МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Київський національний університет будівництва і архітектури

Є.Є. Шабала

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМ

Конспект лекцій
для студентів спеціальностей
123 «Комп’ютерна інженерія»
та 125 «Кібербезпека»

Київ 2024

УДК 004.415.533
Ш-12

Рецензенти: Терентьєв О.О. – д-р техн. наук, професор
Котенко А.М. – канд. техн. наук, доц., доцент

Затверджено на засіданні вченої ради факультету автоматизації і інформаційних технологій, протокол № 6 від 24 січня 2024 року.

Шабала Є.Є.

Ш-12 Тестування програмного забезпечення систем: конспект лекцій / Є.Є.Шабала – Київ: КНУБА, 2024. – 140 с.

Розглянуто основні види тестування програмного забезпечення систем. Представлені принципи побудови чек-листів, тест-кейсів, наборів тест-кейсів, визначено методи пошуку дефектів в програмному забезпеченні та документування їх в баг-репорті, використовуючи баг-трекнгові системи. Розглянуто особливості тестування інтерфейсу користувача та тестування мобільних додатків.

Призначено для студентів спеціальностей 123 "Комп’ютерна інженерія» та 125 «Кібербезпека».

УДК 004.415.533

ЗМІСТ

ТЕМА 1. ІСТОРІЯ ТЕСТУВАННЯ

Тестування програмного забезпечення - процес аналізу програмного засобу та супутньої документації з метою виявлення дефектів і підвищення якості продукту.

У 50-60-х роках минулого століття процес тестування був гранично формалізований, відділений від процесу безпосередньої розробки ПО і «математизований». Фактично тестування представляло собою скоріше налагодження програм (Debugging). Існувала концепція «Вичерпного тестування (Exhaustive testing)»- перевірки всіх можливих шляхів виконання коду з усіма можливими вхідними даними. Але дуже скоро було з'ясовано, що вичерпне тестування неможливо, тому що кількість можливих шляхів і вхідних даних дуже велике, а також при такому підході складно знайти проблеми в документації.

У 70-х роках фактично народилися дві фундаментальні ідеї тестування: тестування спочатку розглядалося як процес доказу працездатності програми в деяких заданих умовах (positive testing), а потім- строго навпаки: як процес доказу непрацездатності програми в деяких заданих умовах (negative testing). Це внутрішня суперечність не тільки не зникла з часом, але і в наші дні багатьма авторами зовсім справедливо відзначається як дві взаємодоповнюючі мети тестування.

«Процес доказу непрацездатності програми» цінується трохи більше, тому що не дозволяє закривати очі на виявлені проблеми.

Отже, ще раз найважливіше, що тестування «набуло» в 70-і роки:

• тестування дозволяє упевнитися, що програма відповідає вимогам;

• тестування дозволяє визначити умови, при яких програма веде себе некоректно.

У 80-х роках відбулося ключова зміна місця тестування в розробці ПЗ: замість однієї з фінальних стадій створення проекту тестування стало застосовуватися протягом усього циклу розробки (software lifecycle), що дозволило в дуже багатьох випадках не тільки швидко виявляти й усувати проблеми, але навіть передбачати і запобігати їх появі.

В цей же період часу відзначено бурхливий розвиток і формалізація методологій тестування і поява перших елементарних спроб автоматизувати тестування.

У 90-х роках відбувся перехід від тестування як такого до більш всеосяжного процесу, який називається «забезпечення якості (quality assurance)», охоп-

лює весь цикл розробки ПЗ і зачіпає процеси планування, проектування, створення і виконання тест-кейсів, підтримку наявних тест-кейсів і тестових оточень. Тестування вийшло на якісно новий рівень, який природним чином привів до подальшого розвитку методологій, появи досить потужних інструментів управління процесом тестування і інструментальних засобів автоматизації тестування, вже цілком схожих на своїх нинішніх нащадків.

«Процес доказу непрацездатності програми» цінується трохи більше, тому що не дозволяє закривати очі на виявлені проблеми.

Отже, ще раз найважливіше, що тестування «набуло» в 70-і роки:

• тестування дозволяє упевнитися, що програма відповідає вимогам;

• тестування дозволяє визначити умови, при яких програма веде себе некоректно.

У 80-х роках відбулося ключова зміна місця тестування в розробці ПЗ: замість однієї з фінальних стадій створення проекту тестування стало застосовуватися протягом усього циклу розробки (software lifecycle), що дозволило в дуже багатьох випадках не тільки швидко виявляти й усувати проблеми, але навіть передбачати і запобігати їх появі.

В цей же період часу відзначено бурхливий розвиток і формалізація методологій тестування і поява перших елементарних спроб автоматизувати тестування.

У 90-х роках відбувся перехід від тестування як такого до більш всеосяжного процесу, який називається «забезпечення якості (quality assurance)», охоплює весь цикл розробки ПЗ і зачіпає процеси планування, проектування, створення і виконання тест-кейсів, підтримку наявних тест-кейсів і тестових оточень. Тестування вийшло на якісно новий рівень, який природним чином привів до подальшого розвитку методологій, появи досить потужних інструментів управління процесом тестування і інструментальних засобів автоматизації тестування, вже цілком схожих на своїх нинішніх нащадків.

У нульові роки нинішнього століття розвиток тестування тривало в контексті пошуку все нових і нових шляхів, методологій, технік і підходів до забезпечення якості. Серйозний вплив на розуміння тестування мала поява гнучких методологій розробки та таких підходів, як «розробка під керуванням тестування (Test-driven development, TDD) ».

Автоматизація тестування вже сприймалася як звичайна невід'ємна частина більшості проектів, а також стали популярні ідеї про те, що на основу процесу тестування слід ставити не відповідність програми вимогам, а її здатність надати

кінцевому користувачеві можливість ефективно вирішувати свої завдання.

Тестувальник, або QA (Quality Assurance) інженер, - це фахівець, який відповідає за перевірку якості програмного забезпечення. Роль тестувальника полягає в тому, щоб виявити помилки, дефекти, та аномалії в програмах, перед тим як вони потраплять до кінцевих користувачів. Основна мета тестування - забезпечити високу якість програми та впевнитися, що вона виконує свої функції правильно та надійно.

Обов'язки тестувальника можуть включати такі дії:

1. Розробка тестових планів та сценаріїв: Специфікація, яка описує, яким чином програма повинна працювати, слугує основою для розробки тестових планів та сценаріїв. Тестувальник розробляє ці документи, щоб перевірити, чи програма відповідає цим вимогам.
2. Виконання тестів: Тестувальник виконує тестові сценарії, використовуючи програму та спостерігаючи за її реакцією. Він реєструє результати тестів та документує виявлені дефекти.
3. Виявлення та документування дефектів: Тестувальник виявляє будь-які аномалії, помилки чи некоректну поведінку програми та документує їх. Важливо надати якомога більше інформації для розробників для виправлення помилок.
4. Регресійне тестування: Після виправлення дефектів тестувальник перевіряє, чи ці виправлення не вплинули на роботу інших частин програми.
5. Тестування продуктивності та навантаження: Виконання тестів, які оцінюють продуктивність та навантаження програми, щоб визначити, як вона працює в умовах великої кількості користувачів або обсягу даних.
6. Автоматизація тестування: Розробка та виконання автоматизованих тестів, які допомагають автоматично перевіряти функціональність програми за допомогою спеціалізованих інструментів.
7. Підтримка випуску продукту: Після завершення тестування та виправлення дефектів, тестувальник може відзначити, що програма готова до випуску або виправити помилки, які виявилися під час експлуатації користувачами.

Тестувальники грають ключову роль у забезпеченні високої якості програмного забезпечення, допомагаючи ідентифікувати та усувати дефекти перед їхнім потраплянням до кінцевих користувачів.

Вимоги до тестувальника можуть змінюватися в залежності від конкретної компанії, проекту та ролі, яку він має виконувати. Однак існують деякі загальні

вимоги та навички, які зазвичай вимагаються від тестувальників:

1. Знання тестування програмного забезпечення: Тестувальник повинен мати розуміння основних принципів та методів тестування ПЗ, включаючи тестові процеси, техніки тестування, тестову документацію тощо.

2. Технічні навички: Тестувальник повинен бути технічно компетентним, здатним розуміти архітектуру програмного забезпечення та взаємодію компонентів.

3. Аналітичні навички: Здатність аналізувати вимоги, документацію та функціональність програми для розробки ефективних тестових сценаріїв.

4. Навички тестування на різних рівнях: Розуміння та досвід у проведенні різних видів тестування, таких як функціональне, інтеграційне, системне, регресійне, навантаження тощо.

5. Здатність створення тестових сценаріїв та тест-кейсів: Вміння розробляти ефективні тестові сценарії та тест-кейси на основі вимог та специфікацій.

6. Навички автоматизації тестування: Досвід у розробці та виконанні автоматизованих тестів за допомогою інструментів для автоматизації тестування.

7. Знання процесів розробки ПЗ: Розуміння циклу розробки програмного забезпечення та взаємодії з розробниками.

8. Комунікативні навички: Здатність ефективно спілкуватися та співпрацювати з розробниками, керівництвом та іншими членами команди.

9. Тестова документація: Знання створення та обслуговування тестової документації, включаючи тест-плани, звіти про тестування та трекінг дефектів.

10. Здатність до аналізу та відкритий мислення: Тестувальник повинен бути здатним розуміти потенційні ризики та проблеми в програмі та виявляти помилки, які інші можуть пропустити.

11. Спеціалізовані навички: Знання та досвід у певних галузях, таких як тестування безпеки, мобільного тестування, тестування вбудованого програмного забезпечення тощо, можуть бути вимаганими в залежності від проекту.

12. Спроможність працювати в команді: Тестувальник повинен бути здатним працювати в команді та співпрацювати з іншими членами розробницького колективу.

Загальна вимога до тестувальника полягає в тому, щоб він був спроможним виявити проблеми та допомогти забезпечити високу якість програмного забезпечення, яке відповідає вимогам та очікуванням користувачів.

Міфи і помилки про тестування

1. Не треба розбиратися в комп'ютерах

2. Обов'язково треба добре знати програмування

3. У тестуванні все просто

4. У тестуванні купа рутини і нудьги

5. Тестувальника повинні всьому-всьому навчити

6. У тестувальники йдуть ті, хто не зміг стати програмістом

7. У тестуванні складно побудувати кар'єру

8. Тестувальник «винен у всьому», тобто з нього попит за все помилки

9. Тестировщики скоро будуть не потрібні, тому що все буде автоматизовано.

Зростання вартості виправлення дефектів в процесі розробки програми[1]:

Рис. 1.1. Зростання вартості виправлення дефектів в процесі розробки програми

ТЕМА 2. ГНУЧКЕ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. AGILE І SCRUM

Agile – це гнучкий підхід до організації роботи та система установок. Даний підхід зосереджений на особистій відповідальності, тісному контакті та командній роботі. Гнучке розроблення за принципом Agile найчастіше використовується, щоб покращувати клієнтський досвід, швидше постачати продукт на ринок, скорочувати кількість нераціональних і неефективних дій та підвищити якість ПП в цілому. Особливо ефективним є застосування Agile при роботі в умовах високої невизначеності.

Основні ідеї Agile:

• люди і їх взаємодія є важливішими за процеси та інструменти;

• працюючий ПП є важливішим за вичерпну документацію;

• співпраця з замовником є важливішою за узгодження умов контракту;

• готовність до змін є важливішою за дотримання початкового плану.

Принцип взаємодії має на увазі, що команда взаємодіє з замовником і, навпаки, замовник з командою. Такий підхід дозволяє обмінюватися досвідом між замовником та учасниками команди розроблення, а також кожному з них брати участь у процесі прийняття рішень. Як наслідок, зниження ризиків втрати коштів і часу, а також підвищення здатності команди до вирішення нестандартних складних завдань з високим рівнем невизначеності.

Недоліки Agile випливають з його переваг. Так взаємодія всіх з усіма може призвести до хаотичного процесу розроблення, що впливає на всі етапи.

Саме тому, використовуючи підходи Agile, потрібно зважати на певні обмеження: команди розроблення повинні бути невеликими, учасники повинні бути мотивованими та (що більш важливо) компетентними, має бути встановлено чіткі обмеження за часом, кожна ітерація має бути короткою з максимально зрозумілими цілями, а кінцевий результат повинен бути очевидним.

Agile прогнозує результат на більш короткий період у порівнянні зі стандартними стратегіями розроблення, тому дуже добре вирішує проблему невизначеності. Для підвищення ефективності використовується наступне правило: чим вище невизначеність, тим коротшою має бути ітерація.

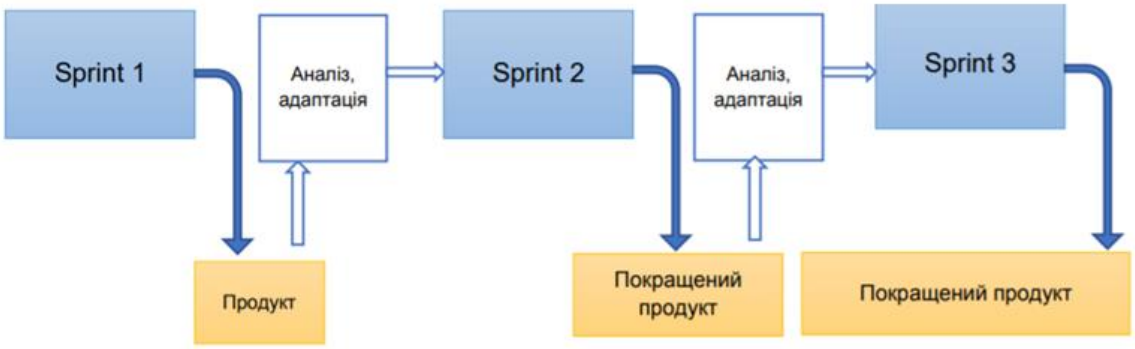

Рис. 2.1. Покроковий, ітераційний процес розробки в Scrum

Варто зазначити, що дуже часто її тривалість може бути меншою (або рівною) 24 годин. У такому випадку на початку кожної ітерації обов'язково необхідно виконати аналіз готової роботи та запланувати наступну ітерацію[2].

Різниця між Agile і Scrum

Agile більш ширше поняття. Agile — це методологія, скоріше навіть філософія зі своїм набором цінностей, котра впливає на поведінку людини і до якої відноситься Scrum. Аджайл придумали для того щоб встигати за змінами на ринку.

Scrum володіє своїм власним набором цінностей і принципів і забезпечує легку «структуру», яка на практиці показує, як застосувати філософію Аджайла, щоб домогтися результату. Перефразуємо Agile — це практика, а Scrum — це процес дотримання цієї практики.

Agile: У Agile кожна ітерація прив'язана до SDLC (циклом розробки програмного забезпечення) включаючи планування, проектування, кодування, аналіз вимог, модульне тестування та приймальне тестування, коли продукт демонструється зацікавленим особам.

Scrum: а у Scrum спринт є основною одиницею розробки. За кожним спринтом йде планова зустріч, де визначаються і оцінюються завдання для спринта (identified and estimated). Під час кожного спринту команда створює готову частину продукту, а по завершенні спринту відбувається доставка.

Scrum

Scrum («штовханина») - методологія управління проектами, яка застосовується при розробці інформаційних систем, а також для гнучкої розробки програмного забезпечення.

Scrum це методологія яка застосовує різноманітні технологічні прийоми і процеси розробки інформаційних продуктів яка дозволяє підвищити ефективність управління розробкою продукту.

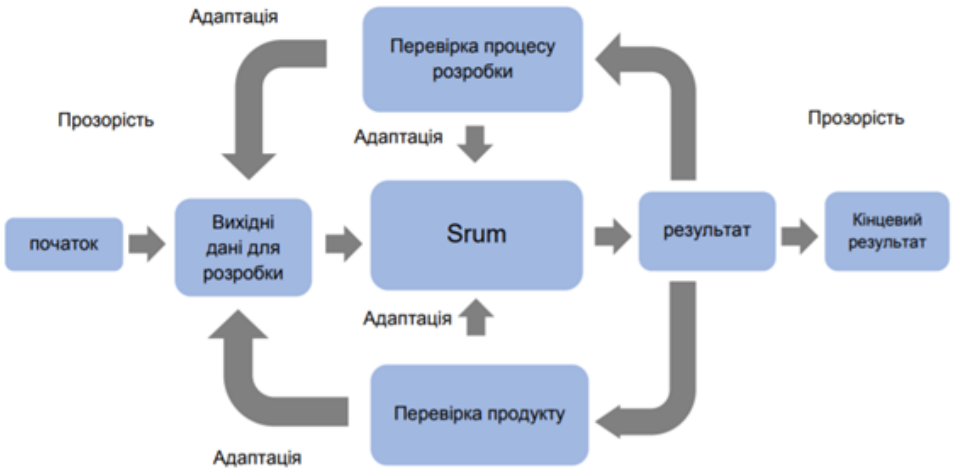

Рис. 2.2. Модель процесів Scrum

Скрам (Scrum) це простий фреймворк який допомагає людям, командам та організаціям генерувати цінність через адаптивні рішення складних проблем. В цілому, Скрам вимагає від Скрам Мастера (Scrum Master) створення середовища в якому:

1. Власник Продукту (Product Owner) додає у певному порядку роботу для вирішення складної проблеми у Беклог Продукту (Product Backlog).

2. Скрам Команда (Scrum Team) перетворює вибірку робіт в Інкремент (Increment), який додає цінність продукту, протягом Спринта (Sprint).

3. Скрам Команда (Scrum Team) та зацікавлені сторони інспектують результати роботи та коректують наступний Спринт.

4. Повторення.

У Скрам фреймворку можна застосовувати різні процеси, техніки та методи. Скрам поширюється навко-ло існуючих практик або робить їх взагалі непотрібними. Скрам робить видимою відносну ефективність поточного управління, навколишнього середовища та технік роботи, таким чином щоб їх можна було вдосконалити.

Теорія Скрам

Скрам ґрунтується на емпіризмі та ощадливому мисленні. Емпіризм стверджує, що знання приходить з досвідом, а прийняття рішень має відбуватись на підставі того, що вже є відомим. Ощадливе мислення зменшує витрати та зосереджується на найнеобхіднішому. Скрам використовує ітеративний, поступовий підхід, щоб покращити прогнозування та контроль ризиків. Скрам залучає групи людей, які в сукупності володіють усіма навичками та досвідом, щоб виконувати роботу, обмінюватися досвідом та набувати необхідних навичок. Скрам поєднує чотири формальні події для перевірки та адаптації у рамках Спринта (Sprint).

Ці події працюють, оскільки вони впроваджують емпіричні опори на яких базується Скрам: прозорість, перевірка та адаптація.

Прозорість. Робочий процес та результати роботи мають бути видимими, як для виконавців цієї роботи, так і для тих, хто отримує її результати. Тому у Скрамі важливі рішення базуються висновках зроблений після оцінки трьох формальних артефактів Скрам.

Артефакти з низькою "прозорістю" можуть призвести до рішень, що зменшують набуту цінність та збільшують ризики. Прозорість надає можливість для перевірки. Перевірка без прозорості є оманливою та непотрібною.

Перевірка. Аби виявити потенційно небажані розбіжності або проблеми, потрібно часто і ретельно перевіряти артефакти Скрам та прогрес у досягненні узгоджених цілей. Для того, щоб допомогти з перевіркою, Скрам забезпечує циклічну послідовніть п'яти своїх подій. Перевірка надає змогу адаптуватись. Перевірка без адаптування немає змісту. Події Скрам розроблені, щоб спровокувати зміни.

Адаптація. У випадку коли будь-які аспекти процесу відхиляються від допустимих меж, або якщо отриманий продукт є неприйнятним, потрібно скорегувати застосовуваний процес або продукт, що виробляється.

Коригування потрібно зробити якомога швидше, щоб мінімізувати подальші відхилення.

Адаптація ускладняється, коли залучені люди не мають повноважень або не управляють собою. Очікується, що команда Скрам адаптується до того моменту, коли вона дізнається щось нове через перевірку.

Цінності Скрам

Успішне використання Скрам залежить від того, наскільки вміло люди втілюють наступні п'ять принципів: Почуття обов'язку, зосередженість, відкритість, повага та сміливість (Commitment, Focus, Openness, Respect, and Courage)

Скрам Команда (Scrum Team) зобов'язується досягати своїх цілей та підтримувати один одного. Їх основна увага зосереджена на роботі Спринта (Sprint), щоб досягти якнайкращого прогресу до цих цілей. Скрам Команда та її зацікавлені сторони відкрито ставляться до роботи та проблем. Члени Скрам Команди поважають один одного та почуваються самодостатніми та незалежними, і тим 5 самим мають таке ж ставлення від людей з якими працюють. Члени Скрам Команди беруть на себе сміливість зробити правильну річ, працюючи над складними проблемами.

Ці принципи задають Скрам Команді напрямок їх роботі, дій та поведінки.

Прийняті рішення, вжиті кроки та спосіб використання Скраму повинні підсилювати ці принципи, а не применшувати або підривати їх.

Члени Скрам Команди вивчають та досліджують ці цінності під час роботи з подіями, ролями та артефактами Скраму. Коли ці принципи втілюються Скрам Командою та людьми з якими вони працюють, емпіричні стовпи прозорості Скраму, перевірки та адаптації оживають, зміцнюючи довіру.

Скрам Команда

Основна одиниця Скраму — це невелика команда людей яку називають Скрам Командою (Scrum Team). Скрам Команда складається з Скрам Мастера (Scrum Master), Власника Продукту (Product Owner), і Розробників (Developers).

У Скрам Команди немає підгруп або ієрархій. Це згуртована група професіоналів, орієнтованих на одну ціль, Ціль Продукту (Product Goal). Скрам Команди є крос-функціональними, тобто члени команди мають усі навички які необхідні для створення цінності кожного Спринту. Вони також є самокерованими, а це означає що вони внутрішньо вирішують хто що, коли і як робить.

Скрам Команда досить мала, щоб залишатися спритною і достатньо велика, щоб виконати значну роботу в Спринті, і як правило складається з десяти або менше людей.

Якщо Скрам Команда стає занадто великою, то варто розглянути можливість реорганізації в кілька згуртованих Скрам Команд, кожна з яких зосереджена на одному продукті. Тому вони повинні мати однакову Ціль Продукту, Беклог Продукту та Власника Продукту.

Скрам Команда несе відповідальність за все пов'язане з продуктом — від співпраці із зацікавленими сторонами, верифікацією, технічним обслуговуванням, експлуатацією, експериментами, дослідження і розробками та всього іншого, що може бути необхідним. Вони структуровані та уповноважені організацією керувати власною роботою. Коли люди працюють в Спринті у стійкому темпі, це покращує фокусування і послідовність Скрам Команди. Вся Скрам Команда несе відповідальність за створення цінного, корисного приросту кожного Спринту.

Скрам визначає три конкретні сфери відповідальності в рамках Скрам Команди: Розробники, Власник Продукту і Скрам Мастер. Розробники Розробники (Developers) — це люди в Скрам Команді які прагнуть створити будь-який аспект корисного інкременту кожного Спринта.

Конкретні навички які необхідні Розробникам, часто є широкими і можуть відрізнятися залежно від сфери роботи.

Однак Розробники завжди несуть відповідальність за наступне:

● Створення плану для Спринта - Спринт Беклог (Sprint Backlog);

● Інтегрування якості, шляхом дотримання Визначення Виконаної Роботи (Definition of Done);

● Кожного дня пристосовувати свій план до Цілі Спринту (Sprint Goal);

● Звітують один перед одним як професіонали.

Власник Продукту Власник Продукту (Product Owner) відповідає за досягнення максимальної якості продукту, який є результатом роботи Скрам Команди. Способи, за допомогою яких цього досягають, можуть відрі-знятися залежно від організацій, Скрам Команд та окремих осіб.

Власник Продукту також відповідає за ефективне управління Беклогом Продукту (Product Backlog), що включає наступне:

● Розробити та точно прокомунікувати Цілі Продукту (Product Goal);

● Створити та чітко прокомунікувати елементи Беклогу Продукту;

● Впорядкувати елементи Беклог Продукту;

● Переконатись, що Беклог Продукту є прозорим, доступним і зрозумілим.

Журнал продукту

ID	Назва	Важливість	Попередня оцінка трудомісткості	Як продемонструвати	Примітки
1	Перегляд журналу особистих транзакцій	10	8	Увійти в систему; перейти на сторінку транзакцій; покласти гроші на рахунок; повернутися на сторінку транзакцій; перевірити, що нова транзакція з'явилася в списку.	Щоб уникнути великих запитів до бази даних, варто скористатися посторінковим виведенням інформації. Дизайн такий же, як і у сторінки перегляду користувачів.
2	...	....	....	....	....

Рис. 2.3. Журнал продукту

Власник Продукту може виконувати перераховані вище функції сам, або ж довірити їх виконання іншим членам команди, однак відповідальність за них несе сам Власник Продукту. Щоб Власник Продукту успішно виконував свої обов'язки, всі члени організації повинні поважати його рішення. Усі рішення Власника Продукту відображаються у вмісті та впорядкуванні Беклогу Продукту, а також у перегляді Інкременту під час Рев'ю Спринту.

Власник Продукту — це одна людина, а не група людей. Власник Продукту може представляти потреби багатьох зацікавлених сторін у Беклозі Продукту. А ті, хто бажає змінити пріоритетність вимог у Беклозі Продукту, повинні звернутись до Власника Продукту та переконати його.

Скрам Мастер Скрам Мастер (Scrum Master) відповідальний за дотримання Скраму саме таким чином, як визначено у Посібнику зі Скраму. Він допомагає, як членам Скрам Команди, так і усій організації, зрозуміти теоретичні засади і практики Скраму.

Скрам Мастер відповідає за ефективніть Скрам Команди. Він робить це, дозволяючи Скрам Команді вдосконалювати свої практики в рамках Скраму. Скрам Мастери є справжніми лідерами, які служать Скрам Команді та усій організації.

Події в Скрам Спринт (Sprint) включає в себе всі необхідні наради. Кожна нарада в Скрамі надає можливість формально перевірити та адаптувати Скрам артефакти. Такі події створені спеціально для того, щоб забезпечити необхідну прозорість. Відмова від проведення однієї з нарад призводить до втрати можливостей для перевірки та адаптації.

У Скрамі проводять чітко визначені типи нарад що дозволяє систематизувати процес розробки та звести до мінімуму потребу в нарадах, не передбачених Скрамом. Для легкості роботи команди, найкраще коли всі наради проводять в один і той же час та в одному місці.

Спринт

Спринти (Sprints) є основою Скраму, адже саме в них прості ідеї перетворюються на цінність. Це події з фіксованою тривалістю близько одного місяця чи менше. Новий Спринт починається одразу після завершення попереднього Спринту. Вся робота, необхідна для досягнення

Цілі Продукту (Product Goal), включаючи Планування Спринту (Sprint Planning), Щоденні Скрами (Daily Scrums), Рев'ю Спринту (Sprint Review) та Спринт Ретроспективу (Sprint Retrospective), відбувається під час Спринтів.

Під час Спринту:

● Не допускається внесення жодних змін, які б ставили під загрозу досягнення Цілі Спринту;

● Вимоги до якості продукту залишаються незмінними;

● Беклог Продукту (Product Backlog) уточнюється за необхідності;

● Об’єм роботи можна уточнити та повторно обговорити з Власником Продукту під час процесу розробки.

Спринти вносять прогнозованість у процес розробки, забезпечуючи проведення перевірки та адаптації на шляху до Цілі Продукту (Product Goal), як мінімум, раз на місяць. Якщо часові рамки Спринту є занадто довгими, то Ціль Спринта може стати неактуальною, складність завдання може зрости, а ризики

збільшитися.

Короткі Спринти можна використовувати для створення більших навчальних циклів та для того щоб зменшити в часі ризик витрат і зусиль. Кожен Спринт можна вважати коротким проектом.

Існують різні практики прогнозування прогресу, такі як графіки типу “скільки залишилось” (burn- downs), “скільки зроблено” (burn-ups) або кумулятивні діаграми (cumulative flows). Хоча вони є корисними, проте не можуть замінити важливість емпіричного підходу. В складних умовах те, що станеться - невідомо. Тільки з огляду на те, що вже сталося, можемо приймати перспективні рішення.

Спринт можна скасувати, якщо Ціль Спринту стає неактуальною. Лише Власник Продукту має право скасувати Спринт. Планування Спринту Планування Спринту (Sprint Planning) ініціює Спринт, та визначає саме ту роботу, яку слід виконати за цей Спринт. У результаті ми отримуємо план, до створення якого була залучена вся Скрам Команда.

Власник Продукту (Product Owner) повинен переконатись, що учасники готові обговорити найважливіші елементи Беклогу Продукту та визначити чи ці елементи відповідають Цілі Продукту (Product Goal).

Скрам Команда може також запросити інших спеціалістів взяти участь у Плануванні Спринту щоб мати змогу з ними проконсультуватись.

Щоденний Скрам

Метою Щоденного Скраму (Daily Scrum) є перевірка прогресу досягнення Цілі Спринту та, якщо виникла така потреба, адаптація Беклогу Спринту, щоб відкоригувати заплановану роботу.

Щоденний Скрам — це 15-хвилинна нарада для Розробників Скрам Команди. Для полегшення співпраці команди, нараду проводять в один і той же час та місці кожного дня Спринту. У випадку, коли Власник Продукту або Скрам Мастер активно працюють над елементами з Беклогу Спринту, вони беруть участь в команді у якості Розробників.

Рев'ю Спринту

Метою Рев'ю Спринту (Sprint Review) є перевірка результатів Спринту та визначення адаптацій в подальшій роботі. Скрам Команда представляє результати своєї роботи зацікавленим особам та обговорює прогрес по досягненню Цілі Продукту (Product Goal).

Протягом наради Скрам Команда та зацікавлені особи переглядають, що було досягнуто у Спринті та що змінилося в їхньому середовищі. На основі цієї інформації всі присутні співпрацюють над тим, що робити далі.

Беклог Продукту (Product Backlog) також можна скоригувати для того, щоб він відпові-дав новим можливостям. Рев'ю Спринту (Sprint Review) — це робоча сесія, тому Скрам Команда не повинна обмежувати себе лише презентацією виконаної роботи. Рев'ю Спринту є передостанньою нарадою у Спринті . Ця нарада також обмежена в часі до чотирьох годин максимум для Спринту, що триває один місяць. Для коротших Спринтів ця нарада зазвичай коротша.

Ретроспектива Спринту

Метою Ретроспективи Спринту (Sprint Retrospective) є планування способів підвищення якості та ефективності роботи. Скрам Команда перевіряє, як пройшов останній Спринт беручи до уваги членів команди, взаємодії, процесів, інструментів та визначення "виконаної" роботи.

Розглянуті елементи часто змінюються в залежності від сфери роботи. Також, Скрам Команда за допомогою припущень визначає причини, які заважають команді виконувати роботу, та досліджує їх походження. Скрам Команда Обговорює що пройшло добре під час Спринту, з якими проблемами команда зіткнулася та як ці проблеми були (або не були) вирішені.

Скрам Команда визначає найбільш корисні зміни для підвищення своєї ефективності. Ці найбільш ефективні покращення розглядаються якомога швидше. Вони навіть можуть бути додані до Беклогу наступного Спринта.

Ретроспектива Спринта (Sprint Retrospective) завершує Спринт. Ця нарада обмежена в часі до трьох годин максимум для Спринту, що триває місяць.

Для коротших Спринтів ця нарада зазвичай коротша.

Артефакти Скраму

Артефакти Скраму (Scrum Artifacts) представляють собою роботу або цінність роботи. Вони спеціально спроектовані таким чином, щоб забезпечити максимальну чіткість ключової інформації. Тому кожен, хто їх переглядає, має однакову основу для адаптації. Ко-жен артефакт включає в себе зобов'язання забезпечити надання інформації, що підвищує прозорість та фокус, і на основі яких можна виміряти прогрес:

- Для Беклогу Продукту (Product Backlog) це Ціль Продукту (Product Goal).
- Для Беклогу Спринта (Sprint Backlog) це Ціль Спринта (Sprint Goal).
- Для Інкременту (Increment) це Визначення "Виконаної" Роботи (Definition of Done).

Ці зобов'язання існують для того, щоб посилити емпіризм та цінності Скраму для Скрам Команди та їхніх зацікавлених осіб [3].

ТЕМА 3. МОДЕЛІ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Поняття життєвого циклу програмного забезпечення (ПЗ) з'явилося, коли програмістське співтовариство усвідомило необхідність переходу від кустарних ремісничих методів розробки програм до технологічного промислового їх виробництва.

Як зазвичай відбувається в подібних ситуаціях, програмісти спробували перенести досвід інших індустріальних виробництв в свою сферу. Зокрема, було запозичене поняття життєвого циклу.

Життєвий цикл програмного забезпечення - період часу, що починається з моменту прийняття рішення про необхідність створення програмного продукту і закінчується в момент його повного вилучення з експлуатації.

Цей цикл - процес побудови і розвитку ПЗ. Поняття ЖЦ виникло під впливом потреби у систематизації робіт у процесі розроблення ПЗ.

Систематизація була першим етапом на шляху до автоматизації процесу розроблення ПЗ. Наступними кроками переходу до автоматизації процесу розроблення ПЗ були такі: встановлення технологічних маршрутів діяльності розробників ПЗ, визначення можливості їх автоматизації та виявлення ризиків, розроблення інструментів для автоматизації.

Використання поняття життєвого циклу дозволяє обрати підходи, які найбільш ефективні для завдань певного етапу життя ПЗ. Залежно від особливостей процесів розроблення та супроводу програм існують різні моделі ЖЦ.

Використання певної моделі ЖЦ дозволяє визначитися з основними моментами процесу замовлення, розроблення та супроводу ПЗ навіть недосвідченому програмісту.

Також використання моделей дозволяє чітко зрозуміти, в який період переходити від версії до версії, які дії з удосконалення виконувати, на якому етапі. Знання про закономірності розвитку програмного продукту, які відбиваються в обраній моделі ЖЦ, дозволяють отримати надійні орієнтири для планування процесу розроблення та супроводу ПЗ, економно витрачати ресурси та підвищувати якість управління усіма процесами.

Також моделі життєвого циклу є основою знань технологій програмування та інструментарію, що їх підтримує. Будь-що, технологія базується на певних уявленнях про життєвий цикл та організує свої методита інструменти навколо фаз та етапів ЖЦ.

Розвиток методологій програмування у 70-х рр. XX ст. привів доформу-

вання потреби вивчення життєвого циклу ПЗ. До цього часу моделі ЖЦ розвиваються і модифікуються, уточнюючи та доповнюючи дві базові моделі – каскадну та ітеративну. Ці зміни обумовлені потребою організаційної та технологічної підтримки проектів з розроблення ПЗ [4].

Модель розробки ПЗ (Software Development Model, SDM) - структура, яка систематизує різні види проектної діяльності, їх взаємодію і послідовність в процесі розробки ПЗ.

Вибір тієї або іншої моделі залежить від масштабу і складності проекту, предметної області, доступних ресурсів і безлічі інших чинників.

Моделей розробки ПЗ багато, але в загальному випадку класичними можна вважати *водоспадну, v-подібну, ітераційну інкрементальну, спіральну і гнучку*.

Ідеальної моделі не існує.

Водоспадна модель (waterfall model)

Водоспадна модель (waterfall model) зараз представляє швидше історичний інтерес, тому що в сучасних проектах практично непридатна. Вона передбачає одноразове виконання кожної з фаз проекту, які, в свою чергу, суворо слідують один за одним.

Дуже спрощено можна сказати, що в рамках цієї моделі в будь-який момент часу команді «відома» лише попередня і наступна фаза. У реальному ж розробці ПО доводиться «бачити весь проект цілком» і повертатися до попередніх фаз, щоб виправити недоробки або щось уточнити.

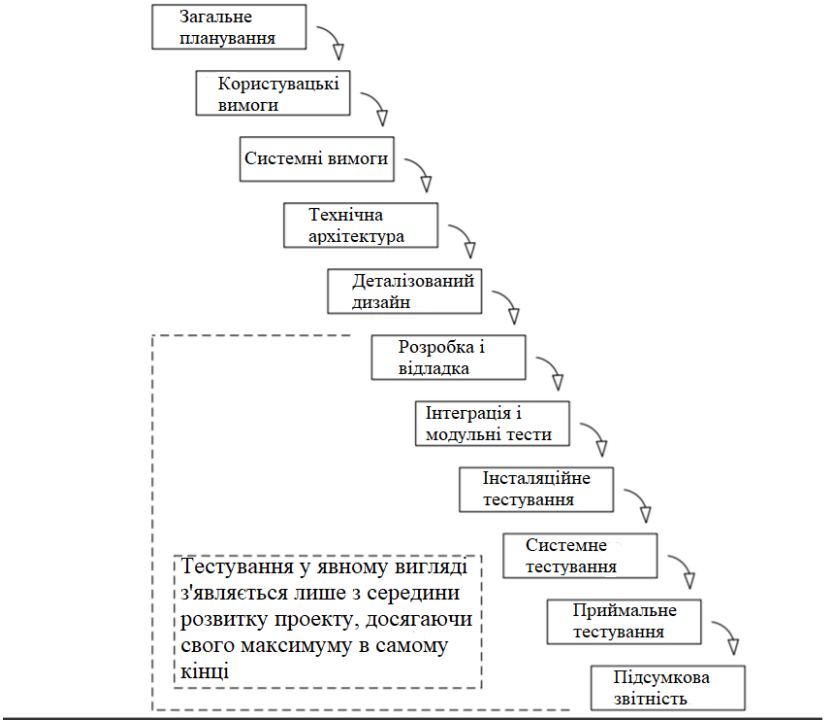

Рис. 3.1. Waterfall model розробки програмного забезпечення

Застосування waterfall model

- Прості завдання
- Відправний пункт для створення нових моделей
- Для великих проектів, в яких вимоги дуже стабільні і можуть бути добре сформульовані на початку проекту (аерокосмічна галузь, медичне ПЗ і т.д.)

V-образна модель (V-model)

V-образна модель є логічним розвитком Водоспадної. Можна помітити, що в загальному випадку як водоспадна, так і vобразна моделі життєвого циклу ПЗ можуть містити один і той же набір стадій, але принципова відмінність полягає в тому, як ця інформація використовується в процесі реалізації проекту.

Дуже спрощено можна сказати, що при використанні v-образної моделі на кожній стадії «на спуску» потрібно думати про те, що і як буде відбуватися на відповідній стадії «на підйомі». Тестування тут з'являється вже на самих ранніх стадіях розвитку проекту, що дозволяє мінімізувати ризики, а також виявити і усунути безліч потенційних проблем до того, як вони стануть проблемами реальними.

Рис. 3.2. V-Model розробки прогамного забезпечення

Застосування V-Model

V-Model можна використовувати, в таких випадках:

- Вимоги добре визначені, задокументовані і зафіксовані.
- Формулювання продукту стабільне.
- Технології не динамічні і проектна команда їх добре знає.
- Немає двозначних чи невизначених вимог.
- Проект не тривалий.

Використання: V-model також використовується у медичних чи державних проектах.

Ітераційна Інкрементальна модель (iterative model, incremental model) є фундаментальною основою сучасного підходу до розробки ПЗ. Як випливає з назви моделі, їй властива певна подвійність:

• з точки зору життєвого циклу модель є ітераційною, тому що має на увазі багаторазове повторення одних і тих же стадій;

• з точки зору розвитку продукту (збільшення його корисних функцій) модель є інкрементальною.

• Ключовою особливістю даної моделі є розбиття проекту на відносно невеликі проміжки (ітерації), кожен з яких в загальному випадку може включати в себе всі класичні стадії, властиві Водоспадній і v-образній моделям.

• Результатом ітерації є приріст (інкремент) функціональності продукту, виражене в проміжному білді (build).

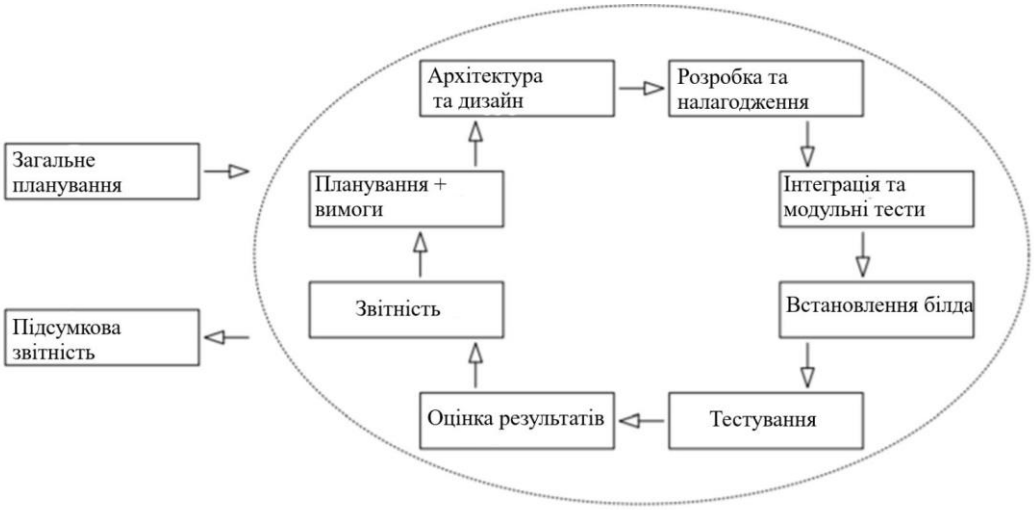

Рис. 3.3. Ітераційна Інкрементальна модель розробки ПЗ

• Довжина ітерацій може змінюватися в залежності від багатьох факторів, однак сам принцип багаторазового повторення дозволяє гарантувати, що і тестування, і демонстрація продукту кінцевому замовнику (з отриманням зворотного зв'язку) буде активно застосовуватися з самого початку і протягом усього часу розробки проекту.

• Ітераційна Інкрементальна модель дуже добре зарекомендувала себе на об'ємних і складних проектах, які виконуються великими командами протягом тривалих термінів. Однак до основних недоліків цієї моделі часто відносять високі накладні витрати, викликані високою «бюрократизацією» і загальної громіздкістю моделі.

Спіральна модель (spiral model)

• **Спіральна модель (spiral model)** являє собою окремий випадок ітераційної інкрементальної моделі, в якому особлива увага приділяється управління ризиками, особливо впливають на організацію процесу розробки проекту і контрольні точки.

Чотири ключові фази:

- опрацювання цілей, альтернатив і обмежень;
- аналіз ризиків і прототипування;
- розробка (проміжної версії) продукту;
- планування наступного циклу.

Спіральна модель являє собою окремий випадок ітераційної інкрементальної моделі, в якому особлива увага приділяється управління ризиками, особливо впливають на організацію процесу розробки проекту і контрольні точки.

З точки зору **тестування і управління якістю** підвищена увага ризиків є відчутною перевагою при використанні спіральної моделі для розробки концептуальних проектів, в яких вимоги є складними і нестабільними (можуть багаторазово змінюватися по ходу виконання проекту).

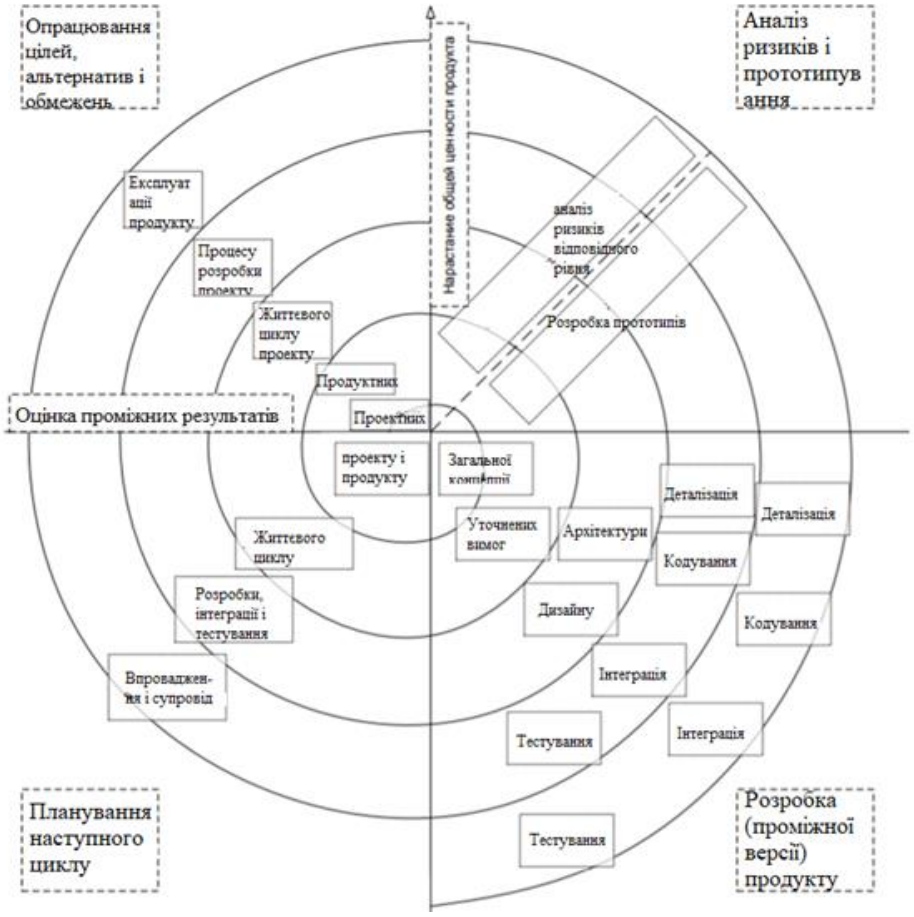

Рис. 3.4. Спіральна модель розробки програмного забезпечення

Гнучка модель (agile model)

Гнучка модель розробки програмного забезпечення, відома також як Agile (гнучка, легка, плавна), представляє собою підхід до розробки програм, який ставить акцент на ітеративність, колаборацію, і здатність адаптуватися до змін.

Основні принципи гнучкої розробки визначені в "Маніфесті гнучкої розробки програмного забезпечення" (Agile Manifesto), який був представлений групою фахівців у 2001 році.

Основні ідеї гнучкої розробки включають:

1. Ітеративність і інкрементальність: Розробка поділяється на короткі ітерації (зазвичай 2-4 тижні), під час яких розробники створюють новий функціонал або вдосконалюють існуючий. Кожна ітерація додає новий функціонал до вже існуючого продукту.

2. Співпраця замість контрактів: Значення надається спілкуванню і співпраці між розробниками, замовниками та іншими учасниками розробки програми. Комунікація вважається більш важливою, ніж створення обширної документації.

3. Реакція на зміни: Гнучка модель акцентує на змінах як необхідному елементі розробки. Зміни вимагають здатності швидко адаптуватися до нових вимог або уточнень.

4. Прототипування і визначення вимог на ранніх етапах: Клієнти та розробники активно співпрацюють на ранніх етапах розробки, щоб визначити реальні потреби та вимоги до програмного продукту.

5. Самоорганізація індивідів і команд: Заснована на ідеї того, що самостійноорганізовані команди є більш продуктивними, а самі індивіди можуть приймати відповідальність за свою роботу та приймати рішення.

6. Доставка функціоналу якнайшвидше: Головною метою є швидка і постійна доставка корисного функціоналу для клієнта.

Гнучка модель дозволяє здійснювати розробку в умовах невизначеності і частих змін у вимогах. Цей підхід добре підходить для сучасного швидкого та динамічного середовища розробки програмного забезпечення.

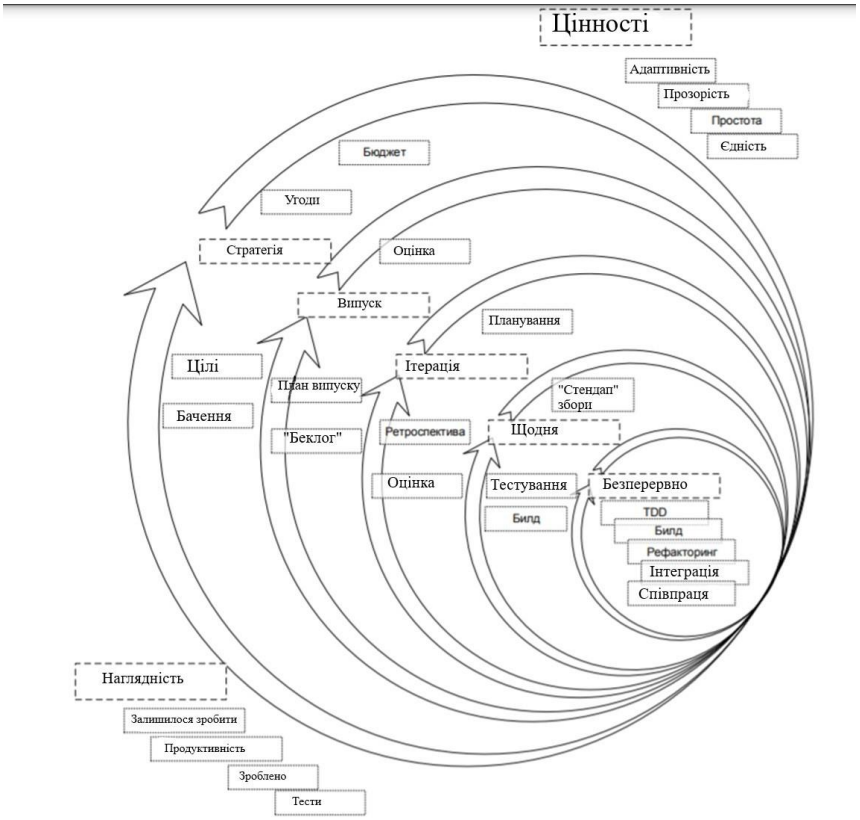

Рис. 3.5. Гнучка модель розробки програмного забезпечення

Життєвий цикл тестування

Життєвий цикл тестування програмного забезпечення — це всі дії, що виконуються під час тестування програмного продукту. Життєвий цикл програмного забезпечення (SDLC – Software Development Life Cycle) – період часу, який починається з моменту прийняття рішення про необхідність створення програмного продукту і закінчується в момент його повного вилучення з експлуатації. Цей цикл – процес побудови і розвитку програмного забезпечення.

Тестоване програмне забезпечення повинно проходити кожен з етапів тестування, обумовлених продакт овнерами, менеджментом компанії розробника та тест-дизайнерами для того, щоб вважати програмний продукт відносно якісним або придатним до використання [5].

Важливо розуміти, що довжина такої ітерації (і, відповідно, ступінь подробиці кожної стадії) може варіюватися в широкому діапазоні – від одиниць годин до десятків місяців. Як правило, якщо мова йде про тривалому проміжку часу, він розбивається на безліч відносно коротких ітерацій, але сам при цьому «тяжіє» до тієї чи іншої стадії в кожен момент часу (наприклад, на початку проекту більше планування, в кінці - більше звітності).

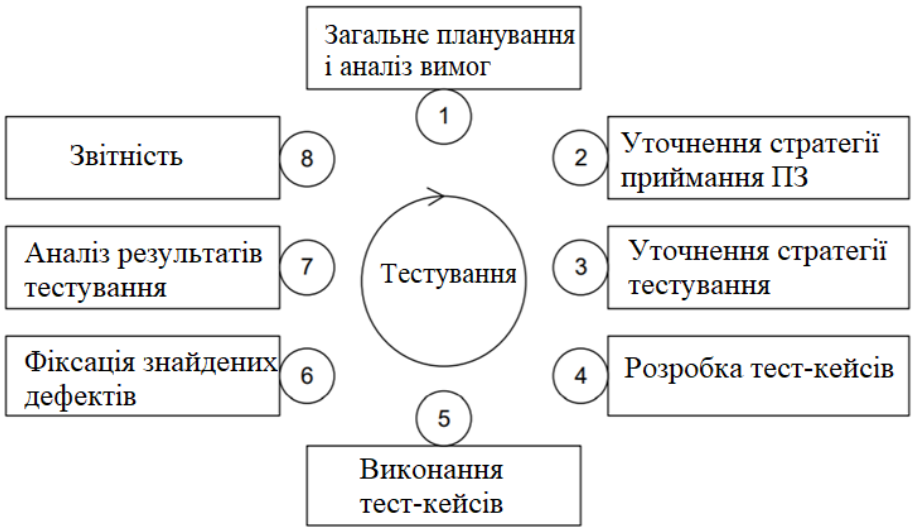

Рис.3.6. Життєвий цикл тестування програмного забезпечення

Стадія 1 (загальне планування і аналіз вимог) об'єктивно необхідна як мінімум для того, щоб мати відповідь на такі питання, як: що нам належить тестувати; як багато буде роботи; які є складності; чи все необхідне у нас є і т.п. Як правило, отримати відповіді на ці питання неможливо без аналізу вимог, тому що саме вимоги є первинним джерелом відповідей.

Стадія 2 (уточнення критеріїв приймання) дозволяє сформулювати або уточнити метрики і ознаки можливості або необхідності початку тестування (entry criteria), припинення (suspension criteria) і відновлення (resumption criteria) тестування, завершення або припинення тестування (exit criteria).

Стадія 3 (уточнення стратегії тестування) являє собою ще одне звернення до планування, але вже на локальному рівні: розглядаються і уточнюються ті частини стратегії тестування (test strategy), які є актуальними для поточної ітерації.

Стадія 4 (розробка тесткейсів) присвячена розробці, перегляду, уточнення, доопрацювання, переробки та іншим діям з тест-кейсами, наборами тест-кейсів, тестовими сценаріями та іншими артефактами, які будуть використовуватися при безпосередньому виконанні тестування.

Стадія 5 (виконання тест-кейсів) і стадія 6 (фіксація знайдених дефектів) тісно пов'язані між собою і фактично виконуються паралельно: дефекти фіксуються відразу по факту їх виявлення в процесі виконання тест-кейсів.

Однак найчастіше після виконання всіх тест-кейсів і написання всіх звітів про знайдених дефектах проводиться явно виділена стадія уточнення, на якій всі звіти про дефекти розглядаються повторно з метою формування єдиного розу-

міння проблеми та уточнення таких характеристик дефекту, як важливість і терміновість.

Стадія 7 (аналіз результатів тестування) і стадія 8 (звітність) також тісно пов'язані між собою і виконуються практично паралельно. Сформульовані на стадії аналізу результатів висновки безпосередньо залежать від плану тестування, критеріїв приймання та уточненої стратегії, отриманих на стадіях 1, 2 і 3. Отримані висновки оформляються на стадії 8 і служать основою для стадій 1, 2 і 3 наступній ітерації тестування. Таким чином, цикл замикається.

ТЕМА 4. ВИМОГИ ДО РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Вимога (requirement) - опис того, які функції і з дотриманням яких умов має виконувати програмне забезпечення в процесі вирішення корисної для користувача завдання.

Важливість вимог

Вимоги є відправною точкою для визначення того, що проектна команда буде проектувати, реалізовувати і тестувати. Якщо у вимогах щось «не те», то і реалізовано буде «не те», тобто колосальна робота безлічі людей буде виконана даремно.

Важливість вимог:

• Дозволяють зрозуміти, що і з дотриманням яких умов система повинна робити.

• Надають можливість оцінити масштаб змін і управляти змінами.

• Є основою для формування плану проекту (в тому числі плану тестування).

• Допомагають запобігати або вирішувати конфліктні ситуації.

• Спрощують розстановку пріоритетів в наборі завдань.

• Дозволяють об'єктивно оцінити ступінь прогресу в розробці проекту.

Проект з поганими вимогами:

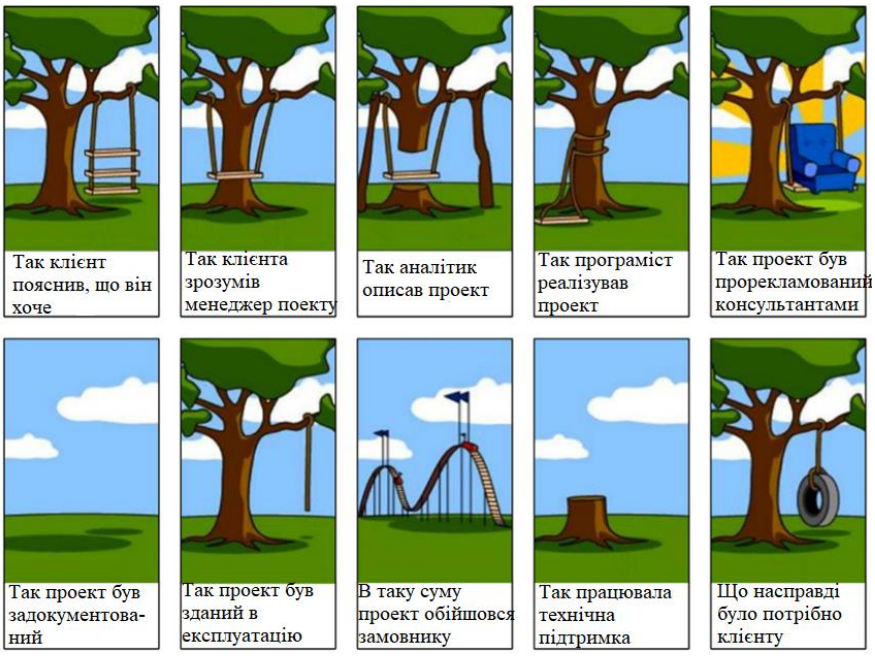

Рис. 4.1. Проект з поганими вимогами

Типи документаії:

1. **Продуктна документація** (product documentation, development

documentation) використовується проектною командою під час розробки та підтримки продукту. Вона включає:

План проекту (project management plan) і в тому числі тестовий план(Test plan).

- Вимоги до програмного продукту (product requirements document, PRD) і функціональні специфікації (functional specifications document, FSD; software requirements specification, SRS).
- Архітектуру і дизайн (architecture and design).
- Тест-кейси і набори тест-кейсів (test cases, Test suites).
- Технічні специфікації (technical specifications), такі як схеми баз даних, опису алгоритмів, інтерфейсів і т.д.

2. Проектна документація (project documentation) включає в себе як продуктну документацію, так і деякі додаткові види документації і використовується не тільки на стадії розробки, але і на більш ранніх і пізніх стадіях (наприклад, на стадії впровадження та експлуатації).

Проектна документація включає:

Призначену для користувача супровідну документацію (user and accompanying documentation), таку як вбудована допомога, керівництво по установці і використанню, ліцензійні угоди і т.д.

Маркетингову документацію (market requirements document, MRD), яку представники розробника або замовника використовують як на початкових етапах (для уточнення суті і концепції проекту), так і на фінальних етапах розвитку проекту (для просування продукту на ринку).

Рис. 4.2. Співвідношення понять «продуктна документація» та «проектна документація»

Джерела і шляхи виявлення вимог

Рис. 4.3. Джерела і шляхи виявлення вимог

Інтерв'ю. Самий універсальний шлях виявлення вимог, що полягає в спілкуванні проектного фахівця (як правило, фахівця з бізнесаналізу) і представника замовника (або експерта, користувача і т.д.). Інтерв'ю може проходити в класичному розумінні цього слова (бесіда у вигляді «питання-відповідь»), у вигляді переписки і т.п. Головним тут є те, що ключовими фігурами виступають двоє – той, у кого беруть інтерв'ю і інтерв'юер (хоча це і не виключає наявності «аудиторії слухачів», наприклад, у вигляді осіб, поставлених в копію листування).

Робота з фокусними групами. Може виступати як варіант «розширеного інтерв'ю», де джерелом інформації є не одна особа, а група осіб.

(Як правило, представляють собою цільову аудиторію, і / або які мають важливою для проекту інформацію, і / або уповноважених приймати важливі для проекту рішення).

Анкетування. Цей варіант виявлення вимог викликає багато суперечок, тому що при невірній реалізації може привести до нульового результату при об'ємних витратах. У той же час при правильній організації анкетування дозволяє автоматично зібрати і обробити величезну кількість відповідей від величезної кількості респондентів. Ключовим фактором успіху є правильне складання анкети, правильний вибір аудиторії і правильне вручення анкети.

Семінари і мозковий штурм. Семінари дозволяють групі людей дуже швидко обмінятися інформацією (і наочно продемонструвати ті чи інші ідеї), а також добре поєднуються з інтерв'ю, анкетуванням, прототипування і моделюванням - в тому числі для обговорення результатів та формування висновків і рішень. Мозковий штурм може проводитися і як частина семінару, і як окремий вид діяльності. Він дозволяє за мінімальний час згенерувати велику кількість ідей, які в подальшому можна не поспішаючи розглянути з точки зору їх викори-

стання для розвитку проекту.

Спостереження. Може виражатися як в буквальному спостереженні за деякими процесами, так і у включенні проектного фахівця в ці процеси як учасника. З одного боку, спостереження дозволяє побачити те, про що (по абсолютно різних міркувань) можуть промовчати ті, в яких беруть інтерв'ю, анкетованих і представники фокус-груп, але з іншого - забирає дуже багато часу і найчастіше дозволяє побачити лише частину процесів.

Прототипування. Складається в демонстрації і обговорення проміжних версій продукту (наприклад, дизайн сторінок сайту може бути спочатку представлений у вигляді картинок, і лише потім зверстаний). Це один з кращих шляхів пошуку єдиного розуміння і уточнення вимог, однак він може привести до серйозних додаткових витрат при відсутності спеціальних інструментів (що дозволяють швидко створювати прототипи) і занадто ранньому застосуванні (коли вимоги ще не стабільні, і висока ймовірність створення прототипу, що має мало спільного з тим, що хотів замовник).

Аналіз документів. Добре працює тоді, коли експерти в предметної області (тимчасово) недоступні, а також в предметних областях, що мають загальноприйняту усталену регламентує документацію. Також до цієї техніки відноситься і просто вивчення документів, що регламентують бізнес-процеси в предметної області замовника або в конкретній організації, що дозволяє придбати необхідні для кращого розуміння суті проекту знання.

Моделювання процесів і взаємодій. Може застосовуватися як до «Бізнес-процесів і взаємодій» так і до «технічних процесів і взаємодій». Дана техніка вимагає високої кваліфікації фахівця з бізнес-аналізу, тому що пов'язана з обробкою великого обсягу складної (і часто погано структурованої) інформації.

Самостійне опис. Є не стільки технікою виявлення вимог, скільки технікою їх фіксації і формалізації. Дуже складно (і навіть не можна!) намагатися самому «придумати вимоги за замовника», але в спокійній обстановці можна самостійно обробити зібрану інформацію і акуратно оформити її для подальшого обговорення і уточнення.

Рівні і типи вимог

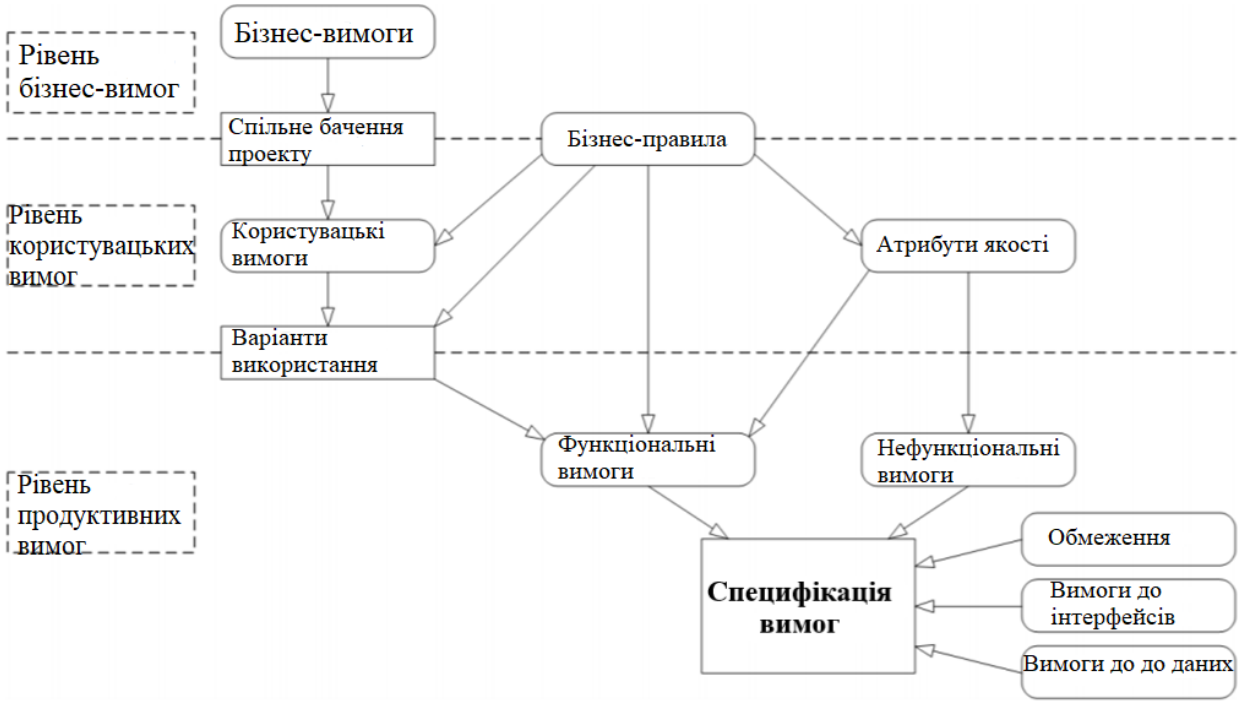

Рис. 4.4. Рівні і типи вимог

Бізнес-вимоги (business requirements) висловлюють мету, заради якої розробляється продукт (навіщо взагалі він потрібен, яка від нього очікується користь, як замовник з його допомогою буде отримувати прибуток). Результатом виявлення вимог на цьому рівні є спільне бачення (vision and scope) - документ, який, як правило, представлений простим текстом і таблицями. Тут немає деталізації поведінки системи і інших технічних характеристик, але цілком можуть бути визначені пріоритети розв'язуваних бізнес-задач, ризики і т.п.

Призначені для користувача вимоги (user requirements) описують завдання, які користувач може виконувати за допомогою розроблюваної системи (реакцію системи на дії користувача, сценарії). Оскільки тут уже з'являється опис поведінки системи, вимоги цього рівня можуть бути використані для оцінки обсягу робіт, вартості проекту, часу розробки і т.д. Призначені для користувача вимоги оформляються у вигляді варіантів використання (use cases), призначених для користувача історій (user stories), призначених для користувача сценаріїв (user scenarios).

Бізнес-правила (business rules) описують особливості прийнятих в предметній області (і / або безпосередньо у замовника) процесів, обмежень і інших правил. Ці правила можуть ставитися до бізнес-процесів, правил роботи співробітників, нюансам роботи ПЗ і т.д.

Атрибути якості (quality attributes) розширюють собою нефункціональні вимоги і на рівні користувача вимог можуть бути представлені у вигляді опису ключових для проекту показників якості (властивостей продукту, не пов'язаних з функціональністю, але є важливими для досягнення цілей створення продукту - продуктивність, масштабованість, відновлюваність).

Функціональні вимоги (functional requirements) описують поведінку системи, тобто її дії (обчислення, перетворення, перевірки, обробку і т.д.). В контексті проектування функціональні вимоги в основному впливають на дизайн системи. Варто пам'ятати, що до поведінки системи відноситься не тільки те, що система повинна робити, а й те, що вона не повинна робити.

Нефункціональние вимоги (non-functional requirements) описують властивості системи (зручність використання, безпеку, надійність, розширюваність і т.д.), якими вона повинна володіти при реалізації своєї поведінки.

Тут наводиться більш технічне і детальний опис атрибутів якості. В контексті проектування нефункціональні вимоги в основному впливають на архітектуру системи.

Обмеження (limitations, constraints) представляють собою фактори, що обмежують вибір способів і засобів (в тому числі інструментів) реалізації продукту.

Вимоги до інтерфейсів (external interfaces requirements) описують особливості взаємодії системи, що розробляється з іншими системами і операційним середовищем.

Вимоги до даних (data requirements) описують структури даних (і самі дані), які є невід'ємною частиною розроблюваної системи. Часто сюди відносять опис бази даних і особливостей її використання.

Специфікація вимог (software requirements specification, SRS) об'єднує в собі опис усіх вимог рівня продукту і може представляти собою досить об'ємний документ (сотні і тисячі сторінок).

Оскільки вимог може бути дуже багато, а їх доводиться не тільки одного разу написати і узгодити між собою, а й постійно оновлювати, роботу проектної команди з управління вимогами значно полегшують відповідні інструментальні засоби (requirements management tools).

Властивості якісних вимог

Рис. 4.5. Властивості якісних вимог

Одиничність – Вимога описує одну і тільки одну річ.

Завершеність – Вимога повністю визначена в одному місці і вся необхідна інформація присутня.

Послідовність – Вимога не суперечить іншим вимогам і повністю відповідає зовнішній документації.

Атомарність – Вимога не може бути розбита на ряд більш детальних вимог без втрати завершеності.

Відстежування – Вимога повністю або частково відповідає діловим потребам як заявлено зацікавленими особами і задокументовано.

Відстеження буває вертикальним (vertical traceability) і горизонтальним (horizontal traceability). Вертикальне дозволяє співвідносити між собою вимоги на різних рівнях вимог, горизонтальне дозволяє співвідносити вимогу з тест-планом, тест-кейсами, архітектурними рішеннями і т.д.

Типові проблеми з простежуваності:

• Вимоги не пронумеровані, не структуровані, не мають змісту, не мають працюючих перехресних посилань.

• Набір вимог неповний, носить уривчастий характер з явними «пробілами».

Актуальність – Вимога не стала застарілою з часом.

Здійснимість – Вимога може бути реалізовано в межах проекту.

Типові проблеми зі здійснимістю:

• ***Так зване «озолочення»*** (gold plating) - вимоги, які вкрай довго і / або дорого реалізуються і при цьому практично не приносять користі для кінцевих користувачів (наприклад: «настройка параметрів для підключення до бази даних повинна підтримувати розпізнавання символів з жестів, отриманих з пристроїв тривимірного введення»).

• ***Технічно не реалізовуються на сучасному рівні розвитку технологій вимоги***

Недвозначність – Вимога коротко визначена без звернення до технічного жаргону та інших прихованих формулювань. Вона виражає об'єктивні факти, можлива одна і тільки одна інтерпретація. Визначення не містить нечітких фраз. Використання негативних тверджень заборонено.

Обов'язковість – Вимога представляє певну характеристику, відсутність якої призведе до неповноцінності рішення, яка не може бути проігнорована.

Обов'язковість, потрібність (obligatoriness) і актуальність (up-to-date). Якщо вимога не є обов'язковою до реалізації, вона повинна бути просто виключена з набору вимог. Якщо вимога потрібна, але «не дуже важлива», для вказівки цього факту використовується вказівка пріоритету. Також вимоги, що втратили актуальність, повинні бути виключені.

Типові проблеми з обов'язковістю і актуальністю:

• Вимогу було додано «про всяк випадок», хоча реальної потреби в ньому не було і немає.

Верифікованість – Реалізованість вимоги може бути визначена через один з чотирьох можливих методів: огляд, демонстрація, тест чи аналіз [6].

Несуперечливість, послідовність (consistency). Вимога не повинна містити внутрішніх протиріч і суперечностей іншим вимогам і документам.

Можливість модифікацій (modifiability). Це властивість характеризує простоту внесення змін в окремі вимоги і в набір вимог. Можна говорити про наявність модифікованості в тому випадку, якщо при доопрацюванні вимог шукану інформацію легко знайти, а її зміна не призводить до порушення інших описаних в цьому переліку властивостей.

Типові проблеми з модифікацією:

• Вимоги неатомарни і непростежувані, тому їх зміна з високою ймовірністю породжує

суперечливість.

• Вимоги спочатку суперечливі. У такій ситуації внесення змін (не пов'язаних з усуненням суперечливості) тільки погіршує ситуацію, збільшуючи суперe-

чливість і знижуючи простежуваність.

• Вимоги представлені в незручній для обробки формі.

Проранжованість за важливістю, стабільностю, терміновостю (ranked for importance, stability, priority).

Важливість характеризує залежність успіху проекту від успіху реалізації вимоги. **Стабільність** характеризує ймовірність того, що в доступному для огляду майбутньому у вимогу не буде внесено ніяких змін.

Терміновість визначає розподіл в часі зусиль проектної команди по реалізації тієї чи іншої вимоги.

Типові проблеми з проранжованістю складаються в її відсутності або неправильної реалізації і призводять до негативних наслідків:

• Проблеми з проранжованістю за важливістю підвищують ризик невірного розподілу зусиль проектної команди, спрямування зусиль на другорядні завдання і кінцевого провалу проекту через нездатність продукту виконувати ключові завдання з дотриманням ключових умов.

Проблеми з проранжованістю за стабільностю підвищують ризик виконання безглуздої роботи по вдосконаленню, реалізації та тестування вимог, які в самий найближчий час можуть зазнати кардинальні зміни (аж до повної втрати актуальності).

Проблеми з проранжованістю по терміновості підвищують ризик порушення бажаної замовником послідовності реалізації функціональності і введення цієї функціональності в експлуатацію.

Рис. 4.6. Залежність важливості і терміновості

Коректність (correctness) і перевіряємість (verifiability). Фактично ці властивості випливають з дотримання всіх перерахованих вище.

Перевіряємість - це можливість створення об'єктивного тест-кейса (тест-кейсів), показує, що вимога реалізована вірно і поведінка додатку в точності відповідає вимозі.

До типових проблем з коректністю також можна віднести:

• помилки, наявність неаргументованих вимог до дизайну та архітектури; погане оформлення тексту і супутньої графічної інформації, невірний рівень деталізації, вимоги до користувача.

Техніки тестування вимог

Взаємний перегляд (peer review). Взаємний перегляд («рецензування») є однією з найбільш активно використовуваних технік тестування вимог і може бути представлений в одній з трьох наступних форм (по наростанню складності і ціни):

Швидкий перегляд (walkthrough) може виражатися як в показі автором своєї роботи колегам з метою створення загального розуміння і отримання зворотного зв'язку, так і в простому обміні результатами роботи між двома і більше авторами з тим, щоб колега висловив свої питання і зауваження.

Це найшвидший, дешевий і найпошитеніший вид перегляду.

Технічний перегляд (technical review) виконується групою фахівців. В ідеальній ситуації кожен фахівець повинен представляти свою область знань. Тестований продукт не може вважатися досить якісним, поки хоча б у одного переглядача залишаються зауваження.

Формальна інспекція (inspection) являє собою структурований, систематизований і документований підхід до аналізу документації. Для його виконання залучається велика кількість фахівців, саме виконання займає досить багато часу, і тому цей варіант перегляду використовується досить рідко (як правило, при отриманні на супровід і доопрацювання проекту, створенням якого раніше займалася інша компанія).

Питання. Наступною очевидною технікою тестування і підвищення якості вимог є (повторне) використання технік виявлення вимог, а також (як окремий вид діяльності) - задавання питань.

Якщо хоч щось у вимогах викликає нерозуміння або підозру – задається питання. Можна запитати представників замовника, можна звернутися до довідкової інформації. З багатьох питань можна звернутися до більш досвідчених колег за умови, що у них є відповідна інформація, раніше отримана від замовника.

Головне, щоб запитання було сформульоване таким чином, щоб отриманий відповідь дозволив покращити вимоги.

Тест-кейси і чек-листи. Якісна вимога є перевіряємою, а значить, повинні існувати об'єктивні способи визначення того, чи правильно реалізована вимога. Продумування чек-листів або навіть повноцінних тест-кейсів в процесі аналізу вимог дозволяє визначити, наскільки вимогу можна перевірити.

Дослідження поведінки системи. Ця техніка логічно випливає з попередньої (продумування тест-кейсів і чек-листів), але відрізняється тим, що тут тестуванню піддається, як правило, не одна вимога, а цілий набір. Тестувальник подумки моделює процес роботи користувача з системою, створеною за тестовими вимогами, і шукає неоднозначні або зовсім неописані варіанти поведінки системи. Цей підхід складний, вимагає достатньої кваліфікації тестувальника, але здатний виявити нетривіальні недоробки, які майже неможливо помітити, тестуючи вимоги окремо.

Малюнки (графічне представлення). Щоб побачити загальну картину вимог цілком, дуже зручно використовувати малюнки, схеми, діаграми, інтелект-карти і т.д. Графічне представлення зручно одночасно своєї наочністю і стислістю. На малюнку дуже легко помітити, що якісь елементи «не стикуються», що десь чогось не вистачає і т.д. Якщо для графічного представлення вимог будете використовувати загальноприйняту нотацію (наприклад, UML), то схему зможуть розуміти і допрацьовувати колеги, а в результаті може вийти гарне доповнення до текстової форми вимог.

Прототипування. Прототипування часто наслідком створення графічного представлення та аналізу поведінки системи.

З використанням спеціальних інструментів можна дуже швидко зробити начерки призначених для користувача інтерфейсів, оцінити придатність тих чи інших рішень і навіть створити не просто «прототип заради прототипу», а заготовку для подальшої розробки, якщо виявиться, що реалізоване в прототипі (можливо, з невеликими доробками) влаштовує замовника.

Типові помилки при аналізі та тестуванні вимог

Аналіз та тестування вимог важливі етапи в процесі розробки програмного забезпечення, і вони можуть включати різноманітні виклики та помилки.

Типові помилки, які можуть виникнути під час аналізу та тестування вимог:

1. Неповні або нечіткі вимоги: Вимоги можуть бути сформульовані неповно або невірно, що може призвести до непорозумінь та помилок у подальшому

розробці.

2. Неоднозначність вимог: Вимоги, які можна розуміти кількома способами, створюють ризик неправильного розуміння і, відповідно, неправильної реалізації.

3. Протиріччя вимог: Різні частини документації можуть містити протиріччя, що може важити для розробників та тестувальників при реалізації та валідації вимог.

4. Неактуальні вимоги: Вимоги можуть застаріти або стати неактуальними у зв'язку зі змінами в бізнес-процесах чи стратегії компанії.

5. Невизначеність вимог до продуктивності і безпеки: Недостатня чіткість щодо очікуваних параметрів продуктивності та вимог до безпеки може призвести до проблем під час етапу тестування.

6. Відсутність визначення критеріїв прийняття (Acceptance Criteria): Неясно визначені критерії успішного завершення робіт можуть призвести до непорозумінь між командами та замовником.

7. Неадекватність тестових сценаріїв: Недостатні або неправильно розроблені тестові сценарії можуть привести до пропуску важливих аспектів та вимог.

8. Не врахування змін у вимогах під час розробки: Недостатній механізм для виявлення та обробки змін у вимогах може призвести до того, що розробка буде неспроможною адекватно реагувати на нові вимоги.

9. Недостатнє тестування реальних сценаріїв використання: Тестування може не враховувати реальних сценаріїв використання продукту, що може призвести до непередбачених проблем.

10. Недостатній зв'язок між тестовими випробуваннями та вимогами: Тестові сценарії можуть бути не достатньо пов'язаними з конкретними вимогами, що може призвести до неповного покриття функціоналу.Уникнення цих помилок важливо для успішної розробки програмного забезпечення, і це може бути досягнуте через вивчення вимог, активну комунікацію з усіма зацікавленими сторонами та ретельне тестування.

11. Зміна формату файлу і документа. З якоїсь незрозумілої причини дуже багато початкових тестувальників прагнуть повністю знищити вихідний документ, замінивши текст таблицями (або навпаки), перенести дані з Word в Excel і т.д. Це можна зробити тільки в одному випадку: якщо ви попередньо домовилися про подібні зміни з автором документа. В іншому випадку ви повністю знищуєте чиюсь роботу, роблячи подальший розвиток документа вкрай негативний.

12. Відмітка того факту, що з вимогою все в порядку. Якщо у вас не виникло питань і / або зауважень до вимоги - не треба про це писати. Будь-які позначки в документі підсвідомо сприймаються як ознака проблеми, і таке «схвалення вимог» тільки дратує і ускладнює роботу з документом - складніше стає помітити позначки, які стосуються проблемам.

13. Опис однієї і тієї ж проблеми в декількох місцях. Позначки, коментарі, зауваження та питання теж повинні мати властивості якісних вимог.

14. Написання питань і коментарів без зазначення місця вимоги, до якого вони належать. В іншому випадку породжується неоднозначність або позначка робиться безглуздою, тому що стає неможливо зрозуміти, про що взагалі йде мова.

15. Задавання погано сформульованих питань. Іноді одне-два слова можуть знищити відмінну ідею, перетворивши хороше запитання в погане.

16. Написання дуже довгих коментарів і / або питань.

17. Критика тексту або навіть його автора. Пам'ятайте, що ваше завдання – зробити вимоги краще, а не показати їх недоліки (або недоліки автора). Тому коментарі виду «погане вимога», «невже ви не розумієте, як нерозумно це звучить», «треба переформулювати» недоречні і недопустимі.

18. Категоричні заяви без обґрунтування. Як продовження помилки «Критика тексту або навіть його автора» можна відзначити і просто категоричні заяви на кшталт «це неможливо», «ми не будемо цього робити», «це не потрібно»

19. Вказівка проблеми з вимогами без пояснення її суті. Автор вихідного документа може не бути фахівцем з тестування або бізнес-аналізу. Тому просто позначка в стилі «неповнота», «двозначність» і т.д. можуть нічого йому не сказати.

20. Погане оформлення питань і коментарів. Намагайтеся зробити ваші запитання та коментарі максимально простими для сприйняття. Варто пам'ятати не тільки про стислості формулювань, а й про оформлення тексту.

21. Опис проблеми не в тому місці, до якого вона належить. Класичним прикладом може бути неточність у виносці, додатку або малюнку, яка чомусь описана не там, де вона знаходиться, а в тексті, посилається на відповідний елемент. Винятком може вважатися суперечливість, при якій описати проблему потрібно в обох місцях.

22. Помилкове сприйняття вимоги як «вимоги до користувача».

23. Приховане редагування вимог. Цю помилку можна сміливо віднести

до розряду вкрай небезпечних. Її суть полягає в тому, що тестувальник довільно вносить правки в вимоги, ніяк не зазначаючи цей факт. Відповідно, автор документа, швидше за все, не помітить такої правки, а потім буде дуже здивований, коли в продукті щось буде реалізовано зовсім не так, як колись було описано в вимогах.

24. Аналіз, який не відповідає рівню вимог. При тестуванні вимог слід постійно пам'ятати, до якого рівня вони відносяться, тому що в іншому випадку з'являються такі типові помилки:

• Додавання в бізнес-вимоги дрібних технічних подробиць.

• Дублювання на рівні користувача вимог частини бізнес-вимог

• Недостатня деталізація вимог рівня продукту (загальні фрази, допустимі, наприклад, на рівні бізнес-вимог, тут вже повинні бути гранично деталізовані, структуровані і доповнені докладної технічною інформацією).

ТЕМА 5. ВИДИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Тестування програмного забезпечення - це важливий етап в розробці, який дозволяє виявляти помилки та переконатися у відповідності програми вимогам та очікуванням користувачів. Існує багато різних видів тестування, які можуть бути використані для різних аспектів програмного забезпечення.

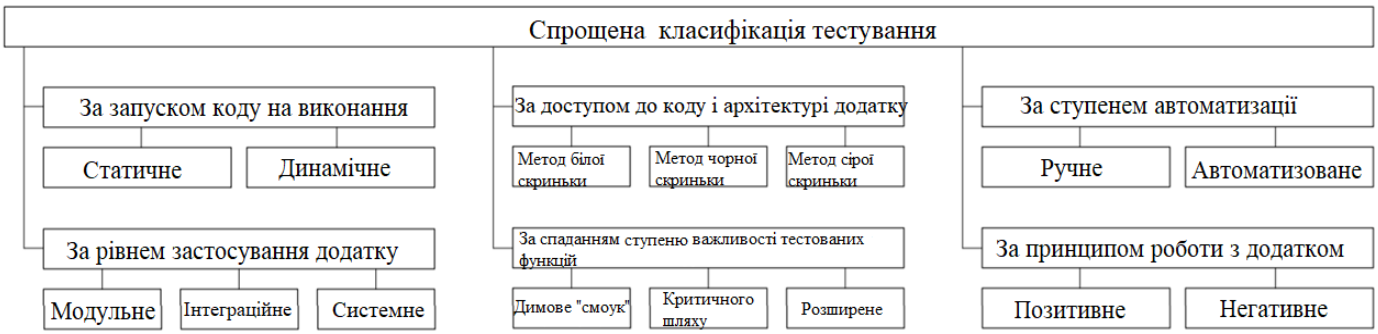

Рис. 5.1 Спрощена класифікація тестування

За запуском коду на виконання:

- ❑ Статичне тестування - без запуску.
- ❑ Динамічне тестування - з запуском.

По доступу до коду та архітектури програми:

- ❑ Метод білого ящика - доступ до коду є.
- ❑ Метод чорного ящика - доступу до коду немає.
- ❑ Метод сірого ящика - до частини коду доступ є, до частини - немає.

За ступенем автоматизації:

- ❑ Ручне тестування - тест-кейси виконує людина.
- ❑ Автоматизоване тестування - тест-кейси частково або повністю виконує спеціальний інструментальний засіб.

За рівнем деталізації додатку (за рівнем тестування):

- ❑ Модульне (компонентне) тестування - перевіряються окремі невеликі частини програми.
- ❑ Інтеграційне тестування - перевіряється взаємодія між декількома частинами програми.
- ❑ Системне тестування - додаток перевіряється як єдине ціле.

За (зменшенням) ступенем важливості тестованих функцій (за рівнем функціонального тестування):

- ❑ Димове тестування - перевірка найважливішою, самої ключової функціональності, непрацездатність якої робить безглуздою саму ідею використання програми.

- ❑ Тестування критичного шляху - перевірка функціональності, яка використовується типовими користувачами в типовій повсякденній діяльності.
- ❑ Розширене тестування - перевірка всієї (решті) функціональності, заявленої в вимогах.

За принципами роботи з додатком:

- ❑ Позитивне тестування - всі дії з додатком виконуються строго по інструкції без жодних неприпустимих дій, некоректних даних і т.д. Можна образно сказати, що додаток досліджується в «тепличних умовах».
- ❑ Негативне тестування - в роботі з додатком виконуються некоректні операції та використовуються дані, потенційно призводять до помилок (класика жанру - ділення на нуль).

Класифікація по запуску коду на виконання

Не всяке тестування передбачає взаємодію з працюючим додатком. Тому в рамках даної класифікації виділяють:

- ❑ **Статичне тестування (static testing)** - тестування без запуску коду на виконання.

В рамках цього підходу тестування можуть потрапити під вплив:

- ✓ Документи (вимоги, тест-кейси, опису архітектури додатку, схеми баз даних і т.д.).
- ✓ Графічні прототипи (наприклад, ескізи призначеного для користувача інтерфейсу).
- ✓ Код додатку (що часто виконується самими програмістами в рамках аудиту коду (code review), що є специфічною варіацією взаємного перегляду в застосуванні до вихідного коду)
- ✓ додатки також можна перевіряти з використанням технік тестування на основі структур коду.
- ✓ Параметри (настройки) середовища виконання програми.
- ✓ Підготовлені тестові дані.
- ✓ **Динамічне тестування (dynamic testing)** - тестування з запуском коду на виконання. Запускатися на виконання може як код всієї програми цілком (системне тестування), так і код декількох взаємопов'язаних частин (інтеграційне тестування), окремих частин (модульне або компонентне тестування) і навіть окремі ділянки коду.

Основна ідея цього виду тестування полягає в тому, що перевіряється реальна поведінка (частини) додатку.

Класифікація по доступу до коду і архітектурі додатка

Метод білого ящика (white box testing, open box testing, clear box testing, glass box testing) - у тестувальника є доступ до внутрішньої структурі та коду додатка, а також є достатньо знань для розуміння побаченого. Виділяють навіть супутнє тестування по методу білого ящика - глобальну техніку - тестування на основі дизайну (design-based testing).

Метод чорного ящика (black box testing, closed box testing, specificationbased testing) - у тестувальника або немає доступу до внутрішньої структурі та коду програми, або недостатньо знань для їх розуміння, або він свідомо не звертається до них у процесі тестування. При цьому абсолютна більшість перерахованих на малюнках видів тестування працюють за методом чорного ящика, ідею якого в альтернативному

визначенні можна сформулювати так: тестувальник надає на додаток вплив (і перевіряє реакцію) тим же способом, яким при реальній експлуатації додатку на нього впливали б користувачі або інші додатки.

Метод сірого ящика (gray box testing) - комбінація методів білого ящика і чорного ящика, яка полягає в тому, що до частини коду і архітектури у тестувальника доступ є, а до частини - немає. На малюнках цей метод позначений особливим пунктиром і сірим кольором тому, що його явна згадка - вкрай рідкісний випадок: зазвичай говорять про методи білого або чорного ящика в застосуванні до тих чи інших частинах додатка, при цьому розуміючи, що «додаток повністю» тестується за методом сірого ящика.

Переваги тестування методом білої скриньки:

1. Глибоке покриття коду: Тестування білої скриньки дозволяє отримати глибоке розуміння програмного коду і покриття всіх можливих шляхів виконання програми.

2. Ефективність в виявленні помилок: Здатність ідентифікувати ізольовані або складні помилки, які можуть залишитися непоміченими при інших методах тестування.

3. Оптимізація коду: Допомагає здійснити оптимізацію та вдосконалення програмного коду.

4. Автоматизація тестування: Біле скринькове тестування легше автоматизувати, оскільки тестери можуть використовувати знання про внутрішню структуру коду для створення ефективних тестів.

5. Можливість розгляду безпеки: Цей метод дозволяє проводити тестування на вразливість безпеки, виявляти потенційні проблеми та захищати систему від атак.

Недоліки тестування методом білої скриньки:

1. Обмежена виразність: Тестування білої скриньки не завжди здатне виявити всі можливі стани програми, оскільки не враховує зовнішніх аспектів та специфікацій.

2. Залежність від коду: Тестування вимагає доступу до вихідного коду, що робить його непридатним для використання в ситуаціях, де вихідний код не доступний (наприклад, при тестуванні комерційного програмного забезпечення).

3. Складність в роботі зі змінами: Зміни в програмному коді можуть вимагати переписування частин тестових сценаріїв, що може призвести до витрат часу та ресурсів.

4. Не завжди зручно для великих систем: Великі системи можуть вимагати значних зусиль для аналізу всього коду та розробки відповідних тестових сценаріїв.

5. Не виявляє функціональні аспекти: Тестування білої скриньки не завжди ефективно для виявлення функціональних аспектів, оскільки акцентується на внутрішній структурі.

6. Незалежність від користувача: Тестування білої скриньки може ігнорувати погляди та очікування реальних користувачів.

Переваги тестування методом чорної скриньки:

1. Незалежність від реалізації: Тестери можуть працювати без відомостей про внутрішню структуру програми, що дозволяє їм фокусуватися на зовнішньому поведінці.

2. Висока ступінь абстракції: Цей метод дозволяє тестерам думати про програму як про "чорний ящик", спрощуючи процес тестування.

3. Спрощення тестів для кінцевих користувачів: Тестування може бути спрощено для кінцевих користувачів, оскільки їм не потрібно розуміти деталі внутрішньої реалізації.

4. Ефективність в виявленні функціональних помилок: Добре підходить для виявлення помилок у функціональності та відповідності вимогам.

5. Придатність для регресійного тестування: Добре підходить для регресійного тестування, оскільки можна перевірити, чи відповідає програмне забезпечення очікуванням після внесення змін.

6. Зручність для некомандних тестерів: Навіть ті, хто не є розробниками програмного забезпечення, можуть бути ефективними тестерами методом чорної скриньки.

Недоліки тестування методом чорної скриньки:

1. Недостатня глибина тестування: Тестування може бути менш ефективним у виявленні деяких внутрішніх помилок, які не впливають на зовнішнє поведінку.

2. Обмеженість у виявленні структурних проблем: Не дозволяє виявити проблеми з ефективністю, дефектами в коді чи іншими структурними аспектами.

3. Потреба в повторному тестуванні при зміні: Зміни у програмному коді можуть вимагати повторного тестування всієї системи.

4. Залежність від специфікацій та документації: Точність та повнота результатів тестування можуть залежати від якості специфікацій та документації.

5. Важко визначити покриття коду: Трудно визначити, наскільки повністю тестові сценарії охоплюють весь код.

6. Великий обсяг тестів може бути неефективним: Велика кількість можливих комбінацій вводу та станів може зробити тестування чорною скринькою неефективним.

Вибір методу тестування повинен базуватися на конкретних вимогах проекту та особливостях програмного продукту. У багатьох випадках комбінація методів тестування (метод сірої скриньки) є оптимальним рішенням.

Класифікація за ступенем автоматизації:

Ручне тестування (manual testing) - тестування, в якому тест-кейси виконуються людиною вручну без використання засобів автоматизації. Незважаючи на те, що це звучить дуже просто, від тестувальника в ті чи інші моменти часу потрібні такі якості, як терплячість, спостережливість, креативність, вміння ставити нестандартні експерименти, а також уміння бачити і розуміти, що відбувається «всередині системи», тобто як зовнішні впливи на додаток трансформуються в його внутрішні процеси.

Автоматизоване тестування (automated testing, test automation) - набір технік, підходів і інструментальних засобів, що дозволяє виключити людини з виконання деяких завдань в процесі тестування.

Тест-кейси частково або повністю виконує спеціальний інструментальний засіб, однак розробка тест-кейсів, підготовка даних, оцінка результатів виконання, написання звітів про виявлені дефекти – все це і багато іншого, як і раніше робить людина.

Переваги та недоліки автоматизованого тестування

Серед головних недоліків автоматизованного тестування можна виділити:

1. Складність. Написання автоматизованих тестів можна порівняти із написанням коду програмного забезпечення, так як слід враховувати архітектурні

особливості, адже будуючи автоматизацію тестування з нуля, потрібно буде приділяти увагу дизайну системи, щоб мати можливість легко підтримувати її в майбутньому.

2. Високі початкові витрати. Оплата послуг спеціаліста з побудови автоматизації більша, ніж інженера по забезпеченю якості, що виконує ручне тестування.

3. Автоматизовані тести не можуть покрити увесь функціонал продукту. Хоча тестування більшої функціональної частини застосунку може бути автоматизоване, такі речі як графіка або звук можна перевірити суто на наявність, а не на зміст.

З іншого боку автоматизоване тестування в певних проектах та для певних його частин може бути рятівною паличкою. Серед головних його переваг можна виділити:

1. Швидкість виконання. Коли система вже написана і функціонує, сам процес виконання тестів значно швидший, ніж мануальне тестування у тому ж випадку.

2. Зменшеннявитратнадистанції.Попривеликіпочатковівитрати,надистанції автоматизоване тестування приводить до їх зменшення, адже необхідність у ручному тестуванні спадає.

3. Надійність та ефективність. Маючи невеликі часові витрати на виконання тестів, з'являється можливість підвищити частоту їх використання й швидкість локалізації проблем у системі. Є можливість налагодити автоматичні запуски тестової системи ітеративно або після розробки нових частин програмного забезпечення. Також не слід забувати про людський фактор, що може призводити до додаткових складнощів в деяких моментах тестування [7].

Класифікація за рівнем деталізації додатків (за рівнем тестування)

Модульне (компонентне) тестування (unit testing, module testing, component testing) направлено на перевірку окремих невеликих частин програми, які (як правило) можна досліджувати ізольовано від інших подібних частин. При виконанні даного тестування можуть перевірятися окремі функції або методи класів, самі класи, взаємодія класів, невеликі бібліотеки, окремі частини програми. Часто даний вид тестування реалізується з використанням спеціальних технологій та інструментальних засобів автоматизації тестування, значно спрощують і прискорюють розробку відповідних тест-кейсів.

Інтеграційне тестування (integration testing, component integration

testing, pairwise integration testing, system integration testing, interface testing, thread testing) направлено на перевірку взаємодії між декількома частинами додатка (кожна з яких, в свою чергу, перевірена окремо на стадії модульного тестування). На жаль, навіть якщо ми працюємо з дуже якісними окремими компонентами, «на стику» їх взаємодії часто виникають проблеми. Саме ці проблеми і виявляє інтеграційне тестування.

Системне тестування (system testing) направлено на перевірку всієї програми як єдиного цілого, складеного з частин, перевірених на двох попередніх стадіях. Тут не тільки виявляються дефекти «на стиках» компонентів, але і з'являється можливість повноцінно взаємодіяти з додатком з точки зору кінцевого користувача, застосовуючи безліч інших видів тестування, перерахованих в цій темі.

З класифікацією за рівнем деталізації додатку пов'язаний цікавий сумний факт: якщо попередня стадія виявила проблеми, то на наступній стадії ці проблеми точно завдадуть удар по якості; якщо ж попередня стадія перестала виявляти проблеми, це ще аж ніяк не захищає нас від проблем на наступній стадії.

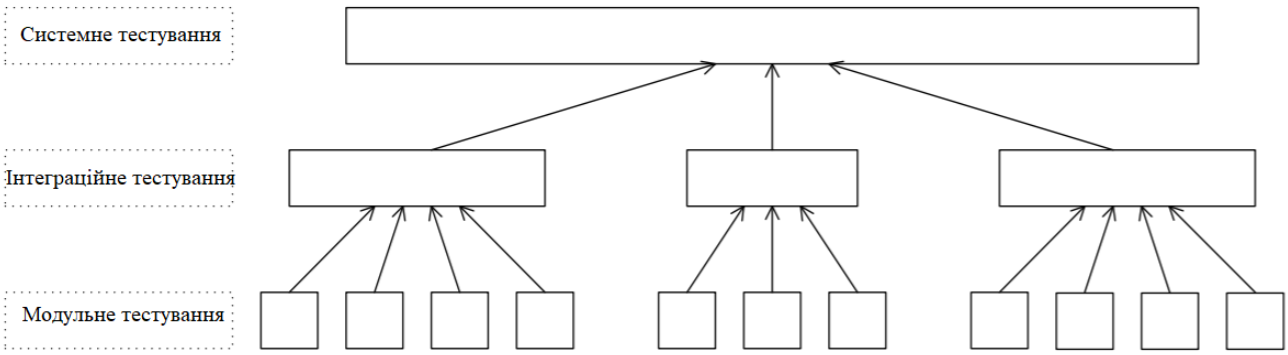

Рис. 5.2. Схематичне уявлення класифікації тестування по рівню деталізації додатку

Рис. 5.3. Рівні тестування програмного забезпечення

Альфа–тестування (Alpha testing) — це вид тестування, який включає імітацію реального використання продукту штатними тестувальниками (співробітниками компанії або групою фахівців незалежного тестування).

Зазвичай Альфа тестування передбачає системну перевірку всіх функцій програми з використанням технік тестування «білого ящика» і «чорного ящика». В той же час Альфа-тестування здійснюється без участі Dev команд девелоперів (програмістів).

Бета-тестування(beta testing) - це інтенсивне використання майже готової версії продукту з метою виявлення максимального числа помилок в його роботі для їх подальшого усунення перед остаточним виходом (релізом) продукту на ринок, до масового споживача. Бета–тестування являється реально працюючою версію програми з повним функціоналом. І завдання бета–тестів оцінити можливості і стабільність роботи програми у якості її майбутніх користувачів. На відміну від альфа–тестування, проведеного силами штатних розробників або незалежних тестувальників, бетатестування зазвичай передбачає залучення добровольців з числа звичайних майбутніх користувачів продукту, яким доступна згадана попередня версія продукту (так звана бета–версія) або ж на вибірці схожих людей.

Бета–тестування може бути:

Закритим: програма тестується в невеликій групі користувачів за запро-

шеннями.

Відкритим: цей варіант дозволяє протестувати додаток в більшій групі і отримати великий обсяг зворотного зв'язку. Будьякий користувач зможе приєднатися до відкритого бета–тестування і відправити особистий відгук.

Переваги бета–тестування:

- Знижує ризик виходу продукту з ладу за допомогою валідації клієнта.
- Бета–тестування дозволяє компанії тестувати інфраструктуру після запуску.
- Підвищує якість продукції завдяки зворотньому зв'язку з клієнтами.

Недоліки бета–тестування:

- Управління тестуванням. У порівнянні з іншими типами тестування, які зазвичай виконуються всередині компанії в контрольованому середовищі, бета–тестування виконується в реальному світі, де у компанії рідко є контроль.
- Пошук правильних користувачів бетаверсії і підтримання їх участі може викликати труднощі.

Гамма-тестування проводиться коли програмне забезпечення вже готове до релізу, перевіряється відповідність вимогам.

Класифікація за (зменшенням) ступенем важливості тестованих функцій (за рівнем функціонального тестування)

Димове тестування (smoke test) - направлено на перевірку найголовнішою, найважливішою, самої ключової функціональності, непрацездатність якої робить безглуздою саму ідею використання програми (або іншого об'єкта, що піддається димовому тестуванню);

Димове тестування проводиться після виходу нового білду, щоб визначити загальний рівень якості додатку і прийняти рішення про (не) доцільність виконання тестування критичного шляху і розширеного тестування. Оскільки тест-кейсів на рівні димового тестування відносно небагато, а самі вони досить прості, але при цьому дуже часто повторюються, вони є хорошими кандидатами на автоматизацію.

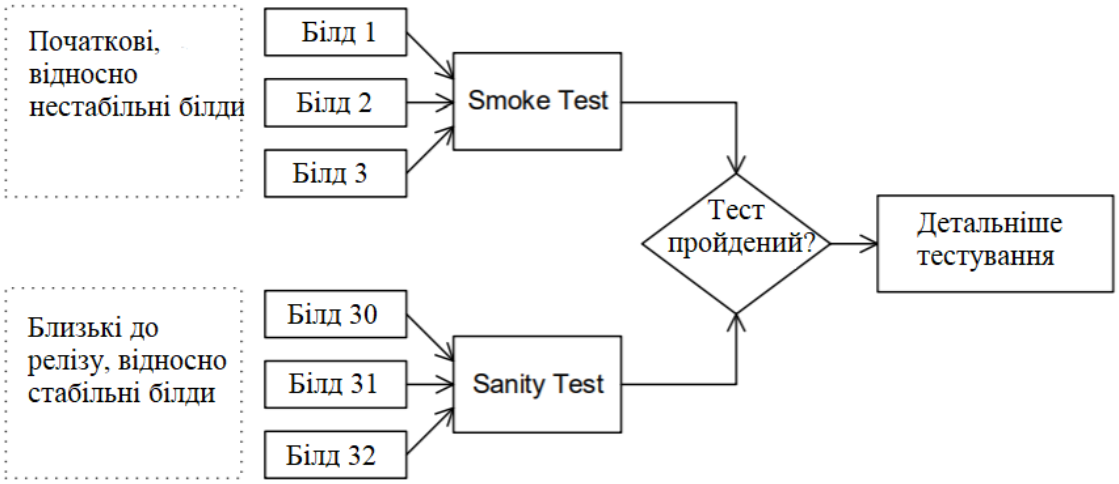

Рис. 5.4. Відмінність «smoke test» та «sanity test»

Тестування критичного шляху (critical path test) направлено на дослідження функціональності, яке використовують типові користувачі в типовій повсякденній діяльності.

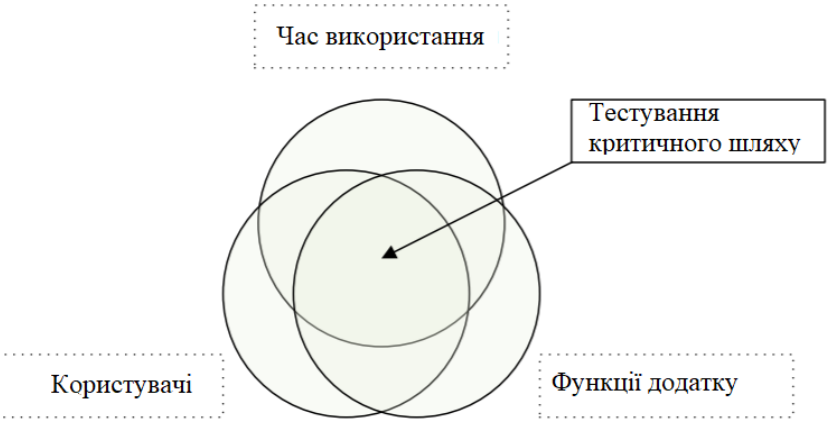

Рис. 5.5. Тестування критичного шляху (critical path test)

Розширене тестування (extended test) направлено на дослідження всієї заявленої в вимогах функціональності - навіть тієї, яка низько проранжована за ступенем важливості. При цьому тут також враховується, яка функціональність є більш важливою, а яка - менш важливою. Але при наявності достатньої кількості часу і інших ресурсів тест-кейси цього рівня можуть торкнутися навіть самих низькопріоритетних вимог.

Ще одним напрямком дослідження в рамках даного тестування є нетипові, малоймовірні, «екзотичні» випадки і сценарії використання функцій і властивостей додатку, згаданих на попередніх рівнях. Граничне значення метрики успішного проходження розширеного тестування істотно нижче, ніж в тестуванні критичного шляху (Іноді можна побачити навіть значення в діапазоні 30-50%, тому

що переважна більшість знайдених тут дефектів не становить загрози для успішного використання програми більшістю користувачів).

Рис. 5.6. Залежність рівня тестування від глибини делатізації додатку іважливості

Класифікація за принципами роботи з додатком

Позитивне тестування (positive testing) направлено на дослідження додатку в ситуації, коли всі дії виконуються строго за інструкцією без будь-яких помилок, відхилень, введення невірних даних і т.д. Якщо позитивні тест-кейси завершуються помилками, це тривожна ознака - додаток не працює належним чином навіть в ідеальних умовах (і можна припустити, що в неідеальних умовах воно працює ще гірше).

Для прискорення тестування кілька позитивних тест-кейсів можна об'єднувати (наприклад, перед відправкою заповнити всі поля форми вірними значеннями) - іноді це може ускладнити діагностику помилки, але суттєва економія часу компенсує цей ризик.

Негативне тестування (negative тестінг, invalid testing) - направлено на дослідження роботи програми в ситуаціях, коли з нею виконуються (некоректні) операції та / або використовуються дані, що потенційно призводять до помилок (класика жанру - поділ на нуль). Оскільки в реальному житті таких ситуацій значно більше (користувачі допускають помилки, зловмисники свідомо «ламають» додаток, в середовищі роботи програми виникають проблеми і т.д.), негативних тест-кейсів виявляється значно більше, ніж позитивних (іноді - в рази або навіть на порядки). На відміну від позитивних негативні тест-кейси не варто об'єднувати, тому що подібне рішення може привести до невірного трактування поведінки програми та пропуску (невиявлення) дефектів.

Класифікація за природою додатку

Даний вид класифікації є штучним, оскільки «всередині» мова буде йти про одних і тих же видах тестування, що відрізняються в даному контексті лише концентрацією на відповідних функціях і особливостях застосування, використанням специфічних інструментів та окремих технік.

• **Тестування веб-додатків (web-applications testing)** пов'язане з інтенсивною діяльністю в області тестування сумісності (особливо - крос-браузерного тестування), тестування продуктивності, автоматизації тестування з використанням широкого спектра інструментальних засобів.

• **Тестування мобільних додатків (mobile applications testing)** також вимагає підвищеної уваги до тестування сумісності, оптимізації продуктивності (в тому числі клієнтської частини з точки зору зниження енергоспоживання), автоматизації тестування із застосуванням емуляторів мобільних пристроїв.

• **Тестування настільних додатків (desktop applications testing)** є самим класичним серед всіх перерахованих в даній класифікації, і його особливості залежать від предметної області додатку, нюансів архітектури, ключових показників якості і т.д.

Класифікація за фокусуванням на рівні архітектури додатку

Даний вид класифікації, як і попередній, також є штучним і відображає лише концентрацію уваги на окремій частині програми.

• **Тестування рівня уявлення (presentation tier testing)** сконцентровано на тій частині програми, яка відповідає за взаємодію з «зовнішнім світом» (як користувачами, так і іншими додатками). Тут досліджуються питання зручності використання, швидкості відгуку інтерфейсу, сумісності з браузерами, коректності роботи інтерфейсів.

• **Тестування рівня бізнес-логіки (business logic tier testing)** відповідає за перевірку основного набору функцій програми та будується на базі ключових вимог до додатку, бізнес-правил і спільної перевірки функціональності.

• **Тестування рівня даних (data tier testing)** сконцентровано на тій частини програми, яка відповідає за зберігання і деяку обробку даних (найчастіше - в базі даних чи іншому сховище). Тут особливий інтерес представляє тестування даних, перевірка дотримання бізнес-правил, тестування продуктивності.

Класифікація за ступенем формалізації

• **Тестування на основі тест-кейсів (scripted testing, test case based testing)** - формалізований підхід, в якому тестування проводиться на основі заздалегідь підготовлених тест-кейсів, наборів тест-кейсів та іншої документації. Це найпоширеніший спосіб тестування, який також дозволяє досягти максима-

льної повноти дослідження додатку за рахунок суворої систематизації процесу, зручності застосування метрик і широкого набору вироблених за десятиліття і перевірених на практиці рекомендацій.

• **Дослідницьке тестування (exploratory testing)** - частково формалізований підхід, в рамках якого тестувальник виконує роботу з додатком за обраним сценарієм, який, в свою чергу, допрацьовується в процесі виконання з метою більш повного дослідження додатку. Ключовим фактором успіху при виконанні дослідницького тестування є саме робота за сценарієм, а не виконання розрізнених бездумних операцій. Існує навіть спеціальний сценарний підхід - сесійне тестування (session-based testing). В якості альтернативи сценаріям при виборі дій з додатком іноді можуть використовуватися чек-листи, і тоді цей вид тестування називають **тестуванням на основі чек-листів (checklist-based testing).**

• **Вільне (інтуїтивне) тестування (ad hoc testing)** - повністю неформалізований підхід, в якому не передбачається використання ні тест-кейсів, ні чек-листів, ні сценаріїв - тестувальник повністю спирається на свій професіоналізм і інтуїцію (experience-based testing) для спонтанного виконання з додатком дій, які, як він вважає, можуть виявити помилку. Цей вид тестування використовується рідко і виключно як доповнення до повністю або частково формалізованого тестування в випадках, коли для дослідження деякого аспекту поведінки додатку (поки що) немає тест-кейсів.

Інсталяційне тестування (installation testing, installability testing) - тестування, спрямоване на виявлення дефектів, що впливають на перебіг стадії інсталяції (установки) додатку. У загальному випадку таке тестування перевіряє безліч сценаріїв і аспектів роботи інсталятора в таких ситуаціях, як:

✓ нове середовище виконання, в якій додаток раніше не було інстальовано;

✓ оновлення існуючої версії («апгрейд»);

✓ зміна поточної версії на більш стару («даунгрейд»);

✓ повторне встановлення програми з метою усунення проблем, які виникли («перевстановлення»);

✓ повторний запуск інсталяції після помилки, що призвела до неможливості продовження інсталяції;

✓ щоб видалити програму;

✓ встановлення нового додатку з сімейства додатків;

✓ автоматична інсталяція без участі користувача.

Регресійне тестування (regression testing) - тестування, спрямоване на

перевірку того факту, що в раніше працездатній функціональності не з'явилися помилки, викликані змінами в додатку або середовищі його функціонування. Фредерік Брукс у своїй книзі «Міфічний людино-місяць» писав: «Фундаментальна проблема при супроводі програм полягає в тому, що виправлення однієї помилки з великою ймовірністю (20-50%) тягне поява нової». Тому регресійне тестування є невід'ємним інструментом забезпечення якості і активно використовується практично в будь-якому проекті.

Повторне тестування (re-testing, Confirmation testing) – виконання тест-кейсів, які раніше виявили дефекти, з метою підтвердження усунення дефектів. Фактично цей вид тестування зводиться до дій на фінальній стадії життєвого циклу звіту про дефект, спрямованим на те, щоб перевести дефект в стан «перевірений» і «закритий».

Приймальне тестування (acceptance testing) - формалізоване тестування, спрямоване на перевірку додатку з точки зору кінцевого користувача / замовника і винесення рішення про те, чи приймає замовник роботу у виконавця (проектної команди). Можна виділити наступні підвиди приймального тестування (хоча згадують їх вкрай рідко, обмежуючись в основному загальним терміном «приймальним тестування»):

Виробниче приймальне тестування (factory acceptance testing) - виконується проектною командою дослідження повноти і якості реалізації програми з точки зору його готовності до передачі замовнику. Цей вид тестування часто розглядається як синонім альфа-тестування.

Операційне приймальне тестування (operational acceptance testing, production acceptance testing) - операційне тестування, яке виконується з точки зору виконання інсталяції, споживання додатком ресурсів, сумісності з програмної і апаратної платформою і т.д.

Підсумкове приймальне тестування (site acceptance testing) - тестування кінцевими користувачами (представниками замовника) додатку в реальних умовам експлуатації з метою винесення рішення про те, чи потребує додаток доробок або може бути прийнято в експлуатацію в поточному вигляді.

Операційне тестування (operational testing) - тестування, що проводиться в реальному або наближеній до реальної операційному середовищі (operational environment), що включає операційну систему, системи управління базами даних, сервери додатків, веб-сервери, апаратне забезпечення і т.д.

Тестування зручності використання (usability testing) - тестування, спрямоване на дослідження того, наскільки кінцевому користувачеві зрозуміло,

як працювати з продуктом (understandability, learnability, operability), а також на те, наскільки йому подобається використовувати продукт (attractiveness). І це не обмовка - дуже часто успіх продукту залежить саме від емоцій, які він викликає у користувачів. Для ефективного проведення цього виду тестування потрібно реалізувати досить серйозні дослідження із залученням кінцевих користувачів, проведенням маркетингових досліджень і т.д

Тестування доступності (accessibility testing) - тестування, спрямоване на дослідження придатності продукту до використання людьми з обмеженим можливостями (слабким зором і т.д.).

Тестування інтерфейсу (interface testing) - тестування, спрямоване на перевірку інтерфейсів додатків або їхніх компонентів. Цей вид тестування відноситься до інтеграційного тестування, і це цілком справедливо для таких його варіацій як тестування інтерфейсу прикладного програмування (API testing) і інтерфейсу командного рядка (CLI testing), хоча воно може виступати і як різновид тестування користувальницького інтерфейсу, якщо через командний рядок з додатком взаємодіє користувач, а не інший додаток.

Тестування безпеки (security testing) - тестування, спрямоване на перевірку здатності додатку протистояти зловмисним спробам отримання доступу до даних або функцій, права на доступ до яких у зловмисника немає.

Тестування інтернаціоналізації (internationalization testing, globalization testing, localizability testing) - тестування, спрямоване на перевірку готовності продукту до роботи з використанням різних мов і з урахуванням різних національних і культурних особливостей.

Цей вид тестування не має на увазі перевірки якості відповідної адаптації (цим займається тестування локалізації), воно сфокусовано саме на перевірці можливості такої адаптації (наприклад: що буде, якщо відкрити файл з ієрогліфом в імені; як буде працювати інтерфейс, якщо все перевести на японську; чи може додаток шукати дані в тексті на корейській і т.д.).

Тестування локалізації (localization testing) - тестування, спрямоване на перевірку коректності та якості адаптації продукту до використання на тій чи іншій мові з урахуванням національних і культурних особливостей. Це тестування слідує за тестуванням інтернаціоналізації і перевіряє правильність перекладу та адаптації продукту, а не готовність продукту до таких дій.

Тестування сумісності (compatibility testing, interoperability testing) - тестування, спрямоване на перевірку здатності додатку працювати в зазначеному окрузі. Тут, наприклад, може перевірятися:

✓ Сумісність з апаратною платформою, операційною системою і мережевою інфраструктурою (конфигурационное тестування, configuration testing)

✓ Сумісність з браузерами і їх версіями (крос-браузерні тестування, cross-browser testing).

✓ Сумісність з мобільними пристроями (mobile testing).

Тестування даних (data quality testing) і баз даних (database integrity testing) - два близьких за змістом види тестування, спрямованих на дослідження таких характеристик даних, як повнота, несуперечливість, цілісність, структурованість і т.д. В контексті баз даних дослідженню може піддаватися адекватність моделі предметної області, здатність моделі забезпечувати цілісність і консистентность даних, коректність роботи тригерів, збережених процедур і т.д.

Тестування використання ресурсів (resource utilization testing, efficiency testing, storage testing) - сукупність видів тестування, перевіряючих ефективність використання додатком доступних йому ресурсів і залежність результатів роботи програми від кількості доступних йому ресурсів. Часто ці види тестування прямо або побічно примикають до технік тестування продуктивності.

Порівняльне тестування (comparison testing) - тестування, спрямоване на порівняльний аналіз переваг і недоліків продукту, що розробляється по відношенню до його основних конкурентів.

Демонстраційне тестування (qualification testing) - формальний процес демонстрації замовнику продукту з метою підтвердження, що продукт відповідає всім заявленим вимогам. На відміну від приймального тестування цей процес більш строгий і всеосяжний, але може проводитися і на проміжних стадіях розробки продукту.

Тестування надійності (reliability testing) - тестування здатності додатку виконувати свої функції в заданих умовах протягом заданого часу або при заданій кількості операцій.

Надійність ПЗ підвищується також за допомогою застосування різних методів тестування. Повне тестування ПЗ неможливо. Зазвичай застосовують такі види тестування:

- тестування гілок;
- математичний доказ правильності алгоритму розв'язання задачі (в деяких роботах саме в цьому сенсі вживається слово верифікація);
- символічне тестування (або з допомогою спеціально підібраних тестових наборів), ще називається статичним тестуванням. Зручно при локалізації помилки, прояв якої виявлено при конкретному вузькому або строго заданому діапазо-

ні вхідних значень;

- динамічне тестування (за допомогою динамічно генерованих вхідних даних), що зручно при швидкому тестуванні у всьому широкому діапазоні вхідних параметрів;
- тестування шляхів виконання програми;
- функціональне тестування;
- перевірки за часом виконання програми;
- перевірка по використанню ресурсів та стресове тестування.

Основні показники надійності ПЗ:

1. **Ймовірність безвідмовної роботи $P(t_з)$** - це ймовірність того, що в межах заданого напрацювання відмова системи не виникає.

2**. Імовірність відмови** - ймовірність того, що в межах заданого напрацювання відмова системи виникає. Це показник, зворотний попередньому.

$$Q(t_з) = 1 - P(t_з), \tag{5.1}$$

де $t_з$ - задане напрацювання, год; $Q(t_з)$ - ймовірність відмови.

3. Інтенсивність відмов системи - це умовна щільність ймовірності виникнення відмови ПЗ в певний момент часу за умови, що до цього часу відмова не виникла.

$$Q(t) = \frac{f(t)}{P(t)}, \tag{5.2}$$

Де $f(t)$ - щільність імовірності відмови в момент часу t.

$$f(t) = \frac{d}{dt}Q(t) = \frac{d}{dt}(1 - P(t)) = \frac{d}{dt}p(t), \tag{5.3}$$

Існує наступний зв’язок між інтенсивністю відмов системи і ймовірністю безвідмовної роботи:

$$P(t) = e - \int_0^t Q(t)dt, \tag{5.4}$$

В окремому випадку, при

$$P(t) = \exp(-Q(t)), \tag{5.5}$$

$$Q(t) = const \tag{5.6}$$

Якщо в процесі тестування фіксується число відмов за певний часовий інтервал, то інтенсивність відмов системи є число відмов в одиницю часу.

4. Середнє напрацювання на відмову T_i - математичне очікування часу роботи ПЗ до чергової відмови:

$$T_i = \int_0^t t * f(t)dt, \tag{5.7}$$

Інакше середню напрацювання на відмові Ti можна представити i=I;

$$T_i = \frac{t_1 + t_2 + t_3 .... t_n}{n} = \left|\frac{i}{n}\right| * \sum_{i=1}^{n} t_i, \tag{5.8}$$

де t – час роботи ПЗ між відмовами, с.
n – кількість відмов.

5. Середній час відновлення T - математичне очікування часу відновлення - t; часу, витраченого на виявлення і локалізацію відмови – t_1; часу усунення відмови – t_2; часу пропускної перевірки працездатності – t_3:

$$t = t_1 + t_2 + t_3,$$

де t_i - час відновлення після i-ї відмови.

$$T = i / nt,$$

$$i = 1,$$

де n - кількість відмов. Для цього показника термін "час" означає час, витрачений фахівцем з тестування на перелічені види робіт.

6. Коефіцієнт готовності K - ймовірність того, що ПЗ очікується в працездатному стані в довільний момент часу його використання за призначенням:

$$K = T / (T + T)$$

Необхідно прагнути підвищувати рівень надійності ПЗ, але досягнення 100 % надійності лежить за межами можливого. Кількісні показники надійності можуть використовуватися для оцінки досягнутого рівня технології програмування, для вибору методу проектування майбутнього програмного засобу.

Основним засобом визначення кількісних показників надійності є моделі надійності, під якими розуміють математичну модель, побудовану для оцінки залежності надійності від заздалегідь відомих або оцінених в ході створення програмних засобів параметрів.

Всі наведені показники надійності ПЗ характеризують наявність помилок програми (виробничих дефектів), але жоден з них не характеризує характер цих помилок і можливі їх наслідки.

Одним із шляхів підвищення рівня надійності ПЗ є використання на етапах тестування і випробувань ПЗ моделей, що дають змогу отримати гарантовані оцінки показників безпеки ПЗ і ефективності технології його розроблення. Більшість таких моделей запозичене з теорії надійності технічних систем, тому в літературі їх часто називають моделями надійності ПЗ.

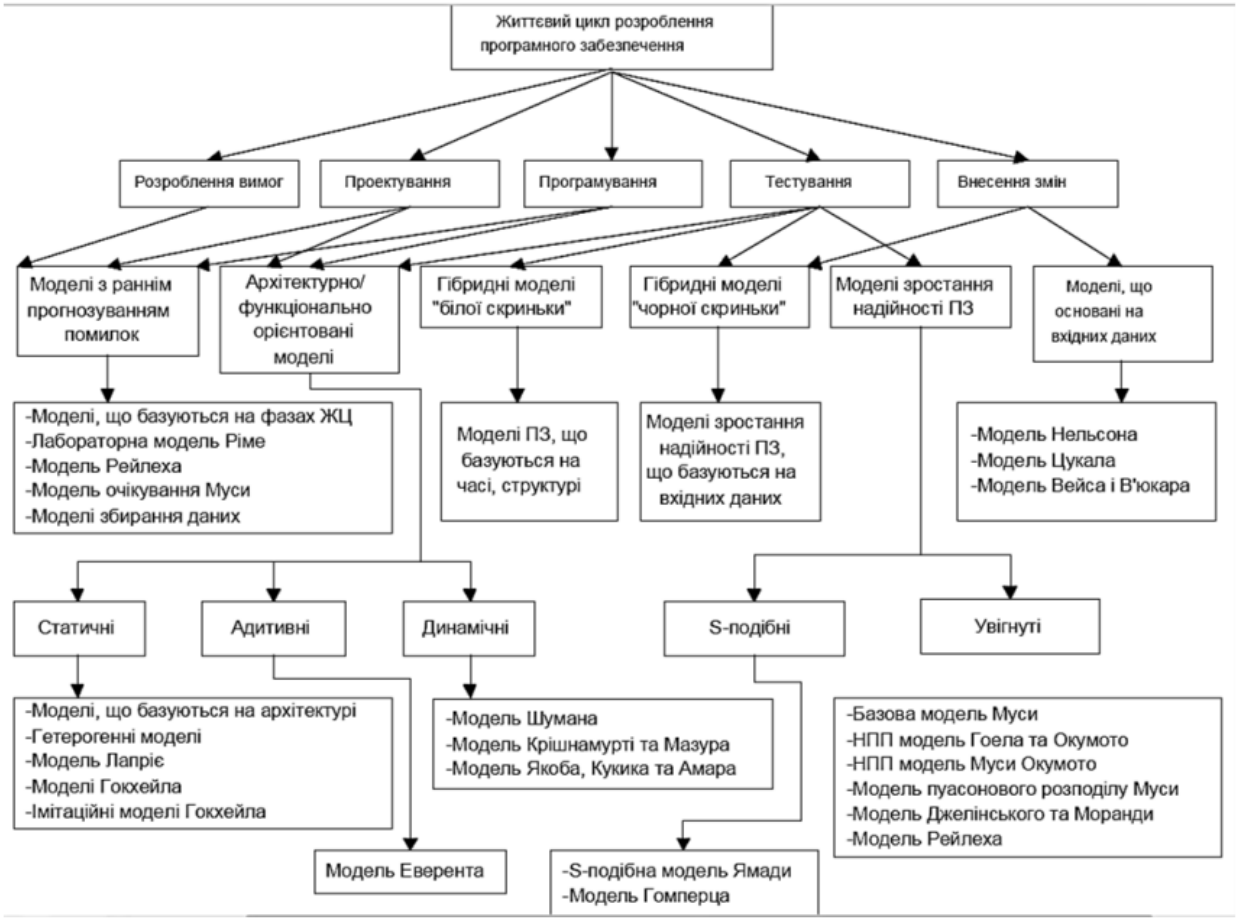

Рис. 5.7. Класифікація моделей надійності ПЗ та їх взаємозв'язок зжиттєвим циклом ПЗ

На ранніх стадіях життєвого циклу ПЗ потрібна модель прогнозування надійності, оскільки даних про відмови немає. Моделі такого типу призначені для передбачення кількості помилок у програмі перед тестуванням, і в деяких літературних джерелах належать до детерміністичних (статичних) моделей надійності ПЗ.

На етапі тестування показники надійності ПЗ покращуються завдяки відлагодженню програми. Модель зростання надійності на цьому етапі потрібна для оцінювання поточного рівня надійності, часу і ресурсів, потрібних для досягнення заданого рівня надійності ПЗ. Впродовж цього етапу оцінка надійності ґрунтується на аналізі цих відмов. Моделі такого типу належать до імовірнісних (динамічних) моделей надійності ПЗ.

Після введення програми в експлуатацію при визначенні її надійності необхідно враховувати додавання нових 82 модулів, усунення старих модулів, усунення виявлених помилок, поєднання нового коду з попередньо написаним кодом, зміну середовища користувача, зміну апаратного забезпечення тощо [8].

Тестування відновлюваності (recoverability testing) - тестування здатності додатку відновлювати свої функції і заданий рівень продуктивності, а також відновлювати дані в разі виникнення критичної ситуації, що призводить до тимчасової (часткової) втрати працездатності додатку.

Тестування відмовостійкості (failover testing) - тестування, яке полягає в емуляції або реальному створенні критичних ситуацій з метою перевірки здатності додатки задіяти відповідні механізми, що запобігають порушення працездатності, продуктивності і пошкодження даних.

Тестування продуктивності (performance testing) - дослідження показників швидкості реакції програми на зовнішні впливи при різних за характером і інтенсивністю навантаженнях. В рамках тестування продуктивності виділяють наступні підвиди:

Навантажувальне тестування (load testing, capacity testing) - дослідження здатності додатка зберігати задані показники якості при навантаженні в допустимих межах і деякому перевищенні цих меж (визначення «запасу міцності»).

Тестування масштабованості (scalability testing) - дослідження здатності додатку збільшувати показники продуктивності відповідно до збільшення кількості доступних додатком ресурсів.

Об'ємне тестування (volume testing) - дослідження продуктивності додатка при обробці різних (як правило, великих) обсягів даних.

Стресове тестування (stress testing) - дослідження поведінки додатку при нештатних змінах навантаження, які значно перевищують розрахунковий рівень, або в ситуаціях недоступності значної частини необхідних додатку ресурсів.

Конкурентне тестування (concurrency testing) – дослідження поведінки додатка в ситуації, коли йому доводиться обробляти велику кількість одночасно запитів, що викликає конкуренцію між запитами за ресурси (бази даних, пам'ять, канал передачі даних, дискову підсистему і т.д.). Іноді під конкурентним тестуванням розуміють також дослідження роботи багатопоточних додатків і коректність синхронізації дій, вироблених в різних потоках.

Класифікація по технікам автоматизації:

Тестування під керуванням даними (data-driven testing) - спосіб розробки автоматизованих тест-кейсів, в якому вхідні дані і очікувані результати виносяться за межі тесткейса і зберігаються поза ним - в файлі, базі даних і т.д.

Тестування під управлінням ключовими словами (keyworddriven testing) - спосіб розробки автоматизованих тест-кейсів, в якому за межі тесткей-

са виноситься не тільки набір вхідних даних і очікуваних результатів, а й логіка поведінки тесткейса, яка описується ключовими словами (командами).

Тестування під управлінням поведінкою (behavior-driven testing) - спосіб розробки автоматизованих тест-кейсів, в якому основна увага приділяється коректності роботи бізнес-сценаріїв, а не окремих деталей функціонування програми.

Класифікація на основі середовища виконання:

Тестування в процесі розробки (development testing) - тестування, яке виконує безпосередньо в процесі розробки додатки і / або в середовищі виконання, відмінною від середовища реального використання програми. Як правило, виконується самими розробниками.

Тестування на основі коду (code based testing). У різних джерелах цю техніку називають по-різному (найчастіше - тестуванням на основі структур, причому деякі автори змішують в один набір тестування по потоку управління і по потоку даних, а деякі строго поділяють ці стратегії). Підвиди цієї техніки також організують в різні комбінації, але найбільш універсально їх можна класифікувати так:

Тестування по потоку управління (control flow testing) - сімейство технік тестування, в яких тест-кейси розробляються з метою активації та перевірки виконання різних послідовностей подій, які визначаються за допомогою аналізу вихідного коду програми.

Тестування по потоку даних (data-flow testing) – сімейство технік тестування, заснованих на виборі окремих шляхів з потоку управління з метою дослідження подій, пов'язаних зі зміною стану змінних.

Тестування по діаграмі або таблиці станів (state transition testing) - техніка тестування, в якій тест-кейси розробляються для перевірки переходів додатки з одного стану в інший. Стану можуть бути описані діаграмою станів (state diagram) або таблицею станів (state table).

Іноді цю техніку тестування також називають «тестуванням з принципом кінцевого автомата»(finite state machine testing). Важливою перевагою цієї техніки є можливість застосування в ній теорії кінцевих автоматів (яка добре формалізована), а також можливість використання автоматизації для генерації комбінацій вхідних даних.

Інспекція (аудит) коду (code review, code inspection) – сімейство технік підвищення якості коду за рахунок того, що в процесі створення або вдосконалення коду беруть участь кілька людей. Ступінь формалізації аудиту коду може

варіюватися від досить побіжного перегляду до ретельної формальної інспекції. На відміну від технік статичного аналізу коду (по потоку управління і потоку даних) аудит коду також покращує такі його характеристики, як зрозумілість, підтримуваність, відповідність угодам про оформлення і т.д. Аудит коду виконується в основному самими програмістами.

Тестування на основі (моделей) поведінки додатку (application behavior / model-based testing):

Тестування з таблиці прийняття рішень (decision table testing) - техніка тестування (за методом чорного ящика), в якій тест-кейси розробляються на основі т.зв. таблиці прийняття рішень, в якій відображені вхідні дані (і їх комбінації) і впливу на додаток, а також відповідні їм вихідні дані і реакції додатки.

Тестування по діаграмі або таблиці станів (розглянуто раніше).

Тестування за специфікаціями (specification-based testing, black box testing) (розглянуто раніше).

Тестування за моделями поведінки додатки (model-based testing) - техніка тестування, в якій дослідження додатки (і розробка тест-кейсів) будується на якійсь моделі: таблиці прийняття рішень, Таблиці або діаграми станів, Призначених для користувача сценаріїв, Моделі навантаження і т.д.

Тестування на основі варіантів використання (use case testing) - техніка тестування (за методом чорного ящика), в якій тест-кейси розробляються на основі варіантів використання. Варіанти використання виступають в основному джерелом інформації для кроків тест-кейса, в той час як набори вхідних даних зручно розробляти за допомогою технік вибору вхідних даних.

Паралельне тестування (parallel testing) - техніка тестування, в якій поведінка нового (або модифікованого) додатки порівнюється з поведінкою еталонного додатки (імовірно працює вірно). Термін «паралельне тестування» також може використовуватися для позначення способу проведення тестування, коли кілька тестувальників або систем автоматизації виконують роботу одночасно, тобто паралельно. Дуже рідко (і не зовсім вірно) під паралельним тестуванням розуміють мутаційне тестування.

Тестування на основі випадкових даних (random testing) - техніка тестування (за методом чорного ящика), в якій вхідні дані, дії або навіть самі тест-кейси вибираються на основі (псевдо) випадкових значень так, щоб відповідати операційного профілю (operational profile) - підмножині дій, відповідних певній ситуації або сценарієм роботи з додатком.

ТЕМА 6. ЧЕК-ЛИСТИ, ТЕСТ-КЕЙСИ, НАБОРИ ТЕСТ-КЕЙСІВ

Чек-лист (checklist) - набір ідей [тест-кейсів]. Останнє слово не дарма взято в дужки, тому що в загальному випадку **чек-лист** - це просто набір ідей: ідей з тестування, ідей з розробки, ідей з планування та управління - будь-яких ідей.

Чек-лист найчастіше представляє собою звичайний і звичний нам список:

• в якому послідовність пунктів не має значення (наприклад, список значень якогось поля);

• в якому послідовність пунктів важлива (наприклад, кроки в короткій інструкції);

• структурований (багаторівневий) список, який дозволяє відобразити ієрархію ідей.

Працюючи над проектом по чек-листі, виключена імовірність повторної перевірки по тим самим тест-кейсам, а також підвищується якість тестування, так як імовірність залишити без уваги якийсь функціонал значно зменшується. Тому дуже важливо знати з яких елементів складається чек-лист та вміти ним ефективно користуватись.

Пункти можуть містити як лінійну, так і деревовидну, з розділами/підрозділами структуру. Пункти повинні бути однозначні, щоб їх не можна було трактувати ніяким іншим чином. Всі пункти повинні бути оформлені на одній мові: англійській чи українській. Як правило, кожний чек-лист має декілька стовпців. Кожний стовпець призначений для тестування на окремій платформі.

Переваги використання чек-листів:

- використання чек-листів сприяє структуруванню інформації у співробітника;

- при правильному записі необхідних дій у співробітника з'являється однозначне розуміння задач. Це сприяє підвищенню швидкості навчання нових співробітників;

- чек-листи допомагають уникнути невизначеності і помилок пов’язаних з людським фактором. Збільшується покриття тестами програмного продукту;

- підвищується степінь взаємозамінності співробітників;

- економія робочого часу. Написав чек-лист один раз і його можна використовувати повторно, враховуючи актуальність інформації;

Використання чек-листів – один із прийомів підвищення бас фактора. В

області розробки програмного забезпечення бас фактор проекту – це міра зосередження інформації серед окремих членів команди [9].

Немає і не може бути ніяких заборон і обмежень при розробці чек-листів - головне, щоб вони допомагали в роботі.Іноді чек-листи можуть навіть виражатися графічно (наприклад, з використанням ментальних карт або концепт-карт), хоча зазвичай їх складають у вигляді багаторівневих списків.Оскільки в різних проектах зустрічаються однотипні завдан-ня, добре продумані і акуратно оформлені чек-листи можутьвикористовуватися повторно, чим досягається економія силі часу.

Mind Map (ментальна карта) – це відображення ефективного способу думати, запам'ятовувати, згадувати, вирішувати творчі завдання, а також можливість представити і наочно висловити свої внутрішні процеси обробки інформації, вносити в них зміни, вдосконалювати.

Інтелект-карти – це інструмент, що дозволяє:

- простіше працювати з інформацією: запам'ятовувати, розуміти, відновлювати логіку;
- зручно використовувати для презентації матеріалу і наочного пояснення своєї позиції співрозмовникам;
- дозволяє простіше приймати рішення, створювати плани, розробляти проекти.

Зміст Mind Map полягає у декомпозиції основної концепції або кінцевої цілі з метою її кращого розуміння і її покрокового досягнення відповідно. Це свого роду засіб організації інформації з урахуванням взаємозв’язків, який допомагає створити цілісну й систематизовану картину проблеми. А виглядає він деревовидною схемою, на якій в центрі розташовується основна система, а всі відгалуження є її підсистемами.

Для того, щоб такий поділ був ефективним, варто притримуватись наступних правил:

1) Система розподіляється на всіх рівнях по єдиному принципу. Тобто всі підсистеми мають відповідати на одне і теж питання по відношенню до «материнської» категорії. Якщо це зробити складно, що часто буває у випадку тестування програмного забезпечення, то доцільно застосовувати це правило для елементів, які знаходяться на одному рівні.

2) Підсистеми мають взаємно виключати одна одну, а разом створювати цілісний продукт. Підсистеми мають взаємодіяти між собою. Якщо якісь елементи складно згрупувати, допускається їх виокремлення в категорію «Інше».

3) На кожному рівні доцільно виділяти 5-7 підсистем. У такому разі інформація представлена в максимально наочному вигляді. При цьому не виникає плутанини і перенасичення схеми.

4) Глибина декомпозиції визначається рівнем кваліфікації і досвідченості кожного конкретного спеціаліста. Якщо з таким проектом тестувальник працює не вперше, то рівнів інтелект-карти буде значно менше. Якщо ж це зовсім нова система для працівника, то ступінь деталізації повинна бути максимальною до виокремлення найменших елементів.

5) Взаємопов'язані завдання доцільно групувати за допомогою виділення одним кольором або виділення їх у блоки.

При використанні ментальних карт команда може виявити, що насправді завдань дуже багато, тому використовують пріоритети, щоб уникнути відтермінування проекту та збільшення кількості працівників. У випадку тестування завдання оцінюються згідно таких категорій:

Вимоги клієнта: замовник завжди орієнтується на просування продукту на ринок і свій прибуток, тому врахування його побажань обов'язкове.

Рівень ризику: найбільш важливий функціонал, збій в роботі якого, завдасть найбільшу шкоду роботі програмного забезпечення та фінансові втрати замовнику, а отже потрібно тестувати в першу чергу.

Складність системи: тестування необхідно розпочинати з найскладнішого функціоналу. Це дозволяє заощадити час та уникнути надлишкових витрат. Обмеження у часі: потрібно тестувати у повній мірі лише той функціонал, який запланований на наступний реліз. На інтелект-картах можна створити спеціальні позначки для пріоритетних завдань, виділяти їх іншим кольором тощо.

MindMap в тестуванні

Зазвичай на проектах використовуються чек-листи, однак для командної роботи вони є не досить зручними, тому часто їх заміняють на інтелектуальні карти. Основними перевагами цих карт є:

1. На інтелект-карті можна побачити цілісну картину та прослідкувати взаємозв'язки між елементами.

2. Мозок користувача ментальної карти може сконцентруватися на першочергових завданнях, розуміючи логіку пріоритезації.

3. Наочно помітні накладки, протиріччя, «вузькі місця» проекту.

4. Зручність відстеження пройдених етапів.

5. Можливість вносити зміни до проекту, редагувати його чи доповнювати. При виявленні нової інформації достатньо домалювати додаткові гілки чи блоки.

6. Відсутність єдиних правил оформлення та структурування.

7. Стимулювання розумової активності, творчого пошуку нових рішень [9].

Властивості чек-листів

- **Логічність.** Чек-лист пишеться не «просто так», а на основі цілей і для того, щоб допомогти в досягненні цих цілей. На жаль, однією з найчастіших і небезпечних помилок при складанні чек-листа є перетворення його в смітник думок, які ніяк не пов'язані один з одним.
- **Послідовність і структурованість.** З структурованістю все досить просто - вона досягається за рахунок оформлення чек-листа у вигляді багаторівневого списку. Щодо послідовності, то навіть в тому випадку, коли пункти чек-листа не описують ланцюжок дій, людині все одно зручніше сприймати інформацію у вигляді якихось невеликих груп ідей, перехід між якими є зрозумілим і очевидним.
- **Повнота і ненадмірність.** Чек-лист повинен являти собою акуратну «суху вижимку» ідей, в яких немає дублювання (часто з'являється через різні формулювань однієї і тієї ж ідеї), і в той же час ніщо важливе не втрачено.

Правильно створювати і оформляти чек-листи також допомагає сприйняття їх не тільки як сховища наборів ідей, але і як «вимоги для складання тесткейсів».

Оскільки ми не можемо відразу «протестувати весь додаток» (це занадто велике завдання, щоб розв'язати цю проблему одним махом), нам вже зараз потрібно вибрати якусь логіку побудови чек-листів - так, їх буде кілька (в результаті їх можна буде структуровано об'єднати в один , але це не обов'язково).

Типовими варіантами такої логіки є створення окремих чек-листів для:

- типових користувальницьких сценаріїв;
- різних рівнів функціонального тестування;
- окремих частин (модулів і подмодулей) додатку;
- окремих вимог, груп вимог, рівнів і типів вимог;
- частин або функцій програми, які мають ризики.

Цей список можна розширювати і доповнювати, можна комбінувати його пункти, отримуючи, наприклад, чек-листи для перевірки найбільш типових сценаріїв, які зачіпають якусь частину програми.

Для того, щоб проілюструвати принципи побудови чек-листів, скористаємося логікою розбиття функцій програми за ступенем їх важливості на три кате-

горії:

• Базові функції, без яких існування додатки втрачає сенс (тобто найважливіші - то, заради чого додаток взагалі створювався), або порушення роботи яких створює об'єктивні серйозні проблеми для середовища виконання.

• Функції, затребувані більшістю користувачів в їх повсякденній роботі.

• Решта функцій (різноманітні «дрібниці», проблеми з якими не сильно вплинуть на цінність додатку для кінцевого користувача.

Функції, без яких існування додатку втрачає сенс

- ❑ Конфігурація і запуск.
- ❑ Обробка файлів.
- ❑ Зупинка.

Конфігурація і запуск. Якщо додаток неможливо налаштувати для роботи в призначеній для користувача середовищі, він марний. Якщо програма не запускається, воно марно. Якщо на стадії запуску виникають проблеми, вони можуть негативно позначитися на функціонуванні додатку і тому також заслуговують на пильну увагу.

Обробка файлів. Заради цього додаток і розроблявся, тому тут навіть на стадії створення чек-листа можна створити матрицю, яка відображатиме всі можливі комбінації допустимих форматів і допустимих кодувань вхідних файлів, щоб нічого не забути і підкреслити важливість відповідних перевірок.

Зупинка. З точки зору користувача ця функція може не здаватися такою вже важливою, але зупинка (і запуск) будь-якої програми пов'язані з великою кількістю системних операцій, проблеми з якими можуть привести до безлічі серйозних наслідків (аж до неможливості повторного запуску програми або порушення роботи операційної системи).

Функції, затребувані більшістю користувачів

Перевірка того, як додаток поводиться в звичайному повсякденному житті, поки не зачіпаючи «екзотичні» ситуації.

Тест-кейс і його життєвий цикл

Тест (test) - набір з одного або декількох тест-кейсів.

Тест-кейс (test case) - набір вхідних даних, умов виконання та очікуваних результатів, розроблений з метою перевірки тієї чи іншої властивості або поведінки програмного засобу.

Під тест-кейсом також може розумітися відповідний документ, який представляє формальний запис тест-кейсу.

Високорівневий тест-кейс (high level test case) - тест-кейс без конкретних вхідних даних і очікуваних результатів.

Як правило, обмежується загальними ідеями і операціями, схожий за своєю суттю з докладно описаними пунктами чек-листа. Досить часто зустрічається в інтеграційному тестуванні і системному тестуванні, а також на рівні димового тестування. Може служити відправною точкою для проведення дослідного тестування або для створення низькорівневих тест-кейсів.

Низькорівневий тест-кейс (low level test case) - тест-кейс з конкретними вхідними даними і очікуваними результатами.

Являє собою «повністю готовий до виконання» тест-кейс і взагалі є найбільш класичним видом тест-кейсів. Початківців тестувальників найчастіше вчать писати саме такі тести, тому що прописати всі дані докладно - набагато простіше, ніж зрозуміти, якою інформацією можна знехтувати, при цьому не знизивши цінність тест-кейсу.

Специфікація тест-кейсу (test case specification) - документ, що описує набір тест-кейсів (включаючи їх цілі, вхідні дані, умови і кроки виконання, очікувані результати) для тестованого елементу (test item, test object).

Специфікація тесту (test specification) - документ, що складається з специфікації тест-дизайну (test design specification), специфікації тест-кейсу (тест case specification) і / або специфікації тест-процедури (test procedure specification).

Тест-сценарій (test scenario, test procedure specification, test script) - документ, що описує послідовність дій по виконанню тесту (також відомий як «тест-скрипт»).

Тестування можна проводити і без тест-кейсів.

Наявність же тест-кейсів дозволяє:

• Структурувати і систематизувати підхід до тестування (без чого великий проект майже гарантовано приречений на провал).

• Обчислювати метрики тестового покриття (test coverage metrics) і вживати заходів по його збільшенню (тест-кейси тут є головним джерелом інформації, без якого існування подібних метрик втрачає сенс).

• Відстежувати відповідність поточної ситуації планом (скільки приблизно знадобиться тест-кейсів, скільки вже є, скільки виконано із запланованого на даному етапі кількості і т.д.).

• Уточнити взаєморозуміння між замовником, розробниками і тестувальниками (тест-кейси часто набагато більш наочно показують поведінку додатку, ніж це відображено у вимогах).

• Зберігати інформацію для тривалого використання і обміну досвідом між співробітниками і командами (або як мінімум - не намагатися утримати в голові сотні сторінок тексту).

• Проводити регресійне тестування і повторне тестування (які без тест-кейсів було б взагалі неможливо виконати).

• Підвищувати якість вимог.

• Швидко вводити в курс справи нового співробітника, недавно підключився до проекту.

Життєвий цикл тест-кейсу

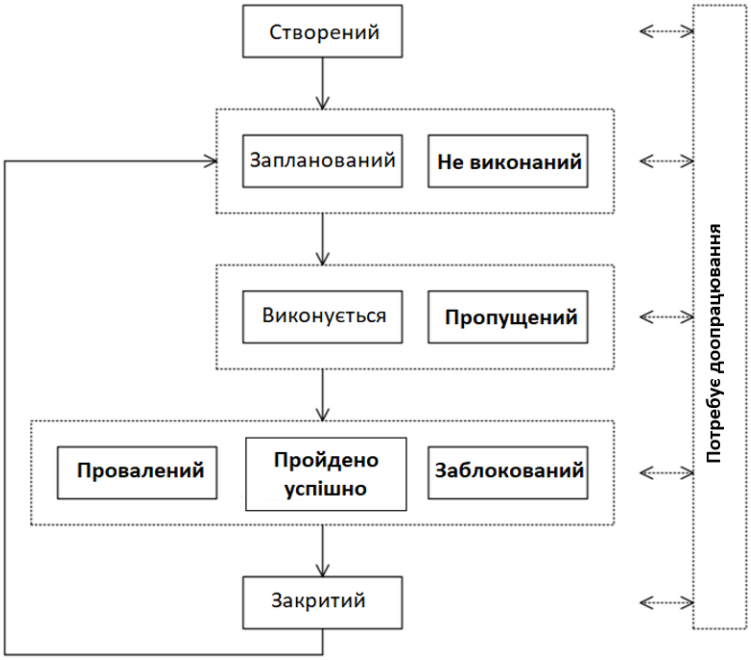

Рис. 6.1 Життєвий цикл тест-кейсу

• **Створено (new)** - типовий початковий стан практично будь-якого артефакту. Тест-кейс автоматично переходить в цей стан після створення.

• **Запланований (planned, ready for testing)** - в цьому стані тест-кейс знаходиться, коли він або явно включений в план найближчої ітерації тестування, або як мінімум готовий для виконання.

Не виконаний (not tested) - в деяких системах управління тест-кейсами це стан замінює собою попереднє («запланований»). Знаходження Тест-кейса в даному стані означає, що він готовий до виконання, але ще не був виконаний.

• **Виконується (work in progress)** - якщо тест-кейс вимагає тривалого часу на виконання, він може бути переведений в цей стан для підкреслення того факту, що робота йде, і скоро можна очікувати її результатів.

Якщо виконання тест-кейса займає мало часу, це стан, як правило, пропус-

кається, а тест-кейс відразу переводиться в одне з трьох наступних станів - «провалений», «пройдено успішно» або «заблокований».

• **Відсутній (skipped)** - бувають ситуації, коли виконання тест-кейса скасовується з міркувань браку часу або зміни логіки тестування.

• **Провалився (failed)** - це стан означає, що в процесі виконання Тест-кейса був виявлений дефект, що полягає в тому, що очікуваний результат по як мінімум одному кроці тест-кейса не збігається з фактичним результатом. Якщо в процесі виконання тест-кейсу був «випадково» виявлений дефект, ніяк не пов'язаний з кроками тест-кейса і їх очікуваними результатами, тест-кейс вважається пройденим успішно (при цьому, природно, по виявленому дефекту створюється звіт про дефект).

• **Пройдено успішно (passed)** - це стан означає, що в процесі виконання тест-кейсу не було виявлено дефектів, пов'язаних з розбіжністю очікуваних і фактичних результатів його кроків.

• **Заблоковано (blocked)** - це стан означає, що з якоїсь причини виконання тест-кейсу неможливо (як правило, такою причиною є наявність дефекту, що не дозволяє реалізувати якийсь користувальницький сценарій).

• **Закрито (closed)** - дуже рідкісний випадок, тому що тест-кейс, як правило, залишають в станах «провалений / пройдено успішно / заблокований / пропущений».

В даний стан в деяких системах управління тест-кейс переводять, щоб підкреслити той факт, що на даній ітерації тестування всі дії з ними завершено.

• **Потребує доопрацювання (not ready)** - як видно зі схеми, в цей стан (і з нього) тест-кейс може бути преведён в будь-який момент часу, якщо в ньому буде виявлена помилка, якщо зміняться вимоги, за якими він був написаний, або настане інша ситуація, що не дозволяє вважати тест-кейс придатним для виконання і переводу в інші стани.

На відміну від життєвого циклу дефекту, який досить стандартизований і формалізований, для тест-кейсу описане вище носить загальний рекомендаційний характер, розглядається скоріше як розрізнений набір станів (а не строгий життєвий цикл) і може сильно відрізнятися в різних компаніях (в зв'язку з наявними традиціями і / або можливостями систем управління тест-кейсами).

Атрибути (поля) тест-кейсу

Атрибути тест-кейсу - це конкретні поля чи атрибути, які визначають його характеристики та параметри.

Рис. 6.2. Атрибути (поля) тест-кейсу

Ідентифікатор (identifier) представляє собою унікальне значення, котре дозволяє однозначно відрізнити один тест-кейс від іншого і використовується у все можливих посиланнях. В загальному випадку ідентифікатор тест-кейсу 160 може представляти собою просто унікальний номер, але може бути і більш складніше представлення.

Пріоритет (priority) показує важливість тест-кейса. Він може бути виражений літерами, цифрами, словами чи іншим зручним способом. Кількість градацій так само не фіксоване, але частіше варіюється від 3 до 5.

Пріоритет тест-кейсу може корелюватись з:

- важливістю вимог, користувацького сценарію чи функції, з якою пов'язаний тест-кейс;

- потенційною важливістю дефекту, на пошук якого направлений тест-кейс;

- ступенем ризику, пов’язаною з перевірочним тест-кейсом вимогою, сценарієм чи функцією.

Пов’язана з тест-кейсом вимога (requirement) показує ту основну вимогу, перевірці виконання котрої присвячений тест-кейс (основну – адже один тест-кейс може зачіпати декілька вимог).

Модуль та підмодуль додатку (module and submodule) вказують на частини додатку, до яких відноситься тест-кейс, і дозволяє краще розуміти його ціль.

Заголовок (зміст) тест-кейсу (title) покликаний спростити та пришвидшити розуміння основної ідеї (цілі) тест-кейса без звернення до його основних атрибутів. Саме це поле є найбільш інформативним при перегляді списку тест-кейсів.

Заголовок тест-кейсу може бути повноцінним реченням, фразою, набором словосполучень – головне, щоб виконувались наступні вимоги:

- інформативність;
- хоча б відносна унікальність.

Вихідні дані, необхідні для виконання тест-кейса (precondition, preparation, initital data, setup) дозволяють описати все те, що повинне бути підготовлене до початку виконання тест-кейса, наприклад:

- стан бази даних;
- стан файлової системи та її об'єктів;
- стан серверів та мережевої інфраструктури тощо.

Кроки тест-кейса (steps) описують послідовність дій, котрі необхідно реалізувати в процесі виконання тест-кейсу. Загальні рекомендації по написанню кроків такі:

- починати із зрозумілого та явного місця, не пишіть зайвих початкових кроків (запуск додатку, явні операції із інтерфейсом тощо);
- навіть якщо в тест-кейсі один крок, нумерувати його, інакше збільшується імовірність в майбутньому випадково «приклеїти» опис цього кроку до нового тексту);
- співвідносити степінь деталізації кроків та їх параметрів з ціллю тесткейсу, його складністю, рівнем тощо;
- посилайтися на попередні кроки та їх діапазони для зменшення об'єму тексту (наприклад «повтори кроки 3-5 зі значеннями…»);
- писати кроки послідовно, без умовних конструкцій виду «якщо..то..».

Очікувані результати (expected results) по кожному кроці тест-кейсу описують реакцію додатку на дії, описані в полі «Кроки тест-кейсу». Номер кроку відповідає номеру результату. По написанню очікуваних результатів можна рекомендувати наступне:

- описувати поведінку системи так, щоб виключити суб'єктивне трактування (наприклад, «додаток працює вірно» - погано, «з'являється вікно з написом…» - добре);
- писати очікуваний результат по всім крокам без виключення, якщо навіть у вас є хоч маленький сумнів в тому, що результат деякого кроку буде ціл-

ком тривіальним та очевидним;

- писати коротко, але не в збиток інформативності;
- уникати умовних конструкцій виду «якщо…то…» [9].

ТЕМА 7. ВЛАСТИВОСТІ ЯКІСНИХ ТЕСТ-КЕЙСІВ

Якісні тест-кейси є важливою частиною процесу тестування, оскільки вони визначають якість тестування та його покриття функціональності. Властивості якісних тест-кейсів включають такі аспекти:

1. Коректність (Correctness): Тест-кейс повинен бути правильною інструкцією для тестера, що дозволяє йому визначити, чи відповідає програмне забезпечення вимогам.

2. Чіткість та Зрозумілість (Clarity and Understandability): Інструкції в тест-кейсі повинні бути зрозумілими та легкими для читання. Це допомагає уникнути непорозумінь та сприяє відзвітуванню.

3. Повторюваність (Repeatability): Тест-кейс повинен бути консистентним і повторюваним. Його виконання повинно давати однакові результати в однакових умовах.

4. Відтворюваність (Reproducibility): Якщо тест-кейс виявляє дефект чи проблему, він повинен бути достатньо деталізованим, щоб інші тестери могли відтворити цю проблему.

5. Покриття (Coverage): Тест-кейс повинен враховувати всі можливі стани, входи та шляхи виконання для певного аспекту програми. Це допомагає забезпечити високий рівень покриття.

6. Придатність до автоматизації (Automatability): Якщо можливо, тест-кейс повинен бути придатним для автоматизації, що полегшує його виконання та забезпечує швидке та ефективне тестування.

7. Валідність даних (Valid Data): Всі вхідні дані, використані в тест-кейсі, повинні бути дійсними та реалістичними, відображаючи реальні сценарії використання.

8. Незалежність (Independence): Тест-кейс повинен бути незалежним від інших тест-кейсів. Це допомагає уникнути взаємозалежності тестів та забезпечити їх ефективне виконання окремо.

9. Адаптованість (Adaptability): Тест-кейс повинен бути легко адаптованим до змін у вихідному коді, які відбуваються в процесі розробки.

10. Вимірюваність (Measurability): Якщо це можливо, тест-кейс повинен мати критерії успішності та метрики, які можна вимірювати для оцінки результатів тестування.

11. Правильна технічна мова, точність і однаковість формулювань. Ця властивість в рівній мірі відноситься і до вимог, і до тест-кейсів, і до звітів про

дефекти - до будь-якої документації:

• писати лаконічно, але зрозуміло;

• використовувати безособову форму дієслів (наприклад, «відкрити» замість «відкрийте»);

• обов'язково вказувати точні імена і технічно вірні назви елементів додатку;

• не пояснювати базові принципи роботи з комп'ютером (передбачається, що колеги знають, що таке, наприклад, «пункт меню» і як з ним працювати);

• всюди називати одні і ті ж речі однаково (наприклад, не можна в одному тест-кейсі якийсь режим роботи програми назвати «графічне представлення», а в іншому той же режим - «візуальне відображення»);

• слідувати прийнятому на проекті стандарту оформлення та написання тест-кейсів (іноді такі стандарти можуть бути вельми жорсткими: аж до регламентації того, назви яких елементів повинні бути приведені в подвійних лапках, а будь - в одинарних).

12. Баланс між специфічністю і загальністю. Тест-кейс вважається тим більш специфічним, чим більш детально в ньому розписані конкретні дії, конкретні значення і т.д., тобто ніж в ньому більше конкретики. Відповідно, тест-кейс вважається тим більш загальним, чим в ньому менше конкретики.

13. Баланс між простотою і складністю. Тут не існує академічних визначень, але прийнято вважати, що простий тест-кейс оперує одним об'єктом (або в ньому явно видно головний об'єкт), а також містить невелику кількість тривіальних дій; складний тест-кейс оперує кількома рівноправними об'єктами і містить багато нетривіальних дій.

Переваги простих тест-кейсів:

• їх можна швидко прочитати, легко зрозуміти і виконати;

• вони зрозумілі початківцям-тестувальникам і новим людям на проекті;

• вони роблять наявність помилки очевидним (як правило, в них передбачається виконання повсякденних тривіальних дій, проблеми з якими видно неозброєним оком і не викликають дискусій);

• вони спрощують початкову діагностику помилки, тому що звужують коло пошуку.

Переваги складних тест-кейсів:

• при взаємодії багатьох об'єктів підвищується ймовірність виникнення помилки;

• користувачі, як правило, використовують складні сценарії, а тому складні

тести більш повноцінно емулюють роботу користувачів;

• програмісти рідко перевіряють такі складні випадки (і вони зовсім не зобов'язані це робити).

14. «Показовість» (висока ймовірність виявлення помилки). Починаючи з рівня тестування критичного шляху.

Можна стверджувати, що тест-кейс є більш гарним, якщо він більш показовий (з більшою ймовірністю виявляє помилку). Саме тому вважається непридатними занадто прості тест-кейси - вони є непоказовими.

15. Послідовність в досягненні мети. Суть цієї властивості виражається в тому, що всі дії в тест-кейсі спрямовані на дотримання єдиної логіки і досягнення єдиної мети і не містять ніяких відхилень.

16. Відсутність зайвих дій. Найчастіше це властивість має на увазі, що

не потрібно в кроках тест-кейсу довго і по пунктах розписувати те, що можна замінити однією фразою.

17. Ненадмірність по відношенню до інших тест-кейсів. У процесі створення безлічі тест-кейсів дуже легко опинитися в ситуації, коли два і більше тест-кейса фактично виконують одні і ті ж перевірки, переслідують одні й ті ж цілі, спрямовані на пошук одних і тих же проблем.

Якщо виявляється кілька тест-кейсів, які дублюють завдання один одного, краще за все або видалити всі, крім одного, самого показового, або перед видаленням інших на їх основі доопрацювати цей обраний показовий тест-кейс.

18. Демонстративність (здатність демонструвати виявлену помилку очевидним чином).

Очікувані результати повинні бути підібрані і сформульовані таким чином, щоб будь-яке відхилення від них відразу ж впадало в очі і ставало очевидним, що сталася помилка.

19. Відстеження. З інформації в якісному тест-кейсі повинно бути зрозуміло, яку частину програми, які функції і які вимоги він перевіряє. Частково ця властивість досягається шляхом заповнення відповідних полів тест-кейсу, а й сама логіка тест-кейса грає не останню роль, тому що в разі серйозних порушень цієї властивості можна довго з подивом дивитися, наприклад, на яку вимогу посилається тест-кейс, і намагатися зрозуміти, як же вони один з одним пов'язані.

20. Відповідність прийнятим шаблонами оформлення і традиціям. З шаблонами оформлення, як правило, проблем не виникає: вони строго визначені наявним зразком або взагалі екранної формою інструментального засобу управління тест-кейсами. Що ж стосується традицій, то вони відрізняються навіть в

різних командах в рамках однієї компанії, і тут неможливо дати іншої ради, крім як «почитайте вже готові тест-кейси перед тим як писати свої».

Набори тест-кейсів

Набір тест-кейсів - це комплекс документів або набір інструкцій, який визначає тестові сценарії для перевірки функціональності, властивостей чи характеристик програмного забезпечення. Цей набір створюється з метою виявлення дефектів та впевненості в правильності роботи програмного продукту.

Набір тест-кейсів (test case suite, test suite, test set) - сукупність тест-кейсів, обраних з деякою загальною метою або по деякому спільною ознакою. Іноді в такій сукупності результати завершення одного тест-кейса стають вхідним станом додатку для наступного тест-кейса.

Безлічі окремих тест-кейсів писати вкрай незручно (більш того, це помилка!) Кожен раз писати в кожному тест-кейсі одні і ті ж підготовки і повторювати одні й ті ж початкові кроки.

Набагато зручніше об'єднати кілька тест-кейсів в набір або послідовність. І тут ми приходимо до класифікації наборів тест-кейсів.

У загальному випадку набори тест-кейсів можна розділити на **вільні** (порядок виконання тест-кейсів не важливий) і **послідовні** (порядок виконання тест-кейсів важливий).

Вільні (Independent) та Послідовні (Sequential) набори тест-кейсів:

Вільні (Independent) набори тест-кейсів:

Характеристики:

- Тест-кейси взаємозалежні мінімально.
- Кожен тест-кейс може бути виконаний окремо без впливу на інші.
- Використовується, коли немає необхідності виконувати тест-кейси в конкретному порядку.
- Дозволяє розподілити тест-кейси між різними тестерами або командами.

Переваги:

- Можливість паралельного виконання тестів, що прискорює процес тестування.
- Зменшення часу виконання, оскільки тест-кейси можуть бути запущені паралельно.
- Легше управління тест-кейсами та їхнім виконанням.

Недоліки:

- Залежно від характеру продукту, може бути складніше визначити вільні тест-кейси.

Послідовні (Sequential) набори тест-кейсів:

Характеристики:

• Тест-кейси взаємозалежні та мають конкретний порядок виконання.

• Використовується, коли результат одного тесту впливає на виконання іншого.

Переваги:

• Дозволяє визначити послідовність тестів, що є критично важливою для деяких продуктів або функціональностей.

• Може допомогти в зручному відлагодженні та виявленні проблем, зокрема, якщо один тест залежить від результату попереднього.

Недоліки:

• Збільшує час виконання, оскільки тест-кейси мають виконуватись послідовно.

• Обмежує можливість паралельного виконання тестів та розподілу завдань між тестерами.

Обираючи між вільними та послідовними наборами тест-кейсів, команди тестування повинні враховувати особливості проекту, обсяг роботи, терміни виконання, технічні вимоги та інші чинники. У багатьох випадках комбінація обох підходів може бути ефективною для досягнення оптимального балансу між продуктивністю та якістю.

Користувацькі сценарії (сценарії використання)

До окремого підвиду послідовних наборів тест-кейсів (або навіть неоформлених ідей тест-кейсів, таких, як пункти чек-листа) можна віднести призначені для користувача сценарії (або сценарії використання), що представляють собою ланцюжок дій, що виконуються в певній ситуації для досягнення певної мети.

Сценарії можуть бути досить довгими і складними, можуть містити всередині себе цикли і умовні розгалуження, але при всьому цьому вони мають ряд цікавих переваг:

• Сценарії показують реальні і зрозумілі приклади використання продукту (на відміну від великих чек-листів, де сенс окремих пунктів може губитися).

• Сценарії зрозумілі кінцевим користувачам і добре підходять для обговорення і спільного поліпшення.

• Сценарії і їх частини легше оцінювати з точки зору важливості, ніж окремі пункти (особливо низькорівневих) вимог.

• Сценарії відмінно показують недоробки в вимогах (якщо стаєне зрозуміло, що робити в тому чи іншому пункті сценарію, - з вимогами явно щось не те).

• У граничному випадку (брак часу та інші форс-мажори) сценарії можна навіть не прописувати детально, а просто назвати - і сама назва вже підкаже досвідченому фахівцеві, що робити.

Детальна класифікація наборів тест-кейсів

		За ізольованістю тест-кейсів один від одного	
		Ізольовані	Узагальнені
За утворенням тест-кейсами суворої послідовності	Вільні	Ізольовані вільні	Узагальнені вільні
	Послідовні	Ізольовані послідовні	Узагальнені послідовні

Рис. 7.1. Детальна класифікація наборів тест-кейсів

Набір ізольованих вільних тест-кейсів — це набір тест-кейсів, які можуть бути виконані незалежно один від одного. Це означає, що результат виконання кожного тест-кейсу не впливає на інші, і їх можна виконувати паралельно без обмежень. Такий підхід дозволяє ефективно розподіляти завдання між тестерами та прискорює процес тестування.

Нижче представлений приклад структури для ізольованих вільних тест-кейсів:

Тест-кейс №1: Реєстрація користувача

- *Опис: Перевірка можливості успішної реєстрації нового користувача.*
- *Кроки виконання:*

1.1 Відкрити сторінку реєстрації.

1.2 Ввести коректні дані в усі обов'язкові поля.

1.3 Натискання на кнопку "Зареєструватися".

- *Очікувані результати: Користувач повинен бути успішно зареєстрований.*

Набір узагальнених вільних тест-кейсів - може включати широкий спектр тестових сценаріїв, що визначаються для різних функціональних або модульних частин продукту. Важливо забезпечити велику покриття, охоплюючи різноманітні аспекти продукту. Нижче наведено приклад узагальненого вільного набору тест-кейсів:

Тест-кейс для сумісності з різними браузерами:

- *Опис: Перевірка, як продукт відображається та працює в різних веб-браузерах.*
- *Кроки виконання:*

1.1 Відкрити продукт у популярних браузерах (Chrome, Firefox, Safari,

Edge).

1.2 Перевірити, чи всі функції працюють коректно в кожному браузері.

• *Очікувані результати: Продукт сумісний з різними браузерами*.

Набір ізольованих послідовних тест-кейсів передбачає виконання тестів в певному порядку, де результат кожного тест-кейсу впливає на наступний. Такий підхід може бути корисним для тестування взаємодії різних компонентів чи функціональностей продукту. Нижче наведено приклад послідовних тест-кейсів:

Тест-кейс для авторизації користувача:

• *Опис: Перевірка можливості успішного входу до системи після реєстрації.*

• *Кроки виконання:*

1.1 Відкрити сторінку авторизації.

1.2 Ввести валідні дані користувача, який був зареєстрований на попередньому кроці.

1.3 Натискання на кнопку "Увійти".

• *Очікувані результати: Користувач повинен успішно увійти до системи.*

Узагальнений набір послідовних тест-кейсів може охоплювати різні аспекти продукту та передбачати виконання кроків в певному порядку, де результат одного тест-кейсу впливає на наступний. Нижче подано приклад узагальненого набору послідовних тест-кейсів:

Тест-кейс для додавання завдання в проект:

• *Опис: Перевірка можливості успішного додавання завдання до створеного проекту.*

• *Кроки виконання:*

1.1 Відкрити раніше створений проект.

1.2 Перейти до розділу "Завдання".

1.3 Натискання на кнопку "Додати завдання".

1.4 Ввести необхідні дані та опис завдання.

1.5 Збереження нового завдання.

• *Очікувані результати: Завдання успішно додано до проекту.*

Головна перевага ізольованості: кожен тест-кейс виконується в «Чистому середовищі», на нього не впливають результати роботи попередніх тест-кейсів.

Головна перевага узагальненості: підготовку не потрібно повторювати (економія часу).

Головна перевага послідовності: відчутне скорочення кроків в кожному тест-кейсі, тому що результат виконання попереднього тест-кейсу є початкової

ситуацією для наступного.

Головна перевага свободи: можливість виконувати тест-кейси в будь-якому порядку, а також те, що при провалі якогось тест-кейса (результат роботи додатку не співпадає з очікуваним станом) інші тест-кейси і раніше можна виконувати.

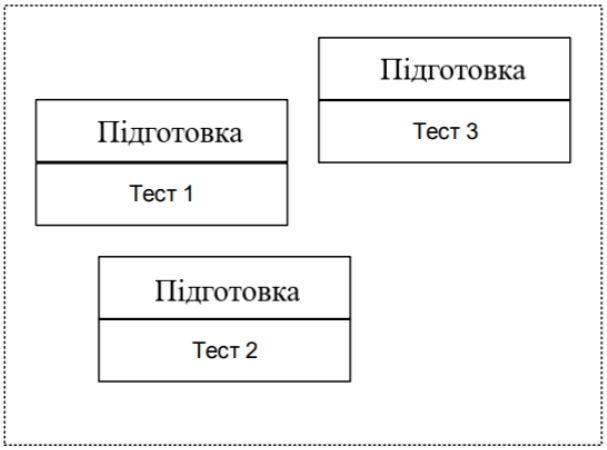

Рис. 7.2. Набір ізольованих вільних тест-кейсів

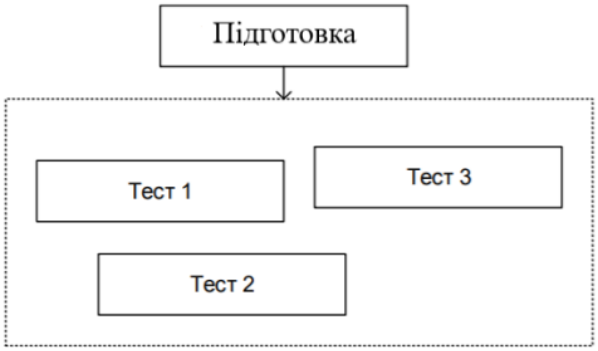

Рис. 7.3. Набір узагальнених вільних тест-кейсів

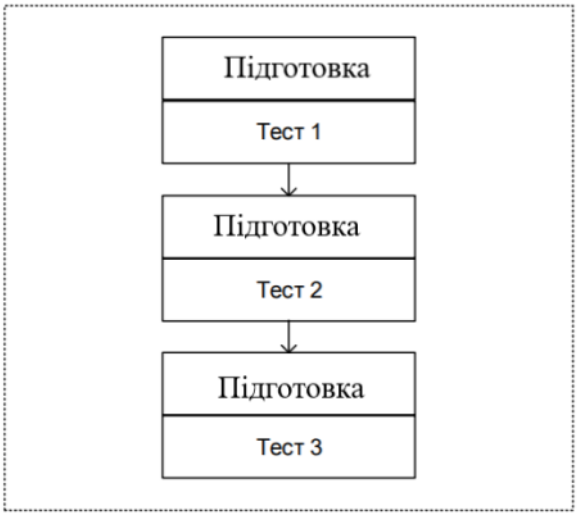

Рис. 7.4. Набір ізольованих послідовних тест-кейсів

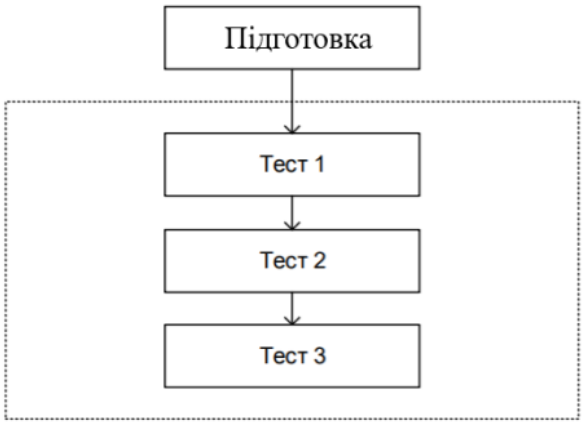

Рис. 7.5. Набір узагальнених послідовних тест-кейсів

Принципи побудови наборів тест-кейсів

Єдине завдання наборів - підвищити ефективність тестування за рахунок прискорення та спрощення виконання тест-кейсів, збільшення глибини дослідження якоїсь галузі застосування або функціональності, проходження типовим для користувача сценаріями або зручною послідовності виконання тест-кейсів і т.д.

Набір тест-кейсів завжди створюється з якоюсь метою, на основі якоїсь логіки, і за цими ж принципами в набір включаються тести, що володіють відповідними властивостями.

Типові підходи до складання наборів тест-кейсів:

• На основі чек-листів.

• На основі розбиття програми на модулі і підмодулі. Для кожного модуля (або його окремих подмодулей) можна скласти свій набір тест-кейсів.

• За принципом перевірки найважливіших, менш важливих і всіх інших функцій програми.

• За принципом угруповання тест-кейсів для перевірки якогось рівня вимог або типу вимог, групи вимог або окремих вимог.

• За принципом частоти виявлення тест-кейсами дефектів в додатку (Наприклад, ми бачимо, що деякі тест-кейси раз по раз завершуються невдачею, значить, ми можемо об'єднати їх в набір, умовно названий «Проблемні місця в додатку»).

• За архітектурним принципом: набори для перевірки користувальницького інтерфейсу і всього рівня уявлення, для перевірки рівня бізнес-логіки, для перевірки рівня даних.

• По області внутрішньої роботи програми, наприклад: «тест-кейси, що зачіпають роботу з базою даних», «тест-кейси, що зачіпають роботу з файловою

системою», «тест-кейси, що зачіпають роботу з мережею », і т.д.

• За видами тестування.

Типові помилки при розробці чек-листів, тест-кейсів і наборів тест-кейсів

При розробці чек-листів для тестування можуть виникати різні помилки, які можуть впливати на ефективність тестування:

Неповне охоплення:

- **Помилка:** Не урахування усіх можливих сценаріїв та випадків в тестуванні.
- **Рекомендація:** Ретельно вивчати вимоги та функціонал продукту, щоб переконатися, що чек-лист охоплює всі важливі аспекти.

Відсутність оновлення:

- **Помилка:** Недостатнє оновлення чек-листів під час змін у вимогах або функціоналі продукту.
- **Рекомендація:** Регулярно переглядати та оновлюйте чек-листи, щоб вони відповідали актуальному стану продукту.

Надмірне дублювання:

- **Помилка:** Надто багато повторюваних або схожих пунктів в чек-листі.
- **Рекомендація:** Уникати надмірного дублювання та стежте за консистентністю.

Невизначені кроки або очікувані результати:

- **Помилка:** Відсутність чітких кроків для виконання або визначення очікуваних результатів.
- **Рекомендація:** Кожен пункт чек-листа повинен мати чіткі кроки виконання та опис очікуваних результатів.

Недостатнє тестування різноманітності:

- **Помилка:** Відсутність тестів, які враховують різноманітність вхідних даних та умов.
- **Рекомендація:** Забезпечити різноманітність у тест-кейсах, щоб охопити різні аспекти функціоналу.

Неправильний порядок виконання тестів:

- **Помилка:** Неправильний порядок виконання тестів, що може впливати на логіку тестування.
- **Рекомендація:** Визначити логічний порядок виконання тестів для зручності та ефективності.

Недостатнє тестування виключень та помилок:

- **Помилка:** Недостатнє тестування сценаріїв виключень та помилок.
- **Рекомендація:** Включити в чек-лист тести, які перевіряють, як продукт поводиться в ситуаціях помилок та виключень.

Відсутність перевірки безпеки:

- **Помилка:** Відсутність тестів безпеки, які перевіряють вразливості продукту.
- **Рекомендація:** Зберегти чек-лист тестів безпеки для перевірки захищеності продукту.

При розробці тест-кейсів можуть виникати різні помилки, які можуть впливати на якість тестування:

Невірна формулювання кроків:

- **Помилка:** Неясні або неправильно сформульовані кроки виконання тесту.
- **Рекомендація:** Кожен крок повинен бути чітким, конкретним та лаконічним.

Відсутність передумов:

- **Помилка:** Невизначені чи неправильно вказані передумови для виконання тесту.
- **Рекомендація:** Вказувати всі передумови, необхідні для коректного виконання тест-кейсу.

Відсутність очікуваних результатів:

- **Помилка:** Не визначення чітких очікуваних результатів після виконання тесту.
- **Рекомендація:** Кожен тест-кейс повинен мати опис того, які результати очікуються в результаті його виконання.

Надмірне тестування:

- **Помилка:** Занадто деталізовані чи зайві тест-кейси, які можуть призвести до витрат часу та ресурсів.
- **Рекомендація:** Створювати тест-кейси, які охоплюють основні сценарії використання та критичні функціональності.

Відсутність ручної перевірки:

- **Помилка:** Всі тести орієнтовані на автоматизацію, і не залишено місця для ручної перевірки.
- **Рекомендація:** Залишати можливість для ручного тестування там, де це

необхідно, оскільки автоматизоване тестування не завжди охоплює всі аспекти.

Недостатнє тестування різноманітності:

• **Помилка**: Недостатнє уваги до тестування різних комбінацій вхідних даних чи умов.

• **Рекомендація:** Включати в тест-кейси різноманітність у вхідних даних для більш повного покриття тестування.

Невідповідність до стандартів:

• **Помилка:** Тест-кейси не відповідають стандартам тестування або внутрішнім гідлайнам.

• **Рекомендація:** Дотримуватися стандартів тестування та створюйте тест-кейси відповідно до них.

Відсутність тестів на винятки:

• **Помилка:** Неврахування сценаріїв винятків та обробки помилок.

• **Рекомендація:** Включати тести, які перевіряють, як система реагує на непередбачувані умови.

Велика кількість залежностей між тест-кейсами:

• **Помилка:** Занадто велика кількість залежностей між тест-кейсами, що ускладнює їх виконання в реальних умовах.

• **Рекомендація:** Зберігати незалежність між тест-кейсами для покращення їхньої виконуваності та стабільності.

Розробка наборів тест-кейсів є важливою частиною процесу тестування, і при їх створенні можуть виникати різні помилки:

Надмірна кількість тест-кейсів:

• **Помилка:** Створення надто великого набору тест-кейсів, що може важко управляти та виконувати.

• **Рекомендація:** Створювати обґрунтовані та ефективні набори тест-кейсів, які охоплюють ключові сценарії та функціональність.

Недостатній рівень деталізації:

• **Помилка:** Набори тест-кейсів, які не включають деталей або не охоплюють конкретні сценарії.

• **Рекомендація:** Забезпечити достатній рівень деталізації для кожного тест-кейсу, щоб уникнути непорозумінь.

Недостатній розподіл функціональності:

• **Помилка:** Набір тест-кейсів, який не враховує розподіл функціоналу та покриття ключових областей.

• **Рекомендація:** Розподілити тест-кейси так, щоб вони відображали різні

аспекти та функціональність продукту.

Відсутність перевірки на великі обсяги даних:

• **Помилка:** Невідсутність тест-кейсів, які перевіряють, як система веде себе при обробці великих обсягів даних.

• **Рекомендація:** Включити в набір тест-кейсів сценарії, які тестують продукт на великі обсяги даних.

Невідповідність до критичних шляхів:

• **Помилка:** Непокриття критичних шляхів та сценаріїв, які можуть впливати на якість продукту.

• **Рекомендація:** Забезпечити, щоб набір тест-кейсів включав тестування критичних шляхів та ключових функцій.

Відсутність тестів безпеки:

• **Помилка:** Невідсутність тестів, що перевіряють безпекові аспекти продукту.

• **Рекомендація:** Включіити в набір тест-кейсів сценарії, які перевіряють безпекові вразливості.

Неправильне керування залежностями:

• **Помилка:** Неправильне керування залежностями між тест-кейсами, що може впливати на послідовність виконання.

• **Рекомендація:** Зберігати незалежність між тест-кейсами та визначайте залежності ясно.

Недостатній огляд та оновлення:

• **Помилка:** Недостатній огляд та оновлення набору тест-кейсів з часом.

• **Рекомендація:** Регулярно переглядати та оновлюйте набір тест-кейсів відповідно до змін у вимогах або функціоналі продукту.

ТЕМА 8. ВИЗНАЧЕННЯ БАГІВ

Помилки, дефекти, збої, відмови

Дефект - розбіжність очікуваного і фактичного результату.

Очікуваний результат - поведінка системи, описана в вимогах.

Фактичний результат - поведінка системи, що спостерігається в процесі тестування.

Помилка (error, Mistake) - дія людини, що приводить до некоректних результатів.

Рис. 8.1. Залежність відмови, збою від появи помилки, дефекту

Дефект (defect, bug, problem, fault) — недолік в компоненті або системі, здатний привести до ситуації збою або відмови.

Збій (interruption) або відмова (failure) - відхилення поведінки системи від очікуваного.

Збій – відмова, яка самоусувається або одноразова відмова, яка усувається незначним втручанням працівника.

Відмова - подія, що полягає в порушенні працездатного стану об'єкта.

Аномалія (anomaly) або інцидент (incident, deviation) - будь-яке відхилення спостережуваного (фактичного) стану, поведінки, значення, результату, властивості від очікувань спостерігача, сформованих на основі вимог, специфікацій, іншої документації або досвіду і здорового глузду.

Джерела (Expected results)

Що таке “некоректна” робота?

Коли тестується додаток, тестувальник знає що додаток повинен робити, іншими словами — який має бути очікуваний результат (expected result).

Далі, тестувальник порівнює те, що отримано в результаті виконання програми (actual result) і порівнює його з очікуваним результатом. Якщо обидва результати збігаються — поведінка вважається очікуваною і ми робимо висновок, що це працює коректно. Якщо очікуваний результат не збігається з фактичним — це дефект (баг).

Звідки ми знаємо, що очікувати?

Очікувана поведінка береться з джерела очікуваної поведінки! Існує багато джерел очікуваної поведінки, серед них можуть бути:

Специфікації вимог (Requirements Specification)

Ще використовують скорочення “RS”. Це документ, який описує, що має бути реалізовано у продукті, що треба протестувати. Він містить інформацію про функціональність і поведінку, які очікуються від продукту.

Бізнес-вимоги (Business Requirements)

Це документ, який описує, як продукт використовуватиметься бізнесом. Він містить інформацію про те, як продукт має бути зручним для користувачів і як він буде генерувати прибуток для компанії.

Дизайн-документ (Design document)

Це документи, які описують, як продукт має виглядати та поводитися. Вони включають дизайн-макети, схеми, діаграми та інші документи, які показують, як повинен працювати продукт.

Знання досвідчених користувачів (Experienced users knowledge)

Це знання про те, як користувачі використовуватимуть продукт. Вони можуть бути отримані шляхом опитувань користувачів, спостереження за їх поведінкою та аналізу їхніх відгуків про попередні версії продукту.

Попередні версії продукту

Якщо продукт уже був випущений, його попередні версії можуть використовуватися як джерело очікуваного результату. Аналіз попередніх версій допоможе визначити, які функції будуть найбільш затребувані користувачами.

Схожі продукти

Поведінка продукту, який розв’язувати схожі задачі, або частина будь-якого продукту, яка робить те саме, що і наш продукт. Такі продукти називають оракулами (oracles)

Експертні знання

Це знання про продукт, які можна отримати від експертів у галузі або від розробників продукту. Експерти можуть дати цінні рекомендації щодо того, як продукт повинен поводитися і що очікувати від тестування.

Але при всьому різноманітті джерел, треба пам’ятати, що головне джерело — це вимоги в специфікаціях (чи те джерело, яке містить вимоги на конкретному проєкті). Якщо в вимогах немає опису очікуваного результату, його треба зрозуміти і узгодити з тими, хто відповідає за прийняття таких рішень з боку клієнта та команди, а потім треба додати до специфікації і узгодити це з людьми, які відповідають за вимоги на стороні клієнта та команди.

Коли є опис якогось функціонала в, наприклад, дизайн документі, але його

немає в вимогах специфікації, то треба погодити додавання цієї вимоги в специфікацію.

Специфікація — це головне джерело очікуваного результату.

Що таке некоректна робота?

✓ збій програми, що призводить до її завершення (crash);

✓ зависання програми (програмою неможливо користуватися) (hang up);

✓ видача програмою системних повідомлень про помилки (system error);

✓ помилки дизайну, що заважають нормальній роботі з програмою (наприклад кнопка, яку неможливо нажати, бо вона перекрита інформаційним повідомленням);

✓ дрібні неточності та текстові помилки (орфографічні чи стилістичні) [10].

Звіт про дефект і його життєвий цикл

Звіт про дефект (defect report) - документ, що описує і надає пріоритет виявленому дефекту, а також сприяє його усуненню.

Цілі звіту про дефект:

• надати інформацію про проблему - повідомити проектну команду і інших зацікавлених осіб про наявність проблеми, описати суть проблеми;

• пріоритезувати проблему - визначити ступінь небезпеки проблеми для проекту і бажані терміни її усунення;

• сприяти усуненню проблеми - якісний звіт про дефекти не тільки надає всі необхідні подробиці для розуміння суті, що сталося, але також може містити аналіз причин виникнення проблеми та рекомендації щодо виправлення ситуації.

Звіт про дефект (і сам дефект разом з ним) проходить певні стадії життєвого циклу:

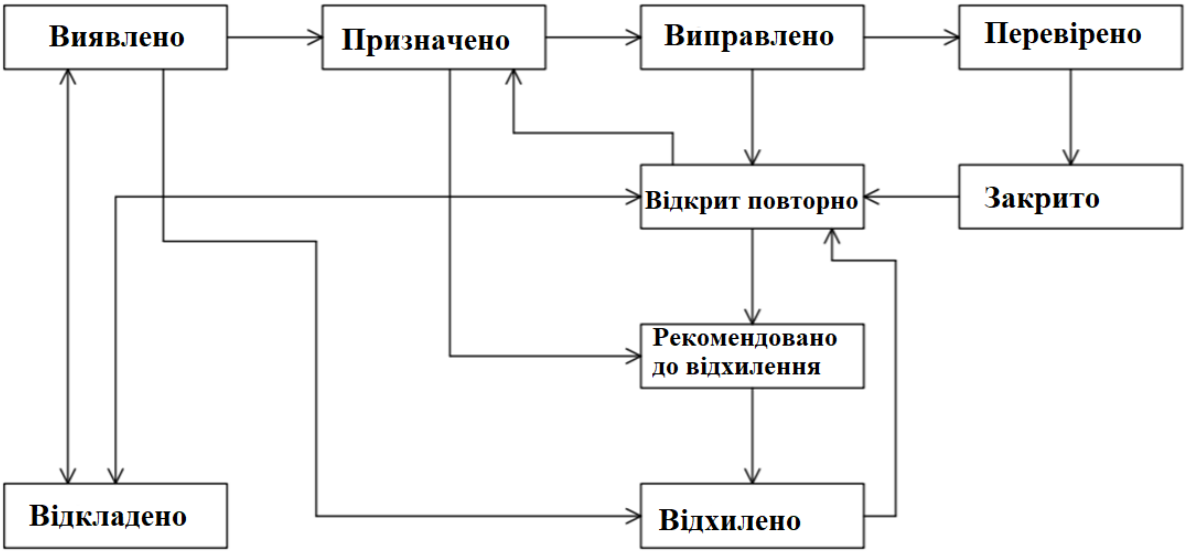

Рис. 8.2. Стадії життєвого циклу дефекту

• **Виявлено (submitted)** - початковий стан звіту (іноді називається «Новий» (new)), в якому він знаходиться відразу після створення. Деякі засоби також дозволяють спочатку створювати чернетку (draft) і лише потім публікувати звіт.

• **Призначено (assigned)** - в цей стан звіт переходить з моменту, коли хтось з проектної команди призначається відповідальним за виправлення дефекту. Призначення відповідального проводиться або рішенням лідера команди розробки, або колегіально, або за добровільним принципом, або іншим прийнятим в команді способом або виконується автоматично на основі певних правил.

• **Виправлено (fixed)** - в цей стан звіт переводить відповідальний за виправлення дефекту член команди після виконання відповідних дій з виправлення.

• **Перевірено (verified)** - в цей стан звіт переводить тестувальник, засвідчуючи, що дефект насправді був усунутий. Як правило, таку перевірку виконує тестувальник, спочатку написавши звіт про дефект.

• **Закрито (closed)** - стан звіту, що означає, що з даного дефекту не планується ніяких подальших дій. Тут є деякі розбіжності в життєвому циклі, прийнятому в різних інструментальних засобах управління звітами про дефекти:

o У деяких засобах існують обидва стану - «Перевірено» і «Закрито», щоб підкреслити, що в стані «Перевірено» ще можуть знадобитися якісь додаткові дії (обговорення, додаткові перевірки в нових білдах і т.д.), у той час як стан «Закрито» означає «з дефектом покінчено, більше до цього питання не повертаємося».

o У деяких засобах одного з станів немає (воно поглинається іншим).

o У деяких засобах в стан «Закрито» або «відхилений» звіт про дефекті може бути переведений з безлічі попередніх станів з резолюціями на зразок:

▪ «Не є дефектом» - додаток так і повинен працювати, описана поведінка не є аномальною.

▪ «Дублікат» - даний дефект вже описаний в іншому звіті.

▪ «Не вдалося відтворити» - розробникам не вдалося відтворити проблему на своєму обладнанні.

▪ «Не буде виправлено» - дефект є, але з якихось серйозних причин його вирішено не виправляти.

▪ «Неможливо виправити» - непереборна причина дефекту знаходиться поза області повноважень команди розробників, наприклад існує проблема в операційній системі або апаратному забезпеченні, вплив якої усунути розумними способами неможливо.

• Відкрито заново (reopened) - в цей стан (як правило, зі стану «Виправлено») звіт переводить тестувальник, засвідчуючи, що дефект як і раніше відтворюється на білді, в якому він вже повинен бути виправлений.

• Рекомендований до відхилення (to be declined) - в цей стан звіт про дефект може бути переведений з безлічі інших станів з метою винести на розгляд питання про відхилення звіту по тій або іншій причині.

Якщо рекомендація є обґрунтованою, звіт переводиться в стан «Відхилений»

• Відхилений (declined) - в цей стан звіт переводиться в випадках, докладно описаних в пункті «Закрито», якщо засіб управління звітами про дефектах передбачає використання цього стану замість стану «Закрито» для тих чи інших резолюцій по звіту.

• Відкладений (deferred) - в цей стан звіт переводиться у разі, якщо виправлення дефекту в найближчий час є нераціональним або НЕ представляється можливим, однак є підстави вважати, що в майбутньому ситуація виправиться (вийде нова версія бібліотеки, повернеться з відпустки фахівець з якоїсь технології, зміняться вимоги замовника і т.д.).

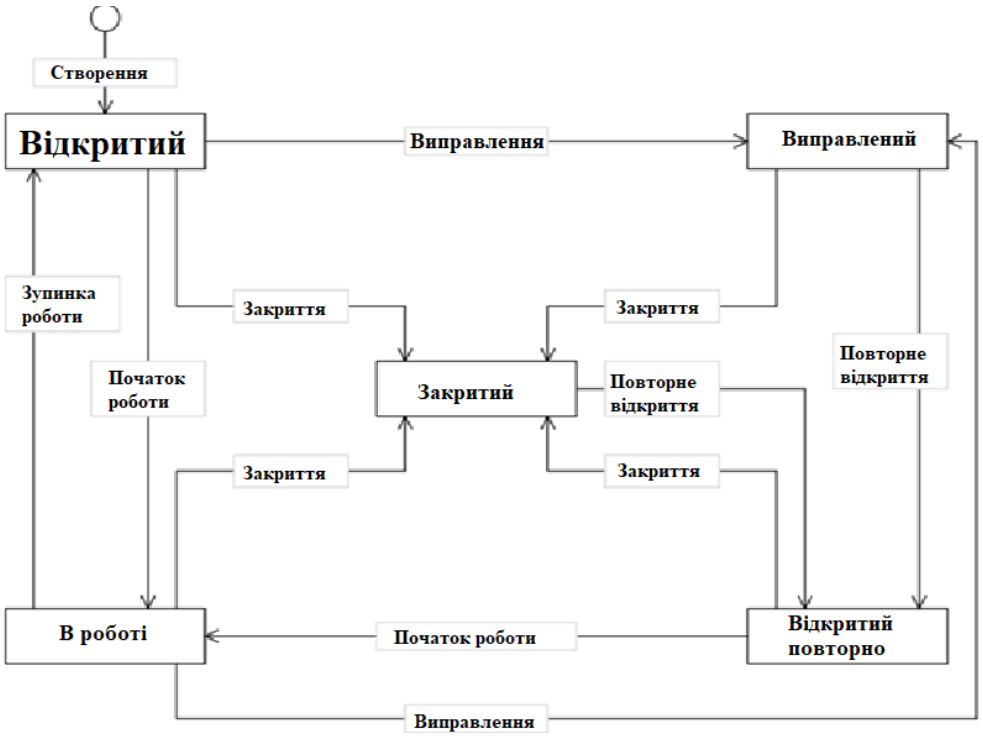

Рис. 8.3. Алгоритм роботи з дефектом

Інструментальні засоби управління звітами про дефекти

На невеликих проєктах для реєстрації помилок (багів) часто використовуються таблиці Excel, проте, по мірі збільшення розміру та складності проєктів,

зручність такого способу сходить нанівець і доводиться задуматися про більш зручне рішення. В процесі роботи QA-інженер може зіткнутися з різними баг-трекінговими системами. У цій статті розглянемо найпопулярніші системи для відстеження помилок.

Що таке баг-трекінгова система?

Баг-трекінгові системи (БТС) або системи відстеження помилок представляють собою програмні продукти, які дозволяють реєструвати і відслідковувати хід вирішення кожної помилки (бага), виявленої тестувальником, до тих пір, поки проблема не буде вирішена.

В широкому розумінні проблемою може бути що завгодно – від повідомлення про помилку до питання клієнта і запиту на розробку. Проблеми часто плутають з помилками, тому що всі помилки є проблемами, але не всі проблеми є помилками. Типова помилка – це дефект в кодовій базі, і одна помилка може виявлятися різними способами. Іноді буває так, що одна і та ж помилка викликає кілька багів, тобто у користувачів може виникнути декілька проблем через одну помилку. В такому випадку БТС стають дуже корисними, так як в них можна зв'язати між собою баг-репорти і встановити залежності.

Особливості баг-трекінгових систем

У кожній БТС є певні можливості керування задачами, які схожі між собою. Деякі задачі можуть мати більш високий пріоритет, ніж інші, можливо, тому, що вони зачіпають велику кількість клієнтів, або тому, що вони являють собою серйозну перешкоду, яку необхідно вирішити. Крім того, завдання можуть також мати низьку або нульову терміновість або пріоритет, і це також потрібно вказувати, щоб розуміти, що такі завдання можуть бути вирішені пізніше, якщо дозволяє час.

Більшість рішень систем відстеження помилок також дозволяють **призначати проблеми** різним відповідальним особам, **відстежувати**, як вони обробляються і скільки часу на них витрачається, **забезпечувати відповідність** внутрішнім робочим процесам, виконувати статистичний аналіз і автоматично генерувати завдання або тікети (від англ. ticket – квиток, картка, оголошення) на основі даних клієнта.

Баг-репорт (звіт про дефект) — це документ, який містить інформацію про виявлені помилки чи дефекти в програмному забезпеченні. Поля баг-репорту можуть варіюватися залежно від використовуваної системи відстеження помилок, але основні поля зазвичай включають:

Заголовок (Title): Короткий та описовий заголовок, який вказує на сут-

ність дефекту.

Опис (Description): Докладний опис помилки, включаючи послідовність кроків для відтворення, якщо це можливо. Також варто включити інші деталі, як-то стан системи, оточення, додаткові відомості, які допоможуть зрозуміти суть проблеми.

Кроки для відтворення (Steps to Reproduce): Деталізована інструкція щодо кроків, необхідних для відтворення дефекту. Це допомагає розробникам легше знайти та виправити проблему.

Очікувані результати (Expected Results): Чітке визначення того, що очікується від системи або програми після виконання вказаних кроків.

Фактичні результати (Actual Results): Опис того, що відбувається насправді, якщо є виявлено дефект.

Прикріплені файли або скріншоти (Attachments): Якщо це можливо, додайте прикріплені файли, які допоможуть зрозуміти або відтворити дефект (скріншоти, лог-файли тощо).

Пріоритет (Priority): Вказівка на те, наскільки важливо ви вважаєте виправлення даного дефекту. Зазвичай використовуються терміни, такі як "високий", "середній", "низький".

Категорія (Category) або Тип (Type): Визначення категорії дефекту, наприклад, інтерфейс, функціональність, продуктивність, безпека тощо.

Версія програми (Version): Версія програми, на якій був виявлений дефект.

Середовище (Environment): Інформація про операційну систему, браузер, версії використаних програм та інше, які можуть впливати на дефект.

Сучасні баг-трекінгові системи забезпечують контроль за виконанням завдань і автоматичну генерацію звітів, а також часто інтегруються з іншими інструментами розробки проєктів.

Далі розглянуто найбільш популярні системи відстеження помилок, які були створені для задоволення потреб як незалежних розробників, так і бюджетних установ.

1. JIRA

Платна система, в якій за допомогою інтерактивної дошки можна стежити за процесом переміщення задач, контролюючи процес їх виконання на проєкті. Це не тільки баг-трекер, але і дуже зручний інструмент для управління проєктами, особливо для agile-команд.

Функціонал даної програми дуже широкий, якщо чогось не вистачає, його завжди можна доповнити за допомогою плагінів.

QA-інженери зазвичай використовують JIRA як баг-трекер, оформляючи в ньому баг-репорти. В системі зазвичай заповнюються завдання (від англ. Tickets або issues), які можуть відрізнятися за типом: task, story, epic, bug. Тестувальники найчастіше вибирають «bug».

При створенні баг-репорту доступні такі атрибути:

- назва проєкту;
- тип завдання;
- тема;
- опис;
- пріоритет;
- мітки;
- оточення;
- вкладення;
- пов'язані задачі.

2. MANTIS

У порівнянні з іншими баг-трекінговими системами це досить простий інструмент. Він доступний як у вигляді web-додатка, так і в мобільній версії. Даний баг-трекер сумісний з декількома базами даних, такими як MySQL, PostgreSQL, MS SQL і дозволяє інтегрувати в нього такі додатки, як чат, відстеження часу, вікі, RSS-канали та багато іншого.

Основні переваги системи:

- відкритий вихідний код;
- можливість відправки нотифікацій на електронну адресу;
- можливість налаштування полів;
- підтримка time tracking;
- можливість роботи в декількох проєктах одночасно;
- доступна історія змін в звітах;
- необмежена кількість користувачів, проєктів і баг-репортів.

За умовчанням доступні наступні атрибути баг-репорт:

- назва проєкту;
- категорія;
- пріоритет;
- серйозність;

- відтворюваність;
- тема;
- опис;
- кроки відтворення;
- додаткова інформація;
- вкладення.

Також, баг-репорт можна призначити на будь-якого користувача, який працює в проєкті.

3. Redmine

Ця система є відкритим серверним web-додатком для керування проєктами та завданнями, в тому числі і для стеження за помилками.

Даний баг-трекер безкоштовний, з відкритим вихідним кодом, його інтерфейс досить простий.

Основні переваги:

- підтримка декількох проєктів;
- контроль доступу на основі ролей;
- календар;
- стрічки і повідомлення по електронній пошті;
- Вікі і форуми по проєкту;
- відстеження часу;
- створення завдань по електронній пошті;
- самостійна реєстрація користувача;
- багатомовний інтерфейс;
- підтримка декількох баз даних.

У Redmine доступні такі атрибути баг-репорт:

- вид завдання (bug, feature, support);
- предмет або тема баг-репорту;
- опис (поле, в якому описуються кроки і результати);
- статус;
- пріоритет;
- на кого призначений;
- дата створення/завершення;
- час виконання;
- вкладення.

Баг-репорт може відслідковуватися будь-яким співробітником, який дода-

ний до проєкту і відзначений як спостерігач в баг-репорті [11].

ТЕМА 9. ВЛАСТИВОСТІ ЯКІСНИХ ЗВІТІВ ПРО ДЕФЕКТИ

Звіт про дефект може виявитися неякісним, якщо в ньому порушена одна з наступних властивостей.

• Ретельне заповнення всіх полів точної і коректної інформацією.

Порушення цієї властивості відбувається за безліччю причин: недостатній досвід тестувальника, неуважність, лінь і т.д. Найяскравішими проявами такої проблеми можна вважати наступні:

• Частина важливих для розуміння полів не заповнена. В результаті звіт перетворюється в набір уривчастих відомостей, використовувати які для виправлення дефекту неможливо.

• Наданої інформації недостатньо для розуміння суті проблеми.

Наприклад, з такого поганого докладного опису взагалі не ясно, про що йде мова: «Додаток іноді невірно конвертує деякі файли».

Надана інформація є некоректною (наприклад, вказані невірні повідомлення додатки, невірні технічні терміни і т.д.).

Найчастіше таке трапляється через неуважність (наслідки помилкового copy-paste і відсутності фінальної вичитки звіту перед публікацією).

• «Дефект» знайдений у функціональності, яка ще не була оголошена як готова до тестування. Тобто тестувальник констатує, що невірно працює те, що і не повинно було (поки що!) Вірно працювати.

• У звіті присутній жаргонна лексика: як в прямому сенсі - нелітературні висловлювання, так і деякі технічні жаргонізми, зрозумілі вкрай обмеженому колу людей. Наприклад: «фігово підчепилися чартнікі». (Малося на увазі: «Не всі таблиці кодувань завантажені успішно».)

• Звіт замість опису проблеми з додатком критикує роботу когось із учасників проектної команди. Наприклад: «Ну яким дурнем треба бути, щоб таке зробити ?!»

• У звіті упущена якась незначна на перший погляд, але по факту критична для відтворення дефекту проблема. Найчастіше це проявляється у вигляді пропуску якогось кроку по відтворенню, відсутності або недостатньо детального опису оточення і т.п.

• Звіту виставлені невірні (як правило, занижені) важливість або терміновість. Щоб уникнути цієї проблеми, варто ретельно досліджувати дефект, визначати його найбільш небезпечні наслідки і аргументовано відстоювати свою точку зору, якщо колеги вважають інакше.

• **До звіту не додано необхідні копії екрану** (особливо важливі для косметичних дефектів) або інші файли. Класика такої помилки: звіт описує невірну роботу програми з деяким файлом, але сам файл не додається.

• **Звіт написаний безграмотно з точки зору людської мови.** Іноді на це можна закрити очі, але іноді це стає реальною проблемою, наприклад: «Not keyboard in parameters accepting values» (це реальна цитата, і сам автор так і не зміг пояснити, що ж малося на увазі).

• **Правильна технічна мова.**

Ця властивість в рівній мірі відноситься і до вимог, і до тест-кейсів, і до звітів про дефекти - до будь-якої документації.

• **Специфічність опису кроків.** В кроках тест-кейсів варто дотримуватися золотої середини між специфічністю і спільністю. У звітах про дефекти перевагу, як правило, віддається специфічності по дуже простій причині: нестача якоїсь дрібною деталі може привести до неможливості відтворення дефекту. Тому, якщо є хоч найменший сумнів у тому, чи важлива якась деталь, вважайте, що вона важлива.

• **Відсутність зайвих дій і / або їх довгих описів.** Найчастіше ця властивість має на увазі, що не потрібно в кроках по відтворенню дефекту довго і по пунктах розписувати те, що можна замінити однією фразою.

• **Відсутність дублікатів.** Коли в проектній команді працює велика кількість тестувальників, може виникнути ситуація, при якій один і той же дефект буде описаний кілька разів різними людьми. А іноді буває так, що навіть один і той же тестувальник вже забув, що колись давно вже виявляв якусь проблему, і тепер описує її заново.

Уникнути подібної ситуації дозволяє наступний набір рекомендацій:

• Якщо не впевнені, що цей дефект був описаний раніше, зробіть пошук за допомогою вашого інструментального засобу управління життєвим циклом звітів про дефекти.

• Писати максимально інформативні словосполучення (тому що пошук в першу чергу проводять по ним). Якщо на вашому проекті накопичиться безліч дефектів з короткими описами в стилі «Кнопка не працює», ви витратите дуже багато часу, раз по раз перебираючи десятки таких звітів в пошуках потрібної інформації.

• Використовувати по максимуму можливості вашого інструментального засобу: вказуйте в звіті про дефект в компонентах програми, посилання на вимоги, розставляйте теги і т.д. - все це дозволить легко і швидко звузити в майбу-

тньому коло пошуку.

• Вказувати в докладному описі дефекту текст повідомлень від додатку, якщо це можливо. За таким текстом можна знайти навіть той звіт, в якому інша інформація наведена в занадто загальному вигляді.

• Якщо виявлено якусь додаткову інформацію, внести її в вже існуючий звіт про дефект (або попросіть зробити це його автора), але не створюйте окремий звіт.

• **Очевидність і зрозумілість**. Описувати дефект так, щоб у читає ваш звіт не виникло жодного сумніву в тому, що це дійсно дефект. Найкраще це властивість досягається за рахунок ретельного пояснення фактичного і очікуваного результату, а також вказівки посилання на вимогу в поле «Детальний опис».

• **Відстеження**. Зі змісту в якісному звіті про дефект повиненно бути зрозуміло, яку частину програми, які функції і які вимоги зачіпає дефект. Найкраще ця властивість досягається правильним використанням можливостей інструментального засобів управління звітами про дефекти: компоненти програми, посилання на вимоги, тест-кейси, суміжні звіти про дефекти, теги і т.д.

Деякі інструментальні засоби навіть дозволяють будувати візуальні схеми на основі таких даних, що дозволяє управління простежуваності навіть на дуже великих проектах перетворити з непосильним для людини завдання в тривіальну роботу.

• **Окремі звіти для кожного нового дефекту.** Існує два непорушних правила:

• **У кожному звіті описується рівно один дефект** (якщо один і той же дефект проявляється в декількох місцях, ці прояви перераховуються в докладному описі).

• **При виявленні нового дефекту створюється новий звіт.** Не можна для опису нового дефекту правити старі звіти, переводячи їх в стан «відкрито заново».

Порушення **першого правила** призводить до об'єктивної плутанини, яку найпростіше проілюструвати так: уявіть, що в одному звіті описано два дефекту, один з яких був виправлений, а другий - ні. У який стан переводити звіт? Невідомо.

Порушення **другого правила** взагалі породжує хаос: мало того, що втрачається інформація про раніше виниклих дефектах, так до того ж виникають проблеми зі всілякими метриками і банальним здоровим глуздом. Саме з метою уникнення цієї проблеми на багатьох проектах правом перекладу звіту зі стану

«закритий» в стан «відкрито заново» володіє обмежене коло учасників команди.

• **Відповідність прийнятим шаблонами оформлення і традиціям.** Як і у випадку з тест-кейсами, з шаблонами оформлення звітів про дефекти проблем не виникає: вони визначені наявним зразком або екранною формою інструментального засобу управління життєвим циклом звітів про дефекти. Але що стосується традицій, які можуть відрізнятися навіть в різних командах в рамках однієї компанії, то єдина порада: почитайте вже готові звіти про дефекти, перед тим як писати свої. Це може заощадити багато часу і сил.

Алгоритм створення ефективних звітів про дефекти

Зрозуміти суть проблеми.

Все починається саме з розуміння того, що відбувається з додатком. Тільки при наявності такого розуміння ви зможете написати по-справжньому якісний звіт про дефект, вірно визначити важливість дефекту і дати корисні рекомендації по його усуненню. В ідеалі звіт про дефект описує саме суть проблеми, а не її зовнішній прояв.

Відтворити дефект

Ця дія не тільки допоможе в подальшому правильно заповнити поле «Відтворюваність», а й дозволить уникнути неприємної ситуації, в якій за дефект додатку буде прийнятий якийсь короткочасний збій, який (швидше за все) стався десь у вашому комп'ютері або в іншій частині IT інфраструктури, яка не має стосунку до тестованого додатку.

Перевірити наявність опису знайденого вами дефекту

Обов'язково варто перевірити, чи немає в системі управління дефектами опису саме того дефекту, який ви тільки що виявили. Це проста дія, яка не застосовується безпосередньо до написання звіту про дефект, але значно скорочує кількість звітів, відхилених з резолюцією «дублікат».

Сформулювати суть проблеми

Формулювання проблеми у вигляді «що зробили (кроки по відтворенню), що отримали (фактичний результат в докладному описі), що очікували отримати (очікуваний результат в докладному описі)» дозволяє не тільки підготувати дані для заповнення полів звіту, а й ще краще зрозуміти суть проблеми.

У загальному ж випадку формула «що зробили, що отримали, що очікували отримати» ефективна з наступних причин:

• Прозорість і зрозумілість: слідуючи цій формулі, ви готуєте саме дані для звіту про дефект, а не абстрактні міркування.

• Легкість верифікації дефекту: розробник, використовуючи ці дані, може

швидко відтворити дефект, а тестувальник після виправлення дефекту упевнитися, що той дійсно виправлений.

• Очевидність для розробників: ще до спроби відтворення дефекту видно, чи дійсно описане є дефектом, або тестувальник десь помилився, записавши в дефекти коректну поведінку програми.

• **Позбавлення від зайвої безглуздою комунікації:** докладні відповіді на «Що зробили, що отримали, що очікували отримати» дозволяють вирішувати проблему і усувати дефект без необхідності запиту, пошуку та обговорення додаткових відомостей.

• **Простота**: на фінальних стадіях тестування із залученням кінцевих користувачів можна відчутно підвищити ефективність надходить зворотного зв'язку, якщо пояснити користувачам суть цієї формули і попросити їх дотримуватися її при написанні повідомлень про виявлені проблеми.

Заповнити поля звіту

Починати краще за все з детального опису, тому що в процесі заповнення цього поля може виявитися безліч додаткових деталей, а також з'являться думки з приводу формулювання стисненого та інформативного короткого опису.

Перечитати звіт (і ще раз перечитати звіт)

Після того як все написано, заповнене і підготовлено, ще раз уважно перечитайте написане. Дуже часто ви зможете виявити, що в процесі доопрацювання тексту десь вийшли логічні нестиковки або дублювання, десь вам захочеться поліпшити формулювання, десь щось поміняти.

Типові помилки при написанні звітів про дефекти

Написання звітів про дефекти вимагає уважності та чіткості, оскільки правильно сформульований звіт допомагає розробникам та тестерам в ефективному виправленні помилок. Однак при цьому можуть виникати різні помилки:

Неповне описання проблеми:

Помилка: Не надання достатньої кількості інформації для розуміння дефекту.

Рекомендація: Включати докладний опис проблеми, відомості про середовище та кроки для відтворення.

Неясні або загальні заголовки:

Помилка: Використання заголовків, які не відображають суті проблеми.

Рекомендація: Використовувати конкретні та описові заголовки, щоб легко зрозуміти суть дефекту.

Відсутність кроків для відтворення:

Помилка: Не надання конкретних кроків для відтворення дефекту.

Рекомендація: Включати послідовність кроків, яка дозволяє іншим відтворити проблему.

Невірна категорізація чи пріоритет:

Помилка: Неправильне визначення категорії чи надання неправильного пріоритету дефекту.

Рекомендація: Оцінювати дефекти згідно з їхнього впливу на систему та користувачів.

Відсутність скріншотів чи прикріплених файлів:

Помилка: Не долучення скріншотів або інших файлів, які можуть полегшити розуміння дефекту.

Рекомендація: Додавати відповідні скріншоти чи файли для наглядності.

Невірне визначення середовища:

Помилка: Не уточнення використаного середовища (операційна система, версія браузера тощо).

Рекомендація: Вказувати точні дані про середовище, де був виявлений дефект.

Надмірне використання технічного жаргону:

Помилка: Використання технічного жаргону, який може бути непонятним для інших учасників процесу.

Рекомендація: Спростовувати опис та уникайте технічного жаргону, якщо це можливо.

Відсутність визначення очікуваних результатів:

Помилка: Не вказання того, як повинна виглядати коректна робота після виправлення дефекту.

Рекомендація: Уточнювати очікувані результати після виправлення для уникнення непорозумінь.

ТЕМА 10. ОЦІНКА ТРУДОВИТРАТ, ПЛАНУВАННЯ І ЗВІТНІСТЬ

Взаємозв'язок (взаємозалежність) планування та звітності:

Без якісного планування не ясно, кому і що потрібно робити.

• Коли незрозуміло, кому і що потрібно робити, робота виконується погано.

• Коли робота виконана погано і не ясні точні причини, неможливо зробити правильні висновки про те, як виправити ситуацію.

• Без правильних висновків неможливо створити якісний звіт про результати роботи.

• Без якісного звіту про результати роботи неможливо створити якісний план подальшої роботи.

Планування (planning) - безперервний процес прийняття управлінських рішень та методичної організації зусиль по їх реалізації з метою забезпечення якості деякого процесу протягом тривалого періоду часу.

Завдання планування:

• зниження невизначеності;

• підвищення ефективності;

• поліпшення розуміння цілей;

• створення основи для управління процесами.

Завдання звітності:

• збір, агрегація і надання в зручній для сприйняття формі об'єктивної інформації про результати роботи;

• формування оцінки поточного статусу і прогресу (в порівнянні з планом);

• позначення існуючих і можливих проблем (якщо такі є);

• формування прогнозу розвитку ситуації і фіксація рекомендацій по усуненню проблем і підвищенню ефективності роботи.

Тест-план і звіт про результати тестування

Testplan (план тестування) – документ, що описує весь обсяг робіт з тестування, починаючи з опису тестованих об’єктів, стратегії, розкладу, критеріїв початку і закінчення тестування, до необхідного в процесі роботи обладнання, спеціальних знань, оцінки ризиків з варіантами їх вирішення.

Тест-план є важливою складовою будь-якого процесу тестування, оскільки містить усю необхідну інформацію, що описує цей процес. Але в більшості випадків Тест-план переважно має формальне значення, але, все ж, наявність надає багато переваг.

Залежно від конкретизації описуваних завдань, тест-план може мати два рівні деталізації: майстер тест-план і детальний тест-план.

Детальний тест-план містить завдання тестування для кожної команди, для кожного релізу або ітерації проекту. Детальний тест-план створюється або для декомпозованих частин проекту, або для невеликих проектів і складається з:

– переліку областей тестування з пріоритетами;

– стратегії тестування;

– переліку можливих ризиків;

– переліку необхідних ресурсів;

– плану виконання проекту.

Майстер тест-план створюється або для організації процесу тестування між декількома командами, які тестують один проект, але мають різні завдання, або для проекту, який складається з безлічі ітерацій, які пов’язує якась загальна інформація, повторення якої в кожному релізі займає надто багато часу.

Майстер тест-план містить:

– загальну інформацію про проект (посилання на документацію, багтрекер, і т.д.);

– положення, що описують процес тестування, закладу дефектів і т. д.;

– критерії готовності продукту до випуску.

Існують кілька шаблонів тест-планів (IEEE, RUP) [12].

До низькорівневих завдань планування в тестуванні відносяться:

• оцінка обсягу та складності робіт;

• визначення необхідних ресурсів і джерел їх отримання;

• визначення розкладу, термінів і ключових точок;

• оцінка ризиків і підготовка превентивних контрзаходів;

• розподіл обов'язків і відповідальності;

• узгодження робіт з тестування з діяльністю учасників проектної команди, що займаються іншими завданнями.

Якісний тест-план має більшість властивостей якісних вимог, а також розширює їх набір наступними пунктами:

• Реалістичність (запланований підхід реально виконаємо).

• Гнучкість (якісний тест-план не тільки модифікується з точки зору роботи з документом, але і побудований таким чином, щоб при виникненні непередбачених обставин допускати швидку зміну будь-якої зі своїх частин без порушення взаємозв'язку з іншими частинами).

• Узгодженість із загальним проектним планом і іншими окремими плана-

ми (наприклад, планом розробки).

Тест-план створюється на початку проекту і допрацьовується в міру необхідності протягом усього часу життя проекту за участю найбільш кваліфікованих представників проектної команди, задіяних у забезпеченні якості.

Відповідальним за створення тест-плану, як правило, є провідний тестувальник («тест-лід»).

Розділи тест-плану:

1. **Вступ (Introduction):** Огляд тест-плану, визначення його мети, обсягу та основних принципів тестування.

2. **Об'єкт тестування (Test Objectives):** Опис функціональності чи продукту, який буде тестуватися, та визначення цілей тестування.

3. **Стратегія тестування (Test Strategy):** Визначення загальної стратегії тестування, включаючи підходи до тестування, обрані методи, ресурси та графік тестування.

Ресурси (resources). В даному розділі тест-плану перераховуються всі необхідні для успішної реалізації стратегії тестування ресурси, які в загальному випадку можна розділити на:

- програмні ресурси (яке ПЗ необхідно команді тестувальників, скільки копій і з якими ліцензіями (якщо мова йде про комерційне ПЗ));
- апаратні ресурси (яке апаратне забезпечення, в якій кількості і до якого моменту необхідно команді тестувальників);
- людські ресурси (скільки фахівців якого рівня і зі знаннями в яких областях має підключитися до команди тестувальників в той чи інший момент часу);
- тимчасові ресурси (скільки по часу займе виконання тих чи інших робіт);
- фінансові ресурси (в яку суму обійдеться використання наявних або отримання відсутніх ресурсів, перерахованих в попередніх пунктах цього списку); у багатьох компаніях фінансові ресурси можуть бути представлені окремим документом, тому що є конфіденційною інформацією.

4. **Обладнання та програмне забезпечення (Test Environment):** Інформація про обладнання, програмне забезпечення, конфігурації та інше, яке використовується під час тестування.

5. **Критерії припинення тестування (Exit Criteria):** Умови, за якими тестування буде визнано завершеним, наприклад, досягнення певного рівня якості чи покриття.

6. **Розклад тестування (Test Schedule):** Графік проведення тестів, вклю-

чаючи дати початку і закінчення різних етапів тестування.

7. **Критерії оцінки ризиків (Risk Assessment):** Аналіз можливих ризиків та прийняття рішень щодо їх управління та контролю.

8. **Поділ обов'язків (Test Roles and Responsibilities):** Визначення обов'язків та відповідальностей учасників команди тестування.

9. **Види тестування (Testing Types):** Опис різних видів тестування, які будуть використані, таких як модульне, інтеграційне, системне, регресійне тестування та інші.

10. **Об'єкти тестування (Test Items):** Перелік елементів програми чи функціональних блоків, які будуть тестуватися.

11. **Критерії прийняття (Acceptance Criteria):** Умови, які повинні бути виконані для того, щоб вважати продукт готовим до випуску.

12. **Спроби тестування (Test Cases):** Посилання на або включення самого тест-кейсу чи сценаріїв тестування.

13. **Стратегія тестування безпеки (Security Testing Strategy):** Якщо тестування безпеки важливо для проекту, може бути вказана окрема стратегія для цього.

14. **Відповідальність за документацію (Documentation Responsibility):** Опис, хто та як буде відповідати за документування результатів тестування.

15. **Методи та інструменти тестування (Testing Methods and Tools):** Опис методів та інструментів, які будуть використовуватися під час тестування.

16. **Метрики (metrics).** Числові характеристики показників якості, способи їх оцінки, формули і т.д. На цей розділ, як правило, формується безліч посилань з інших розділів тест-плану.

Метрика (metric) - числова характеристика показника якості. Може включати опис способів оцінки та аналізу результату.

Щоб оперувати всіма цими числами (а вони потрібні не тільки для звітності, а й для організації роботи проектної команди), їх потрібно якось обчислити. Саме це і дозволяють зробити метрики. Потім обчислені значення можна використовувати для:

• прийняття рішень про початок, припинення, поновлення або припинення тестування;

• визначення ступеню відповідності продукту заявленим критеріям якості;

• визначення ступеню відхилення фактичного розвитку проекту від плану;

• виявлення «вузьких місць», потенційних ризиків і інших проблем;

• оцінки результативності прийнятих управлінських рішень;

• підготовки об'єктивної інформативної звітності.

У тестуванні існує велика кількість загальноприйнятих метрик, багато з яких можуть бути зібрані автоматично з використанням інструментальних засобів управління проектами. Наприклад:

• процентне відношення (не) виконаних тест-кейсів до всіх наявних;

• процентний показник успішного проходження тест-кейсів;

• процентний показник заблокованих тест-кейсів;

• щільність розподілу дефектів;

• ефективність усунення дефектів;

• розподіл дефектів за важливістю і терміновістю.

Варто згадати про так звані **«метрики покриття»**, тому що вони дуже часто згадуються в різній літературі.

Покриття (coverage) – відсоток ступеню, в якому досліджуваний елемент (coverage item) охоплений відповідним набором тест-кейсів.

Тестове Покриття – це одна з метрик оцінки якості тестування, що являє собою щільність покриття тестами вимог або виконуваного коду.

Один із підходів до оцінки та виміру тестового покриття – покриття вимог.

Цей показник показує поточний рівень покриття тестами всіх встановлених вимог до програмного забезпечення. Він є найбільш точним, коли встановлені вимоги є атомарними.

Цей показник розраховується за такою формулою:

*Тестове покриття вимог = (кількість вимог, що перевіряється тест-кейсами/загальна кількість вимог)*100%*

Інші метрики, які використовуються для оцінки якості тест-кейсів:

Продуктивність підготовки тест-кейсів (Test Case Preparation Productivity): використовується для розрахунку кількості підготовлених тест-кейсів та зусиль (Effort), витрачених на їхню підготовку.

Продуктивність підготовки тест–кейсів = Кількість тест-кейсів/Зусилля, витрачені на підготовку тест-кейсів

Охоплення тестового дизайну (Test Design Coverage): відсоток покриття тест-кейсами вимог.

*Охоплення тестового дизайну = (Загальна кількість вимог, відображених у тест-кейсах/Загальна кількість вимог)*100*

Продуктивність виконання тестів (Test Execution Productivity): визначає кількість тест-кейсів, які можуть бути виконані за годину.

Продуктивність виконання тестів = Кількість виконаних тест-

кейсів/Зусилля, витрачені на виконання тест-кейсів

Покриття виконаних тестів (Test Execution Coverage): призначене для вимірювання кількості виконаних тест-кейсів порівняно з кількістю запланованих тест-кейсів.

*Покриття виконаних тестів = (Загальна кількість виконаних тест-кейсів/Загальна кількість тест-кейсів,які планувалося виконати)*100*

Успішні тест-кейси (Test Cases Passed): для вимірювання відсотка пройдених успішно тест-кейсів.

*Успішні тест–кейси = (Загальна кількість успішних тест-кейсів/Загальна кількість виконаних тест-кейсів)*100*

Неуспішні тест-кейси (Test Cases Failed): для вимірювання відсотка завалених тест-кейсів.

*Неуспішні тест–кейси = (Загальна кількість неуспішних тест-кейсів/Загальна кількість виконаних тест-кейсів)*100*

Заблоковані тест-кейси (Test Cases Blocked): для вимірювання відсотка блокованих тест-кейсів.

*Заблоковані тест–кейси = (Загальна кількість заблокованих тест-кейсів/Загальна кількість виконаних тест-кейсів)*100 [13].*

Звіт про результати тестування

Звіт про результати тестування (test progress report, test Суммара report) - документ, що узагальнює результати робіт з тестування і містить інформацію, достатню для співвіднесення поточної ситуації з тест-планом і прийняття необхідних управлінських рішень.

Підсумковий звіт про якість перевіреного функціоналу є невід'ємною частиною роботи, яку кожен тестувальник повинен виконати після завершення тестування.

Звіт про результати тестування в першу чергу потрібен таким особам:

• **менеджеру проекту** - як джерело інформації про поточну ситуацію і основа для прийняття управлінських рішень;

• **керівнику команди розробників («dev-ліду»)** - як додатковий об'єктивний погляд на те, що відбувається на проекті;

• **керівнику команди тестувальників («тест-ліду»)** - як спосіб структурувати власні думки і зібрати необхідний матеріал для звернення до менеджера проекту з нагальних питань, якщо в цьому є необхідність;

• **замовнику** - як найбільш об'єктивне джерело інформації про те, що відбувається на проекті, за який він платить свої гроші.

Підсумковий звіт можна розділити на частини з відповідною інформацією:

1. Загальна інформація.
2. Відомості про те, хто і коли тестував програмний продукт.
3. Тестове оточення.
4. Загальна оцінка якості додатку.
5. Обґрунтування виставленої якості.
6. Графічне представлення результатів тестування.
7. Деталізований аналіз якості по модулях.
8. ТОП-5 найбільш критичних дефектів.
9. Рекомендації.

Далі розглянемо детальніше кожну частину підсумкового звіту.

Загальна інформація включає:

- назву проекту,
- номер збірки,
- модулі, які піддалися тестуванню (в разі, якщо тестувався не весь проект),
- види тестів по глибині покриття (Smoke Test, Minimal Acceptance Test,Acceptance Test), тестові активності (New Feature Test, Regression Testing, DefectValidation),
- кількість виявлених дефектів,
- вид робочої тестової документації (Acceptance Sheet, Test Survey, TestCases).

Відомості про те, хто і коли тестував програмний продукт, включають інформацію про команду тестування із зазначенням контактних даних та часового інтервалу тестування.

Тестове оточення містить: посилання на проект, браузер, операційну систему і іншу інформацію, конкретизують особливості конфігурації.

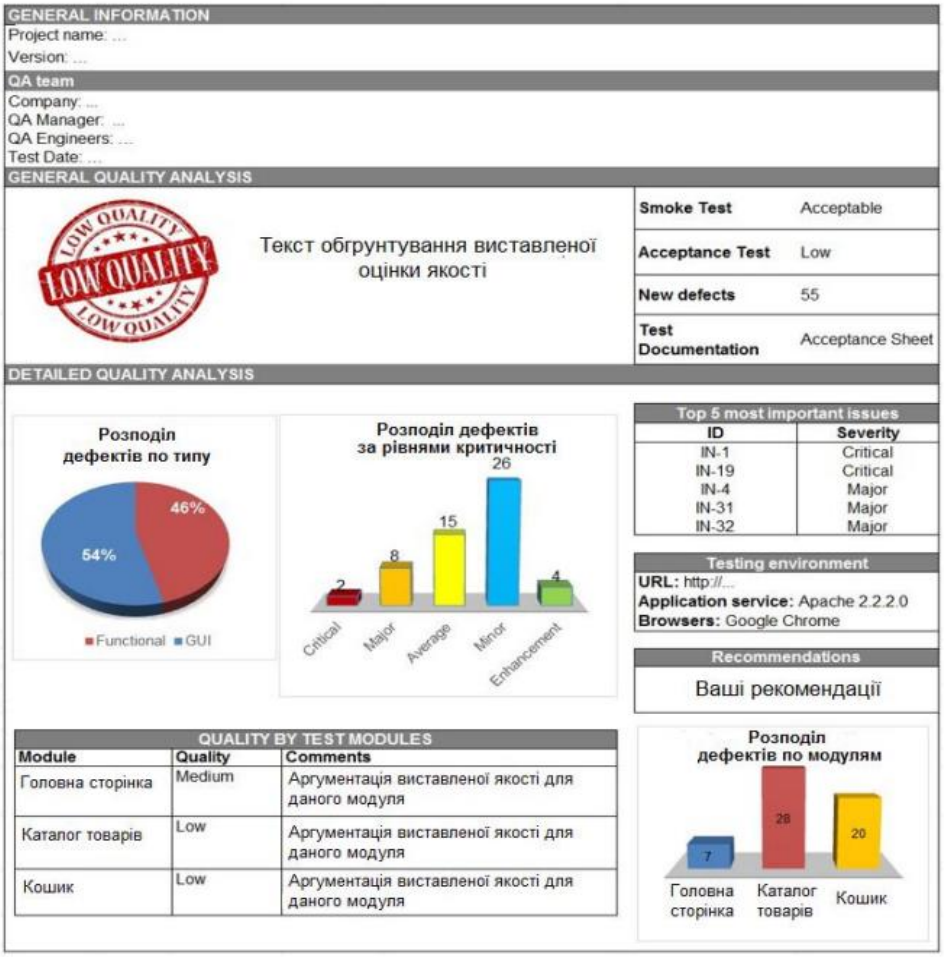

Рис. 10.1. Приклад підсумкового звіту про результати тестування

Загальна оцінка якості додатку виставляється на підставі загального враження від роботи з додатком і внесених дефектів (кількість, важливість). Обов'язково враховується етап розробки проекту – те, що не критично на початку роботи, стає важливим в момент випуску програмного продукту. Рівні якості: Високий (High), Середній (Medium), Низький (Low).

Обгрунтування виставленої якості є найбільш важливою частиною звіту, тому що тут відображається загальний стан збірки, а саме:

- якість збірки на поточний момент,
- фактори, що вплинули на виставлення саме такої якості збірки: вказівка функціоналу, який заблокований для перевірки, перерахування найбільш критичних дефектів і пояснення їх важливості для користувача або бізнесу замовника,
- аналіз якості перевіреного функціоналу: покращено воно або погіршилося в порівнянні з попередньою версією,
- якщо якість збірки погіршилося, то обов’язково повинні бути вказані регресивні місця,
- найбільш нестабільні частини функціоналу з вказанням причин, за яки-

ми вони такими є.

В даному розділі показується аналітична робота тестувальника, найбільш слабкі місця і найбільш критичні дефекти, динаміка зміни якості проекту.

Графічне представлення результатів тестування сприяє більш повному і швидкому розумінню текстової інформації.

Якщо необхідно продемонструвати процентне співвідношення, то доцільно використовувати кругові діаграми (наприклад, процентне співвідношення функціональних дефектів і дефектів GUI).

Стовпчасті діаграми краще підійдуть там, де важливо візуалізувати кількість дефектів в залежності від ступеня їх критичності або в залежності від локалізації (розподіл дефектів за модулями).

Відобразити в підсумковому звіті динаміку якості за всіма збірками найкраще вдасться за допомогою лінійного графіка.

Деталізований аналіз якості по модулях.

У цій частині звіту описується більш докладна інформація про перевірені частини функціоналу, встановлюється якість кожної перевіреної частини функціоналу (модуля) окремо, дається аргументація виставленого рівня якості.

Як правило, цей розділ звіту представляється в табличній формі. Залежно від виду проведених тестових активностей, ця частина звіту буде відрізнятися.

При оцінці якості функціоналу на рівні Smoke тесту, воно може бути або Прийнятним (Acceptable), або Неприйнятним (Unacceptable). Якщо всі найбільш важливі функції працюють коректно, то якість всього функціоналу нарівні Smoke може бути оцінено, як Прийнятне.

Якщо це релізний або предрелізна збірка, то для виставлення Прийнятної якості на рівні Smoke не повинно бути знайдено функціональних дефектів.

У частині про деталізовану інформацію якості збірки слід більш детально описати проблеми, які були знайдені під час тесту.

При оцінці якості функціоналу на рівні Defect Validation вказується якість про проведення валідації дефектів, а саме:

- загальна кількість всіх дефектів, які надійшли на перевірку;
- кількість невиправлених дефектів і їх відсоток від загальної кількості;
- список дефектів, які не були перевірені та причини, за якими цього не було зроблено;
- наочна таблиця з невиправленими дефектами.

За вищевказаними результатами виставляється якість тесту. Якщо відсоток невиправлених дефектів <10%, то якість Прийнятна (Acceptable), якщо> 10%, то

якість Неприпустима (Unacceptable). При оцінці якості функціоналу на рівні New Feature Test (повний тест нового функціоналу) якість окремо перевіреного функціоналу може бути: Висока (High), Середня (Medium), Низька (Low).

Важливо окремо вказувати інформацію про якість кожного модуля нового функціоналу з аргументацією виставленої оцінки.

При оцінці якості функціоналу на рівні Regression Testing потрібно аналізувати динаміку зміни якості перевіреної функціональності в порівнянні з більш ранніми версіями збірки. Для цього наводиться порівняльна характеристика кожної з частин функціонала в порівнянні з попередніми версіями збірки, даються ясні пояснення про виставлення відповідної якості кожної функції окремо. Також як і у попереднього виду тестів, якість цих може бути: Високою (High), Середньою (Medium), Низькою (Low).

ТОП-5 найбільш критичних дефектів містить список посилань на найбільш критичні дефекти із зазначенням їх назви та рівня критичності.

Рекомендації включають коротку інформацію про всі проблеми, характерних збірці, з поясненнями, наскільки не виправлені проблеми є критичними для кінцевого користувача. Обов'язково вказують функціонал і дефекти, якнайшвидше виправлення яких є найбільш пріоритетним. Крім того, якщо збірка є релізною або предрелізною, то будь-яке погіршення якості є критичним і важливо це позначити [14].

ТЕМА 11. АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ

Переваги і недоліки автоматизації:

• Швидкість виконання тест-кейсів може в рази і на порядки перевершувати можливості людини.

• Відсутній впливу людського фактору в процесі виконання тест-кейсів (втоми, неуважності і т.д.).

• Засоби автоматизації здатні виконати тест-кейси, в принципі непосильні для людини в силу своєї складності, швидкості чи інших факторів.

• Засоби автоматизації здатні збирати, зберігати, аналізувати, агрегувати і представляти в зручній для сприйняття людиною формі колосальні обсяги даних.

• Засоби автоматизації здатні виконувати низькорівневі дії з додатком, операційною системою, каналами передачі даних і т.д.

Отже, з **використанням автоматизації ми отримуємо можливість збільшити тестове покриття за рахунок:**

• виконання тест-кейсів, про які раніше не варто було й думати;

• багаторазового повторення тест-кейсів з різними вхідними даними;

• вивільнення часу на створення нових тест-кейсів.

Рис. 11.1. Залежність часу на розробку і відлагодження від типу тестування

З автоматизацією тестування пов'язана серія серйозних недоліків і ризиків:

• Необхідність наявності висококваліфікованого персоналу в силу того факту, що автоматизація - це «проект всередині проекту» (зі своїми вимогами, планами, кодом і т.д.). Навіть якщо забути на мить про «проект всередині проекту», технічна кваліфікація співробітників, що займаються автоматизацією, як правило, повинна бути відчутно вище, ніж у їхніх колег, які займаються ручним тесту-

ванням.

• Розробка і супровід як самих автоматизованих тест-кейсів, так і всієї необхідної інфраструктури займає дуже багато часу.

Ситуація ускладнюється тим, що в деяких випадках (при серйозних змінах в проекті або в разі помилок в стратегії) всю відповідну роботу доводиться виконувати заново з нуля: в разі відчутної зміни вимог, зміни технологічного домену, переробки інтерфейсів (як призначених для користувача, так і програмних) багато тест-кейсів стають безнадійно застарілими і вимагають створення заново.

• Автоматизація вимагає більш ретельного планування і управління ризиками, тому що в іншому випадку проекту може бути завдано серйозної шкоди.

• Комерційні засоби автоматизації стоять відчутно дорого, а наявні безкоштовні аналоги не завжди дозволяють ефективно вирішувати поставлені завдання. І тут ми знову змушені повернутися до питання помилок в плануванні: якщо спочатку набір технологій і засобів автоматизації був вибраний невірно, доведеться не тільки переробляти всю роботу, а й купувати нові засоби автоматизації.

• Засобів автоматизації вкрай багато, що ускладнює проблему вибору того чи іншого засобу, ускладнює планування і визначення стратегії тестування, може спричинити за собою додаткові часові та фінансові витрати, а також необхідність навчання персоналу або найму відповідних фахівців.

Отже, автоматизація тестування вимагає відчутних інвестицій і сильно підвищує проектні ризики, а тому існують спеціальні підходи по оцінці застосовності та ефективності автоматизованого тестування.

Якщо виразити всю їх суть дуже коротко, то в першу чергу слід врахувати:

• Витрати часу на ручне виконання тест-кейсів і на виконання цих же тест-кейсів, але вже автоматизованих. Чим відчутніше різниця, тим вигіднішою видається автоматизація.

• Кількість повторень виконання одних і тих же тест-кейсів. Чим воно більше, тим більше часу ми зможемо заощадити за рахунок автоматизації.

• Витрати часу на налагодження, оновлення та підтримку автоматизованих тест-кейсів. Цей параметр найскладніше оцінити, і саме він представляє найбільшу загрозу успіху автоматизації, тому тут для проведення оцінки слід залучати найбільш досвідчених фахівців.

• Наявність в команді відповідних фахівців і їх робоче завантаження. Автоматизацією займаються самі кваліфіковані співробітники, які в цей час не можуть вирішувати інші завдання.

Особливості тест-кейсів в автоматизації

Часто (а в деяких проектах і «як правило») автоматизації піддаються тест-кейси, спочатку написані простою людською мовою (і, в принципі, придатні для виконання вручну) - тобто звичайні класичні тест-кейси.

Рекомендації з підготовки тест-кейсів до автоматизації і безпосередньо самої автоматизації:

• Очікуваний результат в автоматизованих тест-кейсах повинен бути описаний гранично чітко із зазначенням конкретних ознак його коректності.

• Оскільки тест-кейс може бути автоматизований з використанням різних інструментальних засобів, слід описувати його, уникаючи специфічних для того чи іншого інструментального засобу рішень.

• У продовження попереднього пункту: тест-кейс може бути автоматизований для виконання під різними апаратними та програмними платформами, тому не варто спочатку прописувати в нього щось, характерне лише для однієї платформи.

• Однією з несподіваних проблем досі є синхронізація засобів автоматизації та тестової програми за часом: у випадках, коли для людини ситуація є зрозумілою, засіб автоматизації тестування може зреагувати невірно, «не дочекавшись» певного стану тестованої програми. Це призводить до завершення невдачею тест-кейсів на коректність роботи додатку.

• Не варто спокушати фахівця з автоматизації на вписування в код тест-кейсу константних значень (т.зв. «хардкодінг»). Якщо ви можете зрозуміло описати словами значення і / або сенс якоїсь змінної, зробіть це.

• По можливості слід використовувати найбільш універсальні способи взаємодії з тестованим додатком. Це значно скоротить час підтримки тест-кейсів в разі, якщо зміниться набір технологій, за допомогою яких реалізовано додаток.

• Автоматизовані тест-кейси повинні бути незалежними. З будь-якого правила бувають винятки, але в абсолютній більшості випадків слід припускати, що ми не знаємо, які тест-кейси будуть виконані до і після нашого тест-кейса.

• Варто пам'ятати, що автоматизований тест-кейс - це програма, і варто враховувати хороші практики програмування хоча б на рівні відсутності т.зв. «Магічних значень», «хардкодінга» і тому подібного.

• Варто уважно вивчати документацію по використовуваному засобу автоматизації, щоб уникнути ситуації, коли через невірно обраної команди тест-кейс стає хибно успішними, тобто успішно проходить в ситуації, коли додаток не

працює належним чином.

І ще одна особливість автоматизованих тест-кейсів заслуговує окремого розгляду - це джерела даних і способи їх генерації. Для мануальних тест-кейсів ця проблема не настільки актуальна, тому що при виконанні 3-5-10 раз ми також вручну цілком можемо підготувати потрібну кількість варіантів вхідних даних. Але якщо ми плануємо виконати тест-кейс 50-100-500 раз з різними вхідними даними, вручну стільки даних ми не підготуємо.

Джерелами даних в такій ситуації можуть стати:

• Випадкові величини: випадкові числа, випадкові символи, випадкові елементи з деякого набору і т.д.

• Генерація (випадкових) даних за алгоритмом: випадкові числа в заданому діапазоні, рядки довільної довжини із заданого діапазону з випадкових символів з певного набору (наприклад, рядок довжиною від 10 до 100символів, що складається тільки з букв), файли зі зростаючою по якомусь правилом розміром (наприклад, 10 КБ, 100 КБ 1000 КБ і т.д.)

• Отримання даних із зовнішніх джерел: вилучення даних з бази даних, звернення до якогось веб-сервісу і т.д.

• Зібрані робочі дані, тобто дані, створені реальними користувачами в процесі їх реальної роботи (наприклад, якби ми захотіли розробити власний текстовий редактор, тисячі наявних у нас і наших колег doc (x)-файлів були б такими робочими даними, на яких ми б проводили тестування).

• Ручна генерація - так, вона актуальна і для автоматизованих тест-кейсів. Наприклад, вручну створити по десять (і в тому числі і по 50-100) коректних і некоректних e-mail-адрес вийде куди швидше, ніж писати алгоритм їх генерації.

Використання автоматизованого тестування може бути доцільним у багатьох випадках, але важливо розуміти, коли саме воно може найбільше приносити користі. Ось деякі випадки, коли використання автоматизованого тестування є раціональним:

1. **Часті повторювані тести:** Якщо вам потрібно виконувати тести, які часто повторюються (наприклад, регресійне тестування після змін), автоматизація дозволяє швидко та ефективно виконувати тести без необхідності ручного втручання.

2. **Масштабність проекту:** Для великих проектів з великою кількістю функціональності та кодової бази, автоматизація тестування допомагає забезпечити високий рівень покриття тестами та зберігати стабільність системи.

3. **Тестування проектів з довгим циклом розробки:** У проектах з трива-

лим циклом розробки ручне тестування може виявитися витратним та часовим запізненням. Автоматизація дозволяє виконувати тести швидше та регулярно.

4. **Тестування великої кількості конфігурацій:** Якщо продукт підтримується на різних платформах, пристроях або браузерах, автоматизовані тести можуть допомогти забезпечити широке покриття тестами для різних конфігурацій.

5. **Тестування функціональності, що повторюється:** Якщо у вас є функціональність, яка часто змінюється або вимагає регулярного тестування, автоматизація дозволяє вам швидко виконувати тести для цієї функціональності.

6. **Тестування витривалості та навантаження:** Автоматизоване тестування витривалості та навантаження дозволяє штучно створювати навантаження на систему та визначати її межі стійкості.

7. **Перевірка безпеки:** Для тестування заходів безпеки, таких як перевірка вразливостей, сканування безпеки, автоматизоване тестування може допомагати виявляти та усувати проблеми безпеки.

8. **Регулярні випуски:** У випадку частих випусків програмного забезпечення ручне тестування може бути затратним та витратним. Автоматизація дозволяє швидко виконувати тести перед кожним випуском.

Існують ситуації та контексти, в яких використання автоматизованого тестування може бути не доцільним або не ефективним. Нижче наведено деякі випадки, коли може варто обмежити чи утриматися від автоматизованого тестування:

1. **Невеликий обсяг тестування:** Для проектів із невеликим обсягом функціоналу чи тестування, ручне тестування може бути ефективніше та економічніше.

2. **Часті зміни у функціоналі:** Якщо продукт часто зазнає істотних змін у функціоналі, витрати на постійне поновлення автоматизованих тестів можуть бути великими.

3. **Низька стабільність інтерфейсів:** Якщо інтерфейси програмного забезпечення часто змінюються, наприклад, у випадку частого рефакторингу коду, автоматизовані тести можуть вимагати постійних модифікацій.

4. **Відсутність чітких вимог:** У випадках, коли вимоги чи специфікації недостатньо чіткі чи часто змінюються, розробка та підтримка автоматизованих тестів може бути проблематичною.

5. **Тестування нових функцій або прототипів:** Під час розробки нових функцій або прототипів, коли важко передбачити зміни, ручне тестування може бути більш практичним.

6. **Тестування людської взаємодії (UI/UX):** Якщо потрібно тестувати аспекти користувацького досвіду (UI/UX), де важко або неможливо автоматизувати людську взаємодію або емоції, ручне тестування може бути кращим варіантом.

7. **Тестування для низькочастотних випадків використання:** У випадках, коли тестування націлене на низькочастотні випадки використання, де ручне тестування може бути ефективніше за автоматизоване.

8. **Невиправлення сталих помилок:** Якщо продукт має багато сталих помилок, автоматизовані тести можуть стати менш ефективними, оскільки вони будуть фіксувати ті самі проблеми без їх розв'язання.

9. **Низький рівень стабільності тестового набору:** Якщо автоматизовані тести часто видають помилки, їхне утримання та підтримка може стати витратним та неефективним.

Важливо враховувати, що рішення про використання автоматизованого тестування повинно ґрунтуватися на конкретних умовах проекту, його характеристиках та стратегії тестування.

Найпопулярніші мови програмування для автоматизованого тестування

Важливу роль також відіграє і мова програмування, що використовується в автоматизації. При оцінці будь-якої мови для автоматизації необхідно враховувати вісім основних моментів. Ці критерії спеціально оцінюють мову з точки зору чистоти і зручності використання, не обов'язково з точки зору нагальних потреб проекту.

Usability. Хороша мова автоматизації досить високорівнева і повинна виконувати такі складні завдання, як керування пам'яттю. Швидкість розробки також дуже важлива для термінів проекту.

Elegance. Процес перекладу тестового випадку в код повинен бути простим і зрозумілим. Для зручності обслуговування, тестовий код також слід робити коротким і самодокументованим.

Available Test Frameworks. Фреймворки забезпечують базові потреби, такі як налаштування/очищення, логування та звітність. Прикладами можуть бути Cucumber і xUnit.

Available Packages. Краще використовувати готові пакети для звичайних операцій, такі як веб-драйвери (Selenium), HTTP-запити і SSH.

Powerful Command Line. Хороший CLI полегшує запуск тестів. Це дуже важливо для безперервної інтеграції, коли тести не можуть бути запущені вруч-

ну.

Easy Build Integration. Автоматизація збірки повинна запускати тести і повідомляти про результати. Складна інтеграція – страшний сон розробників.

IDE Support. Тому що Notepad і vim просто не підходять для великих проектів.

Industry Adoption. Підтримка це важливо. Якщо мова залишається популярною, фреймворки і пакети будуть добре підтримуватися.

Далі будуть розглянуті найпопулярніші мови для автоматизації тестування.

Java 

Java – це мова програмування загального призначення, якою володіє корпорація Oracle. Java побудована на принципах об'єктно-орієнтованого програмування. Мова Java дотримується принципу WORA («Write Once, Run Anywhere»), який дає багато переваг для крос-платформного тестування.

Java використовується для підтримки внутрішніх корпоративних систем в багатьох великих корпораціях. Програми, написані на мові Java, працюють більш ніж на 3 мільярдах пристроїв. Незважаючи на те, що для юніт тестування найпопулярнішою платформою є JUnit, багато платформ для автоматизації тестування з відкритим вихідним кодом були розроблені на мові Java. Автоматизоване тестування в браузері для веб-сайту або веб-додатку можна виконати за допомогою JUnit з Selenium WebDriver.

У цій мові є як плюси, так і мінуси. З позитивного – це PageFactory, спрощує код для автоматизації та є можливість писати дуже прості для розуміння тести. У Java існує велике ком'юніті, нею володіють багато розробників і на ній вже написана величезна кількість інструментів. Внаслідок цього тестувальники часто мають можливість запитати поради у розробників і легше знайти готове рішення під певне завдання. З мінусів – код читається складніше, ніж навіть в Python. Ще одним мінусом є повідомлення про помилки, які часто складно зрозуміти.

Python 

Python також використовується в автоматизації тестування, пропонує відкритий вихідний код, використовується в машинному навчанні та багато іншого. Основною перевагою, яку Python має перед іншими мовами програмування для

автоматизації тестування, є легкість в освоєнні завдяки читабельності мови.

Згідно з дослідженням Stack Overflow Developer Survey (2019), колосальні 73,1% проголосували за Python як найбільш затребувану мову програмування, що вказує на популярність Python.

Ця мова програмування високорівнева, плюс має відмінну динамічну систему. Python найкраще підходить для автоматизації тестування.

Переваги Python:

- це мова загального призначення: Python використовується для вирішення практично будь-якої задачі розробки – десктопні і веб-додатки, аналіз даних, написання сценаріїв, автоматизація завдань і т. д .;
- батарейки в комплекті: багата стандартна бібліотека допоможе легко виконувати звичайні завдання розробки. Бібліотеки Selenium і Appium для Python полегшують роботу тестувальникам та автоматизаторам зокрема в крос-браузерному тестуванні на декстопних і мобільних девайсах. PyUnit і Pytest є найпопулярнішими середовищами автоматизації тестування в Python, які призначені для автоматизованого тестування на Selenium для проведення автоматичного крос-браузерного тестування;
- продуктивність кодування: будучи лаконічною мовою, Python дозволяє домогтися багато чого з меншою кількістю коду, що може заощадити час тестування;
- виконання скрипта: Python встановлено в системах Mac/Linux, і можна легко запустити скрипт Python з оболонки на серверах Linux;
- Python легше у вивченні, має гідну підтримку, велику сильну ком'юніті і відкритий вихідний код. Крім того, є безліч інструментів та модулів, які роблять процес простіше;
- Python дозволяє легше вловити суть тесту сторонній людині (завдяки своїй читабельності).

JavaScript 

За результатами досліджень Stack Overflow Developer Survey (2019), JavaScript зайняв перше місце в розділі «Programming, Scripting, and Markup Languages» в категорії «Most Popular Technologies».

JavaScript також є чудовою мовою програмування для автоматизації тестування, яка переважно використовується для фронтенд розробки. Багато великих споживацьких веб-сайтів використовують JavaScript для розробки інтерфейсу, і він однаково популярний для автоматизації тестування.

Однією з головних причин його популярності в автоматизації тестування може бути більш широке впровадження shift-left testing методології, коли розробники також беруть участь в розробці тестового коду. У shift-left testing методології команда тестувальників багато співпрацює з розробниками, що дозволяє реалізовувати автоматизоване тестування максимально ефективно.

JavaScript разом з Selenium також використовується для розробки тестових сценаріїв для автоматизованого тестування в браузерах. Його можна використовувати з віддаленою Selenium Grid, такий як LambdaTest, при цьому не буде необхідності проводити суттєві зміни у вихідному коді.

Однією з переваг, що робить JavaScript кращою мовою для автоматизації тестування, є широкий спектр середовищ тестування для End-to-End і юніт тестування. Деякі з кращих фреймворків автоматизації тестування на JavaScript:

- Jest;
- Mocha;
- Jasmine;
- Nightwatch.

Для автоматизації на JavaScript потрібно володіти базовими основами мови, навчитися роботі в Selenium, а також визначитися з фреймворком.

Переваги JavaScript:

- на JS тести можна писати набагато швидше, ніж на Java або C #;
- може давати більше взаємодії між членами команди;
- надає готові рішення для різних проблем;
- поріг входу для початку роботи на проекті досить низький.

З недоліків JavaScript можна виділити:

- рішення є менш стабільними;
- щоб написати дійсно хороші тести, потрібно глибоке розуміння того, як працює мова.

C# 

Створена Microsoft, C# також є однією з домінуючих мов у сфері автоматизації. Вона дотримується концепції ООП і є однією з найпоширеніших мов, що використовують .NET Framework. 67% респондентів в Stack Overflow Developer Survey (2019) вважають C# найбільш слушною мовою для автоматизації, програмування і багато чого іншого. C# в сфері автоматизованого тестування добре підходить для додатків на операційних системах Android, Windows та iOS.

Ця мова розробки повільно і неухильно набирає обертів в області автоматизованого тестування. Завдяки потужним можливостям мови і її сумісності з Selenium WebDriver, багато автоматизаторів схильні використовувати C# для тестування автоматизації та крос-браузерного тестування. Шаблон проектування Page Object Model (POM) дозволяє тестувальникам написати тестовий код ефективним і легко підтримуваним.

У C# розроблено багато платформ для автоматизованого тестування, створених допомогти в автоматизації тестування на Selenium або для автоматизованого тестування в браузерах. Також існує широкий ряд тестових середовищ, через що багато автоматизаторів використовують C# для написання тестових випадків для крос-браузерного тестування.

Найбільш часто використовувані платформи автоматизації тестування в C#:

- NUnit;
- MSTest;
- xUnit.Net.

C# має хорошу підтримку тестування, але вона живе в «міхурі» Microsoft. Інструменти розробки .NET не завжди безкоштовні, та операції командного рядка можуть проходити досить болісно.

Ruby 

Ruby також є мовою програмування, за допомогою якої можна писати автоматизовані тести. Ця мова зараз набирає популярності в сфері автоматизації та автоматизованому тестуванні в браузерах. Вона має відкритий вихідний код, є досить простою та продуктивною. Ruby, на зразок Python, дуже проста у вивченні і подальшій реалізації. Потужною мовою для програмування і автоматизації її робить те, що вона має простий і зручний для людини синтаксис та гнучку об'єктно-орієнтовану архітектуру.

У Ruby активно підтримується і зростає співтовариство його користувачів, яке вважається найбільш важливою і сильною стороною мови. Розробники поступово все більше віддають перевагу Ruby як мові програмування для розробки веб-додатків, так як за допомогою неї можна створити корисні хороші програми, використовуючи при цьому набагато менше число рядків коду.

Мова Ruby також дружить з платформою Selenium, тому тестувальники автоматизаторів можуть писати тести на Selenium. Навчитися роботі з мовою Ruby і платформою Selenium досить легко, і знадобиться зовсім невелика кіль-

кість рядків коду, щоб написати, наприклад, тести для крос-браузерного тестування.

Ruby і веб-фрейм Ruby on Rails є популярною інтерпретуємою мовою для стартапів. У ній є повне онлайн-навчання, як для початківців, так і для досвідчених програмістів, а також для фахівців в цій області. Високий рівень прийняття означає доступність численних веб-архівів та інструментів, які допомагають веб-розробникам швидко створювати додатки. У Ruby є джерело багаторазових архівів, які легко обслуговуються, а також розгорнуті в формі Ruby Gems. Як і Python, Ruby підходить для автоматизації від Puppet – інструменту управління начерками з відкритим вихідним кодом, спочатку написаного на мові Ruby.

В Ruby для допомоги в крос-браузерному тестуванні розроблений цілий ряд платформ для роботи тестувальників автоматизаторів. Нижче наведені найбільш поширені платформи:

- Capybara;
- RSpec;
- Test :: Unit.

Підводячи підсумки можна сказати, що Python і Java є сьогодні найкращими мовами для автоматизації. На Ruby можна швидко почати писати повноцінний фреймворк для тестування. Такий вид тестування обходиться дешевше і набагато швидше, ніж ручне тестування. У реальності відбувається так, що чим краще фахівець програмує, тим правильніше і точніше зможе автоматизувати. Вивчення будь-якої мови послужить гарною основою для технічного зростання [15].

ТЕМА 12. ТЕСТУВАННЯ ІНТЕРФЕЙСУ КОРИСТУВАЧА

Частина програмної системи, яка забезпечує роботу інтерфейсу з користувачем, – один з найбільш нетривіальних об’єктів для тестування.

З одного боку інтерфейс користувача – це частина програмної системи. Відповідно, на інтерфейс користувача пишуться функціональні та низькорівневі вимоги, за якими потім складаються тест-вимоги. При цьому, як правило, вимоги визначають реакцію системи на кожне введення користувача (за допомогою клавіатури, миші або іншого пристрою введення) та вигляд інформаційних повідомлень системи, що виводяться на екран, пристрій друку або інший пристрій виведення. При верифікації таких вимог мова йде про перевірку функціональної повноти інтерфейсу користувача – наскільки реалізовані функції відповідають вимогам, чи коректно виводиться інформація на екран.

З іншого боку інтерфейс користувача – «обличчя» системи, і від його продуманості залежить ефективність роботи користувача з системою. Фактори, що впливають на ефективність роботи, в меншій мірі піддаються формалізації у вигляді конкретних вимог до окремих елементів, однак повинні бути враховані у вигляді загальних рекомендацій та принципів побудови інтерфейсу користувача програмної системи. Перевірка інтерфейсу на ефективність людино-машинної взаємодії отримала назву перевірки зручності використання (usability verification).

Юзабіліті-тестування – дослідження, що виконується з метою визначення зручності деякого штучного об’єкту (веб-сторінка, інтерфейс користувача або пристрій) для його подальшого застосування. Таким чином, перевірка інтерфейсу користувача – це перевірка ергономічності об’єкту або системи. Перевірка ергономічності зосереджена на певному об’єкті або невеликому наборі об’єктів, у той час як дослідження взаємодії людинакомп’ютер в цілому формулюють універсальні принципи.

Перевірка ергономічності – метод оцінки зручності продукту у використанні, заснований на залученні користувачів в якості тестувальників, випробувачів і підсумовуванні отриманих від них висновків. Тому до перевірки ергономічності також відносять інсталяційне тестування.

Інсталяційне тестування – це вид тестування ПЗ, яке перевіряє чи система встановлена правильно та коректно працює на апаратному забезпеченні конкретного клієнта.

Функціональне тестування інтерфейсу користувача

Функціональне тестування інтерфейсу користувача складається з п’яти фаз:

1) аналіз вимог до інтерфейсу користувача;
2) розробка тест-вимог і тест-планів для перевірки інтерфейсу користувача;
3) виконання тестових прикладів і збір інформації про виконання тестів;
4) визначення повноти покриття вимог для інтерфейсу користувача;
5) складання звітів про проблеми в разі неспівпадання поведінки системи і вимог, або у разі відсутності вимог на окремі інтерфейсні елементи.

Типи вимог до інтерфейсу користувача

Вимоги до інтерфейсу користувача можуть бути розбиті на дві групи:

1) вимоги до зовнішнього вигляду і форм взаємодії з користувачем;
2) вимоги щодо доступу до внутрішньої функціональності системи за допомогою інтерфейсу користувача.

Іншими словами, перша група вимог **описує взаємодію підсистеми інтерфейсу з користувачем**, а друга – **з внутрішньою логікою системи**.

Вимоги до розміщення елементів управління на екранних формах

Дані вимоги можуть визначати загальні принципи розміщення елементів інтерфейсу користувача або вимоги до розміщення конкретних елементів.

При тестуванні даної вимоги досить визначити, що в кожному вікні системи дійсно присутні три частини, які розташовані і притиснуті згідно вимогам навіть при зміні розмірів вікна, при його згортанні/розгортанні, переміщенні по екрану, при перекритті його іншими вікнами.

Вимоги до змісту та оформлення виведених повідомлень

Вимоги до змісту та оформлення виведених повідомлень впливають на текст повідомлень, що виводяться системою, їх шрифтове і кольорове оформлення. Також часто в таких вимогах визначається, в яких випадках виводиться те або інше повідомлення.

Вимоги до форматів введення

Дана група вимог визначає, в якому вигляді інформація надходить від користувача в систему. При цьому крім власне вимог, що визначають коректний формат, до цієї групи належать вимоги, що визначають реакцію системи на некоректне введення. Для перевірки таких вимог необхідно перевіряти як коректне введення, так і некоректне. Бажано при цьому розбивати різні варіанти введення на класи еквівалентності (як мінімум на два – коректні й некоректні).

Вимоги до реакції системи на введення користувача

Даний тип вимог визначає зв’язок внутрішньої логіки системи та інтер-

фейсних елементів.

Вимоги до часу відгуку на команди користувача

В якості окремого типу вимог можна виділити вимоги до часу відгуку системи на різні операції користувача. Це пов'язано з тим, що підсвідомо користувач сприймає операції тривалістю більше 1 секунди як тривалі. Якщо в цей момент система не повідомляє користувачеві про те, що вона виконує якусь операцію, користувач почне вважати, що система зависла або працює в невірному режимі. У зв'язку з цим або всі граничні величини часу відгуку мають бути вказані у вимогах і в документації користувача, або під час тривалих операцій повинні виводитися інформаційні повідомлення (наприклад, індикатор прогресу).

Тестопридатність вимог до інтерфейсу користувача

Тестопридатність вимог до інтерфейсу користувача (UI) — це характеристика, яка визначає, наскільки легко вимоги можуть бути переведені в ефективні та інформативні тести для перевірки функціональності та коректності роботи інтерфейсу. Оцінка тестопридатності допомагає визначити, наскільки добре вимоги визначають тести та чи можна ефективно провести тестування.

Основні аспекти тестопридатності вимог до інтерфейсу користувача включають:

1. **Ясність та Зрозумілість:** Вимоги повинні бути чіткими та зрозумілими для всіх учасників процесу тестування. Це допомагає уникнути непорозумінь та дозволяє створювати ефективні тести.

2. **Коректність та Повнота:** Вимоги повинні бути коректними та повними, оскільки неправильно визначені або відсутні вимоги можуть призвести до неправильного тестування чи пропуску тестових сценаріїв.

3. **Визначення Критеріїв Прийняття:** Критерії прийняття повинні бути визначені чітко та точно, щоб тестери могли визначити, коли тести можна вважати успішними.

4. **Масштабованість та Резиновість:** Інтерфейс повинен бути готовим для масштабування та адаптації до різних розмірів екранів та пристроїв, щоб тести залишалися дієвими на різних платформах.

5. **Легкість Тестування:** Елементи інтерфейсу повинні бути легкими для ідентифікації та тестування. Це може включати унікальні ідентифікатори, доступ до DOM-елементів, атрибути та інші параметри.

6. **Доступність:** Вимоги до інтерфейсу повинні враховувати принципи доступності, щоб усі користувачі, включаючи тих, у кого є обмеження, могли взаємодіяти з програмою.

7. **Сумісність та Кросбраузерність:** Якщо програма підтримується на різних браузерах чи пристроях, вимоги повинні враховувати необхідність тестування на різних платформах.

8. **Ефективність використання Тестових Інструментів:** Вимоги повинні бути сформульовані так, щоб тестери могли легко використовувати автоматизовані тестові інструменти для валідації інтерфейсу.

9. **Стабільність вимог:** Вимоги до інтерфейсу повинні залишатися стабільними, оскільки часті зміни можуть ускладнити роботу тестувальників та вимагати частого апдейту тестових сценаріїв.

Важливо, щоб вимоги до інтерфейсу були спрямовані на покращення тестопридатності, щоб забезпечити ефективне та вичерпне тестування функціональності програмного

Тестування зручності використання інтерфейсу користувача

Тестування зручності використання інтерфейсу користувача (UI/UX тестування) спрямоване на оцінку того, наскільки ефективно та задовільно користувачі можуть взаємодіяти з програмним забезпеченням через його інтерфейс. Основні аспекти тестування зручності використання включають:

1. **Легкість Навігації:** Оцінка того, наскільки легко користувачі можуть переходити між різними частинами програми та виконувати необхідні дії.

2. **Чіткість та Зрозумілість Інтерфейсу:** Перевірка, наскільки чіткі та зрозумілі інструкції, підказки та мітки на елементах інтерфейсу.

3. **Ефективність Використання:** Оцінка того, наскільки швидко та ефективно користувачі можуть виконувати необхідні завдання з використанням програмного забезпечення.

4. **Відповідність Очікуванням Користувачів:** Перевірка того, наскільки інтерфейс відповідає очікуванням користувачів та чи відбувається взаємодія інтуїтивно.

5. **Доступність:** Оцінка доступності інтерфейсу для користувачів з обмеженими можливостями та дотримання принципів доступності.

6. **Гнучкість та Адаптивність:** Визначення, наскільки інтерфейс адаптується до різних розмірів екранів, пристроїв та налаштувань користувача.

7. **Зручність Введення Даних:** Перевірка, наскільки зручно користувачам вводити дані через інтерфейс, наприклад, форми та поля введення.

8. **Естетика та Дизайн:** Оцінка візуального вигляду інтерфейсу, дотримання дизайну та визначення його естетичної привабливості.

9. **Відсутність Помилок:** Перевірка наявності та обробки помилок в інте-

рфейсі, а також оцінка коректності повідомлень про помилки.

10. **Взаємодія з Різними Платформами:** Оцінка того, як інтерфейс взаємодіє з різними операційними системами та браузерами.

11. **Інтерактивні Елементи:** Перевірка роботи інтерактивних елементів, таких як кнопки, меню, вкладки, і переконання в їхній легкій взаємодії з користувачем.

12. **Тестування на Різних Роздільностях Екрану:** Перевірка, як інтерфейс виглядає та працює на різних роздільностях екрану.

Виконання функціонального тестування ІК можливо різними методами, як оператором, так і за допомогою інструментарію, який автоматизує виконання покрокових тестів.

Ручним тестуванням ІК – називається процес, коли тестувальник-оператор виконує покроковий перебір всіх можливих сценаріїв. Сценарій містить послідовність кроків оператора в ході тесту, результати тестування, які відображаються на інтерфейсі порівнюються з вимогами. Таблиця – це типова форма запису послідовності сценарію під час проведення ручного тестування. У першій колонці записується опис кроків сценарію (дії оператора), у другій – час очікування реакції системи, наступна колонка призначена для запису інформації, чи збіглась запланована відповідь системи з дійсною та перерахування розбіжностей. Зручність даного методу у тому, що коректність у зміні елементу інтерфейсу проводиться безпосередньо користувачем.

Автоматизоване тестування — це використання автоматизованих інструментів, які імітують поведінку «споживача», за ручним сценарієм тестування ІК. Такий метод використовується у випадках, коли вхідна інформація записана деякою мовою, а оператори відтворюють дії користувачів — переміщення курсора, введення команд та активізацію елементів інтерфейсу. Очікуваний стан ІК визначають різними методами. Наприклад, передача інформації в ІК і одержання інформації аналізу можуть бути виконані двома способами, якщо є доступ до елементів інтерфейсу:

1) Позиційний — це процес взаємодії з елементом інтерфейсу, який відбувається за допомогою задавання розмірів і точних координат.

2) Ідентифікатор — це взаємодія з елементами за допомогою одержання відповіді від інтерфейсу на основі його унікального ідентифікатора у межах вікна.

Головна мета розроблення ПЗ та ІК – це тестування надійності та комфорту для користувачів. Коли продукт задовольняє вимогам критерію, тільки тоді

передається клієнтам, якщо продукт не відповідає критеріям, він допрацьовується.

Аналіз функціонального тестування дозволяє сформувати базове уявлення про можливі методи тестування інтерфейсу користувача. Кожен із методів по своєму є продуктивним і необхідним в конкретних ситуаціях. Але не потрібно і забувати про недоліки кожного із методу. Наприклад, ручне тестування потребує об'ємні людські та часові ресурси. Цей недолік зазвичай проявляється при проведенні регресійного тестування.

Іншим недоліком ручного тестування є його залежність від людського сприйняття. У свою чергу, якщо виникає необхідність в зміні покрокового тестування, то це легко виконати додавши чи видаливши деякі пункти в сценарії. Автоматизоване тестування у свою чергу використовує позиційний метод тестування, що передбачає зміну значної частини сценаріїв в системі тестів при кожній зміні інтерфейсу системи, що безперечно є недоліком такого методу і свідчить про можливість його використання лише для систем зі сталим інтерфейсом.

Ідентифікаційний метод автоматизації тестування більш стійкий до змін розташування елементів інтерфейсу, але зміни тестових прикладів можуть знадобитись у випадку зміни логіки роботи елементів інтерфейсу. Тому ці два методи підходять для перевірки усіх елементів та самого інтерфейсу [16].

Чек-лист основних критеріїв якісного інтерфейсу:

Інтерфейс готового ПЗ повністю відповідає прототипам. Перевіряємо відповідність як візуально, так і вимірюючи компоненти інтерфейсу та відступи.

Відповідність фірмовому/глобальному стилю. Навіть якщо є документація, здоровій глузд ніхто не відміняв. Усі нові дизайни повинні вписуватися у існуючі: споріднена палітра кольорів, шрифти, стиль декорацій.

Відповідність основним правилам дизайну:

— Не робити більше ніж 3 різних шрифти на сторінці.

— Не робити більше ніж 3 розміри шрифту на сторінці.

— Використовувати однакові відступи для компонентів сторінки.

— Намагатись не використовувати шрифт менший аніж 12 px.

— Не використовувати яскравих кольорів та анімацій, що мерехтитять.

Респонсивність дизайну до розмірів. Перевіряємо відображення сторінки при різних розмірах вікна та екрану (залежить від типу ПЗ). Для усіх основних сучасних розмірів, верстка інтерфейсу повинна виглядати однаково добре.

Перевірка орфографії. Також відноситься до UI Testing (тестування інтерфейсів). Не забудьте її перевірити будь-яким доступним Вам програмним за-

безпеченням.

Спеціальні можливості. Якщо Ваша компанія міжнародна, то це може бути дуже важливо. До цього пункту відносяться можливості: зміни контрасту, перетворення тексту в мову, підписи до зображень та інше.

UI компоненти

Навігація — тестувальнику потрібно перевіряти у інтерфейсі всі меню, пункти меню та панель інструментів для навігації. Це потрібно робити як для клавіатури, так і для миші, щоб перевірити повну сумісність.

Форматування даних — переконайтеся, що все форматування є правильним наприклад, маскування та валідація введення дати, часу. Також таблиці та випадаючі списки повинні мати відсортовані дані.

Прокрутки — всі вертикальні та горизонтальні смуги, що використовуються при прокручуванні, повинні з’являтися лише там, де це необхідно.

Стандартні та комбінації швидкого доступу — всі ярлики та клавіші швидкого доступу повинні бути визначені, слід переконатися, що вони працюють правильно.

Труднощі в тестуванні інтерфейсу

Тестування інтерфейсу може зіткнутися з різними труднощами та викликами:

1. **Різноманітність платформ та пристроїв:** Інтерфейс повинен працювати коректно на різних операційних системах (наприклад, Android, iOS, Windows) та пристроях з різними характеристиками, розмірами екранів та роздільностями.

2. **Браузерна та платформенна сумісність:** Тестування інтерфейсу вимагає перевірки сумісності з різними веб-браузерами, версіями браузерів та платформами, оскільки вони можуть впливати на відображення та взаємодію елементів.

3. **Різноманіття екранів та роздільностей:** Різноманітність розмірів та роздільностей екранів на різних пристроях може призвести до проблем з відображенням та розташуванням елементів інтерфейсу.

4. **Взаємодія з реальними даними:** Тестування інтерфейсу пов'язане з обробкою та відображенням реальних даних, що може бути викликом, особливо коли динамічні дані часто змінюються.

5. **Навігація та взаємодія:** Важливо впевнитися, що користувач може легко навігувати і взаємодіяти з інтерфейсом. Спроектований інтерфейс повинен бути інтуїтивно зрозумілим, а навігація - легкою.

6. **Адаптація до різних мов та культур:** Інтерфейс повинен бути адапто-

ваний до різних мов та культур, забезпечуючи зручність та зрозумілість для користувачів з різних географічних областей.

7. **Тестування на різних пристроях:** Тестування інтерфейсу на різних пристроях може вимагати багато часу та ресурсів, оскільки кожен пристрій може відрізнятися за характеристиками та платформою.

8. **Тестування реакції на різноманітні події:** Інтерфейс повинен правильно реагувати на різноманітні події, такі як кліки, жести, введення з клавіатури, що може бути складним для відтворення в тестах.

9. **Тестування адаптивності:** Розробка адаптивного інтерфейсу, який коректно виглядає та працює на різних пристроях та екранах, вимагає додаткового зусилля у випробуванні.

10. **Забезпечення безпеки:** Тестування інтерфейсу також включає перевірку безпеки, особливо важливої для захисту особистих та конфіденційних даних користувачів.

ТЕМА 13. QA MOBILE (ТЕСТУВАННЯ МОБІЛЬНИХ ДОДАТКІВ)

Розробка мобільних додатків відіграє все більш важливу роль. На сьогоднішній день існує великий вибір мов програмування для розробки мобільних додатків. Це пов'язано з тим, що для різних мобільних пристроїв доводиться використовувати різні мови програмування, що обумовлене тим, що мобільні пристрої мають різні операційні системи (ОС). Цільова платформа (або платформи)– iOS, Android, WindowsPhone, BlackBerry - буде мати значний вплив на вибір мови програмування.

Тестування мобільних додатків суттєво відрізняється від тестування десктопного програмного забезпечення. Адже, монітор комп'ютера та телефон – пристрої зовсім різні, як на вигляд, розмір, внутрішнє наповнення, й по технічних характеристиках.

Отже, при розробці мобільних додатків, і при їх тестуванні, слід зважати на ряд моментів:

- на відміну від монітора комп'ютера, екран мобільних пристроїв може змінювати орієнтацію;
- існує певний список обов'язкових функціональних параметрів мобільних додатків, що приписуються кожним конкретним виробником пристроїв. Їм слідувати потрібно обов'язково;
- мобільний пристрій постійно знаходиться у русі, тому слід очікувати, випадкових дій на пристрої (якщо він не заблокований, якщо мокрими руками, чи в рукавичках натискаєш кнопки або хтось штовхає);
- девайс постійно перебуває в стані пошуку мережі;
- при тестуванні слід перевірити роботу програми на неоднакових швидкостях передачі даних;
- при розробці WEB/Mobile програм потрібно також врахувати перебування в різних погодних умовах. При різному світлі слід використовувати контрастні кольори;
- необхідно не забувати, що основним завданням, наприклад телефону, як й раніше є дзвінки, і додаток ніяк не повинен заважати цій головній, прямій функції пристрою;
- відмінні мобільні девайси мають на додачу ще всілякі примочки. І наповнення вашого застосунку, звісно повинно їм відповідати;
- якщо у вас є можливість під час тестування мобільних додатків, радимо нехтувати емуляторами. Справа в тім, що їх функціонал не завжди відпові-

дає усім реальним можливостям мобільного апарату;

Головні особливості тестування мобільних додатків, на які варто звернути увагу при функціональному і GUI тестуванні мобільних додатків:

Розмір екрану і touch-інтерфейс:

✓ зручний розмфір кнопок – щоб не треба було шукати їх на екрані, і легко потрапляти з першого, а не третього разу по них;

✓ швидкість відгуку елементів (натиснута кнопка повинна візуально виділятися)

Витік пам'яті:

✓ можна перевірити за допомогою програми Instruments (стандартний додаток MacOS)

✓ приділити увагу вікнам з великою кількістю інформації, при тривалому перебуванні користувача в додатку розростатиметься кеш.

Перевірка зображення на екранах і різних версія OS:

✓ коректне відображення різних елементів на екранах;

✓ установка програми на коректну версію OS;

✓ перевірити установку на всі можливі девайси;

✓ різні функції на девайсах: відсутність/наявність камери, автофокусу;

✓ відсутність/наявність GPS;

✓ роботу карт.

Перевірка роботи зворотного зв'язку:

✓ повідомлення при завантаженні контенту/прогрес;

✓ повідомлення про помилку доступу до мережі;

✓ наявність повідомлень при спробі видалити важливу інформацію;

✓ наявність екрану/повідомлення при закінченні процесу/гри, наприклад (екран Game over)

Перевірка роботи оновлень:

✓ перевірка різних шляхів оновлень (wifi, bluetooth, usb);

✓ перевірка роботи встановлених змін, місць, куди вони вносилися;

✓ переконатися в підтримувані оновлень старішими операційками, щоб елементи які на новій системі працюють добре не падали на старіших версіях.

Перевірка реакції програми на зовнішні переривання:

✓ вхідні/вихідні смс, дзвінки;

✓ розряд/вилучення батареї;

✓ відключення мережі/wifi;

✓ підключення кабелю, карти, зарядки.

Реклама в мобільному додатку:

- ✓ реклама не повинна перекривати кнопки управління додатком;
- ✓ реклама повинна мати доступну кнопку закриття, тому що найчастіше користувач її не шукає, а просто видаляє додаток з кінцями.

Перевірка локалізації:

- ✓ іншими словами на екрані повинно вистачити місця для тексту;
- ✓ дати повинні відповідати формату встановленого регіону;
- ✓ тимчасові налаштування повинні дотримуватися.

Перевірка енергоспоживання:

✓ необхідно перевіряти наскільки сильно додаток спустошує батарею пристрою;

бо, швидше за все користувач видалить програму, через яку мобільний пристрій доведеться підзаряджати занадто часто.

Встановлення з різних маркетів

✓ Чи правильно встановиться додаток, якщо його завантажити з Play Маркет, App Маркет, Android Маркет чи iTunes?

Умови офлайну

✓ Що відбуватиметься з додатком в режимі польоту чи в умовах відсутності підключення до інтернету?

Локація

✓ Чи правильно визначається географічне місце розташування, якщо ваш додаток інтегрується з сервісами, базованими на локації? Чи справно ця опція працює як в мережах даних, так і в бездротових мережах?

Соціальні медіа

✓ Чи встановлена політика конфіденційності та чи налаштовані сповіщення, якщо ваш додаток інтегрується з соцмережами і дозволяє юзерам використовувати їхні акаунти для реєстрації і входу?

Соціальні медіа

✓ Чи встановлена політика конфіденційності та чи налаштовані сповіщення, якщо ваш додаток інтегрується з соцмережами і дозволяє юзерам використовувати їхні акаунти для реєстрації і входу?

Швидкодія

✓ Яка мінімальна конфігурація необхідна для запуску програми належним чином? Чи не завадить ваш додаток роботі інших додатків?

Безпека

✓ Чи є які-небудь вразливі місця, такі, як шкідлива реклама чи будь-які

інші кібер-загрози? Чи захищений додаток від них?

Класифікація інструментів тестування мобільних додатків:

- ❑ Справжні реальні телефони, планшети ін. девайси.
- ❑ Хмарні рішення.
- ❑ Емулятори і симулятори
- ❑ Інструменти для ручного і автоматизованого тестування
- ❑ Драйвер (UI Automator, Espresso, Selendroid, Robotium), надбудова (Appium, Calabash), фреймворк (xUnit, Cucumber), (Xmarin, Ranorex).

Емулятори тестування мобільних додатків

Емулятори та симулятори є інструментами, які дозволяють розробникам та тестувальникам тестувати мобільні додатки без прив'язки до фізичних пристроїв. Вони імітують роботу мобільних пристроїв та їх операційних систем для забезпечення швидкого та зручного тестування. Ось деякі емулятори та симулятори для тестування мобільних додатків:

1. **Android Emulator:** Це офіційний емулятор від Google для тестування Android додатків. Він надає можливість емулювати різні версії Android, різні роздільності екрану та характеристики пристроїв.

2. **iOS Simulator:** Це інструмент для розробників, наданий Apple, який імітує роботу iOS пристроїв. Він доступний в середовищі розробки Xcode і дозволяє тестувати додатки на різних моделях iPhone та iPad.

3. **Genymotion:** Це емулятор Android-пристроїв, який є досить популярним серед розробників. Він пропонує велику кількість налаштувань, таких як розміри екранів, версії Android, розширені можливості тестування.

4. **Appium:** Appium не є емулятором в прямому розумінні, але це відкрите рішення для автоматизації тестів мобільних додатків. Він підтримує як Android, так і iOS, і може взаємодіяти з емуляторами та фізичними пристроями.

5. **Xamarin Test Cloud:** Відкрита платформа тестування, яка дозволяє вам використовувати емулятори для тестування Xamarin-додатків на різних пристроях та операційних системах.

6. **BrowserStack:** Це хмарна платформа для тестування, яка надає доступ до різних емуляторів та симуляторів для тестування на різних пристроях та браузерах.

7. **AWS Device Farm:** Amazon Web Services (AWS) пропонує послугу для тестування мобільних додатків на різних пристроях. Вона включає в себе емулятори та фізичні пристрої.

8. **TestProject:** TestProject - це безкоштовна платформа для автоматизації

тестування, яка підтримує тестування мобільних додатків. Вона може використовувати різні емулятори для виконання тестів.

9. **Kobiton:** Кобітон - це платформа для тестування, яка надає доступ до реальних фізичних пристроїв та емуляторів для тестування мобільних додатків.

10. Genymotion - емулятор Genymotion є простим у встановленні, дозволяє емулювати широкий набір реальних пристроїв з різними версіями ОС Android, він працює швидко і підтримує апаратне прискорення графіки. Російська мова інтерфейсу відсутня.

Є можливість емуляції основної та фронтальної камери, функції скрінкасти та віддаленого керування. Серед переваг – емуляція ADB.

Тут є інтеграція з популярними IDE (Android Studio, Eclipse) та імітація вхідних дзвінків, SMS, розряду батареї та багато інших функцій [17].

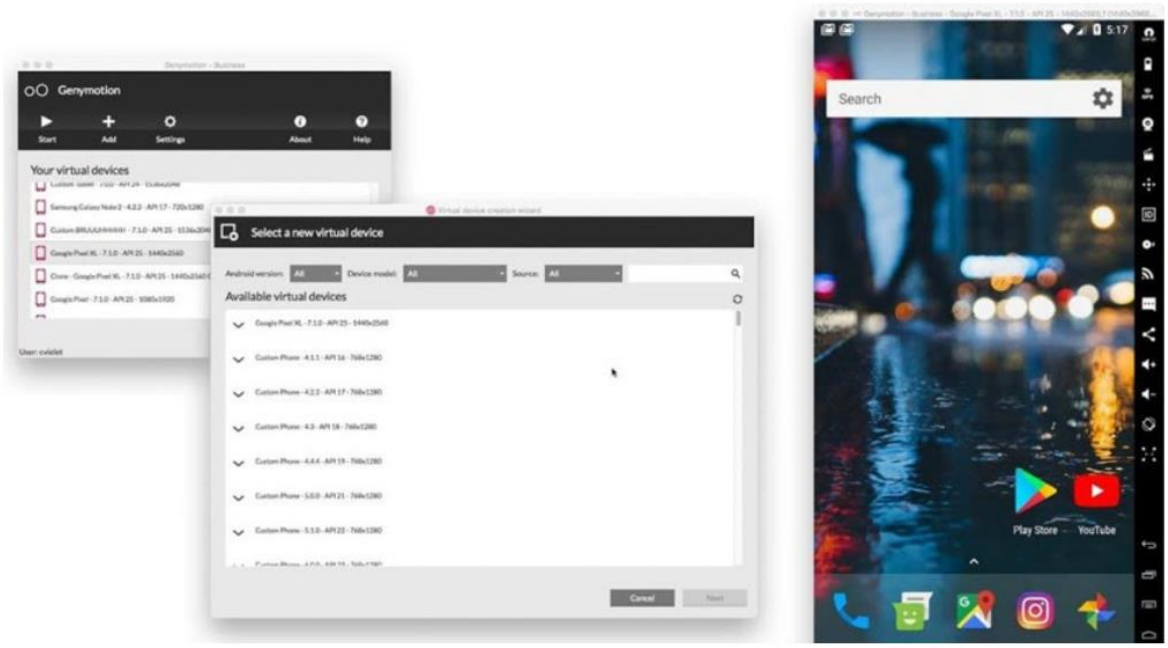

Рис. 13.1. Емулятор Genymotion

Хмарні технології та сервіси для мобільного тестування

Хмарні технології та сервіси для мобільного тестування надають можливість тестувати мобільні додатки в хмарному середовищі, що дозволяє ефективно виконувати тестування на різних пристроях та операційних системах. Ось деякі хмарні сервіси для мобільного тестування:

1. **Sauce Labs:** Sauce Labs є хмарним сервісом для автоматизації та ручного тестування мобільних та веб-додатків. Вони надають доступ до різних пристроїв та браузерів для тестування.

2. **Firebase Test Lab:** Firebase Test Lab - це частина Firebase, яка надає хма-

рні ресурси для тестування Android та iOS додатків. Вона дозволяє тестувати додатки на реальних пристроях в хмарі.

3. **Amazon Device Farm:** Amazon Device Farm - це сервіс для тестування мобільних додатків на різних пристроях, який надається Amazon Web Services (AWS). Він має велику кількість реальних фізичних пристроїв та емуляторів.

4. **BrowserStack:** BrowserStack надає можливість тестування мобільних додатків на реальних пристроях та емуляторах в хмарі. Вони також підтримують тестування на різних операційних системах.

5. **Kobiton:** Kobiton - це хмарна платформа для тестування мобільних додатків, яка надає доступ до реальних фізичних пристроїв та емуляторів.

6. **Perfecto:** Perfecto є хмарним сервісом для тестування мобільних та вебдодатків. Він надає велику кількість реальних пристроїв та емуляторів для тестування на різних платформах.

7. **TestProject:** TestProject - це безкоштовна платформа для автоматизації тестування, яка також надає можливість використання хмарних ресурсів для виконання тестів на різних пристроях.

8. **Experitest:** Experitest пропонує хмарні ресурси для тестування мобільних додатків на різних пристроях та платформах.

9. **Applause:** Applause надає сервіс для тестування якості програмного забезпечення, включаючи мобільні додатки, на різних пристроях та платформах.

Список використаної літератури

1. В*олкова С.О. Трунов О.М.* Валідність. [Електронний ресурс]: https://svitppt.com.ua/rizne/validnist.html

2. Цибульник, С. О. Технології розроблення програмного забезпечення. Частина 1. Життєвий цикл програмного забезпечення [Електронний ресурс] : підручник для здобувачів ступеня бакалавра за спеціальністю 151 «Автоматизація та комп'ютерно-інтегровані технології» / Цибульник С. О., Барандич К. С. ; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 3,43 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 270 с. – Назва з екрана. https://ela.kpi.ua/bitstream/123456789/50515/1/TRPZ_1.pdf

3. Кен Швабер та Джефф СазерлендПосібник зі Скраму.Повний навчальни й посібник зі Скраму: правила гри. Листопад 2020[Електронний ресурс]: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Ukrainian.pdf

4. Цибульник, С. О. Технології розроблення програмного забезпечення [Електронний ресурс] : підручник. Ч.1 : Життєвий цикл програмного забезпечення / С. О. Цибульник, К. С. Барандич. – Київ : КПІ ім. Ігоря Сікорського, 2022. – 270 с.

5. Старух А.І. Конспект лекцій з навчальної дисципліни «Методологія тестування програмного забезпечення». – ЛЬВІВ, 2020 [Електронний ресурс]: https://financial.lnu.edu.ua/wp-content/uploads/2019/09/konspekt-testuvannia.pdf

6. Особливості вимог програмного забезпечення. Методи тестування. Фази тестування. Класи еквівалентності. Навчальний ресурс з тестування програмного забезпечення [Електронний ресурс]: https://qalearning.com.ua/theory/lectures/material/requirements-testing-methods-equivalence/

7. Ромашкін Д.Д. Переваги та недоліки автоматизації тестування програмного забезпечення. Якість, стандартизація та метрологічне забезпечення: [матеріали II міжнародної науково-практичної конференції, Харків - 14-15 березня 2023 року] / за заг. ред. д.т.н., проф. Р. М. Тріща, к.т.н., доц. Г. С. Грінченко. Українська інженерно-педагогічна академія. Харків: УІПА, 2023. - 112 с.

8. Інженерія надійності програмного забезпечення. Матеріали лекцій: навч. посіб. / Укл.: Добровольський Ю.Г. Чернівці: ЧНУ ім. Ю.Федьковича, 2022. 126с.

9. Якість програмного забезпечення та тестування: базовий курс. Навчальний посібник / За ред. Крепич С.Я., Співак І.Я. / для бакалаврів галузі знань 12 «Інформаційні технології» спеціальності 121 «Інженерія програмного забезпе-

чення». – Тернопіль: ФОП Паляниця В.А., 2020. – 478с.

10. Олег Манжос. Дефекти, їх джерела та життєвий цикл. QA Україна. Українська спільнота тестувальників, 2023[Електронний ресурс]: https://qaukraine.online/defekty-ikh-dzherela-ta-zhyttievyj-tsykl/

11. Огляд популярних баг-трекінгових систем. QATestLab, 2023 [Електронний ресурс] : https://training.qatestlab.com/blog/helpful-materials/overview-of-popular-bug-tracking-systems/

12. Методичні вказівки щодо виконання лабораторних робіт з навчальної дисципліни «Основи аналізу якості програмного забезпечення» для студентів денної форми навчання зі спеціальності 123 – «Комп'ютерні системи та мережі».

13. Як дізнатися, що написаних тест-кейсів достатньо? QATestLab, 2023 [Електронний ресурс] : https://training.qatestlab.com/blog/technical-articles/have-you-written-enough-test-cases/

14. Методичні вказівки до практичних робіт по курсу «Тестування та верифікація ПЗ» / Коротенко Г.М., Коротенко Л.М., Шевцова О.С.; М-во освіти і науки України, Нац. техн. ун-т "Дніпровська політехніка". – Дніпро : НТУ «ДП»,– 2020. – 62 с.

15. Огляд мов програмування для автоматизованого тестування. QATestLab, 2020 [Електронний ресурс]: https://training.qatestlab.com/blog/technical-articles/overview-of-programming-languages-for-automated-testing/

16. Котлярчук Д. В. Аналіз методів тестування інтерфейсу користувача [Електронний ресурс] / Д. В. Котлярчук, О. В. Романюк // Матеріали XLIX науково-технічної конференції підрозділів ВНТУ, Вінниця, 27-28 квітня 2020 р. – Електрон. текст. дані. – 2020. – Режим доступу: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2020/paper/view/9764.

17. Емулятори, симулятори, ферми пристроїв для тестування мобільних додатків. QATestLab, 2023 [Електронний ресурс] : https://training.qatestlab.com/blog/technical-articles/emulators-simulators-farm-devices/

Навчальне видання

ШАБАЛА Євгенія Євгенівна

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМ

Конспект лекцій

Редагування та коректура *Є.Є. Шабала*
Комп'ютерне верстання *М.М. Власенко*

Підписано до друку 22.12.2022 Формат 60 х 84 1/ 16
Ум. друк. арк. 8,14 Обл.-вид. арк. 6,22
Електронний документ. Вид. № 10/І-16 Зам. 40/1-16

Видавець і виготовлювач
Київський національний університет будівництва і архітектури

Повітрофлотський проспект, 31, Київ, Україна, 03680

Свідоцтво про внесення до Державного реєстру суб'єктів
видавничої справи ДК № 808 від 13.02.2002 р.