**Лекція 14. Дефекти**

Чому буває так, що програми працюють неправильно? Все дуже просто – вони створюються та використовуються людьми. Якщо користувач допустить *помилку*, це може привести до проблем в роботі програми – вона виконається неправильно, а це означає, може повести себе зовсім не так, як очікувалось.

**Помилка (error)** – це дія людини, яка породжує неправильний результат.

Але програми розробляються та створюються людьми, які також можуть припускатись (і припускаються) помилок. Це означає, що недоліки є і в самому програмному забезпеченні. Вони називаються *дефектами* або *багами* (обидва значення рівносильні). Тут важливо пам’ятати, що програмне забезпечення – це дещо більше, ніж просто код.

Строге визначення

***Помилка (Еrror)*** – *хибне значення величини на виході системи або підсистеми, що викликане несправностями або збоями, яке, в свою чергу, може викликати відмову.*

З точки зору надійності ПЗ помилку можна розглядати, як упущення або неточність, що допущені проектувальниками ПЗ, програмістами, аналітиками та тестувальниками. Наприклад, проектувальник може неправильно зрозуміти завдання, а програміст – неправильно описати змінну тощо.

**Дефект, Баг (Defect, Bug)** – недолік компоненту або системи, який може призвести до відмови певної функціональності. Дефект, виявлений під час виконання програми, може викликати відмову окремого компоненту або всієї системи. В цьому питанні існують розбіжності у визначенні термінів.

***Несправність, дефект (Fault)*** – визнана або передбачувана причина помилки; наслідок відмови деякої системи, що обслуговувала або обслуговує в даний момент часу розглянуту систему. Дефекти також часто називають «***багами***» (від англ. bugs – жучки). Цей термін раніше використовувався, якщо вплив дефекту на роботу програми був незначний. Якщо ж помилка пов’язана із специфікаціями або архітектурою програми, то використовували слово «дефект».

Тепер термін «баг» – це сучасний сленговий вираз, що означає помилку проектування або розробки, що відбувається під час виконання програми. Існує кілька версій історії про те, хто першим використовував вираз «баг» у тому сенсі, в якому воно зараз вживається IT-спільнотою.

Найбільш усталеною легендою про походження слова *баг* вважається історія про те, як в 1947 році вченими Гарвардського університету під час тестування обчислювальної машини Mark II був знайдений метелик, який застряг між контактами електромеханічного реле. Знайдена комаха була вклеєна в технічний журнал з написом: «Перший фактичний випадок виявлення жука (бага)» (англ. «First actual case of bug being found» ). І саме цей вираз прийнято вважати першим вживанням слова *баг* стосовно до комп’ютерної техніки.

У тестуванні програмного забезпечення для позначення помилки прийнято користуватися терміном «дефект», однак сьогодні слово *баг* всюди використовується для позначення будь-якого роду помилок додатків на всіх етапах розробки.

Походження більшості помилок очевидно і, відповідно, їх виправлення не викликає складнощів. Але існують помилки, виявити які не так вже й просто (в силу різних причин). Виділяють такі «породи жуків», яких складно зловити:

1. «**Гейзінбаг**» (англ. heisenbug) – це програмна помилка, яка зникає або змінює свої характеристики при спробі її виявити. Як приклад гейзінбага можна навести помилки, які мають місце при звичайній компіляції, але пропадають в режимі налагодження (при компіляції за допомогою оптимизованого компілятора; іншими словами, при створенні дебаго-версії програми).

2. «**Борбаг**» (англ. bohrbug) – це така програмна помилка, поведінку якої визначено деякою кількістю певних (але, можливо, невідомих) умов. Баги такого типу не змінюють своєї поведінки і не зникають при спробі їх виявити, а також найчастіше зустрічаються серед складно-усунених. Однак вони можуть з’являтися тільки за певних умов (наприклад, якщо були введені деякі специфічні дані, або за певних налаштувань програми). Тому небезпека помилок даного типу полягає в тому, що вони можуть бути пропущені тестувальником, і про їх наявність стане відомо тільки через деякий час після початку експлуатації додатка кінцевим користувачем.

3. «**Шредінбаг**» (англ. schrodinbug ) – це програмна помилка, яка ніяк не проявляється (або є невідомою), до тих пір, поки хто-небудь не виявить її, прочитавши вихідний код або використавши програми в незвичайних (не передбачених) умовах. Після виявлення шредінбага, як правило, незрозуміло, як програма функціонувала до цього моменту (або просто здавалося, що вона функціонує).

4. «**Статистичний баг**» (англ. statistical bug ) – це така програмна помилка, яка може бути виявлена тільки при агрегації досить великої кількості результатів тестів. Іншими словами, окремі випробування (і навіть їх невелика кількість) не виявляють помилки – вона стає видна, тільки якщо розглядати велику кількість результатів одночасно. Даний тип помилок специфічний для програм, які виробляють випадковий або псевдовипадковий висновок. Прикладом може служити алгоритм випадковості, що виробляє нерівномірний висновок (тобто, більша частина вихідних значень зосереджена в якомусь окремому діапазоні). Дефект алгоритму буде не видно при малій кількості випробувань, але якщо провести достатню кількість випробувань і розглянути всі виходи разом, то помилковість алгоритму стане очевидна.

При виконанні коду програми дефекти, закладені ще під час його написання, можуть проявлятися: програма може не робити того, що повинна або навпаки – робити те, що не повинна, – відбувається збій.

**Збій (failure)** – невідповідність фактичного результату (actual result) роботи компоненту або системи очікуваному результату (expected result).

Збій в роботі програми може бути індикатором наявності в ній дефекту.

Таким чином, баг існує при одночасному виконанні трьох умов:

– відомий очікуваний результат;

– відомий фактичний результат;

– фактичний результат відрізняється від очікуваного результату.

Важливо розуміти, що не всі баги стають причиною збоїв – деякі із них можуть ніяк себе не проявляти та залишатися непоміченими (або проявлятися лише при дуже специфічних обставинах).

Причиною збоїв можуть бути не лише дефекти, але й умови навколишнього середовища: наприклад, радіація, електромагнітні поля або забруднення також можуть впливати на роботу як програмного, так і апаратного забезпечення.

Всього існує декілька джерел дефектів і, відповідно, збоїв:

– помилки в специфікації, дизайні, реалізації програмної системи;

– помилки у використанні системи;

– несприятливі умови навколишнього середовища;

– навмисне заподіяння шкоди;

– потенційні наслідки попередніх помилок, умов або навмисних дій.

Більш ретельне визначення:

**Збій, відмова, аварія**

***Збій (Malfunction)*** – перший прояв дефекту в роботі системи (проява несправності, зазвичай в роботі устаткування). Збої мають невелику тривалість в часі і можуть бути усунені без тривалих процедур відновлення. Як правило, збій викликає або короткочасну псування даних користувача без припинення роботи всієї системи в цілому. Наслідки збою можуть бути істотними з точки зору користувача, особливо якщо дані є критично важливими, однак безперебійна робота системи не порушується.

***Відмова (Failure)*** – це більш серйозний прояв дефекту в системі (або підсистеми), при якому вся система або її частина виходять з ладу, виходячи при цьому з працездатного стану, тобто стану в якому всі аспекти функціонування системи відповідають вимогам. Це така зміна робочих параметрів системи (стан), при якому знижується ефективність її функціонування нижче допустимого рівня, або повністю припиняється виконання функцій.

У разі відмови системи для її повернення до нормального функціонування потрібне втручання оператора. Для програмних систем причиною відмови може служити прихований дефект, що виявляються тільки з перебігом великого проміжку часу (переповнення внутрішнього лічильника часу, переповнення даних тощо).

Відмова може виникати внаслідок внутрішніх змін у системі (зміна параметрів всієї системи або її компонентів) або під впливом зовнішніх змін, по відношенню до системи, середовища. Відмова може бути раптовою або поступовою. При раптовій відмові характеристики системи змінюються стрибкоподібно, тоді як поступова відмова характеризується повільною, поступальною зміною параметрів системи, що створює труднощі в сенсі виявлення його причини.

Під час виконання програми або роботи всієї системи тестувальник, розробник або користувач можуть не отримати очікуваних результатів. У деяких випадках така поведінка – симптом помилки. Досвідчений розробник/тестувальник завжди зберігає базу даних помилок, з якими він стикався.

Некоректна поведінка може також означати неправильні значення вихідних даних, неправильний відгук пристрою або неправильне зображення на екрані. У процесі розробки відмови та баги зазвичай виявляються тестувальниками, а дефекти знаходяться і виправляються самими розробниками.

Несправність у коді не завжди веде до відмови. Насправді неправильна частина програми може функціонувати довгий час без прояву яких-небудь недоліків. Проте, за відповідних умов несправність може викликати відмову.

У сенсі допустимості відмов, всі системи можна розділити на дві групи:

* системи, які припускають можливість відмови при деяких передбачених умовах;
* системи, до яких пред’являються високі вимоги надійності, оскільки їх відмова може призвести до незворотних наслідків.

Наприклад, якщо відмовить текстовий редактор, то текст можна набрати заново, або відновити, але, якщо відмовить система навігації літака, то це може призвести до аварії. Звідси виникає поняття надійності, як властивості системи зберігати значення встановлених параметрів в заданих межах, що відповідають режимам і умовам функціонування.

У рамках забезпечення надійності застосовують ряд заходів, пов’язаних із забезпеченням відмовостійкості системи. Відмовостійкість визначається як властивість системи зберігати здатність коректно функціонувати після відмови.

***Аварія*** – відмова системи, при якому система виходить з ладу таким чином, що відновлення її працездатного стану або неможливо, або займає значний час. У разі програмних систем можна уникнути виникнення аварійних ситуацій за допомогою повного дублювання системи як виконуваного програмного коду, так і даних.

Збої і відмови є причиною відмовних ситуацій, в яких працездатний стан системи порушується тимчасово. Аварії є причиною аварійних ситуацій, тобто ситуацій, в яких працездатний стан системи порушується назавжди або на тривалий термін.

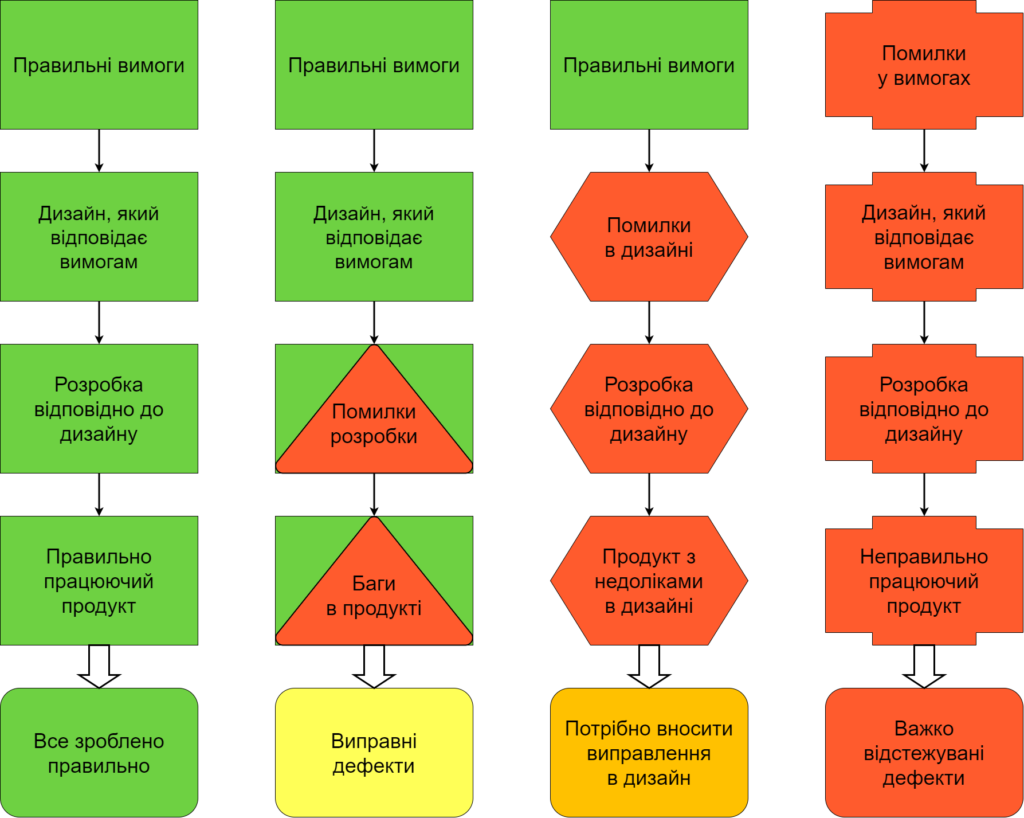
Дефекти можуть виникати на різних рівнях, і від того, чи будуть вони виправлені та коли, буде напряму залежати якість системи.

Розглянемо прояв дефектів з точки зору якості.

**Якість (Quality)** – ступінь, в якій сукупність присутніх характеристик відповідає вимогам.

**Якість програмного забезпечення (Software Quality)** – це сукупність характеристик програмного забезпечення, яка відображає його здатність задовольняти встановлені та допустимі вимоги. [ISO 8402:1994]

**Вимога (Requirement)** – потреба чи очікування, яке встановлено. Зазвичай передбачується або є обов’язковим.

[](https://qalight.ua/wp-content/uploads/2019/11/bz_wherebug_ua.png)

Для демонстрації залежності якості системи від дефектів, що привносяться в неї на різних рівнях процесу розробки, можна привести таку схему:

В першому випадку все було зроблено правильно і ми отримали продукт, що повністю відповідав очікуванням замовника та задовольняв критерії якості.

В другому випадку були припущені помилки вже в кодуванні, що призвело до появи дефектів в готовому продукті. На цьому рівні баги дуже легко виявити та виправити, оскільки ми бачимо невідповідність вимогам.

Третій варіант гірше – тут помилки були допущені на етапі проектування системи. Помітити це можна лише шляхом детального порівняння із специфікацією. Виправити такі дефекти також непросто – потрібно заново пропрацьовувати дизайн продукту.

В четвертому випадку дефекти були закладені ще на етапі формування вимог; вся подальша розробка і навіть тестування з самого початку пішли хибним шляхом. Під час тестування ми не знайдемо багів – програма пройде всі тести, але може бути забракована замовником.

Умовно, можна виділити п’ять причин появи дефектів в програмному забезпеченні.

1. Брак або відсутність спілкування в команді

Найчастіше, бізнес вимоги просто не доходять до команди розробки. У замовника є розуміння того, яким він хоче бачити готовий продукт, але якщо його ідею не пояснити розробникам та тестувальникам належним чином, результат може виявитися не таким, як очікувалось. Вимоги мають бути доступні та зрозумілі всім учасникам процесу розробки ПЗ.

1. Складність програмного забезпечення

Сучасне ПЗ складається із багатьох компонентів, які об’єднуються в складні програмні системи. Багатопоточні додатки, клієнт-серверна та розподілена архітектури, багаторівневі бази даних – програми стають все більш складні в написанні та підтримці, і тим складнішою стає робота програмістів. А чим складніше робота, тим більше помилок може допустити людина, що виконує її.

1. Зміна вимог

Навіть незначна зміна вимог на пізніх етапах розробки потребує великого об’єму робіт по внесенню змін в систему. Змінюється дизайн та архітектура додатку, що, в свою чергу, потребує внесення змін в початковий код і принципи взаємодії програмних модулів. Такі поточні зміни найчастіше стають джерелом дефектів, що складно відслідкувати. Тим не менше, вимоги, що часто змінюються в сучасному бізнесі – скоріше правило, ніж виняток, тому безперервне тестування та контроль ризиків в таких умовах – прямий обов’язок спеціалістів відділу забезпечення якості.

1. Погано задокументований код

Складно підтримувати та змінювати погано написаний або задокументований код. В багатьох компаніях існують спеціальні правила по написанню та документуванню кода програмістами. Хоча на практиці часто буває так, що розробники вимушені писати програми в першу чергу швидко і це впливає на якість продукту.

1. **Помилки програмістів**

Людський фактор проявляється у всьому. Програмісти теж страждають від неуважності, забудькуватості, випадкових натискань на клавіші або брак досвіду і умінь. А позитивний настрій розробника разом з його рівнем вміння програмувати безпосередньо пов'язаний з якістю написаного ним коду.

1. Середовище розробки ПЗ

Засоби візуалізації, бібліотеки, компілятори, генератори скриптів та інші допоміжні інструменти розробки – це також найчастіше програми, які погано працюють та слабко задокументовані – вони можуть стати джерелом дефектів в готовому продукті.

1. **Помилки тестувальників**

Помилки тестувальників можуть бути однією з причин того, чому баги в системі залишаються після релізу, а не з'являються спочатку. Тестувальники можуть не помітити присутні помилки через брак досвіду, упустити проблемні моменти в роботі програмного забезпечення через його складність і неможливість вичерпного тестування.

1. **Неякісний контроль версій коду**

Версійність коду програми – це сукупність виправлень в рамках однієї програми за певний період часу. Версії контролюються при релізах, оновленнях додатків. Повинна бути присутньою підтримка сумісності версій – можливість підтримки новими версіями програми старих версій, гарантії виконання коду програми, написаного на старій версії.

Недостатній контроль версій може призвести до появи великої кількості багів. Така ситуація трапляється, коли версії не збігаються через неякісний контроль мови програмування, на якому пишеться додаток, бібліотек, які використовуються в додатку, а також неякісний контроль версій модуля, з яких складається програма. Нові версії можуть не підтримувати старі версії і навпаки, саме через це виникають помилки.

1. **Архітектура програмного забезпечення**

Непродуманий вибір структурних елементів і інтерфейсів системи, їхньої поведінки в рамках співпраці з іншими структурними елементами, з'єднання обраних елементів і в цілому архітектури всіх елементів ведуть до проблем у функціонуванні ПЗ. Елементи можуть взаємодіяти не так, як очікує замовник, перебувати не там, де очікує користувач, порушуючи бізнес-логіку програми.

1. **Брак фінансування**

Програмний продукт буде протестований рівно настільки, наскільки було надано фінансів на процес тестування. Виділеного бюджету може не вистачити на достатнє тестування системи і, коли бюджет буде вичерпано, тестування зупиниться незалежно від результату тестування.

Підводячи підсумки, можна сказати, що основними причинами появи помилок в програмному забезпеченні є: людський фактор, тимчасові і фінансові обмеження, баги в інструментах для розробки, а також неможливість проведення вичерпного тестування, через що в програмному забезпеченні завжди залишаються помилки. Навіть, якщо здається, що помилок в ПЗ немає – вони там є. Тому, грамотне планування тестування, визначення пріоритетів для тестування і виправлення знайдених помилок – важливі основи якісного тестування ПЗ у встановлені терміни.

**Скільки коштують дефекти?**

Тут все просто: чим раніше ми знайдемо дефект, тим дешевше нам буде коштувати його виправлення. Зрозуміло, що виправити рядок в специфікації або коді простіше та дешевше, ніж вносити зміни в готовий продукт. Не кажучи вже про ситуацію, коли дефект знаходить замовник або кінцевий користувач – тут може постраждати репутація компанії-розробника, а такі збитки важко підрахувати.

Загальну тенденцію відображає графік:



З часом вартість виправлень виростає, в той час як вплив учасників проекту на процес розробки постійно зменшується.

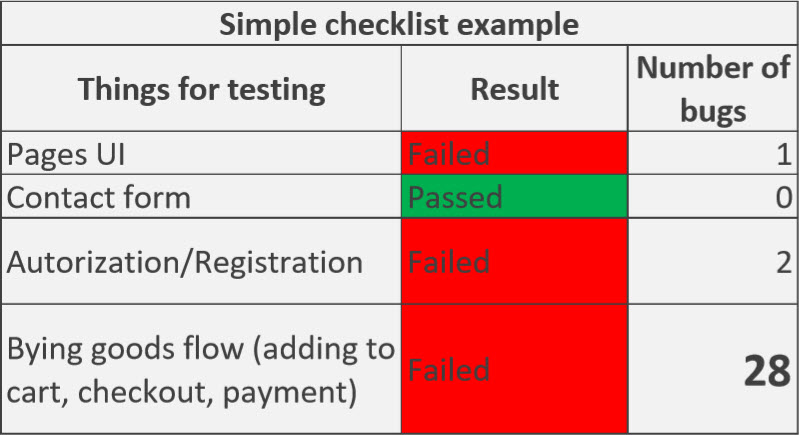
В крайньому разі, якщо серйозний дефект виявляється на пізніх етапах розробки, він може залишатися невиправленим, оскільки вартість внесення змін може виявитися надто високою. Окрім того, навіть якщо готова програма буде відповідати специфікації, замовник все рівно може відмовитися її приймати, якщо сама специфікація була неправильна.

Проектна команда могла розробити один в один те, що було описано у вимогах, але в тому разі, якщо вимоги були описані неправильно, замовника не влаштує результат.

## Скупчення дефектів або Bug clustering

Як один з 7 фундаментальних принципів тестування, скупчення дефектів є не таким вже й теоретичним поняттям. Більшість тих, у кого за плечима є досвід роботи на реальних проєктах, погодяться з цим твердженням. Хоча є проєкти, при першому погляді на які, здається, що цей принцип не відноситься до них, так як, дефекти розкидані по всіх куточках процесу розробки, тестування, організації і т. ін. Але все ж це більше виняток з правил.

**Скупчення дефектів** – принцип, який стверджує, що 80% всіх багів продукту міститься в 20% його складових (модулів).



***Приклад «Bug clustering» на проєкті***

Цей принцип заснований на емпіричній закономірності, названій в честь економіста та соціолога Вільфредо Парето, і яка говорить: *«20% зусиль дають 80% результату, а інші 80% зусиль – лише 20% результату».*

Але не варто сприймати це правило як мотивацію до хаотичного тестування з метою пошуку грааля, який містить ті самі заповітні 80% проблем, дозволить вам здобути славу мега тестувальника і внести ключовий внесок в реалізацію проєкту. Не будемо винаходити велосипед і просто процитуємо Вікіпедію, щодо недоліків закону Парето:

* у якісному формулюванні (тобто без урахування конкретних значень 20/80) закон являє собою тривіальне спостереження, згідно з яким в ситуації, коли результат складається з дій безлічі різних факторів, внесок цих факторів в результат часто буває різним;
* кількісна частина закону математично очевидно некоректна:
  + дійсний розподіл вкладу більшої і меншої частини чинників в реальному житті буває яким завгодно, і зовсім не обов'язково дорівнює 20/80;
  + легко перевірити, що конкретні значення розподілу змінюються навіть при аналізі одних і тих самих даних, досить змінити правила групування вибіркових значень.

Усе вищесказане лише підтверджує правило, яке свідчить, що всі проєкти різні: десь вплив цього закону вище і дуже помітно за підсумками спринту/ітерації, а десь нижче. Так, на проєкті з безліччю модулів, частина з яких стара, а нові створюються постійно, принцип скупчення багів буде добре помітний. У той же час продукт, що містить мінімум взаємодіючих модулів, матиме не таку виражену кластеризацию багів.

Закономірним питанням буде: «Навіщо ж тоді взагалі знати про цей принцип?». А для того, що час і гроші, як основні ресурси розробки програмного забезпечення, обмежені або лімітовані. Покрити всі модулі тестами – це щось на межі фантастики, а тому потрібно бути ефективними і професійно виявляти проблеми і помилки за відведений на це час.

Уявімо ситуацію, коли часу на написання якісних тест-кейсів немає і команда обмежена досить простим чеклістом. Тут і вступає в силу принцип Парето. Можна сміливо стверджувати, що як тільки знаходиться перший баг, перша проблема, перша невідповідність очікуваного результату – варто продовжувати шукати в цьому напрямку. По суті, це можна порівняти з пошуком схованки, де авантюрист, ( у нас тестувальник), постукує палицею по площині в пошуках порожнин за нею і після продовжує рухатися по знову знайденому шляху. Але це не означає, що потрібно кинути все і концентруватися всією командою на одному лише модулі, в якому знайшли перші баги, бо не варто забувати про недоліки цього правила, що були згадані вище.

З досвіду багатьох практикуючих тестувальників можна сказати, що наявність якісного ТЗ/вимог/товариських розробників, а також розуміння продукту і минулий досвід значно підвищують ефективність застосування даного закону.

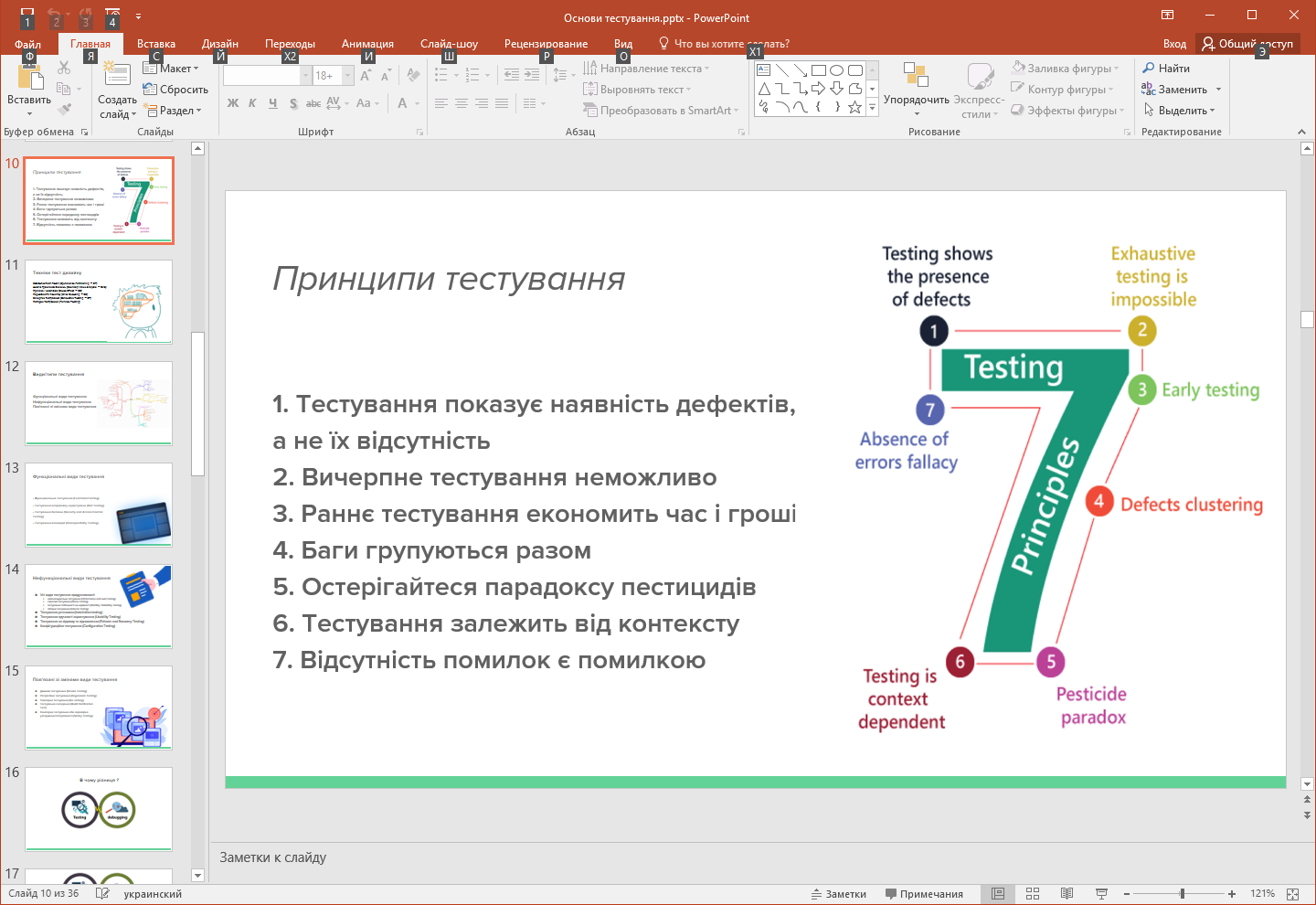
Таким чином, він ідеально доповнює основні ідеї Agile команди, дозволяє полегшити задачу, прискорити роботу на проєкті, а також зробити її більш ефективною.

Цікавою особливістю принципу «Скупчення дефектів» також є те, що, проаналізувавши дані за результатами тестів, можна виявити найпроблемніші модулі продукту і переглянути їх. Наприклад, якщо виявиться, що один з модулів на кожному спринті генерує левову частку багів або багів там трохи, але вони повертаються знову і знову, це відмінний привід задуматися над тим, чи варто далі намагатися реалізовувати його поточним чином? Це може бути як наслідком того, що з'являється конфлікт старих і нових модулів, приходу нових розробників, і таким чином невідповідності якості нового коду попередньому. Закон Парето дозволяє нам подивитися на вирішення проблеми з іншого боку: можливо нам варто переглянути ТЗ? Можливо простіше відмовитися від підтримки такого старого модуля і витратити сили на написання нового, який буде ідеально доповнювати поточні?

Але, здавалося б, до чого тут QA? До того, що позбавлення від такого модуля вирішить проблему куди ефективніше і простіше, ніж його тестування, яке з кожною ітерацією вимагає все більших зусиль команди тестувальників. Таким чином, принцип Парето допомагає покращити якість продукту, а QA команді приділити більше часу функціоналу, що залишився.

Виходячи зі статистики, отриманої незалежною міжнародною консультативною фірмою з досліджень в області IT, 45% функціоналу програми користувачі ніколи не використовують, 19% функціоналу використовується рідко, 16% – іноді, і тільки 20% – часто або постійно. На підставі цього можна припустити, що, сконцентрувавши більшу частину уваги на цих самих 20% часто використовуваних користувачами функціях, можна отримати куди більше вигоди, ніж інакше.

Основоположні принципи тестування спочатку можуть здатися занадто монолітними, непридатними. Лише аналізуючи пройдений етап, ми збираємо по крихтах воєдино мозаїку, роздивляючись яку, можна сказати, що принципи тестування ховаються у дрібницях, те, що здається ніяк не пов'язаним на перший погляд – в ретроспективі доповнює картину, яку вони описують. Не варто ними нехтувати, але також і не потрібно сподіватися застосувати їх в перший же робочий день. Чим більше досвід – тим частіше ці принципи застосовуються, навіть без думки про те, щоб використовувати їх, вони стають природними.



**Bug Report - склад**

**ID/ім’я** : будьте короткими та використовуйте правильні терміни. Найкраще вказати назву функції, у якій ви виявили проблему. Хорошим прикладом може бути «КОШИК – Неможливо додати новий товар у кошик»

**Опис/резюме:** якщо ви вважаєте, що назва недостатня, поясніть помилку кількома словами. Поділіться ним легкою для розуміння мовою. Майте на увазі, що ваш опис може бути використаний для пошуку у вашій програмі відстеження помилок, тому використовуйте правильні слова.

**Середовище:** залежно від вашого браузера, операційної системи, рівня масштабування та розміру екрана веб-сайти можуть по-різному поводитися в одному середовищі. Переконайтеся, що ваші розробники знають ваше технічне середовище.

**Вихідна інформація:** інформація яка допоможе швидше знайти місце помилки та саму помилку (кеш, посилання на сторінку, помилка в консолі …)

**Візуальний доказ:** зображення варте тисячі слів. Хоча цього може бути недостатньо, візуальний елемент, наприклад знімок екрана чи відео, допоможе вашим розробникам краще та швидше зрозуміти проблему.

**Дії для відтворення:** знімок екрана є доказом того, що у вас виникла проблема, але майте на увазі, що ваш розробник може не вдатися відтворити помилку. Обов’язково якомога докладніше опишіть кроки, які ви виконали до того, як виявили помилку.

**Очікувані та фактичні результати ( Expected vs. actual results ) -** поясніть, яких результатів ви очікували – будьте якомога конкретнішими. Просто сказати, що «додаток працює не так, як очікувалося», не має користі. Також корисно описати те, що ви насправді пережили.

**Severity and Priority:** Ви також можете включити додаткову інформацію, таку як серйозність (критична, велика, другорядна, тривіальна, покращення), пріоритет (високий, середній, низький), ім’я особи, яка повідомляє, призначену особу або термін виконання.

**Життєвий цикл помилок**

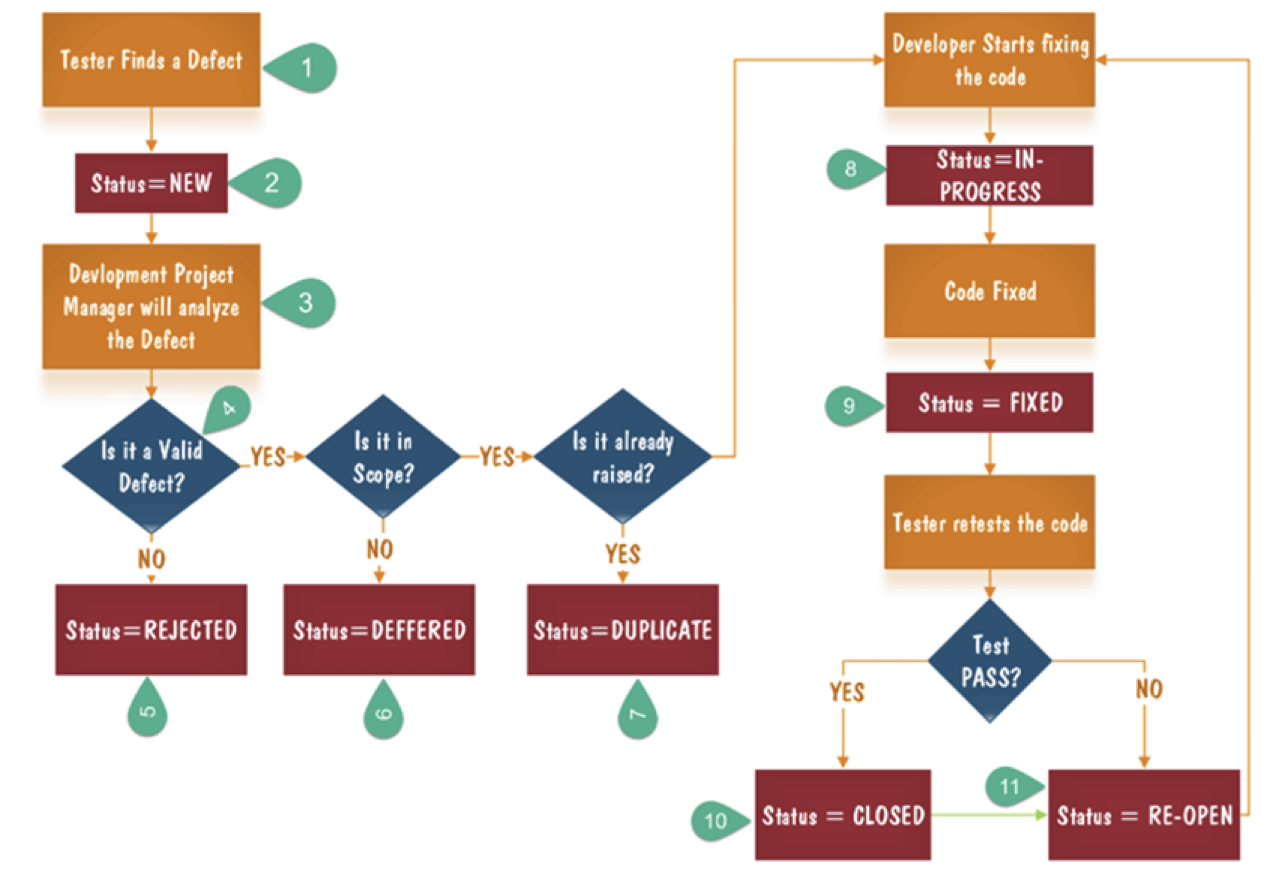


Схема 1 Життєвий цикл помилок

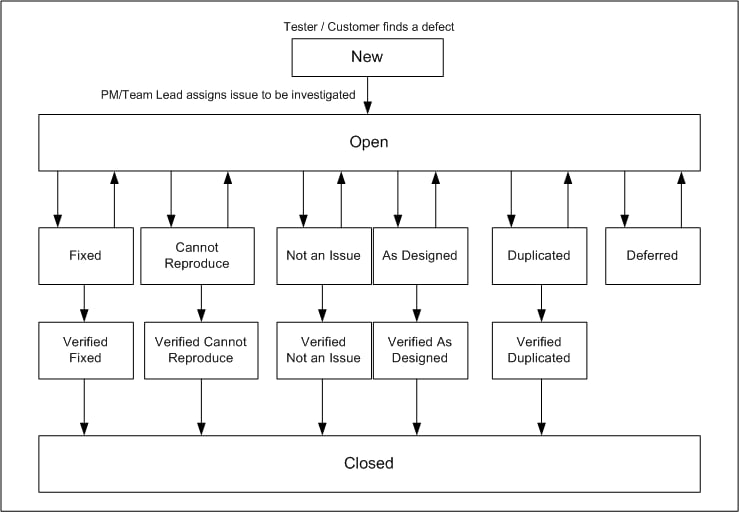


Схема 2 Життєвий цикл помилок

Загальний життєвий цикл виглядає так. Одразу, як опис дефекту створюється в системі трекінгу, він попадає в статус **New**.

Хто може створити дефект? Будь-хто. Той, хто зацікавлений в його виправлені: тестувальник, розробник, менеджер, клієнт. Коли дефект створений, у нього з’являється відповідальний (assignee). Це може бути безпосередньо розробник, чи хтось, хто приймає рішення про те, що робити з дефектом далі. Це може бути менеджер (PM), чи власник продукту (PO), чи хтось інший, хто приймає рішення про розвиток продукту.

Коли робота з дефектом почалася, він переходить в статус **Open**. А відповідальний змінюється на безпосередньо того, хто буде займатись його виправленням. А далі все залежить від того, як assignee оцінить проблему.

Значення статусів:

### Статуси Fixed / Verified Fixed

* **New** – знайдено нову помилку
* **Open** – дефект був переданий на аналіз
* **Fixed** – помилка виправлена програмістом
* **Verified Fixed** – помилка перевірена тестувальником після виправлення (дійсно виправлена)
* **Closed** – проблема закрита (після виправлення та тестування, документування або підтвердження неможливості відтворення)

Найкраще за все, коли знайдений дефект виявився справді дефектом чи багом. Після виправлення він переходить в статус Fixed. В статусі Fixed відповідальним стає той, хто додав дефект (тестувальник). Якщо після перевірки тестувальник бачить, що дефект не було виправлено, то цей дефект знов попадає в статус **Open**. І знов розробник стає відповідальним і виправляє дефект. І переводить його в статус Fixed. Якщо тестувальник не згоден, то виставляє його в статус Open. Якщо дефект виправлено, тоді тестувальник переводить його в статус **Verified Fixed**. В статусі Verified Fixed відповідальним стає той, хто переводить дефекти в статус Open. Після цього дефект зі статусу Verified Fixed переходить в статус **Closed**, де його життєвий цикл завершується.

### Статуси Can Not Reproduce / Verified Can Not Reproduce

* **Cannot Reproduce** – програміст не може відтворити дефект
* **Verified Cannot Reproduce** – тестувальник теж не може відтворити дефект

Рідко, але трапляються ситуації, коли дефект з’являється раз і більше не відтворюється (can’t reproduce). Це може бути викликано сукупністю кількох випадкових умов: стану пам’яті і якихось конфігурацій (наприклад). Якщо спробувати цей дефект відтворити знов, його більше не буде. В такому випадку розробник може встановити статус **Can Not Reproduce**. Якщо тестувальник дійсно не зможе після цього відтворити, то дефект переходить в статус **Verified Cannot Reproduce**. Частіше, такий результат з’являється, бо присутній недостатній опис дефекту. Якісь кроки були забути, чи не дуже явно описані. Особливості оточення, де знайдено дефект, вказані не повністю, чи вказані з помилкою. Коли тестувальник отримує дефект в статусі **Can Not Reproduce**, треба спочатку спробувати відтворити дефект на тому самому оточенні, де дефект був знайдений (та сама версія додатка, браузер чи операційна система) і перевірити, що всі деталі вказані в описі дефекту коректно. Якщо дефект досі можна відтворити, то статус змінюється на Open і послідовність кроків відтворення уточнюється.

### Статуси Not an Issue / Verified Not an Issue

* **Not an Issue** – знайдена проблема не була визнана як дефект
* **Verified Not an Issue** – тестувальник погоджується, oj описана проблема не є дефектом

Однією з причин, чому дефект попадає в цей статус може бути пов’язане з оточенням, апаратним забезпеченням або іншими факторами, які не пов’язані безпосередньо з програмою. У такому разі розробник може вважати, що проблема не є дефектом, а скоріше зовнішнім фактором, на який не можна вплинути. Наприклад – зберігання файлу об’єм якого перевищує очікування файлової системи, чи спроба зберегти файл в шлях, довжина якого перевищує 256 символів. Все це є обмеженнями оточення, в якому працює додаток. Якщо тестувальник згоден з тим, що описана поведінка — не дефект, то такий дефект переходить в статус **Verified Not an Issue**

### Статуси Duplicated / Verified Duplicated

* **Duplicated** – опис проблеми є дублікатом іншого опису тієї ж проблеми
* **Verified Duplicated** – тестувальник погоджується, що заведеній дефект є дублікатом вже існуючого дефекту

Коли команда велика, то знати хто і які дефекти знайшов. Тому, буває, що тестувальник описав дефект, який вже до нього описав хтось інший. І якщо розробник прочитав ваш дефект і зрозумів, що він вже такий десь бачив, то дефект опиняється в статусі **Duplicated**.

Головна умова переводу дефекту в статус Duplicated – це те, що в ньому є номер дефекта, який розробник вважає дублікатом. Уважно подивись на зміст цього дефекту, чи він про те саме? Якщо ні, вертай його в статус Open. Якщо це той самий дефект – **Verified Duplicated**.

### Статуси As Designed / Verified As Designed

* **As Designed** – програміст вважає, що проблема не є помилкою, а є особливістю дизайну
* **Verified As Designed** – тестувальник погоджується, що описана проблема – особливість дизайну

Це найсуперечливіша ситуація. Вона означає, що розробник не вважає описану проблему дефектом. Тоді такий дефект опиняється в статусі **As Designed**

Ситуація може виникати по багатьом причинам. Іноді навіть ліньки щось виправляти, але частіше розробники і тестувальники по різному розуміють очікувальний результат і тому можуть чекати різну поведінку. Що тут треба зробити, це уточнити очікуваний результат у менеджера чи РО. І якщо він не такий, як очікував тестувальник, то цей дефект переходить в статус **Verified As Designed**

### Статус Deferred

* **Deferred** – помилка, можливо, буде виправлена, але виправлення помилки відкладено

Дефект в статусі **Deferred** значить, що опис приймається, як дефект, але цей дефект не є важливим та не потребує негайного виправлення. Або, виправлення дефекту не може бути виконано зараз через якісь технічні труднощі чи відсутність можливості виправлення.

Такі дефекти не закриваються (Сlosed), а лишаються в статусі Deferred до закінчення циклу розробки (випуску версії на ринок). Якщо продукт йде на другу версію, спочатку розглядаються дефекти, відкладені в попередній версії.

На кінець проекту дефекти можуть бути в двох статусах: **Closed** або **Deferred**. Ми розглянули загальний набір статусів життєвого циклу дефекту, який не належить до якоїсь конкретної системи відстеження дефекту.

IT компанії на своїх проектах використовують системи відстеження дефектів, з певними статусами та перехідами (transitions) поміж цими статусами. Але загальний підхід буде той самий.

**Питання для самоконтролю:**

1. Які можуть бути наслідки не виявлення дефектів у тестуванні?
2. Чи можна уникнути дефектів у тестуванні?
3. Як описувати дефекти у тестуванні?
4. Які інші методи виявлення дефектів у тестуванні існують, крім проведення тестів?
5. Чи можна перевищити ліміт дефектів у тестуванні?