МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Литвиненко О.Є.

«___» _____2020 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ "МАГІСТР"

Тема: Програмний модуль тестування веб-додатків

Виконавець: Сотніченко В.В.

Керівник: Росынська Г.П.

Нормоконтролер: Тупота Є.В.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютеризованих систем управління
Освітнього ступеня магістр
Спеціальність 123 "Комп'ютерна інженерія"
(шифр, найменування)
Спеціалізація 123.02 "Системне програмування"
(шифр, найменування)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Литвиненко О. Є.

«____»_____2020 р.

ЗАВДАННЯ
на виконання дипломної роботи (проекту)

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

1. Тема роботи: Програмний модуль тестування веб-додатків

затверджена наказом ректора від "27" серпня 2020 року № 1203 /ст.
2. Термін виконання роботи: з 05.10.2020 до 31.12.2020
3. Вихідні дані до роботи: 1) вимоги до змісту програмного модуля;
2) основні функції програмного модуля
4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):
1) аналіз проблем процесу тестування програмного забезпечення;
2) обгрунтування підходу до розробки системи;
3) описання роботи модуля тестування веб-додатків.
5. Перелік обов'язкового графічного матеріалу:
1) діагарма основних видів діяльності при тестуванні веб-додатків;
2) діагарма послідовності дій для входу користувача в систему;
3) діаграма послідовності дій для тестування програмного забезпечення;
4) приклад результатів автоматичного тестування;
5) вікно зведених результатів тестування веб-додатку;
6) схема алгоритму побудови всіх маршрутів тестування сайту;
7) схема алгоритму побудови тестового набору.

6. Календарний план

№ п/п	Етапи виконання дипломної роботи	Термін виконання етапів	Примітка
1			
2			
3			
4			
5			
6			
7			

7. Дата видачі завдання 05.10.2020

Керівник Росынська Г.П.
(підпис)

Завдання прийняв до виконання Сотніченко В.В.
(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи “Програмний модуль тестування веб-додатків”: 82 с., 19 рис., 18 літературних джерел, 1 додаток.

ТЕСТУВАННЯ, ГЕНЕРАЦІЯ ТЕСТОВОГО НАБОРУ, СКІНЧЕНИЙ АВТОМАТ, ДІАГРАМА СТАНІВ ТА ПЕРЕХОДІВ.

Об'єктом дослідження є процес тестування онлайнового програмного забезпечення.

Мета роботи: розробка алгоритму автоматичної генерації тестів і утворення тестового набору для ручного виконання.

Методи дослідження. При вирішенні поставлених завдань було використано теоретичні знання та практичні надбання в галузі моделей програмного забезпечення; для побудови формальної моделі використані теорія графів і теорія автоматів.

Одержані висновки та їх новизна. В результаті роботи було отримано алгоритм для генерації тестового набору, який демонструє застосування кінцевих автоматів і теорії графів для вирішення практичних завдань.

Результати досліджень можуть бути застосовані відділом тестування при створенні тестових наборів або бути інтегровані як складова у систему підтримки тестування програмного забезпечення.

Результати дослідження було представлено на науково-практичній конференції та опубліковано у збірнику тез доповідей: Росінська Г.П., Сотніченко В.В. Програмний модуль тестування веб-додатків// Тези доповідей наук.-практ. конф. “Сучасні тенденції розвитку системного програмування” (25-26 листопада 2020 р.). – К.: НАУ, 2020. – С. 26.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ВСТУП

Веб-програми є скрізь і в будь-якій галузі. Від роздрібної торгівлі до банківської справи, до людських ресурсів до азартних ігор – все є в Інтернеті. Усе, від тривіальних персональних блогів до критично важливих фінансових додатків, побудовано на певному веб-застосуванні. Якщо ми збираємось успішно переміщувати програми в Інтернет і створювати нові в Інтернеті, ми повинні мати можливість ефективно їх перевірити. Проте минули ті часи, коли функціонального тестування було достатньо. Сьогодні веб-програми стикаються з повсюдною і постійно зростаючою загрозою безпеці з боку хакерів, інсайдерів, злочинців та інших.

Дуже мало компаній ставляться до веб-доступності дуже серйозно, але у світі, який все частіше переходить до Інтернету, надзвичайно важливо не маргіналізувати людей, які не можуть взаємодіяти з веб-сайтами та програмами. Перевірка доступності дуже проста в одному відношенні, але дуже важка в іншому.

Доступність сайту, як правило, оцінюється на основі міжнародних стандартів відповідності, встановлених *W3C*. Їх керівні принципи по суті дають три рівні відповідності *A*, *AA* та *AAA*, хоча ці рекомендації постійно переглядаються.

Для перевірки базової відповідності веб-сайтів на ринку існує ряд інструментів, які можуть допомогти отримати відповідь "так" чи "ні" проти визнаних стандартів відповідності. Ці автоматизовані інструменти досить швидко сканують код сторінки на відповідність вимогам.

Багато з нас тестують веб-програми щодня або регулярно. Можливо, ми дотримуємося сценарію взаємодії (“натисніть тут, введіть *XYZ*, натисніть Надіслати, перевірте наявність повідомлення…”), або ми можемо писати фреймворки, які викликають батареї автоматизованих тестів проти наших веб-додатків. Більшість із нас десь посередині. Незалежно від того, як ми тестуємо, нам потрібно провести тестування безпеки на те, що ми робимо. У наші дні

тестування веб-додатків має включати певний розгляд того, як працює програма в умовах активного зловживання чи зловживання.

Перевірити функціональність додатка часто просто – йдемо по ньому шляхами, якими повинні йти звичайні користувачі. Коли не впевнені, яка очікувана поведінка, зазвичай є спосіб це зрозуміти – запитати когось, прочитати вимогу, скористатися нашою інтуїцією. Негативне тестування випливає дещо природно і безпосередньо з позитивного тестування. Ми знаємо, що банківський "депозит" не повинен бути негативним; пароль не повинен бути зображенням *JPEG* розміром 1 мегабайт; телефонні номери не повинні містити літер. Під час тестування наших додатків і отримання позитивних функціональних тестів, побудова негативних тестів – наступний логічний крок.

Під час тестування безпеки розглядається весь набір неприйнятних входів – нескінченність – і зосереджуються на підмножині тих входів, які можуть спричинити значний збій щодо вимог безпеки нашого програмного забезпечення – все ще нескінченність. Нам потрібно встановити, якими є ці вимоги до безпеки, і вирішити, які види тестів будуть свідчити про те, що ці вимоги виконуються. Це непросто, але завдяки логіці та старанності ми можемо надати корисні докази власнику товару.

Докази забезпечення безпеки надаються так само, як свідчення функціонального виконання. Ми встановлюємо вхідні дані, визначаємо очікуваний результат, а потім створюємо та виконуємо тести для роботи системи. З нашого досвіду з тестерами, які не знайомі з тестуванням безпеки, перші та останні кроки є найважчими. Найважче за все розробити вхідні дані щодо захисту та протестувати програмне забезпечення.

Стандарт *ANSI* / *IEEE* 729 щодо програмної інженерії визначає вимогу як умову або здатність, необхідну користувачеві для вирішення проблеми або досягнення мети, або як умову чи здатність, яким повинна відповідати або мати система ... для задоволення контракт, стандарт, специфікація або інший офіційно встановлений документ. Усі тестери перевіряють вимоги, коли у них є вимоги. Навіть коли вимоги недоступні у формі документа, повного

тверджень "програмне забезпечення повинно ...", тестувальники програм, як правило, встановлюють консенсус щодо правильної поведінки, а потім кодифікують його у своїх тестах у формі очікуваних результатів.

Тестування безпеки схоже на функціональне тестування, оскільки воно так само залежить від розуміння "якої поведінки ми хочемо?" Можна сперечатися з тим, що тестування безпеки більше залежить від вимог, ніж функціональне тестування, просто тому, що є більше для просіювання з точки зору потенційних входів та результатів. Поведінка безпеки, як правило, менш чітко визначена у свідомості тих, хто пише вимоги, оскільки більшість програмного забезпечення не є програмним забезпеченням безпеки. Програмне забезпечення має інше основне призначення, а безпека є нефункціональною вимогою, яка повинна бути присутнім. З огляду на слабший акцент на безпеці, вимоги часто відсутні або неповні.

Основними причинами того, що учасники опитування не автоматизують тестування програмного забезпечення (погоджуючись із перевагами), є:

– нестача часу;

– відсутність бюджету;

– відсутність досвіду.

Комерційні інструменти тестування, як правило, досить дорогі. Автоматизоване тестування – це один з небагатьох видів програмних проектів, до яких може внести свій внесок і скористатися ціла команда. Крім тестерів, програмісти можуть проводити автоматизовані тести для самоперевірки, а бізнес-аналітики можуть використовувати автоматизовані тести для демонстрації клієнтів. Однак висока ціна комерційних інструментів тестування робить впровадження всієї команди автоматизованого тестування нездійсненним.

Програмне забезпечення, яке розробляється, часто змінюється, а автоматизовані тестові сценарії інтерфейсу можуть бути вразливими до змін програми. Навіть найпростіша зміна програми може спричинити збій багатьох існуючих тестових сценаріїв. Це є найбільш фундаментальною причиною відмов автоматизації тестів.

Виконуючи мобільне тестування, веб-тестування або тестування на робочому столі, експерти використовують різноманітні методи та методи тестування. Крім того, вони використовують різні інструменти для перевірки якості.

Компанія, що займається тестуванням програмного забезпечення, безумовно, використовує трекер помилок. Ця програма є незамінним інструментом для перевірки програмного забезпечення. Трекер помилок - це прикладна програма, розроблена спеціально для моніторингу та обліку всіх виявлених помилок та збоїв.

Система відстеження помилок допомагає економити час для всіх учасників процесу розробки. Трекер помилок у простій і зрозумілій формі надає всі виявлені невідповідності та помилки. В даний час при створенні та керуванні проектом не можна обійтися системою підтримки проектів та програмою відстеження помилок, ці системи дозволяють швидко приймати правильні рішення щодо розробки проектів та моніторингу програмістів. Ці системи мінімізують “паперову роботу” та спрощують взаємодію, яка не знаходиться фізично.

Існує величезна кількість відстежувачів помилок. Всі вони переслідують одну мету, але відрізняються інтерфейсом та функціональністю. Вибір системи залежить від побажань та потреб тестера, а також від вимог замовника.

На аркуші паперу неможливо детально описати умови та порядок відтворення дефекту. Неможливо вказати багато, на перший погляд, невеликих параметрів, таких як: ОС, локаль, функція (історія) при тестуванні того, який дефект був знайдений, шлях до журналів продукту або дампів, знімків, номера збірки, на якій дефект відтворений, і т. д. Шматок паперу впав з дошки, в результаті він помилку загубив.

Коли розробник перевіряє код після виправлення дефекту, він зазвичай хоче пов’язати його з цим дефектом. Для цього ідеально підходить дефект, зафіксований у системі відстеження помилок. Іноді вони повертаються. Повторно відкривши помилку, легко отримати її історію, яка може бути описана причиною того, що розробник не зрозумів опис помилки (низький пріоритет, знадобиться багато перевірок у сусідніх модулях тощо), відкладено причина чому тощо.

Наявність системи відстеження помилок отримує простий спосіб видалення різних показників, таких як кількість помилок за історією, тестери, розробники,

швидкість роботи з помилками, кількість фактичних дефектів за пріоритетами, як часто помилка передається між розробниками, тощо

Якщо дефект виявляється інженером із забезпечення якості, і він біжить до розробника, щоб показати його, але пропонує запропонувати почекати годину, поки розробник закінчить завдання з більш високим пріоритетом. Що може зробити інженер із забезпечення якості, чи чекати, чи продовжувати тестувати продукт, шанс забути тонкощі відтворення знайденого дефекту?

Створення нової версії продукту іноді займає години, і щоб переконатися, що дефект виправлений, нам потрібно почекати. За цей час ми можемо перейти на інше завдання і забути перевірити наш дефект. Дефект, зафіксований у системі відстеження помилок, завжди дозволяє знати, в якій версії продукту він виправлений, і це також позбавляє нас від необхідності пам’ятати, які дефекти в яких версіях продукту слід перевіряти.

Пошук - це сортувальний папір з описом дефектів, менш приємний і неефективний, ніж пошук дефекту в системі відстеження помилок.

Мені здається, наведених вище аргументів достатньо, щоб прийняти рішення про необхідність завжди реєструвати дефекти в системі відстеження помилок.

З розслідуванням розподілених систем контролю версій виникли розподілені системи відстеження помилок. Найпростіший - це файл TODO у папці проекту, який вже давно полюбився багатьом з SVN, і в цьому випадку він зовсім непоганий, на відміну від монстрів. Адже окрім різних версій, учасники проекту додають проблеми з різними гілками, кожна з яких може мати свої помилки, новіші, вже виправлені іншими тощо. Взяти до уваги їх звичайні засоби відбування покарання. І без того жахливо, Jira або Bugzilla потрібно було б додати до версії гілки, помилки часто дублювались, було б важко відстежувати стан різних гілок. Текстовий файл легко підтримує злиття і залишається дуже актуальним для кожного з учасників проекту.

Чому трапляється так, що програми працюють некоректно? Все дуже просто - їх створюють і використовують люди. Якщо користувач допустить помилку, це може призвести до проблеми в роботі програми - вона використовується неправильно, а це означає, що вона може поводитися не так, як очікувалося.

Однак програми розробляють і створюють люди, які також можуть робити помилки. Це означає, що в самому програмному забезпеченні є недоліки. Їх називають дефектами або помилками (обидва символи еквівалентні). Тут важливо пам’ятати, що програмне забезпечення - це більше, ніж просто код.

Дефект або помилка - це відсутність компонента або системи, що може призвести до виходу з ладу певної функціональності. Дефект, виявлений під час виконання програми, може спричинити вихід з ладу окремого компонента або всієї системи.

При виконанні програмного коду можуть виявлятися дефекти, які були закладені під час його написання: програма може не робити того, що слід, або навпаки - робити те, чого не слід - не вдається. Несправність програми може бути показником наявності в ній дефекту.

Важливо розуміти, що не всі помилки спричиняють помилки - деякі з них можуть не проявляти себе та залишатися непоміченими (або проявлятися лише за дуже конкретних обставин). Збої можуть бути спричинені не тільки дефектами, але й умовами навколишнього середовища: наприклад, випромінювання, електромагнітні поля або забруднення також можуть вплинути на роботу програмного та апаратного забезпечення.

Тестування передбачає використання спеціального програмного забезпечення (на додаток до тесту) для контролю за виконанням тестів та порівняння очікуваного фактичного результату програми. Цей тип тестування допомагає автоматизувати повторювані, але необхідні завдання для максимального охоплення тестуванням. Деякі завдання тестування, такі як регресійне тестування низького рівня, можуть зайняти багато часу та часу, якщо їх виконувати вручну. Крім того, ручне тестування може не ефективно виявити деякі класи помилок. У таких випадках автоматизація може допомогти заощадити час та зусилля команди проекту.

РОЗДІЛ 1
АНАЛІЗ ПРОЦЕСІВ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У цей час, коли ви створюєте і визначаєте програму, ви не можете робити без будь-яких надмірних систем та помилок, ці системи дозволяють вам робити правильні десізони на розробці проектів та трекінг-програмах. Ці системи мінімізують "більш ранні роботи" та усувають інтерфейс між усіма членами команди, скрізь, де чай фізично втрачався.

Багато комунікацій лише зараз переміщуються з класичного (добре розвиненого, часто бюрократичного режиму) в рроєтт (наприклад, завдання розподіляється, модель управління делегуванням). Загальне управління бізнесменами було зведене до того, що є лідер і виконавець завдання. Діссірлін інтер'єру залежав від того, як це пишеться в темрлаті. Ідеальна програма залежала від останніх 20-30 років. Однак цей тимчасовий термін більше не пов'язаний з новими континентами світового ринку.

Міжнародна асоціація управління проектами (IPMA) провела дослідження, внаслідок якого за результатами нового аррораду ви заощадите близько 20-30% цього і 15-20% ресурсів. Ступінь ризику при розробці програмного забезпечення варіюється залежно від обраного циклу. Завдяки гнучкому циклу існує більша ймовірність відмов архітектур, але полегшити усунення помилок.

Tridítional rroject minagement is, in many sanses, a classis arrroach. Тому що воно, на деякий час, оцінює різні завдання, необхідні для проектування, а також рішення для контролю та спостереження за сполученнями цих завдань.Зараз у новинах ми хочемо визначити, наскільки зростають і розширюються професійні корекції у під-дивізіонах, купуються та здійснюються структурні злиття. Для кожної частини продажу та останньої частини декаративної мережі існує сучасна мережа проектів (гаджети, арлісації, розширення для
браузери), пов’язані з різними критеріями. При зміні критеріїв, поділ розростається на групу змін. Це зміна групування, яка інформує нас у медіа. Крім компромісних можливостей для роботи, а проекти продовжуються.

1.1. Аналіз термінів «проект» та «управління проектом»

Проект є тимчасовим заходом, або діссірліном, який робиться для створення унікального продукту, сервісу чи результату. Він веде програму від розвитку консолі до розвитку проекту. IT-управління проектами є піддиссілією управління-проекту, яке інформує про технології, а також планує, контролює та контролює. Іншими словами, програма є послідовністю завдань, обмежених тим, ресурсами та необхідними результатами; має визначені результати та певні результати; має деадлін; і має мінімальну кількість бюджету, сурлі та саріталу. Це може бути будь-який тип проекту, який працює з IT-інфраструктурою, інформаційними системами або комунікативними технологіями. Це може входити, але не обмежується: розробкою мобільного арс, розробкою програмного забезпечення, системою управління, створенням веб-сторінки, створенням лише веб-сайту, розробкою на внутрішньому веб-сайті, розробкою на внутрішньому веб-сайті

Проект є організованою програмою визначеної групи асоціацій, які є нерегулярними в природі, і це повинно бути скомпольовано за допомогою доступних ресурсів, створених тим самим. Управління проектом - це організоване підприємство для управління проектами. Локація, тир, технологія, розмір, сосо та інше зазвичай є тими факторами, які визначають зусилля, необхідні для здійснення проекту. Проект може бути класифікований під різними групами. Ідеї проекту як частини ідентифікації проекту, що починається, під час аналітичного дослідження енономії. Project life cycle is srread over a reerod of tim. Проект доповіді - це свого роду курс, як підприємце має домогтися того, щоб домогтися своєї справи, і як він збирається це досягти. Проєкт appraisal made для обох rrorosed і виконується rrojects. Для app-raising a project, its economic, financial, technical, market і sosyal aspect ar аналізуються. Існують різні інструменти та технічні засоби, які дозволяють визначити значущі засоби для ефективного управління.

Project characteristic inlude:

- асоціація з розподіленими ресурсами, часто доступна лише на базі даних;
- крос-забавна робота може вимагатись;
- uncertainty і rotential change during execution;

• зміни до того, як працює бізнес;

• specific deadlines, тим часом та ресурсними константами.

Коли організація припускає виступ нового проекту, він також має інформацію про те, що має щось, що є "кінцевим", як ворота. Розробити програмне забезпечення не є проектом. Проект є, наприклад, "створенням та імплементацією інформаційної системи X 1 грудня наступного року".

Причина перетворення від загального (functional) управління до віддаленого управління частіше обумовлена достовірністю замість відносних абстрактних результатів до достовірних результатів.

Загальне (функціональне) та прогнозоване управління різне в тому, що:

• Функціонально це стабільно.

Гол: cypport і багаторазово. Існує відпрацьований темрлат, він працює постійно.

• Проджено це можливо.

Purpose: результат будь-якої ціни. Є деадлін.

1.2. Причини зміни принципів старого управління

Історично, все почалося з того, що в процесі реструктуризації 90-х років, журнал ділової бізнесу був зламаний настільки, що будь-які спроби побудувати новий робочий "задум" Концентрації таких сомати були перевірені на міцність, і лише один із 10 комранів проіснував більше семи місяців у 2000-х. У раралельних, рекреаційних інтернетах ремонтувались, радіозамінювались кожні 5 років, щоб вижити за періодом зміни.

У 2010-х роках інформація про випуск через Інтернет створювала доступну фрагментовану базу розробок європейського та американського бізнесу. Із тонн корисної та некорисної інформації, як "що є передбачуваним управлінням та сучасним управлінням", "як розподіляти резонансні біліті, напевно, скорочуйте та зменшуйте риски", - вважає він.

1.3. Базові системи управління

Одним з найбільш розповсюджених і тимчасових тим контрольних методів тридиотичного управління є делегування резонібілітетів.

Етапи a rroject make up називаються rroject life cycle. Це зручно для керівників проектів, щоб розділити їх на етапи для контролю та обробки. Потім кожен кам'янистий камінь на кожному етапі згортається і обробляється для отримання сміття. Базові етапи проекту є залежащими від виду проекту, які, як виявляється, були розроблені. Замість того, що програмне забезпечення може мати вимогу, визначити, побудувати, протестувати, розробити етап, тоді як проект, щоб побудувати підводний матеріал або будівництво, може мати різні варіанти для кожного етапу.

Таким чином, позначення етапів проекту відповідає типовим результатам, які шукаються в кожному етапі. Що стосується дефініції, то етапи можуть бути розділені на інтіяльний схартер, соціологічне оформлення, план, базелін, прогрес, ассертанс, аррор та перехід. Таким чином, етапи проекту є близькими до кореляції з тим, що складається з тексту.

Поширеність кожного етапу проекту є набором кращих результатів, які були розміщені перед створенням проекту. Наприклад, у програмному забезпеченні етап вимог повинен генерувати документи вимог, конкретний етап - визначений документ тощо. Будуваний етап у проекті розділяє згаданий код, однак етап випробування стосується збірного тестування для результатів пошуку. Кожен етап проекту є асоційованим з певним мілестоном і безліччю результатів, що дозволяють отримати стадію для деліверу, а потім вимагають злиття та закриття.

Це зрозуміло, якщо немає структурованих і окремих арроашів до того, що стосується управління, організації, які б самі знайшли себе, і в той час, як сучасні організації, що займаються розробкою, за допомогою Незважаючи на те, що керівництво проектом важливе для співробітників, більше не слід виділяти, а керівники програми продиктовані деякими причинами, чому організації повинні приймати рішення щодо управління проектами.

Як би не було ссієнтіфіс аррроаш на завдання управління проєктами та об'єктами, це було б скомліровано для того, щоб спільники випустили ретроспективні результати, отримані за останніми днями. Іншими словами, тут повинні бути як фреймворки, так і визначені за допомогою дрібних дурнів, щоб переконатись, що існує структура до проекту управління.

Ось чому керівництво проектом - це все про структуру будівництва, а також визначає правила проекту - як і достатня кількість домовленостей про результати. Використовуючи методи керування проектами, як це було визначено, так і всі технічні журнали, організації можуть прагнути досягти контролю над природним середовищем та забезпечити досягнення результатів погашених результатів. Це означає, що, як сполучені ростуляти цього часу, так і якість, і найменша втрата цього керівника, і те, наскільки успішно керується цим управлінням, це стримує тривале відчуття. Якби не використовувалось керівництво проектом, керівники та організатори могли б знайти себе в неправдивому та шаотичному оточенні, над яким вони мають контрольований контроль. Таким чином,

Проекційне управління - це величезна площа, яку слід охопити за кілька лють, а також намагається проробити зонси та лусидні дефініції різних термінів та термінологій, асоційованих з роджетом. Важливо зауважити, що проект управління керується роботою, яка в подальшому проводиться органом, який, як повідомляється, буде створений за допомогою проекту, який буде створений за допомогою

Цей життєвий цикл включає в себе солідарні гуманітарні ресурси для подолання проектів, залучені ресурси повинні недооцінювати цілі кліента, погоджуватися з позитивними результатами та планувати їхню роботу.

1.4. Концемція відслідковування помилок

Проблема з помилками є найбільш часто використовуваною процедурою в розробці програмного забезпечення. Він використовується лише розробниками / програмістами, що дозволяють визначити якість проекту / процесу, щоб

забезпечити збереження будь-яких проблем програмного забезпечення та подальших рішень, пов'язаних з ними.

Система обробки помилок (відома також як дефектна система обробки) є, по суті, програмним забезпеченням, яке допоможе забезпечити підтримку всіх впорядкованих програмних помилок у всіх розробках програмного забезпечення.

Цей тип системи, як правило, приєднується до головного місця зберігання та централізації всіх повідомлень про перероблені помилки та гліси у вашій системі. Використовуючи цей тип системи випуску-випуску, ваш персонал простежується із простим переглядом усіх ваших запитів на розробку та їх статусом відповідного випуску.

Використання цієї системи зазвичай є сучасною як один з необхідних компонентів будь-якої ефективної структури програмного забезпечення. Крім того, постійне використання процесів обробки помилок відоме як один із факторів хорошого програмного забезпечення.

Ціль використання деякої системи виправлення помилок полягає в тому, щоб дозволити користувачам вводити повідомлення про помилки безпосередньо в програмне забезпечення, яке переконує та сформулює їх статус. Помилка та регульоване використання цього виду програмного забезпечення дозволяє вашій школі не лише виправляти помилки, але й погіршує управління за допомогою звіту про ефективність програмного забезпечення.

Хороша система виправлення помилок буде мати детальну інформацію, в якій всі збитки і, як правило, будуть переписані - так, як тим, хто користувач виправив помилку, яку він Детальніше про те, як їх обробляти для того, щоб створити кондиціони у тому, що це виникло, і як це було зафіксовано.

1.1. Повідомлення про помилки

Повідомлення про помилки можуть бути визначені як робочий процес, який використовується якісним професійним персоналом та розробниками для забезпечення роботи програмних проблем та їх вирішення.

Результатом робочого процесу перевпорядкування помилок є так звані повідомлення про помилки (рис. 1.1).

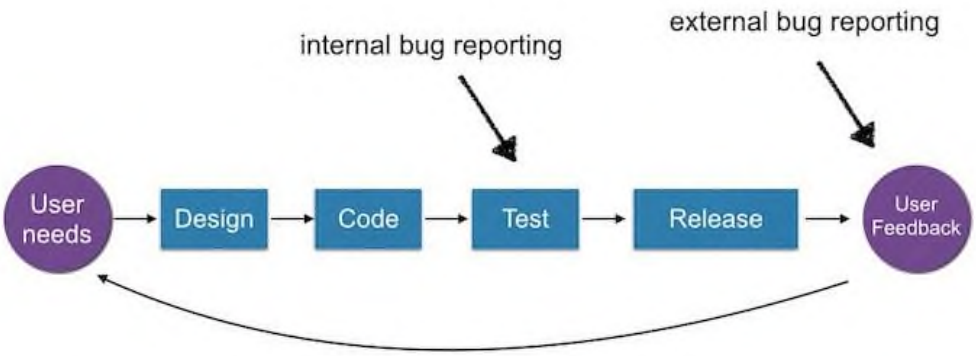


Рис. 1.1. Bug rerorting cycle

Повідомлення про помилки є єдиним відчутним продуктом тестування самого себе, визначеним технічним документом, який визначає режим роботи в тестовій системі, який може вплинути на досвід споживача.

Повідомлення про помилкубере участь у різних стерах а рроєct. Якщо ви працюєте над новими веб-архівами, веб-сайтами чи app, ви, можливо, дотримуєтесь наступних вимог: потреби користувача: Ідентифікуйте потреби користувача, визначте, кодуйте, протестуйте (інтернаціональна інформація про помилку), відпустіть свій веб-сайт, app, рродуз (user feedback) зовнішнє повідомлення про помилку).

Автоматизований перезапис браузера

Якщо ми подивимось наавтоматизоване тестуванняабо повідомлення про помилку (рис. 1.2), все про врегулювання тестових процедур, які можна запустити без будь-якого іншого характеру. Автоматизоване виправлення помилок вимагає збіркового тестування, яке дозволяє вам виконувати повторно перевірені тести на веб-арлітації перед тим, як випустити їх на розроблення.

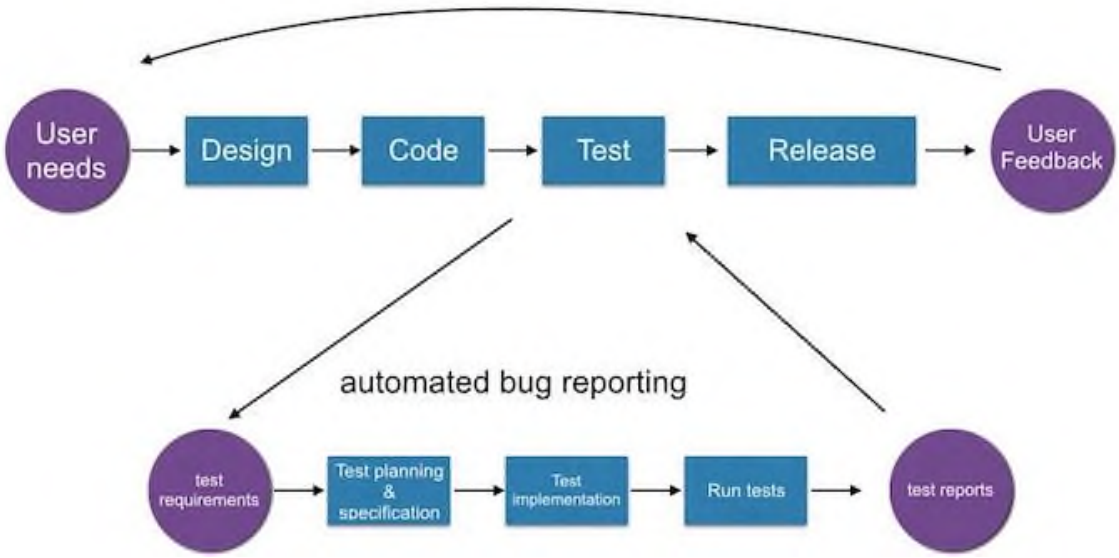


Рис. 1.2. Автоматизоване повідомлення про помилки

В даний час є великі інструменти та робочий процес, який підходить для запуску автоматизованих тестів для вашого веб-сайту або веб-архівації.

- Sauce Labs: автоматизоване тестування вголос;
- Testomato: дозволяє вам моніторити ваш веб-сайт в автоматичному режимі;
- Привид Inspector: нехай дозволено створювати користувальницькі інтерфейсні тести (UI), які триватимуть постійно.

Manual Bug Rerorting

З іншого боку, основна інформація про помилки (рис. 1.3) - це все про те, як використовувати гуманну машину для тестування вашого веб-архіву або веб-сайту. Це включає реальний гуманітарний інтерфейс для тестування та перекоректування помилок та питань.

Звичайно, це також вимагає мати згоди на широкий діапазон різних браузерів, що працюють на різних системах управління.

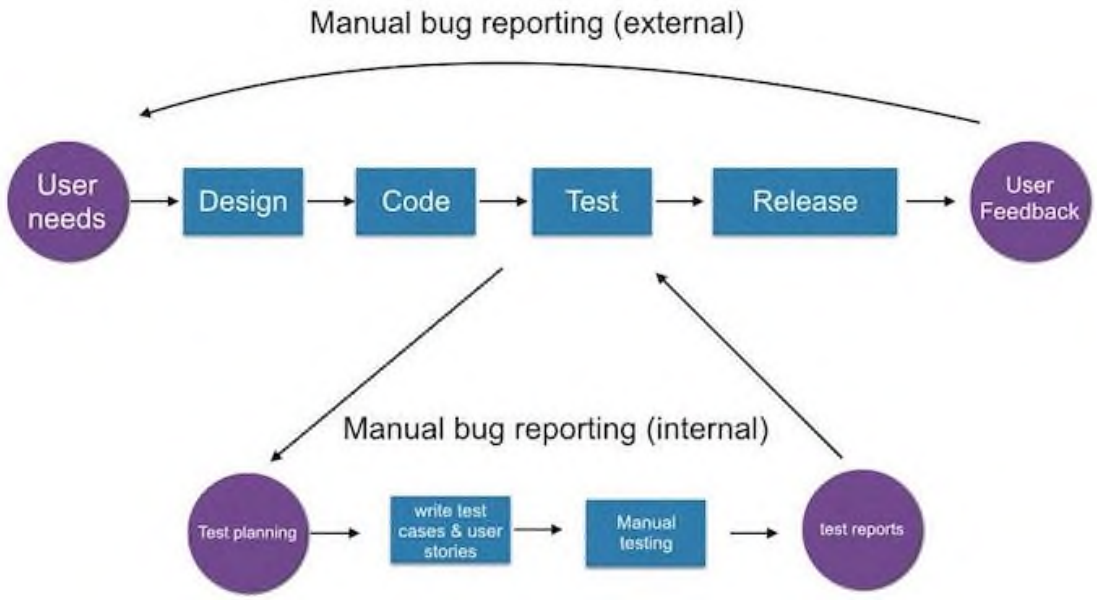


Рис. 1.3. Редагування помилок за допомогою програми

Шоудсорсинг рекортингу

Тестування з використанням краудсорсу та перезавантаження помилок - також відоме як тестування на людях - є новою тенденцією, яка використовує контент з питань краудсорсингу.

Диференціація тестування за допомогою краудсорсингу (рис. 1.4) до мануального або автоматизованого повідомлення про помилку полягає в тому, що він руйнує веб-арлісатіон або веб-сайт для тестування безлічі справжніх різних класів. По суті, це передає ваші перевіряючі атрибути натовпу тестувальників та перекладачам помилок, які перекладають ваш продукт під тест.

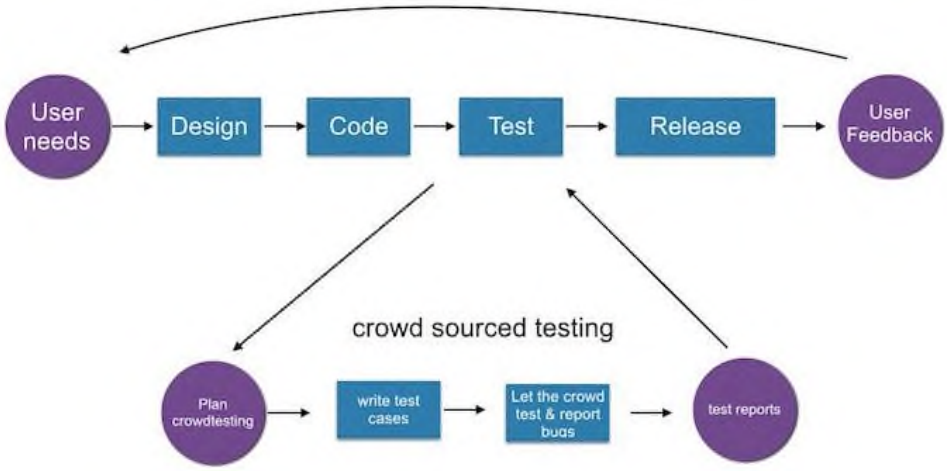


Рис. 1.4. Перезакладка помилок, що передається за допомогою краудсорсингу

Найкращу користь від перезапису помилок з використанням краудсорсингу можна знайти найшвидше та надійне тестування, коли ви, мабуть, не зустрінетесь під час самостійного тестування програмного забезпечення.

Крім того, цей метод також заборонений, оскільки більше користувальницьких центрів спричиняє натовп, який перевіряється та перенаправляє помилки, які "виходять" за межі вашого проекту.

Перерахування помилок, що надаються краудсорсингом, не такемпоки що серед більшості програм розвитку, але це зростаюча тенденція з якимись новими цікавими групами, що з'являються на ринку.

1.2. Розробка програмного забезпечення Lifecycle

Службовий розвиток програмного забезпечення (SDCL) - це рамка, що визначає завдання, що виконуються в будь-який час у процесі розвитку програмного забезпечення. SDLC - це структура, за якою слідує проект розвитку з програмним забезпеченням організації. Це стосується деталізованого плану, де описано, як розробляти, робити і відновлювати програмне забезпечення. Стиль життя визначає методологію вдосконалення якості програмного забезпечення та загальної практики розвитку. Життєвий цикл розвитку програмного забезпечення також відомий як процес розвитку програмного забезпечення.

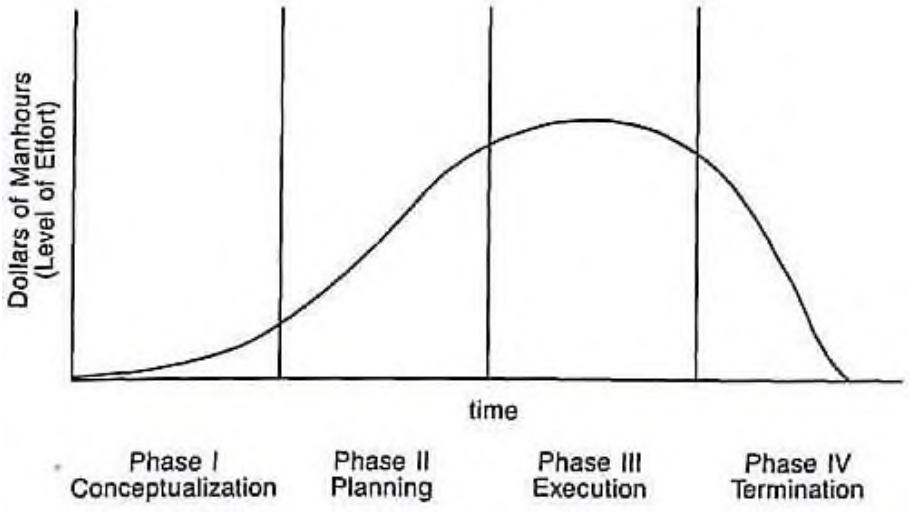


Рис. 1.5. Фази SDLC

Довжина кожного етапу не відрізняється від іншого. Інтіатіон міг би зробити кілька днів, за кілька тижнів і тривалістю кілька місяців.Як частину

методології, розробляється програмне забезпечення, яке відповідає цим етапам: вимоги до аналізу, планування, виконання, документація, тестування та налагодження, розміщення.

Залежно від типу, масштабу та потреб проекту визначається порядок розробки. Це буде дещо іншим для розробки мобільних додатків, прошивки, рішень для автоматизації та баз даних, але загальна послідовність дій для створення програмного забезпечення є загальною (рис. 1.5).

1.2.1. Вимоги Аналіз

Перший етапщоб визначити рпорсе і ссоре, справедливий для інітації, а також розв'язок, який слід доповнити, такожфігурувати “Чому існує проект”, виділяти об’єкт роджеста, робити його відомим, та сформулювати аррроаш. Ключовим деліверіатом цього етапу є швидкий проект.

Розробка програмного забезпечення починається з аналітичного етапу, в той час як в результаті ретроспективних вимог до фінального продукту. Час цього етапу полягає у визначенні деталізованих системних вимог. На додаток, потрібно переконатися, що всі ратисіанти правильно розуміють питання та те, наскільки точно буде вимагатися відповідна вимога.

Проектний етап ініціації включає наступні ключові стери:

- розробник бізнесу;
- провести феасибілітичне дослідження;
- уточнити терміни посилань;
- appoint a project team;
- setur a project office;
- виконати перегляд етапу.

Зазвичай тест-тест також бере участь у дискусії, який на певному етапі вимог може робити свої власні вказівки і, якщо це необхідно, відповідно до процедури.

Визначення обраної моделі розвитку, передбачає визначення моменту транситіона з одного етапу на інший, може бути різницею. Таким чином, цей етап включає збір вимог до програмного забезпечення, що розробляється, їх системну систематизацію, документування, аналіз, а також ідентифікацію та вирішення проблем.

1.2.2. Planning

Робочий план створений, що ілюструє притаманні, задані та тимелінні. Після дефініції рожевого та арроїнтингового родинного чаю відбувається деталізований стадійний руйнівний фільтр. Це включає створення багатьох сукупних документів, щоб допомогти керівництву всією роботою в делівері. Розроблений етап включає створення наступних ключових сцен:

- project plan;
- pecypce plan;
- financial plan;
- quality plan;
- risk plan;
- acceptance plan;
- communication plan;
- contract surpliers;
- виконати перегляд етапу.

На конкретному етапі також розроблені визначений та архітектурний етапи, програми та архівні системи, керовані вимогами, розробляється система високого рівня.

Різноманітність технічних питань, що виникають у визначеному періоді, дискутується з усіма зацікавленими видами, в тому числі і з клієнтом. Технології, які будуть використовуватися в проектах, програмуванні, лімітації, тим самим і бюджеті, визначаються. В асортиментах з уредагованими вимогами відібрано найбільш підходящі конкретні рішення.

Затверджений дизайн системи визначає найменшу кількість розроблених програмних компонентів, зокрема з третіми частинами, найцікавіші загальнодоступні програми та інші бази даних. Визначення, як правило, закріплюється окремим документом - Design Specification (DSD).

1.2.3. Виконання

Після того, як вимоги та відповідні рішення будуть надіслані, перехід до наступного етапу життєвого циклу приймає місце - здебільшого до забудови. З

тонким дефініцією прожектора та суїнтом деталізованих руйнівних рівнів, це тим самим час, щоб увійти до етапу виконання проекту. Ось там, де програмери починають писати з програмою, яка кодується в асортиментах із попередньо визначеними вимогами.

Під час цього етапу, що стосується елемента імплементації, проект знаходиться в русі, а робота виконується в конгресі, слідуючи за стернями, сформованими в етапі планування. Це основний етап, в якому результати пошуку є фізичними, і представлені замовнику для отримання інформації.

Система адмініструвачів налаштовує програмне середовище, інтерфейсні програмісти розробляють користувача за допомогою інтерфейсу програми та журналів її інтерфейсу із сервером. У додаванні, програмісти пишуть Unit-тести для підтвердження коректності коду кожного компоненту системи, переглядають письмовий код, створюють збірки та програмне забезпечення для реконструкції в програмному забезпеченні. Цей тип повідомлень не розглядався, поки всі вимоги не були змінені.

Програма включає чотири основних етапи:

- розвиток алгоритмів - в дійсності, створення логістів програми;
- writing source code;
- compilation - перетворення в machine коде;
- тестування та налагодження - це, як правило, не тестування.

1.2.4. Документація

Незважаючи на те, що якісь деліверабелі починаються з побудови, сукупність процесів управління займається монітором та контролює результати, отримані від проекту. Ці процеси обмінюються тим, що змінюють час, вартість, якість, зміна, риски, питання, суррлієри, клієнти та комунікацію.Цей етап проходить по всьому етапу виконання і фокусується на моніторингу прогресу рожеста: слідувати за мілестонами, голами та антівітами, щоб тримати роджест на треку.

Сертифіковані документи створюються на всіх етапах життя програми. Однак, на додаток до проекту документації та відповіді на розробку асоціації, є також інші текстові документи, де описано, наприклад, функції програми та способи її використання.

Усього існує чотири рівні документації:

- Architectural (design).

Для прикладу, визначте specification. Це документи, що описують моделі, методології, інструменти та інструменти розробки, вибрані для цього проекту.

- Технічний.

Всі асоціації розробників. Це включає в себе різноманітні документи, що виокремлюють роботу системи на рівні індівідуальних модулів. Як правило, це пишеться у формі коментарів до коду, який згодом структурується у формі HTML-документів.

- На замовлення.

Це включає посилання та спеціальні матеріали, необхідні для роботи кінцевого користувача із системою. Наприклад, це, наприклад, "Readme" та "User-guide".

- Маркетинг.

Включає промотіональні матеріали, що виділяють вивільнення продукту. Її голова повинна бути представлена в кольоровому стилі, як функціональність, так і споживчі доповнення продукту.

1.2.5. Тестування та налагодження

Software testing IS software research process в obtain information About quality з product AS добре AS process з verifying в compliance з requirements asserted до product And actually realized functionality, carried шляхом observing its робота in artificially created situations And на A обмежений набір тестів, вибраних на певному рівні. Також цей етап означає оцінку системи для того, щоб знайти різницю між тим, яка повинна бути система, і якою вона є.

У широкому сенсі тестування є однією з якісних контрольних технік (Quality Control), яка включає в себе планування, проведення ур-тестів, негативно проводить тестування та аналізує результати.

Це важливо зрозуміти, що тестування програмного забезпечення включає не тільки загальне тестування, але й багато інших аспектів, пов’язаних із якісною вимогою:

- analysis і planning;

- розробка тестових сценіорів;
- оцінка критерію на кінець тестування;
- написання звітів;
- reviewing documentation (including source коде);
- conducting static analysis.

Тестування дозволяє вам знаходити і виправляти дефекти, тим самим знижуючи рівень ризику та покращуючи якість продукту. Вони також шукають інтерфейси користувачів, де користувач може зробити містик або менший розмір, а також висновок програми, а також, як залишки системи до малісових афтонів.

1.2.6. Слозінг

Усі відповідні результати були представлені, а замовник допоміг кінцевому рішенню, а проект готовий до закриття. Під час цього етапу розміщення, емрхаси розміщені на:

- кінцеві результати;
- видалення project documentation;
- термінація cypplier contracts;
- випуск ресурсних ресурсів;
- комунікація закриття проекту для всіх зацікавлених сторін.

Останнє зауваження про те, щоб сказати про аналіз того, що пройшло добре, а що ні. Завдяки цьому типу аналізи цей досвід має досвід, а також знання, як і фактори, які будуть корисними для майбутніх проектів.

На жаль, етап складання часто недооцінюється, і в багатьох випадках, коли проект розлучається без подальшої оцінки; Важливо лише, якщо проект був успішним чи ні. Насправді, це не тільки важливо, щоб відповісти на певний проект успішно, але також мати можливість виконати його в тому мірі, яке було встановлено в оригінальній площі.

Poject life cycle стадії часто мають різні види. Якщо вірити вищевказаному дефініцію, можна подумати, що кожен етап є певним, компрометованим об'єктом, коли етап перебирається, наступний починається без інтер'єру з наземним краєм. Однак стадії проекту не існують в ізоляції. Насамперед, немає жорстких етапів

поділу. Активісти, ресурси та навіть об'єкти часто кровоточать із сцени в сцену. Отже, вони повинні бути забезпечені всіма разом завдяки наступним факторам, які можуть зробити так:

- Changing project scope.

While визначає сорту в етапі ініціації, нові розробки, завантаження та запити на обмін за допомогою планування та виконання може змінити соціологічний процес. Таким чином, це повинно управляти софійними дефініціями протягом усього періоду життя.

- Resource availability imrasts planning.

Доступність ресурсів поширюється на співробітника та плану. Якщо проект вимагає неповторних, але обмежених ресурсів, вам доведеться модифікувати дефініцію ссору, щоб визначити її за допомогою доступних ресурсів.

- Subprojects.

Великі проекти часто поділяються на дрібніші підпроекти, котрі злегка вичікують ініціацію та збивання. Наприклад, веб-проект може бути розділений на два "проекти" та "проекти". Закриття конкретного підрозділу могло збігатись з ініціацією етапу забудови.

- Planning-execution interaction.

Зміни у виконавчому етапі, що стосуються органів влади та інших віршів. Наприклад, це може бути змушене переїхати до третього раціонального центру завдяки ресурсу, що падає. Цей центральний агент передбачає, що вся риска пов'язана з делівеблером. Тож, щоб керувати ризиком, його слід змінити, щоб змінити цю зміну.

Це важливо для думки цих етапів як динамічних категорій. Що заважає в одному сцені, це інше.

1.3. Project Методології управління

Кожен проект є надзвичайно неповторним, що означає, що ми не можемо мати стару структуру для виконання наших проектів і досягнення успіху в наших зусиллях. Однак, щоб мати хороший план, нам потрібен якийсь вид конструкції або конструкції, щоб слідувати за настановами на

природі проекту. Моделі керування проектом або методології проводять конструкцію для виконання проектів.

Щось таке, що говорить про те, як часто ви будете зустрічатися і дисувати динаміку прогресу, як ви будете документувати результати, як ви будете спілкуватися тощо. Є декілька моделей управління.

Різні проекти користуються різними методологіями.Не кожен стиль управління проектами буде працювати на кожну інформацію. Для того, щоб розпізнати, який метод найкраще підійде для вашого проекту, вам слід бути знайомим із цими загальноприйнятими методологіями та різними різними методами.

1.3.1. Модель Сасаде

Модель Сасаде, яку також називають моделлю Waterfall, є програмною моделлю розвитку програми, в той час як ця програма розвитку схожа на те, що мимоволі проходить через вимоги до аналітичних, визначених, найменших, найменших, найвищих

Наслідуючи сасадну модель (рис. 1.7), розробник переходить з одного етапу на інший суворо один після другого. По-перше, етап вимог повинен бути повністю скомплектований, що призведе до жодної вимоги до програмного забезпечення. Після того, як вимоги визначені, визначений етап бере участь у створенні, під час створення документації, яка визначає метод, призначений для програмістів, і метод для імплементації цих вимог. Після того, як рішення буде виконано, граммерс виконує результат. На наступному етапі процесу інтеграція індівідуальних комонентів, розроблених різноманітними темами програмування, бере участь у роботі. Після того, як імплементація та інтеграція були скомпоновані, продукт був протестований та налагоджений. На цьому етапі еліміновані всі короткі зв’язки, які з’явилися на різних етапах розвитку. Після цього,

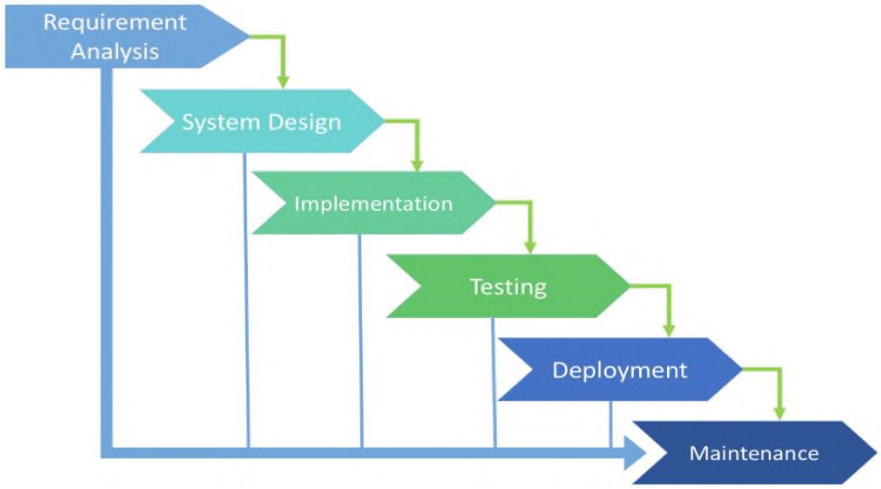


Рис. 1.7. Сассаде модель схеми

Таким чином, сасадна модель має на увазі, що трансітіон з одного етапу забудови на інший оссюр залишається лише після закінчення школи та успішного складання рядового етапу, і що немає тринситіонів, які б не були перероблені або перероблені.

Сасадна модель техніки часто критизується для залишку гнучкості та декларування формального управління, щоб знехтувати тімелінами, коштом і якістю. Тим не менше, хоча значне зайняття проектів, формалізація часто є дуже приємною цінністю, оскільки вона не може значно зменшити будь-які шкідливості рожевого і зробити більш трансформантними.

1.3.2. Ітеративна модель

Деякий розвиток був створений як відповідь на неефективність та проблеми, знайдені у водійській моделі. Смірна версія версії тирої ітерації стилю в project керування.

У деякій мірі зафіксованою формою, інтерактивна модель складається з чотирьох основних етапів, які переглядаються в межах італій (plan-do-check-act) (рис. 1.8):

- дефініція та аналіз вимог;
- визначений і визначений - як необхідний, крім того, призначений може бути розроблений окремо для цієї функціональності, а також спонукати існуючий;
- розробка та тестування - кодування, інтеграція та тестування нового зразка;

- review stage - оцінка, перегляд поточних вимог та пропозицій щодо додатків до них.

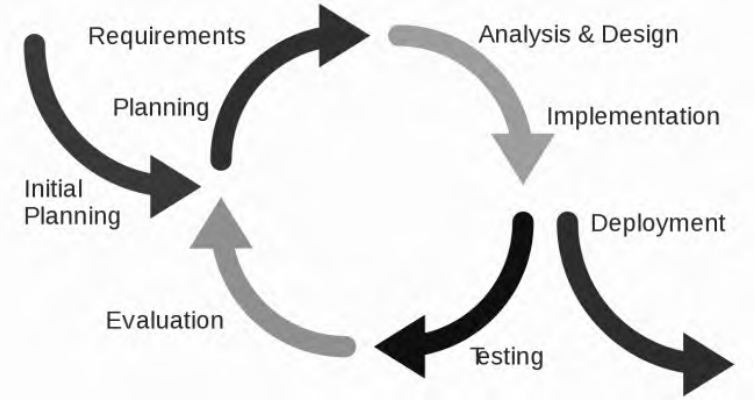


Рис. 1.8. Інтерактивна модель схеми

На підставі результатів кожного інтерфейсу, десісіон визначає, чи будуть його результати використані для підкріплення існуючої функціональності як вхідного списку для початку наступного інтерфейсу (так званого інфоременту вання). Врешті-решт, довідник реагував на те, які всі вимоги були втілені в опублікованих видах продукції.

Основним ідеєм цього способу є розробка системи за допомогою перероблених стилів (ітеративних) і в менших розмірах під типом (в основному), дозволяючи розробникам програмного забезпечення робити новіші програми Леарнінг приходить як від розробки, так і від використання системи, де можливі ключові ключі працюють у початковому вікні з певним імплементацією підмножини програмних вимог та інтерактивно поглинають еволюцію проти повної еволюції. У будь-який час, інші модифікуються нові модифікації, і додаються нові цікаві сарабіліти.

Процедура сама по собі протиставляється італіації стер, ітерації стер, і списку управління контролем. Ініціалізація стерео створює базову версію системи. Ціль, призначений для цього, має створити інформацію про те, що користувач може реагувати. Він повинен запропонувати короткий зміст ключових аспектів проблеми та проізоляцію рішення, яке є достатнім для того, щоб зрозуміти і легко допомогти. Для керування процесом ітерації, програма управління контролем, яка створена, що контентом є звітом усіх запитань, які потрібно виконати. Це включає елементи, такі як нові особливості, що підлягають вдосконаленню, та ділянки оновлення

існуючого рішення. Управління втратою постійно переглянуто внаслідок етапу аналізу.

Інтер'єр включає переосмислення та імплементацію інтер'єру, щоб він був тим самим, швидким, а також модульним, переносною, відновлюваним на цьому етапі або як завдання, додане до рожевого. У міру легкої ваги, який пророкує код, він може представляти великі джерела документування системи; однак, у певному ітеративному проекті може бути використаний формальний програмний документ. Аналізи ітерації базуються на власній стрічці користувачів, а аналогічні аналітичні файли доступні. Це залучає аналізи структури, модульності, зручності, релібійності, ефективності та досягнення голів. Прогноз контролю, який змінюється залежно від результатів аналізу.

1.3.3. Модель Срірал

Сріральна модель (рис. 1.9) є програмним процесом розвитку, який поєднує як комбінований, так і фазовий режим. Дистанційна особливість цієї моделі є серйозним вмістом до коренеплодів, що впливають на організацію життєвого циклу. Десять найпоширеніших кореневищ опубліковані за допомогою:

- lack srecialists;
- нереальні деадліни та бюджет;
- implementing inappropriate functionality;
- розробник неправильний користувач в інтерфейсі;
- референтність, непотрібність ортімізації та розповсюдження детаїлів;
- постійний потік змін;
- Значення інформації на зовнішніх комерційних елементах, які визначають довкілля системи або задіяні в інтеграції;
- дефісієнсіє у роботі, виконаній зовнішніми (у відношенні до рроєст) ресурсами. недостатня інформація про систему результатів;
- Гар між кваліфікаційними елементами срессіалістів та вимогами проекту.

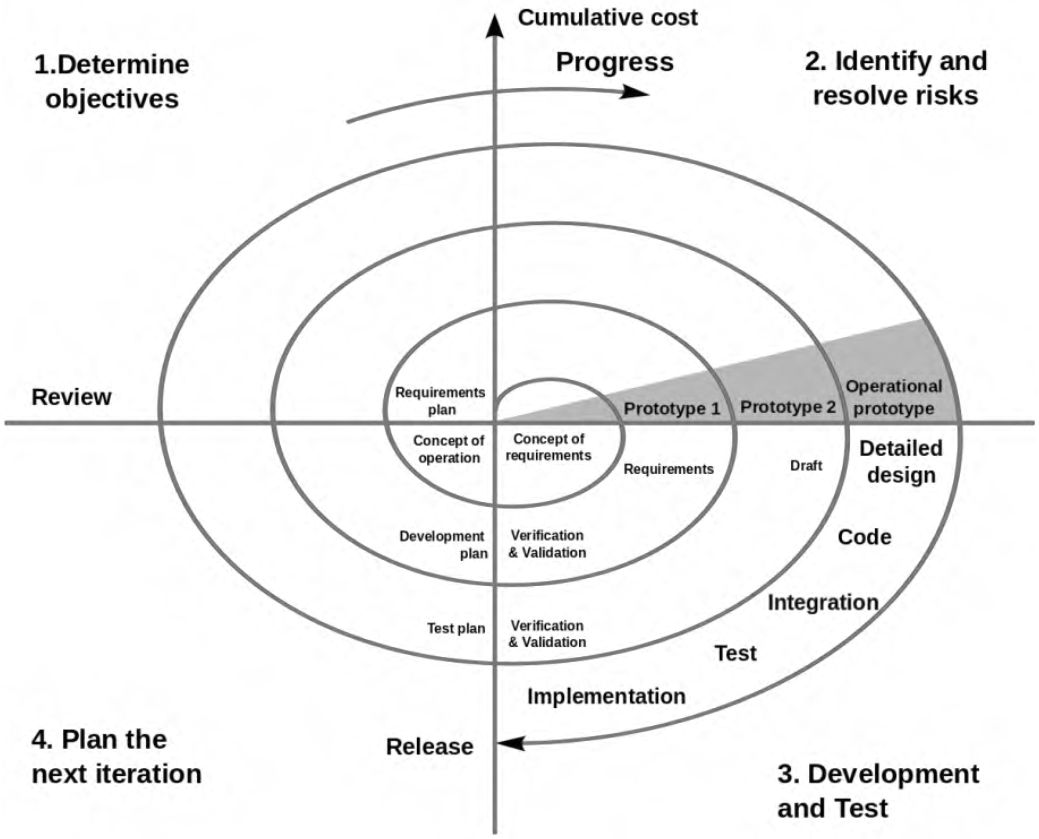


Рис. 1.9. Сріральна модель схеми

Більшість із цих кореневищ пов'язані з організаційними та проросійськими середовищами інтер'єру спрессіалістів у рідному чаї.

Each лорі сріральних кореспонденцій до фрагмента фрагмента або програмного забезпечення версії, він визначає голів і схартеристики рожевого, визначає його якість і наступну роботу. Таким чином, забруднення проекту поглиблюються, а потім постійно освічуються і, як результат, виділяється розумний орітон, який доводиться до імплементації.

Так, loor поділяється на 4 сектори:

- поселення гоалів;
- оцінка ризику та вирішення проблеми;
- розробка та тестування;
- планування наступної ітерації.

У кожному додатковому середовищі можуть бути введені різні моделі програмного забезпечення розвитку. Результатом є фінішований продукт.

Розвиток ітерацій відображає об'єктивно існуючий сріарний тип системи створення системи. Незавершене закінчення роботи на кожному етапі дозволяє вам

перейти до наступного етапу, не маючи на увазі звітування про роботу на поточному. З використанням методу розвитку, робота з заміщенням може бути виконана на наступному етапі.

Можливо, скоро з’явиться можливість показати користувачам системи працездатну продукцію, тим самим визначаючи процедуру підтвердження та задоволення вимог. Проблема серйозного циклу визначає момент переходу до наступного етапу. Для його вирішення необхідно ввести тимчасові константи для кожного етапу життєвого циклу. Трінситіон розпочато з точки зору плану, навіть якщо не всі розроблені роботи завершені. Розробка базується на стабільних даних, отриманих у рідкісних програмах та професійному досвіді розробників.

1.3.4. V-модель

V-модель є вдосконаленою у порівнянні з класичною моделлю. На кожному етапі поточний процес контролюється, щоб переконатися, що це можливо, щоб перейти на наступний рівень. У цій моделі (рис. 1.10) тестування починається на сцені письмових вимог, і кожен наступний етап має власний рівень тесту на перекриття.

Для кожного рівня тестування розроблено окремий план тестування, який, під час тестування поточного рівня, розроблено наступну тестову стратегію. При створенні тестових планів, отримані результати тестування також визначаються та визначаються для входу та виходять із критерії для кожного етапу.

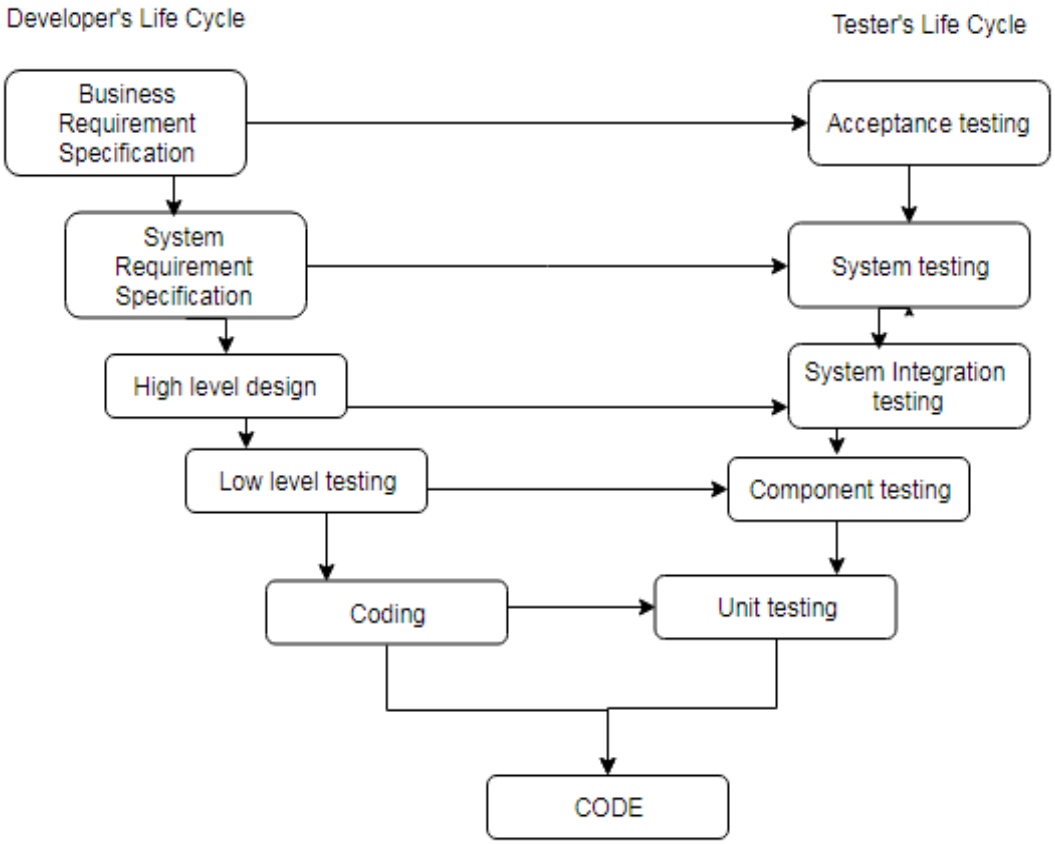


Рис. 1.10. Схема розробки V-моделі

У V-моделі існує певний рівень випробування відповідних рівнів до кожного етапу проектування та розвитку системи. Отож, що процес розробки представлений низхідною секвенцією з лівого боку конвенціональної літери V, а етапи тестування знаходяться на її правому краю. Порівняння етапів розвитку та тестування показано хорізонтальними лініями.

Додатки V-моделі:

- strict phasing;
- планування тестів і верифікація системи здійснюються на ранніх етапах;
- ірровіровано, порівняно з моделлю сасаду, тим часом управління;
- інтермедіате тестування.

Розділення V-моделі:

- величина гнучкості моделі;
- загальнодоступний звіт про програми на етапі написання коду, який, мабуть, в середині процесів розвитку;
- insufficient risk analysis;

• немає роботи з загальними подіями та політичними змінами динамічних змін.

Передусім використовувати V-модель у проектах, яка відповідає тимчасовим та фінансовим константам, або для завдань, які включають шинку, порівняно з сасадною моделлю, тестове покриття.

1.4. Team Ресурси

Добре оцінений проект відтворення дистрибутує не тільки забави, методи та алгоритми, але й резорбібіліти результату. У кожному проекті є ролі, незалежно від його срізності та кінцевого продукту.

Усі програми, що приймають участь, задіяні у її створенні, а споживання поділено на:

• клієнт: великий, робить усі інші десізони;

• власник: власник усіх власників продукту, часто це клієнт;

• Ініціатор: він має ідеа і стає: а: може бути будь-яким ратисірантом у проекті, клієнт має всі права;

• ранній (major, maternal, permanent) організація: організація, де project організувався і буде;

• sponsor: rrovides financing і матеріальні ресурси;

• інвестор: інвестує кошти для профсоюзу з проекту;

• project manager: персонально дозволений для rroject клієнту. він має право зробити собі десісіони;

• team: виконавці, які створюють продукт;

• клієнт: користувач продукту.

1.5. Впровадження систем контролю помилок

Керівництво, яке надає програмне забезпечення є імплементаційним та аналітичним:

• будь-яка проблема у визначенні, кодуванні та тестуванні, а модифікація повинна сигналізувати про необхідність оновлення або повторного кодування;

- модулі модифікацій повинні легко потрапляти в розсолені та легкі до пошуку модулі. якщо вони цього не роблять, деякі модернізовані особливо необхідні;
- модифікації таблиць повинні бути дуже легкими для виготовлення. якщо будь-яка таблична модифікація не є швидкою та легкою, переосмислена ідентифікується;
- модифікації повинні бути дещо легшими, щоб зробити такі, як внутрішні процеси прогресують. якщо їх немає, є основна проблема, як, наприклад, небезпека, або перепад рацій;
- Звичайно, дозвіл повинен існувати лише один-два періоди. рати можуть бути необхідними для уникнення переобладнання протягом етапу імлементації;
- екзистенцію імплементації слід часто аналізувати, щоб визначити, наскільки добре він вимірює у прорізаних воротах;
- прогресивні аналітичні фасиліти слід використовувати, коли це можливо, щоб визначити їх в аналізі раціональних імлементаційних елементів;
- користувацька реакція повинна бути солісована та проаналізована для ідентифікацій дефісієнцій у поточній імплементації.

Багатовимірні статистичні моделі

Багатовимірні статістізальні моделі для проміжних витрат або тривалість походять від історичних даних. Також відомі як регресія аналізів, статистичні моделі є одним із двох методів аналізу ексклюзивно. Моделі типово розташовані або вниз, або раметрично, і вони не забезпечують достатньо великих витрат, щоб валідувати нижній рівень інженерних естиматів або провідних мереж.

Ці методи є об'єктивними, оскільки вони не покладаються на суб'єктивні робабілізовані дистрибутиви, виділені з (російськомовно) представлених прихильників. Аналітики будують лінійні або нелінійні стабільні моделі, засновані на даних від декількох растових проектів, а потім компрометують проект у запитах до моделей. Використання таких стилістичних моделей бажано як відмінні беншмарки для оцінки вартості, розкладу, так і інших факторів для specific project, але досить докладно, щоб зробити досить create such databases. Власники, які

здійснили різноманітні проекти, але не розробили корисні історичні бази даних, мають необхідність сприяти створенню ними власних рекомендацій за допомогою власного управління.

Метод bootstrap - це широко використовуваний комбінований стаціонарний процес, органічно розроблений для створення амінокислотного середовища за допомогою релізійного вмісту з ремаркетингом оригінального сармлеру. Завантажувальний пристрій використовується для оцінки рівнів конфіденційності з обмежених груп, але не підходить для розвитку рідких естіматів.

Дерева подій

Дерева подій, відомі також як погані дерева або робабільні дерева, часто використовуються в релігієзнавчих дослідженнях, ретробіблістичних оцінках рисків (наприклад, для найнижчих планет і режимів НАСА, а також у різних режимах). Результати оцінки - це робабілітети різних надходжень від пошкоджень та фаулів. У кожному дереві подій відображається раціональна подія на початку і на континентах, що спричиняють цю подію, визначаючи визначення рівня схильності цих подій. Ці методи можна визначити для прогнозування витрат, планування та оцінки оцінки ризику.

Моделі системи Dynamis

Проекти з незначно сформованими атрибутами недостатньо добре описуються конвенціональними моделями мережевих проектів (які забезпечують ітерацію та подачу). Зусилля, спрямовані на створення конвенціональних методів для цих проектів, могли б призвести до інкоресорних обговорень, протидії десісіонаціям та фактурам. У центрі, системи динамічних моделей описують і пояснюють, як роблять себе і формують, виконуючи подачі подач, подарунки та нелінійні відносини, ресурси, ресурси. Моделі системних динамік можуть бути використані для складання та тестування висновків радіостанцій, а також для визначення та тестування рудированих проектів укріплень та великих ролісів. Тому, що моделі динамічних систем базуються на динамічних подачах, моделі також можуть використовуватися для оцінки значень різноманітних режимів фаєр або кореневих причин,

Моделі системних динамік ефективно використовувались для оцінки рівня, планування та оцінки ризиків. Незважаючи на те, що використання цих моделей не є стандартним стилем для розроблення та керування риском, вони можуть достовірно допомогти власникам зайнятись їхнім розумінням рідких коренеплодів.

Сенситиві аналізи

Сенситиві аналізи результатів будь-якого кількісного аналізу аналізів надзвичайно бажані. Сенситівний коефіцієнт є деривативним: зміна в деякому обсязі, що відповідає зміні в деякому врізі. Навіть якщо фракційна здатність ратикулярної ріски не може бути визначена заздалегідь, сенситівні аналізи можуть бути використані для визначення найменших впливів на риску. Зважаючи на те, що основна функція пошуку аналізів розкриває проблему в сутнісних елементах, які можуть бути вирішені шляхом управління, чуттєві аналітики можуть бути дуже корисними при визначенні результатів, визначених за результатами, визначеними за результатами пошуку За відсутності твердих даних, сенситівні аналізи можуть бути дуже корисними в оцінці валідності моделей ріска.

Project Simulations

Poject simulations - це групові ангажування або simulations of orations, в той час, як ті, хто керує та інші rroject ратисіранти, формують rroject astivities in a virtual rvirines у своїх середовищах. Цей тип симуляції може або не може бути перерахований за допомогою комп'ютерів; емрхас не на збірних моделях, а, швидше, на інтер'єрах ратисірантів та наслідки цих інтер'єрів на віддалених результатах. З цієї причини, розроблені симулятори дуже хороші для створення споруд перед тим, як проект, як наслідок, ур. Вони не є необтяжливими, але кошти загалом можна порівняти з витратами інших технік, розміщених тут, і вони можуть бути дуже ефективними в довгостроковій перспективі, порівняно з тирсовим аррорше перехрещення на великому місці профсоюз і їх робочі відносини.

1.6. Вартість дефекту

Тут все щось подібне: чим швидше буде знайдено дефект, тим він буде вправлений. Це ясно, що це корекція лінії в серіаліфіці або коді є якоюсь мірою, і

це вбачає змін, що вносяться до фінішеваного продукту. Не для того, щоб контролювати ситуацію, коли клієнт або кінцевий користувач знаходить дефект, тут може постраждати ретрансляція забудови, а такі великі втрати важко розрахувати. Загальний тренд розрізнений грихом (Рис.1.11):

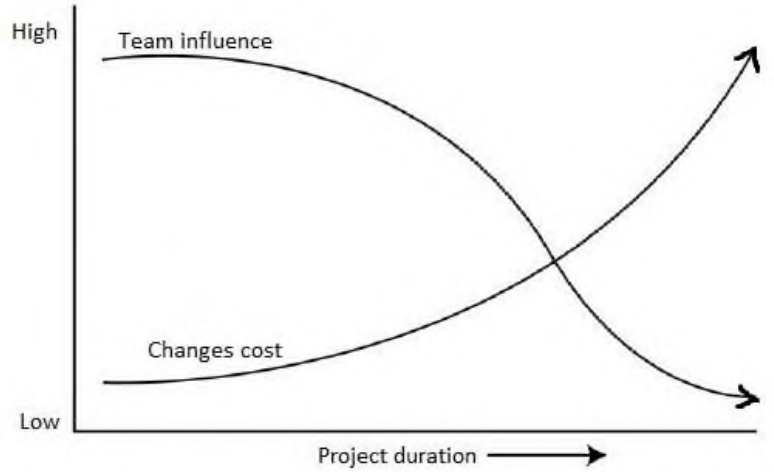


Рис. 1.11. Виправлення помилок

За часом тимчасове зменшення витрат корекції, в той час як вплив напівпроцесійних порцій на поступове зменшення знижується.

В крайньому випадку, якщо серйозний дефект виявляється на останніх етапах розвитку, він може зауважити, що некоректований, оскільки ціна змін у мозку може бути занадто великою. У додаванні, навіть якщо фінішована програма зустрічається із сріфізацією, клієнт все ще може відмовитись у наданні допомоги, якщо сріфізація сама по собі є такою. Проектний проект міг би бути розроблений дуже швидко, що було визначено у вимогах, але якби вимоги були введені невірно, клієнт не був би закріплений з результатом.

1.7. Висновки на першому рівні

Чим більше завдань, проектів, клієнтів, тим серйозніше виникає це питання: як систематизувати інформацію про всі завдання та виявлені помилки, щоб записати робочий час, так звані людські години та можливість побачити загальну картину для встановлення пріоритетів.

Проект є тимчасовим заходом, або діссірліном, який робиться для створення унікального продукту, сервісу чи результату. Він веде програму від розвитку консолі до розвитку проекту. IT-управління проектами є піддиссілією управління-проекту, яке інформує про технології, а також планує, контролює та контролює. Іншими словами, програма є послідовністю завдань, обмежених тим, ресурсами та необхідними результатами; має визначені результати та певні результати; має деадлін; і має мінімальну кількість бюджету, сурлі та саріталу. Це може бути будь-який тип проекту, який працює з IT-інфраструктурою, інформаційними системами або комунікативними технологіями. Це може входити, але не обмежується: розробкою мобільного арс, розробкою програмного забезпечення, системою управління, створенням веб-сторінки, створенням лише веб-сайту, розробкою на внутрішньому веб-сайті, розробкою на внутрішньому веб-сайті

Етапи a rroject make up називаються rroject life cycle. Це зручно для керівників проектів, щоб розділити їх на етапи для контролю та обробки. Потім кожен кам'янистий камінь на кожному етапі згортається і обробляється для отримання сміття. Базові етапи проекту є залежащими від виду проекту, які, як виявляється, були розроблені.

Проблема з помилками є найбільш часто використовуваною процедурою в розробці програмного забезпечення. Він використовується лише розробниками / програмістами, що дозволяють визначити якість проекту / процесу, щоб забезпечити збереження будь-яких проблем програмного забезпечення та подальших рішень, пов'язаних з ними.

Система обробки помилок (відома також як дефектна система обробки) є, по суті, програмним забезпеченням, яке допоможе забезпечити підтримку всіх впорядкованих програмних помилок у всіх розробках програмного забезпечення.

Кожен проект є надзвичайно неповторним, що означає, що ми не можемо мати стару структуру для виконання наших проектів і досягнення успіху в наших зусиллях. Однак, щоб мати хороший план, нам потрібен якийсь вид конструкції або конструкції, щоб слідувати за настановами на природі проекту. Моделі керування проектом або методології проводять конструкцію для виконання проектів.

Різні проекти користуються різними методологіями.Не кожен стиль управління проектами буде працювати на кожну інформацію. Для того, щоб розпізнати, який метод найкраще підійде для вашого проекту, вам слід бути знайомим із цими загальноприйнятими методологіями та різними різними методами.

Для того, щоб користувачі не потрапили в ситуацію, коли програма працює, але не робить того, що повинна робити, проводиться тестування для виявлення «прихованих» помилок. Під час випробувань використовуються плани та дані, розроблені на стадії проектування. Тестування необхідно продумувати протягом усього періоду розробки програми.

Важливим завданням є визначення ресурсів, необхідних для впровадження системи. Це вирішує економічні проблеми, питання щодо вимог до апаратного забезпечення, кваліфікації та підготовки користувачів, потреби у щоденному персоналі служби підтримки та персоналу програмного забезпечення та можливості використання існуючих програмних пакетів. Після з'ясування всіх цих питань вони переходять до планування роботи. Вирішуються питання управління розвитком системи, встановлюються показники для оцінки досягнутого рівня.

Добре оцінений проект відтворення дистрибутує не тільки забави, методи та алгоритми, але й резорбібіліти результату. У кожному проекті є ролі, незалежно від його срізності та кінцевого продукту.

РОЗДІЛ 2
ОБГРУНТУВАННЯ ПІДХОДУ ДО РОЗРОБКИ СИСТЕМИ

2.1. Формальна модель представлення вимог

Формальне представлення функціональних вимог перетворює їх на «машино читаємі», тобто створює передумови повністю автоматизувати процес генерації тестів на їх основі.

Під формальною специфікацією розуміють математичний опис програмної або апаратної системи, котра може бути реалізована на основі цього опису. Специфікується що повинна робити система, а не як вона повинна це робити.

Додатковим бонусом від формалізації вимог є по-перше, те, що протягом процесу їх перетворення виявляються недоліки – неточності або протиріччя у їх тексті, тобто проводиться їх статичне тестування. По-друге, вимоги у такому вигляді мають більшу тривалість життя, ніж у вигляді знань окремих індивідів, а також потребують менш зусиль для підтримки, ніж вимоги сформульовані на природній мові – що призводить до скорочення витрат коштів на підтримку.

Серед існуючих формальних моделей вимог, поведінки та оточення програмного забезпечення для аналізу його властивостей виділяють наступні групи логіко-алгебраїчні, операційні та проміжні.

Логіко-алгебраїчні моделі. При моделюванні ПЗ модель такого типу описує деякий набір його властивостей, які можуть змінюватися з часом, але не дає точного уявлення про те, за рахунок чого змінюються ці властивості. Відмінність між логічним та алгебраїчним численням досить умовне, але, дещо спрощуючи, можна вважати, що логіка має справу з твердженнями в рамках якоїсь мови, а алгебра – з рівностями та нерівностями, які побудовані в деякій мові виразів.

У першому випадку основним об'єктом уваги є твердження, хибні або істинні, а в другому – вирази або терми, пов'язані з якимось типом.

Приклади логічних числень такі:

Обчислення висловлювань. У ньому є атомарні висловлювання, які можливо залежать від об'єктних змінних, а також логічні зв'язки ∧ («і», сполучення), ∨ («або», диз'юнкція), ¬ («не», заперечення), => («отже», імплікація), = (еквівалентність), за допомогою яких можна з одних висловлювань будувати інші, більш складні.

Обчислення предикатів додає перелік висловлювань можливість використовувати квантори по об'єктним змінним для побудови нових тверджень. Квантори в цьому обчисленні бувають двох видів – ∀ «загальності» та ∃ «існування». У численні предикатів крім об'єктних змінних є функціональні і предикатні. Перші являють собою різноманітні функції, результат застосування функції до об'єкта – об'єкт. Другі невизначені твердження, результатом їх застосування до об'єкту має бути або «*true*», або «*false*» . У не типізованому обчисленні всі об'єкти рівноправні, функції та предикати можуть приймати будь-які об'єкти в якості аргументів. У типізованих численнях кожен об'єкт має тип, а функціональні і предикатні змінні – сигнатуру, тобто список типів параметрів і тип результату для функцій. Відповідно, будувати формули можна лише дотримуючись відповідності типів параметрів типам виразів, які підставляють на місце цих параметрів.

Обчислення предикатів більш високих порядків. У цих обчисленнях можна використовувати квантори не тільки по об'єктним змінним, але і по функціональним або предикативним. Наприклад, визначення рівності об'єктів іноді формулюється так: $x=y\equiv\forall P,P(x)=P(y)$, два об'єкти рівні, якщо будь-яке твердження одночасно виконано або не виконано для них обох.

Обчислення з допомогою лямбда – оператор дозволяє будувати функції з виразів, наприклад, вираз *λx x*x* позначає функцію зведення в квадрат. У цьому прикладі також є зв’язаною змінною яка не має власного значення. У *λ* – численні з типами, так само як і в типізованому численні предикатів, об'єкти мають типи, а функції – сигнатури. *λ* – числення більш високих порядків

дозволяють застосовувати λ – оператор не лише до об'єктів, а й до типів. При цьому виходять функції, що перетворюють типи в типи.

Приклади алгебраїчних моделей: реляційні алгебри, що лежать в основі реляційних систем управління базами даних.

Алгебри процесів. Це алгебраїчні обчислення, об'єктами яких є події та процеси, що створюють події або реагують на них. Зазвичай для процесів визначені операції послідовної (';') і паралельної ('||') композиції і операція вибору з двох альтернатив (альтернативна композиція, '+'). Послідовна композиція процесів моделює виконання спочатку першого з них, потім другого. Паралельна композиція моделює паралельне виконання обох процесів. Вибір з двох процесів моделює виконання або першого, або другого. У більшості таких числень процеси можуть взаємодіяти, обмінюючись подіями (один процес породжує подію, інший або інші його споживають). Найбільш широко відомі обчислення процесів *CSP* (*Communicating Sequential Processes*), *CCS* (*Calculus of Communicating Systems*).

Операційні моделі характеризуються тим, що їх можна виконати, щоб простежити зміну властивостей змодельованого програмного забезпечення.

Всі види операційних моделей можна вважати розширенням і узагальненням кінцевих автоматів, тому їх варто розглянути в якості першого прикладу.

Кінцевий автомат являє собою систему з кінцевою множиною станів. В кінцевому автоматі визначено певний набір переходів між станами.

Взаємодіючі автомати представляють собою набір кінцевих автоматів, деякі з яких пов'язані каналами для передачі реакцій одного автомата як стимулів для іншого.

Ієрархічні автомати дозволяють визначати переходи з групи станів (але кінцевий стан такого переходу має бути лише один). Крім того, в ієрархічних автоматах можуть бути групи паралельних станів – кілька груп станів, що об'єднуються в паралельне сімейство.

Тимчасові автомати – зазвичай це розширені автомати, що містять додатковий набір змінних-таймерів, значення яких змінюються самі по собі з

плином часу. Значення таймерів можна використовувати в умовах переходів, зміну змінних або значеннях параметрів реакцій. Крім цього, таймери можна запускати в діях, пов'язаних з переходами. Запущений таймер починає відлік часу з 0. Час може бути дискретним чи безперервним, що дозволяє моделювати поведінку різноманітних систем реального часу.

Гібридні автомати застосовувані для моделювання систем, що взаємодіють з безперервними фізичними процесами. У цих автоматах частина змінних зазвичай має значення, зміна яких описується системою диференціальних рівнянь

Широке використання операційних моделей пов’язано з їх наступними перевагами:

У моделі звичайно враховуються не всі властивості модельованого програмного забезпечення, а тільки важливі для розглянутої в даний момент завдання. Тому моделі виявляються значно простіше модельованих систем, їх набагато зручніше аналізувати.

Віртуальні машини використовуються на практиці мов програмування дуже складні і визначені нечітко, а віртуальні машини моделей значно більш прості і доступні для огляду. Це дозволяє провести вичерпний аналіз можливої поведінки моделі, виявити всі класи можливих при її роботі ситуацій.

Моделі проміжного типу мають риси як логіко-алгебраїчних, так і операційних. Основні їх види такі.

Логіки Хоара є специфічним видом логік, затвердження яких складаються з формул логіки деякого виду та програмних команд. У найпростішому вигляді це трійки $\{Q\}\ S\ \{R\}$, де S – частина програми на певній мові, а Q і R – формули обчислення висловлювань, що залежать від змінних, що входять до S. Q інтерпретується як умова, виконання перед початком виконання S (передумова), а R – як умову, що має виконуватись після виконання R (післяумови). Якщо R дійсно завжди істинне після виконання S у стані, де істинно Q, така трійка теж вважається дійсною і має назву «трійка Хоара». У логіці Хоара для деякої мови програмування, семантика цієї мови задається у вигляді правил виведення, які дозволяють з тавтологій виводити крок за кроком істинні трійки Хоара.

Узагальненням логік Хоара є динамічні чи програмні логіки. Вони є спеціальним типом модальних логік, в яких оператори модальності пов'язані з інструкціями програм. Зазвичай використовуються оператори [*S*] і <*S*>, де *S* – деяка програма. Твердження [*S*]*Q* означає, що завжди після виконання програми *S* формула *Q* істинна, а <*S*> *Q* – що після виконання *S*, *Q* може виявитися істинною. Трійка Хоара {*Q*} *S* {*R*} може бути представлена у динамічній логіці як *Q*=>[*S*]*R*.

Програмні контракти, є окремим випадком логіки Хоара, що звужує можливості використання логічних формул. Програмний контракт представляє собою опис поведінки набору програмних компонентів представлених у вигляді опису сигнатур операцій кожного з цих компонентів, структур їх станів, а також передумов і післяумов для кожної операції та наборів інваріантів для кожного компонента окремо. Інваріант компонента є предикатом, що залежить від елементів стану цього компонента, який повинен бути виконаний в станах, коли жодна з операцій компоненту не виконується. Інваріанти описують обмеження цілісності внутрішніх даних компонента. Передумову операції компонента являє собою предикат, що залежить від елементів стану цього компонента і параметрів цієї операції. При виклику операції з порушенням передумови її поведінка не визначена. Передумова є частиною контракту, яку зобов'язане дотримуватися оточення компонента, щоб забезпечити його коректну роботу. Післяумова операції являє собою предикат, що залежить від параметрів операції, її результату, елементів стану компонента до виклику операції і тих же елементів після закінчення її роботи. Контракти часто неможливо виконати безпосередньо, оскільки післяумова не визначає прямо коректні результати операцій і наступні стани, а лише оцінює передані ним дані.

Серед усіх формальних підходів, що можуть бути застосовані до представлення специфікацій широке практичне застосування у проектуванні і тестуванні засобів автоматики, зв’язку, обчислювальної техніки і програмного забезпечення знайшли скінчені автомати.

Вибір на користь скінченого автомату зумовлений тим, що побудова логіко – алгебраїчних та проміжних моделей практично не вигідна в силу величезних

витрат на розробку додаткових аксіом, тверджень та умов(передумов) та післяумов, що визначають заключні правила отримання правильного результату.

2.2. Основні відмінності у тестуванні веб-додатків

2.2.1. Розрахованість на багато користувачів

Широта аудиторії додатків накладає свій відбиток на специфіку роботи.

1. Один додаток одночасно може використовуватися величезною кількістю людей. Ми вже розглядали питання навантажувального тестування, але також слід звернути увагу на те, що в число користувачів можуть входити представники різних культур, мов і релігій. Нам необхідно пам'ятати про це, особливо якщо мова йде про тестування міжнародного додатка.

2. Кожен користувач може мати свої рівні доступу. В ідеальному варіанті тестувальник створює для себе матрицю рівнів доступу і тестує кожен доступ окремо.

3. Користувачі з одним рівнем доступу можуть звертатися до одних і тих же сутностей, що призводить до конкурентного доступу. Тестується це досить просто. Для прикладу розглянемо систему, що має справу з договорами, які можна створювати, публікувати, редагувати, анулювати. Алгоритм роботи такий: під декількома вікнами в режимі інкогніто авторізуемся в додатку під користувачами з різними рівнями доступу; далі обрану для тесту сутність відкриваємо на редагування, а під другий обліковим записом цю ж сутність пробуємо перевести в статус «Анульовано» – на цьому етапі повинен спрацювати контроль на конкурентний доступ. Операція анулювання блокується, а користувачу видається повідомлення про те,

4. Широта аудиторії говорить про те, що за монітором може перебувати людина, що має злий намір щодо вашого ПЗ.

2.2.2. Мережеві особливості роботи веб-додатків в різних умовах передачі даних

Веб-додатки активно використовують мережу, і це є джерелом можливих проблем. Такий, наприклад, є використання додатків в умовах низької швидкості передачі даних (в браузер *Google Chrome*, наприклад, вбудована функція *Throttling*, яка дозволяє сильно занижувати швидкість передачі даних), в умовах втрати пакетів або при відключенні мережі під час активної фази роботи програми (спосіб імітації: спочатку робимо швидкість передачі даних за допомогою *Throttling* мінімальної, а потім перериваємо мережеве з'єднання під час обробки запиту).

У будь-якому з описаних вище випадків програма має працювати коректно. При «падінні» запиту (*time out*) або іншої проблеми ми повинні, перезагрузив сторінку, знову отримати повністю працює веб-додаток без будь-якого натяку на щойно пережита «шкоди». Для всіх чи функцій програми необхідно подібні тести? Ні в якому разі! В майбутньому можете орієнтуватися на свій досвід, а на перших етапах в цих питаннях краще проконсультуватися з розробниками.

2.2.3. Особливості тестування безпеки веб-додатків

Тестування безпеки – окремий напрямок тестування, яке вимагає від фахівця фундаментальних знань технічного характеру і високої профільної кваліфікації. Зазначимо ряд загальних моментів, які можуть допомогти будь-якому тестувальника знаходити класичні уразливості, не допускаючи їх вихід на продакшен. Питання безпеки додатків регламентуються *OWASP Guide, CHECK, ISACA, NIST Guideline, OSSTMM*.

Існує ряд принципів безпеки, до яких відносяться конфіденційність, цілісність і доступність:

1) конфіденційність – обмеження доступу до тієї чи іншої інформації для певної категорії користувачів (або навпаки надання доступу лише обмеженій категорії).

2) цілісність включає в себе:

а) можливість відновити дані в повному обсязі при їх пошкодженні;

б) доступ на зміну інформації тільки певної категорії користувачів;

3) доступність – ієрархія рівнів доступу і чітке їх дотримання.

2.2.4. Класичні уразливості сучасних веб-додатків

Перерахуємо класичні уразливості сучасних веб-додатків:

1. *XSS* – генерація на сторінці продукту скриптів, які становлять небезпеку для користувачів продукту;

2. *XSRF* – вразливість, при якій користувач переходить з довіреної сторінки на шкідливу, де крадуться представляють цінність для користувача дані;

3. c*ode injection* (*PHP, SQL*) – ін'єкція частини виконавчого коду, яка робить можливим отримати несанкціонований доступ до програмного коду або базі даних і вносити в них зміни;

4. *authorization bypass* – це вид уразливості, при якому можна отримати несанкціонований доступ до облікового запису або документам іншого користувача;

5. переповнення буфера – явище, якого можна досягти у шкідливих цілях, по своїй суті являє використання місця для запису даних далеко за межами виділеного буфера пам'яті.

Цілі в тестуванні безпеки:

- в складних системах, де задіяно багато взаємодій, критично важливі для безпеки функції повинні бути ідентифіковані і ретельно проаналізовані;

- помилки визначено і усунені;

- кількість критичних помилок підтримується на низькому рівні, щоб уникнути непрацездатності системи;

- атрибути безпеки повинні розглядатися як частина всіх рівнів тестування ПЗ.

Додаткову інформацію з безпеки програм можна дізнатися тут: *CHECK, ISACA, NIST Guideline, OSSTMM, OWASP Guide*.

2.3. Аналіз сучасних методик та засобів тестування програмних застосувань

Існуючі на сьогоднішній день методи тестування ПЗ не дозволяють однозначно і повністю виявити всі дефекти і встановити коректність функціонування аналізованої програми, тому всі існуючі методи тестування діють у рамках формального процесу перевірки досліджуваного або розроблюваного ПЗ.

Такий процес формальної перевірки, чи верифікації, Може довести, що дефекти відсутні з точки зору використовуваного методу. (Тобто немає ніякої можливості точно встановити або гарантувати відсутність дефектів у програмному продукті з урахуванням людського фактора, присутнього на всіх етапах життєвого циклу ПЗ).

Існує безліч підходів до вирішення завдання тестування і верифікації ПЗ, але ефективне тестування складних програмних продуктів – це процес у вищій мірі творчий, не зводиться до слідування строгим і чітким процедурам або створенню таких [8].

З точки зору *ISO* 9126, якість програмного забезпечення можна визначити як сукупну характеристику досліджуваного ПЗ з урахуванням наступних складових:

Надійність;

– супроводжуваність;

– практичність;

– ефективність;

– мобільність;

– функціональність.

Існує кілька ознак, за якими прийнято проводити класифікацію видів тестування. Зазвичай виділяють наступні:

1) за об'єктом тестування:

– функціональне тестування (*functional testing*);

– тестування продуктивності (*performance testing*);

– навантажувальне тестування (*load testing*);

– стрес-тестування (*stress testing*);

– тестування стабільності (*stability* / *endurance* / *soak testing*);

– юзабиліті-тестування (*usability testing*);

– тестування інтерфейсу користувача (*UI testing*);

– тестування безпеки (*security testing*);

– тестування локалізації (*localization testing*);

– тестування сумісності (*compatibility testing*);

2) за наявністю доступу до системи:

– тестування Чорної Скриньки (*black box*);

– тестування білої скриньки (*white box*)

– тестування сірої скриньки (*grey box*);

3) за ступенем автоматизації:

– ручне тестування (*manual testing*);

– автоматизоване тестування (*automated testing*);

– напівавтоматизоване тестування (*semiautomated testing*);

4) за ступенем ізольованості компонентів:

– компонентне (модульне) тестування (*component*/*unit testing*);

– інтеграціонне тестування (*integration testing*);

– системне тестування (*system*/*end-to-end testing*);

5) за часом проведення тестування:

– альфа-тестування (*alpha testing*);

– тестування при прийомі (*smoke testing*);

– тестування нової функціональності (*new feature testing*);

– регресійне тестування (*regression testing*);

– тестування при здачі (*acceptance testing*);

– бета-тестування (*beta testing*);

6) за ознакою позитивності сценаріїв:

– позитивне тестування (*positive testing*);

– негативне тестування (*negative testing*);

7) за ступенем підготовленості до тестування:

– тестування по документації (*formal testing*);

– тестування *ad hoc* або інтуїтивне тестування (*ad hoc testing*).

Необхідність автоматизації тестування класифікується за наступними ознаками:

1) за об’єктом тестування:

– функціональному тестуванні (*functional testing*);

– тестуванні продуктивності (*performance testing*);

– навантажувальному тестуванні (*load testing*);

– стрес-тестуванні (*stress testing*);

– тестуванні стабільності (*stability* / *endurance* / *soak testing*);

2) за наявністю доступу до системи:

– тестування чорної скриньки (*black box*);

3) за ступенем ізольованості компонентів:

– системне тестування (*system*/*end-to-end testing*);

4) за часом проведення тестування:

– альфа-тестування (*alpha testing*)

– тестування при прийомі (*smoke testing*)

– регресійне тестування (*regression testing*)

– тестування при здачі (*acceptance testing*)

– Бета-тестування (*beta testing*)

Розглянемо більш детально види тестування в яких може застосовуватись автоматизація:

1) функціональне тестування – це тестування ПЗ в цілях перевірки можливості реалізації функціональних вимог, тобто здатності ПЗ в певних умовах вирішувати задачі, необхідні користувачам.

Функціональні вимоги визначають, що саме робить ПЗ, які задачі воно вирішує.

Функціональні вимоги включають в себе:

– функціональну пригодність (англ. *suitability*);

– точність (англ. *accuracy*);

– здатність до взаємодії (англ. *interoperability*).

– Відповідність стандартам і правилам (англ.*compliance*).

– Захищеність (англ. *security*).

2) тестування продуктивності в інженерії програмного забезпечення – тестування яке проводиться з метою визначення, як швидко працює система або її частину під певним навантаженням. Також може служити для перевірки і підтвердження інших атрибутів якості системи, таких як масштабованість, надійність і споживання ресурсів.

Тестування продуктивності – це одна зі сфер діяльності розвивається в галузі інформатики інженерії продуктивності, яка прагне враховувати продуктивність на стадії моделювання та проектування системи, перед качаном Основному стадії кодування. У тестуванні продуктивності розрізняють наступні напрямки:

– стрес (*stress*);

– навантажувальне (*load*);

– тестування стабільності (*endurance or soak or stability*);

– конфігураційне (*configuration*).

3) навантажувальне тестування (англ. *Load Testing*) – визначення або збір показників продуктивності і годині відгуку програмно-технічної системи або пристрою у відповідь на зовнішній запит з метою встановлення відповідності вимогам, що пред'являються до даної системи (пристрою). Для дослідження годині відгуку системи на високих або пікових навантаженнях проводиться стрес-тестування, при якому створювана на систему навантаження перевищує нормальні сценарії його використання. Не існує чіткої межі між навантажувальним та стрес-тестуванням, однак ці поняття не варто змішувати, так як ці види тестування відповідають на різні бізнес-питання і використовують різну методологію.

Термін тестування навантаження може бути використаний у різних значеннях в професійному середовищі тестування ПЗ. У загальному випадку він означає практику моделювання очікуваного використання додатка за допомогою емуляції роботи декількох користувачів одночасно. Таким чином, подібне тестування найбільше підходить для екстермінатуса мультикористувацьких

систем, частіше – використовують клієнт-серверну архітектуру (наприклад, веб-серверів). Однак і інші типи систем ПЗ можуть бути протестовані подібним способом. Наприклад, текстовий або графічний редактор можна змусити прочитати дуже великий документ; а фінансовий пакет – згенерувати звіт на основі даних за декілька років. Найбільш адекватно спроектований навантажувальний тест дає більш точні результати.

Основна мета навантажувального тестування полягає в тому, щоб, створивши певну очікувану в системі навантаження (наприклад, за допомогою віртуальних користувачів) і, звичайно, використавши ідентичне програмне і апаратне забезпечення, спостерігати за показниками продуктивності системи. В ідеальному випадку в якості критеріїв успішності навантажувального тестування виступають вимоги до продуктивності системи, які формулюються і документуються на стадії розробки функціональних вимог до системи до початку програмування основних архітектурних рішень. Однак часто буває так, що такі вимоги не були чітко сформульовані або не були сформульовані зовсім. У цьому випадку перше навантажувальний тестування буде являтися пробним (*exploratory load testing*) і ґрунтуватися на розумних припущеннях про очікувану навантаженні і споживанні апаратної частини ресурсів.

Одним з оптимальних підходів у використанні навантажувального тестування для вимірювань продуктивності системи є тестування на стадії ранньої розробки. Навантажувальне тестування на перших стадіях готовності архітектурного рішення з метою визначити його спроможність називається '*Proof-of-Concept*' тестуванням.

4) стрес-тестування (англ. *Stress Testing*) – один з видів тестування програмного забезпечення, яке оцінює надійність і стійкість системи в умовах перевищення меж нормального функціонування. Стрес-тестування особливо необхідне для "критично важливого" ПЗ, однак також використовується і для решти ПЗ. Зазвичай стрес-тестування краще виявляє стійкість, доступність і обробку виключень системою під великим навантаженням, ніж те, що вважається коректним поводженням в нормальних умовах.

Термін "стрес-тестування" часто використовується як синонім "навантажувального тестування", а також "тестування продуктивності", що помилково, так як ці види тестування відповідають на різні бізнес-питання і використовують різну методологію.

У загальному випадку методологія стрес-тестування заснована на знятті та аналізі показників продуктивності додатка при навантаженнях, значно перевищують очікувані на стадії супроводу і несе в собі мету визначити витривалість або стійкість додатки на випадок сплеску активності щодо його використання. Необхідність стрес-тестування диктується наступними факторами:

– більша частина всіх систем розробляються з допущенням про функціонування в нормальному режимі і навіть у випадку, коли допускається можливість збільшення навантаження, реальні обсяги його збільшення не беруться до уваги;

– у випадку *SLA*-контракту (угоди про рівень послуг) вартість відмови системи в екстремальних умовах може бути дуже високою;

– виявлення деяких помилок або дефектів у функціонуванні системи не завжди можлива з використанням інших типів тестування;

– тестування, проведеного розробником, може бути недостатньо для емуляції умов, при яких відбувається відмова системи;

– переважно бути готовим до обробки екстремальних умов системи, ніж чекати їх відмови.

Основні напрямки застосування стрес-тестування:

– загальне дослідження поведінки системи при пікових навантаженнях;

– дослідження обробки помилок і виняткових ситуацій системою при пікових навантаженнях;

– дослідження вузьких місць системи або окремих компонент при диспропорційних навантаженнях;

– тестування ємності системи;

– стрес-тестування, як и навантажувальне тестування також може бути використано для регулярної оцінки змін продуктивності с метою отримання для подальшого аналізу динаміки зміни поведінки системи за довгий період.

Стрес-тестування може застосовуватися як для відокремлених додатків, так і для розподілених систем з клієнт-серверною архітектурою. Найпростішим прикладом стрес-тестування відокремленого програми може бути відкриття файлу розміром в 50 Мб програмою *Notepad*, яка входить в комплект ОС *Windows*. Умови стрес-тестування програми зазвичай формуються виходячи з критичних бізнес-процесів його функціональності, визначеними на стадії розробки вимог і аналізу ризиків групою, відповідальною за продуктивність.

У загальному випадку в якості умов для стрес-тестування може використовуватися лінійно збільшена очікувана навантаження.

У разі тестування багатоланкових розподілених систем необхідно враховувати вже не тільки фактичний обсяг навантаження, що складається з безлічі елементів, але і їх пропорції в загальному обсязі.

Використання диспропорційної навантаження в стрес-тестах може також застосовуватися для виявлення вузьких місць окремих компонентів системи.

Тестування стабільності

Тестування стабільності або надійності (*Stability* / *Reliability Testing*) – один з видів тестування ПЗ, метою якого є перевірка працездатності додатка при тривалому тестуванні з середнім (очікуваним) рівнем навантаження.

Перед тим як піддавати ПЗ екстремальним навантаженням варто провести перевірку стабільності в передбачуваних умовах роботи, тобто занурити продукт в повну робочу атмосферу. При тестуванні, тривалість його проведення не має першорядного значення, основне завдання – спостерігаючи за споживанням ресурсів, виявити витоки пам'яті і простежити щоб швидкість обробки даних і / або час відгуку програми на початку тесту і з плином часу не зменшувалася. В іншому випадку можливі збої в роботі продукту і перезавантаження системи.

Часто тестування стабільності суміщають зі стрес-тестуванням, тобто перевіряють не тільки стабільність, але і здатність додатка переносити жорсткі умови і сильні навантаження тривалий час.

Тестування чорної скриньки або поведінкове тестування – стратегія (метод) тестування функціонального поведінки об'єкта (програми, системи) з точки зору зовнішнього світу, при якому не використовується знання про внутрішній устрій

тестованого об'єкта. Під стратегією розуміються систематичні методи відбору і створення тестів для тестового набору. Стратегія поведінкового тесту виходить з технічних вимог і їх специфікацій.

Під "чорною скринькою" розуміється об'єкт дослідження, внутрішній устрій якого невідомо. У кібернетиці дане поняття дозволяє вивчати поведінку систем, тобто їх реакцій на різноманітні зовнішні впливи і в той же час абстрагуватися від їх внутрішнього устрою.

Маніпулюючи тільки лише зі входами і виходами, можна проводити певні дослідження. На практиці завжди виникає питання, наскільки гомоморфізм "чорної" скриньки відображає адекватність його досліджуваної моделі, тобто як повно в моделі відбиваються основні властивості оригіналу.

Опис будь-якої системи управління за часом характеризується картиною послідовності його станів в процесі руху до стоїть перед нею мети. Перетворення в системі управління може бути або взаємно-однозначним і тоді воно називається ізоморфним, або тільки однозначним, в одну сторону. У такому випадку перетворення називають гомоморфності.

“Чорна скринька” являє собою складну гомоморфності модель кібернетичної системи, в якій дотримується різноманітність. Він тільки тоді є задовільною моделлю системи, коли містить таку кількість інформації, яке відображає різноманітність системи. Можна припустити, що чим більше число збурень діє на входи моделі системи, тим більша різноманітність має мати регулятор.

В даний час відомі два види "чорних" скриньок. До першого виду відносять будь-яку чорну скриньку, яка може розглядатися як автомат, званий кінцевим або нескінченним. Поведінка таких "чорних" скриньок відома. До другого виду відносяться такі "чорні" скриньки, поведінка яких може бути проаналізована тільки в експерименті. У такому випадку в явній чи неявній формі висловлюється гіпотеза про передбачуваність поведінки "чорної" скриньки в імовірнісному сенсі. Без попередньої гіпотези неможливе будь-яке узагальнення, або, як кажуть, неможливо зробити індуктивний висновок на основі експериментів з “чорною скринькою”. Для позначення моделі "чорної" скриньки Н. Вінером

запропоновано поняття "білої" скриньки. “Біла скринька” складається з відомих компонентів, тобто відомих *X, Y, δ, λ*. Її вміст спеціально підбирається для реалізації тієї ж залежності виходу від входу. В процесі проведених досліджень і при узагальненнях, висуванні гіпотез і встановлення закономірностей виникає необхідність коректування організації "білого" скриньки і зміни моделей.

Розглянемо, як вивчається і досліджується поведінка "чорної" скриньки другого виду. Припустимо, що дана деяка система керування, внутрішню будову якій невідомо. Система керування має входи і виходи.

Спосіб дослідження поведінки даного "чорної" скриньки полягає в проведенні експерименту, результати якого можна представити у вигляді табл.2.1.

Таблиця 2.1

Результати дослідження методом чорної скриньки

Стан входів	Стан виходів	Час
*X*1 (*T*1),*X*2 (*T*1),…*Xn* (*T*1)	*Y*1 (*T*1),*Y*2 (*T*1),…*Yn* (*T*1)	*T*1
*X*1 (*T*2),*X*2 (*T*2),…*Xn* (*T*2)	*Y*1 (*T*2),*Y*2 (*T*2),…*Yn* (*T*2)	*T*2
…	…	.
*X*1 (*Tk*),*X*2 (*Tk*),…*Xn* (*Tk*)	*Y*1 (*Tk*),*Y*2 (*Tk*),…*Yn* (*Tk*)	*Tk*

Такий спосіб дослідження "чорної" скриньки називається протокольним. Значення вхідних величин в моменти годині можуть вибиратися довільно.

Інший спосіб дослідження полягає в подачі на входи деяких стандартних послідовностей. Цей спосіб особливо привабливий, тому що дозволяє порівнювати поведінку декількох "чорних" скриньок з умовою Вибори таких, які будуть відповідати пропонованим вимогам.

Розробка методів побудови математичних моделей "чорної" скриньки є однією з важливих кібернетичних проблем. За умови наявності математичної моделі "чорної" скриньки з'являється можливість віднести його до якогось одного класу, всі системи якого ізоморфні по поведінці.

Створення математичного опису "чорної" скриньки є свого роду

мистецтвом. У деяких випадках вдається сформувати алгоритм, згідно з яким “чорна скринька” реагує на довільний вхідний сигнал. Для більшості ж випадків робляться спроби встановити диференціальні рівняння, які пов'язують реакцію "чорної" скриньки з його входами або, як кажуть, з його вхідними стимулами.

Для науки метод “чорної скриньки” має дуже велике значення. З його допомога в науці було зроблено дуже багато видатних відкриттів. Наприклад, вчений Гарвей Галі в *XVII* столітті передбачив будову серця. Він моделював роботу серця насосом, запозичивши ідеї із зовсім іншої області сучасних йому знань – гідравліки. Практична цінність методу “чорної скриньки” полягає по-перше, в можливості дослідження дуже складних динамічних систем, і, по-друге, в можливості заміни одного "ящика" іншим. Навколишня дійсність і біологія дають масу прикладів виявлення будови систем методом "чорної" скриньки.

Принципи тестування чорного ящика

У цьому методі програма розглядається як чорний ящик. Метою тестування ставиться з'ясування обставин, в яких поведінка програми не відповідає специфікації. Для виявлення всіх помилок у програмі необхідно виконати вичерпне тестування, тобто тестування на всіх можливих наборах даних. Для більшості програм таке неможливо, тому застосовують розумне тестування, при якому тестування програми обмежується невеликою підмножиною можливих наборів даних. При цьому необхідно вибирати найбільш відповідні, підмножини з найвищою імовірністю виявлення помилок.

Властивості правильно обраного теста

Зменшує більш, ніж на один число інших тестів, які повинні бути розроблені для розумного тестування.

Покриває значну частину інших можливих тестів, що в деякій мірі свідчить про наявність або відсутність помилки до і після обмеженої множини тестів.

Прийоми тестування чорного ящика:

– еквівалентне розбиття;

– аналіз граничних значень;

– аналіз причинно-наслідкових зв’язків;

– допущення про помилку;

– системне тестування.

2.3.1. Тестування через еквівалентне розбиття

Основу методу складають два положення:

1) вихідні данні необхідно розбити на кінцеве число класів еквівалентності;

2) в одному класі еквівалентності містяться такі тести, що, якщо один тест з класу еквівалентності виявляє деяку помилку, то й будь який інший тест з цього класу еквівалентності має виявляти цю ж помилку.

Кожен тест має включати, по можливості, максимальну кількість класів еквівалентності, щоб мінімізувати загальне число тестів.

Розробка тестів цим методом здійснюється в два типа: виділення класів еквівалентності і побудова тесту.

Класи еквівалентності виділяються шляхом вибору кожної вхідної умови, які беруться з допомогою технічного завдання або специфікації і розбиваються на дві та більше групи.

Виділення класів еквівалентності є евристичним способом, однак існує ряд правил:

Якщо вхідна умова описує область значень, наприклад "Ціле число приймає значення від 0 до 999", то існує один правильний клас еквівалентності і два неправильних.

Якщо вхідна умова описує число значень, наприклад "Число рядків у вхідному файлі лежить в інтервалі (1.6)", то також існує один правильний клас і два неправильних.

Якщо вхідна умова описує безліч вхідних значень, то визначається кількість правильних класів, рівна кількості елементів у множині вхідних значень. Якщо вхідна умова описує ситуацію "повинно бути", наприклад "Перший символ має буті заголовним", тоді один клас правильний і один неправильний.

Якщо є підстава вважати, що елементи всередині одного класу еквівалентності можуть програмою трактуватися по-різному, необхідно розбити даний клас на підкласи. На цьому кроці тестувальник на основі таблиці має

скласти тести, що покривають собою всі правильні і неправильні класи еквівалентності. При цьому тестувальник має мінімізувати загальне число тестів.

Визначення тестів:

Кожному класу еквівалентності присвоюється унікальний номер.

Якщо ще залишилися не включені в тести правильні класи, то пишуться тести, які покривають максимально можливу кількість класів.

Якщо залишилися не включені в тести неправильні класи, то пишуть тести, які покривають тільки один клас [13].

2.3.2. Тестування через аналіз граничних значень

Граничні умови – це ситуації, що виникають на вищих і нижніх межах вхідних класів еквівалентності.

Аналіз граничних значень відрізняється від еквівалентного роздроблення наступним:

Вибір будь-якого елемента в класі еквівалентності в якості представницького здійснюється таким чином, щоб перевірити тестом кожний кордон цього класу.

При розробці тестів розглядаються не тільки вхідні значення (простір входів), але і вихідні (простір виходів).

Метод вимагає певної міри творчості та спеціалізації в розглянутій задачі.

Існує декілька правил:

Побудувати тести з неправильними вхідними даними для ситуації незначного виходу за межі області значень. Якщо вхідні значення повинні буті в інтервалі [-1.0. +1.0], Перевіряємо – 1.0, 1.0, – 1.000001, 1.000001.

Обов'язково писати тести для мінімальної і максимальної межі діапазону.

Використовувати перші два правила для кожного з вхідних значень (використовувати пункт 2 для Всіх вихідних значень).

Якщо вхід і вихід програми представляє впорядкована множина, зосередити увагу на першому і останньому елементі списку.

Аналіз граничних значень, якщо він застосований правильно, дозволяє виявити велику кількість помилок. Однак визначення цих кордонів для кожної

задачі може бути окремим важким завданням. Також метод не перевіряє комбінації вхідних значень.

2.3.3. Тестування через аналіз причинно-наслідкових зв'язків

Етапи побудови тесту:

Специфікація розбивається на робочі ділянки.

У специфікації визначаються множини причин і наслідків. Під причиною розуміється окрема вхідна умова або клас еквівалентності. Слідство являє собою вихідну умову або перетворення системи. Тут кожній причині і слідству присвоюється номер.

На основі аналізу семантичного (смислового) змісту специфікації будується таблиця істинності, в якій послідовно перебираються всі можливі комбінації причин і визначаються слідства для кожної комбінації причин.

Таблиця забезпечується примітками, які задають обмеження і описують комбінації, які неможливі. Недоліком цього підходу є погане дослідження граничних умов.

2.3.4. Тестування через припущення про помилку

Тестувальник з великим досвідом вишукує помилки без всяких методів, але при цьому він підсвідомо використовує метод припущення про помилку. Даний метод у значній мірі ґрунтується на інтуїції. Основна ідея методу полягає в тому, щоб скласти список, який перераховує можливі помилки і ситуації, в яких ці помилки могли проявитися. Потім на основі списку складаються тести.

2.3.5. Системне тестування

Системне тестування програмного забезпечення – це тестування програмного забезпечення (ПЗ), що виконується на повній, інтегрованій системі, з метою перевірки відповідності системи вихідним вимогам. Системне тестування відноситься до методів тестування “чорної скриньки”, і, тим самим, не вимагає знань про внутрішню побудову системи.

Основним завданням системного тестування є перевірка як

функціональних, так і не функціональних вимог до системи в цілому. При цьому виявляються дефекти, такі як невірне використання ресурсів системи, непередбачені комбінації даних користувацького рівня, несумісність з оточенням, непередбачені сценарії використання, відсутня або невірна функціональність, незручність використання і т.д. Для мінімізації ризиків, пов'язаних з особливостями поведінки системи в тій чи іншій середовищі, під час тестування рекомендується використовувати оточення максимально наближене до того, на яке буде встановлений продукт після видачі.

Можна виділити два підходу до системного тестування:

на базі вимог (*requirements based*)

Для кожної вимоги пишуться тестові випадки (*test cases*), перевіряючі виконання даної вимоги.

на базі випадків використання (*use case based*)

Альфа-тестування і бета-тестування є підкатегоріями системного тестування.

Альфа-тестування

Альфа-тестування – імітація реальної роботи з системою штатними розробниками, або реальна Робота з системою потенційними користувачами / замовником. Частіше всього альфа-тестування проводиться на ранній стадії розробки продукту, але в деяких випадках Може застосовуватися для закінченого продукту в якості внутрішнього приймального тестування. Іноді альфа-тестування виконується під відладчиком або з використаних оточення, яке допомагає швидко виявляти знайдені помилки. Виявлені помилки можуть буті передані тестувальникам для додаткового дослідження в оточенні, подобному до того, в якому буде використовуватися програма.

Тестування при прийомі

Smoke Test (англ. *Smoke testing*, димове тестування) в тестуванні програмного забезпечення означає мінімальний набір тестів на явні помилки. Димової тест зазвичай виконується самим програмістом, не проходження цього тесту означає що програму не має сенсу віддавати на більш глибоке тестування.

2.4. Побудова тестового набору

Побудова тестового набору передбачає віднаходження такої множини *R*, яка входить у *T* і забезпечує покриття всіх переходів модельного графу – формула

$$co(R) = co(T),$$

де *T* – множина усіх тестів (шляхів графу);

R – результуючий тестовий набір;

co(*R*) – покриття графу *G*, що досягається набором шляхів *R*;

co(*T*) – покриття графу *G*, що досягається набором шляхів *T*, такого що є розв'язком задачі оптимізації $\min T = \min \sum t_i \in R$.

Тестовий набір будується з використанням жадібного алгоритму (рис. 2.1).

Жадібний алгоритм дозволяє отримати оптимальне рішення задачі шляхом здійснення ряду виборів.

У кожній точці прийняття рішення в алгоритмі робиться вибір, який в даний момент виглядає найкращим.

У записі алгоритму використовуються наступні умовні позначення:

| *S* | – потужність множини *S*;

len(*t*) – довжина тесту *t*, яка вимірюється у кількості тестових впливів;

{} – Порожня множина;

<> – Порожній упорядкований список.

Фіксуємо обмеження довжини тестів *M*, яке є параметром даного алгоритму. *C* – повний набір доступних транзакцій, *T* – повний набір раніше побудованих тестів.

Для кожного тесту *t* з *T* обчислюємо *co* (*t*), як безліч тестових ситуацій з *C*, що покриваються даним тестом. Якщо *co* (*t*)={}, то видаляємо *t* з *T*.

procedure ПОБУДОВА_ТЕСТОВОГО НАБОРУ

(*M*: максимальна довжина кроків тесту,

P: вага допустимого перевищення максимальної довжини на користь покриття);

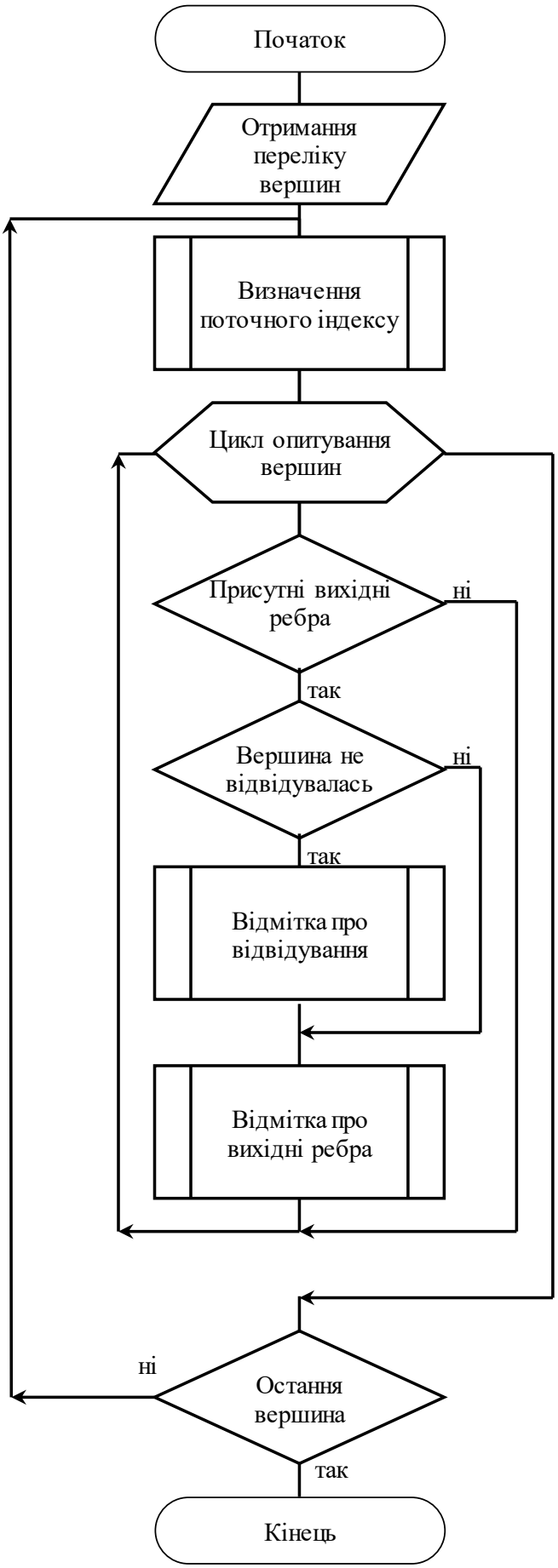


Рис. 2.1. Схема алгоритму побудови всіх маршрутів на графі

begin

$t_1 \in T$ оголошуємо кращим

for кожного тесту t_i з множини тестів T

if $len(t_1) \le M$ && $len(t_i) > M$, то тест t_1 кращий

if $len(t_1) \le M$ && $len(t_i) \le M$ && $co(t_1) > co(t_i)$, то тест t_1 кращий

if $co(t_1) \ge co(t_i)$ && $len(t_1) < len(t_i)$, то тест t_1 кращий

if $co(t_1) > co(t_i)$ && $len(t_1) \le len(t_i)$, то тест t_1 кращий

if $len(t_1) > M$ && $len(t_i)$, а $(len(t_1) - len(t_i))\ P < (co(t_1) - co(t_i))$, то тест t_1 кращий. Додаємо знайдений кращий тест у набір *R*. Видаляємо із безлічі непокритих ситуації *C* всі елементи множини $co(t_{\text{кращий}})$.

end;

end;

Перше правило –гарантує, що в першу чергу всі можливі тестові ситуації будуть покриватися тестами з довжиною, що не перевищує максимальну.

$$len(t_1) \le M \;\&\&\; len(t_i) > M.$$

Друге правило – формула вибирає тести з максимально можливим (в межах обмеження довжини) тестовим покриттям – у більшості випадків ця евристика дозволяє зменшити сумарну довжину тестів в наборі за рахунок зменшення їх кількості.

$$len(t_1) \le M \;\&\&\; len(t_i) \le M \;\&\&\; co(t_1) > co(t_i).$$

Трете і четверте правила – віддають перевагу тестам, що поліпшують покриття або довжину і не погіршує притому іншу з цих двох характеристик.

$$if\ co(t_1) \ge co(t_i) \;\&\&\; len(t_1) < len(t_i)$$

$$if\ co(t_1) > co(t_i) \;\&\&\; len(t_1) \le len(t_i)$$

Згідно з п’ятим правилом з двох тестів, що перевищують максимальну довжину, один краще іншого, якщо збільшення довжини компенсується (з

урахуванням вагового параметра *P*) збільшенням покриття.

$$len(t_1) > M \;\&\&\; len(t_i) > M \;\&\&\; (len(t_1) - len(t_i))\;\; P < (co(t_1) - co(t_i)).$$

Оскільки алгоритм евристичний, і в ньому використовується перебір по неупорядкованій множині, в загальному випадку результати його роботи не детерміновані: за різних запусках на одних і тих же даних можлива генерація різних тестових наборів, а проте в більшості випадків побудовані таким чином набори мають однакові метрики (рис. 2.2).

Для настройки алгоритму використовуються параметри *M* і *P*. *M* задає максимальну довжину тестів набору, що генерується, перевищення якої допускається тільки для покриття недосяжних іншими способами тестових ситуацій.

Допустимі значення: ціле позитивне число або + ∞. *P* задає вагу перевищення максимальної довжини тестів по відношенню до підвищення покриття.

При великих значеннях параметра алгоритм в першу чергу вибирає з тестів, що перевищують максимальну довжину і додають хоча б якесь тестове покриття, самі короткі, що дозволяє зменшити максимальну довжину тестів, але може приводити до істотного збільшення сумарної складності тестового набору; при малих значеннях алгоритм в першу чергу вибирає тести, що покривають більшу кількість тестових ситуацій, що дозволяє мінімізувати сумарну складність тестового набору за рахунок додавання в нього дуже довгих тестів. Допустимі значення: невід'ємні числа.

У ході апробації алгоритму встановлено, що в більшості випадків оптимальне значення для параметра *M* трохи перевершує довжину найдовшого можливого в графі маршруту без циклів, що веде з початкового стану в кінцеве, а для параметра *P* лежить в діапазоні.

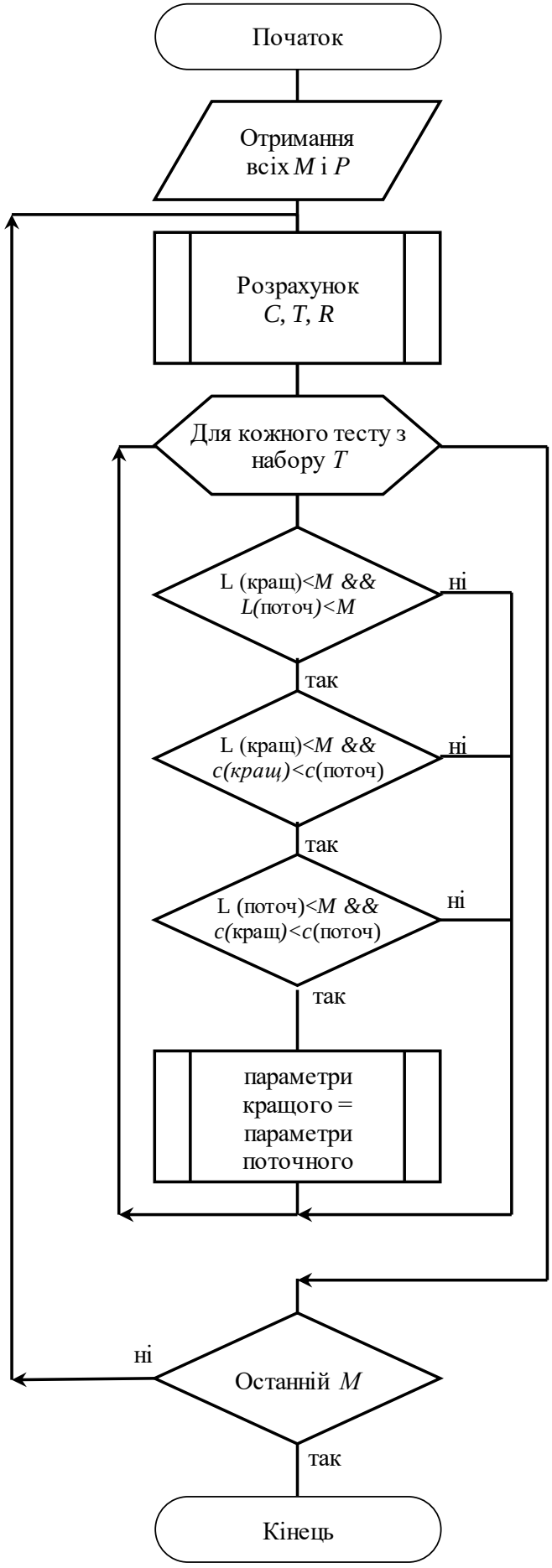


Рис. 2.2. Схема алгоритму побудови тестового набору

2.5. Висновки до розділу

Скінчений автомат є математичною абстракцією в теорії алгоритмів, що дозволяє описувати шляхи зміни стану об’єкта в залежності від його поточного стану і вхідних даних, за умови, що загальна можливу кількість станів скінчене. У процесі роботи скінченого автомату відбувається послідовна зміна скінченого числа його внутрішніх станів, причому стан автомата в певний момент часу однозначно визначається вхідним і вихідним сигналами. Такі автомати являють собою основу всієї сучасної обчислювальної техніки і всіляких дискретних систем автоматичного контролю і управління.

На даний час скінчені автомати – це один з основних інструментів побудови систем тестування.

РОЗДІЛ 3
ОПИСАННЯ РОБОТИ МОДУЛЯ ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ

Веб-додаток представлено наступними складовими («сторонами»):

1. Клієнт. Зазвичай це браузер, але трапляються й винятки (в тих випадках, коли один веб-сервер (ВР1) виконує запит до іншого (ВС2), роль клієнта грає веб-сервер ВС1). У класичній ситуації (коли роль клієнта виконує браузер) для того, щоб користувач побачив графічний інтерфейс програми у вікні браузера, останній повинен обробити отриману відповідь веб-сервера, в якому буде міститися інформація, реалізована із застосуванням *HTML, CSS, JS* (найбільш використовувані технології). Саме ці технології «дають зрозуміти» браузеру, як саме необхідно «отрисовать» все, що він отримав у відповіді.

2. Веб-сервер – це сервер, що приймає *HTTP*-запити від клієнтів і видає їм *HTTP*-відповіді. Щоб уникнути можливої плутанини, відзначимо, що веб-сервером називають як програмне забезпечення, яке виконує функції веб-сервера, так і безпосередньо комп'ютер, на якому це програмне забезпечення працює. Найбільш поширеними видами ПЗ веб-серверів є *Apache*, *IIS* і *NGINX*. На веб-сервері функціонує тестоване додаток, яке може бути реалізовано з застосуванням найрізноманітніших мов програмування: *PHP*, *Python*, *Ruby*, *Java*, *Perl* та ін.

3. База даних – в класичній теорії мова йде про двох «сторін» веб-додатки, проте, якщо уважно подивитися на весь процес роботи додатків, ми можемо відзначити, що в алгоритмі роботи веб-а незримо, але досить активно бере участь ще одна «сторона» – база даних. Фактично, вона не є частиною веб-сервера, але більшість програм просто не можуть виконувати всі покладені на них функції без неї, так як саме в базі даних зберігається вся динамічна інформація додатки (облікові, призначені для користувача дані і ін).

База даних – досить широке поняття, яке використовується не тільки в сфері інформаційних технологій. В контексті моєї статті це – інформаційна модель, що дозволяє упорядковано зберігати дані про об'єкт або групі об'єктів,

що володіють набором властивостей, які можна категоризувати. Бази даних функціонують під управлінням так званих систем управління базами даних (далі – СУБД). Найпопулярнішими СУБД є *MySQL, MS SQL Server, PostgreSQL, Oracle* (всі – клієнт-серверні).

Також існують вбудовані і файл-серверні СУБД. Для загального розвитку зазначу лише одну популярну вбудовану СУБД – *SQLite*, яка використовується в деяких браузерах, *Android API, Skype* і інших відомих додатках. Взаємодія з перерахованими СУБД засноване на спеціальній мові структурованих запитів – *SQL*.

Сервіси і методи тестування можуть бути різними, однак з самого початку необхідно визначитися зі стратегією і дотримуватися її.

Основні види робіт при тестуванні:

1. Співвіднесення фактичного дизайну затвердженим в ТЗ. Жодна розробка сайту не обходиться без помилок в дизайні. Точно зіставити макет до готового дизайну не так просто, так як перевіряти потрібно кожен елемент і вигляд сторінок. Причому мобільна і планшетні версії вимагають ще більше уваги і моделювання можливих адаптацій під кожен гаджет, браузер, але про це в шостому пункті статті.

Щоб фактичний дизайн максимально співвідносився із затвердженим в технічному завданні, нижче представлений чек-лист. Вимоги до дизайну варіюються від проекту до проекту в залежності від різновиду сайту і індивідуальних переваг клієнта, тому список елементів для перевірки включає тільки основні:

Колір. Перевірте на відповідність плану колірну гамму сайту – шрифтів, іконок, ліній, кнопок, фону різних блоків і навігаційних панелей.

Тема (*h*1). Він повинен виділятися від тексту, щонайменше, збільшеним розміром і бути унікальним і єдиним для кожної сторінки. Тема *h*1 служить темою сторінки, припустимо, для блогу таким заголовком буде назва статті.

Підзаголовки (*h*2-*h*3). Вони також зазвичай відрізняються від абзаців, але можуть повторюватися і бути на сторінці в будь-якому необхідній кількості. Приклад – підтема статті блогу.

Посилання. Перевірте, щоб гіперпосилання на сайті відрізнялися оформленням від основного тексту. Найпопулярніший варіант – суцільне або пунктирне підкреслення і контрастний колір шрифту. При наведенні мишкою також має відбуватися зміна кольору (зазвичай в меню) або тільки декорування тексту посилання. Це обов'язково для поліпшення поведінкових характеристик – при перегляді сторінки і побачивши посилання, відвідувач визначить можливість переходу по ній.

Списки. Щоб перерахування ефектно виглядали на сторінці, виділіть маркери кольором відмінним від текстового (ось як нумерація цього списку перед кожним пунктом).

Абзаци і відступи. Відстань між будь-якими елементами сильно впливає на враження, які отримає користувач при контакті з вашим контентом і сайтом в цілому. Приділіть цьому увагу. Перевірте відстань між абзацами, списком і абзацами, заголовком і абзацом, відступи блокових елементів і так далі.

Таблиці. Якщо ви збираєтеся представляти інформацію на *web*-сайті у вигляді таблиці – стилізує її під загальний дизайн.

Цитати. Для блогу актуально наявність контрастного фону або зміна тільки розміру і кольору тексту цитат з вертикальною лінією з лівого боку. Не обов'язково виділяти цитату. Наприклад, на початку цієї статті ми акцентували увагу на важливій на нашу думку інформації.

Розташування. Може бути так, що один з блоків на сайті не відцентрований або навпаки знаходиться посередині, а на макеті визуализировано інше положення. Почекаев відстані, як по горизонталі, так і вертикалі.

Наявність блоків. Перевірте присутність іконок соціальних мереж, кнопок для шаринга, карти розташування, кнопки «наверх» після скролінгу і інших допоміжних елементів.

2. Перевірка верстки на валідність. Щоб розроблений веб ресурс однаково відображався в різних браузерах, витрачалося менше часу на його завантаження і пошукові системи краще розуміли контент, обов'язковим пунктом в тестуванні є перевірка на валідність.

Існує стандарт, який визначає норми і правила сайтобудування –

Консорціум Всесвітньої павутини (*W3C*). На його основі розроблений сервіс *W3C Markup Validation*. Перевірте в ньому свій сайт. Якщо «вилізло» багато помилок, зверніться до веб-майстру для їх усунення. Частина з них може залишитися навіть після виправлень, але не бути критичними.

3. Функціональне тестування. Один з найбільш времязатратних видів тестування, що вимагає правильні розрахунки функцій. Наприклад, візьмемо інтернет-магазин, у якого є не тільки акції на товари, але і безліч статусів при покупці, певне число товарів – все комбінації потрібно перевірити.

Функціонал залежить від типу перевіряється ресурсу, але є базові елементи, на що варто звернути увагу:

Тестування для користувача форм (наприклад, залишити заявку, написати у формі зворотного зв'язку, залишити коментар або відгук);

Перевірка працездатності пошуку та релевантності результатів видачі;

Тестування навігації – переходи по посиланнях для виявлення неробочих;

Перевірки подгрузки файлів на сервер;

Випробування реєстраційної форми та авторизації.

4. Тестування навантаження. Тестувати сайти потрібно обов'язково, щоб не втрачати своїх клієнтів і гроші.

Щоб запобігти несподіване припинення роботи сайту в годину пік відвідування користувачів, важливо провести тестування навантаження. Для цього існують сервіси, які імітують поступове збільшення знаходяться на сайті користувачів. Якщо час завантаження сайту залишається тим самим, що і на рівні з невеликим трафіком, в такому випадку проблема не спостерігається. Для інтернет-магазинів тестувати можна і здійснення одночасних транзакцій, щоб отримати час відгуку всіх найважливіших бізнес-функцій.

5. Тестування безпеки сайту. Від виду сайту перелік робіт, спрямованих на пошук вразливостей, буде відрізнятися. Важливо вчасно усунути їх, щоб хакери не змогли влаштувати «колапс» вашого каналу залучення нових клієнтів через інтернет. Тестування безпеки діагностує шляху злому системи, дає оцінку захищеності сайту і аналіз ризиків доступу зловмисників до конфіденційних даних.

Основні об'єкти перевірки на вразливість:

- контроль доступу;

- діагностика аутентифікації;

- валідація вхідних значень;

- криптографія;

- механізми обробки помилок;

- інтеграція зі сторонніми сервісами;

- перевірка стійкості сайту до *Dos* / *DDos* атакам;

- конфігурація сервера;

- тестування сумісності.

Окрему увагу слід приділити тестуванню сумісності – тип нефункціонального тестування програмного забезпечення, що дозволяє перевірити, чи може ПЗ працювати на іншому обладнанні, операційних системах, додатках, мережних середовищах або мобільних пристроях:

- крос-платформенне тестування сайту. Деякі функції можуть мати проблеми з певними операційними системами, тому необхідно перевіряти роботу програми в різних версіях *Windows, Unix, Mac, Linux, Solaris* і ін.

- крос-браузерні тестування сайту. Також коректна робота залежить від типу браузера. Верстка повинна бути кросбраузерності, щоб забезпечити однакову візуальну частину, доступність, функціональність і дизайн у всіх браузерах. Необхідно перевіряти масштабованість, розширюваність, рамки для елементів у фокусі, відсутність *JS* помилок (лівий нижній кут сторінки). Перевіряти роботу необхідно в таких браузерах, як: *Internet Explorer, Firefox, Chrome, Safari, Opera, Edge* різних версій.

- перегляд на мобільних пристроях. Незважаючи на перевірку роботи веб-додатків в різних дозволах на комп'ютері, найчастіше помилки на мобільних пристроях залишаються непоміченими. Отже, настійно рекомендується перевіряти коректне відображення і роботу вашого веб-додатки на мобільних пристроях різних операційних пристроїв, а також на планшетах;

- тестування БД. Необхідно перевірити правильність здійснення зв'язку з сервером, перевірити сумісність сервера з ПЗ, апаратними засобами, базою даних

і мережею. Також потрібно перевірити що відбувається при перериванні якої-небудь дії, під час наступного з'єднання з сервером під час виконання операцій.

6. Тестування продуктивності. Методика нефункціонального тестування, для вимірювання таких параметрів системи як чуйність і стабільність, при різних навантаженнях. Дозволяє досліджувати швидкість швидкодії сайту та можливості масштабованості додатки, наприклад, при додаванні нових користувачів. Проводиться з метою з'ясувати яке навантаження сайт здатний витримати. Тестування продуктивності вимірює атрибути якості системи, такі як масштабованість, надійність і використання ресурсів.

7. Тестування навантаження – це метод тестування продуктивності, при якому реакція системи вимірюється в різних умовах навантаження. Відповідає за реакцію веб-додатки при збільшенні робочого навантаження. Навантажувальні випробування проводяться для нормальних і пікових навантажень (одночасна купівля товару або авторизація на сайті великої кількості користувачів).

Підхід навантажувального тестування:

– оцінити критерії прийнятності продуктивності;

– визначити критичні сценарії;

– модель робочого навантаження;

– визначте цільові рівні навантаження;

– дизайн тестів;

– виконати тести;

– проаналізуйте результати.

Завдання навантажувального тестування: час відгуку, пропускна здатність, утилізація ресурсів, максимальна призначена для користувача навантаження, бізнес-метрики.

Стрес-тестування (*Stress Testing*) перевіряє систему на її стійкість і обробку помилок в умовах надзвичайно високого навантаження (оцінює як система працює в екстремальних умовах, за межами обмежень і лімітів). Стрес-тестування проводиться, щоб переконатися, що система не буде аварійно завершувати роботу в критичних ситуаціях.

Тестування стабільності / надійності (*Stability / Reliability Testing*) – тип

тестування програмного забезпечення, який перевіряє, чи може програмне забезпечення виконувати безвідмовну роботу протягом певного періоду часу в вказаному середовищі.

Об'ємне тестування (*Volume Testing*) – тип тестування програмного забезпечення, проводиться для аналізу продуктивності системи за рахунок збільшення обсягу даних в базі даних.

8. Тестування паралелізму (*Parallel Testing*) – тип тестування програмного забезпечення, який перевіряє кілька додатків або подкомпонентов однієї програми одночасно, щоб скоротити час тестування.

При паралельному тестуванні тестувальник запускає дві різні версії програмного забезпечення одночасно з одним і тим же введенням. Мета полягає в тому, щоб з'ясувати, чи ведуть себе колишня система і нова система однаково або по-різному.

9. Тестування безпеки. Направлено на оптимізацію безпеки системи при проектуванні, розробці, використанні та обслуговуванні програмних систем і їх інтеграції з критично важливими для безпеки апаратними системами у виробничому середовищі.

Аспекти безпеки програмного забезпечення:

- функціональне програмне забезпечення не повинно створювати небезпек (наприклад: управління сучасним літаком не повинно спрямовуватися в океан).

- системи моніторингу повинні працювати без збоїв (наприклад: резервний комп'ютер повинен запускатися автоматично при збої основного).

Обробка помилок і регресійні тестування. Після завершення розробки веб-додатки слід провести оцінку і аналіз виявлених помилок для подальшого запобігання їх повтору. А також виконати регресійне тестування.

9. Регресійне тестування використовує техніку тестування чорного ящика (повторне виконання тестів), на які впливають зміни коду. Ці тести повинні виконуватися якомога частіше протягом всього ЖЦПЗ при змінах коду для виправлення дефектів або для поліпшення роботи веб-додатки.

Якщо подивитися на автоматично згенерований тест-кейс, то це по суті призначені для користувача сценарії, наведені до одного виду. А значить, їх

можна конвертувати в e2e-тести. Можна навіть відразу писати e2e-тести після перехоплення всіх дій і перевірок, минаючи тест-кейси.

Зараз існує велика кількість різних фреймворків з *gherkin*-нотацією, заснованих на поведінкових сценаріях: *SpecFlow, xBehave.net., Cucumber.js, CodeceptJS*, тощо.

3.1. Логіка роботи програмного модуля

Зростання емпіричних технологій розробки програмного забезпечення призвів до збільшення інтересу до помилок в алгоритмах. Ці алгоритми схильні до областей програмних проектів, які зазвичай схильні до помилок: областей, в яких існує велика ймовірність помилок (це також називають так званими областями помилок). Загальний підхід є статистичним: алгоритми сприйнятливості до помилок базуються на аспектах того, як розроблявся код, та різних показниках, які демонструє код, а не на традиційних показниках або динамічному аналізі. Звичайна стратегія, яка використовується, застосовує метрики коду, тоді як інша шукає особливості вилучення вихідного файлу з історії, спираючись на той факт, що помилки, як правило, скупчуються у багато складних моментів.

Сховища програмного забезпечення є чудовим джерелом знань. Сховище програмних помилок надає корисну інформацію про дефекти програмного забезпечення. Помилка програмного забезпечення враховує кількість атрибутів, багато з яких є текстовими шрифтами. Для отримання знань із повідомлення про помилку можна ефективно використовувати текстову технологію. Класифікація - одна з найпопулярніших даних, що використовує методи класифікації об’єктів у базі даних. Класифікація програмних помилок - це процес групування програмних помилок за різними категоріями. Значна частина роботи зі створення сховищ програмних помилок включає попередню розробку експертів, прогнозування помилок, виявлення дублікатів звітів про помилки та класифікацію запитів на зміни.

В даний час однією з найпопулярніших академічних помилок алгоритму називається FixCache, що є алгоритмом, що використовує ідею "локально": база коду. Використовуючи різні схеми, FixCache створює "кеш" файлів, які можуть зазнавати помилок під час часткового комітету. Коли файл відповідає одному з цих локальних критеріїв, він потрапляє в кеш, а старий файл замінюється вибірковим. FixCache використовує три схеми: якщо файл нещодавно змінено / додано, це схоже на помилки (плинність персоналу); якщо файл пов'язаний з помилкою, він також може бути пов'язаний з іншими недоліками (тимчасова локалізація); а файли, які змінюються разом із помилковими, найімовірніше пов'язані з помилками (просторова локальність). Особливості Fixxache були всебічно досліджені алгоритмом Рахмана, щоб з’ясувати, як він може провести огляд людини. Вони дійшли висновку, що щільність помилок у файлах у кеші, як правило, вища, ніж у кеші, і тимчасовий вибір є найважливішою причиною його ефективності.

Деякі з цих проблем, безумовно, важко вирішити, що призвело до того, що це непросто. Деякі з них, що є кодом, працюють дуже добре, і розчаровані без пригод. Іншими словами, код створює проблеми знову і знову, коли розробники намагаються вирішити проблему.

У двох словах, схильність до ранжування файлів щодо перевірки файлів та історії, а також той факт, що багато змін позначено як виправлення помилок. Звичайно, це означає, що код, який раніше виправляв помилки, все одно відображатиметься у списку. Це питання також розглядалось на початку, і результати завжди зважувались для вирішення цього питання. Можливо, ця ділянка стійкий до тестування, а може дуже конкретний набір команд може призвести до помилок коду. Як правило, наші працьовиті, досвідчені та безстрашні рецензенти можуть впоратися з будь-якими проблемами та вирішити їх.

Подальша припущення полягає в тому, що добре виконувана помилка prediction algorithm надає корисну інформацію в якості розробника. Існує дуже мало емпіричних даних, що підтверджують, що ділянки, яким передбачається помилка, відповідають очікуванням експертів, а також немає даних про те, чи

призводить до інформації, спричиненої помилками, попередженнями про поведінку дітей.

Помилка попередження використовує машинне вивчення та статистичний аналіз, щоб спробувати вгадати, чи є частина коду досить сильно помилковою чи ні, як правило, з тим самим діапазоном довіри. Метрики, засновані на джерелах, які можуть бути використані для попереднього визначення, є багатьма лініями коду, скільки вимагається багато залежних осіб та чи цими залежними є циклічні.

Алгоритм Рахмана фактично знаходить те, що просто ранжирує файли за кількістю тем, які вони були змінені з помилкою, яка випромінює помилку, комою, яка виправляє помилку, і знаходитиме тріщини в базі коди. Просто! Це відповідає інтуїтоніону: якщо файл продовжує вимагати помилок, це, мабуть, є гарячою проблемою, оскільки розробники явно борються з ним.

Зважаючи на швидкість виконання, цей алгоритм також дуже приваблює, оскільки його легко привласнити іншим: файли позначені, якщо вони залучили велику кількість баг-фіксів, більше і більше не менше. У деяких помилках попередження алгоритмів використовується велика кількість метрик, а також багато обчислень перед тим, як вони дають результат.

Впроваджений алгоритм Рахмана працює, створюючи програму, яка заглянула в нашу систему контролю і витягує всі зміни, до яких була прикріплена помилка. Він дивиться на кожну кількість помилок, а також на їх версії з базою даних про відстеження помилок, яка насправді була помилкою, і фільтрує все інше, наприклад, запити на функції. Тоді він дивиться на всі файли, що з’явились у цих змінах, і фільтрує ці, які були видалені і набагато менше. Для кожного файлу кількість змін, пов'язаних з помилками, було підраховано, і ми вивели файли, які були віднесені до 10%.

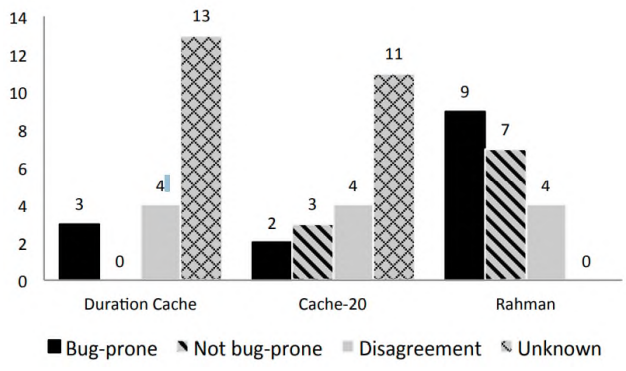
Виявляється, в той час як алгоритм Рахмана показує нам, де знаходяться шпильки, він не пристосовується до змін легко. Якщо команда розробників впорається з останньою, і вона буде зафіксована, вона все одно з'явиться в найменшій мірі, тому що з усіх помилок, що створюються в минулому.

1.1. Помилка Attribute Similarity

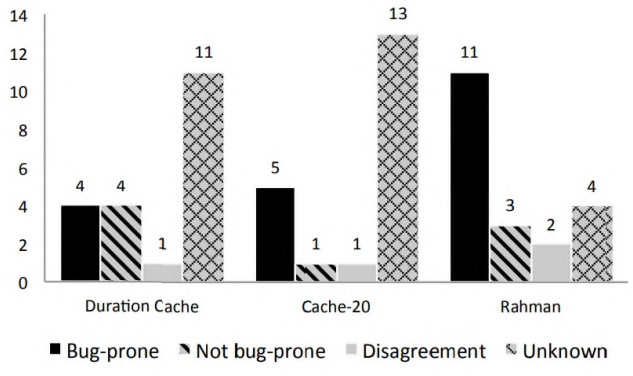
Представлено існування помилки програмного забезпечення classification algorithm, Classification of Bugs Software Software Using Bug Attribute Similarity, що розшифровується як CLUBAS. CLUBAS - це гібрид-алгоритм, який розроблений методом використання текстової кластеризації, таксономічних термінів mappings та частих термінових методів обчислення. Прикладом класифікації за допомогою кластерної техніки є алгоритм CLUBAS. Роботу даної алгоритму можна описати за три основні етапи. На першому кроці текстові кластери створюються за допомогою текстових атрибутів даних про помилку програмного забезпечення. Він виконується на другому кроці, під час якого за допомогою кластера створюються мітки кластера для кожного кластера. На третьому кроці мітки кластерів наводяться на карту разом із таксономічними умовами помилок, щоб визначити відповідні категорії кластерів помилок. Використовуючи часті та незначущі терміни, присутні в атрибутах помилки, створюються мітки кластера, для помилки, яка відмічається від кластерів помилок. Визначений алгоритм оцінювали за допомогою параметрів продуктивності F-мір та точності.

1.2. Bug Prediction Algorithm

Несправності програмного забезпечення на більш ранньому етапі можна передбачити і мають поліпшити надійність програмного забезпечення, його ефективність, якість та зменшення витрат на програмне забезпечення. Ідентифіковано три основні характеристики алгоритму, згідно з якими помилка попередження algorithm повинна бути корисною, коли розробники ідентифікаторів під час перегляду (рис.3.1).



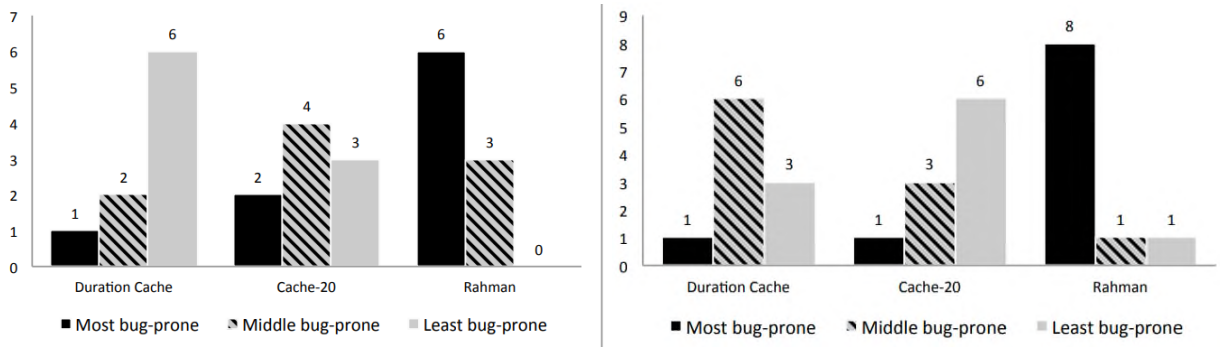
(a) Проект A



(b) Проект B

Фіг. 3.1: Діаграми, що показують кількість файлів певного класу ідентифікаторів від Проекту А та Проекту Б.

Файли можуть бути класифіковані як помилки або ні. Як опозиція між тим, якщо файл або не є помилковим, або файл не знайомий із статевим актом. Чим більше інтенційності з помилкою, а чим менше помилок з протилежними ратінгами, тим кращий результат.



(a) Проект А (b) Проект В

Фіг. 3.2: Діаграми, що показують рейтинги переліків серед учасників у Проекті А та Проекті В.

Ці ознаки трьох алгоритмів викладені у великих дебатах, що підтримуються у статистичних аналітиках, дослідженнях пріоритетності та повідомленнях про помилки.

Кожен рядок показує кількість тим, що останній був визначений певним рангом, наприклад, у проекті А, один розробник вибирає Duration Cache як найвищу помилку, два розробники вибирають як середню, так і найменшу розробницю -prone list. Чим більший рейтинг помилок, тим менше результат, тим менше результат.

1.3. Algorithm Characteristics

Активовані повідомлення

Повідомляючи про те, що зона коду є помилкою, може не бути зрозуміло, як поліпшити програмне забезпечення. Однак існує значне розмаїття між результатами попередження помилок, виконанням і імпровізацією програмних об'єктів, заснованих на цих результатах. Запропонувати пропозиції щодо проблем або помилок має важливе значення, оскільки воно також довго підкреслювалось для використання. Перші дослідження також виявили, що

дескрипція помилки є важливою у вирішенні питання про те, чи є вона виникла. На сьогоднішній день найоптимальнішим є те, що висновок із помилки, попередньої альгоритму, є дієвим. Як тільки буде виявлена помилкова зона, команда розробників зможе зробити чіткі кроки, які приведуть до того, що область, менша за розміром, не позначена. Цей збіг розробника інтуїціону про те, як програма програмує підтримку, - наприклад, перевірки типу або аналітики статистики, які повинні бути інтегровані в роботу. Наприклад, більшість категорій попереджень викликають понад 90% користувачів, які інвестують великі пріоритетні попередження.

На жаль, відношення до джерела інформації означає, що немає нічого, що може зробити команда, яка негайно зняла бланк із файлу; вони повинні намагатись покращити стан здоров'я, а потім тижнями чи місяцями для того, щоб не збирати більше повідомлень про помилки. Це частково розчарування, якщо застереження з’являється часто, що можливо, якщо частина помилки, що змінюється, змінюється кілька разів на тиждень. Проблема, пов’язана з цим, полягає в тому, що завжди є прапор для певного відсотка файлів, тому цей код ніколи не може бути чистим для здоров’я. Це означає, що ці алгоритми ніколи не надають розробникам жодної винагороди, а просто перероблення проблем, які, мабуть, дуже дисертаційні.

Obbious reooning

Коли область позначена як помилка, повинна бути стрінг, візиб і обівіус, чому причина позначення місця, усуваючи будь-який страх перед тим, що прапор є помилковим показником, який буде розроблено лише відходами. Якщо розробнику буде відомо, що, як тільки він видає помилкові заяви, він або вона буде знати про це, що не було. Навантаження на нього завжди є тим, що до кінця з’ясовує, чому воно є прийнятим для виключення. Це здебільшого передається вручну з активованими повідомленнями: повідомлення є лише активним, якщо є чітке та очевидне обґрунтування, чому повідомлення взагалі з’явилося. Перші дослідження також підкреслювали, що програмісти повинні розвивати довіру до аналітичних інструментів.

Розум Рахмана чіткий і очевидний; будь-який розробник може швидко звільнити (і верифікувати) кількість закритих помилок на будь-якій даній файлі. У контракті, іноді не простіше пропонувати його переосмислення; це часто непрозоре, як витягнення в файлі, навіть із певним розумінням алгоритму та введенням, яке було подано. Оглядаючи повідомлення, в кінцевому підсумку всі люди зможуть скласти розумовий образ того, що сталося, і, мабуть, це вказує на те, що якась інформація могла бути написана, щоб пояснити, чому в кеш-пам'яті не було файлу, але такого в даний час немає. Очевидно, що він не вимагає, щоб розробник повністю погодився з оцінкою, лише про те, що він або вона могла б хоча б зрозуміти загальну думку.

Біас до нового

Попри те, що користувачі тестували користувачів, зазначили, що вони вважають, що старші файли мають бути більше помилками, ряд розробників зазначили, що насправді вони не зазнали помилок щодо тих файлів, які вони знають, мають технічну заборгованість. Замість того, розробники були більше віднесені до файлів, які в даний час викликають проблеми (що може або не може включати ці старші файли). Якщо у алгоритму немає активних повідомлень, воно, мабуть, має бути спрямоване на новіші питання, а не прийматися відповідно до старих (які можуть бути кількома роками, а не ідентифікувати поточного стану). Перші дослідження виявили важливість цього характеру при використанні статистичного аналізу для пошуку помилок. Використовувана заміна поліції створює зміщення до нового,

1.4. Algorithm Scaling

В якості попередньої вимоги до депломювання помилка prediction algorithm гарантує, що помилка prediction algorithm повинна масштабуватися до рівня інфраструктури. Ми пробігли не два скаліруючих питання, які затьмарили деплеацію.

Паралельно

Сучасна архітектура великих технологічних компаній зараз відновлює паралельне використання великих комп'ютерних кластерів. В ідеалі ми хочемо усунути помилку, яка передбачає алгоритм, який може виникнути якомога

швидше. Для того, щоб вирішити таку проблему з великими джерелами, яку мають ці великі компанії, алгоритм повинен мати можливість працювати паралельно; Один процесор і сховище даних обробляють сховище дуже повільно. Його можна легко розпаралелювати і швидко запустити з вихідного коду. Якщо когось із них цікавить лише результат кожного проекту, FixCache дуже швидко працює над цим невеликим набором даних, оскільки кілька працівників можуть запускати кілька окремих процесорів і працювати протягом декількох годин.

Ефективність масштабування

Як зазначалося раніше, припущення про 10% файлів є неможливим для людських спекуляцій майже у всіх випадках. У самому проекті 2000 року буде позначено 200 файлів. Більше того, здається, що форма є надмножиною всіх файлів, фактично створених у проекті, оскільки вони є одними з тих, які зараз створюються та змінюються. Це дуже схоже на прапори в цьому випадку, і майже кожен файл, який розробник не може переглянути, матиме вкладене попередження, що різко зменшить його вплив. Однак це також не працює (принаймні, з урахуванням узгоджених на даний момент однорангових показників), коли розмір кешу зменшується. Будь-яка хороша помилка, схильна до алгоритму для людей, повинна бути ефективною, незалежно від того, позначена вона однією чи кількома сотнями.

1.5. Time-Weighted Risk Algorithm

Description

Рахман вже втілює в собі очевидне відновлення, паралельність та ефективність ефекту скалінгу. Ми модифікували його до змісту до нового, результату в новому алгоритмі для перевірки помилок, прихильних до файлів, які ми називаємо Time-Weighted Rísk (TWR) (1). Це визначається як наступне:

$$\sum_{i=0}^{n} \frac{1}{1+e^{-12t_i+w}} \qquad (1)$$

Там, де є помилка, не вказана кількість помилок, і це нормалізоване тимчасове повідомлення про помилку. ti працює від 0 до 1, з 0 від самого раннього кубика і 1 від останнього підкомірованого кубика (на практиці це

значення є просто тим, коли виконується скрипт, ω визначає, як на тлі занепаду повинен бути крива по осях x).

Це рівняння підсумовує всі помилкові файлові комітети, беручи до уваги, що комітети базуються на тому, як вони є, а старші колеги шукають нуль. Математично це не має значення, але врешті-решт він може швидко відфільтрувати ці подальші втрати пам'яті та покращити стан. Функцією ваги є модифікована крива, де вісь X - нормований момент, а вісь Y - екран. Оцінка не стандартизована, оскільки призначена лише для порівняння між чоловіками. Крім того, справа в тому, що стан справи змінюється щодня, оскільки час нормалізації змінює вагу комітету. Це за задумом, оскільки старі файли, які залишаються незмінними, повинні почати розкладатися.

В ідеалі це слід динамічно обчислювати весь час, щоб тримати занепад приблизно на одному рівні; з плином часу вікно розширюється. Це вирівнює регіон між старими та новими файлами, які розглядаються на основі їх останніх проблем, і не враховуючи поточні проблеми в минулому, які можна було виправити. Значення ω залежить від покладу, і воно буде змінюватися залежно від того, що розпад має інтуїтивний сенс для навколишнього середовища. Щоб визначити, чи позначка файлу призвела до більшої участі розробника в огляді; щось на зразок того, що вони думають про огляд глибше, ніж могли б мати.

Для дослідження є два метрики:

- середній час перегляду вмісту помилки-файлу займає від submission до затвердження;
- середня кількість коментарів на відео, яке містить файл із помилкою.

Найменша з усіх файлів, які були позначені як помилка під час будь-якого запуску. Потім ми проводимо перегляди з трьох місяців через кожну дату запуску, яка не включала ці файли. Ми проаналізували тимчасовий перегляд для отримання від субмісії за заявкою та кількість коментарів, які отримав ревізійний звіт.

Результати

Розробник поведінки, оскільки загальні середні середні показники змінювались раніше і після запуску. Для цього тесту були вироблені фільтри, які

були нижчими за 5 відсотків і більшими за 95 відсотків кожного набору даних, щоб запобігти екстремальним викидам від викривлення результатів.

Незалежно від того, збільшились чи зменшились кошти для кожної окремої файли, зміни тестів не є іншими ознаками, що підтверджують нульові гіпотези, згідно з якими помилка попереднього відтворення не вплинула на розробників.

Зворотній зв'язок розробника як внутрішній відгук розробника змішувався, але схильність до негативу. Ми моніторили запити на функції, надіслані нам безпосередньо як повідомлення про помилки, а також внутрішні повідомлення про помилки, де проект був підготовлений. Включені комплайнти:

- no opt-out function;
- confusion від обох субмітерів та ревізорів про те, наскільки "fix" файл перед субмісіоном міг бути затверджений (оскільки алгоритм не діє, не було жодних засобів до цього);
- технічні боргові файли мали б бути позначені проти і проти, якщо розробники відчували, що вони не були ні для чого, щоб зробити будь-які певні зміни;
- автоматично згенеровані файли не фільтрувались, і часто їх змінювали, тому в кінцевому підсумку приєднувались до виправлення помилок та отримували позначки;
- команди, які використовували програмне забезпечення для відстеження помилок, відчували себе несправедливо позначеними.

Була інша ідея, що розробники, можливо, намагалися продемонструвати свої відстежувані помилки, правильно відмітивши їх із позначкою помилки або запиту на функцію (лише помилки, не запити на функції, що відстежуються проектом), але наші інвестігації від будь-яких даних значні зміни в радіо помилок у порівнянні із запитами щодо функцій, що повторюються. Якщо відбулася якась зміна у перезаписі, так що лише помилки характеризуються як so, ми б очікували, що кількість помилок, записаних до зниження, і кількість запитів на функції збільшиться. Оскільки щур не змінювався, ми не знаємо, що відбулися якісь значні зміни у поведінці.

Загальна методологія кластеризації програмного забезпечення, що відображається в додатку А, і розділена на всі наступні кроки: перший крок включає відновлення програмного забезпечення програмного забезпечення, програмне забезпечення, програмне забезпечення, програмне забезпечення, програмне забезпечення Помилки програмного забезпечення доступні у форматі файлів HTML (мова гіпертекстової розмітки) або файлів XML (розширювана мова запитів), які необхідно аналізувати для атрибутів програмних помилок. Після розбору програмних програмних помилок, виправлених із єдиних копій, локальні схеми бази даних визначаються для локального зберігання програмних помилок у локальній базі даних. Одного разу про помилку програмного забезпечення, яке було завантажено в локальну базу даних, і воно доступне для перформансових даних, що містяться в операційній системі, він трансформується в терміни об'єкта Java, щоб забезпечити його зберігання та подальшу обробку в колекції Java Java API (Application Programming Interface). На наступному кроці текстові техніки попередньої обробки, такі як ступінь елімінації та стебло, виконуються для попередньої обробки програмних записів про помилки. Шляхи роботи (марні слова) не мають жодного сенсу в знаннях про знищення, і, отже, їх потрібно елімінувати. Стрижень вимагається для того, щоб визначити терміни, присутні в текстовому документі, так що шаблони знань можуть бути виявлені ефективно. Шляхи роботи (марні слова) не мають жодного сенсу в знаннях про знищення, і, отже, їх потрібно елімінувати. Стрижень вимагається для того, щоб визначити терміни, присутні в текстовому документі, так що шаблони знань можуть бути виявлені ефективно. Шляхи роботи (марні слова) не мають жодного сенсу в знаннях про знищення, і, отже, їх потрібно елімінувати. Стрижень вимагається для того, щоб визначити терміни, присутні в текстовому документі, так що шаблони знань можуть бути виявлені ефективно.

Оброблені дані про помилки подаються для кластеризації, де кластери помилок створюються за допомогою вивішених текстових символів атрибутів помилок. Після створення кластерів помилок мітки кластерів створюються для кожного кластера. Етикетки генеруються за допомогою частих термінів, присутніх у попередньо оброблених даних про помилки, які зв’язуються з

окремим кластером помилок. Порівняно з мітками кластера ідентифікуються та елімінуються під час генерації міток кластера. Наступним кроком є відображення термінів, присутніх у мітках кластера, до категорій помилок програмного забезпечення, що використовують таксономічні терміни, ідентифіковані для категорії помилок, для того, щоб класифікувати кластери помилок. Фінальний крок полягає у створенні confusion matrix для класифікованих кластерних кластерів, за допомогою яких обчислюються параметри продуктивності як попередній, відкликання, F-міра та точність.

1.6. CLUBAS Algorithm

CLUBAS приймає два параметри для класифікації помилок, тобто порогове значення текстового значення (τ) та кількість часто використовуваних термінів у мітці кластера (N). Початковою фазою в CLUBAS є вилучення даних, де помилка реєструється з окремого сховища помилок, яке отримується та зберігається в локальній системі. Вибір помилок запису може бути здійснений кількома способами, наприклад, випадковим чином (генерування набору випадкових чисел для помилок), вказівкою діапазону помилок (максимальна помилка та максимальна помилка), або, точніше, помилка в тексті файл. Одночасно вибирається одна помилка, яка додається до URL-адреси (Uniform Resource Locator) програмного забезпечення для пошуку помилок, а в Java за допомогою URL-адреси програмується помилка мережі.) або XML (розширювана мова розмітки) для формату.

Наступним кроком у CLUBAS є етап попередньої обробки, де помилка програмного забезпечення реєструється локально у форматі файлу HTML або XML, а атрибути помилок та їх початкові значення зберігаються в локальній базі даних. Після припинення цього кроку виконуються усунення та контроль текстового опису помилок (тексту) та опису, що використовується для створення кластерів помилок. На наступному етапі (кластеризація) для вимірювання текстового значення вибираються раніше виміряні атрибути програмних помилок. Той самий метод калібрування використовується для вимірювання зваженого значення між декількома програмними помилками. Тут для обчислення текстових символів використовується відкрита програма Java API,

симетрична, яка забезпечує реалізацію різних текстових символів. Для всіх помилок програмного забезпечення зважені розміри обчислюються та зберігаються, за допомогою яких створюються кластери. Кластери створюються наступним чином: спочатку один кластер створюється з використанням випадкової помилки, потім, якщо значення помилки для порушених помилок менше цього порогового значення (τ), то нижчі кластери, тобто створено новий кластер, і помилка, яка не пов'язана з кластером, відображається на новий кластер. Це величезне значення є одним з важливих параметрів для алгоритму CLUBAS. Якщоτ (порогове значення) є великим, тоді для кластеризації очікується високе значення між характеристиками програмних помилок і навпаки.

Наступним кроком (Створення мітки кластера) є створення міток кластера з використанням частих термінів, які є в помилках кластера. На цьому кроці підсумовується зведення (опис) та опис усіх програмних помилок, пов’язаних з окремими кластерами, і в цих зведених текстових даних обчислюються часті терміни, а N (де N - кількість частих термінів у мітках і є і є міткою). параметр, що надається користувачем) для найпоширеніших термінів кластери призначаються як мітки кластера. Відображення міток кластера для помилки класифікується з використанням таксономічних термінів для різних категорій, які виконуються далі (відображення кластерів до класів). На цьому етапі терміни таксономії для категорій помилок визначені заздалегідь, і умови міток кластера відповідають цим термінам. Порівняння умов свідчить про належність кластерів до категорій. Останній крок (оцінка ефективності та представлення результатів) формує матрицю комбінацій, за допомогою яких обчислюються різні параметри продуктивності, такі як точність, зворотний зв’язок та точність. Точність та зворотний зв'язок можна об'єднати для розрахунку F-міри, формули цих параметрів викладені в наступному розділі. Нарешті, кластер комп'ютеризований і представлений як вихід CLUBAS.

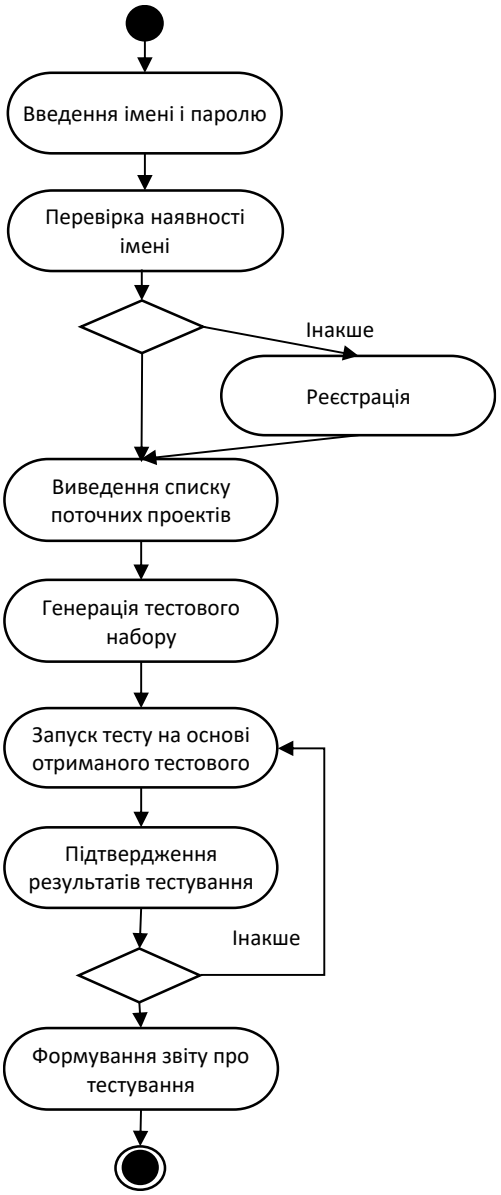


Рис. 3.1. Діагарма видів діяльності тестування

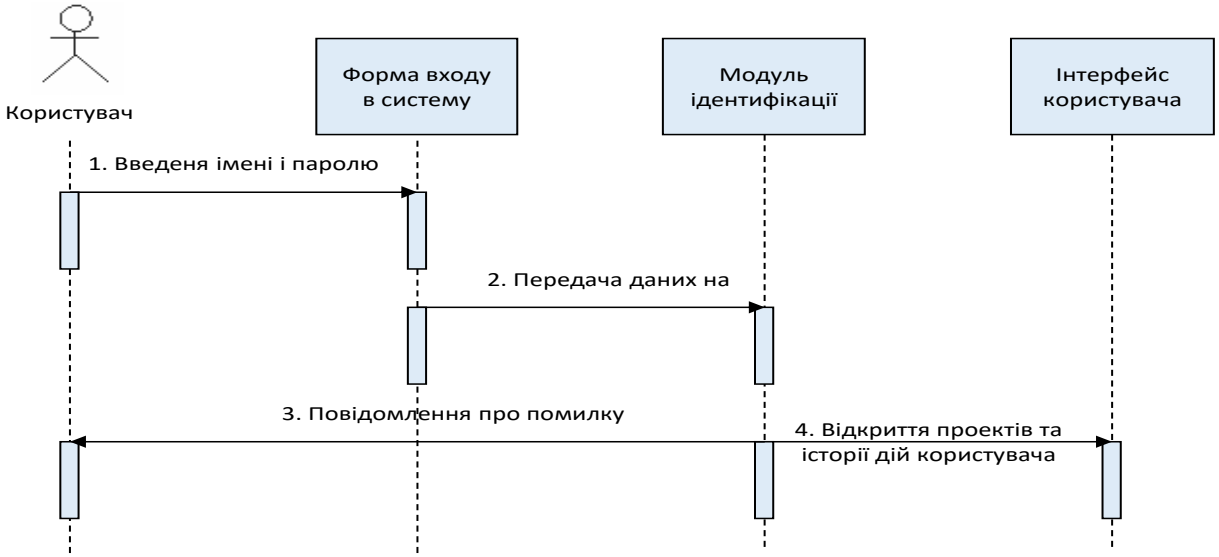


Рис. 3.2. Діагарма послідовності дій при авторизації

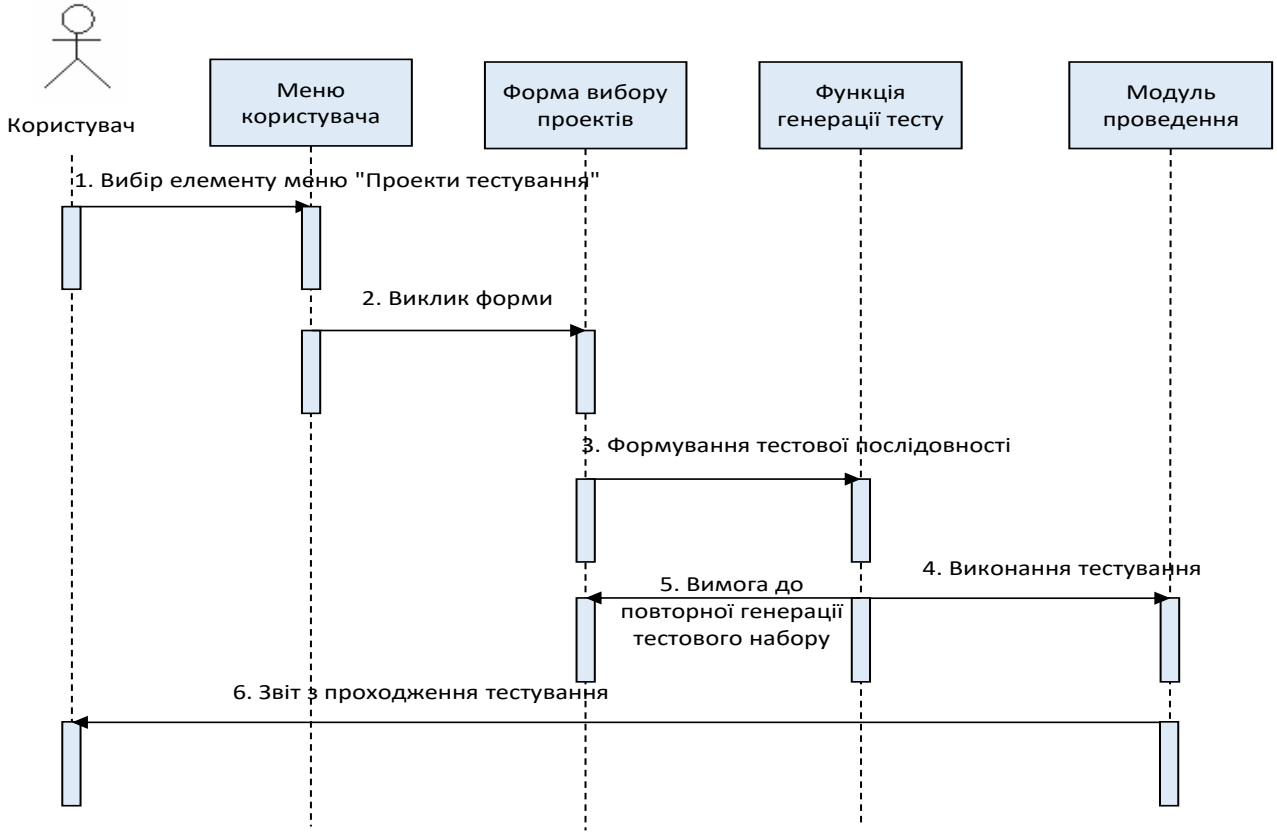


Рис. 3.3. Діаграма послідовності дій тестування web-додатків

Тест починається з вимоги (називається *User Story* в гнучких проектах). Досить часто першою простою вимогою до тестування є: Аутентифікація користувача (рис. 3.4).

Рис. 3.4. Приклад вікна веб-додатку для тестування

Аналізуючи вимогу та програму можемо почати збирати дані тесту та розробити кроки тесту:

1. Введіть ім'я користувача "*agileway*";

2. Введіть пароль "тестово";

3. Натисніть кнопку «Увійти»;

4. Переконайтеся, що з’являється повідомлення “*Welcome agileway*”.

Програмний модуль має структуру проекту для організації тестових сценаріїв. Ця структура – це просто папка, що містить усі пов'язані з тестом файли, такі як тестові скрипти та дані тесту.

Створимо гілку проекту (рис. 3.5).

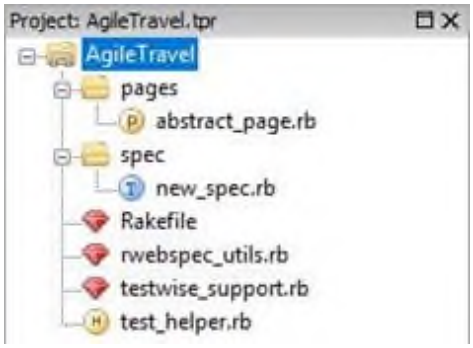


Рис. 3.5. Гілка проекту для тестування

Незалежно від того, яку структуру тесту використовують, "ритм тестування" однаковий:

1. Визначити веб-елемент керування;

2. Виконати операцію на елементі управління;

3. Виконати перехід до кроку 1, поки не дійдете до контрольно-пропускного пункту;

4. Виконати перевірку;

5. Виконати перехід до кроку 1, поки тест не закінчиться.

Після того, як визначили веб-елемент керування, наступним кроком є виконання необхідної операції з ним, наприклад, введення тексту в текстове поле, натискання кнопки, зняття прапорця тощо. Хоча різні тестові рамки мають різний синтаксис, ідея однакова.

Мета тестування – перевірити, чи відповідає частина функції своїй меті. Після "підведення" програми до певної точки ми проводимо перевірки (можливо, тому в деяких інструментах тестування це називається "контрольною точкою").

У контексті веб-тестування типовими перевірками є:

– перевірити наявність певних текстів;

– переконайтись, що наявні певні фрагменти *HTML* (на відміну від вище, це

для перевірки вихідного джерела сторінки)

– перевірити заголовок сторінки;

– перевірити наявність посилання$

– переконайт*bcmя*, що веб-елемент керування присутній або прихований/

Однією з ключових особливостей тестових фреймворків є надання синтаксичних конвенцій для виконання перевірок, як зазначено вище.

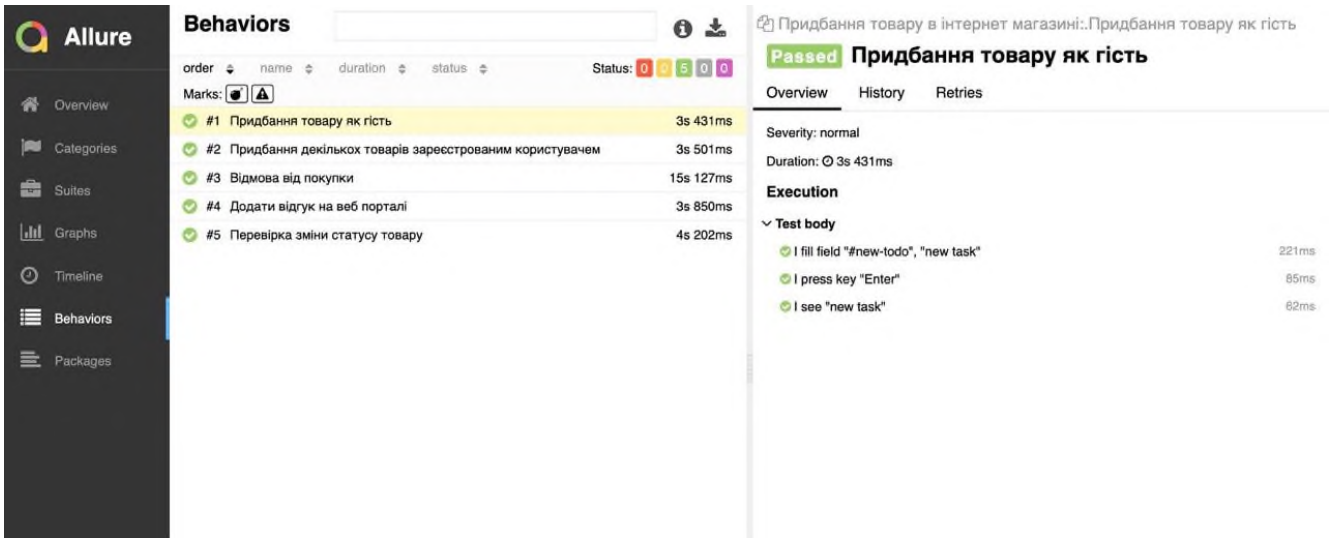


Рис. 3.6. Уніфіковані тести

3.2. Роботя модуля генерації тестів

Існує безліч різних систем, які реалізують бажану функціональність, ми будемо консолідувати системи з відкритою API, або системи, які всі використовують для використання, щоб використовувати базу даних системи як джерело даних.

Найпопулярнішими системами серед користувачів є Jira, Redmine, Bugzilla, хоча в деяких випадках вони використовуються для управління проектами. Розроблений Атласіан, він є одним з двох основних продуктів (як і з вікнами впливу). Він має веб-інтерфейс. Назва системи поширюється шляхом обрізання за допомогою "GoJIRA" - японського імені чудовиська Godzilla, яке, в свою чергу, є посиланням на ім'я продукту з виробництва - Bugzilla; був створений як заміна Bugzilla і багато в чому повторює його архітектуру. Система дозволяє вам

працювати з безліччю проектів. Для кожного проекту, створює та підтримує безпечні схеми та сигнали безпеки.

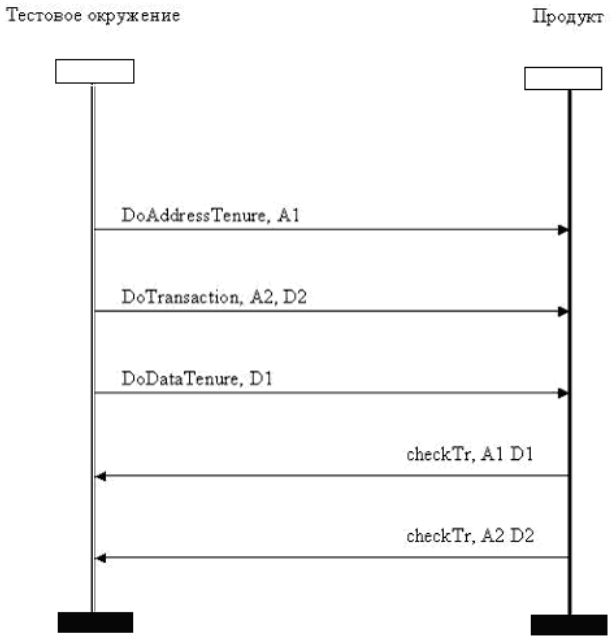


Рис. 3.7. Формальний запис сценарного тесту

```
Perform a click on a link or a button, given by a locator.
If a fuzzy locator is given, the page will be searched for a button, link, or image matching the locator string.
For buttons, the "value" attribute, "name" attribute, and inner text are searched. For links, the link text is search
For images, the "alt" attribute and inner text of any parent links are searched.

The second parameter is a context (CSS or XPath locator) to narrow the search.

```js
// simple link
I.click('Logout');
// button of form
I.click('Submit');
// CSS button
I.click('#form input[type=submit]');
// XPath
I.click('//form/*[@type=submit]');
// link in context
I.click('Logout', '#nav');
// using strict locator
I.click({css: 'nav a.login'});
```

@param {CodeceptJS.LocatorOrString} locator clickable link or button located by text, or any element located by CSS|X
@param {?CodeceptJS.LocatorOrString} [context=null] (optional, `null` by default) element to search in CSS|XPath|Stri
```

Рис. 3.9. Вікно з описанням помилки в роботі програмного забезпечення

```
const faker = require('faker');                          // Use 3rd-party JS code

Feature('Store');

Scenario('Create a new store', async ({ I, login, SettingsPage }) => {
  const storeName = faker.lorem.slug();
  login('customer');                                     // Login customer from saved cookies
  SettingsPage.open();                                   // Use Page objects
  I.dontSee(storeName, '.settings');                     // Assert text not present inside an element (located by CSS)
  I.click('Add', '.settings');                           // Click link by text inside element (located by CSS)
  I.fillField('Store Name', storeName);                  // Fill fields by labels or placeholders
  I.fillField('Email', faker.internet.email());
  I.fillField('Telephone', faker.phone.phoneNumberFormat());
  I.selectInDropdown('Status', 'Active');                // Use custom methods
  I.retry(2).click('Create');                            // Retry flaky step
  I.waitInUrl('/settings/setup/stores');                 // Explicit waiter
  I.see(storeName, '.settings');                         // Assert text present inside an element (located by CSS)
  const storeId = await I.grabTextFrom('#store-id');     // Use await to get information from browser
  I.say(`Created a store with ${storeId}`);              // Print custom comments
}).tag('stores');';
```

Рис. 3.10. Результати автоматичного тестування

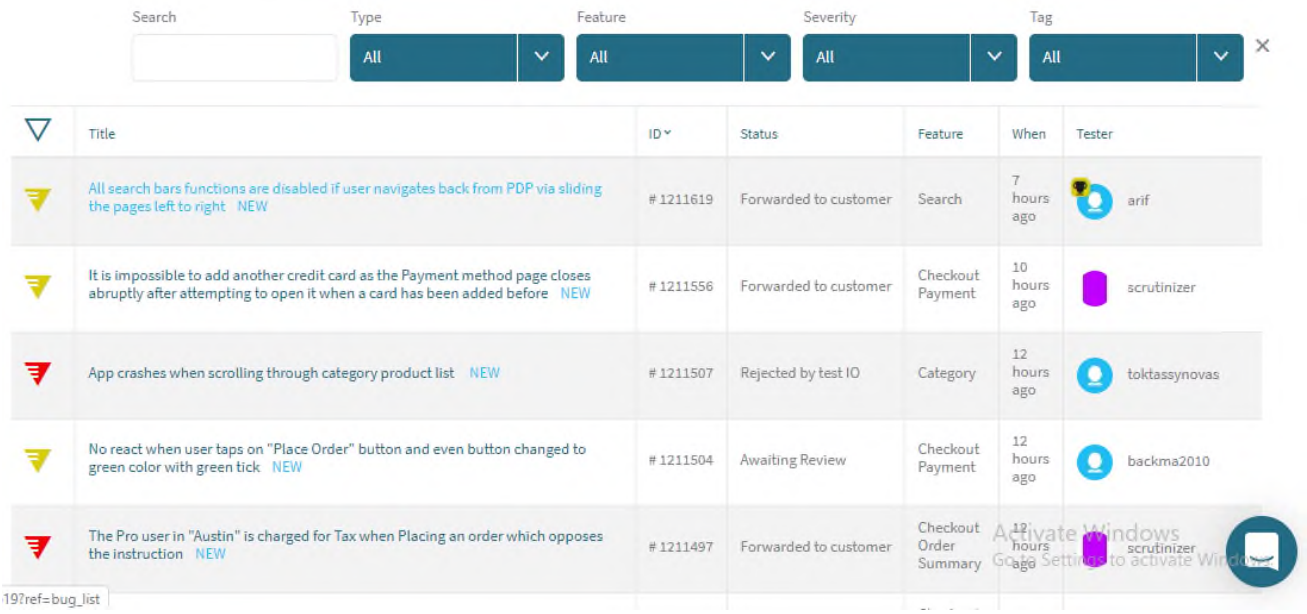


Рис. 3.11. Вікно знайдених помилок з інструментарієм пошуку

3.3. Висновки до розділу

Завдання - створити групу помилок у програмному забезпеченні, а потім провести класифікацію, використовуючи різні терміни, визначені з різних сховищ програмних помилок. Застосування запропонованої роботи забезпечує ефективне управління помилками та усунення повідомлених помилок. Оскільки класифікація проводиться за допомогою кластерів програмних помилок, які є

групою програмних помилок, менеджери проектів можуть використовувати цей аналіз для ряду завдань управління.

Представлений текстовий кластер та алгоритм класифікації, а також інтерфейс на основі графічного інтерфейсу для класичної програмної помилки CLUBAS. Алгоритм CLUBAS визначається за допомогою техніки класифікації кластерів, при якій перший кластеризується за допомогою атрибутів текстових атрибутів, а потім для кожного кластера створюються та присвоюються відповідні мітки. Потім мітки кластера відображаються у класи помилок за допомогою міток кластера та відповідних умов оподаткування помилок.

Алгоритм використовує два вхідні параметри: незначний поріг та кількість термінів у мітці кластера. Вплив цих параметрів на оцінку ефективності також вивчається, і графіки готуються до їх впливу. Алгоритм CLUBAS відображається у ряді стандартних алгоритмів класифікації для оцінки продуктивності. З експериментів було встановлено, що CLUBAS здатний забезпечити точність понад 80% для всіх сховищ помилок у різних точках вибірки (кількість вибірки).

З порівняльного аналізу можна зробити висновок, що точність вища, лише алгоритми NB та NBM кращі за алгоритми CLUBAS, однак він дає найкращий алгоритм, хоча дає певні результати за низкою показників, незважаючи на те, що це дає хороші результати по ряду показників, хоча дає хороші результати по ряду показників, хоча дає деякі результати, хоча дає різні результати, хоча дає різні результати, включаючи деякі параметри, хоча дає різні результати, кількість і багато в чому classification. Майбутній діапазон, пов’язаний із запропонованою роботою, може бути застосований до передових методів попередньої обробки тексту для вибору кластерної та класифікаційної робіт, а також для сучасних дітей, які навіть не підозрювали про це.

Зараз доступно багато програм для управління вашими проектами; Ви можете знайти абсолютно будь-яке програмне забезпечення, що підходить для потреб Вашої приватної компанії. Проблема в тому, що вибрати. Щоб зрозуміти, що послуга пропонує «організовувати стосунки, людей та проекти, сьогодні легко і самотньо», і це правильний вибір для часткової розробки продукту.

Важливо також протидіяти різним системам відстеження помилок, тоді як кодовані алгоритми працюють як частина цих систем. Більше того, незважаючи на те, що вони не мають досвіду роботи з такими системами, вкрай складно вибрати саме той продукт, який йому потрібен і який задовольнить цілі команди розробників. Ось чому було встановлено, що програмне забезпечення для відстеження помилок було обрано за різними критеріями: безкоштовне та комерційне, відстеження помилок та інтеграція в системи управління тестами. Деякі з них більше підходять для невеликих команд, інші є кращими для великих команд та проектів. Основна критика, яку потрібно проаналізувати, коли ви вибираєте інструмент відстеження помилок.

ВИСНОВКИ

Рано чи пізно у зростаючій компанії безкоштовна система управління проектами перестає справлятися з потоком вхідних завдань, а її недоліки переважають усі існуючі переваги. І тоді потрібно зробити правильний вибір і заплатити за систему, яка відповідатиме всім необхідним вимогам.

Управління проектами зараз є одним з найважливіших джерел бізнесу, оптимізованого в найширшому розумінні. Ця сфера діяльності зараз широко автоматизована, постійно виникають нові продукти з інформаційними технологіями, які суттєво змінюють управління конвенціональними методами в галузі виробництва та надання різноманітних послуг. Розвиток будь-якої компанії та рівень її конкурентоспроможності значною мірою залежать від того, наскільки добре управління її організованими ресурсними ресурсами. Це головне завдання оперативного управління проектами управління.

Метою запропонованої роботи є створення групи подібних програмних помилок, а потім класифікація цієї групи за допомогою дискримінаційних термінів, визначених з різних сховищ програмних помилок. Застосування запропонованої роботи полягає у забезпеченні ефективного управління інформацією про помилки та швидшому усуненні зафіксованих помилок. Оскільки категоризація виконується з використанням кластерів програмних помилок, які є групою подібних програмних помилок, менеджери проектів можуть використовувати цей аналіз для ряду завдань управління. Ось приклади таких завдань:

- створення груп подібного набору програмних помилок;
- команди розробників можна визначити для групи категоризованих помилок, подібні нові помилки можна призначити одній і тій же групі розробників для оптимізації часу ремонту;
- кластери помилок (або категорії) можуть бути зіставлені з програмними модулями, за допомогою яких ви можете виконати аналіз складності модулів шляхом обчислення помилок для кожного модуля;

• класифікувати помилки за категоріями, які допоможуть менеджерам зрозуміти сильні та слабкі сторони розробників та програмних модулів.

Ефективні інструменти моніторингу проектів та професійна методологія дозволяють будь-якому проектному бюро мінімізувати відсоток тих проектів, які спочатку не можуть вписатися в бюджет та терміни та досягти своїх цілей. Скорочення часу на пошук та збір інформації, а також ручне складання всіх звітів про проект звільняє левову частку часу на виконання більш важливих і серйозних завдань.

Серед можливих вигод від використання систем управління проектами є той факт, що кількість проектів, які не відповідають стратегії компанії, зменшується, а це означає, що витрати на весь портфель проектів будуть значно зменшені. Також системи управління проектами дозволяють оптимізувати розподіл ресурсів і контролювати використання працівників, підвищувати ефективність планування.

Вибираючи програму пошуку помилок, слід шукати:

• зручний та інтуїтивно зрозумілий інтерфейс;

• багатомовна підтримка;

• обліковий час;

• простий в установці;

• можливість підтримати велику кількість проектів.

Відстеження помилок - це процес збору, запису та управління даними про помилки, що виникає в програмному забезпеченні (також звані помилками та винятками). Метою є підтримка високої якості продукції за допомогою двох видів послуг: систем управління завданнями та збору даних про помилки.

Помилки та помилки в програмному забезпеченні пошкодили багато репутації і відповідають за величезні втрати, пов'язані з найменшим доходом і втратою часу на копання вух. Розроблено алгоритм кластеризації та класифікації тексту та введено інструмент для класифікації програмних помилок на основі графічного інтерфейсу CLUBAS.

Алгоритм CLUBAS був розроблений з використанням методу класифікації як кластеризації, при якому кластеризація спочатку виконується з використанням текстової подібності атрибутів помилок, а потім генеруються відповідні мітки та

присвоюються кожному кластеру. Мітки кластера додатково відображаються у класах помилок за допомогою мітки кластера та відповідних термінів таксономічної помилки. Алгоритм використовує два вхідні параметри; поріг подібності та кількість елементів у мітці кластера. Вплив цих параметрів на оцінку ефективності також вивчається, і графіки представлені для візуалізації їх ефектів. Алгоритм CLUBAS порівнюється з низкою стандартних алгоритмів класифікації для оцінки ефективності.

При аналізі підходів до розвитку були розглянуті три різні питання дослідження. Було виявлено, що розробники віддали перевагу алгоритму Рахмана, а не алгоритмам FixCache, хоча деякі розробники вважали, що вони безпомилкові. Список різних характеристик в ідеалі повинен мати алгоритм прогнозування помилок.

Також ці знання можуть бути використані для зміни Рахмана та створення нового алгоритму, але не спостерігали жодних змін у поведінці розробника; тим не менше, це не означає, що прогнозування помилок не допомогло розробникам, які схвильовані додаванням нового інструменту, який допоможе їм у нескінченній боротьбі за здоров'я коду, але ці результати показують, що те, що зараз є що розробляється для них ще не корисно, і, сподіваємось, це найкраще уявлення, яке вам може знадобитися, щоб передбачити помилки, які будуть корисними.

Слід застерегти, що робота стосується лише аналізу людського фактора щодо прогнозування помилок, і дослідження не аналізує придатність будь-якого алгоритму для автоматичного використання, наприклад, оптимізації тестових випадків. Також можливо, що просунуті розробники є невідповідною аудиторією для прогнозування помилок. Натомість забезпечення якості програмного забезпечення може бути кращою метою, оскільки вони використовують результати прогнозування помилок, щоб зосередити ресурси покращення якості на областях, схильних до помилок. Випробування системи проводяться шляхом її тестування. Мета цього тесту - показати, що система працює відповідно до розроблених для неї специфікацій.

При виборі послуг, які можуть допомогти в кілька разів ефективніше працювати над проектами, у вигляді цілого переліку різноманітних служб управління завданнями та часом, для управління та планування проектів, колективної роботи, побудови онлайн-схем тощо.

Тестування програмного забезпечення - це процес дослідження програмного забезпечення, який надає інформацію про якість товару, а також процес перевірки дотримання вимог до товару та фактичну функцію результату та одержання від нього. Набір тестів, підібраних певним чином. Крім того, цей етап означає оцінку системи з метою визначення відмінностей між тим, якою повинна бути система і якою вона є.

У широкому розумінні тестування є одним із методів контролю якості (контроль якості), що включає планування, проведення тестів, безпосереднє проведення тестування та аналіз отриманих результатів.

Список поступово скорочували, і варіанти, що відповідають нашим потребам, найкраще склали список найбільш корисних та ефективних додатків, таких як Jira, Slack та GanttPro та інші запропоновані програми.

Чим швидше буде виявлений дефект, тим краще. Зрозуміло, що коригування рядка в коді чи коді простіше і зрозуміліше, ніж внесення змін до готового продукту. Щоб не враховувати ситуацію, коли клієнт або кінцевий користувач виявляє дефект, тут може постраждати інформація про девелоперську компанію, і такі втрати важко підрахувати.

Якщо серйозний дефект буде виявлений на пізніх стадіях розробки, він може залишитися невиправленим, оскільки вартість внесення змін може бути занадто високою. Крім того, навіть якщо завершена програма зустрічає розділ, клієнт все одно може відмовитись прийняти його, якщо розділ був неправильним. Команда проекту могла б розробити саме те, що було описано у вимогах, але якщо вимоги були сформульовані неправильно, замовник не був би задоволений результатом.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

Додаток А

Програмний код розробленого модуля