МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
Тернопільський національний економічний університет
Факультет комп'ютерних інформаційних технологій
Кафедра комп'ютерної інженерії

Козак Роман Ігорович

Алгоритми управління процесами контролю якості програмного забезпечення / Algorithms for managing of software quality control processes

спеціальність:123 – Комп'ютерна інженерія
освітньо-професійна програма – Комп'ютерна інженерія

Випускна кваліфікаційна робота

Виконав студент групи КІм-21
Р. І. Козак
_____

Науковий керівник:
д.т.н., професор, Цмоць І. Г.
_____

ТЕРНОПІЛЬ - 2019

РЕЗЮМЕ

Магістерська робота на тему «Алгоритми управління процесами контролю якості програмного забезпечення» на здобуття освітньо-кваліфікаційного рівня “Магістр” зі спеціальності “Комп’ютерні системи та мережі” написана обсягом 63 сторінок і містить 14 ілюстрацій, 3 таблиці, 3 додатки та 50 джерел за переліком посилань.

Метою даної роботи є розроблення алгоритмів управління на моніторингу процесів контролю якості програмного забезпечення. Об’єкт дослідження – сфера розробки програмного забезпечення.

Методи досліджень базуються на використанні методів проектного менеджменту (для покращення ефективності процесів управління ризиками); системного та бізнес-аналізу (для аналізу та опису моделей і процесів); алгоритмізації та об’єктно-орієнтованого програмування (для проектування програмних засобів).

Наукова новизна одержаних результатів: удосконалено та систематизовано алгоритми і методики управління та моніторингу процесами контролю якості, що дає можливість більш ефективно планувати процес та ефективно контролювати стан якості програмного продукту під час розробки.

Розроблено алгоритм управління процесом контролю якості програмного забезпечення, що дозволило ефективно контролювати стан якості програмного продукту під час розробки. Спроектовано архітектуру програмної системи та інтерфейсу для інтеграції між сервером і клієнтом, що дозволило провести попереднє дослідження запропонованого рішення.

КЛЮЧОВІ СЛОВА: КОНТРОЛЬ ЯКОСТІ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, МЕТРИКИ ТЕСТУВАННЯ

RESUME

Master's thesis on "Algorithms for managing of software quality control processes" for obtaining the Master's degree in Computer Systems and Networking has 63 pages and contains 14 illustrations, 3 tables, 3 appendices and 50 sources for a list of links.

The purpose of this work is to develop control algorithms for monitoring software quality control processes. The object of study is the area of software development.

Research methods are based on the use of project management methods (to improve the effectiveness of risk management processes); system and business analysis (to analyze and describe models and processes); algorithms and object-oriented programming (for software design).

Scientific novelty of the obtained results: algorithms and methods of control and monitoring of quality control processes have been improved and systematized, which enables to plan the process more efficiently and effectively control the quality of the software product during development.

The algorithm of control of the process of quality control of the software was developed, which allowed to effectively control the quality of the software product during development. The architecture of the software system and interface for server-client integration has been designed, which allowed preliminary research of the proposed solution.

KEYWORDS: QUALITY CONTROL, SOFTWARE, TEST METRICS

ЗМІСТ

ВСТУП

Актуальність теми. Випускна кваліфікаційна робота виконана згідно вимог [1, 2] і присвячена дослідженню питань, пов'язаних з процесами контролю якості програмного забезпечення. Контроль якості є актуальним через те, що без розробки ефективних методик управління та контролю якості неможливо досягнути високої якості програмних продуктів.

Інформаційні технології тісно інтегрувалися в життя сучасної людини. Їх можна розглядати як елемент і функцію інформаційного суспільства, спрямовану на регулювання, збереження, підтримку і вдосконалення системи управління нового мережевого суспільства. Якщо протягом століть інформація і знання передавалися на основі правил і приписів, традицій і звичаїв, культурних зразків і стереотипів, то сьогодні головна роль відводиться технологіям. Крім того, вони впорядковують потоки інформації на глобальному, регіональному і локальному рівнях. Вони відіграють ключову роль у формуванні техноструктури, в підвищенні ролі освіти і активно впроваджуються в усі сфери соціально-політичного і культурного життя, включаючи домашній побут, розваги та дозвілля [3].

Для того щоб програмні системи ефективно виконували покладені на них функції, вони повинні бути якісними. Сьогодні висока якість сприймається як обов'язковий компонент програмного забезпечення. Тому дуже важливо інтегрувати процес контролю якості в процес планування і реалізації проектів розробки ПЗ з самого початку [4].

Мета і завдання дослідження. Метою даної роботи є розроблення алгоритмів управління на моніторингу процесів контролю якості програмного забезпечення.

Об’єкт дослідження – сфера розробки програмного забезпечення.

Предмет дослідження – контроль якості в контексті розробки програмного забезпечення.

Методи досліджень базуються на використанні методів проектного менеджменту (для покращення ефективності процесів управління ризиками); системного та бізнес-аналізу (для аналізу та опису моделей і процесів); алгоритмізації та об’єктно-орієнтованого програмування (для проектування програмних засобів).

Наукова новизна одержаних результатів: удосконалено та систематизовано алгоритми і методики управління та моніторингу процесами контролю якості, що дає можливість більш ефективно планувати процес та ефективно контролювати стан якості програмного продукту під час розробки.

Практичне значення отриманих результатів. Розроблено програмне забезпечення для автоматизації управління процесом контролю якості програмного забезпечення. Експериментально доведена ефективність запропонованих алгоритмів.

Впровадження результатів. Результати роботи планується використати в роботі науково-дослідному інституті інтелектуальних комп’ютерних систем (додаток В).

Публікації та апробація випускної кваліфікаційної роботи. Отримані результати апробовані в межах Науково-практичної конференції молодих вчених та студентів «Інтелектуальні комп’ютерні системи та мережі» і ІІ Науково-практичної конференції молодих вчених та студентів «Інтелектуальні комп’ютерні системи та мережі» та опубліковано дві тези доповіді по темі роботи [5, 6].

Випускна кваліфікаційна робота складається із трьох розділів, висновків, списку використаної літератури та додатків. У першому розділі систематизовано підходи до побудови процесу контролю якості ПЗ.

В другому розділі розроблено алгоритм управління та моніторингу процесом контролю якості програмного забезпечення.

У третьому розділі здійснено програмну реалізацію розроблених алгоритмів та системи автоматизації процесу контролю якості. Проведено експериментальне дослідження розроблених алгоритмів.

1 ХАРАКТЕРИСТИКА ПРОЦЕСІВ КОНТРОЛЮ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Поняття якості програмного забезпечення

Якість програмного забезпечення — це характеристика програмного забезпечення, ступінь відповідності ПЗ до вимог. При цьому вимоги можуть трактуватись по-різному, що породжує декілька незалежних визначень терміну. Якість ПЗ – набір властивостей продукту (сервісу або програм), що характеризують його здатність задовольнити встановлені або передбачувані потреби замовника. Поняття якості має різні інтерпретації залежно від конкретної програмної системи і вимог до неї.

Стандарт ISO/IEC 9126 регламентує зовнішні і внутрішні характеристики якості. Перші відображають вимоги до функціонування програмного продукту. Для кількісного встановлення критеріїв якості, за якими буде здійснюватися перевірка і підтвердження відповідності ПЗ заданим вимогам, визначаються відповідні зовнішні вимірювані властивості (зовнішні атрибути) ПЗ, метрики (наприклад, час виконання окремих компонентів), діапазони зміни значень і моделі їх оцінки. Метрики використовуються на стадії тестування або функціонування і називаються зовнішніми метриками. Вони являють собою моделі оцінки атрибутів. [7]

Внутрішні характеристики якості і внутрішні атрибути ПЗ використовуються для складання плану досягнення необхідних зовнішніх характеристик якості продукту. Для квантифікації внутрішніх характеристик якості застосовують внутрішні метрики, як інструмент перевірки відповідності проміжних продуктів внутрішнім вимогам до якості, які формулюються на процесах, що передують тестуванню.

Зовнішні і внутрішні характеристики якості відображають властивості самого ПЗ (працюючого або не працюючого), а також погляд замовника і розробника на таке ПЗ. Безпосереднього кінцевого користувача ПЗ цікавить експлуатаційна якість ПЗ – сукупний ефект від досягнення характеристик якості,

що виміряється строком результату, а не властивістю самого ПЗ. Це поняття ширше, ніж будь-яка окрема характеристика (наприклад, зручність використання або надійність).

Тестування програмного забезпечення (англ. Software Testing) — це процес технічного дослідження, призначений для виявлення інформації про якість продукту відносно контексту, в якому він має використовуватись. Техніка тестування також включає як процес пошуку помилок або інших дефектів, так і випробування програмних складових з метою оцінки [8].

Тестування програмного забезпечення - процес перевірки відповідності заявлених до продукту вимог і реально реалізованої функціональності, здійснюваний шляхом спостереження за його роботою в штучно створених ситуаціях і на обмеженому наборі тестів, обраних певним чином.

Може оцінюватись:

- відповідність вимогам, якими керувалися проектувальники та розробники;
- правильна відповідь для усіх можливих вхідних даних;
- виконання функцій за прийнятний час;
- практичність;
- сумісність з програмним забезпеченням та операційними системами;
- відповідність задачам замовника.

Оскільки число можливих тестів навіть для нескладних програмних компонент практично нескінченне, тому стратегія тестування полягає в тому, щоб провести всі можливі тести з урахуванням наявного часу та ресурсів. Як результат програмне забезпечення тестується стандартним виконанням програми з метою виявлення багів (помилок або інших дефектів).

Тестування ПЗ може надавати об'єктивну, незалежну інформацію про якість ПЗ, ризики відмови, як для користувачів так і для замовників[9].

Тестування може проводитись, як тільки створено виконуваний код (навіть частково завершено). Процес розробки зазвичай передбачає коли та як буде відбуватися тестування. Наприклад, при послідовному процесі, більшість тестів

відбувається після визначення системних вимог і тоді вони реалізуються в тестових програмах. На противагу цьому, відповідно до вимог гнучкої розробки ПЗ, програмування і тестування часто відбувається одночасно.

Для більш точного тлумачення якості був створений стандарт ISO/IEC 25010 [10], який впроваджує модель якості програмного продукту. Модель якості продукту можна застосовувати як для програмного продукту, так і для комп'ютерної системи, до складу якої входить програмне забезпечення, оскільки більшість підхарактеристик може бути застосовано і до програмного забезпечення, і до систем [11].

Відповідно до цього стандарту виділяють такі характеристики якості (схематично зображені на рисунку 1.1):

- функціональна придатність;
- рівень продуктивності;
- сумісність;
- зручність використання;
- надійність;
- захищеність;
- супроводжуваність;
- переносимість.

Підхарактеристиками функціональної придатності є:

- функціональна повнота;
- функціональна коректність;
- функціональна доцільність.

Підхарактеристиками рівня продуктивності є:

- часові параметри;
- використання ресурсів;
- ємність.

Підхарактеристиками сумісності є:

- співіснування;
- інтероперабільність.

Підхарактеристиками зручності використання є:

- визначеність придатності;
- вивчаємість;
- управляємість;
- захищеність від помилки користувача;
- естетика користувацького інтерфейсу;
- доступність.

Підхарактеристиками надійності є:

- завершеність;
- готовність;
- відмовостійкість;
- відновлюваність.

Підхарактеристиками захищеності є:

- конфіденціальність;
- цілісність;
- непідроблюваність;
- відслідковуваність;
- правдивість.

Підхарактеристиками супроводжуваності є:

- модульність;
- можливість багаторазового використання;
- аналізованість;
- модифікованість;
- тестованість.

Підхарактеристиками переносимості є:

- адаптованість;
- встановлюваність;
- взаємозаміняємість.

Об’єктом моделі якості продукту є комп'ютерна система, в яку входить цільової програмний продукт, а мета моделі якості при використанні - це сукупна

людино-машинна система, яка включає в себе і цільову комп'ютерну систему, і цільової програмний продукт [12]. У цільову комп'ютерну систему входять також комп'ютерне обладнання, нецільові програмні продукти, нецільові дані і цільові дані, які, в свою чергу, є об'єктом аналізу моделі якості даних. Цільова комп'ютерна система є частиною інформаційної системи, до складу якої можуть бути також включені одна або більше комп'ютерних систем і системи зв'язку, такі як локальна мережа та Інтернет. До складу інформаційної системи в більшій людино-машинної системи (такий як корпоративна система, вбудована система або великомасштабна система управління) можуть входити користувачі, технічна і фізична середовище використання. Рамки цільової системи визначаються виходячи з області застосування вимог або оцінки і з того, хто розглядається в якості користувачів.

Рисунок 1.1 – Характеристики моделі якості за стандартом ISO/IEC 25010

Моделі якості продукції можуть бути використані для визначення вимог, показників і виконання оцінки якості. Певні характеристики можуть використовуватися в якості контрольного списку для забезпечення детального дослідження вимог до якості, забезпечуючи таким чином основу для оцінки необхідних в процесі розробки систем наступних трудовитрат і дій. Характеристики в моделі якості при використанні і моделі якості продукту призначені для використання в якості набору при специфікації або оцінці якості програмного продукту або комп'ютерної системи.

Практично неможливо визначити або виміряти всі підхарактеристики для всіх частин великої комп'ютерної системи або програмного продукту.

Аналогічно в більшості випадків практично не застосовується визначення або вимірювання якості при використанні для всіх можливих сценаріїв завдань користувача. Відносна важливість характеристик якості залежить від цілей високого рівня і цілей проекту. У зв'язку з цим перед використанням для виділення з вимог тих характеристик і підхарактеристик, які найбільш важливі, модель повинна бути відповідним чином адаптована, а ресурси розподілені між різними типами показників в залежності від цілей зацікавлених осіб і цілей продукту.

1.2 Становлення процесів тестування та контролю якості програмного забезпечення

Перші програмні системи розроблялися в рамках програм наукових досліджень або програм для потреб міністерств оборони. Тестування таких продуктів проводилося строго формалізовано із записом всіх тестових процедур, тестових даних, отриманих результатів. Тестування виділялося в окремий процес, який починався після завершення кодування, але при цьому, як правило, виконувалося тим же персоналом [13].

У 1960-х багато уваги приділялося «вичерпному» тестуванню, яке повинно проводитися з використанням усіх шляхів у коді або всіх можливих вхідних даних. Було відзначено, що в цих умовах повне тестування ПЗ неможливе, тому що, по-перше, кількість можливих вхідних даних дуже велика, по-друге, існує безліч шляхів, по-третє, складно знайти проблеми в архітектурі та специфікаціях. З цих причин «вичерпне» тестування було відхилено й визнано теоретично неможливим.

На початку 1970-х тестування ПЗ розглядалося як «процес, спрямований на демонстрацію коректності продукту» або як «діяльність з підтвердження правильності роботи ПЗ» [14]. У програмній інженерії, яка в той час зароджувалася, верифікація ПЗ визначалася як «доказ правильності». Хоча

концепція була теоретично перспективною, на практиці вона вимагала багато часу й не охоплювала всі аспекти тестування. Було вирішено, що доказ правильності — неефективний метод тестування ПЗ. Однак, у деяких випадках демонстрація правильної роботи використовується і в наші дні, наприклад, приймально-здавальні випробування. У другій половині 1970-х тестування представлялося як виконання програми з наміром знайти помилки, а не довести, що вона працює. Успішний тест — це тест, який виявляє раніше невідомі проблеми. Даний підхід цілком протилежний попередньому. Зазначені два визначення являють собою «парадокс тестування», в основі якого лежать два протилежних твердження: з одного боку, тестування дозволяє переконатися, що продукт працює добре, а з іншого — виявляє помилки у ПЗ, показуючи, що продукт не працює. Друга мета тестування є більш продуктивною з точки зору покращення якості, оскільки не дозволяє ігнорувати недоліки ПЗ [15].

У 1980-х тестування розширилося таким поняттям як запобіганням дефектам. Проектування тестів — найбільш ефективний з відомих методів запобігання помилок. В цей же час почали висловлюватися думки, що необхідна методологія тестування, зокрема, що тестування повинно включати перевірки впродовж усього циклу розроблення, при цьому це має бути керований процес. В ході тестування треба перевірити не тільки зібрану програму, але й вимоги, код, архітектуру, самі тести. «Традиційне» тестування, яке існувало до початку 1980-х, відносилося тільки до компілювання готової системі (зараз це зазвичай називається системне тестування), але надалі тестувальники стали залучатися в усі аспекти життєвого циклу розроблення [16]. Це дозволяло раніше знаходити проблеми у вимогах та архітектурі й тим самим скорочувати терміни та бюджет розроблення. У середині 1980-х з'явилися перші інструменти для автоматизованого тестування. Передбачалося, що комп'ютер зможе виконати більше тестів, ніж людина, причому зробить це більш надійно. Спочатку ці інструменти були вкрай простими й не мали можливості написання сценаріїв на скриптових мовах [17].

На початку 1990-х у поняття «тестування» стали включати планування, проектування, створення, підтримку й виконання тестів та тестових оточень, а це означало перехід від тестування до забезпечення якості, що охоплює весь цикл розроблення ПЗ. У цей час починають з'являтися різні програмні інструменти для підтримки процесу тестування: більш просунуті середовища для автоматизації з можливістю створення скриптів і генерації звітів, системи управління тестами, ПЗ для проведення навантажувального тестування. У середині 1990-х з розвитком Інтернету й розробленням великої кількості веб-застосунків особливої популярності стало набувати «гнучке тестування» (за аналогією з гнучкими методологіями програмування).

У 2000-х з'явилося ще більш широке визначення тестування, коли в нього було додано поняття «оптимізація бізнес-технологій» [17]. Основний підхід полягає в оцінці та максимізації значущості всіх етапів життєвого циклу розроблення ПЗ для досягнення необхідного рівня якості, продуктивності, доступності.

1.3 Фундаментальний процес тестування та рівні тестування

Становлення ідеї фундаментального тестового процесу на всіх рівнях тестування зайняло роки. В рамках цього процесу можна виділити ключові етапи:

- планування і управління (planning and control);
- аналіз і проектування (analysis and design);
- впровадження та реалізація (implementation and execution);
- оцінка критеріїв виходу і написання звітів (evaluating exit criteria and reporting);
- дії по завершенню тестування (test closure activities).

Тут дії описані в логічній послідовності, але в умовах реального проекту вони можуть накладатися, відбуватися одночасно або навіть повторюватися. Зазвичай, відбувається адаптація цих кроків під потреби конкретної системи або проекту [19].

Планування тестування включає дії, спрямовані на визначення основних цілей тестування і завдань, виконання яких необхідне для досягнення цих цілей [20].

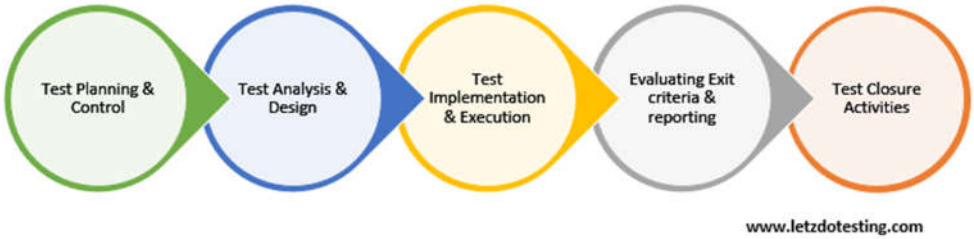

Рисунок 1.2 – Фундаментальний процес тестування

У процесі планування ми переконуємося в тому, що ми правильно зрозуміли цілі і побажання замовника і об'єктивно оцінили рівень ризику для проекту, після чого ставимо мети і завдання для, власне, тестування.

Для більш ясного опису цілей і завдань тестування складаються такі документи як тест-політика, тест-стратегія і тест-план [21].

Тест-політика – високорівневий документ, що описує принципи, підходи і основні цілі компанії в сфері тестування [21].

Тест-стратегія – високорівневий документ, що містить опис рівнів тестування і підходів до тестування в межах цих рівнів. Діє на рівні компанії або програми (одного або більше проектів) [22].

Тест-план – документ, що описує засоби, підходи, графік робіт і ресурси, необхідні для проведення тестування. Крім іншого, визначає інструменти тестування, функціональність, яку потрібно протестувати, розподіл ролей в команді, тестове оточення, використовувані техніки тест-дизайну, критерії

початку і закінчення тестування і ризики. Тобто, це докладний опис всього процесу тестування.

У будь-якій діяльності, управління не закінчується плануванням. Нам потрібно контролювати і вимірювати прогрес. Саме тому управління тестуванням - безперервний процес.

Управління тестуванням – зіставлення поточної ситуації в процесі тестування з планом і складання звітності [23].

У свою чергу, дані, отримані в ході контролю над процесом, враховуються при плануванні подальших дій.

Аналіз і проектування тестів – це процес написання тестових сценаріїв і умов на основі загальних цілей тестування.

У процесі аналізу і проектування ми розробляємо тестові сценарії на підставі загальних цілей тестування, визначених під час планування.

Тестовий сценарій – документ, що визначає встановлену послідовність дій при виконанні тестування.

Під час виконання тестування відбувається написання тест-кейсів, на основі написаних раніше тестових сценаріїв, збирається необхідна для проведення тестів інформація, готується тестове оточення і запускаються тести.

Тест-кейс – документ, що містить набір вхідних значень, перед- і постумовою, а також очікуваний результат проведення тесту, розроблений для перевірки відповідності певної функціональності системи заданим для цієї функціональності вимогам [24].

Тестове оточення (testware) – апаратне і програмне забезпечення та інші засоби, необхідні для виконання тестів [25].

Критерії виходу (exit criteria) визначають, коли можна завершувати тестування. Вони необхідні для кожного рівня тестування, оскільки нам необхідно знати, чи достатньо було проведено тестів [26].

При оцінці критеріїв виходу необхідно:

- перевірити, чи було проведено достатню кількість тестів, чи досягнута потрібна ступінь забезпечення якості системи;

- переконатися в тому, що немає необхідності проводити додаткові тести. Якщо все ж таки така необхідність є, можливо, буде потрібно змінити встановлений критерій виходу.

Після закінчення тестування відбувається написання звіту, який буде доступний всім зацікавленим сторонам. Адже не тільки тестувальники повинні знати результати виконання тестів, - ця інформація може бути необхідна багатьом учасникам процесу створення ПЗ [27].

При завершенні тестування ми збираємо, систематизуємо і аналізуємо інформацію про його результати. Вона може стати в нагоді пізніше - під час випуску готового продукту. Можуть бути й інші причини для згортання тестування, наприклад, дострокове закриття проекту або завершення певного етапу розробки.

Основні цілі цього етапу [28]:

- переконатися, що вся запланована функціональність дійсно була реалізована;

- перевірити, що всі звіти про помилки, подані раніше, були, так чи інакше, закриті;

- завершення роботи тестового забезпечення, тестового оточення та інфраструктури;

- оцінити загальні результати тестування і проаналізувати досвід, отриманий в його процесі.

Протягом всього процесу тестування відбувається на кількох різних рівнях (зображені на рисунку 1.3):

- модульний / компонентний (англ. module / component / unit level);

- інтеграційний (англ. integration level);

- системний (англ. system level);

- приймальний (англ. acceptance level).

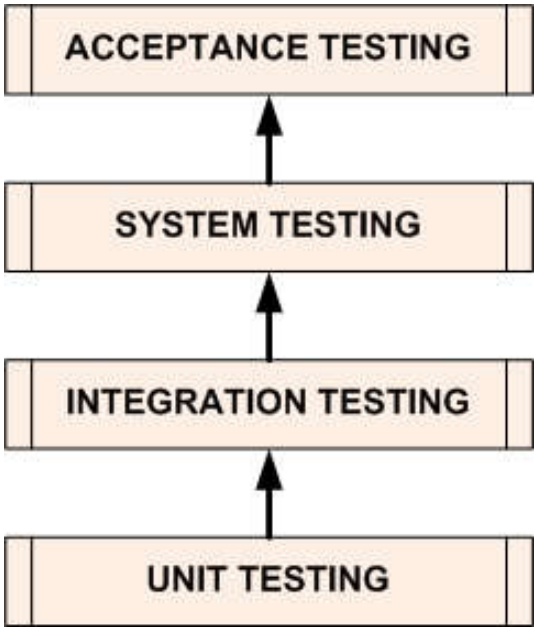

Рисунок 1.3 – Рівні тестування

Модульне тестування – тестування кожної атомарної функціональності програми окремо, в штучно створеному середовищі. Саме потреба в створенні штучної робочого середовища для певного модуля, вимагає від тестувальника знань в автоматизації тестування програмного забезпечення, деяких навичок програмування [29]. Дане середовище для деякого модуля створюється за допомогою драйверів і заглушок.

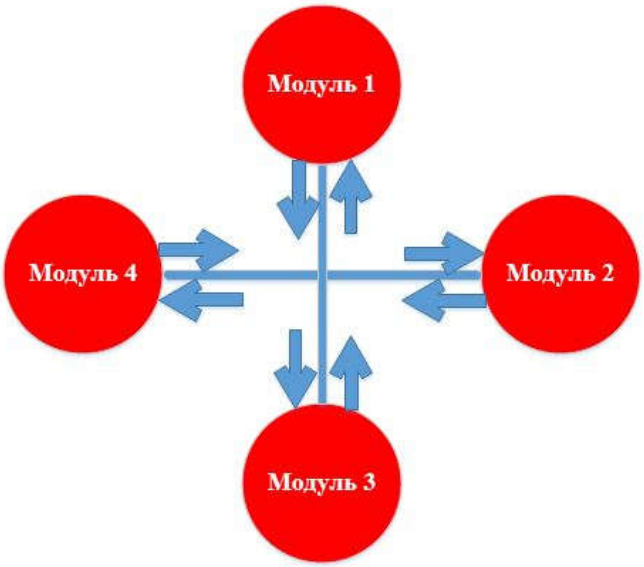

Рисунок 1.4 – Модульне тестування

Драйвер (англ. driver) – певний модуль тесту, який виконує тестований нами елемент [30].

Заглушка (англ. stub) – частина програми, яка симулює обмін даними з тестованим компонентом, виконує імітацію робочої системи.

Заглушки потрібні для:

- імітування відсутніх компонентів для роботи даного елемента;
- подачі або повернення модулю певного значення, можливість надати тестеру самому ввести потрібне значення;
- відтворення певних ситуацій (виключення або інші нестандартні умови роботи елемента).

Для модульного тестування необхідне виконання певних умов. Перш за все, потрібно окреслити межі, в яких модульне тестування виправдано. По-перше, архітектура проекту повинна бути спроектована відповідно до ідей ООП (чіткий розподіл на класи, кожен з яких виконує свою певну функцію), що забезпечить систему грамотним розподілом на модулі. Також, модульне тестування повинно бути менш витратним при пошуку дефектів, ніж інші види тестів і має знижувати час налагодження коду.

В той же час у такого тестування є такі переваги:

1) модульне тестування мотивує програмістів писати код максимально оптимізованим, проводити рефакторинг (спрощення коду програми, не зачіпаючи її функціональність), так як за допомогою модульного тестування можна легко перевірити працездатність даного компонента;

2) необхідність відділення реалізації від інтерфейсу (зважаючи на особливості модульного тестування), що дозволяє мінімізувати залежності в системі;

3) документація юніт-тестів може служити прикладом «живого документа» для кожного класу, що тестується даними способом;

4) модульне тестування допомагає краще зрозуміти роль кожного класу на тлі всієї програмної системи;

5) також, при «розробці через тестування», яка активно використовується в екстремальному програмуванні, модульне тестування є одним з основних інструментів, що дозволяє розробляти модулі відповідно до вимог до даного модулю.

Системне тестування – це тестування програмного забезпечення виконується на повної, інтегрованій системі, з метою перевірки відповідності системи вихідним вимогам, як функціональним, так і не функціональним [31].

Виконуючи системне тестування, можна виявити такі типи дефектів:

- неправильне використання системних ресурсів;
- непередбачені комбінації призначених для користувача даних;
- проблеми з сумісністю оточення;
- непередбачені сценарії використання;
- невідповідність з функціональними вимогами;
- погана зручність використання.

Системне тестування виконується методом «чорного ящика», тому що перевіряються лише ті сутності, які не вимагають взаємодії з внутрішньою будовою програми. Також виконувати його рекомендується в оточенні, максимально наближеному до оточення кінцевого користувача.

Можна виділити два підходи до системного тестування:

1) на базі вимог. Тестування проводиться відповідно до функціональних або нефункціональними вимогами, для кожного з яких пишеться Test Case (тестові прецеденти);

2) на базі випадків використання. Тестування відбувається відповідно до варіантів використання продукту, на основі яких створюються Use Cases (призначені для користувача прецеденти). Для кожного з даних користувача прецедентів створюються свої тестові прецеденти.

Виходячи з відмінностей між модульним тестуванням і системним тестуванням, інтеграційне тестування є перехідним етапом між поданням програми у вигляді окремих модулів у вид повністю функціональної системи.

Інтеграційне тестування – вид тестування, при якому на відповідність вимог перевіряється інтеграція модулів, їх взаємодія між собою, а також інтеграція підсистем в одну загальну систему. Для інтеграційного тестування використовуються компоненти, вже перевірені за допомогою модульного тестування, які групуються в безлічі. Дані безлічі перевіряються відповідно до плану тестування, складеним для них, а об'єднуються вони через свої інтерфейси.

Так як модулі з'єднуються між собою за допомогою передбачених реалізацією інтерфейсів і в процесі тестування у нас немає потреби розглядати внутрішню структуру компонентів, можна стверджувати, що інтеграційне тестування виконується методом «чорного ящика».

Існує кілька підходів до інтеграційного тестування:

1) знизу вгору (bottom-up). Спочатку збираються і тестуються модулі самих нижніх рівнів, а потім по зростанню до вершини ієрархії. Даний підхід вимагає готовності всіх зібраних модулів на всіх рівнях системи;

2) зверху вниз (top-down). Даний підхід передбачає рух з високорівневих модулів, а потім прямує вниз. При цьому використовуються заглушки для тих модулів, які знаходяться нижче за рівнем, але включення яких в тест ще не відбулося;

3) великий вибух (big bang). Всі модулі всіх рівнів збираються воєдино, а потім тестується. Даний метод економить час, але вимагає ретельного опрацювання тест кейсів.

При автоматизації тестування використовується система безперервної інтеграції. Принцип її дії полягає в наступному:

1) система безперервної інтеграції проводить моніторинг системи контролю версій;

2) при зміні вихідних кодів в репозиторії проводиться оновлення локального сховища;

3) виконуються необхідні перевірки і модульні тести;

4) тексти програм компілюються в готові виконувані модулі;

5) виконуються тести інтеграційного рівня;

6) генерується звіт про тестування.

Це дозволяє тестувати систему відразу після внесення змін, що істотно скорочує час виявлення та виправлення помилок.

Приймальне (acceptance) – рівень тестування, що проводиться на етапі здачі готового продукту (або готової частини продукту) замовнику. Метою приймального тестування є визначення готовності продукту, що досягається шляхом проходу тестових сценаріїв і випадків, які побудовані на основі специфікації вимог щодо розроблюваного ПЗ.

Результатом приймального тестування може стати:

- відправлення проекту на доопрацювання;

- ухвалення його замовником як виконаного завдання.

Це фінальний етап тестування продукту перед його релізом. При цьому, він не є всеохоплюючим і повним – тестується, здебільшого, тільки основний функціонал.

Приймальне тестування проводититься або самим замовником, або групою тестувальників, які представляють інтереси замовника, або тестувальниками компанії-розробника. Залежить від уподобань замовника.

Всі види тестування програмного забезпечення, в залежності від поставлених цілей, можна умовно розділити на наступні групи:

- функціональні (англ. functional);

- нефункціональні (англ. non-functional);

- структурні (англ. structural);

- пов'язані зі змінами (англ. related to changes).

У функціональному тестуванні в основному виконується тестування функцій компоненту або системи. Функціональний тест, зазвичай, відповідає на такі запитання як «чи може користувач це робити» або «чи працює ця конкретна функція». Це все, як правило, описано в специфікації вимог або у функціональних специфікаціях.

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути виміряні різними величинами. В цілому, це тестування того, «як система працює».

Нефункціональне тестування включає в себе:

- тестування надійності (reliability testing);
- тестування юзабіліті (usability testing);
- тестування ефективності (efficiency testing);
- тестування підтримуваності (maintainability testing);
- тестування документації (documentation testing);
- тестування витривалості (endurance testing);
- тестування навантаження (load testing);
- тестування продуктивності (performance testing);
- тестування сумісності (compatibility testing);
- тестування безпеки (security testing);
- тестування розширюваності (scalability testing);
- тестування відновлюваності (recovery testing);
- локалізаційне тестування (localization testing).

Структурне тестування часто називають «тестуванням білої чи скляної скриньки» або «тестуванням прозорої коробки», оскільки в даному виді тестування ми зацікавлені в тому, що відбувається «всередині системи / програми».

У структурному тестуванні тестувальники повинні мати знання про внутрішні реалізації компонентів, бути знайомими з архітектурою коду.

Під час структурного тестування інженер зосереджується на тому, як програмне забезпечення працює. Наприклад, можна застосувати структурний підхід, щоб дізнатися як працюють цикли усередині програмного забезпечення. Різні тестові випадки можуть бути згенеровані для виконання циклу один, два, і багато разів. Це може бути зроблено незалежно від функціональності програмного забезпечення.

Структурні випробування можуть бути використані на всіх рівнях тестування. Розробники застосовують структурне тестування для тестування компонентів, особливо там, де існує хороша підтримка інструментів для покриття коду. Структурні випробування також використовуються на системному і приймальному рівні.

Після проведення необхідних змін, таких як виправлення дефекту, програмне забезпечення повинно бути перетестовано для підтвердження того факту, що проблема була дійсно вирішена. Нижче перераховані види тестування, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності додатки або правильності здійсненого виправлення дефекту [32]:

- димове тестування (smoke testing);
- регресійне тестування (regression testing);
- санітарне тестування (sanity testing).

1.4 Аналіз існуючих засобів контролю якості програмного забезпечення

Аналізуючи поточну ситуацію на ринку, варто зазначити, що систем для управління процесом контролю якості ПЗ досить небагато. Здебільшого такі системи були розроблені досить давно і не завжди відповідають вимогам часу.

Однією з них є TestLink. Система є безкоштовною та володіє досить багатим функціоналом, дозволяє створювати вимоги, тест-плани та тест-кейси, однак вона не є інтуїтивно зрозумілою та складною у користуванні, дизайн не відповідає вимогам часу.

Іншим аналогом є qTest – система управління процесами контролю якості від компанії QASymphony. На відміну від попередньої система має приємніший дизайн. Система зручна та інтуїтивно зрозуміла, однак вона має дещо меншу кількість функціоналу в порівнянні з попередньою.

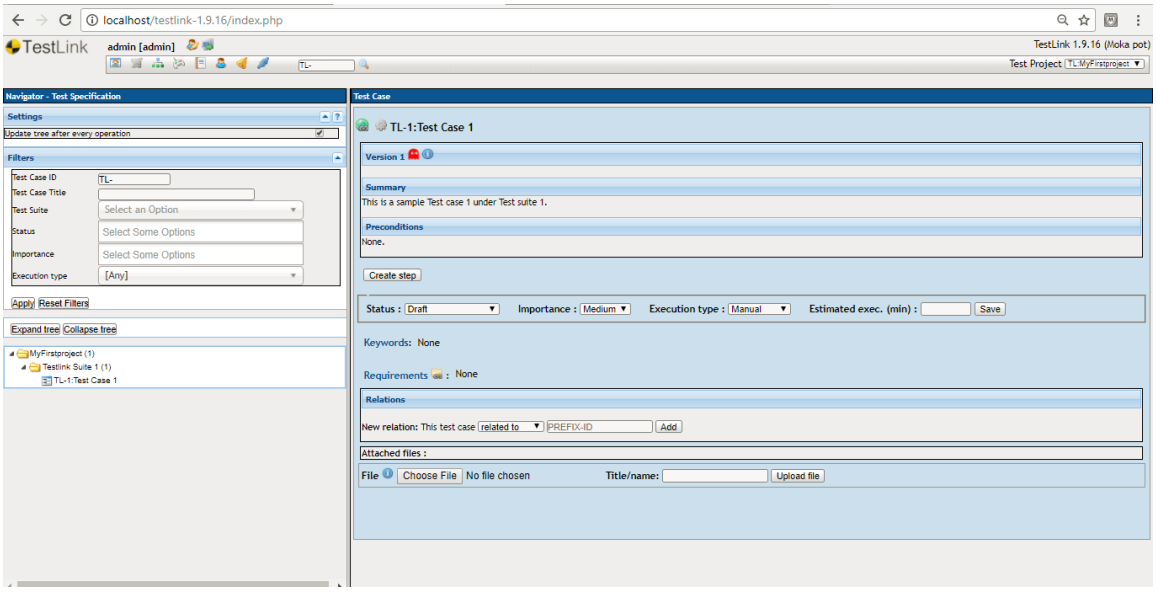

Рисунок 1.5 – Інтерфейс системи TestLink

Zephyr – ще одна система управління процесами контролю якості. Zephyr не є повноцінною незалежною системою, а лише плагіном для системи відстеження помилок JIRA. Це і є основним недоліком, оскільки JIRA – це пропрієтарне програмне забезпечення, яке доступне в основному тільки за корпоративними ліцензіями.

TestRail – одна з найбільш вдалих реалізацій ідей управління процесами контролю ПЗ. Це система з хорошим дизайном та навігацією. Єдиним та все ж значним недоліком є складність розгортання системи на власному сервері.

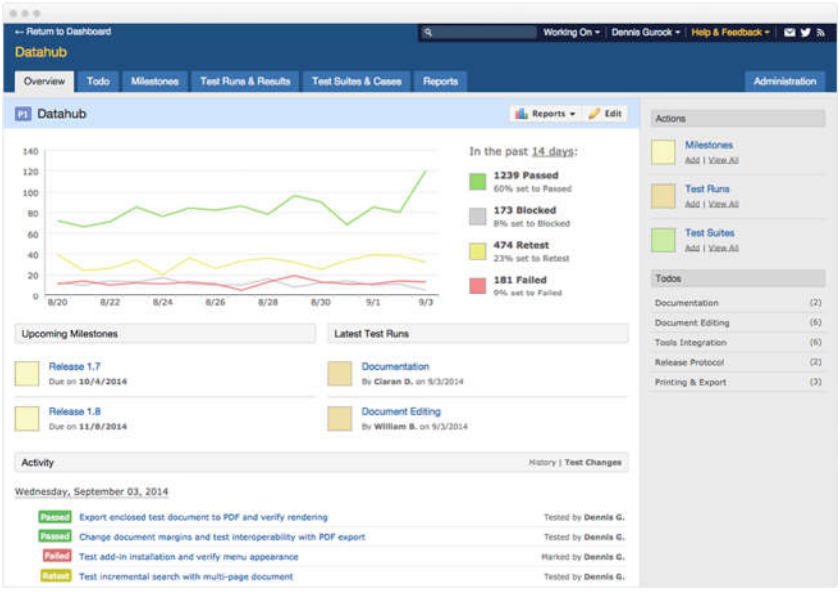

Рисунок 1.6 – Інтерфейс системи TestRail

Провівши аналіз існуючих аналогів, можна дійти до висновку, що у всіх є вагомі недоліки, уникнення яких дозволить зробити новий та успішний продукт.

1.5 Постановка задач дослідження

Якість — один із найважливіших параметрів проекту поряд з часом, вартістю і ресурсами. Але на відміну від іншим параметрів, якість дуже важко виміряти чи відобразити у цифрах. Висока якість та відповідність очікуванням кінцевих користувачів досягається завдяки точному визначенню потреб, ретельному плануванню та проектуванню, а також забезпеченню ефективного процесу контролю та моніторингу з можливістю якнайшвидшого внесення змін у проблемних ситуаціях.

Метою даної випускної кваліфікаційної роботи є розробка ефективних алгоритмів для управління процесами контролю якості програмного забезпечення.

Для досягнення поставленої мети необхідно виконати наступні задачі:

1. провести аналіз існуючих рішень для управління процесом контролю якості ПЗ;
2. розробити ефективні алгоритми планування та моніторингу за процесами контролю якості ПЗ;
3. реалізувати програмну систему на основі розроблених алгоритмів для автоматизації управління процесом контролю якості ПЗ на усіх його етапах.

2 АЛГОРИТМИ УПРАВЛІННЯ ПРОЦЕСАМИ КОНТРОЛЮ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Алгоритми та методи планування процесів контролю якості ПЗ

Планування тестування включає дії, спрямовані на визначення основних цілей тестування і завдань, виконання яких необхідне для досягнення цих цілей.

У процесі планування ми переконуємося в тому, що правильно зрозуміли цілі і побажання замовника і об'єктивно оцінили рівень ризику для проекту, після чого ставимо мети і завдання для, власне, тестування.

Для кожного рівня тестування планування починається на початку процесу тестування і триває протягом проекту аж до завершення активностей в рамках рівня. Воно включає в себе визначення активностей і ресурсів, необхідних для досягнення цілей і завдань, визначених у стратегії тестування. Планування тестування також включає виявлення методів збору і відстеження метрик, які будуть використовуватися при управлінні проектом, визначення ступеня відповідності з планом і оцінці досягнення цілей.

Рисунок 2.1 – Алгоритм процесу планування

При визначенні важливих метрик на етапах планування можна вибрати інструменти, запланувати графік навчання і розробити приблизну документацію.

Стратегія (або стратегії) тестування, вибрані для проекту, допомагає виявити завдання, які потрібно виконати в ході етапів планування [33].

Наприклад, при використанні стратегії на основі ризиків аналіз ризиків використовується для управління процесом планування тестування з метою зниження ідентифікованих продуктових ризиків і для допомоги в плануванні на випадок надзвичайних ситуацій. Якщо виявлено велику кількість ймовірних і потенційно серйозних дефектів, пов'язаних з безпекою, то потрібно докласти серйозних зусиль для тестування безпеки. Крім того, якщо буде встановлено, що в специфікації проектування виявлені серйозні дефекти, то процес планування тестування може призвести до додаткового статичного тестування (рецензування) специфікації проектування.

Інформація про ризик може також бути використана для визначення пріоритетів різних активностей по тестуванню. Наприклад, в тих випадках, коли продуктивність системи є ризиком високого рівня, то тестування продуктивності може проводитися відразу, як тільки інтегрований код стане доступним. Схожим чином, якщо застосовується реактивна стратегія, то є доцільним впровадження тестування і програмних засобів динамічного тестування, таких як дослідницьке тестування.

Крім того, на етапі планування тестування тест-менеджер чітко формулює підхід до тестування, який включає задіяні рівні тестування, цілі і завдання кожного рівня, а також методи тестування, що використовуються на кожному рівні тестування. Наприклад, при тестуванні, заснованому на ризиках, для авіаційних електронних систем оцінка ризику передбачає необхідний рівень покриття коду і, отже, визначає, які методи тестування можуть бути використані.

Можуть існувати складні зв'язки між базисом тестування (наприклад, конкретними вимогами або ризиками), тестовими умовами і тестами, які покривають їх. У робочому продукті часто існують зв'язку «багато-до-багатьох».

Це необхідно розуміти для здійснення ефективного моніторингу, планування і контролю тестування. Вибір інструмента також залежить від розуміння принципів роботи ПЗ.

Зв'язки можуть також існувати між результатами, отриманими командою розробників і командою тестувальників. Наприклад, матриця трасування (traceability matrix) може потребувати відстеження зв'язків між елементами детальної специфікації проектування проектувальників системи, бізнес-вимогами бізнес-аналітиків і результатами тестування, отриманими командою тест-інженерів. Якщо повинні бути спроектовані і використані низькорівневі сценарії тестування, то вимога, описана на етапі планування, деталізована в документах для команди розробки, має бути погоджено перед створенням сценаріїв тестування. При проходженні життєвого циклу на основі методології Agile неформальні активності по передачі інформації можуть використовуватися для взаємодії між командами перед початком тестування.

У плані тестування можливе перерахування конкретних властивостей ПЗ, які входять в область тестування (засноване на аналізі ризиків, якщо необхідно), а також явно певних властивостей, які виходять за рамки області тестування. Залежно від рівня формалізації та відповідної документації проекту кожна функція може бути пов'язана з відповідною специфікацією проектування тестів.

На цьому етапі може також існувати потреба для керівника тестування взаємодіяти з архітектором проекту для визначення початкової специфікації тестового оточення, перевірки доступності необхідних ресурсів, гарантій того, що люди, які будуть конфігурувати оточення, зацікавлені в цьому і мають уявлення про вартість робіт і терміни, необхідних для підготовки і настройки тестового оточення [34].

Нарешті, повинні бути визначені всі зовнішні залежності, пов'язані з угодою про рівень надання послуг (SLA) і, якщо необхідно, повинен бути встановлений початковий контакт. Прикладами залежностей є запити на ресурси до зовнішніх груп, які залежать від інших проектів (які працюють в рамках

програми), зовнішніх розробників ПЗ або партнерів по розробці, команди впровадження і адміністраторів баз даних.

Дуже важливим етатом при плануванні є оцінка трудозатрат на тестування продукту. Неправильна оцінка може бути призвести до фінансових втрат для бізнесу або ж недостатньої якості тестованого ПЗ.

Одним з найбільш популярних методів є метод трьох точок. Він передбачає три прогнози – оптимістичний, реалістичний та песимістичний. Розрахунок кінцевої оцінки відображений у формулі 2.1.

$$E = \frac{E_O + 4 \times E_L + E_P}{6}, \quad (2.1)$$

де E_O – це оптимістична оцінка;

E_L – найбільш вірогідна оцінка;

E_P – песимістична оцінка.

2.2 Алгоритми та методи управління та моніторингу процесів контролю якості ПЗ

Для забезпечення ефективного контролю тест-менеджеру необхідно створити графік тестування і схему моніторингу. схема повинна включати докладні метрики і цілі, які повинні зв'язати статус робочих продуктів тестування, заплановані активності і стратегічні цілі.

Для невеликих і нескладних проектів може бути відносно легко зв'язати робочі продукти тестування, заплановані активності і стратегічні мети. Зазвичай для досягнення цього потрібно визначити більш детальні цілі [35].

Вони можуть включати метрики і критерії, відповідно до встановлених цілей тестування і покриттю базису тестування. Важливо мати можливість зв'язати статус робочих продуктів тестування і активностей, пов'язаних з базисом

тестування, способом, який є зрозумілим і придатним для учасників проекту та учасників з боку бізнесу.

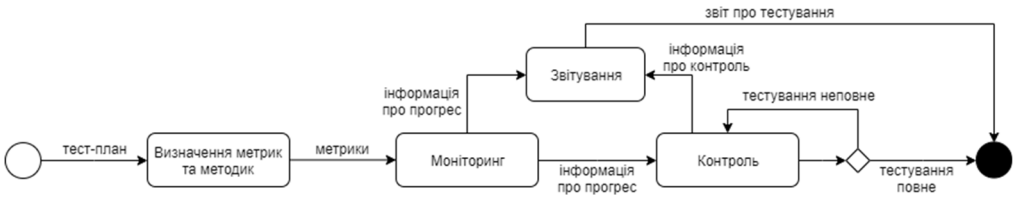

Рисунок 2.2 – Діаграма станів для процесу моніторингу і контролю

Визначення цілей і вимір прогресу на основі тестових умов і груп тестових умов можна використовувати в якості способу їх досягнення, пов'язуючи робочі продукти тестування ПЗ з базисом тестування при допомоги тестових умов. Трасування, налаштована таким чином, щоб отримувати звіт про її статус, робить складні зв'язки, що існують між результатами розробки, базисом тестування і робочими продуктами по тестування ПЗ, більш прозорими і доступними для сприйняття.

Загальна ефективність процесу тестування може вимірюватися за наступною формулою

$$Testing\ Efficiency = \left(\frac{Def_R}{Def_S}\right) \times 100, \tag{2.2}$$

де Def_R – це кількість вирішених дефектів;

Def_S – загальна кількість.

Іноді детальні вимірювання та цілі, які учасники хочуть моніторити, безпосередньо не пов'язані з функціональністю або специфікацією, особливо якщо документації мало або вона зовсім відсутня. Наприклад, учасник з боку бізнесу може бути більше зацікавлений в створенні покриття за рахунок операційного бізнес-циклу, хоча специфікація визначена в термінах функціональності системи. Залученість учасників проекту з боку бізнесу на ранніх етапах проекту може допомогти визначити критерії оцінки і мети, які

можна використовувати не тільки для забезпечення кращого контролю проекту, але також для актуалізації та визначення додаткових активностей тестування в рамках всього проекту. Наприклад, критерії оцінки учасників і мети можуть призвести до структуризації проектування тестів і реалізації тестування робочого продукту і / або перевірці графіка виконання тестування ПЗ для полегшення точного моніторингу ходу тестування для цих критеріїв оцінки. Ці цілі також допомагають забезпечити відстеження конкретного рівня тестування і мають можливість надавати інформацію про зв'язок між різними рівнями тестування.

Контроль тестування – це постійна активність. Вона включає порівняння з планом фактичного прогресу виконання активностей з тестування та усунення недоліків, коли це необхідно. Контроль тестування дозволяє керувати тестуванням для виконання цілей, стратегії і задач, включаючи при необхідності активності з планування тестування. робляться аналіз керуючих даних залежить від деталізації інформації в передбачуваній інформації.

Універсальним завданням управління тестуванням є правильний вибір, розподіл і пріоритизація тестів. Тобто, з практично нескінченного числа тестових умов і сполучень умов, які можуть бути покриті, команда тестування повинна вибрати кінцевий набір умов, визначити відповідні трудовитрати, щоб розподілити покриття кожного умови тестовим сценарієм, і впорядкувати отримані тестові сценарії в пріоритетному порядку, який оптимізує ефективність і результативність планованої роботи тестування. Поряд з іншими факторами керівником тестування, можуть бути використані виявлення і аналіз ризику, щоб допомогти вирішити цю проблему, хоча багато взаємодіючі обмеження і змінні можуть зажадати компромісного рішення.

Ризик – це можливість негативного або небажаного результату або події. Ризики існують щоразу, коли може відбутися деяка проблема, яка зменшила б сприйняття якості продукту або успіху проекту клієнтом, користувачем, учасником або зацікавленою стороною. Там де основна ймовірність цієї проблеми впливає на якість продукту, потенційні проблеми називаються

ризиками якості, ризиками продукту або ризиками якості продукту. Там, де основний вагомість цієї проблеми впливає на успішне завершення проекту, потенційні проблеми називаються ризиками проекту або ризиками планування.

У тестуванні, заснованому на ризиках, ризики якості виявляють і оцінюють під час аналізу з зацікавленими особами ризиків якості продукції. потім команда тестування розробляє, впроваджує і виконує тести для зниження ризиків якості. Якість включає в себе сукупність функцій, поведінки, характеристик і атрибутів, які впливають на задоволення клієнта, користувача і всіх зацікавлених осіб. Таким чином, ризик якості – це потенційна ситуація, коли проблеми якості можуть існувати в продукті. Приклади ризиків якості для системи включають в себе: невірні обчислення в звітах (функціональний ризик, пов'язаний з точністю), повільну реакцію на призначений для користувача введення (нефункціональний ризик, пов'язаний з ефективністю і часом відгуку), а також труднощі в розумінні екранів і полів (нефункціональний ризик, пов'язаний з практичністю і зрозумілістю).

Коли тести виявляють дефекти, тестування пом'якшує ризик якості, забезпечивши поінформованість про дефекти і можливості для боротьби з ними перед випуском. Якщо тести не знаходять дефекти, тестування пом'якшує ризик якості шляхом гарантії, щоб під час виконання тестових умов система працює правильно.

Тестування, засноване на ризиках, використовує ризики якості продукту для вибору тестових умов, для розподілу трудозатрат на тестування цих умов, а також для визначення пріоритетів отриманих тестових сценаріїв.

Існує різні методи тестування, заснованого на ризиках, з істотним змінами в типі і рівні зібраної документації, і рівня застосування формальності. Явно або неявно тестування, засноване на ризики, має на меті за допомогою тестування зменшити загальний рівень ризику якості, і, зокрема, зниження цього рівня ризику до прийнятного.

Тестування, засноване на ризиках, складається з чотирьох основних видів діяльності:

- визначення ризиків;

- оцінка ризиків;

- пом'якшення ризиків;

- управління ризиками.

Ці види діяльності перетинаються. Для найбільшої ефективності, визначення та оцінка ризиків повинна включати представників всіх груп зацікавлених осіб продукту і проекту, хоча іноді в проектної реальності в результаті деякі зацікавлені особи виступають в якості заступників для інших зацікавлених осіб. наприклад, в розробці вільно поширюваного програмного забезпечення невелика вибірка потенційних клієнтів може бути запрошена допомогти виявити потенційні дефекти, які можуть вплинути на використання програмного забезпечення найбільшою мірою; в цьому випадку вибірка потенційних клієнтів служить в якості заміни всієї можливої клієнтської бази. Через їх особливий досвід роботи з ризиками якості продукту і відмов, тестувальники повинні брати активну участь в процесі визначення і оцінки ризику.

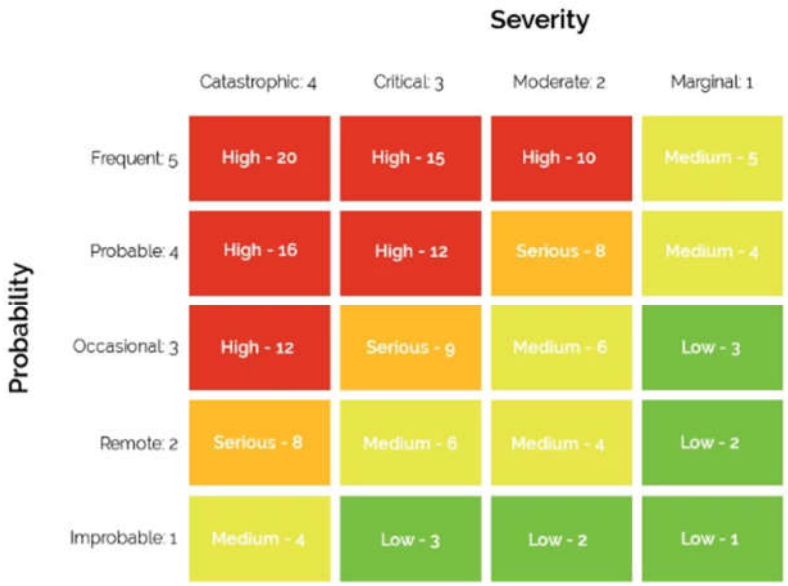

Рисунок 2.4 – Матриця ризиків

Зацікавлені особи можуть визначити ризики за допомогою одного або декількох з наступних методів:

- експертні інтерв'ю;

- незалежні оцінки;

- використання шаблону ризиків;

- ретроспектива проекту;

- групи фахівців для обговорення ризиків;

- мозковий штурм;

- чек-лісти;

- використання минулого досвіду.

Залучення ширшої вибірки зацікавлених осіб в процес визначення ризиків, швидше за все, визначить більшість значущих ризиків якості продукції [36]. Визначення ризику часто призводить до побічних продуктів, тобто визначенню проблем, які не є ризиком якості продукту. приклади включають загальні питання / проблеми продукту або проекту, проблеми в довідкових документах, таких як вимоги і специфікації дизайну. Ризики проекту також часто визначаються як побічний продукт визначення ризиків якості, але не є основним напрямком тестування, заснованого на ризиках.

В ідеалі, управління ризиками відбувається на протязі всього життєвого циклу. Якщо організація має документовану політику і / або стратегію тестування, то в них повинен бути описаний загальний процес управління ризиками проекту і продукту в тестуванні, і показано, як управління ризиками інтегровано і впливає всі етапи тестування.

Існує ще одна методика, яка може мати застосування у багатьох випадках. Методика сесійного тестування, розроблена Джеймсом Бахом (James Bach), полягає в поділі тестового навантаження на сеанси, кожен з яких вирішує свою задачу (отримання чітко визначених результатів, очікуваних від даного сеансу). Кожен сеанс має певну тривалість (від 20 до 40 хвилин), і тест-інженер повинен працювати безперервно протягом сеансу.

Тест-інженер наче поміщається на деякий час в замкнутий простір тестування, що дозволяє йому зосередитися на пошуку дефектів в програмному забезпеченні. В ході сеансу виконується набір контрольних тестів, але тест-

інженер може також виконувати тестування у вільному режимі. Таким чином, сесійне тестування являє собою суміш формального та інноваційного тестування, оскільки дає простір дослідження і інтуїції - тест-інженер отримує час і свободу дій, щоб виявити незвичайні дефекти або заглибитися в конкретні деталі програмного забезпечення.

В ході сеансу тест-інженер повинен документувати поведінку ПЗ, робити знімки станів і записувати поведінку ПЗ при конкретних вхідних даних і настройках. Після завершення сеансу його стенограма обговорюється з керівником групи або технічним менеджером. В ході обговорення визначається, яку поведінку можна вважати нормальною, а яку ні, і на основі цього обговорення створюються звіти про дефекти.

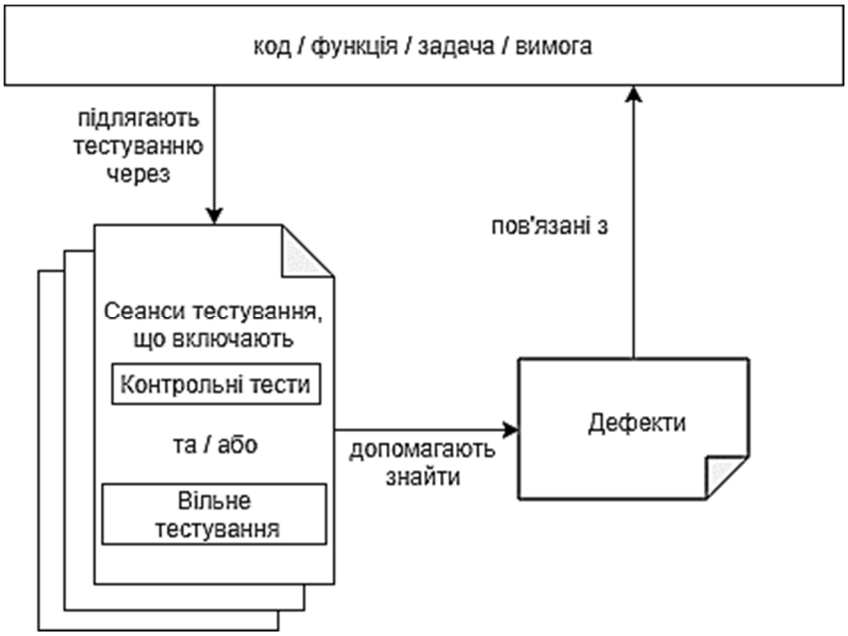

Рисунок 2.5 – Процес сесійного тестування

На рисунку 2.5 зображена методика сесійного тестування. Для будь-якої зміни в ПЗ плануються сеанси тестування, кожен з яких має певні цілі і завдання. Під час сеансу тестувальник виконує або контрольні тести, або вільне тестування, або і те, й інше. Після завершення сеансу складається звіт про виявлені дефекти.

2.3 Використання метрик в процесі контролю ПЗ

Метрики тестування можуть бути класифіковані як такі, що належать до однієї або до кількох категорій.

Метрики проекту, які вимірюють прогрес до встановлених у проекті критеріям виходу, таким як відсоток виконаних, пройдених і не пройдених сценаріїв тестування.

Метрики продукту, які вимірюють деякий атрибут продукту, такий як ступінь тестування або щільність дефектів. Як приклад можна навести як визначається покриття продукту вимогами.

$$Coverage_R = \left(\frac{Req_C}{Req_T}\right) \times 100 \quad (2.3)$$

Метрики процесу, які вимірюють можливості тестування або процесу розробки, такі як відсоток дефектів, виявлених при тестуванні. Як приклад варто навести як визначається витік дефектів.

$$Leakage_D = \left(\frac{Def_{UAT}}{Def_T}\right) \times 100 \quad (2.4)$$

Людські метрики, які вимірюють можливості окремих співробітників або груп, таких як реалізація сценаріїв тестування в рамках даного проекту-графіка.

Будь-яка надається метрика може відноситися до двох, трьох або навіть чотирьох категорій. Наприклад, діаграма спрямованості, що демонструє частоту надходження дефектів в день, може бути пов'язана з критеріями виходу (нуль нових дефектів, знайдених за тиждень), якістю продукту (тестування більше не може виявити в ньому дефектів) і можливістю процесу тестування (тестування виявило велику кількість дефектів на ранній стадії під час виконання тестів). Використання метрик дає можливість тестувальникам звітувати за результатами

в несуперечливої формі і включати зв'язок з фіксацією прогресу протягом довгого часу. Керівники тестування часто вимагають представляти метрики на нарадах, які можуть відвідувати зацікавлені боку різного рівня, починаючи від технічного персоналу до вищого керівництва. Так як метрики часто використовуються при визначенні загального успіху проекту, потрібна особлива обережність при визначенні того, що відстежувати, як часто готувати звіт і який метод використовувати при наданні інформації. При цьому керівник тестування повинен враховувати наступне.

Визначення метрик. Повинен бути визначений обмежений набір корисних метрик. Метрики повинні бути визначені відповідно до конкретними цілями проекту, процесу і / або продукту. метрики повинні бути збалансовані, так як окрема метрика може створити невірне враження про статус або тенденції. Так як метрики були визначені, їх адаптація повинна бути узгоджена всіма зацікавленими сторонами, щоб уникнути безладу під час обговорення вимірювань. при цьому часто існує тенденція визначати занадто багато метрик замість того, щоб концентруватися на найбільш підходящих.

Відстеження метрик. Підготовка звіту та об'єднання метрик повинно бути автоматизовано настільки, наскільки можливо для скорочення часу на збір і виконання вимірювання. Варіанти вимірювання конкретної метрики протягом тривалого часу можуть відобразити інформацію, крім інтерпретації, узгодженої під час етапу визначення метрик. Керівник тестування повинен бути готовий уважно аналізувати можливе відхилення вимірювання від очікуваного, і причини цієї розбіжності.

Звітність по метриках. Мета – забезпечити кожного безпосереднім розумінням інформації для цілей управління. презентації можуть демонструвати поточний стан метрики в певний час або показувати динаміку метрики протягом тривалого часу, так щоб можна було оцінити тенденції.

Точність метрик. Керівник тестування може тільки перевіряти інформацію, яка була надана. Зроблені для метрик вимірювання можуть не відображати справжній статус проекту або можуть відображати занадто

позитивні або негативні тенденції. Перед будь-якою датою презентації керівник тестування повинен прорецензувати їх на предмет точності і щодо даних, які вони ймовірно передають.

Важливо, щоб керівник тестування розумів, як інтерпретувати і використовувати покриття метрик для розуміння і відстеження статусу тестування. Для більш високих рівнів тестування, таких як системне тестування, системне інтеграційне тестування і приймальне тестування, найважливішою основою тестів є зазвичай робочі продукти, такі, як специфікації вимог, специфікація дизайну, сценарії використання, призначені для користувача історії, ризики продукту, підтримувані оточення і підтримувані конфігурації. Структурні метрики покриття коду застосовуються частіше для низьких рівнів тестування, таких як модульного тестування (наприклад, покриття умов і гілок) і компонентне інтеграційне тестування (наприклад, покриття інтерфейсу). В той час як керівник тестування може використовувати метрики покриття для вимірювання ступеня, до якої ці тести вивчають структуру тестованої системи, звітність результатів високорівневих тестів, як правило, не повинна включати метрики покриття коду. Крім того, керівник тестування повинен розуміти що, навіть якщо модульне тестування та компонентно-інтеграційне тестування на 100% досягнуто цілей структурного покриття, дефекти і ризики якості залишаються для більш високих рівнів тестування.

Для заданого набору метрик вимірювання можуть бути представлені усно в оповідній формі, в цифровій формі, у вигляді таблиць і графіків.

Вимірювання можуть бути використані в різноманітних цілях, включаючи:

1) аналіз, який виявляє, які тенденції і причини можуть бути виявлені в результатах тестів;

2) звіт, який повідомляє результати тестування учасникам або зацікавленим сторонам проекту;

3) контроль, який змінює напрямок тестування або проекту в цілому, і відстежує результати коригування напрямку.

З метою контролю тестування важливо, щоб метрики під час процесу тестування забезпечували керівника тестування необхідною інформацією для успішного завершення місії тестування, стратегій і цілей. Більш того, планування має враховувати цю необхідну інформацію, і моніторинг повинен включати збір всіх необхідних метрик робочих продуктів. Обсяг необхідної інформації і зусилля, витрачені на збір, залежать від різних факторів проекту, включаючи розмір, складність і ризик.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ УПРАВЛІННЯ ПРОЦЕСАМИ КОНТРОЛЮ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Узагальнена структура та архітектура програмної системи

Для реалізації програмного рішення необхідно підібрати цілий набір інструментів. Потрібно обрати сервер, технологію та мову програмування, систему управління базами даних та допоміжні компоненти.

В якості сервера можна використати популярний Apache, nginx або ж середовище Node.js. Але оскільки мова програмування та сервер є дуже пов'язаними компонентами, то варто обирати їх разом, а не окремо. Отож, варіантів кілька: PHP, Node.js, Python, Java. Методом аналізу можливостей та недоліків всіх технологій було одразу відкинуто варіанти з PHP та Java. В PHP є певні проблеми з швидкодією при великих навантаженнях, а з Java можуть виникнути проблеми при налаштуванні середовища на сервері.

Вибір між Python та Node.js видався дуже складним. Перший приваблює своєю простотою та різноманіттям вбудованих можливостей, зокрема і для роботи з базою даних. Node.js – це швидкий та потужний інструмент для розробки REST API [36]. Але найбільшим плюсом Node.js є його гнучкість. Розробнику надаються лише базові методи та концепції, все інше може бути реалізоване на власний розсуд. Саме тому в кінцевому результаті було вирішено будувати систему за допомогою Node.js.

Допоміжними компонентами при роботі з Node.js стали веб-фреймворк Express, бібліотека для роботи з базою даних Sequelize, модуль для логування Bunyan, бібліотека для попередження вразливостей Helmet, а також модуль для аутентифікації JWT JsonWebToken.

Найбільш популярною системою управління базами даних для Node.js є документ-орієнтована MongoDB. З реляційних баз даних можна вибирати між MySQL та PostgreSQL. MongoDB була відкинута через специфіку роботи з нею. Документ-орієнтована база даних буде складною в плані підтримки в

майбутньому, оскільки кількість спеціалістів з нереляційних баз даних на ринку доволі невелика [37].

MySQL є, можливо, найпопулярнішою, системою управління базами даних. MySQL проста у встановленні та використанні, вирізняється високою швидкістю виконання команд, а також наявністю ефективної системи безпеки. Втім, і у неї є свої недоліки. Зокрема мова запитів SQL, яка використовується в ній, не повністю відповідає міжнародному стандарту [16]. PostgreSQL, на відміну від попередньої, є ближчою до стандарту і її буде легше підтримувати у подальшому. Крім того, дана система управління базами даних має безліч додаткових можливостей, які можуть бути корисними при масштабуванні системи. Саме тому кінцевим вибором стала PostgreSQL.

Система буде складатися з багатьох компонентів, пов’язаних між собою. Архітектура системи зазначена на рисунку 2.2. Загальний принцип виглядає наступним чином. Клієнт надсилає HTTP-запит на веб-сервер з зазначенням потрібного порту. Якщо порт закріплений за процесом Node.js, то керування передається йому.

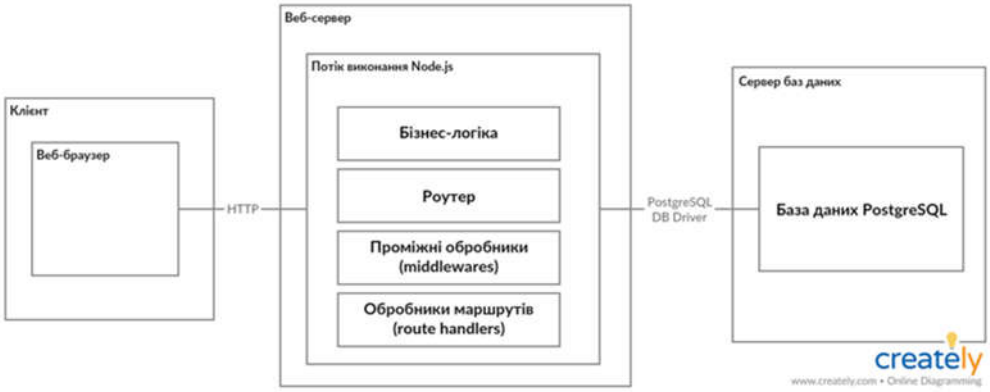

Рисунок 3.1 – Загальна архітектура системи

Першочергово дані запиту надходять на роутер – компонент, який відповідає за аналіз вхідних даних і делегування обробки запиту іншим компонентам. Компоненти системи, які відповідають за кінцеву обробку запиту називаються обробниками маршрутів. Вони перевіряють правильність даних від клієнта, вибирають потрібну інформацію з бази даних або ж формують

інформацію про помилку. На проміжному рівні – між роутером та обробником – працюють проміжні обробники (англ. middleware). Дані компоненти виконують певні типові операції над даними запиту, зокрема перевірки авторизації, валідності даних.

Окремо зберігається бізнес-логіка системи. До бізнес-логіки відносяться компоненти, які забезпечують зв'язок з драйвером баз даних, роботу з автентифікацією користувачів, логуванням системних помилок тощо.

Для звернень до бази даних використовується бібліотека Sequelize, яка базується на драйвері pg для PostgreSQL.

3.2 Структура серверного модуля

При розробці серверного модуля необхідно враховувати специфіку роботи з Node.js та фрейворком Express.

Express - це мінімалістичний та гнучкий фреймворк для веб-додатків, побудованих на Node.js, що надає широкий набір функціональності. Маючи в своєму розпорядженні безліч допоміжних HTTP-методів та проміжних обробників, створювати надійні API можна легко і швидко. Express забезпечує тонкий прошарок базової функціональності для веб-застосунків, що не спотворює звичну та зручну функціональність Node.js [38].

Express та всі інші потрібні бібліотеки поставляються як окремі модулі менеджера пакетів npm. Тому насамперед необхідно налаштувати менеджер пакетів. Конфігурація npm задає всі залежності, які буде використовувати система.

Наступним кроком буде задання структури папок і файлів. Отже, в кореневому каталозі буде міститись файл package.json, який задає налаштування npm, службовий файл системи контролю версій .gitignore, файл конфігурації бібліотеки Sequelize .sequelizerc та службові файли eslint .eslintignore, .eslintrc. Також кореневий каталог містить кілька папок, зокрема папку app, в якій

знаходиться основний код для роботи системи, папки migrations та seeders, які відповідають за зберігання міграцій і початкових даних для бази даних, і папку test, в яку помістимо інтеграційні тести.

Детальніше варто зупинитися на папці app. В ній є всього лиш два файли app.js і run.js. Вони відповідають за початковий запуск сервера і обробку критичних помилок. Окрім файлів тут також є такі папки: assets, config, database, forms, libs, middlewares, models і routes.

Перш за все треба винести загальні конфігурації системи в окремі файли. Вони будуть знаходитися в папці app/config. Конфігурації будуть різними залежно від середовища, в якому відбувається робота. Середовище задається в аргументах скрипта при старті сервера. Середовищ у системі буде три:

- development – система запускається на локальній машині розробника;
- production – система запускається на віддаленому сервері і готова для експлуатації кінцевими користувачами;
- test – система запускається на локальному або віддаленому сервері для запуску тестів.

Файли конфігурацій мають розширення .json і зберігають в собі інформацію про термін життя авторизаційних токенів, порт сервера, кореневі папки для логування та завантаження файлів, а також дані для з’єднання з базою даних PostgreSQL. Файли мають імена відповідно до середовища, для якого застосовуються. Додатково використовується модуль nconf, який відповідає за експорт потрібного конфігураційного файлу в інші модулі системи відповідно до середовища.

Важливим моментом є логування всіх операцій, які відбуваються в системі. Це дозволяє швидко знайти причину помилок при їх виникненні, а також при аналізі методів рефакторингу й оптимізації роботи системи. Для логування використовуємо зовнішню бібліотеку bunyan. Загальні конфігурацій модуля для логування знаходяться у файлі app/libs/logger.js.

Додатково у папці libs потрібно створити кілька власних бібліотек та службових файлів. Файли з закінченням .enum.js слугують для зручності роботи

з типами перерахування (enumeration). Бібліотека response-messages.js відповідає за збереження типів помилок та загального формату відповідей про помилки. Останні два файли form.js та validator.js є службовими модулями для обробки форм та валідації вхідних даних.

Наступним кроком є написання модуля роботи з базою даних. Допоміжною бібліотекою для роботи з PostgreSQL було обрано Sequelize. Перевагою бібліотеки Sequelize є автоматичне створення моделей та міграцій, а також можливість керування ними за допомогою скриптів npm. Початкове налаштування цієї бібліотеки записується у файл .sequelizerc у кореневому каталозі. У папці database основним буде файл postgres.js, в який помістимо функцію для з'єднання з базою даних. Для оптимізації взаємодії з сервером бази даних буде використовуватися пул з'єднань (connection pool) – своєрідний кеш для запитів, таким чином з'єднання можуть перевикористовуватися для майбутніх запитів. Пул з'єднань дозволяє значно збільшити продуктивність виконання команд в базі даних, оскільки у високонавантажених системах такі операції можуть бути часо- та ресурсозатратними.

Після першочергових і обов'язкових модулів можна переходити до написання модуля запуску сервера. Цим займається файл app/app.js, який містить функції для безпосереднього запуску сервера, перехоплення фатальних помилок та підключення інших основних модулів, зокрема модулів баз даних, генерації ключів для шифрування токенів аутентифікації, проміжних обробників та обробників маршрутів. Оскільки операції підключення до бази даних та запуску сервера можуть займати відносно багато часу та ресурсів, то вони виконуються асинхронно. Асинхронність реалізована за допомогою вбудованих в JavaScript об'єктів Promise.

У файлі app/run.js записуються функції для розподілу навантаження в багатоядерних процесорах серверів.

Основні компоненти серверного модуля готові і його запуск вже можливий. Тепер необхідно реалізувати функціональність системи.

Почати варто з моделей. Моделі являють собою правила для представлення даних структури у базі даних. Моделі є одних з базових понять концепції ORM (Object-Relational Mapping) – об'єктно-реляційної проекції. Дана концепція передбачає поєднання двох парадигм – об’єктно-орієнтованого програмування та реляційних баз даних. Модель представляється у вигляді об’єкту з властивостями, які позначають атрибути таблиці бази даних. В системі використовуватимуться вже раніше спроектовані моделі. Для генерації файлів моделей у Sequelize існують вбудовані скрипти.

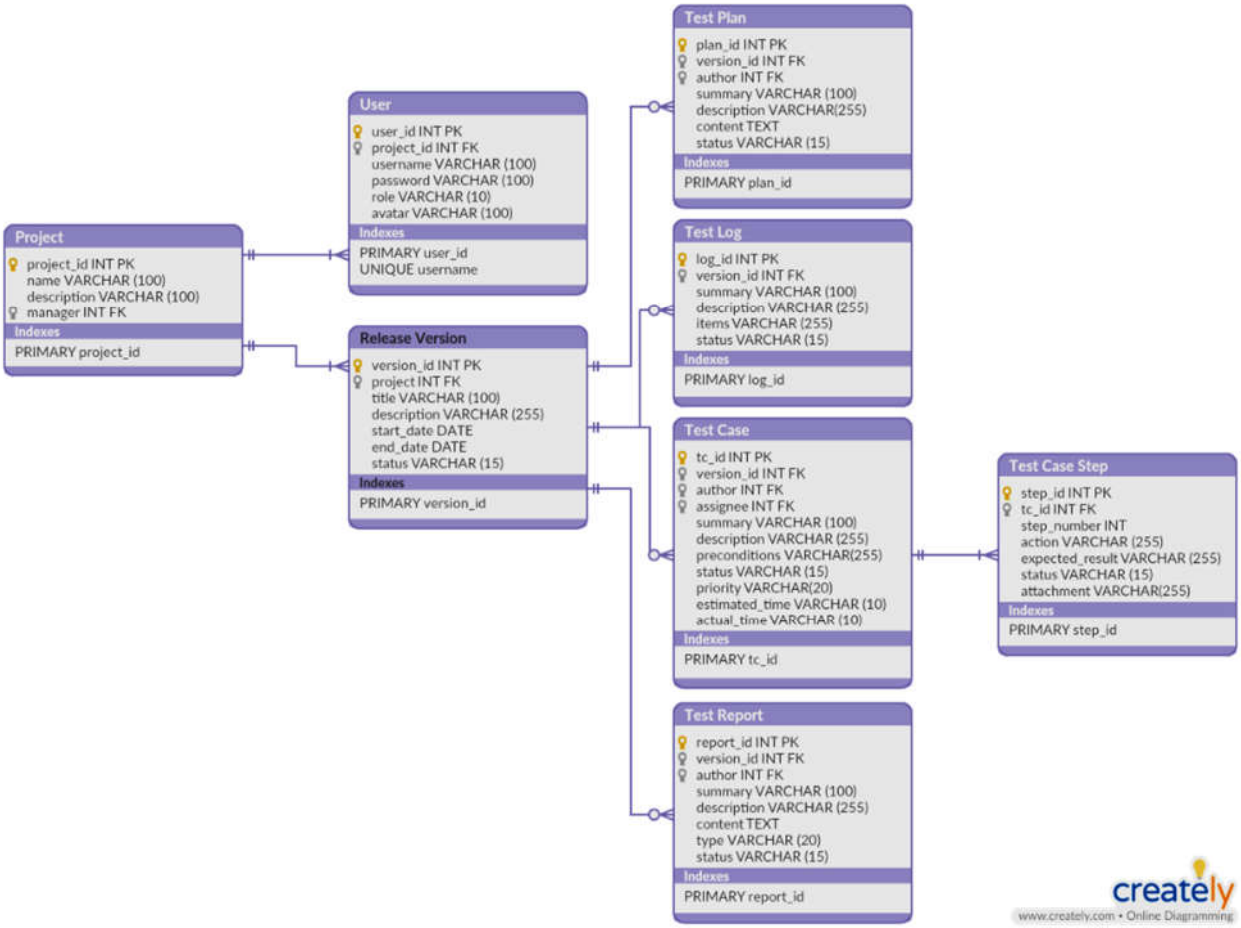

Рисунок 3.2 – Структура бази даних

На основі моделей будують форми. Форми являють собою загальні вимоги і критерії для вхідних даних. При надходженні запиту з корисним навантаженням (payload), яке може містити дані авторизації або ідентифікатори ресурсів, сервер повинен перевірити правильність формату цих даних, і в разі

невідповідності до зразка надіслати повідомлення про помилку. Зразком тут виступає саме форма. Форми зберігатимуться у папці app/forms.

Важливу роль при обробці вхідного запиту відіграють проміжні обробники (middleware). Концепція проміжних обробників передбачає використання окремих компонентів для зв'язку між компонентами, які отримують запит, і кінцевими обробниками маршрутів. Тобто вони вбудовуються в запит, виконують з ним певні операції і відправляють модифікований запит кінцевому обробнику. В системі будуть реалізовані одразу кілька проміжних обробників:

- auth (перевіряє чи користувач аутентифікований);
- error (виконує логування помилок під час виконання);
- logger (виконує логування всіх вхідних запитів);
- response-extender (розширює можливості вбудованого об'єкта відповіді сервера);
- roles (перевіряє роль та дозволи користувача);
- validator (перевіряє вхідні дані у запитах з корисним навантаженням);
- version (додає до відповіді заголовок Content-Version з актуальною версією системи).

Коли всі моделі, бібліотеки, форми та проміжні обробники реалізовані можна переходити до реалізації обробників маршрутів (route handlers). Якщо провести аналогію з концепцією Model-View-Controller, то обробник маршруту виконує роль вигляду і частково контролера. Кінцевий обробник формує дані перед відправкою клієнту. У фреймворку Express є вбудований клас Router, за допомогою якого можна з легкістю керувати всіма обробниками. Вбудовані механізми дозволяють також використовувати вкладені маршрутизатори, тобто призначати на маршрут не один обробник, а весь маршрутизатор, який зробить можливим обробку всіх похідних маршрутів. Під час побудови схеми маршрутів варто користуватися логікою і загальними практиками побудови REST API.

На найвищому рівні буде реалізовано два маршрутизатора – один для обробки маршрутів, які потребують авторизації, інший – для обробки всіх інших маршрутів. В перший буде включено проміжний обробник auth. На нижчих

рівнях працюватимуть маршрутизатори для обробки окремих запитів. В маршрути включаються відповідні проміжні обробники за необхідності. В системі використовуються кілька типів кінцевих точок (endpoint) маршрутів:

- запит на читання списку ресурсів (GET);
- запит на читання окремого ресурсу (GET);
- запит на створення ресурсу (POST);
- запит на модифікацію ресурсу (PUT);
- запит на часткову модифікацію ресурсу (PATCH);
- запит на видалення ресурсу (DELETE) [21].

Алгоритм роботи кінцевого обробника залежить від контексту його використання. В загальному випадку кінцевий розробник витягує дані з запиту і виконує асинхронні операції з моделлю.

Веб-сервіси можуть приймати одночасно кілька сотень, а то й тисяч запитів від різних клієнтів. Таким чином веб-сервіси – це високонавантажені системи, що від них вимагає максимальної надійності. Для забезпечення такого рівня надійності варто оптимізувати операції, які є найбільш довготривалими та ресурсозатратними. Цього можна досягти за допомогою принципів паралелізму. Паралелізмом називають властивість систем, коли декілька процесів обчислення відбуваються водночас. В мові JavaScript існує зручний механізм асинхронних операцій за допомогою об’єкту Promise. В реалізованій системі цей інструмент використовується повсюди, зокрема:

- операції читання/запису в файл;
- операції звернення до бази даних;
- операції звернення до зовнішніх сервісів.

Node.js позиціонується як платформа, що забезпечує достатню швидкодію серверних додатків, навіть незважаючи на те, що вона є однопоточною. Це відбувається завдяки потужному вбудованому механізму циклу подій (event loop). Згадані вище ресурсозатратні операції в даному циклі є блокуючими операціями, тобто вони блокують потік і є неможливим виконання інших

операцій до їх завершення. Інструменти асинхронності дозволяють обійти це обмеження.

Існує ще один ефективний метод балансування навантаження на рівні, близькому до апаратного. Завдяки окремому модулю cluster можна реалізувати систему в якості кластера. За замовчуванням додатки на платформі Node.js виконуються лише на одному логічному ядрі процесора. Модуль cluster дозволяє запустити по додатку для кожного ядра процесора. Це забезпечує додатковий приріст в швидкодії та продуктивності. Це досягається за допомогою досить простого алгоритму. Створюється батьківський процес (master process) і від нього породжуються дочірні процеси (worker). Кількість дочірніх процесів дорівнює кількості логічних ядер процесора. При кластеризації зберігається можливість вивантажити всі складні обчислення у фонові процеси і забезпечити комунікацію між ними через сервер черги повідомлень. Загальна схема зображена на рисунку 3.3.

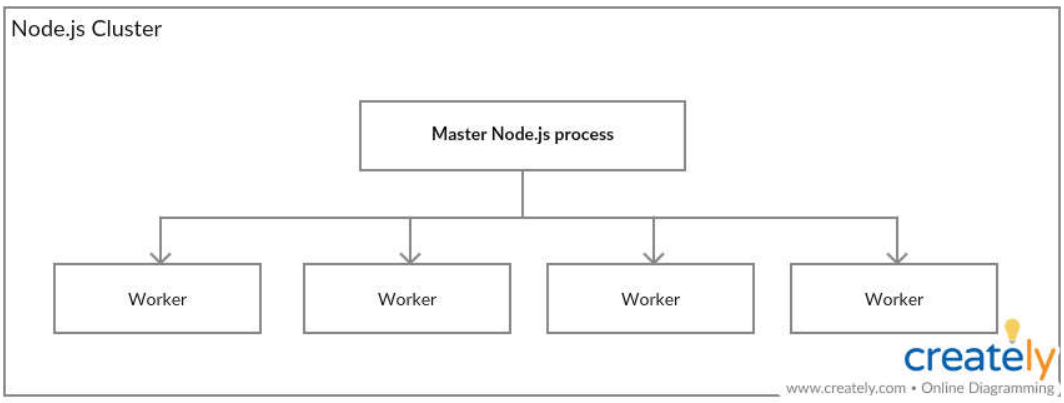

Рисунок 3.3 – Ілюстрація кластера у Node.js

У випадку, якщо визначеного приросту продуктивності буде недостатньо, можна використати окремий сервер у якості балансувальника навантаження. Принцип такого балансування ілюстрований на рисунку 3.3. Одним із найкращих варіантів для такої ролі може стати сервер nginx [39]. Він буде проміжним сервером між клієнтом і серверами Node.js та вирішуватиме на який сервер відправляти запит.

3.3 Структура клієнтської частини

Для розробки клієнтського модуля був обраний фреймворк Vue.js через його популярність, простоту застосування та широкі можливості масштабування.

Для створення готового середовища і структури проекту зазвичай використовують Vue CLI. Vue CLI – це набір інструментів для швидкого створення прототипів, легкого застосування методу скаффолдінгу та ефективного управління проектами. Він складається з трьох головних інструментів:

- CLI – глобально встановлений пакет npm, який забезпечує основну функціональність за допомогою команди vue. Це дозволяє нам легко скласти новий проект (vue create) або просто швидко переробити сирі ідеї (vue serve). Для більш конкретного та візуального контролю над нашими проектами, ми можемо відкрити GUI версію CLI, запустивши команду vue ui.
- CLI Service – це розробка залежностей (бінарний файл vue-cli-service), локально встановлений у кожен проект, створений за допомогою CLI. Це дозволяє нам розробити наш проект (vue-cli-service serve), упакувати його на продакшн (vue-cli-service build), а також перевірити конфігурацію внутрішнього вебпаку проекту (vue-cli-service inspect).
- CLI Plugins - це пакети npm, які надають додаткові можливості нашим проектам. Їх назви починаються з @vue/cli-plugin (для вбудованих плагінів) або vue-cli-plugin- (для плагінів спільноти). Ми можемо їх інтегрувати в процес розробки у будь-який час за допомогою команди vue add.

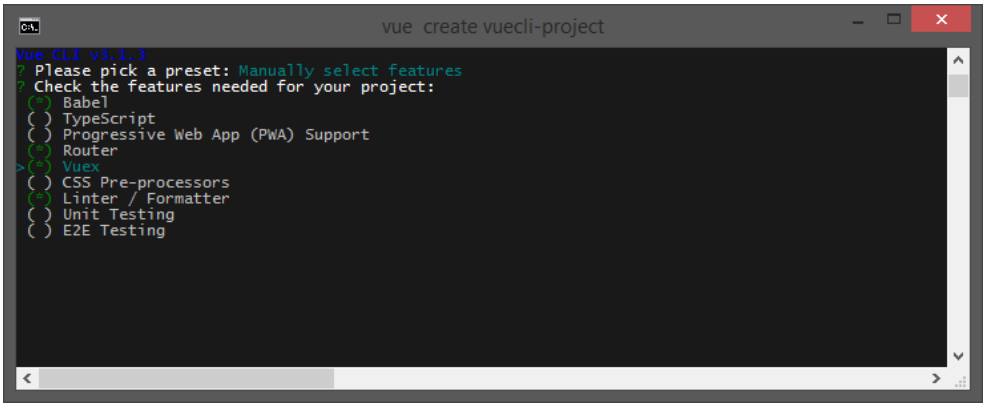

Рисунок 3.4 – Створення проекту в Vue CLI

Створення проекту відбувається за допомогою команди vue create. Інструмент задає структуру папок, яка виглядає наступним чином:

- папка node_modules містить пакети, потрібні для інструментів додатків та розробки;
- public папка містить статичні ресурси проекту, які не будуть включені в процес комплектування;
- папка src містить додаток Vue.js з усіма ресурсами;
- .gitignore містить список файлів і папок, виключених з елемента керування версією Git;
- babel.config.js містить налаштування конфігурації для компілятора Babel;
- package.json містить список пакетів, необхідних для розробки Vue.js, а також команди, що використовуються для інструментів розробки;
- package-lock.json містить повний список пакетів, необхідних для проекту та залежних файлів;
- README.md містить загальну інформацію про проект.

У папці src, показаній вище, ми маємо такі файли та папки:

- assets папка, що використовується для статичних ресурсів, які вимагає додаток, і які будуть включені в процес комплектування;
- components папка використовується для компонентів програми;
- views папка використовується для компонентів, які будуть відображатися за допомогою функції URL-маршрутизації;

- App.vue є кореневим компонентом;
- main.js - це файл JavaScript, який створює поточні об'єкти Vue;
- router.js використовується для налаштування маршрутизатора Vue;
- store.js використовується для налаштування бази даних, створених за допомогою Vuex.

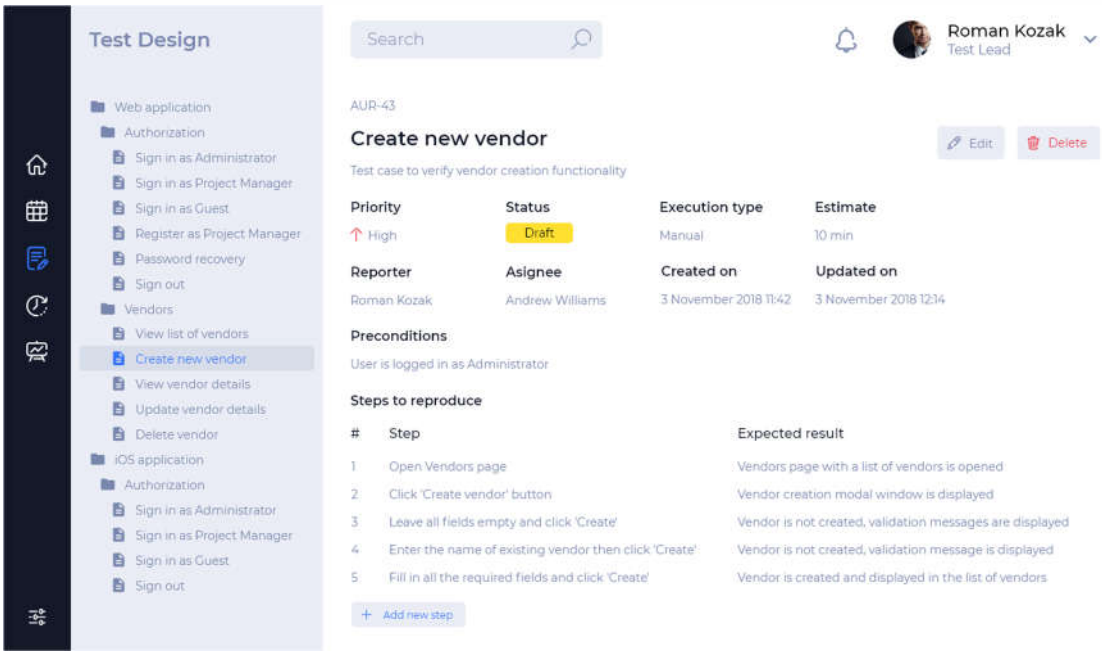

Рисунок 3.5 – Загальний макет сторінки

Для підтримки статичної типізації, повноцінних класів та можливості підключення модулів було використано мову програмування Typescript. Це дозволить пришвидшити процес розробки, спростити рефакторинг та повторне використання коду.

TypeScript є зворотньо сумісним з JavaScript і компілюється в останній. Фактично, після компіляції програму на TypeScript можна виконувати в будь-якому сучасному браузері або використовувати спільно з серверної платформою Node.js. Код експериментального компілятора, який транслює TypeScript в JavaScript, поширюється під ліцензією Apache. Його розробка ведеться в публічному репозиторії через сервіс GitHub [40].

TypeScript по своїй суті це розширення мови ECMAScript 5. Розробниками були додані наступні опції:

- анотації типів;
- виведення типів;
- класи;
- інтерфейси;
- enumeration типи;
- mixin;
- узагальнене програмування (шаблони);
- модулі;
- скорочений синтаксис «стрілок» для анонімних функцій;
- розширені можливості пошуку і параметри за замовчуванням;
- кортежі.

Для спрощення роботи зі стилями було обрано мову LESS. Вона забезпечує наступні розширення CSS: змінні, вкладені блоки, міксини, оператори і функції [41].

Щоб зібрати усі модулі разом необхідно використати збірник модулів (бандлер) Webpack. Webpack приймає бібліотеки та генерує графік залежностей, що дозволяє веб-розробникам використовувати модульний підхід для розробки веб-додатків. Його можна запустити як з командного рядка, так і налаштувати за допомогою конфігураційного файлу webpack.config.js. Цей файл використовується для визначення правил, плагінів для проекту.

Webpack також пропонує вбудований сервер розробки, який називається Webpack Dev Server, і може використовуватися як сервер HTTP для обслуговування файлів на етапі розробки. Додатково він забезпечує можливість використання гарячої заміни модулів (hot module replacement) [42].

Масштабованість – одна з фундаментальних властивостей інформаційної системи. На рівні архітектури програмного забезпечення, масштабованість – це здатність підтримувати великі кількості архітектурних компонентів та з’єднань між ними [43].

Масштабованість у розробці програмного забезпечення тісно переплітається з поняттям «розширюваність». Розширюваність вказує на можливість розширення функціональності системи в разі необхідності [44].

З самого початку система проектувалася таким чином, щоб її можна було масштабувати та розширювати. Виходячи з особливостей архітектури фреймворка Vue проектовану систему можна розширювати додатковими компонентами та модулями без падіння продуктивності. Ієрархія та структура компонентів спроектована в такий спосіб, щоб додавання нових було максимально простим і швидким. Всі однотипні модулі реалізовувалися за подібним принципом та поміщалися в одну папку. Як приклад, для моделей і форм існують загальні класи, які повинні перевикористовуватися при створенні нових сутностей. Це гарантує уніфікацію коду в межах розробленої системи.

3.4 Тестування розробленого програмного забезпечення

Метою тестування даного програмного забезпечення є виявлення та усунення помилок, допущених на етапі розробки, а також запобігти появі нових дефектів [45]. Тестування необхідно проводити, тому що всі люди роблять помилки. Також помилки можуть виникати і через збої в роботі апаратного забезпечення або через зовнішнє середовище.

Цілями тестування даної системи є:

- знаходження дефектів;
- запобігання появі нових дефектів;
- отримання даних про рівень якості продукту;
- отримання інформації про відповідність продукту очікуванням кінцевих користувачів.

Тестування проводиться в ручному та автоматизованому режимах, як з доступом до коду та документації, так і без нього.

Під час проведення тестування потрібно виконувати верифікацію та валідацію.

Верифікація (англ. verification) – це процес оцінки системи або її компонентів з метою визначення чи задовольняють результати поточного етапу розробки умовам, сформованим на початку цього етапу (чи виконуються цілі, терміни, завдання, по розробці проекту, визначені на початку поточної фази) [46].

Валідація (англ. validation) – це визначення відповідності програмного забезпечення очікуванням і потребам користувача, вимогам до системи.

Щоб покрити всі можливі варіанти виникнення помилок, тестування необхідно проводити на кількох рівнях, зокрема на рівні компонентів та на рівні системи. Окремим, але не менш важливим, є інтеграційне тестування, адже розроблена система буде інтегруватися зі стороннім програмним забезпеченням.

Компонентні та інтеграційні тести повинні бути автоматизованими та покривати як мінімум 80% функціоналу продукту.

Системне тестування проводиться в ручному режимі за допомогою спеціальних інструментів, таких як Postman та Fiddler. Приклад тестування за в Postman зображений на рисунку 3.6. Ефективне тестування можливе за умови продумування всіх комбінацій вхідних даних. Для цього можна застосувати техніки, що базуються на структурі та специфікації, такі як умови, рішення, твердження, аналіз граничних значень, розбиття на класи еквівалентності.

Для забезпечення максимальної якості продукту треба проводити функціональне тестування наряду з нефункціональним [47]. З нефункціонального тестування варто звернути увагу на продуктивність, надійність та підтримуваність системи.

Тестування є одним з найважливіших етапів, оскільки воно може надавати об'єктивну, незалежну інформацію про якість програмного забезпечення, ризики відмови, як для користувачів так і для замовників [48].

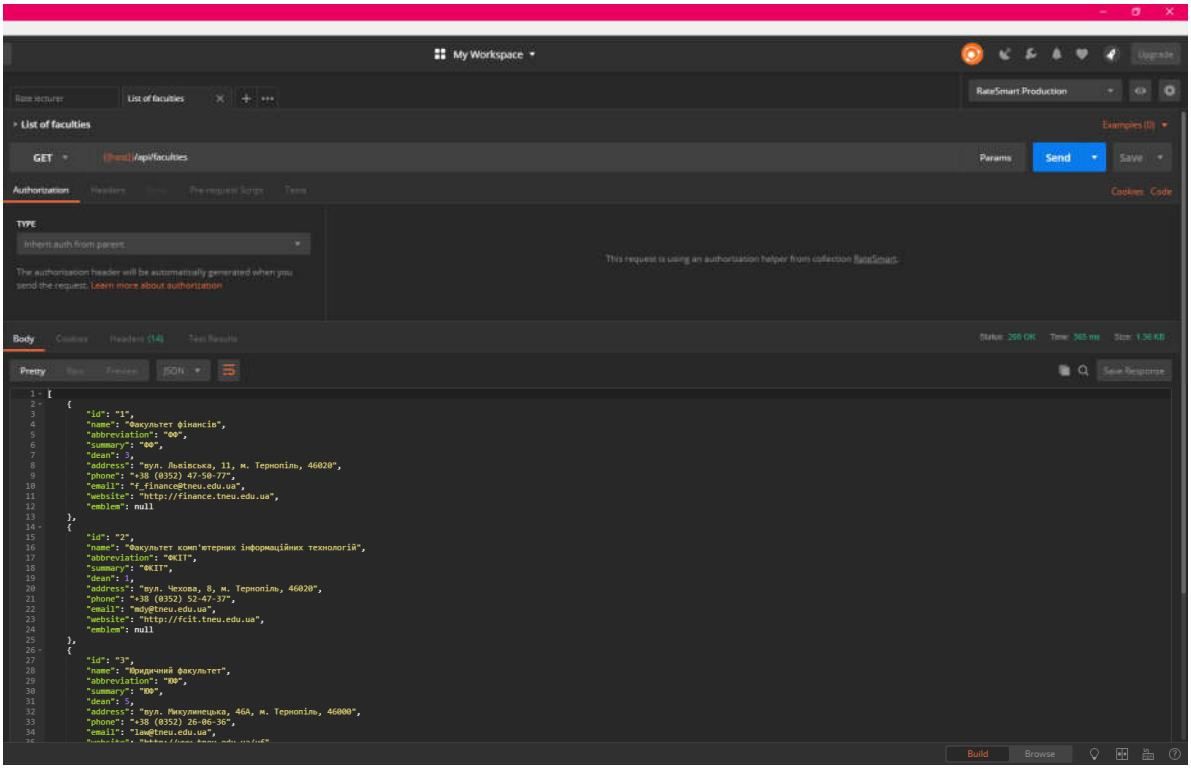

Рисунок 3.6 – Інтеграційне тестування за допомогою Postman

Розділ присвячено тестуванню розроблених алгоритмів.

ВИСНОВКИ

На основі аналізу сучасних методів проектування та розробки програмного забезпечення можна зробити наступні висновки.

1. Проведено аналіз методів та найкращих практик управління та моніторингу процесів контролю якості, а також існуючих програмних рішень, що дозволило виділити переваги та недоліки .

2. Розроблено алгоритм управління процесом контролю якості програмного забезпечення, що дозволило ефективно контролювати стан якості програмного продукту під час розробки.

3. Спроектовано архітектуру програмної системи та інтерфейсу для інтеграції між сервером і клієнтом, що дозволило провести попереднє дослідження запропонованого рішення.

4. Реалізовано серверний модуль для системи управління процесом контролю якості програмного забезпечення.

5. Реалізовано клієнтську частину для системи управління процесом контролю якості програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Березький О. М. Методичні рекомендації до виконання магістерської роботи з освітнього ступеня «Магістр». Спеціальність: 123 – Комп'ютерна інженерія. Магістерська програма – Комп'ютерна інженерія / О. М. Березький, Л. О. Дубчак, Г. М. Мельник / Під ред. О. М. Березького. Тернопіль: ТНЕУ, 2018. 41 с.

2. Методичні вказівки до оформлення курсових проектів, звітів про проходження практики, випускних кваліфікаційних робіт для студентів спеціальності «Комп'ютерна інженерія» / І. В. Гураль, Л. О. Дубчак / Під ред. О. М. Березького. Тернопіль: ТНЕУ, 2019. 33 с.

3. Козак Р. І. Управління процесами контролю якості програмного забезпечення / Науково-практична конференція молодих вчених і студентів «Інтелектуальні комп'ютерні системи та мережі», Тернопіль, 15 квітня 2019 р., с. 51-52.

4. Козак Р. І. Забезпечення та контроль якості програмного забезпечення / ІІ науково-практична конференція молодих вчених і студентів «Інтелектуальні комп'ютерні системи та мережі», Тернопіль, 14 листопада 2019 р., с. 73-75.

5. Сексенбаєв К., Султанова Б. К., Кісіна М. К. Інформаційні технології в розвитку сучасного інформаційного суспільства // Молодий вчений. - 2015. - №24. - С. 191-194.

6. Елгебелі А. Проблемы качества программного обеспечения и практические рекомендации [Електронний ресурс] / Айя Елгебелі. – 2013. – Режим доступу до ресурсу: https://www.ibm.com/developerworks/ru/library/r-software-quality-challenges-practice-recommendations/index.html.

7. Stebbing L. Quality Assurance: The Route to Efficiency and Competitiveness / Louise Stebbing., 1993. 300 с. (3).

8. Larman C. Agile and Iterative Development: A Manager's Guide / Craig Larman., 2004. 27 с.

9. What is Agile Software Development? [Електронний ресурс] // Agile Alliance Режим доступу до ресурсу: https://www.agilealliance.org/agile101/.

10. Кріспін Л. Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд - Agile Testing: A Practical Guide for Testers and Agile Teams / Л. Кріспін, Г. Джанет. Москва: Вильямс, 2010. 464 с.

11. Канер К. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений / К. Канер, Д. Фолк, Е. Нгуен. Київ: ДіаСофт, 2001. 544 с.

12. Куліков С. Тестирование программного обеспечения. Базовый курс / Святослав Куліков. – Київ: EPAM Systems, 2018. 298 с.

13. ISTQB Certified Tester Foundation Level Syllabus. // International Software Testing Qualifications Board. 2018. С. 96.

14. Standard Glossary of Terms used in Software Testing // International Software Testing Qualifications Board. 2018. С. 38.

15. Дідковська М. В. Тестування: Основні визначення, аксіоми та принципи / М. В. Дідковська, Ю. О. Тимошенко. Київ: Кафедра математичних методів системного аналізу КПІ, 2010. 62 с.

16. Kenneth R. Project Quality Management: Why, what and how / Rose Kenneth. Fort Lauderdale: J. Ross Publishing, 2005. 192 с.

17. Roseke B. The Quality Control Process [Електронний ресурс] / Bernie Roseke // ProjectEngineer. 2016. Режим доступу до ресурсу: https://www.projectengineer.net/the-quality-control-process/.

18. Quality Assurance, Quality Control, and Quality Assessment Measures [Електронний ресурс] // US Environmental Protection Agency Режим доступу до ресурсу: https://archive.epa.gov/water/archive/web/html/132.html.

19. How to Establish Quality Control Processes [Електронний ресурс] // Score.org. 2019. Режим доступу до ресурсу: https://www.score.org/blog/how-establish-quality-control-processes.

20. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. 3. MIT Press, 2009. 1312 с.

21. Albert Endres, Dieter Rombach. A Handbook of Software and Systems Engineering. Addison Wesley, 2003.

22. Микола Глибовець. Основи комп'ютерних алгоритмів. — Видавничий дім «Києво-Могилянська Академія», 2003. 452 с.

23. Буч Г., Рамбо Дж., Джекобсон А. Язык UML. Руководство пользователя. Пер. с англ. М.: ДМК, 2000. 432 с.

24. Крег Ларман. - Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3-е вид. М.: Вільямс, 2006. 736 с.

25. Cornett S. Code Coverage Analysis [Електронний ресурс] / Steve Cornett // Bullseye Testing Technology. 2014. Режим доступу до ресурсу: https://www.bullseye.com/coverage.html.

26. Binder R. Testing object-oriented systems / Robert Binder., 2000.

27. Lewis W. Software Testing and Continuous Quality Improvement / William E. Lewis., 2016. 688 с. (3).

28. Браун І. Веб-разработка с применением Node и Express. Полноценное использование стека JavaScript / І. Браун. Санкт-Петербург: Пітер, 2017. 336 с.

29. Бенкер К. MongoDB в действии / К. Бенкер., 2014. 394 с.

30. Васвані В. MySQL: использование и администрирование / В. Васвані. Санкт-Петербург: Пітер, 2011. 368 с.

31. Richardson L. RESTful Web APIs / L. Richardson, M. Amudsen, S. Ruby., 2013.

32. Berners-Lee T. A Short History of Resource in web architecture [Електронний ресурс] / Tim Berners-Lee // W3. 2018. Режим доступу до ресурсу: https://www.w3.org/DesignIssues/TermResource.html.

33. Брукс Ф. Міфічний людино-місяць, або як створюються програмні системи / Ф. Брукс. Санкт-Петербург: Символ-Плюс, 1999. 304 с.

34. Мірошниченко Е. А. Технології програмування: навчальний посібник / Е. А. Мірошниченко. Томськ: Видавництво Томського політехнічного університету, 2008. 128 с.

35. Express фреймворк для веб-застосунків, побудованих на Node.js [Електронний ресурс] // Node.js Foundation. 2018. Режим доступу до ресурсу: http://expressjs.com/uk.

36. Masse M. REST API Design Rulebook / M. Masse., 2011. 26 с.

37. Kyriakidis A. The Majesty of Vue.js / A. Kyriakidis, K. Maniatis. Бірмінгем: Packt Publishing Ltd, 2016.

38. Filipova O. Learning Vue.js 2 / Olga Filipova., 2016.

39. Hamilton N. The A-Z of Programming Languages: JavaScript / Naomi Hamilton., 2008.

40. Fenton S. Pro TypeScript: Application-Scale JavaScript Development / Steve Fenton., 2013.

41. LESS [Електронний ресурс] Режим доступу до ресурсу: http://lesscss.org/.

42. Gulati S. Webpack: The Missing Tutorial [Електронний ресурс] / Shekhar Gulati. 2016. Режим доступу до ресурсу: https://github.com/shekhargulati/52-technologies-in-2016/blob/master/36-webpack/README.md.

43. Kaner C. Lessons Learned in Software Testing: A Context-Driven Approach / C. Kaner, J. Bach, B. Pettichord., 2001.

44. Lönnberg J. Visual testing of software / Jan Lönnberg. Гельсінкі: Helsinki University of Technology.

45. Канер К. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений / К. Канер, Дж. Фолк, К. Нгуен. Київ: ДіаСофт, 2001. 544 с.

46. Бейзер Б. Тестирование чёрного ящика. Технологии функционального тестирования программного обеспечения и систем / Б. Бейзер. Санкт-Петербург: Пітер, 2004. 320 с.

47. Bach J. Risk and Requirements-Based testing / James Bach. // IEEE Computer Society. 1999.

48. Kaner C. Testing Computer Software / C. Kaner, J. Falk, Q. Nguyen. New York: John Wiley and Sons, 1999. 48 c.

49. Binder R. Testing object-oriented systems / Robert Binder., 2000.

50. Lewis W. Software Testing and Continuous Quality Improvement / William E. Lewis., 2016. 688 c. (3).