О. С. Губка, С. О. Губка

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

О. С. Губка, С. О. Губка

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Конспект лекцій

Харків «ХАІ» 2024

УДК 004.054
Г28

Рецензенти: д-р техн. наук, проф. С. І. Чалий,
д-р техн. наук, проф. І. П. Гамаюн

Губка, О. С.

Г28 Тестування програмного забезпечення [Електронний ресурс] : консп. лекцій / О. С. Губка, С. О. Губка. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2024. – 56 с.

Наведено основні лекції з мануального тестування програмного забезпечення. Розглянуто термінологію, принципи, цілі та завдання тестування, моделі та методології розроблення програмного забезпечення. Подано детальну інформацію про види тестування та принципи складання баг-репортів.

Для студентів спеціальностей 122 «Комп'ютерні науки», 126 «Інформаційні системи та технології», які навчаються за освітніми програмами «Комп'ютеризація обробки інформації та керування», «Розподілені інформаційні системи».

Іл. 8. Табл. 4. Бібліогр.: 7 назв

УДК 004.054

ЗМІСТ

ВСТУП

Тестування відіграє важливу роль у процесі розроблення і створення якісного програмного забезпечення. Необхідно серйозно ставитися до аналізу і проєктування структурованого процесу, який забезпечить своєчасний і успішний випуск проєкту.

Тестування є необхідним, оскільки всі роблять помилки. Деякі з помилок можуть бути незначними, водночас інші – мати негативні наслідки. Усе, що створюється людиною, може містити помилки (так званий людський фактор). Саме тому будь-який продукт потребує перевірки – тестування, перш ніж його можна буде ефективно та безпечно використовувати. Те саме стосується програмного забезпечення.

Програмне забезпечення (Software) – комп'ютерні програми, функції, а також їх документація та дані, що стосуються експлуатації комп'ютерної системи.

Комп'ютерні технології стають все більш поширеними у всіх сферах щоденного життя людини. Програмне забезпечення керує роботою багатьох речей навколо нас – від мобільних телефонів і комп'ютерів до пральних машин і кредитних карт. У будь-якому разі нам усім траплялися ті чи інші помилки в програмах: уповільнена робота текстового редактора під час написання дипломного проєкту, банкомат, який «з'їв» картку, чи сайт, що не завантажується, – усе це зовсім не полегшує життя.

Не всі помилки, проте, однаково небезпечні – для різних програмних систем рівні ризику можуть відрізнятися.

Ризик (risk):

– фактор, який може призвести до негативних наслідків у майбутньому; зазвичай виражається через вірогідність виникнення таких наслідків і їх подальший вплив на систему;

– те, що ще не відбулось і може взагалі не відбутися – потенційна проблема.

Рівень ризику, окрім того, буде залежати від вірогідності виникнення негативних наслідків.

Одна й та сама незначна помилка, наприклад друкарська, може мати абсолютно різні рівні ризику для різних програм: в описі інтересів на особистій сторінці в соціальній мережі навряд чи буде мати серйозні наслідки, хіба що викличе посмішку друзів; в описі діяльності великої компанії, розміщеної на її сайті, уже небезпечна, оскільки неопосередковано свідчить про непрофесіоналізм її співробітників; у коді програми, яка підраховує рівні опромінення під час роботи рентгенівського апарата (100 замість 10), може мати найневтішніші наслідки – шкода, завдана здоров'ю та безпеці людей, призведе до втрати довіри до компанії та судових позовів зі значними витратами.

Важливо пам'ятати, що довіру користувачів дуже просто втратити, а виправити допущені помилки може коштувати дорожче, ніж спочатку провести повну підготовку й тестування.

Лекції № 1, 2. ВСТУП ДО ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1. Основні визначення

Якість програмного забезпечення (Software Quality) – це сукупність параметрів програмного забезпечення, які стосуються його здатності задовольняти встановлені та передбачувані потреби (рис.1).

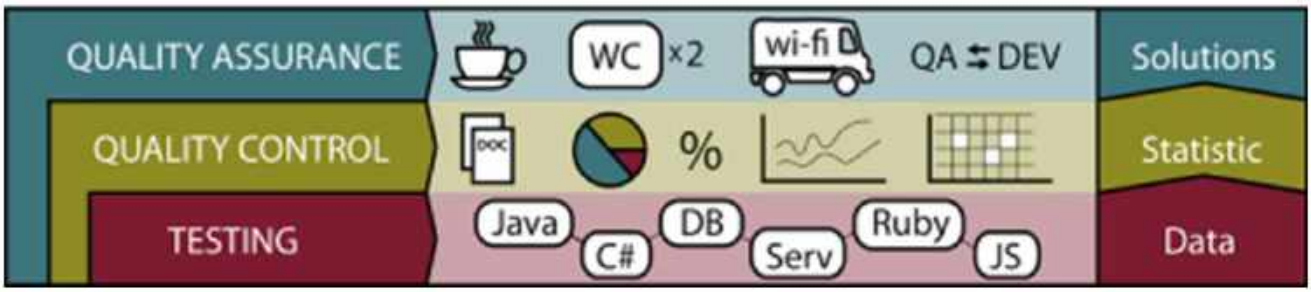

Рис. 1. Якість ПЗ

Забезпечення якості (Quality Assurance – QA) – це сукупність заходів, що охоплюють усі технологічні етапи розроблення, випуску й експлуатації програмного забезпечення (ПЗ) інформаційних систем, що вживаються на різних стадіях життєвого циклу, для забезпечення якості продукту, що випускається.

Контроль якості (Quality Control – QC) – це сукупність дій, які проводяться над об'єктом тестування в процесі розроблення для отримання інформації про актуальний стан об'єкта тестування в таких аспектах: готовність продукту до випуску, відповідність зафіксованим вимогам, відповідність заявленому рівню якості продукту.

Тестування програмного забезпечення (ПЗ) – це процес дослідження програмного забезпечення з метою виявлення недоліків і перевірки його якості. Також тестування ПЗ – це перевірка відповідності між реальною та очікуваною поведінкою програми, що здійснюється на кінцевому наборі тестів, вибраному певним чином. Тестування передбачає активності з планування робіт (Test Management), проєктування тестів (Test Design), виконання тестування (Test Execution) і аналіз отриманих результатів (Test Analysis).

Верифікація (Verification) – це процес оцінювання системи або її компонентів із метою визначення, чи задовольняють результати поточного етапу розроблення умов, сформульованих на початку цього етапу.

Валідація (Validation) – це визначення відповідності розроблюваного ПЗ очікуванням і потребам користувача (*BS 7925-1*).

План тестування (Test Plan) – це документ, що описує весь обсяг робіт із тестування від опису об'єкта, стратегії, розкладу, критеріїв початку

та закінчення тестування до необхідного в процесі роботи обладнання, спеціальних знань, а також оцінювання ризиків із варіантами їх усунення.

Тест-дизайн (Test Design) – це етап процесу тестування ПЗ, на якому проєктуються та створюються тестові випадки (тест-кейси), відповідно до визначених раніше критеріїв якості та цілей тестування.

Тестовий випадок (Test Case) – це документ, що описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації функції або її частини, що тестується.

Баг- / дефект-репорт (Bug Report) – це документ, що описує ситуацію або послідовність дій, що призвела до некоректної роботи об'єкта тестування, із зазначенням причин та очікуваного результату.

Тестове покриття (Test Coverage) – це одна з метрик оцінювання якості тестування, що являє собою відсоток покриття тестами вимог або виконуваного коду.

2. Принципи тестування ПЗ

Для кращого розуміння тестування слід розглянути наявні принципи тестування [1]. Нижче наведено їх список із короткими поясненнями.

Принцип 1 – тестування показує наявність дефектів. Тестування може показати, що дефекти наявні, але не може довести, що дефектів немає. Інакше кажучи, скільки б успішних тестів не було проведено, не можна стверджувати, що немає таких тестів, які б не знайшли помилку. Але якщо знайдено хоча б один дефект, можна стверджувати, що в цьому ПЗ наявні дефекти.

Зазвичай це не означає, що тестування проводилося марно і не може підвищити рівень якості ПЗ. Тестування зменшує ймовірність невиявлених дефектів, які залишаються в ПЗ, але завжди потрібно пам'ятати, що навіть якщо дефекти не знайдено, це ще не є доказом того, що їх немає.

Принцип 2 – вичерпне тестування неможливе. Цей принцип пов'язаний із запитанням «скільки потрібно тестувати?». Можна протестувати все, нічого не тестувати чи протестувати якусь частину. Ідеальний варіант – протестувати все, але слід розібратися, чи необхідно це і чи можливо.

Розглянемо приклад – скільки тестів потрібно виконати, щоб повністю протестувати поле введення, яке приймає одну цифру – номер дня тижня. Залежить від того, що ми розуміємо під словом «повністю». Є сім валідних значень, але, крім того, треба ще переконатися, що невалідні значення (0, 8, 9) правильно обробляються, а ще 26 великих і 26 малих латинських літер, не менше шести знаків пунктуації та пробіл. Виходить 62 варіанти, а ще залишається кирилиця, спецсимволи.

На практиці все ще складніше, оскільки системи зазвичай мають більше одного поля введення, а ці поля мають різний розмір. А ще є

різноманітні середовища, на яких потрібно виконати тести. Візьмемо для прикладу форму, на якій 15 полів введення, кожне з яких набуває п'яти значень, щоб протестувати всі валідні комбінації введення, нам знадобиться 30 517 578 125 (5^{15}) тестів. Дуже малоймовірно, що це вкладеться в межі звичайного проєкту.

Для розв'язання цієї проблеми необхідно використовувати техніки тест-дизайну й автоматизацію, а також пам'ятати про те, що зазвичай неможливо провести повне тестування. Під час визначення достатнього обсягу тестування необхідно враховувати рівень ризику, зокрема технічні ризики та ризики, пов'язані з бізнесом, та такі обмеження проєкту, як час і бюджет. Оцінювання та керування ризиками – одна з найважливіших активностей у будь-якому проєкті. Це дає змогу варіювати трудовитрати на тестування, ґрунтуючись на рівні ризику в різних галузях. Крім того, тестування має забезпечувати необхідну інформацію, щоб зацікавлені особи могли ухвалювати поінформовані рішення про дозвіл продукту або передання замовникам.

Принцип 3 – раннє тестування. Тестові активності мають починатися якомога раніше в циклі розроблення та бути сфокусованими на певних цілях.

Цей принцип пов'язаний із поняттям «ціна дефекту» (cost of defect). Ціна дефекту суттєво зростає протягом життєвого циклу розроблення ПЗ. Чим раніше виявлено дефект, тим швидше, простіше та дешевше його виправити.

Дефект, знайдений у вимогах, коштує найдешевше. Якщо дефект виявлено на етапі розроблення архітектурного рішення, виправити його теж нескладно. Якщо ж дефект, внесений ще на рівні вимог, «дожив» до етапу системного або приймального тестування, його виправлення буде дуже дорогим – адже доведеться внести зміни не тільки в код, але, можливо, і в архітектуру, і у вимоги. Крім того, один дефект у вимогах може проявитися в множинні дефекти на рівні архітектури та коду, а після внесення виправлень потрібно знову проводити тестування.

Можлива ситуація, що ПЗ постачається та формально відповідає узгодженим вимогам, але не відповідає потребам користувачів. Це також спричиняє цілу низку проблем – небажання користувачів переходити на нову систему, складнощі з продажем і впровадженням тощо. Це означає, що вимоги спочатку були неповними, але цю ваду не було виявлено. Ось чому важливо починати тестування зі статичних технік якомога раніше.

Ще одна важлива перевага раннього тестування – економія часу. Тестові активності можуть починатися ще до того, як написано перший рядок коду. У міру того як готуються вимоги та специфікації, тестувальники можуть розпочинати розроблення та перегляд тест-кейсів. І коли з'явиться перша версія, можна буде відразу розпочинати виконання тестів.

Принцип 4 – скупчення дефектів. Невелика кількість модулів містить більшість дефектів, виявлених на етапі передрелізного

тестування, або демонструє найбільшу кількість відмов на етапі експлуатації.

Багато тестувальників спостерігали такий ефект – дефекти «скупчуються». Це може статися тому, що певна область коду особливо складна і заплутана, або тому, що внесення змін робить «ефект доміно». Це знання часто використовується для оцінювання ризиків під час планування тестів – тестувальники фокусуються на відомих «проблемних зонах».

Про те, де «скупчуються» дефекти, можна дізнатися ще на ранніх етапах, коли проводиться статичне тестування (наприклад, code review та аналіз коду за допомогою спеціальних інструментів). Коли дійде до динамічного тестування, можна сфокусуватися на тих областях, де було виявлено більше дефектів статичними методами.

Також корисно проводити аналіз першопричин (root cause analysis), щоб запобігти повторній появі дефектів, виявити причини виникнення скупчень дефектів і спрогнозувати потенційні скупчення дефектів у майбутньому.

Принцип 5 – парадокс пестициду. Якщо повторювати ті самі тести знову і знову, у якийсь момент цей набір тестів перестане виявляти нові дефекти.

Ті скупчення дефектів, про які йдеться в 4-му принципі, мають тенденцію переміщатися. З'ясуємо, чому так відбувається.

Цю аналогію запровадив Борис Бейзер 1983 р. Він навів приклад оброблення полів пестицидами. Поле обробляється певним пестицидом уперше, і значна частина шкідників гине, але деякі все ж таки виживають, тому що їхні організми виявилися стійкими до дії отрути. Якщо повторно обробити поле тим самим пестицидом, то ті, що вижили після першого оброблення, з великою ймовірністю виживуть і після другої.

Повторне застосування тих самих тестів і тих методик призводить до того, що в продукті залишаються саме ті дефекти, проти яких ці тести і ці методики неефективні.

Щоб подолати «парадокс пестицидів», необхідно регулярно переглядати наявні тест-кейси та створювати нові, різноманітні тести, які виконуватимуться на різних частинах системи. Це дасть змогу виявити більше дефектів.

Принцип 6 – тестування залежить від контексту. Тестування виконується по-різному залежно від контексту. Наприклад, тестування систем, критичних із погляду безпеки, проводиться не так, як тестування сайту інтернет-магазину.

Цей принцип тісно пов'язаний із поняттям ризику. Ризик – це потенційна проблема. У ризику є ймовірність (likelihood), яка завжди вища за 0 і нижча за 100%, і є вплив (impact) – ті негативні наслідки, яких ми побоюємося. Аналізуючи ризики, ми завжди зважуємо ці два аспекти: імовірність і вплив.

Оскільки різні системи пов'язані з різними рівнями ризику, вплив того чи іншого дефекту також сильно варіюється (одні проблеми досить тривіальні, інші можуть дорого обійтися і призвести до великих втрат грошей, часу, ділової репутації, а в разі систем керування транспортом – навіть до аварій). Відповідно, рівень ризику впливає на вибір методологій, технік і типів тестування.

Принцип 7 – помилка про відсутність помилок. Виявлення та виправлення дефектів є марним, якщо побудована система незручна для використання та не відповідає потребам та очікуванням користувачів. Замовники ПЗ (компанії та люди, які купують і використовують його, щоб виконувати свої повсякденні завдання) насправді зовсім не цікавляться дефектами та їх кількістю, крім тих випадків, коли вони безпосередньо стикаються з нестабільністю продукту. Їм також нецікаво, наскільки ПЗ відповідає формальним вимогам, задокументованим. Користувачі ПЗ більш зацікавлені в тому, щоб воно допомагало їм ефективно виконувати завдання. ПЗ має відповідати їхнім потребам, і саме із цього погляду вони його оцінюють.

Навіть якщо виконані всі тести та помилки не виявлено, то це ще не гарантія того, що ПЗ відповідатиме потребам та очікуванням користувачів. Інакше кажучи, верифікація не є валідацією.

Розглянуті раніше терміни можно сформулювати простіше: верифікація – перевірка на відповідність вимогам; валідація – перевірка щодо відповідності потребам і очікуванням, відповідність заданим цілям. Частину тестових активностей має бути спрямовано на верифікацію, частину – на валідацію.

Теоретично, якщо вимоги було зібрано та проаналізовано правильно і якщо на етапі розроблення архітектури та коду не зв'явилися спотворення, не має бути протиріч.

3. Цілі та завдання тестування ПЗ

Цілями тестування програмного забезпечення є [2]:

- підвищення ймовірності того, що додаток, призначений для тестування, працюватиме правильно за будь-яких обставин;
- підвищення ймовірності того, що додаток, призначений для тестування, відповідатиме всім описаним вимогам;
- проведення повного тестування програми за встановлений термін.

Тестування програмного забезпечення також передбачає завдання:

- перевірити, чи система працює відповідно до певних таймінгів клієнта та сервера;
- перевірити, що найбільш критичні послідовності дій із системою кінцевого користувача виконуються правильно;

- перевірити роботу інтерфейсів користувача;
- перевірити, що зміни в базах даних не впливають на наявні програмні модулі;
- мінімізувати під час проєктування тестів їх перероблення за можливих змін програми;
- використовувати інструменти автоматизованого тестування там, де це доцільно;
- проводити тестування таким чином, щоб не лише виявляти, а й запобігати дефектам;
- використовувати під час проєктування автоматизованих тестів стандарти розроблення таким чином, щоб створити скрипти, що багаторазово використовуються і супроводжуються.

3.1. Цілі та завдання тестування з RUP

Rational Unified Process (RUP) – методологія розроблення програмного забезпечення, створена компанією Rational Software.

В основі RUP лежать такі **принципи**:

- рання ідентифікація та безперервне (до закінчення проєкту) усунення основних ризиків;
- концентрація на виконанні вимог замовників до виконуваної програми (аналіз і побудова моделі прецедентів – варіантів використання);
- очікування змін у вимогах, проєктних рішеннях і реалізації в процесі розроблення;
- компонентна архітектура, що реалізується та тестується на ранніх стадіях проєкту;
- постійне забезпечення якості на всіх етапах розроблення проєкту (продукту);
- робота над проєктом у згуртованій команді, ключова роль якої належить архітекторам.

Процеси та стадії RUP. RUP використовує ітеративну модель розроблення (рис. 2). Наприкінці кожної ітерації (яка в ідеалі триває від 2 до 6 тижнів) проєктна команда повинна досягти запланованих на цю ітерацію цілей, створити або доопрацювати проєктні артефакти й отримати проміжну, але функціональну версію кінцевого продукту. Ітеративне розроблення дає змогу швидко реагувати на мінливі вимоги, виявляти й усувати ризики на ранніх стадіях проєкту, а також ефективно контролювати якість продукту.

Повний життєвий цикл розроблення продукту складається із чотирьох фаз, кожна з яких охоплює одну або кілька ітерацій. Нижче подано графічний опис процесів за RUP.

Початок (Inception). У фазі «Початок» (Inception) відбуваються такі процеси:

1. Формування бачення та меж проєкту.

2. Створення економічного обґрунтування (business case).

3. Визначення основних вимог, обмежень і ключової функціональності продукту.

4. Створення базової версії моделі прецедентів.

5. Оцінювання ризиків.

На етапі завершення фази «Початок» оцінюється досягнення етапу цілей життєвого циклу (Lifecycle Objective Milestone), яке передбачає угоду заінтересованих сторін про продовження проєкту.

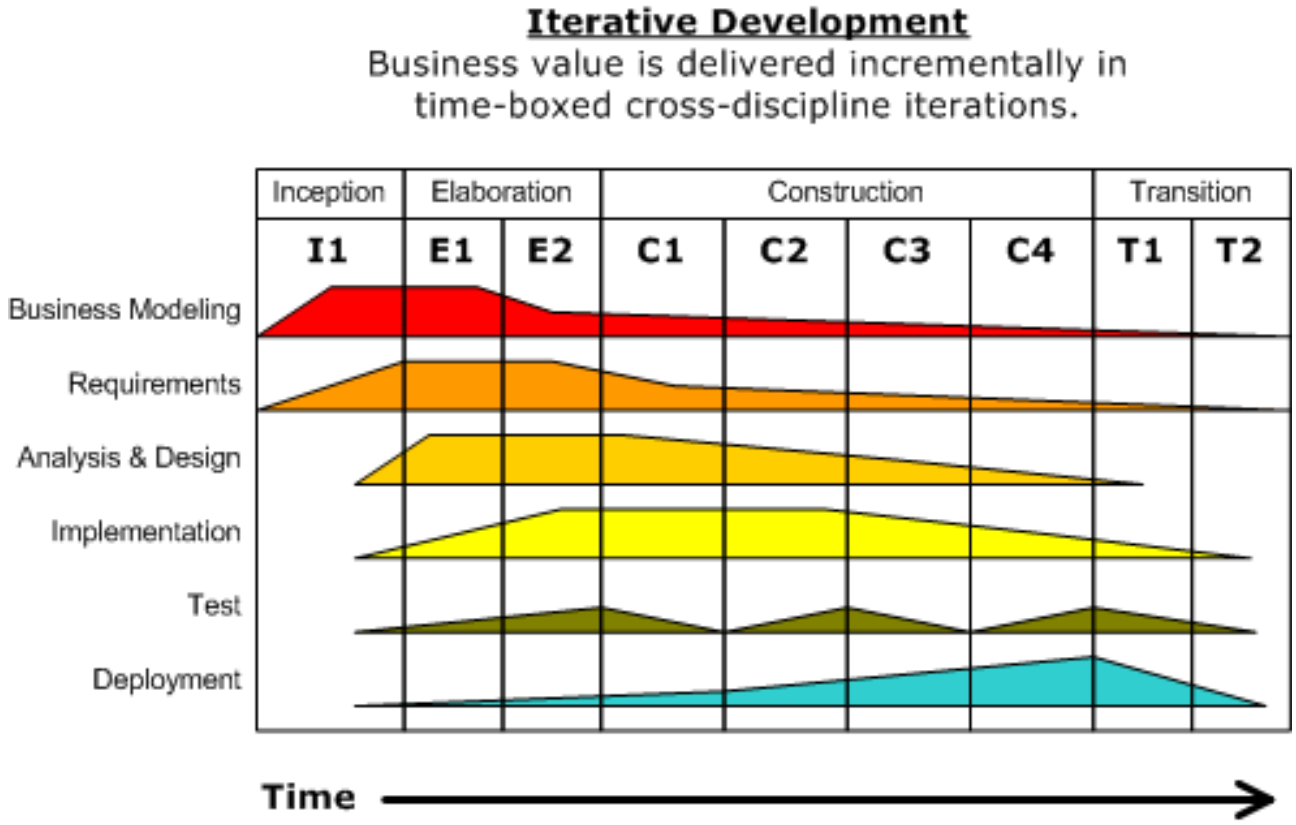

Рис. 2. Цикл розроблення ПЗ за RUP

Уточнення (Elaboration). У фазі «Уточнення» проводиться аналіз предметної області та побудова архітектури, що виконується. Ця фаза передбачає:

1. Документування вимог (зокрема детальний опис більшості прецедентів).

2. Спроєктовану, реалізовану та відтестовану архітектуру, що виконується.

3. Оновлене економічне обґрунтування та більш точне оцінювання термінів і вартості.

4. Знижені основні ризики.

Успішне виконання фази розроблення означає досягнення етапу архітектури життєвого циклу (Lifecycle Architecture Milestone).

Побудова (Construction). У фазі «Побудова» відбувається реалізація більшої частини функціональності продукту. Фаза «Побудова»

завершується першим зовнішнім релізом системи і етапом початкової функціональної готовності (Initial Operational Capability).

Упровадження (Transition). У фазі «Упровадження» створюється фінальна версія продукту передається від розробника до замовника. Ця фаза охоплює бета-тестування, навчання користувачів, а також визначення якості продукту. Якщо якість не відповідає очікуванням користувачів або критеріям, встановленим у фазі «Початок», фаза «Упровадження» повторюється знову. Виконання всіх цілей означає досягнення етапу готового продукту (Product Release) та завершення повного циклу розроблення.

4. Стандарти SWEBOK та ISTQB

SWEBOK (Software Engineering Body of Knowledge) – документ, який готує комітет Software Engineering Coordinating Committee, до якого залучено співтовариство IEEE Computer Society. Призначення SWEBOK полягає в поєднанні знань з інженерії програмного забезпечення (розроблення програмного забезпечення).

Документ є одним із трьох документів, створених спільними зусиллями IEEE-CS та ACM, покликаних забезпечити визначити:

● необхідний набір знань і рекомендовані практики;

● етичні та професійні стандарти;

● навчальну програму для студентів, аспірантів, які продовжують навчання.

Цей документ являє собою перший компонент – необхідний набір знань і рекомендовані практики.

Другий документ випущено 1998 року та присвячено етичним і професійним стандартам для інженерії ПЗ.

Третій документ (SE2004) випущено 2004 року та присвячено складанню навчального плану з програмної інженерії.

Наприкінці 2013 року побачила світ нова версія SWEBOK – SWEBOK V3.

Наразі випущено стандарт ISO/IEC TR 19759, що чинний від 2015 року.

Поточна опублікована версія SWEBOK V3 охоплює п'ятнадцять областей знань у сфері програмної інженерії:

- software requirements – вимоги до ПЗ;
- software design – проєктування ПЗ;
- software construction – конструювання ПЗ;
- software testing – тестування ПЗ;
- software maintenance – супровід ПЗ;
- software configuration management – керування конфігурацією;
- software engineering management – керування IT-проєктом;
- software engineering process – процес програмної інженерії;

- software engineering models and methods – моделі та методи розроблення;
- software quality – якість ПЗ;
- software engineering professional practice – опис критеріїв професіоналізму та компетентності;
- software engineering economics – економічні аспекти розроблення ПЗ;
- computing foundations – основи обчислювальних технологій, які застосовуються під час розроблення ПЗ;
- mathematical foundations – базові математичні концепції та поняття, які застосовуються під час створення ПЗ;
- engineering foundations – основи інженерної діяльності.

ISTQB® (International Software Testing Qualifications Board) – організація сертифікації тестувальників ПЗ. Заснована в Единбурзі в листопаді 2002 року. ISTQB® є некомерційною асоціацією, офіційно зареєстрованою в Бельгії. ISTQB® відповідає за міжнародну схему кваліфікації, яка називається «ISTQB® Certified Tester». Кваліфікація ґрунтується на програмі навчання (Syllabus) і складається з кількох рівнів. ISTQB® є найбільшою організацією в галузі кваліфікації тестувальників програмного забезпечення, представленою більш ніж у 81 країні світу (дані на квітень 2021), що видала понад 720 тис. сертифікатів.

ISTQB® Certified Tester. Базовий рівень. Зміст програми базового рівня містить:

- основи тестування програмного забезпечення;
- життєвий цикл тестування;
- динамічне тестування;
- статичне тестування;
- керування тестуванням;
- засоби тестування.

Кваліфікація базового рівня призначена для всіх, хто будь-яким чином залучений до тестування програмного забезпечення. Наприклад, для тестувальників, тест-аналітиків, інженерів із тестування, консультантів у галузі тестування, керівників тестування, тестувальників приймання та розробників ПЗ. Базовий рівень корисний також тим, кому необхідно розуміти базові принципи тестування, наприклад менеджерам, менеджерам із якості, керівникам груп розробників, бізнес-аналітикам, IT-директорам і консультантам із менеджменту.

Власники сертифікатів базового рівня можуть претендувати на більш високий рівень кваліфікації в тестуванні.

Контрольні запитання

1. Що таке тестування?
2. Чим відрізняється валідація від верифікації?

3. Що передбачає план тестування?
4. Які цілі тестування?
5. У чому полягає завдання тестування?
6. Які ви знаєте стандарти у сфері тестування?
7. Що таке якість програмного забезпечення?
8. Чи можна знайти 100 % помилок?

Лекції № 3, 4. ОСНОВНІ МОДЕЛІ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1. Життєвий цикл програмного забезпечення

Життєвий цикл програмного забезпечення (ЖЦ ПЗ) – період часу, який починається з моменту ухвалення рішення про необхідність створення програмного продукту та закінчується в момент його повного вилучення з експлуатації. Цей цикл – процес побудови та розвитку ПЗ (рис. 3) [3].

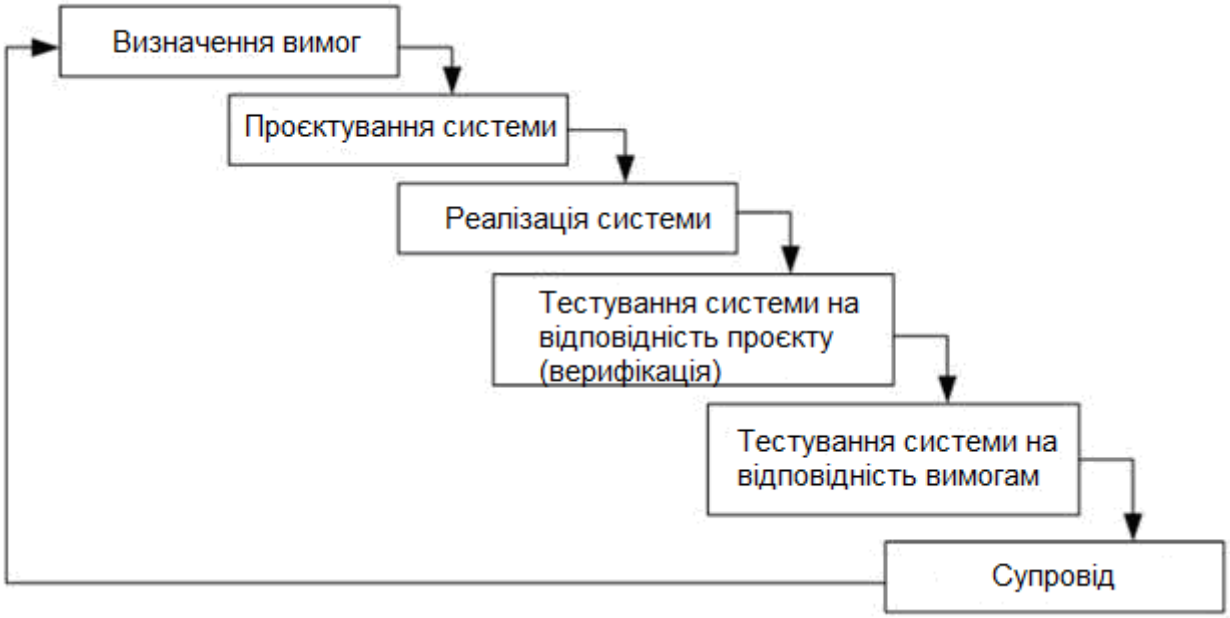

Рис. 3. Життєвий цикл програмного забезпечення

2. Стандарт ДСТУ ISO/IEC/IEEE 12207:2018

Національний стандарт ДСТУ ISO/IEC/IEEE 12207:2018 «Інженерія систем і програмних засобів. Процеси життєвого циклу програмних засобів» відповідає міжнародному стандарту ISO/IEC/IEEE 12207:2017 «System and software engineering – Software life cycle processes». Цей стандарт, використовуючи усталену термінологію, установлює загальну структуру процесів життєвого циклу програмних засобів, на яку можна орієнтуватися в програмній промисловості. Стандарт визначає процеси, види діяльності та завдання, що використовуються під час придбання програмного продукту або послуги, а також під час постачання, розроблення, застосування за призначенням, супроводом і припиненням застосування програмних продуктів.

Процеси життєвого циклу ПЗ. Стандарт групує різні види діяльності, які можуть виконуватися протягом життєвого циклу програмних систем у сім груп процесів. Кожен із процесів життєвого циклу в межах цих груп описується в термінах мети та бажаних результатах, списків дій і завдань, які необхідно виконувати для досягнення цих результатів.

Кількість процесів ЖЦ ПЗ поділяються таким чином:

- процеси угоди – два процеси;
- процеси організаційного забезпечення проєкту – п'ять процесів;
- процеси проєкту – сім процесів;
- технічні процеси – одинадцять процесів;
- процеси реалізації програмних засобів – сім процесів;
- процеси підтримки програмних засобів – вісім процесів;
- процеси повторного застосування програмних засобів – три процеси.

Кожен процес передбачає низку дій. Наприклад, процес придбання охоплює такі дії:

- ініціювання придбання;
- підготовка заявкових пропозицій;
- підготовка та коригування договору;
- нагляд за діяльністю постачальника;
- приймання та завершення робіт.

Кожна дія охоплює низку завдань. Наприклад, підготовка заявкових пропозицій має передбачати:

1. Формування вимог до системи.
2. Формування списку програмних продуктів.
3. Установлення умов та угод.
4. Опис технічних обмежень (середовище функціонування системи тощо).

Модель життєвого циклу ПЗ – структура, що визначає послідовність виконання та взаємозв'язку процесів, дій і завдань протягом життєвого циклу. Модель життєвого циклу залежить від специфіки, масштабу та складності проєкту та специфіки умов, у яких система створюється та функціонує.

Стандарт не пропонує конкретної моделі життєвого циклу. Його положення є спільними для будь-яких моделей життєвого циклу, методів і технологій створення інформаційних систем. Документ описує структуру процесів життєвого циклу не конкретизуючи, як реалізувати чи виконати дії та завдання, які входять до цих процесів.

Модель ЖЦ ПЗ включає:

1. Стадії.
2. Результати виконання робіт на кожній стадії.
3. Ключові події – етапи завершення робіт і ухвалення рішень.

Стадія – частина процесу створення ПЗ, обмежена певними часовими межами, що закінчується випуском конкретного продукту (моделей, програмних компонентів, документації) і визначається заданими для цієї стадії вимогами.

На кожній стадії можуть виконуватися кілька процесів, визначених у стандарті, і навпаки, той самий процес може виконуватися на різних стадіях. Співвідношення між процесами та стадіями також визначається використовуваною моделлю життєвого циклу.

3. Моделі життєвого циклу ПЗ

3.1. Водоспадна (каскадна, послідовна) модель

Водоспадну модель життєвого циклу (waterfall model) було запропоновано в 1970 р. Вінстоном Ройсом. Ця модель передбачає послідовне виконання всіх етапів проєкту в строго фіксованому порядку. Перехід на наступний етап означає повне завершення робіт на попередньому етапі. Вимоги, визначені на стадії їх формування, суворо документуються як технічне завдання і фіксуються протягом розроблення проєкту. Кожна стадія завершується випуском повного комплекту документації, достатнього для того, щоб технологію могло бути продовжено іншою командою розробників.

Етапи проєкту відповідно до каскадної моделі:

1. Формування вимог.
2. Проєктування.
3. Реалізація.
4. Тестування.
5. Упровадження.
6. Експлуатація та супровід.

Перевагами каскадної моделі є повна й узгоджена документація на кожному етапі, а також просте визначення термінів і витрат на проєкт.

Недоліками цієї моделі є те, що у водоспадній моделі перехід від однієї фази проєкту до іншої передбачає повну коректність результату (виходу) попередньої фази. Однак неточність будь-якої вимоги або некоректна її інтерпретація в результаті призводить до того, що доводиться «відкочуватися» до ранньої фази проєкту, зокрема, потрібне перероблення не просто «вибиває» проєктну команду з графіка, а й часто веде до якісного зростання витрат, і не виключено, що до припинення проєкту в тій формі, у якій він спочатку замислювався. На думку сучасних фахівців, основна помилка авторів водоспадної моделі полягає в припущеннях, що проєкт проходить через увесь процес один раз, спроєктована архітектура зручна і проста у використанні, проєкт здійснення розумний, а помилки в реалізації легко усуваються в міру

тестування. Ця модель виходить із того, що всі помилки будуть зосереджені в реалізації, тому їх усунення відбувається поетапно під час тестування компонентів і системи. Таким чином, водоспадна модель для великих проєктів є мало реалістичною і може бути ефективно використана тільки для створення невеликих систем.

3.2. Ітераційна модель

Альтернативою послідовної моделі є так звана модель ітеративного та інкрементального розроблення (iterative and incremental development, IID), що дістала також від Т. Гілба в 70-х роках назву еволюційної моделі. Також цю модель називають ітераційною моделлю та інкрементальною моделлю.

Модель IID (рис. 4) передбачає поділ життєвого циклу проєкту на послідовність ітерацій, кожна з яких нагадує мініпроєкт, охоплюючі всі процеси розроблення щодо створення менших фрагментів функціональності порівняно з проєктом загалом. Мета кожної ітерації – отримання версії програмної системи, що працює і включає функціональність, визначену інтегрованим змістом усіх попередніх і поточної ітерації. Результат фінальної ітерації містить усю необхідну функціональність продукту. Таким чином, із завершенням кожної ітерації продукт отримує приріст – інкремент – до його можливостей, які розвиваються еволюційно. Ітеративність, інкрементальність і еволюційність у цьому випадку є вираженням одного й того самого сенсу, але різними словами з різних поглядів.

За словами Т. Гілба, еволюція – прийом, призначений для створення видимості стабільності. Шанси успішного створення складної системи будуть максимальними, якщо вона реалізується в серії невеликих кроків і якщо кожен крок містить чітко певний успіх, а також можливість «відкату» до попереднього успішного етапу в разі невдачі. Перед тим як задіяти всі ресурси, призначені для створення системи, розробник має можливість отримувати з реального світу сигнали зворотного зв'язку та виправляти можливі помилки в проєкті [3].

Підхід IID має і свої недоліки, які, по суті, – зворотний бік переваг. По-перше, цілісне розуміння можливостей та обмежень проєкту дуже довгий час відсутнє. По-друге, за ітерацій доводиться відкидати частину зробленої раніше роботи. По-третє, сумлінність фахівців під час виконання робіт все ж таки знижується, що психологічно зрозуміло, адже команда постійно має відчуття, що все одно все можна буде переробити і покращити пізніше.

Різні варіанти ітераційного підходу реалізовані в більшості сучасних методологій розроблення (RUP, MSF, XP).

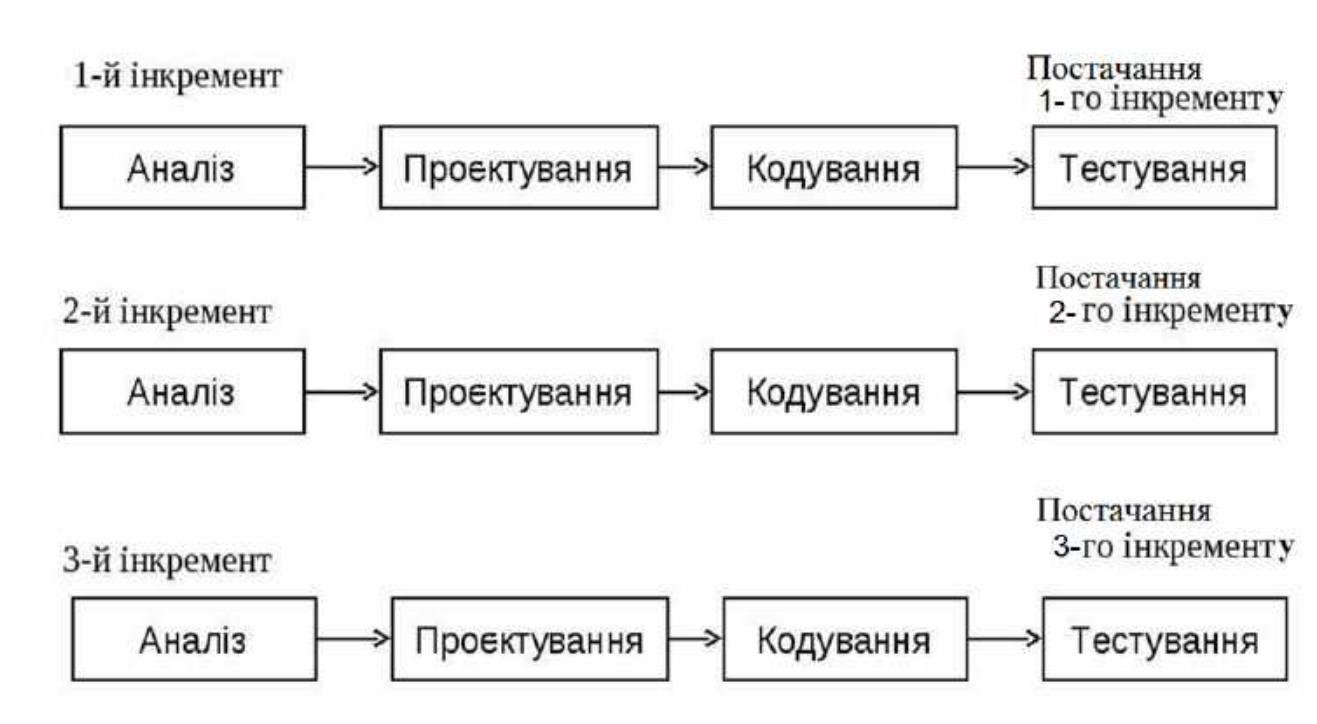

Рис. 4. Модель розроблення ПЗ IID

3.3. Спіральна модель

Спіральну модель (англ. spiral model) було розроблено в середині 1980-х років Баррі Боемом (рис. 5). Цю модель основано на класичному циклі Демінга PDCA (plan-do-check-act). Під час використання цієї моделі програмне забезпечення створюється в кілька ітерацій (витків) методом прототипування.

Рис. 5. Спіральна модель розроблення ПЗ

Кожна ітерація відповідає створенню фрагмента або версії ПЗ, на ній уточнюються цілі та характеристики проєкту, оцінюється якість отриманих результатів і плануються роботи наступної ітерації.

На кожній ітерації оцінюються:

- ризик перевищення термінів і вартості проєкту;
- необхідність виконання ще однієї ітерації;
- ступінь повноти і точності розуміння вимог до системи;
- доцільність припинення проєкту.

Важливо розуміти, що спіральна модель не є альтернативою еволюційної моделі (моделі IID), а спеціально опрацьованим варіантом. На жаль, нерідко спіральну модель або хибно використовують як синонім еволюційної моделі загалом, або (не менш хибно) згадують як цілком самостійну модель поряд з IID.

Відмітною особливістю спіральної моделі є спеціальна увага, що приділяється ризикам, які впливають на організацію життєвого циклу, та контрольним точкам.

Баррі Боем формулює десять найбільш поширених (за пріоритетами) ризиків:

1. Дефіцит спеціалістів.
2. Нереалістичні терміни та бюджет.
3. Реалізація невідповідної функціональності.
4. Розроблення неправильного інтерфейсу користувача.
5. Перфекціонізм, непотрібна оптимізація та відточування деталей.
6. Безперервний потік змін.
7. Нестача інформації про зовнішні компоненти, що визначають середовище системи або залучені в інтеграцію.
8. Недоліки в роботах, що виконуються зовнішніми (стосовно проєкту) ресурсами.
9. Недостатня продуктивність одержуваної системи.
10. Різна кваліфікація спеціалістів різних галузей.

На сьогодні в спіральній моделі визначено такий загальний набір контрольних точок:

1. Concept of Operations (COO) – концепція (використання) системи.
2. Life Cycle Objectives (LCO) – цілі та зміст життєвого циклу.
3. Life Cycle Architecture (LCA) – архітектура життєвого циклу; готовність концептуальної архітектури цільової програмної системи.
4. Initial Operational Capability (IOC) — перша версія продукту, що створюється, придатна для досвідченої експлуатації.
5. Final Operational Capability (FOC) – готовий продукт, розгорнутий (встановлений і настроєний) для реальної експлуатації.

3.4. V-Model

V-Model (або VEE-модель) є моделлю розроблення інформаційних систем (ІС), спрямованою на спрощення розуміння складнощів, пов'язаних із розроблення систем (рис. 6). Ця модель використовується для визначення єдиної процедури розроблення програмних продуктів, апаратного забезпечення і людино-машинних інтерфейсів.

Концепцію V-подібної моделі було розроблено в Німеччині та США наприкінці 1980-х років незалежно одна від одної.

Німецьку V-модель було розроблено аерокосмічною компанією IABG в Оттобрунні біля Мюнхена за сприяння Федерального департаменту із закупівлі озброєнь у Кобленці для Міністерства оборони Німеччини. Модель було прийнято німецькою федеральною адміністрацією для цивільних потреб улітку 1992 року.

Американську V-Model (VEE) було розроблено національною радою з системної інженерії (міжнародною – з 1995 року) для супутникових систем, зокрема для обладнання, програмного забезпечення та взаємодії з користувачами.

Сучасною версією V-Model є V-Model XT, яку було затверджено в лютому 2005 року. V-модель використовується для керування процесом розроблення програмного забезпечення для німецької федеральної адміністрації. Наразі вона є стандартом для німецьких урядових та оборонних проєктів, а також для виробників ПЗ в Німеччині. V-Model є ймовірніше набором стандартів у галузі проєктів, що стосуються розроблення нових продуктів. Ця модель багато в чому схожа на PRINCE2 і визначає методи як проєктного керування, так системного розвитку.

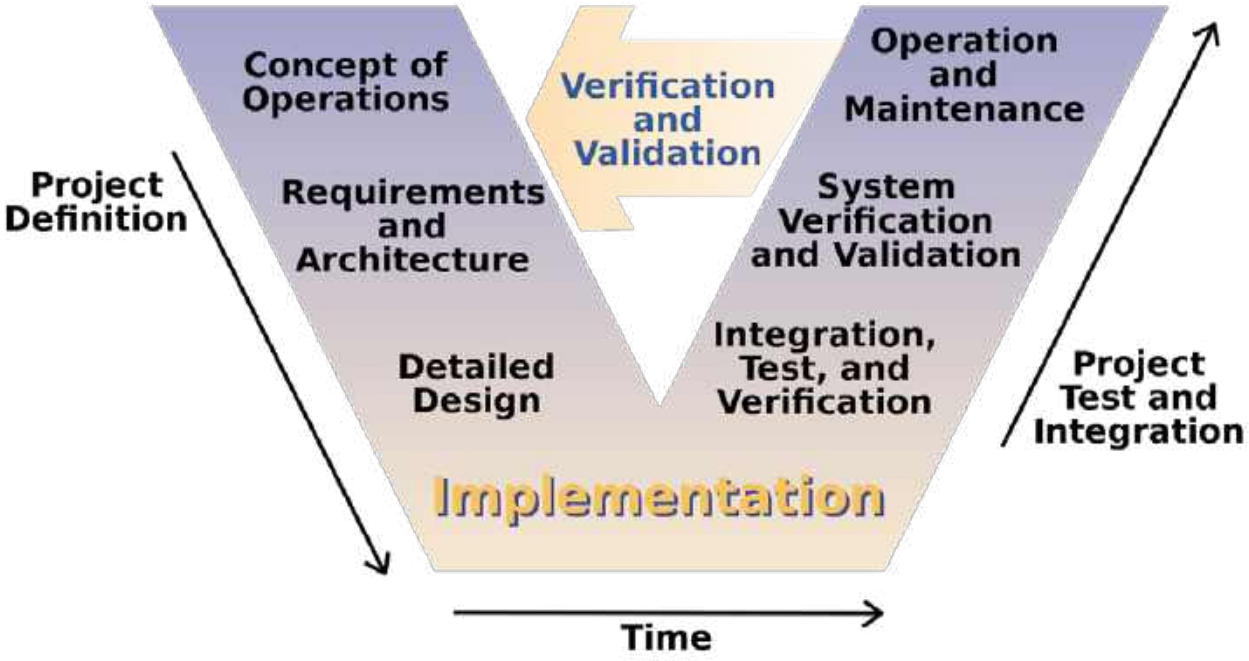

Рис. 6. V-model розроблення ПЗ

Основний принцип V-подібної моделі полягає в тому, що деталізація проєкту зростає за умови руху зліва направо одночасно з плином часу, і ні те, ні інше не може повернути назад. Ітерації в проєкті проводяться по горизонталі – між лівою та правою сторонами літери.

Щодо розроблення інформаційних систем V-Model – варіація каскадної моделі, у якій завдання розроблення йдуть зверху вниз по лівій стороні літери V, а завдання тестування – вгору праворуч літери V. Усередині V проводяться горизонтальні лінії, що показують, як результати кожної з фаз розроблення впливають на розвиток системи тестування на кожній із фаз тестування. Модель базується на тому, що приймально-здавальні випробування ґрунтуються насамперед на вимогах, системне тестування – на вимогах та архітектурі, комплексне тестування – на вимогах, архітектурі й інтерфейсах, а компонентне тестування – на вимогах, архітектурі, інтерфейсах й алгоритмах.

Нижче наведено аналогію з життя, принцип роботи V-Model під час надсилання листа від однієї компанії до іншої та відповідь на отриманий лист (рис. 7).

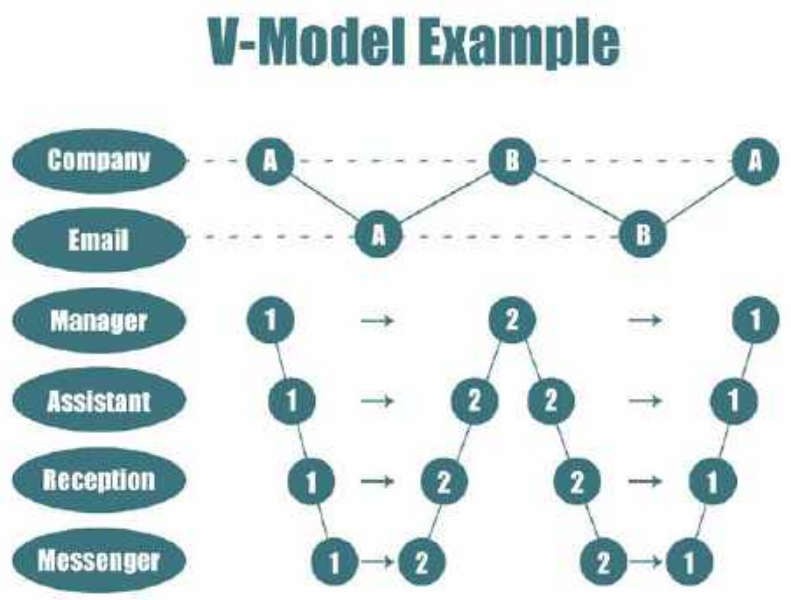

Рис. 7. Приклад V-моделі

V-модель забезпечує підтримку в плануванні та реалізації проєкту. Під час проєкту ставляться такі завдання:

1. **Мінімізація ризиків.** V-подібна модель робить проєкт прозорішим і підвищує якість контролю проєкту шляхом стандартизації проміжних цілей та опису відповідних їм результатів і відповідальних осіб. Це дає змогу виявляти невідповідності в проєкті та ризики на ранніх стадіях і покращує якість керування проєктів, зменшуючи ризики.

2. **Підвищення та гарантії якості.** V-Model – стандартизована модель розроблення, що дає змогу досягти від проєкту результатів

бажаної якості. Проміжні результати можна перевірити на ранніх стадіях. Універсальне документування полегшує прочитання, зрозумілість і перевірюваність.

3. **Зменшення загальної вартості проєкту.** Ресурси на розроблення, виробництво, керування та підтримку можуть бути заздалегідь прораховані та проконтрольовані. Отримані результати також універсальні та легко прогнозуються. Це зменшує витрати на наступні стадії та проєкти.

4. **Підвищення якості комунікації між учасниками проєкту.** Універсальний опис усіх елементів та умов полегшує порозуміння всіх учасників проєкту. Таким чином, зменшуються непорозуміння між користувачем, покупцем, постачальником і розробником.

Перевагами V-Model вважають такі характеристики:

1. Користувачі V-Model беруть участь у розробленні та підтримці V-моделі. Комітет із контролю за змінами підтримує проєкт і збирається щорічно для оброблення всіх отриманих запитів на внесення змін до V-Model.

2. На старті будь-якого проєкту V-подібну модель може бути адаптовано до цього проєкту, оскільки ця модель не залежить від типів організацій і проєктів.

3. V-model дає змогу розбити діяльність на окремі кроки, кожен із яких передбачатиме необхідні для нього дії, інструкції до них, рекомендації та докладне пояснення діяльності.

4. У моделі особливе значення надається плануванню, спрямованому на верифікацію та атестацію продукту, що розробляється на ранніх стадіях його розроблення. Фаза модульного тестування підтверджує правильність детального проєктування. Фази інтеграції та тестування реалізують архітектурне проєктування чи проєктування на найвищому рівні. Фаза тестування системи підтверджує правильність виконання етапу вимог до продукту та його специфікації.

5. У моделі передбачені атестація та верифікація всіх зовнішніх і внутрішніх отриманих даних, а не лише самого програмного продукту.

6. У V-подібній моделі визначення вимог виконується перед розробленням проєкту системи, а проєктування – перед розробленням компонентів.

7. Модель визначає продукти, які мають бути отримані після процесу розроблення, причому кожні отримані дані повинні тестуватися.

8. Завдяки моделі менеджери проєкту можуть відстежувати перебіг процесу розроблення, тому що в цьому випадку цілком можливо скористатися тимчасовою шкалою, а завершення кожної фази є контрольною точкою.

У V-Model наявні також обмеження, які можна розглянути окремо або їх адаптувати до моделі. Обмеженнями V-Model є такі:

- не регулюється розміщення контрактів обслуговування;

– не враховуються організація та виконання керування, обслуговування, ремонту й утилізації системи, однак планування та підготовка до цих операцій моделлю розглядаються;

– більше стосується розроблення програмного забезпечення в проєкті, ніж усієї організації процесу.

Недоліками V-Model є:

– не передбачено роботи з паралельними подіями;

– не передбачено внесення вимог динамічних змін на різних етапах життєвого циклу;

– тестування вимог у життєвому циклі відбувається надто пізно, унаслідок чого неможливо внести зміни, не вплинувши при цьому графік виконання проєкту;

– не входять дії, створені задля аналізу ризиків;

– деякі результати можна побачити лише за умови досягнення низу літери V.

4. Методології розроблення ПЗ

4.1. Гнучка методологія розроблення

Гнучка методологія розроблення (Agile software development) – серія підходів до розроблення програмного забезпечення, орієнтованих на використання ітеративного розроблення, динамічне формування вимог і забезпечення їх реалізації внаслідок постійної взаємодії всередині робочих груп, що самоорганізуються і що складаються з фахівців різного профілю. Існує кілька методик, що належать до класу гнучких методологій розроблення, зокрема відомі як гнучкі методики екстремального програмування, DSDM, Scrum [5].

Більшість гнучких методологій націлені на мінімізацію ризиків шляхом обмеження розроблення серією коротких циклів, які називаються ітераціями, які зазвичай тривають два-три тижні. Кожна ітерація сама по собі має вигляд програмного проєкту в мініатюрі й охоплює всі завдання, необхідні для надання мініприросту за функціональністю: планування, аналіз вимог, проєктування, програмування, тестування і документування. Хоча окремої ітерації недостатньо для випуску нової версії продукту, мається на увазі, що гнучкий програмний проєкт готовий до випуску наприкінці кожної ітерації. Після закінчення кожної ітерації команда переоцінює пріоритети розроблення.

Agile-методи наголошують на безпосередньому спілкуванні віч-на-віч. Більшість agile-команд перебувають в одному офісі, іноді в так званому bullpen. Щонайменше в команді є «замовники» (product owner – замовник або його повноважний представник, який визначає вимоги до продукту; цю роль може виконувати менеджер проєкту, бізнес-аналітик чи

клієнт). В офісі можуть бути також тестувальники, дизайнери інтерфейсу, технічні письменники та менеджери.

Основною метрикою agile-методів є робочий продукт. Віддаючи перевагу безпосередньому спілкуванню, agile-методи зменшують обсяг письмової документації, проте іншими методами. Це призвело до критики цих методів як недисциплінованих.

Історія. У лютому 2001 року в штаті Юта США було випущено «Маніфест гнучкої методології розроблення програмного забезпечення». Маніфест був альтернативою класичним практикам розроблення програмного забезпечення, таким як «метод водоспаду», який був золотим стандартом розроблення на той час. Цей маніфест було схвалено та підписано представниками методологій: екстремального програмування, Crystal Clear, DSDM, Feature driven development, Scrum, Adaptive software development, Pragmatic Programming. Гнучка методологія розроблення використовувалася багатьма компаніями і до прийняття маніфесту, проте саме після цієї події відбулося входження Agile-розроблення до Mac.

Принципи. Agile – сімейство процесів розроблення, а не єдиний підхід у розробленні програмного забезпечення, і визначається Agile Manifesto. Agile не передбачає практик, а визначає цінності та принципи, яких дотримуються успішні команди.

Agile Manifesto розроблено й ухвалено 11–13 лютого 2001 року на лижному курорті The Lodge at Snowbird у горах Юти. Маніфест підписали представники таких методологій: Extreme programming, Scrum, DSDM, Adaptive software development, Crystal Clear, Feature driven development, Pragmatic Programming. Agile Manifesto містить чотири основні ідеї та дванадцять принципів. Показово, що Agile Manifesto не містить практичних порад.

Основні ідеї Agile Manifesto:

- особи та їх взаємодії важливіші, ніж процеси й інструменти;
- програмне забезпечення, що працює, є важливішим, ніж повна документація;
- співпраця із замовником важливіша, ніж контрактні зобов'язання;
- реакція на зміни важливіша, ніж дотримання плану.

Принципи, які пояснює Agile Manifesto:

- задоволення клієнта завдяки ранньому та безперебійному постачанню цінного програмного забезпечення;
- схвалення змін вимог навіть наприкінці розроблення (це може підвищити конкурентоспроможність одержаного продукту);
- часте постачання робочого програмного забезпечення (кожен місяць / тиждень / ще частіше);
- тісне щоденне спілкування замовника з розробниками протягом усього проєкту;
- проєктом займаються мотивовані особи, які забезпечені необхідними умовами роботи, підтримкою та довірою;

- рекомендований метод передання – особиста розмова (віч-на-віч);
- програмне забезпечення, що працює, – найкращий вимірювач прогресу;
- спонсори, розробники та користувачі повинні мати можливість підтримувати;
- постійний темп на невизначений термін;
- постійна увага покращення технічної майстерності та зручного дизайну;
- простота – мистецтво не робити зайвої роботи;
- найкращі технічні вимоги, дизайн та архітектура виходять у самоорганізованої команди;
- постійна адаптація до обставин, що змінюються.

Критика. Багато керівників проєктів, які працюють у традиційних методологіях на зразок «водоспаду», критикують agile-методи.

Один із пунктів критики, що повторюється: при agile-підході часто нехтують створенням плану («дорожньої карти») розвитку продукту, так само як і керуванням вимогами, у процесі якого і формується така «карта». Гнучкий підхід до керування вимогами не передбачає далекосяжних планів (по суті, керування вимогами просто не існує в цій методології), а передбачає можливість замовника раптом і несподівано наприкінці кожної ітерації виставляти нові вимоги, що часто суперечать архітектурі вже створеного продукту. Таке іноді призводить до катастрофічних «авралів» із масовим рефакторингом і переробленнями майже на кожній черговій ітерації.

Крім того, вважається, що робота в Agile мотивує розробників вирішувати всі завдання, що надійшли найпростішим і найшвидшим можливим способом, при цьому часто не звертаючи уваги на правильність коду з погляду вимог нижче платформи (підхід – «працює, і добре», при цьому не враховується, що може припинити працювати за найменшої зміни або дати складні до відтворення дефекти після реального розгортання в клієнта). Це призводить до зниження якості продукту та накопичення дефектів.

Методології. Існують методології, які дотримуються цінностей і принципів заявлених в Agile Manifesto. Розглянемо деякі з них.

Agile Modeling – набір понять, принципів і прийомів (практик), що дають змогу швидко й просто виконувати моделювання та документування в проєктах розроблення програмного забезпечення. Agile Modeling не містить детальної інструкції з проєктування, описів, як будувати діаграми на UML. Основна мета цієї методології – ефективне моделювання та документування. Мета не охоплює програмування та тестування, питання керування проєктом, розгортання та супроводу системи, однак передбачає перевірку моделі кодом.

Agile Unified Process (AUP) – це спрощена версія IBM Rational Unified Process (RUP), розроблена Скоттом Амблером, яка описує просту

та зрозумілу модель для створення програмного забезпечення для бізнес-додатків.

Agile Data Method – група ітеративних методів розроблення програмного забезпечення, у яких вимоги та рішення досягаються в межах співпраці різних крос-функціональних команд.

DSDM, оснований на концепції швидкого розроблення програм (Rapid Application Development, RAD), є ітеративним та інкрементним підходом, який надає особливого значення тривалій участі в процесі користувача / споживача.

Extreme programming, XP – найпопулярніша серед гнучких методологій. Має на меті поліпшення якості програмного забезпечення та чутливість до змін у вимогах замовників. Як вид гнучких методологій, XP радить часті «випуски» програми в коротких циклах розроблення, що має на меті поліпшити продуктивність праці та покращити можливості виконання вимог замовника, що змінюються.

Feature driven development (FDD) – функціонально-орієнтована технологія. Поняття функції або властивості, що використовується в FDD системи досить близьке до поняття прецеденту використання, що використовується в RUP, істотна відмінність – це додаткове обмеження: кожна функція має допускати реалізацію не більше ніж за два тижні. Інакше кажучи, якщо сценарій використання досить малий, його можна вважати функцією. Якщо ж великий, його треба розбити на кілька щодо незалежних функцій.

Getting Real – ітеративний підхід без функціональних специфікацій для вебдодатків. У цьому методі спочатку розробляється інтерфейс програми, а потім її функціональна частина.

OpenUP – це ітеративно-інкрементальний метод розроблення програмного забезпечення. Позиціонується як легкий і гнучкий варіант RUP. OpenUP ділить життєвий цикл проєкту на чотири фази: початкова фаза, фази уточнення, побудови та впровадження. Життєвий цикл проєкту забезпечує надання заінтересованим особам і членам колективу точок ознайомлення й ухвалення рішень протягом усього проєкту. Це дає змогу ефективно контролювати ситуацію та вчасно ухвалювати рішення щодо прийнятності результатів. План проєкту визначає життєвий цикл, а кінцевим результатом є остаточний додаток.

Scrum встановлює правила керування процесом розроблення та дає змогу використовувати вже наявні практики кодування, коригуючи вимоги або вносячи тактичні зміни. Використання цієї методології дає можливість виявляти й усувати відхилення від бажаного результату на ранніх етапах розроблення програмного продукту.

Lean software development – ощадливе розроблення програмного забезпечення, що використовує підходи з концепції ощадливого виробництва.

4.2. Методологія Rational Unified Process

Rational Unified Process (RUP) – методологія розроблення програмного забезпечення, створена компанією Rational Software.

В основі RUP лежать такі принципи:

- рання ідентифікація та безперервне (до закінчення проєкту) усунення основних ризиків;
- концентрація на виконанні вимог замовників до виконуваної програми (аналіз та побудова моделі прецедентів (варіантів використання));
- очікування змін у вимогах, проєктних рішеннях і реалізації в процесі розроблення;
- компонентна архітектура, що реалізується й тестується на ранніх стадіях проєкту;
- постійне забезпечення якості на всіх етапах розроблення проєкту (продукту);
- робота над проєктом у згуртованій команді, ключова роль якої належить архітекторам.

Життєвий цикл розроблення. RUP використовує ітеративну модель розроблення. Наприкінці кожної ітерації (в ідеалі, що триває від двох до шести тижнів), проєктна команда повинна досягти запланованих на цю ітерацію цілей, створити або доопрацювати проєктні артефакти й отримати проміжну, але функціональну версію кінцевого продукту. Ітеративне розроблення дає змогу швидко реагувати на мінливі вимоги, виявляти й усувати ризики на ранніх стадіях проєкту, а також ефективно контролювати якість продукту.

Повний життєвий цикл розроблення продукту складається із чотирьох фаз, кожна з яких включає одну або кілька ітерацій:

1. Початок (Inception). У фазі «Початок» відбувається таке:

- формуються бачення та межі проєкту;
- створюється економічне обґрунтування (business case);
- визначаються основні вимоги, обмеження та ключова функціональність продукту;
- створюється базова версія моделі прецедентів;
- оцінюються ризики.

Під час завершення фази «Початок» оцінюється досягнення етапу цілей життєвого циклу (Lifecycle Objective Milestone), яка передбачає угоду заінтересованих сторін про продовження проєкту.

2. Уточнення (Elaboration). У фазі «Уточнення» проводиться аналіз предметної області та побудова архітектури, що виконується, отож фаза передбачає:

- документування вимог (зокрема детальний опис для більшості прецедентів);

- спроєктовану, реалізовану та відтестовану архітектуру, що виконується;
- оновлене економічне обґрунтування та більш точне оцінювання термінів і вартості;
- знижені основні ризики.

Успішне виконання фази розроблення означає досягнення етапу архітектури життєвого циклу (Lifecycle Architecture Milestone).

3. Побудова (Construction). У фазі «Побудова» відбувається реалізація більшої частини функціональності продукту. Фаза «Побудова» завершується першим зовнішнім релізом системи й етапом початкової функціональної готовності (Initial Operational Capability).

4. Упровадження (Transition). У фазі «Упровадження» створюється фінальна версія продукту і передається від розробника до замовника. Ця фаза передбачає бета-тестування, навчання користувачів, а також визначення якості продукту. Якщо якість не відповідає очікуванням користувачів або критеріям, установленим у фазі «Початок», фаза «Упровадження» повторюється знову. Виконання всіх цілей означає досягнення етапу готового продукту (Product Release) та завершення повного циклу розроблення.

5. Ролі та завдання в тестуванні

Розглянемо докладніше наявні активності / завдання, пов'язані з тестуванням.

Планування тестів (Plan Test) передбачає:

- визначення вимог до тестів (identify requirements for test);
- оцінювання ризиків (assess risk);
- розроблення стратегії тестування (develop test strategy);
- визначення ресурсів (identify test resources);
- створення розкладу / послідовностей (create schedule);
- розроблення плану тестування (generate Test Plan).

Дизайн тестів (Design Test) охоплює:

- аналіз обсягу робіт (prepare workload analysis);
- визначення й опис тестових випадків (identify and describe test cases);
- визначення та структурування тестових процедур (identify and structure test procedures);
- огляд та оцінювання тестового покриття (review and assess test coverage).

Розроблення тестів (Implement Test) передбачає:

- запис або програмування тестових скриптів (record or program test scripts);
- визначення критичної функціональності в дизайні та моделі реалізації;

– створення / підготовка зовнішніх наборів даних (establish external data sets).

Виконання тестів (Execute Test) охоплює:

– виконання тестових процедур (execute Test procedures);
– оцінювання виконання тестів (evaluate execution of Test);
– відновлення після збійних тестів (recover from halted Test);
– перевірка результатів (verify the results);
– дослідження несподіваних результатів (investigate unexpected results);
– запис помилок (log defects).

Оцінювання тестів (Evaluate Test) передбачає:

– оцінювання покриття тестовими випадками (evaluate Test-case coverage);
– оцінювання покриття коду (evaluate code coverage);
– аналіз дефектів (analyze defects);
– визначення критеріїв завершення й успішності тестування.

Опис ролей під час тестування наведено в табл. 1.

Таблиця 1

Ролі в тестуванні (roles)

Роль	Опис
Test Manager – тест-менеджер, менеджер проєкту з тестування	Здійснює управлінський контроль (management oversight). Відповідальність: • забезпечує технічний напрям; • отримує необхідні ресурси; • забезпечує управлінську звітність
Test Designer – тест-дизайнер	Визначає, пріоритезує та забезпечує розроблення тестових випадків. Відповідальність: • розробляє план тестування; • розробляє модель тестування; • оцінює ефективність тестування
Tester – тестувальник, інженер з тестування	Виконує тести. Відповідальність: • виконує тести; • фіксує результати; • відновлює тести та систему після збоїв; • документує запити на зміну

Закінчення табл. 1

Роль	Опис
Test System Administrator – адміністратор тестової системи або додатків, які підтримують життєвий цикл тестування	Забезпечує керування та підтримку тестових середовищ і даних. Відповідальність: • адмініструє системи керування тестуванням; • інсталює та керує доступом до тестових систем
Database Administrator / Manager – адміністратор баз даних, менеджер баз даних	Забезпечує керування та підтримку тестових даних (баз даних). Відповідальність: • адмініструє тестові дані (бази даних)

Отже, тестування – цілком певний процес із виділеними ролями та зоною відповідальності для різних гравців проєкту. Порядок перелічення завдань визначає звичайний (повний) цикл проведення тестування. Такий цикл може застосовуватися як для проєктів орієнтованих на тривалі ітерації, так і для «швидких» проєктів, що ведуть за еволюційними методиками (evolutionary) або відповідно до методології XP.

Контрольні запитання

1. Назвіть і прокоментуйте стадії життєвого циклу ПЗ.
2. Які моделі життєвого циклу ви знаєте?
3. Які ви знаєте методології розроблення програмного забезпечення?
4. Які ролі в тестуванні ПЗ?
5. Поясніть, які знання повинен мати тестувальник.
6. Які завдання виконують на етапі планування тестів (Plan Test)?

Лекції № 5–8. КЛАСИФІКАЦІЯ ВИДІВ ТЕСТУВАННЯ

1. Види тестування програмного забезпечення

Усі види тестування програмного забезпечення залежно від цілей можна умовно поділити на такі групи:

- функціональні;
- нефункціональні;
- пов'язані зі змінами.

Розглянемо докладніше кожен окремий вид тестування, його призначення та використання під час тестування програмного забезпечення [6].

1.1. Функціональні види

Функціональні тести базуються на функціях та особливостях, а також взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування: компонентному або модульному (Component / Unit testing), інтеграційному (Integration testing), системному (System testing) і приймальному (Acceptance testing). Функціональні види тестування розглядають зовнішню поведінку системи. Серед найпоширеніших видів функціональних тестів:

- функціональне тестування (Functional Testing);
- тестування безпеки (Security and Access Control Testing);
- тестування взаємодії (Interoperability Testing).

Тестування функціональності (Functional Testing). Тестування функціональності, або функціональне тестування, розглядає заздалегідь зазначену поведінку і ґрунтується на аналізі специфікацій функціональності компонента чи системи загалом.

Функціональні тести ґрунтуються на функціях, що виконуються системою, і можуть проводитися на всіх рівнях тестування (компонентному, інтеграційному, системному, приймальному). Зазвичай ці функції описуються у вимогах, функціональних специфікаціях чи у вигляді випадків використання системи (use cases). Тестування функціональності може проводитись у двох аспектах:

- вимоги;
- бізнес-процеси.

Тестування в аспекті «вимоги» використовує специфікацію функціональних вимог до системи як основу для дизайну тестових випадків (Test Cases). У цьому разі необхідно зробити список того, що буде тестуватися, а що ні, пріоритезувати вимоги на основі ризиків (якщо це не зроблено в документі з вимогами), а на основі цього пріоритезувати тестові сценарії (test cases). Це дасть змогу сфокусуватися і не пропустити

під час тестування найбільш важливий функціонал. Тестування в аспекті «бізнес-процеси» використовує знання цих самих бізнес-процесів, які описують сценарії щоденного використання системи. У цій перспективі тестові сценарії (test scripts) зазвичай ґрунтуються на випадках використання системи (use cases).

Перевагою функціонального тестування є те, що воно імітує фактичне використання системи.

Недоліками функціонального тестування вважають можливість пропустити логічні помилки в програмному забезпеченні та ймовірність надлишкового тестування.

Досить поширеною є автоматизація функціонального тестування.

Тестування безпеки (Security and Access Control Testing). Тестування безпеки – це стратегія тестування, яка використовується для перевірки безпеки системи, а також для аналізу ризиків, пов'язаних із забезпеченням цілісного підходу до захисту програми, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних.

Принципи безпеки програмного забезпечення. Загальна стратегія безпеки ґрунтується на трьох основних принципах:

- конфіденційність;
- цілісність;
- доступність.

Конфіденційність. Конфіденційність – це приховування певних ресурсів чи інформації. Конфіденційність розуміють як обмеження доступу до ресурсу певної категорії користувачів, інакши кажучи, за яких умов користувач має доступ до цього ресурсу.

Цілісність. Існують два основні критерії щодо поняття цілісності:

1. Довіра. Очікується, що ресурс буде змінено лише відповідним способом певною групою користувачів.

2. Пошкодження та відновлення. Якщо дані пошкоджуються або неправильно змінюються авторизованим або неавторизованим користувачем, користувач повинен визначити, наскільки важливою є процедура відновлення даних.

Доступність. Доступність є вимогами про те, що ресурси мають бути доступними авторизованому користувачеві, внутрішньому об'єкту або пристрою. Зазвичай чим критичніший ресурс, тим вищий рівень доступності має бути.

Види вразливості. Нині найпоширенішими видами вразливості в безпеці програмного забезпечення є:

- XSS (CrossSite Scripting) – це вид уразливості програмного забезпечення (вебдодатків), коли на генерованій сервером сторінці виконуються шкідливі скрипти з метою атаки клієнта.
- XSRF / CSRF (Request Forgery) – це вид уразливості, що дає змогу використовувати недоліки HTTP-протоколу, при цьому зловмисники працюють за такою схемою: посилання на шкідливий сайт встановлюється

на сторінці, якій довіряє користувач, у разі переходу за шкідливим посиланням виконується скрипт, що зберігає особисті дані користувача (паролі, платіжні дані тощо), або відправляє СПАМ- повідомлення від імені користувача, або змінює доступ до облікового запису користувача для отримання повного контролю над ним.

- Code injections (SQL, PHP, ASP тощо) – це вид уразливості, при якому стає можливо здійснити запуск коду, що виконується з метою отримання доступу до системних ресурсів, несанкціонованого доступу до даних чи виведення системи з експлуатації.
- ServerSide Includes (SSI) Injection – це вид уразливості, що використовує інтеграцію серверних команд у HTML-код або запуск їх безпосередньо із сервера.
- Authorization Bypass – це вид вразливості, при якому можна отримати несанкціонований доступ до облікового запису або документів іншого користувача.

Тестування програмного забезпечення на безпеку. Наведемо приклади тестування на предмет уразливості в системі безпеки. Для цього необхідно перевірити програмне забезпечення на наявність відомих видів уразливостей, які наведено нижче.

XSS (Cross-Site Scripting). Самі собою XSS-атаки можуть бути дуже різноманітними. Зловмисники можуть спробувати вкрасти куки браузера користувача, перенаправивши його на сайт, де станеться більш серйозна атака, завантажити в пам'ять якийсь шкідливий об'єкт тощо, розмістивши шкідливий скрипт у користувача на сайті. Як приклад можна розглянути поданий нижче скрипт, що виводить на екран куки браузера користувача:

```
<script>alert(document.cookie);</script>
```

або фрагмент коду здійснює повторний вхід на заражену сторінку:

```
<script>window.parent.location.href='http://hacker_site';</script>
```

або створює шкідливий об'єкт із вірусом тощо:

```
<object type="text/xscriptlet»data="http://hacker_site"></object>
```

XSRF / CSRF (Request Forgery). Найбільш частими CSRF атаками є атаки, що використовують HTML <IMG> тег або Javascript об'єкт image. Найчастіше атакувальник додає необхідний код до електронного листа або викладає на вебсайт у такий спосіб, що під час завантаження сторінки здійснюється запит, що виконує шкідливий код. Наприклад:

IMG SRC

<img src="http://hacker_site/?command">

SCRIPT SRC

<script src="http://hacker_site/?command">

Javascript об'єкт Image

<script>

var foo = новий Image();

foo.src = «http://hacker_site/?command";

</script>

Code injections (SQL, PHP, ASP тощо). Вставки виконуваного коду розглянемо на прикладі коду SQL. Форма входу в систему має два поля – ім'я та пароль. Оброблення відбувається в базі даних шляхом виконання SQL-запиту:

SELECT Username

FROM Users

WHERE Name = 'tester'

AND Password = 'testpass'

Уводимо коректне ім'я 'tester', а в полі пароль вводимо рядок:

testpass' OR '1'='1'

Якщо поле не має відповідних валідацій або обробників даних, може виявитися вразливість, що дає змогу зайти в захищену паролем систему, тому що SQL-запит набуде такого вигляду:

SELECT Username FROM Users

WHERE Name = 'tester'

AND Password = 'testpass' OR '1'='1'

Умова '1'='1' завжди буде істинною, тому SQL-запит завжди повертатиме багато значень.

Server-Side Includes (SSI) Injection. Залежно від типу операційної системи команди можуть бути різними, як приклад розглянемо команду, яка виводить на екран список файлів у OS Linux:

< !#exec cmd="ls»> Authorization Bypass

Користувач А може отримати доступ до документів користувача Б. Припустимо, є реалізація, де під час перегляду свого профілю, що містить конфіденційну інформацію, URL-сторінки передається userID, а в цьому випадку є сенс спробувати зазначити замість свого userID номер іншого користувача. І якщо буде видно дані користувача, то це означає, що знайдено дефект.

Отже, прикладів уразливостей та атак існує величезна кількість. Навіть провівши повний цикл тестування безпеки, не можна бути на 100 % упевненим, що система справді безпечна. Але можна бути впевненим у тому, що відсоток несанкціонованих проникнень, крадіжок інформації, втрат даних буде нижчим, ніж у тих, хто не проводив тестування безпеки.

Тестування взаємодії (Interoperability Testing).

З розвитком мережевих технологій та інтернету взаємодія різних систем, сервісів і додатків один з одним набула значної актуальності, оскільки будь-які пов'язані із цим проблеми можуть призвести до зниження авторитету компанії, що, як наслідок, призведе до фінансових втрат. З огляду на це до тестування взаємодії слід ставитися серйозно.

Тестування взаємодії (Interoperability Testing) – це функціональне тестування, що перевіряє здатність програми взаємодіяти з одним і більше компонентами або системами й охоплює тестування сумісності (compatibility testing) та інтеграційне тестування (integration testing).

Програмне забезпечення з хорошими характеристиками взаємодії може бути легко інтегровано з іншими системами, не потребуючи серйозних модифікацій. У цьому випадку кількість змін і час, необхідний на їх виконання, можуть бути використані для вимірювання можливості взаємодії.

1.2. Нефункціональні види тестування

Нефункціональне тестування описує тести, необхідні визначення характеристик програмного забезпечення, які можна виміряти різними величинами. Загалом це тестування того, як система працює. Розглянемо основні види нефункціональних тестів.

Усі види тестування продуктивності:

1. Навантажувальне тестування (Performance and Load Testing).
2. Стресове тестування (Stress Testing).

3. Об'ємне тестування (Volume Testing).
4. Тестування стабільності / надійності (Stability / Reliability Testing).
5. Тестування установки (Installation testing).
6. Тестування зручності користування (Usability Testing).
7. Тестування на відмову та відновлення (Failover and Recovery Testing).
8. Конфігураційне тестування (Configuration Testing).

Навантажувальне тестування / тестування продуктивності. Навантажувальне тестування, або тестування продуктивності, – це автоматизоване тестування, що імітує роботу певної кількості бізнес-користувачів на якомусь загальному ресурсі, який вони розділяють між собою.

Завданням тестування продуктивності є визначення масштабованості програми під навантаженням, при цьому відбувається:

- вимірювання часу виконання обраних операцій за певних інтенсивностей виконання цих операцій;
- визначення кількості користувачів, що одночасно працюють із додатком;
- визначення меж прийнятної продуктивності зі збільшенням навантаження (у разі збільшення інтенсивності виконання цих операцій);
- дослідження продуктивності на високих, граничних, стресових навантаженнях.

Load vs Performance Testing. В англомовній термінології можна знайти ще один вид тестування – Load Testing – тестування реакції системи на зміну навантаження (у межі допустимого). Load і Performance мають однакову мету – перевірка продуктивності (часу відгуку) на різних навантаженнях. Саме тому їх не поділяють. Водночас ці поняття можуть поділяти. Головне – розуміти мету того чи іншого виду тестування і намагатися їх досягти.

Стресове тестування (Stress Testing). Стресове тестування дає змогу перевірити, наскільки додаток і система загалом працездатні за умов стресу, й оцінити здатність системи до регенерації, тобто до повернення до нормального стану після припинення дії стресу. Стресом у цьому контексті може бути підвищення інтенсивності виконання операцій до дуже високих значень або зміна аварійної конфігурації сервера. Також одним із завдань під час стресового тестування може бути оцінювання деградації продуктивності, таким чином, цілі стресового тестування можуть збігатися з метою тестування продуктивності.

Об'ємне тестування (Volume Testing). Завданням об'ємного тестування є отримання оцінки продуктивності зі збільшенням обсягів даних у базі даних програми, з урахуванням таких параметрів:

- вимірювання часу виконання вибраних операцій за певних інтенсивностей виконання цих операцій;

- можливість визначення кількості користувачів, що одночасно працюють із додатком.

Тестування стабільності / надійності (Stability / Reliability Testing). Завданням тестування стабільності (надійності) є перевірка працездатності програми під час тривалого (багатогодинного) тестування із середнім рівнем навантаження. Час виконання операцій може відігравати в цьому виді тестування другорядну роль. При цьому на перше місце стає відсутність витоків пам'яті, перезапусків серверів під навантаженням та інші аспекти, що впливають саме на стабільність роботи.

Тестування установки (Installation Testing). Тестування установки спрямоване на перевірку успішної інсталяції та настроювання, а також оновлення або видалення програмного забезпечення.

Зараз найбільш поширеною установкою ПЗ за допомогою інсталяторів (спеціальних програм, які самі по собі так само потребують належного тестування).

Реальних умов інсталяторів може не бути. У цьому випадку доведеться самостійно виконувати встановлення програмного забезпечення, використовуючи документацію у вигляді інструкцій або readme-файлів, що крок за кроком описують усі необхідні дії та перевірки.

У розподілених системах, де програма розгортається на середовищі, що вже працює, простого набору інструкцій може бути мало. Для цього часто пишеться план установки (Deployment Plan), що включає не тільки кроки з інсталяції програми, але і кроки відкату (rollback) до попередньої версії в разі невдачі. План установки також має пройти процедуру тестування, щоб уникнути проблем під час видання в реальну експлуатацію. Особливо це актуально, якщо установка виконується на системи, де кожна хвилина простою – це втрата репутації та великої кількості коштів, наприклад банки, фінансові компанії або навіть банерні мережі. Отже, тестування установки можна назвати одним із найважливіших завдань щодо забезпечення якості програмного забезпечення.

Саме такий комплексний підхід із написанням планів, покроковою перевіркою встановлення та відкату інсталяції, можна назвати тестуванням установки, або інсталяційним тестуванням.

Тестування зручності користування (Usability Testing). Іноді користувачі стикаються з незрозумілими й нелогічними додатками, багато функцій і способів використання яких часто не очевидні. Після такої роботи рідко виникає бажання використовувати програму знову, і користувач шукає більш зручні аналоги. Щоб додаток був популярним, йому недостатньо бути функціональним – він має бути ще й зручним. Якщо замислитися, інтуїтивно зрозумілі програми заощаджують час користувачів і витрати роботодавця на навчання. Отже, такі додатки є більш конкурентоспроможними. Отож тестування зручності використання, про

яке йтиметься далі, є невід'ємною частиною тестування будь-яких масових продуктів.

Тестування зручності користування – це метод тестування, спрямований на встановлення ступеня зручності використання, навченості, зрозумілості та привабливості для користувачів продукту, що розробляється в контексті заданих умов (ISO 9126).

Критерії тестування зручності користування. Тестування зручності користування дає оцінку рівня зручності використання програми за такими пунктами:

- продуктивність, ефективність (efficiency) – скільки часу та кроків знадобиться користувачеві для завершення основних завдань програми, наприклад розміщення новини, реєстрації, покупки тощо (чим менше, тим краще);
- правильність (accuracy) – скільки помилок зробив користувач під час роботи з програмою (чим менше, тим краще);
- активізація в пам'яті (recall) – як багато користувач пам'ятає про роботу програми після призупинення роботи з нею на тривалий час (повторне виконання операцій після перерви має відбуватися швидше, ніж у нового користувача);
- емоційна реакція (emotional response) – як користувач почувається після завершення завдання: розгублений, відчув стрес чи рекомендує користувач систему своїм друзям (позитивна реакція краще).

Рівні проведення. Перевірка зручності використання може проводитися як до готового продукту за допомогою тестування чорної скриньки (black box testing), так і до інтерфейсів програми (API), що використовуються під час розроблення тестування білої скриньки (white box testing). У цьому випадку перевіряється зручність використання внутрішніх об'єктів, класів, методів і змінних, а також розглядається зручність зміни, розширення системи й інтеграції її з іншими модулями чи системами. Використання зручних інтерфейсів (API) може покращити якість, збільшити швидкість написання та підтримки коду, що розробляється, і, як наслідок, покращити якість продукту в цілому. Отож стає очевидним, що тестування зручності користування може проводитись на різних рівнях розроблення програмного забезпечення: модульному, інтеграційному, системному і приймальному. При цьому воно цілком і повністю буде залежить від того, хто буде використовувати додаток на виділеному конкретному рівні – розробник, бізнес-користувач системи тощо.

Поради щодо покращення зручності користування. Для дизайну зручних програм корисно дотримуватися принципів poka-yoke, або failsafe. У нас це більш відомо як «захист від дурня». Наприклад, якщо поле потребує цифрового значення, логічно обмежити користувачеві діапазон введення лише цифрами – буде менше випадкових помилок.

Для підвищення зручності використання наявних програм можна використовувати цикл Демінга Plan-Do-Check-Act, збираючи відгуки про роботу та дизайн програми в наявних користувачів і плануючи та проводячи покращення відповідно до їх зауважень.

Помилки про тестування зручності користування. Тестування зручності користування немає нічого спільного з тестуванням функціональності інтерфейсу, воно лише проводиться на інтерфейсі, як і на багатьох інших можливих компонентах продукту. При цьому тип тестування та тест-кейси будуть зовсім іншими, тому що може йтися про зручність використання не візуальних компонентів (якщо такі є) або процес адміністрування, наприклад, розподіленого клієнт-серверного продукту тощо.

Тестування зручності користування не можна провести без участі експерта. Людина, яка не знається на предметній області, не здатна провести її самостійно. Уявіть, що тестувальнику потрібно протестувати зручність користування стратегічним бомбардувальником. Йому доведеться перевірити основні функції: зручність ведення бою, навігації, пілотування, обслуговування, наземного транспортування тощо. Очевидно, що без залучення експерта це буде дуже проблематично, і можна сказати, що неможливо.

Тестування на відмову та відновлення (Failover and Recovery Testing). Цей вид тестування перевіряє тестований продукт з погляду здатності протистояти й успішно відновлюватися після можливих збоїв, що виникли у зв'язку з помилками програмного забезпечення, відмови обладнання або проблемами зв'язку (наприклад, відмова мережі). Метою цього виду тестування є перевірка систем відновлення (систем, що дублюють основний функціонал), які в разі виникнення збоїв забезпечать збереження та цілісність даних продукту, що тестується.

Тестування на відмову та відновлення дуже важливе для систем, що працюють за принципом 24/7. Якщо створюється продукт, який працюватиме, наприклад, в інтернеті, то без проведення цього виду тестування просто не обійтися. Оскільки кожна хвилина простою або втрата даних у разі відмови обладнання може коштувати грошей, втрати клієнтів і репутації на ринку.

Методика подібного тестування полягає в симулюванні різних умов збою.

Для наочності розглянемо деякі варіанти такого тестування та загальні методи їх проведення. Об'єктом тестування здебільшого є ймовірні експлуатаційні проблеми, такі як:

- відмова електрики на комп'ютері;
- незавершені цикли оброблення даних (переривання роботи фільтрів даних, переривання синхронізації);
- оголошення або внесення до масивів даних неможливих чи помилкових елементів;

- відмова від носіїв даних.

Ці ситуації можуть бути відтворені, щойно досягнуто певного етапу в розробленні, коли всі системи відновлення або дублювання готові виконувати свої функції. Технічно реалізувати тести можна такими шляхами симулювання:

- раптової відмови електрики на комп'ютері (знеструмити комп'ютер);
- втрати зв'язку з мережею (вимкнути мережний кабель, знеструмити мережевий пристрій);
- відмови носіїв (знеструмити зовнішній носій даних);
- ситуації наявності в системі неправильних даних (спеціальний тестовий набір чи база даних).

Після досягнення відповідних умов збою та за результатами роботи систем відновлення можна оцінити продукт із погляду тестування на відмову. У всіх випадках після завершення процедур відновлення має бути досягнуто певного необхідного стану даних продукту:

- втрата або псування даних у допустимих межах;
- звіт чи система звітів із зазначенням процесів чи транзакцій, які були завершені внаслідок збою.

Слід зауважити, що тестування на відмову та відновлення – це дуже специфічне тестування. Розроблення тестових сценаріїв має проводитися з урахуванням усіх особливостей системи, що тестується. З огляду на досить жорсткі методи впливу слід також оцінити доцільність проведення цього виду тестування для конкретного програмного продукту.

Конфігураційне тестування (Configuration Testing). Це спеціальний вид тестування, спрямований на перевірку роботи програмного забезпечення при різних конфігураціях системи (заявлених платформах, драйверах, що підтримуються, при різних конфігураціях комп'ютерів і т.д.).

Залежно від типу проєкту конфігураційне тестування може мати різні цілі:

1. Проєкт із профілювання роботи системи. Мета такого тестування – визначити оптимальну конфігурацію обладнання, що забезпечує необхідні характеристики продуктивності та часу реакції системи, що тестується.
2. Проєкт міграції системи з однієї платформи на іншу. Мета тестування – перевірити об'єкт тестування на сумісність з оголошеним у специфікації обладнанням, операційними системами та програмними продуктами третіх фірм.

У ISTQB Syllabus взагалі не йдеться про такий вид тестування як конфігураційне. Відповідно до глосарію цей вид тестування розглядається як тестування портованості.

Рівні проведення тестування. Для клієнт-серверних додатків конфігураційне тестування можна умовно поділити на два рівні (для деяких типів додатків може бути актуальним лише один):

1. Серверний.
2. Клієнтський.

На першому (серверному) рівні, тестується взаємодія програмного забезпечення із середовищем, у яке воно буде встановлено:

– апаратні засоби (тип і кількість процесорів, ємність пам'яті, характеристики мережі / мережевих адаптерів тощо);

– програмні засоби (ОС, драйвери та бібліотеки, стороннє ПЗ, що впливає на роботу програми тощо).

Основний акцент тут робиться на тестування з метою визначення оптимальної конфігурації обладнання, що відповідає необхідним характеристикам якості (ефективність, портативність, зручність супроводу, надійність).

На другому (клієнтському) рівні програмне забезпечення тестується з позиції його кінцевого користувача та конфігурації його робочої станції. На цьому етапі будуть протестовані такі характеристики: зручність використання, функціональність. Для цього необхідно буде провести низку тестів із різними конфігураціями робочих станцій:

1. Тип, версія та бітність операційної системи (подібний вид тестування називається кросплатформне тестування).
2. Тип і версія веббраузера, якщо тестується вебдодаток (подібний вид тестування називається кросбраузерне тестування).
3. Тип і модель відеоадаптера (під час тестуванні ігор це дуже важливо).
4. Робота програми за умов різних роздільних здатностей екрана.
5. Версії драйверів, бібліотек тощо (для JAVA-додатків версія JAVA-машини є дуже важливою, те саме можна сказати про .NET-додатків щодо версії .NET-бібліотеки) тощо.

Порядок проведення тестування. Перед початком конфігураційного тестування рекомендується:

- створювати матрицю покриття (матриця покриття – це таблиця, у яку заносять всі можливі конфігурації);
- проводити пріоритезацію конфігурацій (на практиці, найімовірніше, всі бажані конфігурації перевірити не вдасться);
- крок за кроком відповідно до розставлених пріоритетів перевіряють кожну конфігурацію.

Уже на початковому етапі стає очевидним, що чим більше вимог до роботи програми при різних конфігураціях робочих станцій, тим більше тестів нам необхідно буде провести. У зв'язку із цим краще наскільки можна автоматизувати цей процес, оскільки саме за конфігураційного тестування автоматизація реально допомагає заощадити час і ресурси.

Зазвичай автоматизоване тестування не є «панацеєю», але в цьому випадку воно виявиться дуже ефективним помічником.

Отже, конфігураційним називається тестування сумісності виробленого продукту (програмне забезпечення) з різними апаратними і програмними засобами. Тестування має основні цілі визначення оптимальної конфігурації та перевірку сумісності програми з необхідним середовищем (обладнанням, ОС тощо), а автоматизація конфігураційного тестування дає змогу уникнути зайвих витрат.

1.3. Види тестування, пов'язані зі змінами

Після проведення необхідних змін, таких як виправлення бага / дефекту, програмне забезпечення має бути перевірене для підтвердження того факту, що проблему було дійсно розв'язано. Нижче наведено види тестування, які необхідно проводити після встановлення програмного забезпечення для підтвердження працездатності програми або правильності здійсненого виправлення дефекту:

1. Димове тестування (Smoke Testing).
2. Регресійне тестування (Regression Testing).
3. Тестування складання (Build Verification Test).
4. Перевірка узгодженості / справності (Sanity Testing).

Димове тестування (Smoke Testing). Поняття «димове тестування» походить з інженерного середовища. Під час введення в експлуатацію нового обладнання («заліза») вважалося, що тестування пройшло вдало, якщо з установки не пішов дим.

В галузі програмного забезпечення димове тестування розглядається як короткий цикл тестів, що виконується для підтвердження того, що після складання коду (нового або виправленого) додаток, що встановлюється, стартує і виконує основні функції.

Висновок про працездатність основних функцій робиться на підставі результатів поверхневого тестування найбільш важливих модулів додатка на предмет можливості виконання необхідних завдань і наявності критичних і дефектів, що блокуються і швидко знаходяться. У разі відсутності таких дефектів димове тестування вважається виконаним, і додаток передається щодо повного циклу тестування, інакше, димове тестування вважється невдалим, і додаток доопрацьовується.

Аналогами димового тестування є Build Verification Testing і Acceptance Testing, виконувані на функціональному рівні командою тестування, за результатами яких робиться висновок про те, чи приймається / не приймається встановлена версія програмного забезпечення в тестування, експлуатацію або на постачання замовнику.

Для полегшення роботи, економії часу та людських ресурсів рекомендується впровадити автоматизацію тестових сценаріїв для димового випробування.

Регресійне тестування (Regression Testing). Це вид тестування, спрямований на перевірку змін, зроблених у додатку або навколишньому середовищі (лагодження дефекту, злиття коду, міграція на іншу операційну систему, базу даних, вебсервер або сервер програми), для підтвердження того факту, що наявна раніше функціональність працює як і раніше. Регресійними можуть бути як функціональні, так і дисфункційні тести.

Зазвичай для регресійного тестування використовуються тест-кейси, написані на ранніх стадіях розроблення та тестування. Це дає гарантію того, що зміни в новій версії програми не пошкодили вже наявну функціональність. Рекомендується робити автоматизацію регресійних тестів для прискорення подальшого процесу тестування та виявлення дефектів на ранніх стадіях розроблення програмного забезпечення.

Термін «регресійне тестування» залежно від контексту використання може мати різний зміст. Сем Канер, наприклад, описав три основні типи регресійного тестування:

1. Регресія багів (Bug regression) – спроба довести, що виправлена помилка насправді не виправлена.
2. Регресія старих багів (Old bugs regression) – спроба довести, що зміна коду чи даних зламало виправлення старих помилок, тобто старі баги почали знову відтворюватись.
3. Регресія побічного ефекту (Side effect regression) – спроба довести, що недавня зміна коду або даних зламала інші частини програми, що розробляється.

Під час тестування змін у системі дуже важливо зрозуміти різницю між поняттями «регресійне тестування» (regression testing) та «повторне тестування» (retesting) (табл. 2).

Регресійне тестування (regression testing) проводиться з метою перевірки працездатності наявного функціонала та відсутності сторонніх помилок після внесення поправок чи доповнень до системи.

Повторне тестування (Retesting) проводиться для підтвердження виправлення помилки та роботи цього функціонала.

Тестування складання (Build Verification Test). Тестування спрямоване на визначення відповідності випущеної версії критеріям якості початку тестування. За своїми цілями є аналогом димового тестування, спрямованого на прийняття нової версії в подальше тестування чи експлуатацію. Углиб воно може поширюватися залежно від вимог якості випущеної версії.

Перевірка узгодженості / справності (Sanity Testing). Санітарне тестування – це вузькоспрямоване тестування, достатнє для доказу того, що конкретна функція працює відповідно до заявлених у специфікації вимог. Є підмножиною регресійного тестування. Використовується для визначення працездатності певної частини програми після змін вироблених у ній чи навколишньому середовищі. Зазвичай виконується вручну.

Відмінність санітарного тестування від димового (Sanity vs Smoke testing). Деяких джерелах помилково вважають, що санітарне та димове тестування – це те саме. Проте ці види тестування мають «вектори руху», напрямки в різні боки. Санітарне тестування (Sanity testing), на відміну від димового (Smoke testing), спрямоване вглиб функції, що перевіряється, тоді як димове направлено вшир, для покриття тестами якомога більшого функціонала в найкоротші терміни.

Таблиця 2

Різниця між видами тестування

Регресійне тестування	Повторне тестування
Виконується лише за умови додавання нової додаткової функціональності ПЗ або суттєвої зміни і функціонала в системі	Ретест виконується в тому самому середовищі і з тими самими даними, але на новій збірці (build) ПЗ
Регрес можна проводити паралельно із повторним тестуванням	Має вищий пріоритет, має бути виконано до регресійного
Тест-кейси може бути автоматизовано	Тест-кейси не може бути автоматизовано
У межах регресійного тестування тест-кейси, які було позначено раніше як Passed, має бути перевірено повторно	У межах повторного тестування (ретесту) перевіряються лише невдалі (зі статусом Failed) тест-кейси

2. Рівні тестування програмного забезпечення

Тестування на різних рівнях проводиться протягом усього життєвого циклу розроблення та супроводу програмного забезпечення. Рівень тестування визначає те, над чим виробляються тести: над окремим модулем, групою модулів чи системою загалом. Проведення тестування на всіх рівнях системи – це запорука успішної реалізації проєкту.

Існують такі рівні тестування ПЗ:

1. Модульне чи компонентне тестування (Unit Testing or Component Testing).
2. Інтеграційне тестування (Integration Testing).
3. Системне тестування (System Testing).
4. Приймальний тест (Acceptance Testing).

Контрольні запитання

1. Які існують класифікації за видами тестування?
2. Що включає регресійне тестування?
3. У яких випадках використовується інтуїтивне тестування?
4. Що являє собою тестування навантаження?
5. Що таке інтеграційне тестування?
6. Чим відрізняється тестування чорної скриньки від білої?

Лекції № 9, 10. ЗВІТ ПРО ПОМИЛКУ

1. Життєвий цикл дефекту. Атрибути дефекту

Будь-який дефект має різні варіанти життєвого циклу. Найпоширеніші варіанти різновидів життевого циклу дефекту наведено на рис. 8.

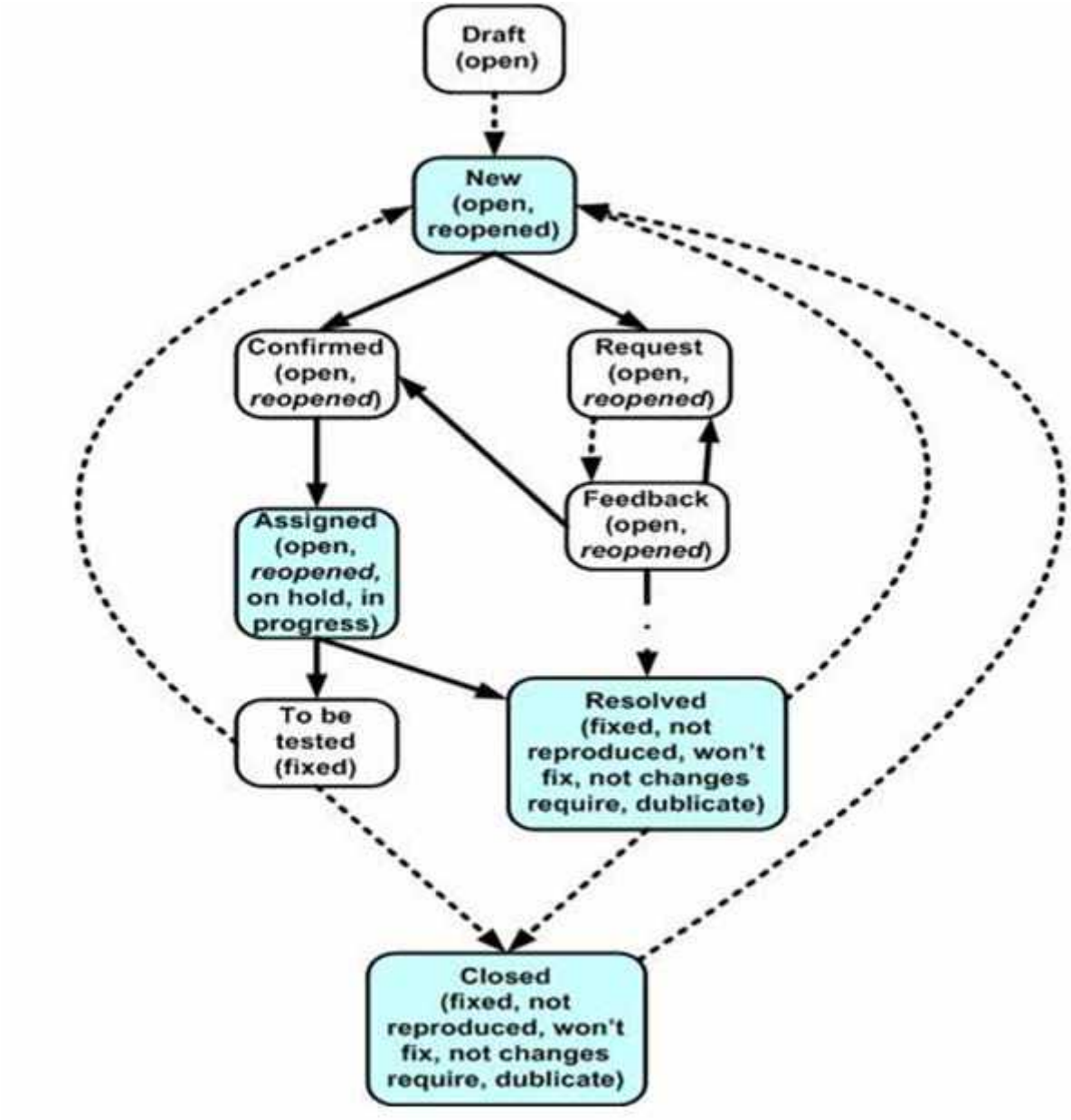

Рис. 8. Життєвий цикл помилки (бага)

Розглянемо основні етапи ЖЦ помилки:

- draft (чернетка) – новий робочий запис із незакінченим оформленням;

- new (новий) – новий запис (питання, баг, пропозиція тощо) – уточнюється відповідною критичністю (severity);
- feedback (інформація) – відповіді тестувальників на запитання розроблювачів, відновлення або додавання інформації із запису;
- request (запит) – запит розроблювачів до тестувальників з уточнення інформації або інших причин, пов'язаних із цим записом;
- confirmed (визнаний) – розроблювачами визнана наявність дефекту або необхідності ухвалення рішення з порушеного в записі питання;
- assigned (призначений) – призначений відповідальний розроблювач із виправлення дефекту або ухвалення рішення щодо розв'язання проблеми або вирішення питання, порушеного в цьому записі;
- to be tested (тестувати) – можливо вирішений, потрібна перевірка;
- resolved (вирішений) – ухвалене рішення – розроблювач виправив баг або ухвалив відповідне рішення;
- closed (закритий) – запис закритий – підтвердження тестувальниками виправленності дефекту або рішення, ухваленого розроблювачами щодо цього запису.

2. Баг- / дефект-репорт (Bug / issue report)

Баг- або дефект-репорт – це документ, що описує ситуацію або послідовність дій, яка призвела до некоректної роботи об'єкта тестування, із вказівкою причин і очікуваного результату [7].

Щоб отримати більш детальну інформацію про баг-репорт, слід ознайомитись з необхідною інформацію, щоб мати вичерпне уявлення про структуру, особливість написання й деякі інші нюанси, необхідні для написання вдалих баг-репортів, зокрема про таке:

- структуру баг-репорту;
- серйозність і пріоритет дефекту;
- написання баг-репортів.

Наведемо коментар одного розроблювача: «Прочитавши короткий опис бага (Bug Summary), я повинен зрозуміти в чому полягає проблема, прочитавши детальний опис бага (Bug Description), я повинен знати рядок коду, який правити».

Із цим висловлюванням можна погоджуватися або не погоджуватися, але зміст цього висловлення в тім, що треба робити все так, щоб було менше питань по суті, описаних у баг-репорті проблеми. Оскільки кожний повернутий баг-репорт зі статусом «Відхилений», «Не відтворюється», «Потрібна інформація» (Rejected, Can't Reproduce, More info) – це втрата часу. А час, як відомо, – це гроші, які одержують за те, що роблять роботу краще за всіх!

Якщо дотримуватися всіх запропонованих рекомендацій, то якість баг-репортів буде на високому рівні, а в процесі роботи буде найменше претензій як від менеджерів, так від розроблювачів.

2.1. Основні поля баг- / дефект-репорту

Різні системи менеджменту дефектами, пропонують різні поля для заповнення й різні структури опису дефектів. Подана нижче таблиця – це спроба показати те, що на підставі отриманого досвіду, слід використати у вигляді шаблону баг-репорту (табл. 3).

Таблиця 3

Основні поля баг-репорту

Назва	Опис
Короткий опис (Summary)	Короткий опис проблеми, що явно вказує на причину й тип помилкової ситуації
Проєкт (Project)	Назва проєкту, що тестується
Компонент додатка (Component)	Назва частини або функції продукту, що тестується
Номер версії (Version)	Версія, на якій було знайдено помилку
Серйозність (Severity)	Найпоширеніша п'ятирівнева система градації серйозності дефекту: S1 блокувальна (Blocker); S2 критична (Critical); S3 значна (Major); S4 незначна (Minor); S5 тривіальна (Trivial)
Пріоритет (Priority)	Пріоритет дефекту: P1 високий (High); P2 середній (Medium); P3 низький (Low)
Статус (Status)	Статус бага. Залежить від використовуваної процедури й життєвого циклу бага (bug workflow and life cycle)
Автор (Author)	Творець баг-репорту

Закінчення табл. 3

Назва	Опис
Призначений для (Assigned To)	Ім'я співробітника, призначеного для розв'язання проблеми
Кроки відтворення (Steps to Reproduce)	Кроки, за якими можна легко відтворити ситуацію, що призвела до помилки
Фактичний результат (Result)	Результат, отриманий після проходження кроків до відтворення
Очікуваний результат (Expected Result)	Результат, отриманий відповідно до специфікації
Прикріплений файл (Attachment)	Файл із логами, скриншот або будь-який інший документ, що може допомогти прояснити причину помилки або вказати на спосіб розв'язання проблеми

Серйозність і пріоритет дефекту. Різні системи баг-трекінгу пропонують нам різні шляхи опису серйозності й пріоритету баг-репорту, незмінним залишається лише зміст, вкладений у ці поля. Усі знають такий баг-трекер, як Atlassian JIRA. У ньому з якоїсь версії замість одночасного використання полів Severity і Priority залишилися тільки Priority, що мало в собі властивості обох полів (Originally, JIRA did have both a Priority and a Severity field. The Severity field was removed for a number of reasons...). Таким чином, ті, хто звик працювати з JIRA, не завжди розуміють різницю між цими поняттями, тому що не мали досвіду їх спільного використання. З огляду на практичний досвід можна стверджувати, що краще поділяти ці поняття – Severity і Priority, тому що ці поля мають різне призначення.

Серйозність (Severity) – це атрибут, що характеризує вплив дефекту на працездатність додатка.

Пріоритет (Priority) – це атрибут, що вказує на черговість виконання завдання або усунення дефекту. Можна сказати, що це інструмент менеджера з планування робіт. Чим вищий пріоритет, тим швидше потрібно виправити дефект.

Градація серйозності дефекту (Severity):

– S1 блокувальна помилка (Blocker) – помилка, яка блокує і яка призводить до неробочого стану додаток, унаслідок чого подальша робота із системою, що тестується, або її ключовими функціями стає неможливою; розв'язання проблеми необхідно для подальшого функціонування системи;

– S2 критична (Critical) – критична помилка, ключова бізнес-логіка, яка неправильно працює, прогалина в системі безпеки, проблема, що призвела до тимчасового «падіння» сервера або що призводить до неробочого стану деякі частини системи без можливості розв'язання проблеми з використанням інших варіантів входу; розв'язання проблеми необхідно для подальшої роботи із ключовими функціями системи, що тестується;

– S3 значна (Major) – значна помилка, частина основний бізнес-логіки працює некоректно; помилка не критична або є можливість для роботи з функцією, що тестується, використовуючи інші варіанти входу;

– S4 незначна (Minor) – незначна помилка, що не порушує бізнес-логіку частини додатка, що тестується, очевидна проблема користувацького інтерфейсу;

– S5 тривіальна (Trivial) – тривіальна помилка, що не стосується бізнес-логіки додатка, погано відтворена проблема, малопомітна в користувацькому інтерфейсі, проблема сторонніх бібліотек або сервісів, проблема, що жодним чином не впливає на загальну якість продукту.

Градація пріоритету дефекту (Priority):

– P1 високий (High) – помилку має бути виправлено якнайшвидше, тому що її наявність є критичною для проєкту;

– P2 середній (Medium) – помилку має бути виправлено, її наявність не є критичною, але потребує обов'язкового усунення;

– P3 низький (Low) – помилку має бути виправлено, її наявність не є критичною, але потребує термінового усунення.

Порядок виправлення помилок за їх пріоритетами: High -> Medium -> Low.

Вимоги до кількості відкритих багів. Розглянемо інший підхід до визначення вимог до кількості відкритих багів.

Наявність відкритих дефектів P1, P2 і S1, S2 вважається неприйнятною для проєкту. Усі подібні ситуації потребують термінового вирішення і контролю менеджерів проєкту.

Наявність суворо обмеженої кількості відкритих помилок P3 і S3, S4, S5 не є критичною для проєкта та допускається у виданому додатку. Кількість відкритих помилок залежить від розміру проєкту та встановлених критеріїв якості.

Усі вимоги до відкритих помилок обмовляються й документуються на етапі ухвалення рішення про якість розроблювального продукту.

Написання баг-репорту. Баг-репорт – це технічний документ, тому у зв'язку із цим слід відзначити, що мова опису проблеми має бути технічною. Повинна використовуватися правильна термінологія під час використання назв елементів користувацького інтерфейсу (editbox, listbox, combobox, link, text area, button, menu, popup menu, title bar, system tray тощо), дій користувача (click link, press the button, select menu item тощо) і

отриманих результатах (window is opened, error message is displayed, system crashed тощо).

Вимоги до обов'язкових полів баг-репорту. Зауважимо, що обов'язковими полями баг-репорту є короткий опис (Bug Summary), серйозність (Severity), кроки до відтворення (Steps to reproduce), результат (Actual Result), очікуваний результат (Expected Result). Нижче наведено вимоги й приклади щодо заповнення цих полів.

Короткий опис. В одному реченні треба вмістити зміст усього баг-репорту коротко і ясно і, використовуючи правильну термінологію, сказати що й де не працює. Наприклад:

1. Додаток «зависає» під час спроби збереження текстового файлу розміром більше 50 Мб.
2. Дані на формі «Профайл» не зберігаються після натискання кнопки «Зберегти».

Для роботи з баг-репопртами необхідно також вивчити принцип «Де? Що? Коли?»:

1. Де? У якому місці інтерфейсу користувача або архітектури програмного продукту міститься проблема; слід починати речення з іменника, а не прийменника.
2. Що? Що відбувається або не відбувається згідно зі специфікацією або вашим уявленням про нормальну роботу програмного продукту, при цьому слід вказувати на наявність або відсутність об'єкта проблеми, а не його зміст (його вказують в описі); якщо зміст проблеми варіюється, усі відомі варіанти вказуються в описі.
3. Коли? У який момент роботи програмного продукту, після настання якої події або за яких умов проблема проявляється, чому послідовність має бути саме такою.

У такому вигляді незнайомі дефекти зручніше сортувати за summary, як показує практика (адже, найімовірніше, саме серед дефектів інших інженерів буде виконуватися пошук дублікатів). Якщо ви іншої думки – придумайте свою послідовність, але вона має стати єдиною для всіх без винятку членів проєкту, інакше не буде досягнуто необхідного результату.

Серйозність. Якщо проблему знайдено в ключовій функціональності додатка, а після її виникнення додаток стає повністю недоступним, то подальша робота з ним неможлива – вона є блокувальною. Зазвичай всі проблеми, що блокують, відшукуються під час первинної перевірки нової версії продукту (Build Verification Test, Smoke Test), оскільки їх наявність не дає змоги повноцінно проводити тестування. Якщо ж тестування може бути продовжено, то серйозність цього дефекту буде критичною. Питання значних, незначних і тривіальних помилок досить зрозуміле і не потребує зайвих пояснень.

Кроки до відтворення / Результат / Очікуваний результат. Дуже важливо чітко описати всі кроки, зі згадуванням усіх вхідних даних (імені

користувача, даних для заповнення форми) і проміжних результатів. Наприклад, кроки відтворення:

1. Увійти в систему: користувач Тестер1, пароль xxxXXX > вхід у систему здійснено.

2. Клікнути на лінк «Профайл» > сторінку «Профайл» відкрито.

3. Увести нове ім'я користувача: Тестер2.

4. Натиснути кнопку «Зберегти результат».

Фактичний результат: На екрані з'явилася помилка. Нове ім'я користувача не було збережено.

Очікуваний результат: Сторінка профайл перевантажилася. Нове значення імені користувача збережено.

Основні помилки під час написання баг-репортів:

1. Недостатність наданих даних. Не завжди та сама проблема проявляється під час всіх значень, що вводяться, і для будь-якого користувача, що ввійшов у систему, тому рекомендується вносити всі необхідні дані в баг-репорт.

2. Визначення серйозності. Дуже часто відбувається або завищення, або заниження серйозності дефекту, що може призвести до неправильної черговості під час розв'язання проблеми.

Мова опису. Часто під час опису проблеми використовується неправильна термінологія або складні мовні звороти, які можуть ввести в оману людину, відповідальну за розв'язання проблеми.

Відсутність очікуваного результату. У випадках, якщо не вказано, що ж має бути необхідною поведінкою системи, витрачається час розроблювача на пошуки цієї інформації, тим самим сповільнюєте виправлення дефекту. Треба вказати пункт у вимогах, написаний тест-кейс або ж особисту думку, якщо цю ситуацію не було документовано.

Заповнення полів баг-репорту. В описаній нижче таблиці подано основні поля баг-репорту й роль працівника, відповідального за заповнення цього поля. Поля, позначені зірочкою, є обов'язковими для заповнення (табл. 4).

Таблиця 4

Поля баг-репорту

Поле	Відповідальний за заповнення поля
Короткий опис (Summary)*	Автор баг-репорту (зазвичай це тестувальник)
Проєкт (Project)*	Автор баг-репорту (зазвичай це тестувальник)
Компонент додатка (Component)	Автор баг-репорту (зазвичай це тестувальник)
Номер версії (Version)	Автор баг-репорту (зазвичай це тестувальник)

Закінчення табл. 4

Поле	Відповідальний за заповнення поля
Серйозність (Severity)	Автор баг-репорту (зазвичай це тестувальник), однак цей атрибут може бути змінено вищим менеджером
Пріоритет (Priority)*	Менеджер проєкту або менеджер, відповідальний за розроблення компонента, на який написано баг-репорт
Статус (Status)*	Автор баг-репорту (зазвичай це тестувальник), але багато систем баг-трекінгу встановлюють статус за стандартною настройкою
Автор (Author)	Установлюється за стандартною настройкою, якщо ні, то вказується ім'я автора баг-репорту
Призначений на (Assigned To)	Менеджер проєкту або менеджер, відповідальний за розроблення компонента, на який написано баг-репорт
ОС / Сервіс-Пак тощо / Браузера + версія / ...	Автор баг-репорту (зазвичай це тестувальник)
Кроки відтворення (Steps to Reproduce)*	Автор баг-репорту (зазвичай це тестувальник)
Фактичний результат (Result)*	Автор баг-репорту (зазвичай це тестувальник)
Очікуваний результат (Expected Result)*	Автор баг-репорту (зазвичай це тестувальник)
Прикріплений файл (Attachment)	Автор баг-репорту (зазвичай це тестувальник), а також будь-який член командної групи, який вважає, що прикріплені дані допоможуть у виправленні бага

Контрольні запитання

1. Що таке життєвий цикл помилки?
2. Який стан має помилка після її внесення до системи багтрекінгу?
3. Що означає стан помилки Closed?
4. Що таке багтрекінгова система і для чого вона призначена?
5. Які ви знаєте багтрекінгові системи?

6. Перерахуйте та прокоментуйте основні поля під час занесення помилки багтрекінгової системи.

7. За яким принципом заповнюється поле Summary?

8. Як визначити критичність помилки?

9. Що таке кроки щодо відтворення помилки і для чого вони потрібні в багтрекінговій системі?

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Gayathri, M. Full Stack Testing. A Practical Guide for Delivering High Quality Software / M. Gayathri. – Sebastopol : O'Reilly, 2022. – 406 p.
2. Крепич, С. Я. Якість програмного забезпечення та тестування: базовий курс: навч. посіб. / С. Я. Крепич, І. Я. Спивак. – Тернопіль : ФОП Паляниця В. А., 2020. – 479 с.
3. Якість та тестування інформаційних систем: навч. посіб. / О. А. Золотухіна, О. В. Негоденко, С. Ю. Резник, С. Я. Разіна. – Київ : ННІТ ДУТ, 2020. – 128 с.
4. Литвин, В. В. Проєктування інформаційних систем: навч. посіб. / В. В. Литвин, Н. Б. Шаховська, В. В. Пасічник. – Львів : Магнолія-2006, 2023. – 380 с.
5. Myers, G. J. The Art Of Software Testing. / G. J. Myers, C. Sandler, T. Badgett. – N.Y. : John Wiley & Sons, Inc. 2011. – 256 p.
6. Губка, С. А. Основы тестирования информационных управляющих систем: учеб. пособие / С. А. Губка, А. С. Губка, П. Е. Ельцов. – Харьков : Нац. аэрокосм. ун-т им. Н. Е. Жуковского «Харьков. авиац. ин-т», 2016. – 67 с.
7. Губка, С. О. Особливості тестування мобільних додатків: навч. посіб. / С. О. Губка, О. С. Губка. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2020. – 80 с.

Навчальне видання

Губка Олексій Сергійович
Губка Сергій Олексійович

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Редактор А. Г. Литвин

Зв. план, 2024
Підписано до видання 16.09.2024
Ум. друк. арк. 3,1. Обл.-вид. арк. 3,5. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http:// www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001