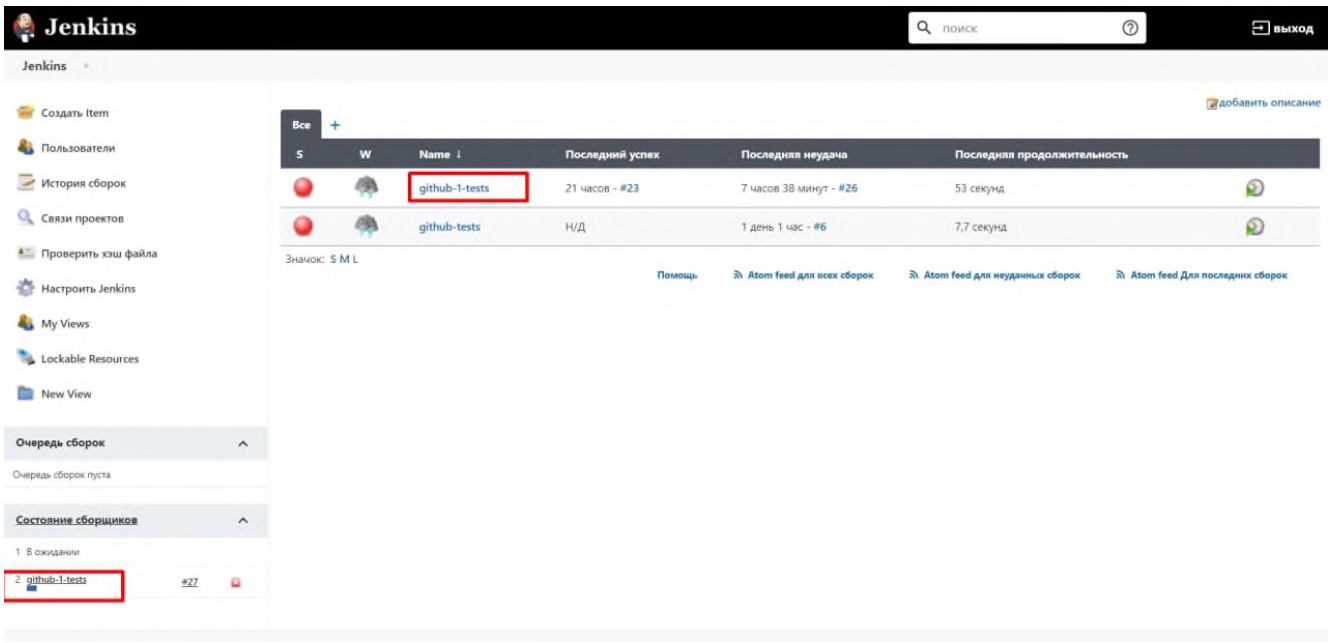

Рис. 4.9. Список збірок у багатогалузевих мережах *Jenkins*

Графік, який представлено на рис 4.10, показує, скільки збірок було зроблено для цього продукту. З лівого боку виконуються збірки і кількість збірок в завданні. Кольори означають результат виконання збірки, якщо збірка була виконана червоним кольором, це означає невдалий, синій – успішний запуск.

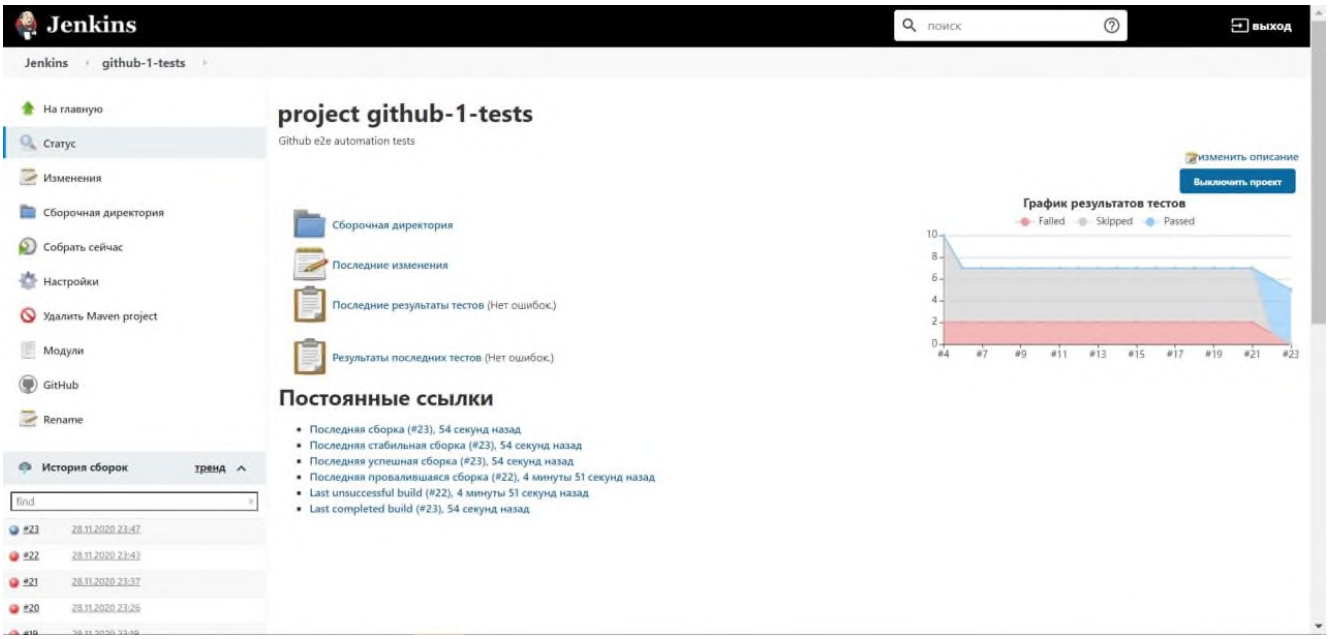

Рис. 4.10. Графік аналізу результату збірки в *Jenkins*

Кожен білд також означає, скільки разів змінений код і наскільки успішний. На графіку пройшло кілька збірок, і деякі з них були з помилками. Ґрунтуючись на результатах проходження або невдачі збірки, які представлені на рис 4.10, можемо

проаналізувати якість програмного забезпечення. *Jenkins* відстежив зміни і почав будувати новий білд на основі змін в *SVN*.

Він використовує той же самий скрипт, який запускає всі збірки. Менше одного завдання може бути стільки ж збірок, скільки змін у вихідному коді, які були зафіксовані в контролі версій.

Під час побудови нового білда, можна спостерігати, що в даний час виконує плагін. Вся інформація виводиться у вікні Консоль. В цьому вікні можна подивитись, які тести були успішно виконані а які впали в помилку. Після закінчення виконання білда, можна відкритий кожен тест і проаналізува, що було виконано успішно а в чому була помилка.

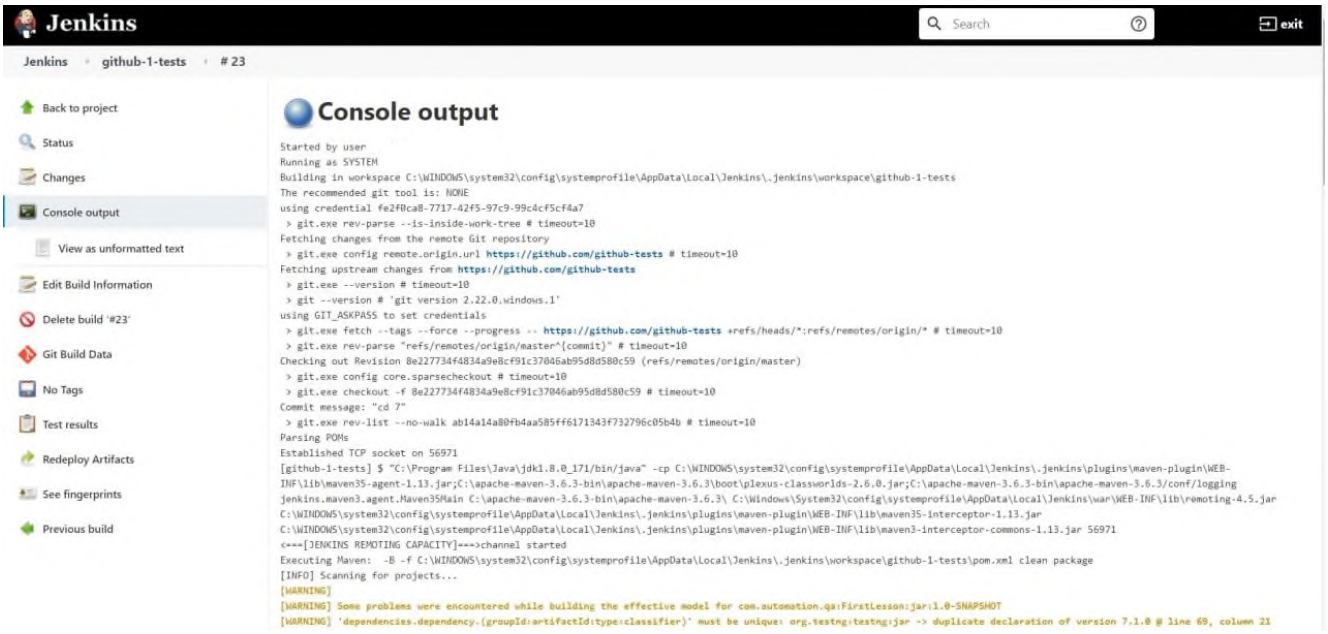

Рис. 4.11. Графік аналізу результату збірки в *Jenkins*

На рис 4.11 показано вікно Консолі виводу успішного білда. Даний негативний кейс показує успішну помилку по *timeout*, тобто ми не дочекались відповіді від нашого сайту. Код виводу консолі буде додано в Додаток А.

4.4. Висновки до розділу

В даному розділі наведено опис та написання коду за допомогою *Jenkins*, суть якого полягає у проріджені потоку вхідних заявок на процесор, для зменшення завантаженості буферу В цілому, ця робота була спрямована на демонстрацію концепції створення додатка в *Jenkins*. Під час експерименту і реалізації процесу вибірки, побудувати систему з нуля з розумінням, як мав працювати над проектами в своїй компанії. Реалізовано автоматизація програми яка написана нна мові програмування *Java*. Описані основні класи, за допомогою яких можна виконувати автоматизоване тестування *web*-додатку.

Також користувались ресурсом *Git* та плагінами для з'єднання написаної програми та сервером *Jenkins*.

Описано як підключались до серверу, плагінів *Jenkins* та конект між *Git* і *Jenkins*. Продемонстрована робота програми, вивід в консоль помилок або успішних кейсів та графік виконаних білдів.

Було введено тестування програмного забезпечення, описані різні рівні тестування і підходи до тестування. Також описано багато теоретичної інформації, яка необхідна, і без неї було б важко налаштувати роботу програми.

ВИСНОВКИ

Ця робота почалася з вивчення і розуміння необхідного і потрібного програмного забезпечення для успішного виконання роботи.

Завдяки цій роботі було важливо навчитися писати автоматизовані тести на мові *Java* і написати перший модульний тест, щоб мати можливість запускати тести на новому сервері *Jenkins* на коректної продуктивності. Також було проаналізовані теоретичні знання для написання коду.

Безперервна інтеграція з *Jenkins* в результаті чого курс відмінно підходить показав, що це покращило тестове покриття і стримував кількість невдалих білдів під час всього цикл розвитку.

Безперервна інтеграція – це метод розробки, при якому розробники часто включають код в повністю інтегровану систему. Автоматизована побудова і автоматизовані тести можуть потім перевірити будь-яку інтеграцію. Автоматичне тестування, як правило, не є суворої частиною *CI*. Одним з основних переваг щоденної інтеграції є те, що ви можете легко виявити і швидко виявити помилки. Так як кожне внесене зміна, як правило, невелика, ви можете швидко визначити конкретне зміна, в результаті якого було виявлено дефект. Останнім часом *CI* став таким стандартним протоколом і набором основних елементів для розробки програмного забезпечення. *CI* гарантує, що програмне забезпечення працює безперервно і підрозділи розробників не сильно відрізняються від стовбура. Дослідження показують, що *CI* має більш високу швидкість розгортання, більш надійні системи і більш якісні програми. Переваги *CI* значні. Основні аспекти ефективної безперервної інтеграції повинні включати в себе взаємодію, стимулюючий збірку програмного забезпечення. Послідовність автоматизації тестування повинна бути активована протягом декількох хвилин для кожного комітів.

Початковий етап в *CI* полягає в наданні автоматизованого скрипта, який створює пакети для будь-якого оточення. Останні дослідження підтверджують це твердження, допомагаючи підкреслити зв'язку між створенням, проектуванням і

впровадженням програмного забезпечення. Безперевна інтеграція в проекти по розробці програмного забезпечення допомагає запобігти відключення і знизити ризик.

В ході написання програми виникли труднощі із з'єднанням усіх частин разом, де важливо було правильно написати сценарій маршрутів і шляхів, щоб вихідний код був імпортований з потрібного місця. Ця робота зайняла кілька тижнів, щоб розібратися в правильних скриптах для створення з'єднань. Навчитися розуміти вихідний код і його продуктивність було важливо, тому що написання інших юніт-тестів було б неможливо.

Основною вимогою було створення середовища тестування з автоматизацією безперервної інтеграції, що використовує *Jenkins* як інструмент тестування, а *Java* в зв'язку з тим, що тестові приклади будуть написані на мові *Java*. Підключіться до контролю версій, зберіть завдання і протестуйте функціональність системи, запустивши юніт-тести:

1. Цілі проекту;
2. Підготовча середу;
3. Завантажити необхідні плагіни;
4. Поєднуючи всі частини разом;
5. Визначити параметри тесту;
6. Установка нових параметрів завдання;
7. Виконання тестових завдань;
8. Аналіз результатів.

До кінця проекту всі частини були об'єднані, і, нарешті, система почала працювати і реагувати на зміни у вихідному коді. Блок-тест починає працювати автоматично. Робота успішно завершена.

Інтегрований цикл *Scrum*, який був прийнятий в експлуатацію в той же час, в той час як цей проект мав місце, був складним. Неодноразово траплялося так, що деякі роботи через деяких проблем доводилося продовжувати на наступний день, іноді кілька днів.

Jenkins є *Java*-сервером з відкритим вихідним кодом і *Java*-сервером безперервної інтеграції (*CI*) або інструментом безперервного моніторингу збірки для багаторазового виконання завдань, також відомим як інструмент автоматизації тестування. Він постійно відстежує помилки розробки на ранній стадії розробки програмного забезпечення.

Jenkins концентрується на двох основних завданнях: по-перше, виконувати тестування, а по-друге, безперервно будувати проекти.

Jenkins – одна з найбільших відкритих систем збирання *CI* з відкритим вихідним кодом. Завдяки своїй гнучкості, сотням плагінів і стійкості до різних типів систем. Вона дозволяє працювати в різних середовищах і взаємодіяти з різними зацікавленими сторонами процесу.

Основною метою даної роботи було дослідження того, як автоматизоване тестування може бути застосоване до *web*-інтерфейсу і які доступні фреймворки і інструменти тестування можуть бути використані для цього. Мета була повністю досягнута, і був представлений ряд можливих випробувальних систем/інструментів. В результаті впровадження були розроблені тестові приклади для регресивного тестування, які можуть бути використані в якості основи для подальшого розвитку компанією, що також було частиною поставленої мети.

Також в роботі було описано як підключати плагіни та які плагіни можна використувати в роботі. Ідея використання плагінів для модульної функціональності не нова. Однак є небагато задокументованих випадків використання плагінів як методу, що дозволяє набору тестувати доступ до *API*, а також додавати або змінювати *API* без істотних перезаписів. Цей підхід дозволяє розробити основу тестування для багатьох служб, що використовують одну мову або сервер. Що ще важливіше, плагіни також дозволяють розподілити кодування між кількома розробниками, щоб прискорити тестування на платформі.

Автоматизація в цілому, а не тільки в зв'язку з тестуванням, дійсно є найважливішою частиною всіх бізнес-процесів. Автоматизація здатна забезпечити більш стабільні, більш ефективні і більш послідовні результати, такі ж, як і результати, отримані при виконанні однієї і тієї ж діяльності знову і знову. Крім

того, краще розробити систему, яка буде робити одне і те ж справа один раз, як протилежність багаторазовим повторенням одного і того ж справи. Автоматизація дуже популярна, і кожна усталена компанія намагається інтегрувати її в свої бізнес-процеси.

Отже, забезпечення якості – невід'ємна частина циклу розробки програмного продукту. При кожній незначній зміні програмного продукту існують аргументи щодо необхідності виконання його автоматизованого тестування. Проте деякі програмісти вважають, що перевірка функціональності програмного забезпечення повинна здійснюватися лише у найважливіших частинах циклу розробки продукту.

Ця робота представлена в середовищі *Jenkins*, яка орієнтована на інтеграцію автоматизації тестування. Застосувати нове середовище безперервної інтеграції (*CI*) для автоматизації модульних тестів, використовуючи *Jenkins* як інструмент.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *D. L. D. Yuan-Fang Li, Paramjit K. Das, "Two decades of web application testing – a survey of recent advances," Information Systems, vol.* 43, *pp*. 20 – 54, 2014.

2. *Jeff Kramer and Orit Hazzan. The role of abstraction in software engineering. In Proceedings of the 28th international conference on Software engineering, pages* 1017–1018. *ACM*, 2006

3. *Janardhanudu, Girish, "White Box Testing", https://buildsecurityin.us-cert.gov/daisy/bsi/articles/bestpractices/white-box/259-BSI.html, February* 08, 2009.

4. *H. Liu and H. B. Kuan Tan, "Covering code behavior on input validation in functional testing," Information and Software Technology, vol.* 51, *no*. 2, *pp*. 546–553, *Feb*. 2009.

5. *Shari Lawrence Pfleeger and Joanne M. Atlee. Software engineering: theory and practice (internationaledition). Pearson,* 2010.

6. *Dorota Huizinga and Adam Kolawa. Automated defect prevention: best practices in software management. Wiley-Interscience, IEEE Computer Society*, 2007.

7. *Simon Stewart and David Burns. Webdriver* 2014-06-18.

8. *NORSOK standard*, *I*-005, *Rev.* 2, 2005.

9. *Simulink PLC Coder 1.0. http://www.mathworks.com/products/slplc-coder/, last retrieved on Aug* 31*st*, 2010.

10.*http://www.deifwindpower.com/Products/PLC_Link.aspx?M=Shop&PID=19159&ProductID=*1344, *last retrieved on Aug* 31*st*, 2010.

11. *Wonderware platform, http://global.wonderware.com/EN/Pages/default.aspx, last retrieved on Jan* 31*st*, 2011.

12. *Veronica G Vergara Larrea, Wayne Joubert, and Christopher B Fuson. 2015. Use of Continuous Integration Tools for Application Performance Monitoring. Technical Report. Oak Ridge National Laboratory (ORNL); Oak Ridge Leadership Computing Facility (OLCF).*

13. *Joseph Voss, Joe A Garcia, W Cyrus Proctor, and R Todd Evans. 2017. Automated System Health and Performance Benchmarking Platform: High Performance Computing Test Harness with Jenkins. In Proceedings of the HPC Systems Professionals Workshop.*

14. *Charles Anderson. 2015. Docker [software engineering]. IEEE Software* 32, 3 (2015), 102–c3.

15. *Singularity Registry. https://doi.org/10.5281/zenodo.1012531. (2018). [Online; accessed 06-March-2018].*

16. *“What is jenkins? the ci server explained,” https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html.*

17. *David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton, “The combinatorial design approach to automatic test generation,” IEEE Software, September* 1996, *pp.* 83–88.

18. *G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," PloS one, vol. 12, p. e0177459, 2017.*

19. *E. Dustin, T. Garrett, and B. Gauf, Implementing Automated Software Testing. Pearson Education,* 2009.

20. Бойченко С. В., Іваненко О. В. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету. – Київ: НАУ, 2017. – 63 с.

21. ГОСТ 19.701-90 ЕСКД. Схемы алгоритмов и программ, данных и систем. Условные обозначения и правила выполнения.

22. ДСТУ ГОСТ 3008-95 «Документація. Звіти у сфері науки і техніки. Структура і правила оформлення»

23. *Berg, A. M., 2015, “Jenkins continuous integration cookbook: Over 90 recipes to produce great results from Jenkins using pro-level practices, techniques, and solutions,” Birmingham, UK: Packt Publishing*

24. *J. Humble and D. Farley, Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.*

25. *Alan Berg “Jenkins Continuous Integration Cookbook Paperback” – June 21, 2012 p. 344*

Додаток А

Код виводу консолі

T E S T S

Running TestSuite

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".

SLF4J: Defaulting to no-operation (NOP) logger implementation

SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.

Starting ChromeDriver 87.0.4280.20 (c99e81631faa0b2a448e658c0dbd8311fb04ddbd-refs/branch-heads/4280@{#355}) on port 16692

Only local connections are allowed.

Please see https://chromedriver.chromium.org/security-considerations for suggestions on keeping ChromeDriver safe.

ChromeDriver was started successfully.

ноя 29, 2020 11:45:37 AM org.openqa.selenium.remote.ProtocolHandshake createSession

INFO: Detected dialect: W3C

ноя 29, 2020 11:45:37 AM org.openqa.selenium.devtools.CdpVersionFinder findNearestMatch

WARNING: Unable to find an exact match for CDP version 87, so returning the closest version found: 86

ноя 29, 2020 11:45:37 AM org.openqa.selenium.devtools.CdpVersionFinder findNearestMatch

INFO: Found CDP implementation for version 87 of 86

Tests run: 6, Failures: 2, Errors: 0, Skipped: 0, Time elapsed: 37.11 sec <<< FAILURE! - in TestSuite

newRepository(tests.GitHubPageTests) Time elapsed: 1.794 sec <<< FAILURE!

org.openqa.selenium.TimeoutException: Expected condition failed: waiting for element to be clickable: By.cssSelector: [class='details-reset details-overlay select-menu'] (tried for 1 second(s) with 500 milliseconds interval)

Build info: version: '4.0.0-alpha-7', revision: 'de8579b6d5'

System info: host: 'LAPTOP-6SS8GI2Q', ip: '192.168.31.149', os.name: 'Windows 10', os.arch: 'amd64', os.version: '10.0', java.version: '1.8.0_171'

Driver info: org.openqa.selenium.chrome.ChromeDriver

Capabilities {acceptInsecureCerts: false, browserName: chrome, browserVersion: 87.0.4280.66, chrome: {chromedriverVersion: 87.0.4280.20 (c99e81631faa0..., userDataDir: C:\WINDOWS\TEMP\scoped_dir1...}, goog:chromeOptions: {debuggerAddress: localhost:59125}, javascriptEnabled: true, networkConnectionEnabled: false, pageLoadStrategy: normal, platform: WINDOWS, platformName: WINDOWS, proxy: Proxy(), setWindowRect: true, strictFileInteractability: false, timeouts: {implicit: 0, pageLoad: 300000, script: 30000}, unhandledPromptBehavior: dismiss and notify, webauthn:virtualAuthenticators: true}

Session ID: 4a3aa19330fe1a826c6c0cf97ca0d0ad

at org.openqa.selenium.support.ui.FluentWait.until(FluentWait.java:211)

at helpers.ElementsHelper.isElementClickable(ElementsHelper.java:43)

at tests.GitHubPageTests.newRepository(GitHubPageTests.java:25)

Додаток Б

Код програми для автоматизації

```
package helpers;
import org.openqa.selenium.By;
import org.openqa.selenium.NoSuchElementException;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebDriverException;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import java.time.Duration;
public class ElementsHelper {
private WebDriver driver;
 public ElementsHelper(WebDriver driver) {
 this.driver = driver;
    }
 public boolean isElementPresent(By element, int timeout) {
 WebDriverWait wait = new WebDriverWait (driver,Duration.ofSeconds(timeout));
 try {
 wait.until(ExpectedConditions.presenceOfElementLocated(element));
 return true;
} catch (NoSuchElementException e) {
 throw new RuntimeException("Web element is not present: " + element, e);
}           }
 public boolean isElementVisible(By element, int timeout) {
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(timeout));
 try {
wait.until(ExpectedConditions.visibilityOfElementLocated(element));
return true;
 } catch (NoSuchElementException e) {
```

```
throw new RuntimeException("Web element is not visible: " + element, e);
        }    }
 public boolean isElementClickable (By element, int timeout) {
 WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(timeout));
 try {
wait.until(ExpectedConditions.elementToBeClickable(element));
return true;         }
 catch (NoSuchElementException e) {
throw new RuntimeException("Web element is not clickable:" + element, e);
        }    }
public void clickOnVisibleAndClickableElement (By element,int timeout) {
 WebDriverWait wait = new WebDriverWait(driver,Duration.ofSeconds(timeout));
try {
 wait.until(ExpectedConditions.visibilityOfElementLocated(element));
wait.until(ExpectedConditions.elementToBeClickable(element));
driver.findElement(element).click();           }
 catch (NoSuchElementException e) {
throw new RuntimeException("Web element is not visible or not clickable within timeout:" + element + "Time" + timeout, e);
        }
catch (WebDriverException e) {
wait.until(ExpectedConditions.visibilityOf(driver.findElement(element)));
 driver.findElement(element).click();
        }    }
public String getElementTextVisibilityOf(By element, int timeout) {
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(timeout));
try {
wait.until(ExpectedConditions.visibilityOfElementLocated(element));
return driver.findElement(element).getText();                }
 catch (NoSuchElementException e) {
```

throw new RuntimeException("Does not see the text of the element within timeout:" + element + "Time" + timeout, e);

} }

public void textInputField(By inputFieldElement, int timeout, String inputText) {

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(timeout));

try {

wait.until(ExpectedConditions.presenceOfElementLocated(inputFieldElement));

driver.findElement(inputFieldElement).sendKeys(inputText);

}

catch (NoSuchElementException e)

{throw new RuntimeException("Text input field is not present:" + inputFieldElement + "Time" + timeout, e);

} } }