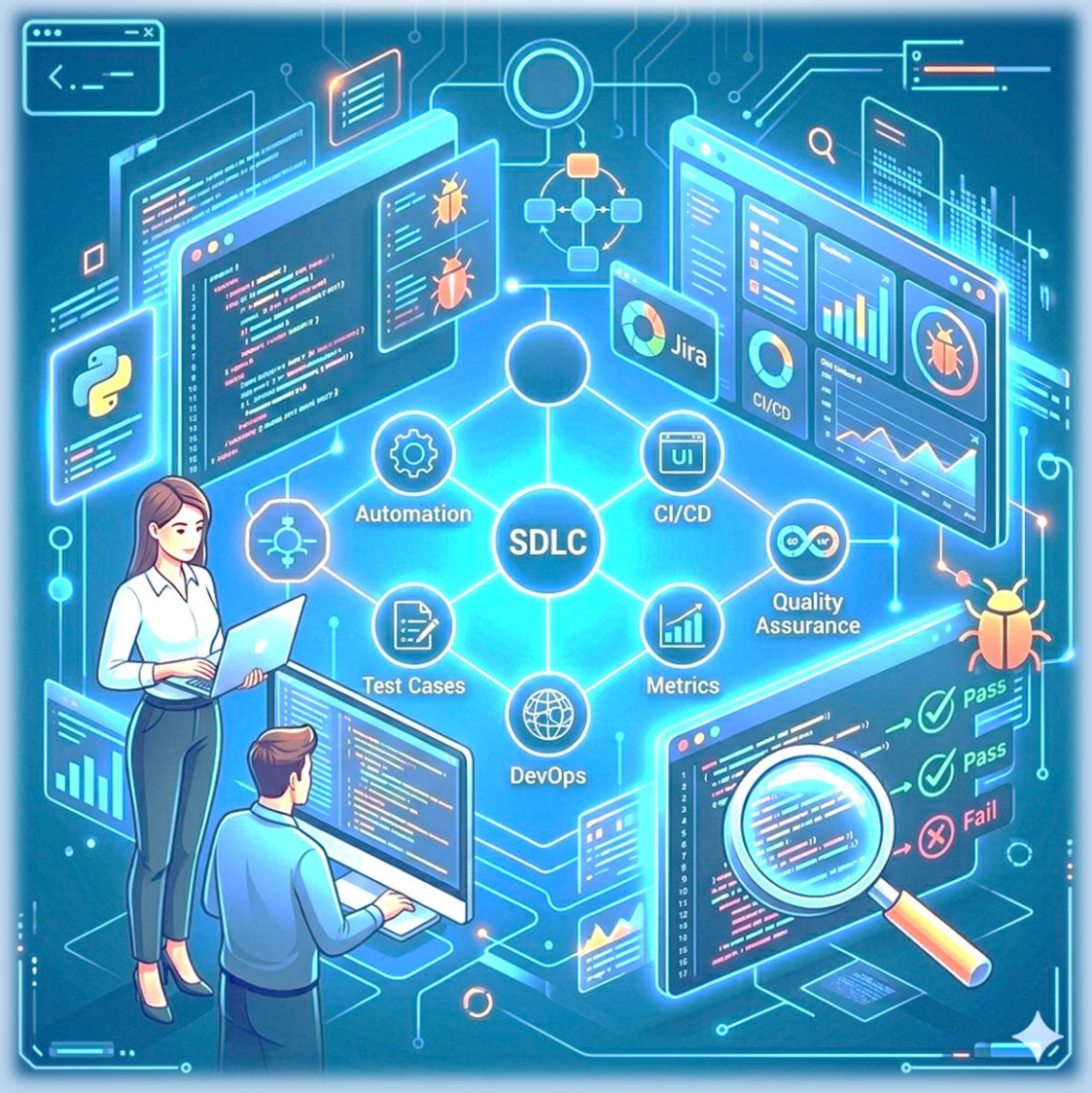
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МИКОЛАЇВСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ МЕНЕДЖМЕНТУ

Кафедра економічної кібернетики,
комп'ютерних наук та інформаційних технологій

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Конспект лекцій
для здобувачів першого (бакалаврського) рівня вищої освіти
ОПП «Комп'ютерні науки» спеціальності F3 (122) «Комп'ютерні науки»
денної форми здобуття вищої освіти



Миколаїв – 2025

УДК 004.05-048.24
Т36

Друкується за рішенням науково-методичної комісії факультету менеджменту Миколаївського національного аграрного університету (протокол №1 від 28 серпня 2025 року)

Укладачі:

О. Ю. Пархоменко — канд.фіз.-математ. наук, доцент, доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

С. І. Тищенко — канд.педагог.наук, доцент, доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

С. І. Ємельянов — PhD, старший викладач кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

О. О. Жебко — асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

О. Є. Богатєнкова — асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету

Рецензенти:

Ю. В. Грицук - канд. техн. наук, доцент кафедри загальної інженерної підготовки Донбаської національної академії будівництва і архітектури

О. С. Садовий - канд. техн. наук, доцент, завідувач кафедри агроінженерії Миколаївського національного аграрного університету

Тестування програмного забезпечення : конспект лекцій для здобувачів Т36 першого (бакалаврського) рівня вищої освіти ОПП «Комп'ютерні науки» спеціальності F3 (122) «Комп'ютерні науки» денної форми здобуття вищої освіти / уклад. О. Ю. Пархоменко, С. І. Тищенко, С. І. Ємельянов, О. О. Жебко, О. Є. Богатєнкова . Миколаїв : МНАУ, 2025. 94 с.

УДК 004.05-048.24

© Миколаївський національний аграрний університет, 2025

ЗМІСТ

ПЕРЕДМОВА

Сучасний світ неможливо уявити без програмного забезпечення. Воно оточує нас усюди: у смартфонах, банківських терміналах, автомобілях, медичному обладнанні, побутовій техніці та навіть у космічних апаратах. Ми щодня користуємося десятками програмних продуктів, навіть не замислюючись про це. Але що робить ці продукти надійними, безпечними та зручними? Що дозволяє мільйонам користувачів довіряти їм свої дані, фінанси, здоров'я? Відповідь проста – якісне тестування.

Навчальний посібник, який ви тримаєте в руках (або відкрили на екрані свого пристрою), створено для того, щоб допомогти вам опанувати одну з найважливіших і найзатребуваніших професій у галузі інформаційних технологій – професію тестувальника програмного забезпечення. Цей курс є результатом багаторічного досвіду викладання та практичної роботи в IT-індустрії, узагальненням кращих світових практик та адаптацією їх до потреб сучасного українського ринку праці.

Чому варто вивчати тестування програмного забезпечення? По-перше, це надзвичайно динамічна сфера, яка постійно розвивається разом із технологіями. По-друге, це професія, яка дає змогу швидко увійти в IT-галузь і будувати успішну кар'єру. По-третє, це діяльність, яка потребує не лише технічних знань, але й аналітичного мислення, уважності до деталей, комунікабельності та відповідальності. Тестувальник – це не просто "той, хто ламає програми", це адвокат користувача, гарант якості, людина, яка захищає репутацію компанії та довіру клієнтів.

Структура посібника побудована за логікою поступового занурення у професію. Перші лекції знайомлять із фундаментальними поняттями: що таке якість програмного забезпечення, чому тестування є критично важливим, які ролі існують у команді, як організований життєвий цикл розробки та тестування. Ви дізнаєтеся про різні методології розробки – від класичної каскадної моделі до сучасних гнучких підходів Agile, Scrum та Kanban, і зрозумієте, як вибір методології впливає на роботу тестувальника.

Особливу увагу приділено роботі з вимогами, адже саме чітко сформульовані вимоги є основою якісного тестування. Ви навчитеся класифікувати вимоги, визначати їхні властивості, працювати з документацією та виявляти помилки ще до того, як написано жодного рядка коду.

Значний блок матеріалу присвячено видам тестування та тестовій документації. Ви дізнаєтеся про різноманітні класифікації тестування, навчитеся створювати чек-листи, тест-кейси, ментальні карти, баг-репорти, а також розуміти, які властивості роблять ці документи дійсно корисними для команди. Окрема лекція присвячена мистецтву написання якісних звітів про дефекти – адже від того, наскільки добре описана проблема, залежить швидкість її виправлення.

Кульмінацією курсу є вивчення технік тест-дизайну. Саме ці техніки дозволяють перейти від інтуїтивного "тицяння по клавішах" до системного, науково обґрунтованого підходу. Еквівалентне розбиття, аналіз граничних

значень, таблиці прийняття рішень, діаграми переходів станів, парне тестування – ці інструменти допоможуть вам створювати ефективні тестові набори, які максимально виявляють дефекти при мінімальних витратах часу.

Завершується курс знайомством із тестуванням API – однією з найзатребуваніших навичок на сучасному ринку праці. Ви опануєте роботу з Postman, навчитеся створювати запити, аналізувати відповіді, автоматизувати тести та розуміти основи тестування продуктивності.

Кожна лекція супроводжується питаннями для обговорення, які допоможуть закріпити матеріал, розвинути критичне мислення та підготуватися до реальних робочих ситуацій. Адже тестування – це не просто набір технічних навичок, це спосіб мислення, який формується через постійні запитання, сумніви, перевірки та висновки.

Сподіваюся, що цей посібник стане для вас надійним путівником у світ тестування програмного забезпечення, допоможе здобути необхідні знання та навички, а можливо, й визначити подальший професійний шлях. Пам’ятайте: якісне програмне забезпечення починається з якісного тестування, а якісне тестування починається з вас.

Бажаємо успіхів у навчанні та цікавих відкриттів!

Лекція 1
Вступ до тестування програмного забезпечення

Вступ

Ласкаво просимо до курсу тестування програмного забезпечення. Сьогодні ми розпочнемо нашу подорож у надзвичайно важливий та багатогранний світ забезпечення якості в IT-індустрії. Можливо, у декого з вас тестування асоціюється виключно з пошуком помилок у чужому коді – процесом рутинним і майже механічним. Але насправді це набагато глибша дисципліна, яка є невід'ємною частиною життєвого циклу розробки, філософією створення якісного, надійного та безпечного продукту. На цій лекції ми закладемо фундамент для подальшого вивчення: поговоримо про те, чому тестування є критично важливим, як воно еволюціонувало, які професійні ролі існують у цій сфері та з чого почати свій шлях у цій захопливій професії.

Історичний контекст та важливість тестування

Перш ніж ми заглибимося в технічні деталі, варто зрозуміти, чому питання якості програмного забезпечення загалом і тестування зокрема набули такого вагомого значення. Історія тестування – це не просто хронологія подій, це історія усвідомлення ціни помилки.

На початку комп'ютерної ери, у 50-60-х роках, програмування було сферою обраних – математиків та інженерів, які працювали з громіздкими машинами. Програми були порівняно невеликими, а їхня вартість і вплив на навколишній світ – обмеженими. Тестування, як окремий процес, фактично було відсутнє, воно зливалося з процесом розробки або налагодження коду. Вважалося, що програміст, який написав код, і є найкращою людиною для перевірки його працездатності.

Однак у 70-80-х роках, з появою більш складних програмних продуктів, зростанням обчислювальної потужності та інтеграцією IT у бізнес і державне управління, ситуація кардинально змінилася. З'явилося поняття "програмний збій", який почав мати реальні фінансові та репутаційні наслідки. Саме в цей період тестування почало виокремлюватися в окрему діяльність, з'явилися перші методики та підходи. Інженери зрозуміли, що якість потрібно не контролювати постфактум, а планувати та забезпечувати на всіх етапах.

Нарешті, 90-ті роки та початок XXI століття стали епохою тотальної цифровізації. Інтернет, мобільні додатки, вбудовані системи в автомобілях та медичному обладнанні – програмне забезпечення проникло в кожну сферу нашого життя. Вартість помилки зросла експоненціально. Помилка в коді банківської системи може призвести до багатомільйонних збитків, у системі управління літаком – до людських жертв, а в мобільному додатку – до втрати

мільйонів користувачів. Сьогодні ми живемо у світі, де якість програмного забезпечення є не просто конкурентною перевагою, а часто – критичною умовою безпеки та функціонування суспільства. І саме тому тестування перетворилося на окрему, потужну індустрію з власними методологіями, інструментами та професійними стандартами.

Якість програмного забезпечення як багатовимірне поняття

Говорячи про тестування, ми неминуче говоримо про якість. Але що таке якість програмного забезпечення? Це не просто відсутність помилок. Це комплексна характеристика, яка включає в себе багато аспектів, і завдання тестувальника – оцінити продукт з усіх цих сторін.

Уявімо собі будь-який сучасний додаток. Перше, що спадає на думку – це його функціональність. Чи робить програма те, для чого вона призначена? Чи правильно виконуються обчислення, чи коректно зберігаються дані? Функціональне тестування – це база, перевірка відповідності заявленим вимогам.

Але запитайте себе: чи готові ви користуватися додатком, який постійно "падає" або зависає у найвідповідальніший момент? Це аспект надійності. Програмне забезпечення має працювати стабільно, без збоїв, протягом певного часу, відновлюючись після помилок. Тісно пов’язана з цим і продуктивність. Чи швидко завантажується сторінка? Чи не "гальмує" додаток, коли з ним одночасно працює тисяча користувачів? Продуктивність – це те, що відрізняє професійний продукт від аматорського виробу.

Навіть найстабільніший додаток буде приречений на провал, якщо ним важко користуватися. Зручність використання або юзабіліті – це те, наскільки інтуїтивно зрозумілим, зручним та естетично прийнятним є інтерфейс для кінцевого користувача. Чи легко знайти потрібну функцію? Чи зрозумілі підказки? Юзабіліті-тестування допомагає зробити продукт "дружнім" до людини.

В сучасному світі, де кібератаки стають дедалі витонченішими, безпека виходить на перший план. Чи захищені дані користувачів? Чи можна зламати систему або отримати несанкціонований доступ? Тестування безпеки – це окрема, дуже складна спеціалізація, яка покликана виявити вразливості до того, як це зроблять зловмисники.

Для розробників і бізнесу не менш важливими є такі характеристики, як підтримуваність (наскільки легко вносити зміни в код і додавати новий функціонал) та сумісність (чи правильно працює програма в різних браузерах, операційних системах, на різних пристроях). Як бачите, якість – це мозаїка, і завдання тестувальника – оцінити цілісність усієї картини.

Від тестування до забезпечення якості: QA та QC

У професійному середовищі часто можна почути два терміни: QA (Quality Assurance) та QC (Quality Control). Хоча в побуті їх іноді використовують як

синоніми, вони позначають різні рівні роботи з якістю. Важливо зрозуміти цю різницю, оскільки вона формує філософію нашої професії.

Quality Control, або контроль якості – це процес, спрямований на виявлення дефектів у готовому продукті. Це те, що найчастіше мають на увазі під "тестуванням" у вузькому сенсі. QC-інженер бере готовий продукт, виконує певні дії та шукає невідповідності. Його мета – знайти якомога більше помилок до релізу. Це реактивний підхід: продукт створили, ми його перевіряємо.

Quality Assurance, або забезпечення якості – це поняття значно ширше. Це проактивний підхід, спрямований на запобігання дефектам ще на етапі їх виникнення. QA – це не просто пошук помилок, це вибудовування процесів розробки таким чином, щоб помилки виникали якомога рідше. QA-інженер аналізує вимоги ще до написання коду, перевіряє їх на несуперечливість, бере участь в обговоренні архітектури, пропонує стандарти кодування та впроваджує практики, які підвищують якість роботи всієї команди. Якщо QC лікує хворобу, то QA займається профілактикою, створюючи умови, за яких захворіти неможливо.

Професійний портрет тестувальника та його виклики

Хто ж він, сучасний тестувальник? Якими навичками має володіти людина, яка вирішила присвятити себе цій справі? Часто існує стереотип, що це "нижча" інженерна позиція, з якої всі мріють "вирости" в розробники. Насправді, тестування вимагає унікального поєднання різноманітних навичок.

По-перше, це, безумовно, технічні навички. Тестувальник повинен розуміти архітектуру додатку, вміти працювати з базами даних, писати прості SQL-запити для перевірки даних, знати основи мережевих протоколів, таких як HTTP, та вміти користуватися інструментами розробника в браузері. Знання мов програмування стає обов’язковим, якщо ви плануєте займатися автоматизацією.

Але технічні знання – це лише половина успіху. Набагато важливішими є аналітичні та організаційні здібності. Тестувальник постійно аналізує вимоги, намагаючись передбачити, де саме може критися проблема. Він повинен вміти розбивати складну систему на частини, будувати гіпотези та планувати свою роботу так, щоб за обмежений час перевірити найважливіші сценарії.

Крім того, тестувальник – це завжди комунікатор. Він знаходиться на перетині інтересів замовника, розробника, менеджера та дизайнера. Його головне завдання – не просто знайти помилку, але й коректно її описати. Він має написати такий баг-репорт, щоб розробник одразу зрозумів, що сталося, за яких умов і як це відтворити. Якщо помилку не можна виправити прямо зараз, тестувальник має вміти аргументовано пояснити ризики менеджеру. Тому комунікаційні навички є ключовими.

Нарешті, це м’які навички. До них належать уважність до деталей, допитливість, наполегливість, вміння працювати в команді та, що дуже важливо, емпатія до користувача. Здатність поставити себе на місце людини,

яка вперше бачить ваш додаток, і передбачити, де вона може розгубитися – це ознака справжнього таланту.

Звісно, ця робота пов'язана з викликами. Тестувальник часто працює з неповною інформацією, змушений приймати рішення в умовах дефіциту часу, відстоювати свою думку перед розробниками та пояснювати бізнесу важливість, здавалося б, незначних дрібниць. Але саме це робить професію цікавою та динамічною.

Мануальне та автоматизоване тестування: два крила однієї птахи

Говорячи про інструменти та підходи, ми неминуче доходимо до фундаментального поділу тестування на мануальне та автоматизоване. Це не конкуруючі, а взаємодоповнюючі стратегії.

Мануальне тестування – це процес, коли тести виконує жива людина. Вона сідає за комп'ютер і, керуючись логікою та інтуїцією, натискає кнопки, вводить дані та спостерігає за реакцією системи. Цей підхід є незамінним там, де потрібен "людський фактор": в оцінці зручності інтерфейсу, в дослідницькому тестуванні, де неможливо передбачити всі сценарії заздалегідь. Коли ви перевіряєте нову функцію, яка тільки з'явилася в продукті, мануальне тестування часто є єдиним можливим варіантом. Воно дешевше на старті, гнучке і дозволяє швидко адаптуватися до змін. Однак, з часом, коли продукт зростає, ручне виконання тисяч повторюваних тестів стає дуже дорогим і повільним.

І тут на допомогу приходить автоматизоване тестування. Людина пише програму (скрипт), яка імітує дії користувача або перевіряє внутрішню логіку системи. Ці скрипти можна запускати будь-яку кількість разів, вони працюють швидко й не знають втоми. Саме автоматизація є королевою регресійного тестування – перевірки того, що нова версія програми не зламала старий, вже працюючий функціонал. Вона незамінна для навантажувального тестування, де треба імітувати роботу тисяч користувачів одночасно. Автоматизація вимагає значних інвестицій на початку (потрібно написати якісні скрипти та підтримувати їх), але в довгостроковій перспективі вона економить величезну кількість часу та ресурсів.

Вибір між мануальним та автоматизованим тестуванням залежить від конкретного завдання. Ідеальна стратегія – це їх розумне поєднання.

Роль тестування в життєвому циклі розробки (SDLC)

Дуже важливо зрозуміти, що тестування – це не окремий етап, який починається після того, як програмісти написали весь код. Сучасна філософія якості пронизує всі фази життєвого циклу розробки програмного забезпечення (SDLC).

Роль тестувальника починається ще на етапі аналізу вимог. Він разом із замовником та аналітиком вивчає, що саме має робити майбутня система, ставить запитання, шукає невідповідності, допомагає зробити вимоги чіткими

та тестованими. Адже змінити вимогу на папері набагато дешевше, ніж переписувати код через непорозуміння.

Потім настає етап планування. Тестувальник, спираючись на вимоги та розуміння ризиків, планує свою роботу: які тести будуть написані, яке обладнання для цього потрібне, скільки часу це займе.

Навіть під час написання коду, на етапі розробки, тестувальник може брати участь у код-рев'ю, аналізувати проміжні результати. Часто розробники пишуть модульні тести для власного коду, і QA-інженер може допомагати їм з цим.

І ось, коли код готовий, починається класичний етап виконання тестів. Мануальна перевірка, автоматизація, пошук багів, їх документування. Але на цьому робота не закінчується. Коли знайдені помилки виправлені, настає етап аналізу результатів та підготовки звіту про готовність продукту до релізу.

І навіть після випуску продукту на ринок, на етапі супроводу, тестувальник аналізує відгуки користувачів та допомагає тестувати виправлення критичних помилок у гарячому режимі. Таким чином, QA-інженер є повноцінним членом команди протягом усього життя продукту.

Спростування міфів про тестування

Насамкінець, варто розвіяти кілька поширених міфів, які існують навколо нашої професії. Часто можна почути, що тестування – це просто, і з нього всі починають, щоб потім стати програмістами. Це не так. Хороший тестувальник – це висококваліфікований фахівець, який володіє унікальним набором компетенцій.

Інший міф: мета тестувальника – знайти якомога більше помилок. Насправді, його головна мета – надати об'єктивну інформацію про якість продукту команді та бізнесу, щоб вони могли прийняти зважене рішення про готовність до релізу. Іноді знайти 100 дрібних помилок менш важливо, ніж переконатися, що працює основний, критичний для бізнесу сценарій.

Також існує хибна думка, що ретельне тестування може гарантувати повну відсутність помилок. Це неможливо. Будь-яка складна система містить потенційні дефекти. Завдання тестування – знизити ризики до прийнятного рівня, враховуючи наявні ресурси та час.

Підсумок

Отже, сьогодні ми зробили перший, але дуже важливий крок. Ми зрозуміли, що тестування – це не просто технічна дисципліна, а філософія якості, що пронизує весь процес створення програмного забезпечення. Ми побачили, наскільки багатогранним є поняття якості, які різноманітні навички потрібні тестувальнику, чим відрізняється мануальна перевірка від автоматизованої, і чому без цієї професії сучасний цифровий світ був би неможливим. Сподіваюся, цей вступ допоміг вам сформувати цілісне уявлення про майбутній курс. На наступних лекціях ми детально розглянемо конкретні

техніки, види тестування та інструменти, які допоможуть вам стати справжніми професіоналами.

Питання для обговорення

1. Чому, на вашу думку, на початку комп'ютерної ери тестування не вважалося окремою важливою діяльністю, а сьогодні є критично необхідним? Наведіть приклади сучасних програмних збоїв, які мали серйозні наслідки.
2. Як ви розумієте різницю між Quality Assurance (QA) та Quality Control (QC)? Чому важливо не просто шукати помилки, а й вибудовувати процеси так, щоб їх запобігати?
3. Чи може програмний продукт бути якісним, якщо він має ідеальний код та функціональність, але є вкрай незручним для користувача? Обґрунтуйте свою відповідь, спираючись на поняття якості ПЗ.
4. Які з атрибутів якості програмного забезпечення (функціональність, надійність, продуктивність, зручність, безпека, підтримуваність, сумісність) ви вважаєте найважливішими для мобільного банківського додатку? Чому?
5. Уявіть, що ви керівник проекту і вам потрібно найняти тестувальника. Які три особисті якості (soft skills) ви б шукали в кандидаті насамперед і чому?
6. Чи погоджуєтеся ви з твердженням, що автоматизоване тестування завжди краще за мануальне? У яких ситуаціях ручне тестування є незамінним, а в яких автоматизація є обов'язковою?
7. Як ви вважаєте, чи можна створити програмне забезпечення, в якому взагалі не буде помилок (багів)? Чому тестування не може гарантувати повну відсутність дефектів?
8. Проаналізуйте твердження: "Тестувальник – це адвокат користувача". Що це означає в контексті його роботи в команді розробки?
9. На вашу думку, чи повинен тестувальник вміти програмувати, навіть якщо він займається виключно мануальним тестуванням? Які переваги це може дати?
10. Чому в сучасних IT-компаніях тестувальники залучаються до роботи не після написання коду, а на набагато раніших етапах, наприклад, під час обговорення вимог?

Лекція 2
Життєвий цикл програмного забезпечення та цикл тестування

Вступ

На минулій лекції ми заклали фундамент нашого курсу, розібравшись із базовими поняттями якості програмного забезпечення, роллю тестувальника та важливістю тестування в сучасному IT-світі. Сьогодні ми зробимо наступний логічний крок і заглибимося в процеси, які структурують роботу над програмним продуктом. Ми розглянемо, як саме народжується, розвивається та підтримується програмне забезпечення протягом свого існування, і яке місце в цьому безперервному процесі займає тестування. Тема нашої лекції – життєвий цикл програмного забезпечення (SDLC) та нерозривно пов'язаний з ним життєвий цикл тестування (STLC). Ми також поговоримо про документацію, яка супроводжує роботу тестувальника, про різницю між повторним та регресійним тестуванням, і про те, як керувати ризиками, що неминуче виникають у будь-якому проекті.

Життєвий цикл програмного забезпечення (SDLC)

Перш ніж говорити про тестування, необхідно чітко розуміти контекст, у якому воно існує. Цим контекстом є життєвий цикл розробки програмного забезпечення, або SDLC (Software Development Life Cycle). За визначенням, SDLC – це структурований процес, який використовується індустрією для проектування, розробки та тестування високоякісного програмного забезпечення. Мета цього процесу – створити продукт, який максимально відповідає очікуванням замовника та кінцевих користувачів, укладаючись при цьому в бюджет і часові рамки.

Будь-яка методологія розробки, чи то класична каскадна модель, чи гнучкі (Agile) підходи, так чи інакше базується на загальних фазах SDLC. Традиційно виділяють кілька ключових етапів, які утворюють цикл: від виникнення ідеї до виведення продукту з експлуатації. Першим етапом є збір та аналіз вимог – найважливіша фаза, на якій визначається, що саме має робити майбутня система. На основі цих вимог створюється проект майбутньої архітектури, де описується структура програми, її компоненти та зв'язки між ними. Далі настає етап безпосередньої розробки, тобто написання коду. І тільки після цього починається тестування – перевірка створеного продукту на відповідність вимогам та виявлення дефектів. Коли продукт визнано готовим, він впроваджується в експлуатацію, де працює, приносячи користь користувачам, і водночас потребує підтримки, оновлень та виправлення помилок, які можуть бути виявлені вже в процесі реального використання.

Важливо розуміти, що SDLC – це не просто лінійна послідовність дій. Існують різні моделі, які по-різному організовують ці етапи. Класична каскадна

модель (Waterfall) передбачає чітке послідовне виконання фаз: перехід до наступної стадії можливий лише після повного завершення попередньої. Це зручно для проектів з дуже чіткими, незмінними вимогами, але робить її надзвичайно негнучкою. У сучасній же розробці найпопулярнішими стали гнучкі методології (Agile), такі як Scrum або Kanban. Вони базуються на ітеративному підході: робота ведеться короткими циклами (спринтами), після кожного з яких команда отримує працюючий інкремент продукту. Це дозволяє швидко реагувати на зміни вимог та отримувати зворотний зв'язок від замовника на ранніх етапах. Кожна модель має свої переваги та недоліки, і вибір конкретної залежить від специфіки проекту, команди та бізнес-контексту.

Життєвий цикл тестування програмного забезпечення (STLC)

Тепер, коли ми маємо загальне уявлення про SDLC, ми можемо детальніше розглянути, як саме тестування вбудовується в цей процес. Для цього існує поняття життєвого циклу тестування, або STLC (Software Testing Life Cycle). Це послідовність кроків, які виконує команда тестування для забезпечення якості продукту. Як і SDLC, STLC складається з окремих фаз, кожна з яких має чітко визначені вхідні та вихідні критерії.

Початковою фазою є аналіз вимог. На цьому етапі тестувальники детально вивчають документацію, спілкуються з аналітиками та замовниками, щоб зрозуміти, який функціонал потрібно перевіряти. Вони шукають невідповідності, неточності або прогалини у вимогах, адже чим раніше буде знайдено помилку, тим дешевше буде її виправлення. Після того, як вимоги зрозумілі, настає етап планування тестування. Тут визначаються стратегія тестування, оцінюються необхідні ресурси та час, створюється тест-план – ключовий документ, який описує, що саме і як буде тестуватися, хто це робитиме, які інструменти використовуватимуться, і як будуть звітувати результати.

Наступний крок – розробка тестових сценаріїв та створення тестової документації. На основі вимог тестувальники пишуть тест-кейси – детальні інструкції для перевірки конкретної функції, в яких зазначено кроки, очікуваний результат та передумови. Вони також готують тестові дані, які будуть використовуватися під час виконання тестів. І тільки після цього починається безпосереднє виконання тестів. Це фаза, на якій тестувальники вручну або за допомогою автоматизованих скриптів перевіряють роботу програми, порівнюють фактичні результати з очікуваними і, у разі розбіжностей, фіксують дефекти.

Після завершення виконання тестів настає етап аналізу результатів та підготовки звітності. Команда тестування збирає всі знайдені дефекти, аналізує їхню критичність, оцінює загальний стан якості продукту та готує звіт для команди та замовника. На основі цього звіту приймається рішення про готовність продукту до випуску. Але і після релізу робота не завершується: під час фази супроводу тестувальники беруть участь у перевірці виправлень,

аналізують відгуки користувачів і тестують продукт у реальних умовах експлуатації.

Тестова документація

Протягом усього життєвого циклу тестування створюється велика кількість документації. Це не просто бюрократія – це інструмент комунікації, планування та контролю якості. Основним документом є згаданий вище тест-план. Він визначає загальну стратегію і є своєрідною "конституцією" тестування для конкретного проекту.

На нижчому рівні деталізації знаходяться тест-кейси. Якісний тест-кейс – це мистецтво. Він має бути зрозумілим будь-якому члену команди, містити чіткі кроки та однозначний очікуваний результат. Група тест-кейсів, призначена для перевірки певної функції, може об’єднуватися в тестові набори (test suites). Крім того, тестувальники створюють чек-листи – більш вільну форму опису того, що потрібно перевірити, без детальних кроків. Чек-листи часто використовуються для швидких перевірок або в дослідницькому тестуванні.

І нарешті, найважливішим документом, що створюється за результатами виконання тестів, є баг-репорт (звіт про помилку). Це спосіб комунікації з розробниками. Хороший баг-репорт має містити унікальний ідентифікатор, короткий, але інформативний заголовок, детальний опис кроків для відтворення помилки, фактичний та очікуваний результати, а також інформацію про середовище (версію ПЗ, операційну систему, браузер тощо) та серйозність дефекту. Від того, наскільки якісно складено баг-репорт, залежить швидкість і правильність виправлення помилки.

Повторне та регресійне тестування

Коли розробник виправляє знайдену помилку, тестувальник отримує нову версію продукту на перевірку. Тут важливо розрізняти два різні процеси: повторне тестування (re-testing) та регресійне тестування (regression testing). Повторне тестування – це перевірка того, що конкретна помилка, яка була зафіксована раніше, дійсно виправлена. Тестувальник бере той самий тест-кейс, який раніше призводив до збою, і виконує його знову, щоб упевнитися, що тепер усе працює як очікувалося.

Однак, виправлення однієї помилки може ненавмисно зламати інший, раніше працюючий функціонал. Саме для виявлення таких непередбачуваних наслідків існує регресійне тестування. Це перевірка того, що нова версія програми не призвела до появи нових дефектів у вже існуючих, раніше протестованих частинах системи. Регресійне тестування зазвичай включає виконання великої кількості тестів, які підтверджують загальну стабільність продукту. Через його повторюваний характер, регресійне тестування є ідеальним кандидатом для автоматизації. Автоматизовані регресійні набори

можна запускати щоразу після внесення змін до коду, що дозволяє швидко виявляти побічні ефекти.

Приймальне тестування: фінальна перевірка перед релізом

Завершальним етапом тестування перед випуском продукту в світ є приймальне тестування (acceptance testing). Його головна мета – не знайти помилки (хоча вони теж можуть бути виявлені), а підтвердити, що продукт готовий до використання і відповідає очікуванням замовника та користувачів. Існує кілька різновидів приймального тестування.

Альфа-тестування проводиться всередині компанії-розробника, але із залученням внутрішніх співробітників, які не є членами команди розробки. Вони імітують роботу реальних користувачів у контрольованому середовищі. Бета-тестування – це наступний крок, коли продукт надається обмеженому колу реальних користувачів ззовні. Вони використовують його у своїх звичайних умовах і повідомляють про знайдені проблеми та враження. Це дозволяє отримати безцінний зворотний зв’язок перед масовим релізом.

Крім того, існує користувацьке приймальне тестування (UAT), де представники кінцевих користувачів перевіряють, чи зручно їм буде працювати з системою та чи виконує вона їхні щоденні завдання. Бізнес-приймальне тестування (BAT) фокусується на тому, чи досягає продукт бізнес-цілей, заради яких він створювався. Контрактне приймальне тестування (CAT) перевіряє відповідність умовам контракту, а правове (RAT) – відповідність законодавчим та регуляторним нормам. Нарешті, експлуатаційне приймальне тестування (OAT) оцінює, чи готовий продукт до стабільної роботи в продуктивному середовищі з точки зору адміністрування, резервного копіювання та відновлення.

Управління ризиками в тестуванні

Жоден проект не застрахований від несподіванок. Тому важливою складовою роботи тестувальника є управління ризиками. У контексті тестування ризики поділяються на дві великі категорії: ризики продукту та ризики проекту. Ризики продукту стосуються самого програмного забезпечення – це потенційні проблеми з його функціональністю, надійністю, безпекою або продуктивністю, які можуть вплинути на кінцевих користувачів. Ризики проекту пов’язані з процесом розробки та тестування: це можливі затримки через брак ресурсів, нечіткі вимоги, недостатню кваліфікацію команди або поломку інструментів.

Завдання тестувальника – не просто виявити ризик, але й оцінити його. Оцінка ризику зазвичай базується на двох факторах: ймовірності його виникнення та ступені потенційного впливу на проект. Ризики з високою ймовірністю та високим впливом потребують негайної уваги та розробки

заходів для їх пом’якшення. Наприклад, якщо ми розуміємо, що в новому модулі безпеки використовується складна криптографія, а в команді немає експерта з цього питання (високий ризик проекту), ми повинні заздалегідь запланувати час на додаткове навчання або залучення зовнішнього консультанта. Якщо ж ми виявили, що критично важлива функція авторизації має дуже складну логіку (високий ризик продукту), ми повинні приділити її тестуванню максимум уваги, створивши якомога більше різноманітних тест-кейсів. Таким чином, управління ризиками дозволяє нам ефективно розподіляти обмежені ресурси тестування, зосереджуючись на найважливіших і найнебезпечніших частинах системи.

Висновок

Підсумовуючи сьогоднішню лекцію, ми можемо сказати, що тестування – це не хаотичний процес, а чітко структурована діяльність, яка тісно інтегрована в життєвий цикл розробки програмного забезпечення. Розуміння фаз SDLC та STLC, вміння працювати з тестовою документацією, чітко розрізняти повторне і регресійне тестування, знати види приймального тестування та володіти основами управління ризиками – усе це складає фундамент професійних компетенцій тестувальника. Ці знання дозволяють йому не просто "ламати" програму, а бути повноцінним учасником команди, який свідомо будує процес забезпечення якості, прогнозує проблеми та допомагає створити дійсно надійний і успішний продукт. На наступних заняттях ми розглянемо конкретні техніки тест-дизайну та інші практичні інструменти, які допоможуть вам втілити ці знання в життя.

Питання для обговорення

1. Чому етап аналізу вимог вважається одним із найважливіших у життєвому циклі розробки ПЗ? Які наслідки може мати помилка, допущена на цьому етапі, і чому її виправлення на пізніх стадіях коштує найбільше?
2. Поясніть різницю між повторним тестуванням (re-testing) та регресійним тестуванням (regression testing). Чому для регресійного тестування особливо важлива автоматизація?
3. Які ризики для проекту несе в собі ситуація, коли тестувальник знаходить багато дрібних, неприорітетних помилок, але не встигає перевірити критичний бізнес-сценарій? Як цього уникнути?
4. Порівняйте альфа-тестування та бета-тестування. У чому полягає головна цінність бета-тестування для розробників, і яку інформацію воно дозволяє отримати, недоступну при інших видах тестування?
5. Чим, на вашу думку, відрізняється ризик продукту від ризику проекту? Наведіть приклад кожного з цих видів ризиків для розробки нового мобільного додатку.
6. Як ви розумієте поняття "критерії входу" та "критерії виходу" в контексті фази тестування? Чому важливо їх чітко визначити перед початком та після завершення тестування?

7. Чи можна вважати юзабіліті-тестування (тестування зручності) частиною приймального тестування? Обґрунтуйте свою думку.

8. Уявіть, що ви тестувальник і отримали завдання перевірити нову функцію. Яку інформацію ви шукатимете в тест-плані, а яку – в тест-кейсах?

9. Чому важливо вести історію змін вимог до проекту? Як це допомагає тестувальникам і розробникам уникати плутанини?

10. Як ви вважаєте, чи можна керувати ризиками проекту так, щоб повністю їх уникнути? Якщо ні, то яка справжня мета управління ризиками?

Лекція 3
Методології розробки програмного забезпечення та їх вплив на тестування

Вступ

На попередніх заняттях ми детально розглянули життєвий цикл програмного забезпечення (SDLC) і з'ясували, що тестування є невід'ємною його частиною. Однак SDLC – це лише загальна схема. Те, як саме організована робота над проектом, як взаємодіють члени команди, коли починається тестування і як часто воно проводиться, визначається обраною методологією розробки. Методологія – це набір принципів, практик і правил, які регламентують процес створення програмного продукту. Вибір методології має колосальний вплив на роботу тестувальника, його обов'язки, інструменти та місце в команді. Сьогодні ми розглянемо основні методології розробки, порівняємо класичний каскадний підхід із сучасними гнучкими методологіями, зокрема Agile, Scrum та Kanban, і проаналізуємо, як кожна з них змінює підхід до тестування.

Класичні моделі розробки: Waterfall та V-model

Історично першою широко вживаною моделлю стала водоспадна модель, або Waterfall. Вона передбачає суворо послідовне виконання етапів проекту: спочатку збираються та затверджуються всі вимоги, потім створюється детальний дизайн системи, після чого починається етап кодування, і лише після його повного завершення настає черга тестування. Уявіть собі справжній водоспад, де вода перетікає з одного рівня на інший, і повернутися назад неможливо. Такий підхід має свої переваги: він простий для розуміння, дозволяє ретельно документувати кожен крок і зручний для проектів з дуже стабільними, незмінними вимогами, наприклад, для розробки систем керування літаками чи медичного обладнання, де будь-які зміни в процесі можуть бути вкрай дорогими або навіть небезпечними.

Однак головний недолік Waterfall стає очевидним, коли ми говоримо про тестування. Оскільки тестування виконується лише в самому кінці, всі помилки, закладені на ранніх етапах – у вимогах чи архітектурі – виявляються надто пізно. Виправлення таких фундаментальних дефектів на фінальній стадії коштує величезних зусиль, часу та грошей. Крім того, замовник бачить працюючий продукт тільки в кінці проекту, і якщо його очікування розійшлися з реальністю, змінити щось буває практично неможливо

Спроба подолати цей недолік привела до появи V-подібної моделі (V-model). Вона отримала свою назву через графічне зображення, де етапи розробки (аналіз вимог, проектування, кодування) розташовані зліва, а відповідні їм етапи тестування – справа, утворюючи літеру V. Головна ідея

полягає в тому, що тестування планується паралельно з відповідними фазами розробки. Наприклад, коли аналітики формують вимоги до системи, тестувальники вже починають розробляти плани приймального тестування. Коли створюється архітектура високого рівня, закладаються основи для інтеграційного тестування. Це дозволяє знаходити невідповідності та потенційні дефекти набагато раніше, ніж у класичному Waterfall. Проте V-модель все ще залишається відносно жорсткою: вимоги мають бути чітко визначені на початку, і повернення до попередніх етапів ускладнене.

Ітераційні та інкрементальні моделі

Наступним кроком еволюції стали ітераційні та інкрементальні моделі. Їхня суть полягає в тому, що проект розбивається на послідовні цикли – ітерації, кожна з яких триває від кількох тижнів до кількох місяців. На кожній ітерації створюється працюючий інкремент продукту – невелика, але повноцінна частина функціоналу. Цей інкремент проходить повний цикл: від аналізу вимог до тестування та інтеграції. Після завершення ітерації команда отримує зворотний зв’язок від замовника або користувачів, що дозволяє скоригувати плани на наступну ітерацію.

Такий підхід має низку незаперечних переваг. Ризики знижуються, оскільки проблеми виявляються на ранніх етапах. Замовник бачить реальний прогрес і може впливати на розвиток продукту. Тестування виконується постійно, що підвищує якість кінцевого продукту. Саме на основі ітераційної моделі виникли гнучкі методології, які сьогодні домінують у світі розробки програмного забезпечення.

Agile: маніфест і принципи

На початку 2000-х років група досвідчених розробників сформулювала Agile Manifesto – маніфест гнучкої розробки програмного забезпечення. Він проголошує чотири ключові цінності: люди та взаємодія важливіші за процеси та інструменти; працюючий продукт важливіший за вичерпну документацію; співпраця із замовником важливіша за узгодження умов контракту; готовність до змін важливіша за дотримання первісного плану. Це не означає, що документація чи планування не потрібні, але в гнучкому світі головним пріоритетом є створення цінності для користувача та швидка адаптація до змін.

На основі цих цінностей було сформульовано дванадцять принципів Agile, серед яких – задоволення замовника через ранню та безперервну поставку цінного програмного забезпечення, зміна вимог вітається навіть на пізніх стадіях розробки, працюючий продукт випускається якомога частіше (з періодичністю від кількох тижнів до кількох місяців), тісна щоденна взаємодія бізнесу та розробників, мотивація та довіра до команди, безпосереднє спілкування як найефективніший спосіб передачі інформації, головним показником прогресу є працюючий продукт, постійна увага до технічної

досконалості, простота як мистецтво не робити зайвої роботи, самоорганізація команд, регулярна рефлексія та коригування своєї роботи.

Scrum: ролі, артефакти, події

Найпопулярнішим фреймворком для реалізації принципів Agile є Scrum. Він базується на трьох стовпах: прозорість (усі аспекти процесу мають бути видимі для тих, хто відповідає за результат), перевірка (учасники регулярно перевіряють артефакти та прогрес для виявлення відхилень) та адаптація (у разі виявлення відхилень процес або матеріали коригуються якомога швидше).

Scrum визначає чіткі ролі в команді. Product Owner – це власник продукту, який відповідає за максимізацію цінності, що створюється командою. Він формує та пріоритезує вимоги в списку, який називається Product Backlog. Scrum Master – це фасилітатор, який допомагає команді дотримуватися принципів Scrum, усуває перешкоди та навчає процесу. Development Team – це група професіоналів (розробники, тестувальники, аналітики), які безпосередньо створюють інкремент продукту. Важливо, що команда є крос-функціональною, тобто володіє всіма необхідними навичками для виконання роботи, і самоорганізованою – вона сама вирішує, як найкраще досягти поставленої мети.

Основними артефактами Scrum є Product Backlog (упорядкований список усього, що може знадобитися в продукті), Sprint Backlog (набір елементів з Product Backlog, обраних для виконання в поточному спринті, плюс план їх реалізації) та Increment (сума всіх елементів Product Backlog, виконаних за поточний та всі попередні спринти, яка є працюючою версією продукту).

Процес Scrum організований у вигляді циклів – спринтів, фіксованої тривалості (зазвичай 1-4 тижні). Кожен спринт містить набір подій. Sprint Planning – зустріч на початку спринту, де команда визначає, яка частина роботи буде виконана і як саме. Щодня проводиться Daily Scrum (щоденна 15-хвилинна зустріч), на якій команда синхронізує дії та планує роботу на найближчі 24 години. Наприкінці спринту відбувається Sprint Review – огляд виконаної роботи, де команда демонструє інкремент замовнику та отримує зворотний зв’язок. Завершується спринт Sprint Retrospective – ретроспективою, на якій команда аналізує свій процес роботи, щоб знайти способи покращення.

Тестування в Scrum

У Scrum тестування перестає бути окремою фазою і стає невід’ємною частиною кожного спринту. Тестувальник є повноцінним членом команди розробки. Його робота починається ще на етапі планування спринту: він допомагає оцінити зусилля, необхідні для тестування конкретних історій, аналізує acceptance criteria (критерії приймання) на повноту та тестованість. Протягом спринту, в міру того, як розробники пишуть код, тестувальник створює та уточнює тест-кейси, готує тестові дані, виконує дослідницьке тестування нової функціональності, а також займається автоматизацією

регресійних тестів. Коли функція готова, вона одразу ж тестується, що дозволяє знаходити та виправляти дефекти в межах того самого спринту. Наприкінці спринту команда демонструє повністю протестований і готовий до випуску інкремент продукту. Ретроспектива дає змогу тестувальнику порушити питання щодо покращення процесів, які впливають на якість.

Kanban: безперервний потік

Іншим популярним гнучким підходом є Kanban. На відміну від Scrum, він не має фіксованих ітерацій. Робота організована як безперервний потік завдань, які візуалізуються на Kanban-дошці з колонками (наприклад, "Треба зробити", "В роботі", "На тестуванні", "Готово"). Головний принцип Kanban – обмеження кількості завдань, що одночасно знаходяться в роботі (Work in Progress, WIP limits). Це дозволяє уникнути перевантаження команди та зосередитися на завершенні розпочатих завдань, а не на постійному старті нових.

У Kanban тестування також є безперервним. Як тільки розробник завершує роботу над функцією, вона переміщується в колонку "На тестуванні". Тестувальник, якщо він вільний (тобто не перевищено WIP-ліміт для тестування), бере це завдання в роботу, перевіряє його і, у разі успіху, переміщує далі. Залежно від організації команди, тестувальники можуть бути виділеними фахівцями або ж тестування виконується спільно всією командою. Автоматизація відіграє ключову роль для підтримки темпу безперервної поставки.

Порівняння Agile та Waterfall з точки зору тестування

Підсумовуючи, можна виділити кардинальні відмінності в тому, як організовано тестування в класичному Waterfall та в Agile. У Waterfall тестування – це окрема фаза в кінці проекту, що призводить до пізнього виявлення дефектів і високої вартості їх виправлення. В Agile тестування інтегроване в кожну ітерацію, що дозволяє знаходити помилки на ранніх стадіях і швидко їх виправляти. У Waterfall вимоги фіксуються на початку, і будь-які зміни сприймаються як проблема. В Agile зміни вітаються, і тестувальник готовий адаптувати свої тести до нових вимог. Документація в Waterfall є всеосяжною і часто занадто детальною, тоді як в Agile команда віддає перевагу працюючому продукту, а документація створюється лише в тій мірі, в якій вона дійсно необхідна.

Роль тестувальника в Agile-команді

У гнучкому середовищі тестувальник перестає бути просто "контролером якості" на фініші. Він стає активним учасником команди, який впливає на якість на всіх етапах. Він бере участь у плануванні, допомагаючи уточнювати acceptance criteria, щоб вони були чіткими та тестованими. Він тісно співпрацює з розробниками: обговорює особливості реалізації, радить, які перевірки варто

автоматизувати, допомагає відтворювати складні помилки. Завдяки ранній інтеграції тестування він може запобігти багато дефектів ще до того, як код буде написано. Автоматизація стає його найкращим другом, оскільки дозволяє швидко перевіряти регресію після кожної зміни. Він також використовує дослідницьке тестування для пошуку неочевидних проблем у новому функціоналі. Він забезпечує швидкий зворотний зв'язок команді, щоб розробники могли оперативно виправляти помилки, не відкладаючи їх на потім. І нарешті, на ретроспективах він ініціює обговорення покращень процесів, які підвищать якість продукту в майбутньому.

Документація в Agile та концепція MVP

Важливо розуміти, що Agile не відкидає документацію, але вимагає від неї бути "легкою" та актуальною. Замість багатосторінкових специфікацій команда використовує user stories (користувацькі історії) з чіткими acceptance criteria. Будь-яка документація, що створюється, має додавати цінність. Якщо документ не допомагає команді рухатися вперед, від нього варто відмовитися.

З гнучкою філософією тісно пов'язане поняття MVP – Minimum Viable Product, мінімально життєздатного продукту. Це версія продукту, яка містить лише найнеобхідніший функціонал, достатній для того, щоб вийти на ринок і отримати зворотний зв'язок від перших користувачів. Такий підхід дозволяє мінімізувати витрати на початковому етапі та переконатися, що команда рухається в правильному напрямку. Тестування MVP має бути особливо ретельним, адже від першого враження користувачів залежить успіх усього проекту.

Вибір методології: проблеми та реальність

На практиці вибір методології не завжди є однозначним. Команди часто адаптують і комбінують різні підходи, створюючи гібридні моделі. Важливо розуміти контекст проекту: його розмір, складність, вимоги до безпеки, очікування замовника, корпоративну культуру. Наприклад, для великого державного проекту з жорсткими регуляторними вимогами може бути обрана V-модель або навіть Waterfall, тоді як для інноваційного стартапу, який швидко змінюється, найкращим вибором стане Scrum або Kanban.

Висновок

Методологія розробки визначає не лише те, як пишеться код, але й те, як будується процес забезпечення якості. Від вибору методології залежить, коли тестувальник долучається до роботи, як часто він виконує тести, які інструменти використовує, і як він взаємодіє з іншими членами команди. Розуміння різних методологій – від класичного Waterfall до гнучких Agile-фреймворків – є обов'язковою складовою професійної компетентності сучасного тестувальника. Це дозволяє йому ефективно працювати в будь-якому

оточенні, швидко адаптуватися і робити максимальний внесок у створення якісного програмного продукту. На наступних лекціях ми перейдемо до конкретних технік тест-дизайну, які допоможуть вам створювати ефективні тестові набори незалежно від обраної методології.

Питання для обговорення

1. Уявіть, що ви працюєте над проектом зі створення програмного забезпечення для космічного корабля. Яку модель розробки (Waterfall, V-model, Agile) ви б обрали і чому? Які ризики несе використання Agile у такому проекті?
2. У чому, на вашу думку, полягає головна перевага Agile-методологій перед каскадною моделлю з точки зору тестування? Чому раннє виявлення дефектів в Agile є більш імовірним?
3. Проаналізуйте основні цінності Agile-маніфесту. Як принцип "Працюючий продукт важливіший за вичерпну документацію" впливає на роботу тестувальника та обсяг тестової документації?
4. Порівняйте роль тестувальника в Scrum та в Kanban. У якій з методологій, на вашу думку, легше планувати свою роботу, а в якій – важче? Чому?
5. Поясніть важливість щоденних Scrum-зустрічей (Daily Scrum) для тестувальника. Яку інформацію він може там отримати, а яку – надати команді?
6. Чи може проект, який розробляється за Waterfall, бути успішним у сучасному світі? Наведіть приклади проектів або сфер, де цей підхід досі є виправданим.
7. Як змінюється підхід до документування вимог при переході від Waterfall до Agile? Чи зникає документація взагалі, чи вона просто набуває іншої форми?
8. Що таке MVP (Minimum Viable Product) і яка роль тестування на етапі його створення? Чи повинні тести для MVP бути особливими?
9. Як ви вважаєте, хто несе відповідальність за якість продукту в Agile-команді: тестувальник, розробник чи скрам-майстер? Чи є це індивідуальною чи командною відповідальністю?
10. Чи можна поєднувати практики Scrum та Kanban в одному проекті? Якби вам довелося це робити, які елементи з кожної методології ви б узяли?

Лекція 4
Вимоги до програмного забезпечення та робота з ними

Вступ

На попередніх лекціях ми детально розглянули життєвий цикл розробки програмного забезпечення, методології, за якими працюють сучасні IT-команди, та місце тестування в цих процесах. Сьогодні ми підійшли до фундаментальної теми, яка є основою для всього, що відбувається далі в проекті – до вимог програмного забезпечення. Саме вимоги визначають, що саме має бути створено, які функції виконуватиме система, якими характеристиками вона володітиме. Для тестувальника вимоги є головним орієнтиром, своєрідною "конституцією", на відповідність якій перевіряється продукт. Якість вимог безпосередньо впливає на якість кінцевого продукту, терміни розробки та бюджет проекту. Сьогодні ми навчимося класифікувати вимоги, визначати їхні властивості, працювати з документацією та тестувати самі вимоги ще до того, як написано жодного рядка коду.

Чому вимоги настільки важливі?

Уявіть собі, що ви замовили архітектору проєкт будинку, але не змогли чітко пояснити, скільки поверхів вам потрібно, з яких матеріалів має бути побудований будинок і скільки кімнат ви хочете. Архітектор намалює щось на свій розсуд, будівельники почнуть працювати за цим кресленням, а за півроку ви побачите споруду, яка зовсім не відповідає вашим мріям. Переробляти все буде довго, дорого і практично неможливо. Та сама логіка працює і в розробці програмного забезпечення. Вимоги – це той самий архітектурний проєкт, який дає відповіді на ключові питання: що саме ми будуємо, для кого, навіщо і якими характеристиками цей продукт має володіти.

Проблеми з вимогами є однією з найпоширеніших причин провалу IT-проектів. Якщо вимоги нечіткі, суперечливі або неповні, розробники створюють не те, що потрібно замовнику. Якщо вимоги змінюються на пізніх етапах без належного контролю, це ламає архітектуру, призводить до появи нових помилок і зриває терміни. Якщо вимоги неможливо перевірити, тестувальники не можуть дати однозначну відповідь, чи правильно працює система. Тому робота з вимогами – це не просто бюрократична формальність, а критично важливий процес, від якого залежить успіх усього проекту.

Типи документації у розробці програмного забезпечення

Перш ніж заглиблюватися у класифікацію самих вимог, варто розібратися, в яких документах вони можуть бути зафіксовані. У світі розробки програмного забезпечення існує кілька рівнів документації. На найвищому рівні знаходиться

бізнес-документація, яка описує, навіщо взагалі створюється продукт, які бізнес-цілі він має досягти, яка його цільова аудиторія. Це може бути концепція продукту або бізнес-план.

Далі йде документація, що описує вимоги до системи. Це можуть бути технічні завдання, специфікації вимог до програмного забезпечення (SRS – Software Requirements Specification) або, в гнучких методологіях, user stories (користувацькі історії) з acceptance criteria (критеріями приймання). Цей рівень документації безпосередньо цікавить тестувальників, оскільки саме тут описано, що саме має робити система.

Нарешті, існує технічна документація, яка описує, як саме система побудована зсередини: архітектурні схеми, дизайн бази даних, API-специфікації. Ця документація важлива для розробників та інтеграційних тестувальників, які перевіряють взаємодію компонентів системи.

Джерела та шляхи виявлення вимог

Звідки ж беруться вимоги? Основним джерелом, безумовно, є замовник або власник продукту. Саме він має бачення того, яку проблему має вирішувати продукт. Однак замовник рідко може сформулювати всі вимоги одразу і в ідеальному вигляді. Тому вимоги збираються та уточнюються в процесі комунікації. Це можуть бути інтерв’ю, анкетування, мозкові штурми, аналіз існуючих бізнес-процесів.

Іншим важливим джерелом є кінцеві користувачі. Саме вони будуть щодня працювати з системою, і їхні потреби, больові точки та побажання мають бути враховані. Для цього проводяться опитування, фокус-групи, аналіз відгуків на попередні версії продукту. Крім того, вимоги можуть виникати з аналізу конкурентів, вивчення ринку, законодавчих та регуляторних норм, технічних обмежень та стандартів, прийнятих в індустрії.

Класифікація вимог

Для того щоб ефективно працювати з вимогами, їх прийнято класифікувати за різними ознаками. Найважливішим є поділ на функціональні та нефункціональні вимоги.

Функціональні вимоги описують, що саме система має робити. Вони визначають конкретну поведінку, функції, які має виконувати програмне забезпечення. Наприклад, "система повинна дозволяти користувачеві авторизуватися за допомогою логіна та пароля" або "при натисканні кнопки ‘Зберегти’ дані мають бути записані в базу даних". Функціональні вимоги відповідають на питання "що?".

Нефункціональні вимоги описують, якою має бути система, тобто визначають обмеження та атрибути якості. Вони стосуються продуктивності (система має обробляти 1000 запитів за секунду), надійності (система має працювати без збоїв 99.9% часу), безпеки (доступ до даних має бути захищеним паролем), зручності використання (інтерфейс має бути інтуїтивно зрозумілим),

підтримуваності (код має бути написаний з дотриманням стандартів) тощо. Нефункціональні вимоги відповідають на питання "як добре?".

Крім того, вимоги можна класифікувати за рівнями: бізнес-вимоги (цілі організації), вимоги користувачів (завдання, які мають виконувати користувачі), функціональні вимоги (власне функції системи). Також важливою є класифікація за пріоритетом, яка допомагає команді розуміти, що потрібно зробити в першу чергу, а що можна відкласти.

Атрибути якості вимог

Хороші вимоги мають володіти певними атрибутами, які роблять їх корисними та придатними для використання. Перш за все, кожна вимога має бути однозначною – тобто її можна інтерпретувати лише одним способом, без двозначностей. Уявіть собі вимогу "інтерфейс має бути зручним". Що означає "зручний"? Для різних людей це можуть бути різні речі. Натомість вимога має бути сформульована чітко: "кнопка входу має бути розташована у верхньому правому куті екрана".

Вимога має бути повною, тобто містити всю необхідну інформацію без потреби звертатися до інших джерел. Вона має бути несуперечливою – не суперечити іншим вимогам. Наприклад, якщо одна вимога говорить, що пароль має містити не менше 6 символів, а інша – що не більше 4, це суперечність. Вимога має бути здійсненною – технічно можливою та реалістичною в рамках бюджету та термінів проекту.

Найважливішою для тестувальника є властивість тестованості (verifiability). Вимога має бути сформульована так, щоб можна було однозначно визначити, виконана вона чи ні. Вимога "система має працювати швидко" є нетекстованою. Вимога "час відповіді системи не має перевищувати 2 секунд при 100 одночасних користувачах" є тестованою. Саме такі вимоги дозволяють тестувальникам створювати чіткі тест-кейси та давати однозначну відповідь про якість продукту.

Життєвий цикл вимог

Вимоги, як і сам програмний продукт, проходять певний життєвий цикл. Він починається з ініціації, тобто збору вихідних вимог від замовника та інших стейкхолдерів. На цьому етапі важливо зафіксувати якомога більше інформації, не відкидаючи навіть ідеї, які здаються сумнівними.

Далі настає етап уточнення та аналізу. Зібрані вимоги деталізуються, перевіряються на суперечливість, оцінюється їх реалістичність. Тут же відбувається пріоритезація – визначення, що має бути зроблено в першу чергу, а що можна відкласти. Після аналізу вимоги документуються – фіксуються у відповідному форматі, чи то детальна специфікація, чи то набір user stories.

Наступним етапом є верифікація та узгодження. Вимоги перевіряються всіма зацікавленими сторонами – замовником, розробниками, тестувальниками

– на предмет повноти, однозначності та здійсненності. Після узгодження вимоги затверджуються і стають базою для подальшої роботи.

Однак на цьому життєвий цикл не закінчується. Протягом проекту вимоги можуть змінюватися – з'являються нові потреби бізнесу, змінюється законодавство, виявляються технічні обмеження. Тому існує етап управління змінами. Будь-яка зміна вимоги має проходити через чіткий процес: ініціація зміни, оцінка її впливу на проект (терміни, бюджет, архітектуру), узгодження та, власне, внесення зміни до документації з одночасним інформуванням усієї команди.

Матриця простежуваності вимог (RTM)

Для того щоб відстежувати зв'язок між вимогами та іншими артефактами проекту – дизайном, кодом, тестами – використовується матриця простежуваності вимог (Requirements Traceability Matrix, RTM). Це документ, зазвичай у формі таблиці, який показує, як кожна вимога реалізована в продукті та як вона перевіряється тестами.

Для тестувальника RTM є надзвичайно корисним інструментом. Вона дозволяє переконатися, що кожна вимога покрита тестами (немає "сірих зон", які ніхто не перевіряє). І навпаки, вона допомагає зрозуміти, які тести перевіряють які саме вимоги, що важливо при змінах – якщо змінюється певна вимога, RTM одразу показує, які тести потрібно оновити та перезапустити. Матриця також допомагає аналізувати вплив змін: якщо вимога видаляється або модифікується, легко визначити, які частини системи це зачепить.

Управління змінами та пріоритезація вимог

У сучасній розробці, особливо в Agile-середовищі, зміни вимог є нормою, а не винятком. Тому важливо мати чіткий процес управління змінами (change management process). Він має включати кілька обов'язкових кроків. Спочатку хтось ініціює зміну – це може бути замовник, менеджер продукту або навіть член команди, який побачив можливість покращення. Далі зміна аналізується: які зусилля потрібні для її реалізації, як вона впливає на інші частини системи, які ризики з нею пов'язані. Після аналізу приймається рішення – впроваджувати зміну чи відхилити її. Якщо зміна схвалена, вона вноситься до списку вимог з відповідним пріоритетом.

Пріоритезація вимог – це окреме мистецтво. Існує багато технік, але найпоширенішою є MoSCoW: Must have (обов'язкові вимоги, без яких продукт не має сенсу), Should have (бажані вимоги, які варто реалізувати, якщо дозволяють ресурси), Could have (можливі вимоги, які реалізуються, якщо залишається час) та Won't have (вимоги, від яких вирішено відмовитися в поточній версії). Пріоритезація допомагає команді фокусуватися на найважливішому і не розпорошувати зусилля на другорядні функції, особливо коли час та ресурси обмежені.

Формати документації вимог

Залежно від методології та масштабу проекту, вимоги можуть документуватися в різних форматах. У класичних, документально-орієнтованих підходах використовуються детальні специфікації вимог (SRS) – багатосторінкові документи, що містять вичерпний опис усіх функцій та обмежень системи.

У гнучких методологіях популярним форматом є user stories (користувацькі історії). Типова user story має вигляд: "Як [роль користувача], я хочу [виконати певну дію], щоб [досягти певної цілі]". Наприклад: "Як зареєстрований користувач, я хочу мати можливість скинути пароль, якщо я його забув, щоб відновити доступ до свого акаунта". User stories зазвичай доповнюються acceptance criteria – чіткими умовами, за яких історія вважатиметься виконаною. Саме acceptance criteria є основою для тестування в Agile.

Техніки тестування вимог

Тестувальник має долучатися до роботи з вимогами якомога раніше. Існують різні техніки, які допомагають виявити проблеми у вимогах ще до початку розробки. Одна з найефективніших – це рецензування вимог (requirements review). Команда (аналітик, розробник, тестувальник, замовник) збирається разом і колективно перевіряє вимоги на повноту, однозначність, несуперечливість та тестованість. Це дозволяє знайти багато проблем на ранньому етапі, коли їх виправлення коштує мінімальних зусиль.

Іншою корисною технікою є прототипування. Створення швидких макетів інтерфейсу або навіть клікабельних прототипів дозволяє замовнику та користувачам "помацати" майбутню систему і скоригувати вимоги до того, як почнеться розробка. Також використовується техніка тестування acceptance criteria: тестувальник намагається "зламати" acceptance criteria ще до написання коду, продумуючи негативні сценарії та граничні випадки. Це допомагає зробити acceptance criteria більш повними та стійкими до помилок.

Типові помилки при роботі з вимогами

На жаль, у реальних проектах помилки при роботі з вимогами трапляються дуже часто. Найпоширенішою є нечіткість та двозначність формулювань, про яку ми вже говорили. Інша поширена проблема – неповнота вимог, коли забувають описати важливі сценарії, особливо обробку помилкових ситуацій. Наприклад, вимога описує, що має статися, коли користувач вводить правильний пароль, але нічого не каже про те, що робити, якщо пароль неправильний.

Часто вимоги бувають суперечливими – одна частина документа говорить одне, а інша – прямо протилежне. Інколи вимоги є принципово нетекстованими, тобто неможливо визначити, виконані вони чи ні. Нарешті, вимоги можуть бути нереалістичними або технічно неможливими, що призводить до зриву термінів та марно витрачених зусиль. Завдання тестувальника – допомогти команді виявити ці помилки якомога раніше.

Роль тестувальника у роботі з вимогами

Підсумовуючи, можна сказати, що роль тестувальника у роботі з вимогами є надзвичайно важливою і багатогранною. Він не просто пасивно читає готові вимоги, а є активним учасником процесу їх створення та вдосконалення. Тестувальник бере участь в обговореннях, ставить уточнюючі питання, допомагає виявити прогалини та суперечності. Він оцінює тестованість вимог і пропонує формулювання, які роблять їх більш придатними для перевірки. Він створює тест-кейси на основі вимог і за допомогою RTM відстежує повноту покриття. При змінах вимог він аналізує їхній вплив на існуючі тести та ініціює їх оновлення. Фактично, тестувальник виступає захисником якості ще на етапі формулювання ідей, закладаючи фундамент для створення дійсно якісного продукту.

Висновок

Вимоги – це наріжний камінь будь-якого IT-проекту. Чіткі, повні, несуперечливі та тестовані вимоги є запорукою того, що команда розроблятиме саме те, що потрібно замовнику, і що кінцевий продукт відповідатиме очікуванням користувачів. Для тестувальника вміння працювати з вимогами, аналізувати їх, виявляти в них помилки та відстежувати їхній життєвий цикл є однією з ключових професійних компетенцій. Чим раніше тестувальник долучається до роботи з вимогами, тим більше проблем він може запобігти і тим вищою буде якість кінцевого продукту. На наступних лекціях ми перейдемо до практичних технік тест-дизайну, які дозволять вам на основі якісних вимог створювати ефективні тестові набори.

Питання для обговорення

1. Чому нечітко сформульовані вимоги є однією з головних причин провалу IT-проектів? Наведіть приклад двозначної вимоги та запропонуйте, як її можна було б переписати, щоб зробити однозначною.
2. Поясніть різницю між функціональними та нефункціональними вимогами. Наведіть приклади обох типів для веб-сайту інтернет-магазину.
3. Яка властивість вимог є найважливішою для тестувальника? Чому без цієї властивості тестування стає неможливим або вкрай суб’єктивним?
4. Як ви розумієте поняття "матриця простежуваності вимог" (RTM)? Яку практичну користь вона дає тестувальнику під час роботи над проектом та при внесенні змін?

5. Чому тестувальник має долучатися до аналізу та рецензування вимог якомога раніше? Які проблеми він може допомогти виявити ще до початку розробки?

6. У чому різниця між бізнес-вимогами, вимогами користувачів та функціональними вимогами? Чи може однакова вимога бути представлена на всіх трьох рівнях?

7. Що таке пріоритезація вимог і для чого вона потрібна? Якби ви були власником продукту, за яким принципом ви б визначали, що має найвищий пріоритет?

8. Як ви вважаєте, чи може тестувальник впливати на пріоритезацію вимог? Якщо так, то яким чином, спираючись на свій досвід та знання?

9. Опишіть типовий життєвий цикл вимоги. Чому управління змінами у вимогах є настільки важливим процесом, а не просто бюрократичною процедурою?

10. Як ви розумієте твердження: "Мета тестувальника – не просто перевірити, що програма відповідає вимогам, а й знайти те, чого у вимогах не вистачає"? Наведіть приклад ситуації, коли тестувальник може виявити прогалину у вимогах.

Лекція 5
Види тестування програмного забезпечення

Вступ

Ми поступово заглиблюємося у професійний світ тестування програмного забезпечення. На попередніх лекціях ми розглянули життєвий цикл розробки, методології, вимоги та документацію. Сьогодні ми підійшли до однієї з найважливіших тем, яка є основою практичної діяльності кожного тестувальника – класифікації видів тестування. Чому важливо розуміти цю класифікацію? Тому що програмне забезпечення можна перевіряти з різних боків, на різних рівнях, різними методами і з різною метою. Тестування функціональності суттєво відрізняється від тестування продуктивності, а модульне тестування не схоже на приймальне. Розуміння всього розмаїття видів тестування дозволяє тестувальнику обирати правильні інструменти для кожної конкретної ситуації, ефективно планувати свою роботу та забезпечувати максимальне покриття якості продукту. Сьогодні ми розглянемо класифікацію тестування в різних вимірах – від простих, спрощених схем до детальних, професійних класифікацій, що базуються на міжнародних стандартах.

Спрощена класифікація тестування

Для початківців тестування часто видається єдиним процесом – "поламати програму". Насправді ж існує безліч різновидів тестування, і перше, що варто засвоїти – це спрощена класифікація, яка допомагає структурувати розуміння. У найзагальнішому вигляді тестування можна поділити за кількома ключовими ознаками: за рівнем (модульне, інтеграційне, системне, приймальне), за доступом до коду (метод білої, сірої та чорної скриньки), за ступенем автоматизації (ручне та автоматизоване), за позитивністю сценаріїв (позитивне та негативне). Це база, на яку нашаровуються інші, більш специфічні види тестування.

Класифікація за запуском коду на виконання

Перший важливий поділ – це статичне та динамічне тестування. Статичне тестування виконується без запуску програмного коду. Воно включає в себе рецензування документації, код-рев'ю, аналіз вимог, перевірку архітектурних схем. Це надзвичайно ефективний спосіб виявлення дефектів на ранніх етапах, коли їх виправлення коштує найдешевше. Динамічне тестування, навпаки, передбачає запуск програми та виконання коду. Саме його зазвичай мають на увазі, коли говорять про тестування в побутовому сенсі. Тестувальник запускає додаток, виконує певні дії, вводить дані і спостерігає за поведінкою системи.

Класифікація за доступом до коду і архітектури

Наступний фундаментальний поділ базується на тому, наскільки глибоко тестувальник розуміє внутрішню будову програми. Тестування методом білої скриньки (white box testing) передбачає, що тестувальник має повний доступ до вихідного коду, знає внутрішню структуру, алгоритми, логіку роботи програми. Він може писати тести, які перевіряють конкретні шляхи виконання коду, окремі умови, цикли. Цей метод зазвичай використовують розробники під час модульного тестування, але й тестувальники-автоматизатори можуть застосовувати його для написання інтеграційних тестів.

Тестування методом чорної скриньки (black box testing) – це підхід, при якому внутрішня структура програми залишається невідомою. Тестувальник бачить лише входи (наприклад, екранні форми, API-запити) та виходи (результати на екрані, відповіді сервера). Його завдання – перевірити, чи відповідає поведінка системи очікуванням, описаним у вимогах. Саме так працює більшість мануальних тестувальників. Існує також проміжний варіант – тестування методом сірої скриньки (gray box testing), коли тестувальник має часткове уявлення про внутрішню будову, наприклад, знає структуру бази даних або архітектуру взаємодії компонентів, що дозволяє йому створювати більш цілеспрямовані тести.

Класифікація за ступенем автоматизації

Цей поділ ми вже детально розглядали на одній з попередніх лекцій, але варто нагадати. Ручне тестування (manual testing) виконується людиною вручну. Воно є незамінним для дослідницького тестування, оцінки зручності використання, перевірки нового, нестабільного функціоналу. Автоматизоване тестування (automated testing) виконується за допомогою спеціальних програмних засобів, які запускають заздалегідь написані скрипти. Воно ідеально підходить для регресійного тестування, навантажувальних тестів, часто повторюваних сценаріїв.

Класифікація за рівнем деталізації додатків (за рівнем тестування)

Це один із найважливіших поділів, який визначає ієрархію тестування. На найнижчому рівні знаходиться модульне (компонентне) тестування (unit/component testing). Воно перевіряє окремі, ізольовані частини програми – функції, методи, класи. Зазвичай виконується розробниками і має на меті переконатися, що кожен окремий "цеглинок" працює правильно.

Наступний рівень – інтеграційне тестування (integration testing). Воно перевіряє взаємодію між різними модулями або компонентами системи. Навіть якщо кожен модуль окремо працює ідеально, при їх об’єднанні можуть виникати проблеми – дані передаються не в тому форматі, виклики функцій відбуваються в неправильній послідовності, виникають конфлікти. Інтеграційне тестування покликане виявити саме такі дефекти.

Системне тестування (system testing) перевіряє всю систему в цілому, як єдиний комплекс. Воно виконується в середовищі, максимально наближеному до продуктивного, і оцінює відповідність системи функціональним та нефункціональним вимогам. Це той рівень, на якому працює більшість мануальних тестувальників.

І нарешті, найвищий рівень – приймальне тестування (acceptance testing), яке ми детально розглядали в лекції 2. Воно виконується замовником або кінцевими користувачами для підтвердження того, що система готова до експлуатації та відповідає їхнім очікуванням.

Класифікація за ступенем важливості функцій

Тестування може фокусуватися на різних за важливістю функціях. "Smoke" тестування (димне тестування) – це поверхнева перевірка найкритичніших функцій системи одразу після отримання нової збірки. Його мета – переконатися, що система взагалі запускається і основні сценарії працюють, перш ніж заглиблюватися в детальне тестування. Якщо smoke-тести не проходять, збірка вважається нестабільною і повертається розробникам на доопрацювання.

Критичне тестування шляху (critical path testing) зосереджується на перевірці ключових бізнес-сценаріїв, які є найважливішими для користувачів. Розширене тестування (extended testing) охоплює всю функціональність, включаючи другорядні функції та периферійні сценарії.

Класифікація за принципами роботи із застосунком

Цей поділ стосується того, які саме сценарії ми перевіряємо. Позитивне тестування (positive testing) перевіряє, чи працює система відповідно до специфікації при введенні коректних даних та виконанні правильних дій. Наприклад, при введенні правильного логіна та пароля користувач успішно входить у систему. Це база, яка підтверджує, що основна функціональність працює.

Негативне тестування (negative testing) перевіряє, як система поводиться при введенні некоректних даних або виконанні неправильних дій. Що станеться, якщо ввести неправильний пароль? Якщо залишити поле порожнім? Якщо ввести дуже довгий рядок? Якщо натиснути кнопку "Купити" двічі поспіль? Негативне тестування має на меті переконатися, що система коректно обробляє помилкові ситуації, видає зрозумілі повідомлення і не "падає" при неочікуваних діях користувача.

Класифікація за природою застосунку

Програмне забезпечення може бути різним за своєю природою, і це суттєво впливає на підходи до тестування. Тестування веб-додатків передбачає перевірку роботи в різних браузерах, адаптивності до різних розмірів екрану,

швидкості завантаження сторінок, безпеки передачі даних. Тестування мобільних застосунків додає нові виміри: різні операційні системи (iOS, Android), різні версії ОС, різні розміри екранів, сенсорне управління, робота з апаратними датчиками, поведінка при перериваннях (вхідний дзвінок, зміна орієнтації), енергоспоживання. Тестування десктопних додатків фокусується на сумісності з різними версіями операційних систем, роботі з файловою системою, використанні системних ресурсів.

Класифікація з фокусуванням на рівні архітектури

У багаторівневих додатках тестування може фокусуватися на окремих архітектурних рівнях. Тестування рівня подання (presentation layer testing) перевіряє користувацький інтерфейс, відображення даних, роботу елементів управління. Тестування рівня бізнес-логіки (business logic layer testing) перевіряє реалізацію правил та алгоритмів, які становлять суть програми. Тестування рівня даних (data layer testing) фокусується на роботі з базою даних: коректності запису та читання, цілісності даних, виконанні транзакцій.

Класифікація за ступенем формалізації

Цей поділ стосується того, наскільки чітко визначені процедури тестування. Тестування за тест-кейсами (formal testing) – це виконання заздалегідь написаних, детальних інструкцій з чіткими кроками та очікуваними результатами. Такий підхід забезпечує повторюваність і повноту покриття.

Дослідницьке тестування (exploratory testing) – це одночасне вивчення програми, проектування тестів та їх виконання. Тестувальник не має наперед визначеного сценарію, він досліджує додаток, керуючись своєю інтуїцією, досвідом та розумінням ризиків. Це надзвичайно ефективний спосіб знаходження неочевидних дефектів, які важко передбачити при написанні тест-кейсів. Адхок тестування (ad-hoc testing) – це різновид дослідницького, але ще більш неформальний, без будь-якого планування, просто "погратися" з програмою.

Класифікація із залученням кінцевих користувачів

Ми вже торкалися цієї теми в лекції про приймальне тестування. Альфа-тестування проводиться всередині компанії-розробника, але із залученням співробітників, які не є членами команди. Бета-тестування виконується реальними користувачами в їхньому природному середовищі. Гамма-тестування – це, по суті, фінальна версія продукту, яка випускається на широкий ринок, але все ще може збирати відгуки для майбутніх оновлень.

Класифікація за цілями

Цей вимір класифікує тестування залежно від того, яку саме характеристику якості ми перевіряємо. Функціональне тестування перевіряє, чи правильно система виконує свої функції. Тестування продуктивності оцінює швидкодію, час відгуку, пропускну здатність. Навантажувальне тестування перевіряє поведінку системи при очікуваному навантаженні, а стрес-тестування – при навантаженнях, що перевищують нормальні, аж до "падіння" системи, щоб побачити, як вона відновлюється.

Тестування безпеки виявляє вразливості та оцінює захищеність системи від несанкціонованого доступу. Тестування зручності використання (usability testing) оцінює, наскільки інтуїтивно зрозумілим та зручним є інтерфейс для користувача. Тестування сумісності (compatibility testing) перевіряє роботу програми в різних середовищах – браузерах, операційних системах, пристроях. Тестування локалізації перевіряє коректність перекладу та адаптацію до культурних особливостей різних регіонів.

Класифікація за ступенем втручання у роботу програми

Інвазивне тестування (invasive testing) передбачає втручання в роботу програми – наприклад, модифікацію даних безпосередньо в базі даних, підміну модулів, імітацію збоїв. Неінвазивне тестування (non-invasive testing) виконується виключно через стандартні інтерфейси користувача, без втручання у внутрішні процеси.

Класифікація на основі вибору вхідних даних

Тестування може базуватися на різних стратегіях вибору вхідних даних. Тестування еквівалентного розбиття (equivalence partitioning) передбачає поділ множини можливих вхідних даних на класи еквівалентності, де поведінка програми очікується однаковою, і вибір одного представника з кожного класу. Аналіз граничних значень (boundary value analysis) фокусується на перевірці значень на границях цих класів, оскільки саме там найчастіше ховаються помилки. Тестування на основі таблиць рішень (decision table testing) використовується для перевірки складних бізнес-логік з багатьма умовами.

Класифікація на основі середовища виконання

Тестування може виконуватися в різних середовищах. Тестування на продуктивному середовищі (production testing) виконується на живій системі з реальними даними, але зазвичай обмежується моніторингом та A/B-тестами. Тестування на стейджингу (staging testing) виконується в середовищі, максимально наближеному до продуктивного, але ізольованому від реальних користувачів. Саме там зазвичай проходить основна фаза тестування перед релізом.

Класифікація на основі структур коду та поведінки програми

Тестування на основі структур коду (code-based testing) включає в себе такі техніки, як тестування покриття операторів (statement coverage), гілок (branch coverage), шляхів (path coverage). Воно належить до методів білої скриньки і використовується переважно на рівні модульного тестування. Тестування на основі моделей поведінки (model-based testing) передбачає створення моделей очікуваної поведінки системи та генерацію тестів на основі цих моделей.

Класифікація за моментом виконання (хронології)

Цей поділ стосується того, в якому порядку виконується тестування різних рівнів. Висхідне тестування (bottom-up testing) починається з найнижчих рівнів (модулів) і поступово переходить до інтеграції та системи в цілому. Низхідне тестування (top-down testing) починається з верхніх рівнів, а нижчі рівні імітуються за допомогою "заглушок". Гібридне тестування (hybrid testing) поєднує обидва підходи.

Послідовність тестування

У реальному проекті тестування виконується в певній послідовності, яка логічно випливає з усіх розглянутих класифікацій. Спочатку, ще до запуску коду, проводиться статичне тестування документації. Після отримання першої збірки виконується smoke-тестування, щоб переконатися в її базовій працездатності. Потім проводиться детальне функціональне тестування нового функціоналу, включаючи як позитивні, так і негативні сценарії. Після стабілізації функціональності можна переходити до нефункціональних видів тестування – продуктивності, безпеки, сумісності. І весь цей час паралельно можна займатися дослідницьким тестуванням для пошуку неочевидних дефектів.

Додаткові класифікації та стандарти

Існують також професійні стандарти, які узагальнюють та систематизують види тестування. Один із найавторитетніших – міжнародний стандарт ISO/IEC/IEEE 29119-4, який надає детальну класифікацію технік тестування. Знання таких стандартів допомагає тестувальникам говорити спільною професійною мовою, порівнювати підходи та обирати найефективніші техніки для конкретних завдань.

Висновок

Як ми побачили, тестування програмного забезпечення – це не єдиний процес, а ціла сукупність різноманітних видів діяльності, кожен з яких має свою мету, свої методи та своє місце в життєвому циклі розробки. Розуміння цієї класифікації дозволяє тестувальнику свідомо підходити до планування своєї роботи, обирати правильні інструменти для кожної ситуації та ефективно комунікувати з командою про те, що саме і як буде перевірятися. На наступних лекціях ми детальніше розглянемо окремі техніки тест-дизайну та специфіку різних видів тестування, що дозволить вам застосовувати ці знання на практиці.

Питання для обговорення

1. Поясніть різницю між статичним та динамічним тестуванням. Чи можна вважати, що статичне тестування є менш важливим, оскільки воно не передбачає запуску коду? Чому?
2. Уявіть, що ви тестуєте новий мобільний додаток. Які види тестування (функціональне, нефункціональне, пов’язані зі змінами) ви застосуєте в першу чергу, а які – пізніше? Обґрунтуйте свою відповідь.
3. Чому тестування методом «білої скриньки» зазвичай виконують розробники, а тестування методом «чорної скриньки» – тестувальники? Чи можлива тут взаємозамінність?
4. Поясніть різницю між smoke-тестуванням та критичним тестуванням шляху (critical path testing). Коли і для чого виконується кожен з цих видів?
5. Наведіть приклади позитивного та негативного тестування для функції "відновлення паролю". Чому негативне тестування часто виявляє більше прихованих дефектів, ніж позитивне?
6. Як відрізняються підходи до тестування веб-додатків, мобільних застосунків та десктопних програм? Які специфічні аспекти потрібно враховувати в кожному випадку?
7. Поясніть різницю між модульним, інтеграційним та системним тестуванням. Чи можна пропустити інтеграційне тестування і одразу перейти до системного? Які ризики це несе?
8. У чому полягає різниця між тестуванням продуктивності, навантажувальним тестуванням та стрес-тестуванням? Які цілі переслідує кожен з цих видів?
9. Як ви розумієте термін «дослідницьке тестування» (exploratory testing)? Чому, маючи детальні тест-кейси, команда все одно практикує дослідницьке тестування?
10. Проаналізуйте ієрархію тестування: модульне → інтеграційне → системне → приймальне. Чи може дефект, не виявлений на нижчому рівні, бути знайдений на вищому, і яка ціна такого виправлення?

Лекція 6
Основи тестової документації

Вступ

Протягом попередніх лекцій ми неодноразово згадували різні види документів, які супроводжують процес тестування: тест-плани, чек-листи, тест-кейси, баг-репорти. Сьогодні настав час розглянути цю тему системно і глибоко. Тестова документація – це не просто бюрократична вимога чи марна трата часу, як іноді здається початківцям. Це фундаментальний інструмент професійної комунікації, планування, контролю якості та передачі знань. Якісно складена документація дозволяє команді говорити однією мовою, забезпечує повторюваність процесів, зберігає знання про продукт та допомагає новим членам команди швидко ввійти в курс справи. Сьогодні ми розглянемо основні види тестової документації, їх призначення, структуру, переваги та недоліки, а також навчимося розрізняти ситуації, коли доцільно використовувати той чи інший документ.

Тестова документація: визначення та призначення

За визначенням, тестова документація – це набір документів, що описують підходи, стратегії, методи та процедури тестування програмного забезпечення. Вона відіграє ключову роль у забезпеченні якості продукту та комунікації між командами розробки, тестування і замовниками. Навіщо вона потрібна? По-перше, вона формалізує процес тестування, роблячи його зрозумілим і передбачуваним. По-друге, вона забезпечує єдине розуміння того, що і як має бути протестовано. По-третє, вона слугує доказом виконаної роботи та основою для звітності. По-четверте, вона зберігає знання про продукт, які інакше могли б залишитися лише в головах окремих співробітників, створюючи ризик їх втрати при звільненні.

Тестову документацію можна поділити на формальну та неформальну. Формальна документація має чітко визначену структуру, обов’язкові атрибути, проходить процес узгодження і затвердження. Вона використовується в проектах з високими вимогами до відповідності стандартам, наприклад, у державних установах, банках, медичних закладах. Неформальна документація є більш гнучкою, створюється "для себе" або для внутрішнього використання в команді, не потребує складних процедур узгодження і дозволяє швидко фіксувати ідеї та плани.

Формальна тестова документація

До формальної документації належить насамперед технічне завдання (ТЗ) та інші документи з вимогами, про які ми детально говорили на минулій лекції. Вони є вихідною точкою для всього тестування. На основі вимог створюються

тест-кейси – детальні інструкції для перевірки конкретних функцій. Тест-кейси мають чітку структуру, обов'язкові атрибути, і їхнє виконання дає однозначний результат. Іншим важливим видом формальної документації є баг-репорти – звіти про знайдені дефекти, які також мають стандартизовану структуру, що забезпечує ефективну комунікацію між тестувальниками та розробниками. Ми детально розглянемо структуру баг-репортів в окремій лекції.

Неформальна тестова документація

До неформальної документації належить насамперед чек-листи. Чек-лист – це список перевірок, які необхідно виконати під час тестування. Він не містить детальних покрокових інструкцій, а лише нагадує тестувальнику, що саме потрібно перевірити. Наприклад, чек-лист для тестування форми реєстрації може містити пункти: "перевірка валідації порожніх полів", "перевірка валідації некоректної електронної пошти", "перевірка успішної реєстрації з коректними даними". Як виконувати ці перевірки – вирішує сам тестувальник.

Чек-листи надзвичайно популярні в Agile-командах завдяки своїй гнучкості та швидкості створення. Вони дозволяють швидко зафіксувати обсяг перевірок, не витрачаючи час на детальний опис кожного кроку. Крім того, чек-листи дають тестувальнику певну свободу, заохочуючи дослідницьке мислення, оскільки він сам вирішує, як саме виконувати перевірку. Однак у чек-листів є й недоліки: результати тестування залежать від кваліфікації та уважності конкретного тестувальника, а самі перевірки можуть бути неоднаково виконані різними людьми.

Іншим видом неформальної документації є тест-план – документ, що описує загальну стратегію тестування, обсяг робіт, ресурси, розклад, ризики. Тест-план може бути як формальним, багатосторінковим документом, так і неформальним, створеним "для себе" в простій таблиці. Також до неформальної документації належать матриці трасування (traceability matrices), які ми розглядали в лекції про вимоги.

Детальніше про чек-листи

Чек-листи можуть мати різний ступінь деталізації. Простий чек-лист – це звичайний список пунктів. Більш складний чек-лист може бути організований у вигляді таблиці, де вказуються різні середовища тестування, результати для кожного середовища, примітки. Наприклад, чек-лист для перевірки сумісності веб-сайту з різними браузерами може мати колонки для Chrome, Firefox, Safari, Opera, а в рядках – конкретні функції для перевірки. У клітинках таблиці проставляються результати: Passed (пройдено), Failed (провалено), Blocked (заблоковано іншою помилкою), Skipped (пропущено).

Переваги використання чек-листів очевидні. Вони прості у створенні та підтримці, їх легко оновлювати при змінах у продукті. Вони дозволяють швидко охопити великий обсяг функціональності. Вони є чудовим інструментом для планування тестування. Вони дають тестувальнику свободу

та заохочують до творчого підходу. Вони особливо ефективні для регресійного тестування, коли потрібно швидко перевірити стабільність продукту після змін.

Однак є й недоліки. Чек-листи не гарантують однакового виконання тестів різними людьми. Вони можуть бути недостатньо детальними для складних сценаріїв. Вони не підходять для початківців, які ще не знають, як саме виконувати ту чи іншу перевірку. І найголовніше – результати тестування за чек-листами важко відтворити, оскільки не фіксуються точні кроки, які призвели до помилки.

Бус-фактор і документація

Говорячи про документацію, не можна оминути поняття "bus factor" (автобус-фактор). Це жартівлива, але дуже серйозна метафора, яка означає ризик втрати критично важливої інформації у разі, якщо ключовий співробітник раптово залишить проект (потрапить під автобус). Якщо всі знання про тестування певної складної функції зберігаються лише в голові одного тестувальника, то його звільнення або хвороба можуть паралізувати роботу. Саме тому документація – чек-листи, тест-кейси, інструкції – є важливим засобом зниження bus factor. Вона зберігає знання в доступній для всієї команди формі.

Mind Maps (ментальні карти) в тестуванні

Окремим і дуже корисним інструментом, який часто використовують тестувальники, є ментальні карти (mind maps). Це спосіб візуалізації інформації у вигляді дерева, де в центрі знаходиться основна тема, а від неї відходять гілки з підтемами, деталями, зв’язками. У тестуванні ментальні карти використовуються для різних цілей.

По-перше, для планування тестування. Тестувальник може створити mind map, де в центрі буде "Тестування функції X", а від неї відходитимуть гілки: "Позитивні сценарії", "Негативні сценарії", "Граничні значення", "Сумісність", "Продуктивність" тощо. Кожна гілка далі деталізується. Це дозволяє побачити загальну картину тестування, не заглиблюючись у деталі кожного тест-кейсу.

По-друге, для аналізу вимог. Mind map допомагає візуалізувати всі аспекти вимоги, побачити зв’язки між різними частинами, виявити прогалини та суперечності. По-третє, для документування результатів дослідницького тестування. Під час дослідження додатка тестувальник може фіксувати свої дії, знайдені дефекти, ідеї для подальших перевірок у вигляді mind map. Це створює наочну карту дослідження. Перевага mind maps у тому, що вони є надзвичайно гнучкими, легкими для сприйняття та дозволяють охопити великий обсяг інформації на одному аркуші.

Тест-кейси та їх життєвий цикл

Тепер перейдемо до більш формального виду документації – тест-кейсів. Якщо чек-лист відповідає на питання "що перевірити?", то тест-кейс детально описує "що і як тестувати". Тест-кейс – це набір вхідних даних, умов виконання та очікуваних результатів, розроблений для перевірки певної функції або сценарію використання програми. Тест-кейси створюються в спеціалізованих системах управління тестуванням, таких як TestRail, Zephyr, TestLink, або навіть у простих електронних таблицях.

Життєвий цикл тест-кейсу починається з його створення на основі вимог. Потім тест-кейс проходить рецензування – інші тестувальники або розробники перевіряють його на коректність, повноту, однозначність. Після рецензування тест-кейс додається до тестового набору і використовується для тестування. При виконанні тест-кейс може мати різні статуси: Passed (пройдено успішно), Failed (провалено, фактичний результат не збігається з очікуваним), Blocked (виконання неможливе через іншу, вже відому помилку), Not Run (ще не виконувався). Якщо у вимогах відбуваються зміни, тест-кейс має бути оновлений відповідно до нових реалій. Застарілі тест-кейси можуть бути позначені як deprecated і з часом видалені.

Атрибути тест-кейсу

Якісний тест-кейс має містити певний набір обов’язкових атрибутів. По-перше, це унікальний ідентифікатор, який дозволяє однозначно посилатися на тест-кейс у звітах та обговореннях. По-друге, це назва або заголовок, який коротко описує, що саме перевіряється. По-третє, це посилання на вимогу, яку перевіряє даний тест-кейс.

Далі йде розділ "Передумови" – умови, які мають бути виконані до початку тестування. Наприклад, "користувач має бути авторизований в системі", "в базі даних має бути створено тестовий продукт з ідентифікатором 123". Потім – власне "Кроки виконання" – детальний, покроковий опис дій, які має виконати тестувальник. Кожен крок має бути сформульований чітко і однозначно. Для кожного кроку або для тест-кейсу в цілому вказується "Очікуваний результат" – те, що тестувальник має побачити після виконання дій.

Додатковими атрибутами можуть бути пріоритет тест-кейсу, тестові дані, середовище виконання, статус (активний, застарілий), автор, дата створення та останньої зміни.

Чек-лист чи тест-кейс: що обрати?

Вибір між чек-листом і тест-кейсом залежить від багатьох факторів. Тест-кейси є незамінними, коли потрібна висока точність і повторюваність, наприклад, при тестуванні критичних функцій, де кожна деталь має значення. Вони також необхідні для навчання нових співробітників, які ще не знають

продукту. Тест-кейси є обов’язковими в проектах, що вимагають сертифікації за стандартами якості.

Чек-листи, з іншого боку, чудово підходять для швидких перевірок, для регресійного тестування, де важлива ширина покриття, а не глибина деталізації. Вони є "золотим стандартом" в Agile-командах, де цінується швидкість і гнучкість. Часто використовується комбінований підхід: для критичних сценаріїв пишуться детальні тест-кейси, а для менш критичних – чек-листи. Також тест-кейси можуть створюватися на основі чек-листів для складних або нестабільних функцій, де потрібна додаткова увага до деталей.

Матриця відповідності вимог

Повернемося до матриці трасування (RTM), яку ми згадували в лекції про вимоги. У контексті тестової документації вона набуває особливого значення. Матриця відповідності вимог – це таблиця, яка показує зв’язок між бізнес-вимогами, тестовими сценаріями, тест-кейсами та знайденими дефектами. Наприклад, у рядку для вимоги BR1 ми бачимо, що вона покривається тестовим сценарієм TS1, який складається з тест-кейсів TS1.TC1 та TS1.TC2, і під час виконання цих тест-кейсів було знайдено дефект D01. Така матриця дозволяє наочно побачити, які вимоги вже протестовані, які ще ні, і які дефекти з якими вимогами пов’язані. Це незамінний інструмент для звітності та аналізу повноти тестування.

State Transition

На слайді згадується "State Transition". Це не окремий вид документації, а техніка тест-дизайну, заснована на моделюванні станів системи та переходів між ними. Але результатом застосування цієї техніки може бути діаграма станів, яка є різновидом тестової документації. Вона візуалізує, які стани може приймати система (наприклад, замовлення може бути в станах "Нове", "Сплачене", "Відправлене", "Скасоване") і які події призводять до переходу між станами. Це допомагає створювати тести, які перевіряють усі можливі переходи, включаючи некоректні.

Тестова документація на реальному проекті

У реальному проекті набір тестової документації залежить від багатьох факторів: методології розробки, розміру команди, вимог замовника, регуляторних норм. У невеликому стартапі, що працює за Agile, вся документація може складатися з кількох чек-листів у Google Docs та карток у Trello. У великому банківському проекті це будуть сотні сторінок формальних специфікацій, тест-кейсів у TestRail, баг-репортів у Jira, детальних тест-планів та звітів. Важливо розуміти, що документація – це не самоціль, а засіб. Вона має бути достатньою для ефективної роботи команди, але не надмірною, щоб не перетворюватися на бюрократію, яка гальмує розвиток.

Висновок

Тестова документація є невід'ємною частиною професійного тестування. Вона забезпечує чіткість, повторюваність, контрольованість та прозорість процесу. Розуміння різних видів документації – від простих чек-листів і ментальних карт до детальних тест-кейсів і матриць трасування – дозволяє тестувальнику обирати правильні інструменти для кожної ситуації. Вміння створювати якісну, зрозумілу та корисну документацію є однією з ключових професійних компетенцій, яка відрізняє досвідченого фахівця від початківця. На наступній лекції ми детально розглянемо, як правильно створювати баг-репорти – один із найважливіших видів тестової документації, що забезпечує ефективну комунікацію між тестувальниками та розробниками.

Питання для обговорення

1. Для чого потрібна тестова документація, якщо тестувальник і так знає, що і як перевіряти? Чи не є вона просто зайвою бюрократією?
2. Поясніть різницю між чек-листом та тест-кейсом. У яких ситуаціях доцільніше використовувати чек-лист, а в яких – детальний тест-кейс?
3. Що таке «bus factor» (автобус-фактор) і як якісна тестова документація допомагає знизити цей ризик для проекту?
4. Як ви розумієте поняття «ментальна карта» (mind map) у тестуванні? Для вирішення яких завдань її можна використовувати найефективніше?
5. Які атрибути є обов'язковими для будь-якого тест-кейсу? Чому важливо вказувати очікуваний результат до виконання тесту, а не після?
6. У чому переваги та недоліки зберігання тест-кейсів у простих Excel-таблицях порівняно зі спеціалізованими системами (TestRail, Zephyr)?
7. Що таке матриця трасування (traceability matrix) і яку користь вона приносить тестувальнику, розробнику та менеджеру проекту?
8. Уявіть, що до вашої команди приєднався новий тестувальник-початківець. Який вид тестової документації (детальні тест-кейси чи чек-листи) ви б йому запропонували для роботи в перший тиждень і чому?
9. Як ви вважаєте, чи можна в Agile-проектах обійтися зовсім без тестової документації? Якщо ні, то який мінімальний набір документів все ж потрібен?
10. Поясніть поняття «State Transition» у контексті тестової документації. Як ця техніка допомагає документувати складну логіку роботи програми?

Лекція 7
Властивості якісних тест-кейсів

Вступ

На попередній лекції ми розглянули основи тестової документації, познайомилися з чек-листами, тест-кейсами, ментальними картами та іншими інструментами, які використовуються в роботі тестувальника. Сьогодні ми заглибимося в одну з ключових тем практичного тестування – властивості якісних тест-кейсів. Навчитися писати тест-кейси нескладно, але навчитися писати хороші, якісні тест-кейси, які будуть корисні команді, зрозумілі новачкам, ефективні для пошуку дефектів і легкі в підтримці – це справжнє мистецтво, яке приходить з досвідом. Сьогодні ми розберемо, якими властивостями мають володіти ідеальні тест-кейси, як формулювати кроки, як балансувати між простотою та складністю, як організовувати тест-кейси в набори і яких типових помилок слід уникати.

Основні властивості якісних тест-кейсів

Перш ніж заглиблюватися в деталі, варто визначити, що взагалі робить тест-кейс якісним. Хороший тест-кейс має бути зрозумілим, повним, однозначним, повторюваним, актуальним і мати чітко визначений очікуваний результат. Він має бути написаний так, щоб будь-який член команди, навіть той, хто ніколи не працював з цим модулем, міг взяти тест-кейс, виконати його і отримати той самий результат, який очікував автор. Крім того, хороший тест-кейс має бути ефективним – тобто з максимальною ймовірністю виявляти дефекти при мінімальних витратах часу на його виконання. І нарешті, він має бути легким у підтримці – при змінах у продукті його має бути легко оновити.

Правила формулювання тест-кейсів

Мова, якою написаний тест-кейс, має колосальне значення. Погано сформульований тест-кейс може призвести до неправильного виконання, хибних результатів і втрати часу. Існує кілька золотих правил формулювання, яких варто дотримуватися.

Перше правило – писати лаконічно, але зрозуміло. Тест-кейс не має бути занадто довгим, але водночас він має містити всю необхідну інформацію. Короткі, рубані фрази краще сприймаються, ніж довгі, складні речення. Наприклад, замість "Користувач повинен клікнути лівою кнопкою миші на кнопку, яка має назву ‘Зберегти’ і розташована в правому верхньому кутку екрану" краще написати "Натиснути кнопку ‘Зберегти’".

Друге правило – використовувати безособову форму дієслів або дієслова в наказовому способі. "Відкрити сторінку реєстрації", "Ввести логін", "Натиснути

кнопку" – це стандартний стиль, який використовується в більшості проектів. Важливо бути послідовним і не змішувати форми, наприклад, "Користувач відкриває сторінку" в одному кроці і "Натиснути кнопку" в іншому.

Третє правило – обов'язково вказувати точні імена і технічно вірні назви елементів додатку. Якщо кнопка називається "Створити новий продукт", саме так її і треба називати в тест-кейсі, а не "кнопка створення" чи "кнопка додати". Якщо поле має підказку "Введіть електронну пошту", в тест-кейсі воно має називатися "поле 'Електронна пошта'", а не "поле для мила". Це особливо важливо, коли в додатку є багато схожих елементів.

Четверте правило – не пояснювати базові принципи роботи з комп'ютером. Тест-кейс пишеться для професіоналів, які знають, як натискати кнопки, як вводити текст, як користуватися мишею. Не потрібно писати "натисніть ліву кнопку миші один раз", достатньо просто "натиснути". Не потрібно писати "введіть текст з клавіатури", достатньо "ввести".

П'яте правило – всюди називати одні й ті ж речі однаково. Якщо в одному тест-кейсі ви назвали головну сторінку "домашня сторінка", а в іншому – "головна", це створить плутанину. Варто виробити єдиний глосарій термінів і дотримуватися його в усій документації.

І останнє правило – слідувати прийнятому на проекті стандарту оформлення та написання тест-кейсів. У різних компаніях можуть бути різні вимоги до структури, формату, використання шрифтів, кольорів тощо. Важливо їх знати і дотримуватися, щоб документація виглядала єдиною і професійною.

Баланс між простотою та складністю тест-кейсів

Однією з ключових дилем при створенні тест-кейсів є вибір рівня їх складності. Чи варто робити тест-кейси максимально простими, кожен з яких перевіряє одну невелику функцію? Чи краще створювати складні, багатокрокові сценарії, які імітують реальну поведінку користувача? Відповідь, як це часто буває, залежить від контексту, і важливо розуміти переваги та недоліки обох підходів.

Прості тест-кейси мають низку незаперечних переваг. Їх легко читати, розуміти і виконувати. Вони зрозумілі навіть початківцям-тестувальникам і новим людям на проекті. Вони спрощують початкову діагностику помилки, оскільки звужують коло пошуку – якщо простий тест-кейс провалився, проблема, швидше за все, саме в тій маленькій функції, яку він перевіряє. Крім того, прості тест-кейси роблять наявність помилки очевидною і не викликають дискусій – якщо при натисканні кнопки "Зберегти" нічого не відбувається, це однозначна помилка.

Однак у простих тест-кейсів є й недоліки. Вони не відображають реальної поведінки користувачів, які рідко виконують лише одну дію. Користувачі зазвичай працюють зі складними сценаріями: авторизуються, шукають товар, додають його в кошик, оформлюють замовлення, вводять платіжні дані. Саме в таких складних взаємодіях часто ховаються дефекти, які неможливо виявити простими тест-кейсами. Крім того, програмісти рідко перевіряють складні,

багатокрокові випадки під час модульного тестування, тому саме тестувальники мають їх покривати.

Ідеальним є розумний баланс. Набір тест-кейсів має містити як прості, "атомарні" перевірки для швидкої діагностики та регресії, так і складні, наскрізні сценарії, які імітують реальні шляхи користувачів. Важливо також розуміти, що при взаємодії багатьох об’єктів ймовірність помилки підвищується, тому складні сценарії часто бувають більш ефективними для пошуку дефектів, ніж прості.

Інші важливі властивості тест-кейсів

Окрім правильного формулювання та балансу складності, якісні тест-кейси мають відповідати ще кільком важливим критеріям. Вони мають бути актуальними – тобто відповідати поточній версії продукту. Застарілі тест-кейси, які не оновлюються при змінах у продукті, приносять більше шкоди, ніж користі, оскільки вводять в оману і змушують витрачати час на непотрібні перевірки.

Тест-кейси мають бути повними, але без надмірності. Вони мають містити всі необхідні передумови, кроки та очікувані результати, але без зайвих деталей, які не впливають на результат. Наприклад, не потрібно вказувати, що після кожного кроку треба чекати завантаження сторінки – це мається на увазі.

Тест-кейси мають бути незалежними, де це можливо. Тобто результат виконання одного тест-кейсу не має впливати на можливість виконання іншого. Якщо тест-кейси залежать один від одного, це ускладнює паралельне виконання, ускладнює аналіз результатів (незрозуміло, чи тест-кейс провалився через свою помилку, чи через помилку в попередньому залежному тест-кейсі) і робить набір тестів крихким.

Набори тест-кейсів

Окремі тест-кейси рідко існують самі по собі. Зазвичай вони об’єднуються в набори (test suites) за певними ознаками. Набори тест-кейсів можуть бути організовані по-різному, і розуміння різних типів наборів допомагає ефективно структурувати тестування.

Вільні (Independent) набори тест-кейсів

Вільні набори складаються з тест-кейсів, які не залежать один від одного. Кожен тест-кейс можна виконувати окремо, в будь-якому порядку, і результат одного не впливає на інші. Головна перевага вільних наборів – вони дозволяють розподілити тест-кейси між різними тестувальниками або навіть командами для паралельного виконання. Це значно прискорює процес тестування. Крім того, якщо один тест-кейс провалюється, це не блокує виконання інших.

Прикладом вільного набору можуть бути тест-кейси для перевірки різних, не пов’язаних між собою функцій: тест-кейс для реєстрації, тест-кейс для

пошуку, тест-кейс для додавання товару в кошик. Всі вони можуть виконуватися незалежно, за умови, що в системі вже є тестові дані, необхідні для їх виконання.

Послідовні (Sequential) набори тест-кейсів

Послідовні набори складаються з тест-кейсів, які логічно випливають один з одного. Результат виконання попереднього тест-кейсу є передумовою для виконання наступного. Наприклад, тест-кейс "Реєстрація користувача" має бути виконаний перед тест-кейсом "Авторизація користувача", а той – перед тест-кейсом "Оформлення замовлення". Такі набори добре імітують реальні бізнес-сценарії і дозволяють перевіряти наскрізну роботу системи.

Однак у послідовних наборів є суттєві недоліки. Якщо перший тест-кейс провалюється, всі наступні стають невиконуваними (блокуються). Це ускладнює аналіз результатів і робить набір крихким. Крім того, послідовні набори важко розпаралелювати – вони мають виконуватися одним тестувальником у строго визначеному порядку.

Ізольовані та узагальнені тест-кейси

У презентації наводяться приклади різних типів тест-кейсів. Ізольований вільний тест-кейс – це тест-кейс, який перевіряє одну конкретну функцію і не залежить від інших. Наприклад, тест-кейс для реєстрації користувача, який містить всі необхідні кроки: відкрити сторінку реєстрації, ввести дані, натиснути кнопку, перевірити результат.

Узагальнений вільний тест-кейс – це тест-кейс, який перевіряє деяку загальну властивість системи в різних умовах. Наприклад, тест-кейс для перевірки сумісності з різними браузерами: відкрити продукт у Chrome, Firefox, Safari, Edge і перевірити, чи всі функції працюють коректно. Такий тест-кейс насправді є шаблоном, який при виконанні розкладається на кілька конкретних перевірок.

Ізольований послідовний тест-кейс – це тест-кейс, який є частиною послідовного набору і залежить від попередніх тест-кейсів. Наприклад, тест-кейс для авторизації користувача, який передбачає, що користувач вже зареєстрований на попередньому кроці. У такому тест-кейсі важливо чітко вказати цю залежність у передумовах.

Переваги різних типів тест-кейсів

Кожен тип тест-кейсів має свої переваги. Ізольовані вільні тест-кейси є найпростішими у виконанні, аналізі та підтримці. Вони дозволяють швидко локалізувати проблему. Вони ідеально підходять для автоматизації. Узагальнені тест-кейси дозволяють охопити велику кількість варіантів одним тест-кейсом, що економить час на написання та підтримку документації. Послідовні тест-

кейси найкраще імітують реальну поведінку користувачів і дозволяють виявити дефекти, які виникають лише при взаємодії кількох функцій.

На практиці хороший набір тест-кейсів містить усі типи в розумній пропорції. Ізольовані вільні тест-кейси використовуються для швидкої перевірки окремих функцій та регресії. Послідовні набори – для перевірки критичних бізнес-сценаріїв. Узагальнені – для перевірки сумісності, конфігурацій та інших параметризованих аспектів.

Принципи побудови наборів тест-кейсів

Існує кілька підходів до організації тест-кейсів у набори, і вибір конкретного підходу залежить від структури продукту, методології розробки та уподобань команди.

Перший підхід – на основі розбиття програми на модулі та підмодулі. Це найприродніший спосіб: для кожного модуля (наприклад, "Реєстрація", "Авторизація", "Пошук", "Кошик", "Оформлення замовлення") створюється окремий набір тест-кейсів. Це дозволяє легко орієнтуватися в документації і швидко знаходити тести для конкретної частини продукту.

Другий підхід – за принципом важливості функцій. Можна створити набори для критичних функцій (smoke tests), для менш важливих функцій та для всіх інших. Це допомагає пріоритезувати тестування і швидко перевіряти стабільність продукту після змін.

Третій підхід – за принципом угруповання для перевірки певного рівня вимог або типу вимог. Наприклад, можна створити набір тест-кейсів для перевірки функціональних вимог, окремий набір для перевірки вимог до безпеки, окремий – для вимог до продуктивності.

Четвертий підхід – за архітектурним принципом: набори для перевірки користувацького інтерфейсу (рівня подання), набори для перевірки бізнес-логіки, набори для перевірки рівня даних. Це особливо актуально для багаторівневих додатків, де різні рівні можуть тестуватися окремо.

П’ятий підхід – за областю внутрішньої роботи програми: тест-кейси, що зачіпають роботу з базою даних, тест-кейси, що зачіпають роботу з файловою системою, тест-кейси, що зачіпають роботу з мережею. Це допомагає фокусуватися на певних технічних аспектах.

І нарешті, шостий підхід – за видами тестування: функціональні тести, інтеграційні тести, регресійні тести, тести продуктивності, тести безпеки тощо. Це, мабуть, найпоширеніший спосіб організації в професійних командах.

Типові помилки при розробці чек-листів для тестування

Хоча основна увага сьогодні приділяється тест-кейсам, варто згадати й про типові помилки при створенні чек-листів, оскільки вони також є важливою частиною тестової документації. Перша помилка – надмірна деталізація, коли чек-лист перетворюється на набір тест-кейсів, втрачаючи свою головну перевагу – стислість. Друга помилка – недостатня деталізація, коли пункти

сформульовані настільки загально, що незрозуміло, що саме треба перевіряти ("перевірити все", "переконатися, що працює").

Третя помилка – відсутність логічної структури, коли пункти в чек-листі перемішані в безладному порядку. Четверта помилка – неоднозначні формулювання, які можна інтерпретувати по-різному. П'ята помилка – використання застарілих чек-листів, які не оновлюються відповідно до змін у продукті. І остання, але не менш важлива помилка – відсутність контексту, коли незрозуміло, для якого середовища, якої версії продукту або яких даних призначений чек-лист.

Типові помилки при розробці наборів тест-кейсів

При створенні наборів тест-кейсів також часто припускаються помилок. Перша і найпоширеніша – надмірна залежність тест-кейсів один від одного, про яку ми вже говорили. Це робить набір крихким і ускладнює аналіз. Друга помилка – неповне покриття вимог, коли деякі вимоги залишаються без тест-кейсів, а деякі, навпаки, покриваються надмірною кількістю дублюючих тестів.

Третя помилка – ігнорування негативних сценаріїв, коли набір містить лише позитивні тест-кейси, які перевіряють "happy path". Четверта помилка – відсутність пріоритезації, коли всі тест-кейси вважаються однаково важливими, що ускладнює планування тестування при обмеженому часі. П'ята помилка – неврахування різних середовищ виконання, коли тест-кейси пишуться для одного конкретного середовища і не працюють в інших.

Шоста помилка – надмірна складність тест-кейсів, коли вони намагаються перевірити занадто багато в одному тест-кейсі. Сьома помилка – відсутність чітких очікуваних результатів, коли незрозуміло, що вважати успішним виконанням. І восьма помилка – нехтування підтримкою тест-кейсів, коли вони не оновлюються при змінах у продукті і поступово стають марними.

Висновок

Створення якісних тест-кейсів – це навичка, яка розвивається з практикою та увагою до деталей. Хороший тест-кейс має бути зрозумілим, повним, однозначним, повторюваним, актуальним і ефективним. Він має бути написаний відповідно до стандартів проекту, з використанням точної термінології та без зайвих деталей. Важливо знаходити правильний баланс між простими, атомарними тест-кейсами для швидкої діагностики та складними, наскрізними сценаріями, які імітують реальну поведінку користувачів.

Організація тест-кейсів у набори має бути логічною і відповідати структурі продукту, видам тестування або іншим критеріям, важливим для команди. Важливо уникати типових помилок, таких як надмірна залежність тест-кейсів, неповне покриття вимог, ігнорування негативних сценаріїв та нехтування підтримкою документації.

Розуміння властивостей якісних тест-кейсів і вміння застосовувати ці знання на практиці відрізняє професійного тестувальника від аматора. На

наступній лекції ми розглянемо ще один надзважливий вид тестової документації – баг-репорти, і навчимося правильно описувати знайдені дефекти, щоб розробники могли їх швидко та ефективно виправляти.

Питання для обговорення

1. Як ви розумієте принцип «балансу» при створенні тест-кейсів? Чому не можна покладатися виключно на прості або виключно на складні, багатокрокові тест-кейси?
2. Проаналізуйте переваги та недоліки простих (атомарних) тест-кейсів. У яких ситуаціях вони є незамінними, а коли можуть бути неефективними?
3. Поясніть різницю між вільними (independent) та послідовними (sequential) наборами тест-кейсів. Які проблеми можуть виникнути при виконанні послідовного набору, якщо перший же тест-кейс провалився?
4. Чому важливо, щоб тест-кейси були незалежними один від одного (там, де це можливо)? Як залежність тест-кейсів впливає на можливість їх паралельного виконання?
5. Як ви розумієте вимогу «писати лаконічно, але зрозуміло» стосовно кроків тест-кейсу? Наведіть приклад поганого і хорошого формулювання одного і того ж кроку.
6. Чому в тест-кейсах не потрібно пояснювати базові принципи роботи з комп’ютером (наприклад, «натисніть ліву кнопку миші»)? Для кого пишуться тест-кейси?
7. Які принципи угруповання тест-кейсів у набори ви знаєте? За яким принципом ви б організували набори для тестування великого інтернет-магазину?
8. Назвіть основні типові помилки, яких припускаються при розробці чек-листів для тестування. Чим поганий надмірно деталізований чек-лист?
9. Чому тест-кейси мають бути актуальними? Які наслідки можуть бути, якщо команда використовує застарілі тест-кейси, які не оновлювалися після змін у продукті?
10. Як ви розумієте принцип «узагальненого вільного тест-кейсу», наведений у презентації? У яких випадках такі тест-кейси є особливо корисними?

Лекція 8
Робота з багами: від виявлення до закриття

Вступ

Протягом попередніх лекцій ми детально розглянули, як планувати тестування, як створювати тестову документацію, як писати якісні тест-кейси. Але кульмінацією роботи будь-якого тестувальника є момент, коли він знаходить дефект. Саме для цього, власне, існує тестування – щоб виявляти проблеми до того, як їх знайдуть користувачі. Однак знайти помилку – це лише половина справи. Друга, не менш важлива половина – правильно її задокументувати, донести до розробника, простежити за її виправленням і переконатися, що проблема дійсно вирішена. Сьогоднішня лекція присвячена саме цьому – роботі з багами. Ми розглянемо, що таке дефект, яким він буває, який життєвий цикл проходить, як правильно складати баг-репорти та які інструменти використовуються для управління дефектами в сучасних IT-проектах.

Що таке баг? Перше визначення

Термін "баг" (bug) має цікаву історію. Легенда говорить, що в 1947 році інженери Гарвардського університету, працюючи над комп’ютером Mark II, виявили причину збою – метелик, який застряг між контактами електромеханічного реле. Вони вилучили комаху і вклеїли її в журнал з позначкою "first actual case of bug being found". З тих пір термін "баг" міцно увійшов до лексикону програмістів для позначення помилки в програмному забезпеченні.

Сьогодні під багом (або дефектом) розуміють невідповідність між фактичною поведінкою програми та очікуваною поведінкою, описаною у вимогах, специфікаціях або просто здоровому глузді. Баг – це коли програма робить щось, чого не повинна робити, або не робить щось, що повинна. Важливо розуміти, що не кожна незвичайна поведінка є багом. Якщо функція не була описана у вимогах і не очікувалася замовником, але програма поводиться дивно, це може бути недоліком вимог, а не дефектом реалізації. Але в реальному житті межа часто розмита, і тестувальник має керуватися професійним судженням.

Що вважати некоректною роботою?

Існує багато проявів некоректної роботи програми, і тестувальник має вміти їх розпізнавати. Найочевидніший випадок – це збій програми, що призводить до її аварійного завершення (crash). Користувач працює, і раптом

програма закривається, а на екрані з’являється системне повідомлення про помилку. Це критичний дефект, який робить подальшу роботу неможливою.

Інший поширений випадок – зависання програми (hang up), коли вона перестає реагувати на дії користувача, курсор миші може змінитися на "пісочний годинник" або індикатор очікування, і ніякі кліки не призводять до результату. Користувач не може нічого зробити, окрім як завершити програму примусово.

Далі – видача програмою системних повідомлень про помилки (system error). Користувач бачить вікно з незрозумілим текстом, повним технічних деталей, адрес пам’яті та кодів помилок. Такі повідомлення лякають звичайних користувачів і свідчать про те, що програма не обробила виняткову ситуацію належним чином.

Помилки дизайну, що заважають нормальній роботі з програмою, також є дефектами. Наприклад, кнопка, яку неможливо натиснути, бо вона перекрита інформаційним повідомленням, або елемент інтерфейсу, який виходить за межі екрану і стає недоступним, або текст, який зливається з фоном через неправильні кольори.

І нарешті, дрібні неточності та текстові помилки – орфографічні, граматичні, стилістичні. Вони не ламають роботу програми, але псують враження від продукту, знижують довіру до нього і можуть свідчити про загальну недбалість розробників. У професійних продуктах такі помилки є неприпустимими.

Джерела інформації про дефекти

Звідки тестувальник дізнається про те, що знайдена поведінка є дефектом? Основним джерелом є вимоги та специфікації. Якщо в документі написано, що після натискання кнопки "Зберегти" дані мають бути записані в базу, а вони не записуються – це баг. Якщо написано, що пароль має містити не менше 8 символів, а система дозволяє ввести 5 – це баг.

Але що робити, якщо вимоги неповні або відсутні? Тоді тестувальник керується здоровим глуздом, очікуваннями користувача, загальноприйнятими стандартами та власним досвідом. Наприклад, навряд чи в будь-яких вимогах написано, що кнопка "Відправити" не має накладатися на поле введення. Але якщо це відбувається, це очевидна помилка дизайну. Також джерелом можуть бути попередні версії продукту, конкуруючі продукти, законодавчі та регуляторні норми.

Звіт про дефект (баг-репорт)

Коли тестувальник знаходить дефект, він має його задокументувати. Документ, в якому описується знайдена помилка, називається баг-репортом (bug report) або звітом про дефект. Це ключовий інструмент комунікації між тестувальником та розробником. Від того, наскільки якісно складений баг-репорт, залежить, чи зможе розробник зрозуміти проблему, відтворити її і

виправити. Поганий баг-репорт призводить до марної витрати часу, зайвих уточнень, непорозумінь, а іноді – до того, що дефект так і залишається невиправленим.

Якісний баг-репорт має містити кілька обов'язкових елементів. По-перше, це унікальний ідентифікатор (ID), за яким можна однозначно ідентифікувати дефект у системі. По-друге, це короткий, але інформативний заголовок (summary), який дає зрозуміти суть проблеми з першого погляду. По-третє, це детальний опис кроків для відтворення (steps to reproduce) – покрокова інструкція, виконавши яку, будь-хто зможе побачити проблему. По-четверте, це фактичний результат (actual result) – те, що відбувається насправді при виконанні кроків. По-п'яте, це очікуваний результат (expected result) – те, що мало б відбуватися за задумом.

Додатковими, але не менш важливими атрибутами є середовище (environment) – інформація про те, де виникла помилка (операційна система, браузер, версія продукту, роздільна здатність екрану тощо), серйозність (severity) та пріоритет (priority), які ми розглянемо окремо, а також вкладення (attachments) – скріншоти, відео, логи, які допомагають візуалізувати проблему.

Життєвий цикл дефекту

Баг, як і будь-який інший артефакт розробки, проходить певний життєвий цикл. Він починається з моменту виявлення і закінчується моментом закриття. Існують різні варіації цього циклу залежно від процесів у конкретній компанії, але загальна логіка є спільною.

Початковий статус, який присвоюється новому дефекту – це "NEW" (новий) або "OPEN" (відкритий). Тестувальник створив баг-репорт, і він потрапляє на розгляд команди. Далі настає етап тріаджу (triage), на якому менеджер, провідний розробник або вся команда вирішують, чи дійсно це помилка, чи, може, це нова вимога або неправильне розуміння функціоналу. Якщо визнано, що це дійсно дефект, йому присвоюється пріоритет, і він передається розробнику. Статус змінюється на "ASSIGNED" (призначений).

Розробник аналізує проблему, локалізує її у коді, вносить зміни і виправляє дефект. Після цього він змінює статус на "FIXED" (виправлений) або "RESOLVED" (вирішений) і передає баг тестувальнику на перевірку. Тестувальник отримує нову версію продукту з виправленням, виконує повторне тестування (re-test) за тими самими кроками, що описані в баг-репорті. Якщо проблема більше не відтворюється, він змінює статус на "VERIFIED" (перевірений). Згодом, після виходу релізу, баг може бути закритий остаточно – статус "CLOSED" (закритий).

Але життєвий цикл не завжди такий лінійний. Якщо після перевірки тестувальник виявляє, що проблема залишилася або виправлена лише частково, він змінює статус назад на "REOPENED" (перевідкритий) і повертає баг розробнику. Якщо під час тріаджу вирішено, що це не дефект, а особливість роботи (works as intended), баг може бути закритий зі статусом "NOT A BUG" (не баг) або "WON'T FIX" (не виправлятиметься). Також може бути прийнято

рішення відкласти виправлення до наступних версій – статус "DEFERRED" (відкладений).

Алгоритм роботи з дефектом

Узагальнюючи, алгоритм роботи з дефектом для тестувальника виглядає так. Перший крок – виявлення. Тестувальник виконує тест-кейс або досліджує програму і помічає невідповідність очікуваній поведінці. Другий крок – перевірка на відтворюваність. Важливо переконатися, що проблема виникає не випадково, а стабільно відтворюється при виконанні певних дій. Якщо проблема плаваюча, варто спробувати різні варіанти, щоб звузити умови її виникнення.

Третій крок – аналіз та локалізація. Тестувальник намагається зрозуміти, за яких саме умов виникає помилка, чи залежить вона від даних, від середовища, від послідовності дій. Це допоможе точніше описати проблему в баг-репорті. Четвертий крок – перевірка на дублікати. Перш ніж створювати новий баг-репорт, варто переконатися, що така сама проблема ще не була зареєстрована раніше. Якщо дублікат існує, новий баг не створюється, а інформація додається до існуючого.

П’ятий крок – створення баг-репорту. Тестувальник заповнює всі необхідні поля: заголовок, кроки, результати, середовище, серйозність, пріоритет, прикріплює скріншоти. Шостий крок – відстеження. Після створення баг-репорту тестувальник слідкує за його статусом, відповідає на запитання розробників, надає додаткову інформацію, якщо потрібно. Сьомий крок – перевірка виправлення. Коли розробник позначає баг як виправлений, тестувальник обов’язково перевіряє це в новій версії продукту. Восьмий крок – закриття. Якщо все добре, баг закривається.

Інструменти управління звітами про дефекти

У сучасній розробці для управління дефектами використовуються спеціалізовані інструменти – баг-трекери (bug trackers) або системи управління завданнями (issue tracking systems). Вони дозволяють централізовано зберігати всі баг-репорти, відстежувати їхній статус, призначати відповідальних, коментувати, додавати вкладення, будувати звіти. Розглянемо найпопулярніші з них.

Jira – безумовний лідер ринку. Це потужна платформа для управління проектами, розроблена компанією Atlassian. Jira підтримує різні методології (Scrum, Kanban), має гнучкі налаштування робочих процесів, дозволяє створювати дошки завдань, будувати звіти та дашборди. Для тестувальника Jira є основним інструментом для реєстрації та відстеження багів. Вона інтегрується з іншими інструментами Atlassian, зокрема Confluence для документації та Bitbucket для коду, а також з CI/CD системами.

Mantis – простий і легкий баг-трекер з відкритим кодом. Він орієнтований саме на відстеження дефектів, має простий інтерфейс, гнучкі налаштування і не

вимагає значних ресурсів. Mantis часто використовується в невеликих командах або проектах з обмеженим бюджетом, де не потрібна вся потужність Jira.

Redmine – ще один відкритий серверний додаток для управління проектами. Він підтримує баг-трекінг, календарі, форуми, Вікі, діаграми Ганта. Redmine дозволяє вести кілька проектів одночасно, має гнучке налаштування прав доступу і популярний у спільноті open-source розробників.

Azure DevOps (колишній Team Foundation Server) – комплексна платформа від Microsoft для управління повним життєвим циклом розробки. Вона включає Azure Boards (управління завданнями та багами), Azure Repos (зберігання коду), Azure Pipelines (CI/CD), Azure Test Plans (управління тестуванням) та Azure Artifacts (управління пакетами). Для тестувальника Azure DevOps надає інтегроване середовище, де можна створювати тест-плани, тест-кейси, виконувати їх і одразу реєструвати баги, які автоматично пов'язуються з тест-кейсами.

Confluence – це не баг-трекер, а платформа для управління знаннями, також від Atlassian. Але вона тісно інтегрується з Jira і використовується для зберігання тестової документації, вимог, інструкцій, звітів. У Confluence можна створювати бази знань, вести Вікі-сторінки, публікувати звіти за результатами тестування. Для тестувальника це місце, де зберігається вся необхідна інформація про продукт та процеси.

Порівняння інструментів

Кожен інструмент має свої сильні сторони. Jira надає найширші можливості для управління процесами, але вимагає навчання і налаштування. Mantis простий і швидкий у використанні, але обмежений у функціональності. Redmine є хорошим компромісом між функціональністю та простотою, особливо для open-source проектів. Azure DevOps пропонує повну інтеграцію всього життєвого циклу, що особливо цінно для команд, які працюють у Microsoft-екосистемі. Confluence є незамінним інструментом для документації.

Вибір інструменту залежить від потреб конкретної команди, бюджету, технологічного стеку та вподобань. Часто в одному проекті використовується кілька інструментів: Jira для багів, Confluence для документації, Azure DevOps для CI/CD.

Інтеграція інструментів

Сучасна розробка рідко обмежується одним інструментом. Важливо, щоб інструменти інтегрувалися між собою. Наприклад, інтеграція Jira з Confluence дозволяє вставляти в баг-репорти посилання на сторінки з вимогами. Інтеграція Jira з CI/CD системами (Jenkins, GitLab CI, Azure Pipelines) дозволяє автоматично оновлювати статус багів залежно від результатів збірок. Інтеграція з системами контролю версій (Git) дозволяє прив'язувати до баг-репортів коміти з виправленнями. Інтеграція з месенджерами (Slack, Teams) дозволяє отримувати сповіщення про зміни статусів багів у реальному часі. Чим краще

інтегровані інструменти, тим прозорішим і ефективнішим стає процес розробки.

Впровадження інструментів у реальні проекти

Перехід на нові інструменти або впровадження їх у новому проекті – це складний процес, який вимагає ретельного планування. Перший етап – планування та аналіз потреб. Команда має визначити, які саме завдання мають вирішувати інструменти, яка функціональність є критичною, який бюджет доступний. Другий етап – інтеграція та адаптація. Інструменти потрібно налаштувати відповідно до процесів команди, інтегрувати з існуючими системами, налаштувати права доступу. Третій етап – навчання та підтримка користувачів. Навіть найкращий інструмент буде марним, якщо команда не вміє ним користуватися. Потрібно провести тренінги, написати інструкції, надавати підтримку на початковому етапі. Четвертий етап – моніторинг та оптимізація. Після впровадження важливо збирати відгуки, аналізувати ефективність використання інструментів, вносити корективи в налаштування.

Виклики при впровадженні нових інструментів

Впровадження нових інструментів майже завжди пов’язане з викликами. Організаційні та культурні виклики – це опір змінам з боку команди, звикли до старих інструментів, небажання вчитися новому. Технічні виклики – це складність інтеграції з існуючими системами, проблеми з продуктивністю, необхідність додаткових налаштувань. Навчальні виклики – це час, необхідний для навчання всієї команди, різний рівень підготовки користувачів. Фінансові виклики – це вартість ліцензій, витрати на навчання, можливі витрати на консультантів. Подолання цих викликів вимагає чіткого плану, підтримки керівництва та активної участі всієї команди.

Висновок

Робота з багами – це центральний елемент діяльності тестувальника. Вміння знаходити дефекти – це лише початок. Набагато важливіше вміння грамотно їх описати, зареєструвати в баг-трекері, провести через весь життєвий цикл до закриття. Якісний баг-репорт – це не просто формальність, а інструмент ефективної комунікації, який економить час всієї команди. Розуміння життєвого циклу дефекту дозволяє тестувальнику правильно діяти на кожному етапі – від створення до перевірки виправлення. Володіння сучасними інструментами баг-трекінгу є обов’язковою навичкою для професійного тестувальника. На наступній лекції ми перейдемо до практичних технік тест-дизайну і розглянемо, як створювати ефективні тести, які максимізують ймовірність знаходження дефектів при мінімальних витратах.

Питання для обговорення

1. Що, на вашу думку, слід вважати дефектом (багом), а що – особливістю роботи програми (feature)? Хто приймає остаточне рішення в спірних випадках?
2. Опишіть типовий життєвий цикл дефекту. Чому дефект може бути переведений зі статусу «Виправлений» (Fixed) назад у статус «Відкритий» (Reopened)?
3. Як ви розумієте різницю між серйозністю (severity) та пріоритетом (priority) дефекту? Наведіть приклад дефекту з високою серйозністю, але низьким пріоритетом, і навпаки.
4. Чому вважається поганою практикою описувати кілька різних дефектів в одному баг-репорті? До яких проблем це може призвести?
5. Яку інформацію має містити якісний баг-репорт, щоб розробник міг швидко зрозуміти та відтворити проблему?
6. Порівняйте різні інструменти для відстеження дефектів (Jira, Mantis, Redmine, Azure DevOps). Які критерії слід враховувати при виборі інструменту для конкретної команди?
7. Як інтеграція баг-трекера з іншими інструментами (системами контролю версій, CI/CD, месенджерами) покращує процес розробки та тестування?
8. Уявіть, що ви знайшли дефект, але не можете його стабільно відтворити – він виникає час від часу. Як ви будете діяти? Що варто включити в баг-репорт у такому випадку?
9. Які основні виклики можуть виникнути при впровадженні нового інструменту для баг-трекінгу в команді? Як їх можна подолати?
10. Чому важливо аналізувати причини виникнення дефектів, а не просто їх виправляти? Яку користь це може принести проекту в довгостроковій перспективі?

Лекція 9
Властивості якісних звітів про дефекти

Вступ

На минулій лекції ми розглянули загальні принципи роботи з багами, їхній життєвий цикл та інструменти для їх відстеження. Сьогодні ми заглибимося в одну з найважливіших тем практичного тестування – властивості якісних звітів про дефекти. Чому це настільки важливо? Тому що баг-репорт є основним, а часто і єдиним каналом комунікації між тестувальником та розробником. Від того, наскільки якісно складений цей документ, залежить, чи зможе розробник зрозуміти проблему, чи витратить він години на спроби відтворити те, що тестувальник мав на увазі, чи буде дефект виправлений взагалі. Поганий баг-репорт – це джерело конфліктів, втраченого часу і, зрештою, невиправлених помилок, які потрапляють до користувачів. Хороший баг-репорт – це ознака професіоналізму тестувальника, його поваги до колег і турботи про якість продукту. Сьогодні ми розберемо, яким має бути ідеальний звіт про дефект, яких помилок слід уникати, які існують стандарти та метрики для оцінки якості баг-репортів.

Неякісний звіт про дефект: як виглядає проблема

Перш ніж говорити про те, як писати хороші звіти, варто подивитися на приклади поганих. Що таке неякісний баг-репорт? Це звіт, який не дає розробнику достатньо інформації для розуміння та відтворення проблеми. У ньому може бути заголовок на кшталт "Все зламалося" або "Не працює кнопка". Опис може бути відсутній або містити загальні фрази на кшталт "при спробі зробити щось вилітає помилка". Кроки для відтворення можуть бути неповними або незрозумілими. Може бути відсутня інформація про середовище, версію продукту, браузер. Можуть бути відсутні скріншоти або логи. Такий баг-репорт змушує розробника витрачати час на з’ясування деталей, ставити уточнюючі питання, намагатися вгадати, що мав на увазі тестувальник. Це демотивує, викликає роздратування і, зрештою, знижує ймовірність того, що дефект буде виправлений швидко і якісно.

Уявіть собі баг-репорт із заголовком "Проблема з формою реєстрації". Що саме не так? Форма не відкривається? Не приймає дані? Видає помилку після відправки? Не відправляє листа з підтвердженням? Без деталей розробник змушений витрачати час на дослідження, яке вже зробив тестувальник. Або баг-репорт, де в кроках написано просто "зареєструватися і побачити помилку". Як саме зареєструватися? З якими даними? На якому середовищі? У якому браузері? Такий звіт є марним.

Рекомендації з написання якісних звітів

Щоб уникнути описаних проблем, слід дотримуватися певних рекомендацій. Перша і найважливіша – заголовок має бути коротким, але інформативним. Він має одразу давати зрозуміти, про яку саме проблему йде мова. Наприклад, замість "Помилка в особистому кабінеті" краще написати "При спробі змінити аватар з'являється повідомлення 'Не вдалося завантажити файл' для зображень розміром більше 5 МБ". Такий заголовок вже містить ключову інформацію про те, де, за яких умов і яка саме проблема виникає.

Друга рекомендація – кроки для відтворення мають бути детальними, покроковими, однозначними. Вони мають бути написані в наказовому способі, чітко і зрозуміло. Кожен крок має бути окремим пунктом. Наприклад:

1. Відкрити сторінку реєстрації за посиланням https://example.com/register.
2. Ввести в поле "Електронна пошта" адресу "test@example.com".
3. Ввести в поле "Пароль" значення "Password123".
4. Ввести в поле "Підтвердження пароля" значення "Password123".
5. Натиснути кнопку "Зареєструватися".
6. Спостерігати результат.

Третя рекомендація – обов'язково вказувати фактичний результат (те, що відбувається насправді) та очікуваний результат (те, що мало б відбуватися за вимогами або здоровим глуздом). Наприклад: "Фактичний результат: після натискання кнопки 'Зареєструватися' сторінка перезавантажується, але повідомлення про успішну реєстрацію не з'являється, і новий користувач не створюється в базі даних. Очікуваний результат: після натискання кнопки 'Зареєструватися' має з'явитися повідомлення 'Реєстрація пройшла успішно. Перевірте свою електронну пошту для підтвердження', і новий користувач має бути створений у базі даних."

Четверта рекомендація – обов'язково вказувати середовище, в якому виникла помилка: операційна система, браузер та його версія, версія продукту, роздільна здатність екрану, будь-які інші релевантні деталі. Якщо помилка виникає лише в певному браузері, це критично важлива інформація для розробника.

П'ята рекомендація – додавати вкладення. Скріншот, на якому видно помилку, часто вартий тисячі слів. Якщо помилка складна або динамічна, може знадобитися відео. Логи (консоль браузера, логи сервера) можуть містити технічні деталі, які допоможуть розробнику швидше знайти причину.

Шоста рекомендація – правильно визначати серйозність (severity) та пріоритет (priority). Серйозність показує, наскільки сильно проблема впливає на роботу програми: критична (критичний збій), висока (важлива функція не працює), середня (функція працює з обмеженнями), низька (дрібні недоліки, косметичні проблеми). Пріоритет показує, як швидко проблема має бути виправлена з точки зору бізнесу: високий (виправити негайно), середній (виправити в найближчому релізі), низький (виправити, коли буде час). Це різні

поняття: критична помилка може мати низький пріоритет, якщо вона виникає в рідко використовуваній функції, а косметична помилка в головній сторінці може мати високий пріоритет, бо впливає на імідж компанії.

Дуже важливо!

У презентації наголошується на двох надзвичайно важливих принципах. Перший: окремі звіти для кожного нового дефекту. Якщо тестувальник знайшов кілька різних проблем, для кожної з них має бути створений окремий баг-репорт. Не можна писати один звіт на кілька різних помилок, навіть якщо вони здаються пов'язаними. По-перше, це ускладнює відстеження статусу кожної помилки окремо. По-друге, розробник може виправити одну з них і закрити звіт, а інші залишаться невиправленими. По-третє, різні помилки можуть мати різну серйозність, пріоритет і бути призначені різним розробникам. Один звіт – один дефект.

Другий принцип: відповідність прийнятим шаблонам оформлення і традиціям. У кожній компанії, на кожному проекті можуть бути свої стандарти оформлення баг-репортів. Десь прийнято писати заголовки за схемою "[Модуль] Короткий опис проблеми", десь використовуються певні мітки, десь обов'язково додавати посилання на тест-кейс, який виявив помилку. Важливо знати ці стандарти і неухильно їх дотримуватися. Це робить документацію єдиною, зрозумілою і професійною.

Алгоритм створення ефективних звітів про дефекти

Створення якісного баг-репорту можна представити як алгоритм, який варто виконувати щоразу при виявленні дефекту.

Перший крок – відтворення. Перш ніж писати звіт, переконайтеся, що проблема відтворюється стабільно. Виконайте кроки кілька разів, спробуйте різні варіанти, щоб зрозуміти, за яких саме умов виникає помилка. Якщо проблема плаваюча, спробуйте звузити коло пошуку і зафіксувати всі можливі деталі.

Другий крок – локалізація. Спробуйте зрозуміти, чи залежить помилка від даних, від середовища, від послідовності дій. Чи виникає вона лише з певним типом файлів? Лише в певному браузері? Лише після виконання певної попередньої дії? Це допоможе точніше описати проблему.

Третій крок – пошук дублікатів. Перевірте в системі, чи не була вже зареєстрована така сама або дуже схожа проблема. Якщо була, не створюйте новий звіт, а додайте свою інформацію (наприклад, про інше середовище) до існуючого. Якщо дублікатів немає, переходьте до наступного кроку.

Четвертий крок – документування. Створіть новий звіт і заповніть всі обов'язкові поля: заголовок, опис, кроки, фактичний результат, очікуваний результат, середовище, серйозність, пріоритет. Додайте скріншоти, відео, логи.

П'ятий крок – перевірка. Перед тим як зберегти звіт, перечитайте його очима іншої людини. Чи все зрозуміло? Чи достатньо деталей? Чи можна за цими кроками відтворити проблему? Якщо ви самі, через місяць, візьмете цей

звіт, чи зможете згадати, про що йшлося? Якщо є сумніви, додайте більше інформації.

Шостий крок – відправка. Збережіть звіт і переконайтеся, що він потрапив до правильної черги, призначений правильному розробнику (якщо це прийнято в процесі) і має правильні мітки.

Типові помилки при написанні звітів про дефекти

Розглянемо найпоширеніші помилки, яких припускаються тестувальники-початківці, і яких слід уникати.

Перша помилка – неповне описання проблеми. Тестувальник описав лише частину того, що побачив, або не вказав важливі деталі. Наприклад, написав "кнопка не працює", але не уточнив, що саме відбувається: кнопка не натискається, не реагує на клік, або після натискання нічого не відбувається.

Друга помилка – неясні або загальні заголовки. "Проблема з базою даних", "Помилка в адмінці", "Все падає" – такі заголовки не дають жодної інформації і змушують розробника відкривати звіт, щоб зрозуміти, про що взагалі йде мова.

Третя помилка – відсутність кроків для відтворення. Це робить звіт марним, оскільки розробник не знає, як побачити проблему на власні очі. Навіть якщо проблема здається очевидною, кроки мають бути описані.

Четверта помилка – невірна категоризація чи пріоритет. Надання критичного пріоритету косметичній помилці або, навпаки, ігнорування серйозної проблеми спотворює процес планування і може призвести до того, що важливі речі будуть відкладені, а другорядні – виправлятимуться першочергово.

П’ята помилка – відсутність скріншотів чи прикріплених файлів. Скріншот часто дає більше інформації, ніж текстовий опис. Він показує, як саме виглядає проблема, де саме в інтерфейсі вона виникає, яке саме повідомлення з’являється. Логи можуть містити технічні деталі, які неможливо описати словами.

Шоста помилка – невірне визначення середовища. Якщо не вказано, в якому браузері, на якій ОС, на якому пристрої виникла проблема, розробник може намагатися відтворити її в іншому середовищі, де вона не виникає, і вирішити, що проблеми немає.

Сьома помилка – надмірне використання технічного жаргону. Звіт має бути зрозумілим не тільки розробникам, але й менеджерам, аналітикам, іншим тестувальникам. Не треба писати "при виклику API /user/create з невалідним payload повертається 500". Краще написати "при спробі створити користувача з некоректними даними сервер повертає внутрішню помилку (код 500)". Якщо технічні деталі важливі, їх можна додати, але пояснити сутність простою мовою.

Восьма помилка – відсутність визначення очікуваних результатів. Без цього незрозуміло, що взагалі вважається помилкою. Можливо, тестувальник неправильно зрозумів вимоги, і така поведінка є очікуваною. Опис очікуваного результату знімає цю невизначеність.

Методики та стандартні підходи

Існують різні методики та стандартні підходи до написання баг-репортів, які допомагають структурувати інформацію і зробити її максимально корисною. Одна з найвідоміших – це методика "Five Ws" (Хто, Що, Де, Коли, Чому), адаптована для тестування: Хто (який користувач, яка роль), Що (яка дія виконувалась, що сталося насправді), Де (на якій сторінці, в якому модулі, в якому середовищі), Коли (за яких умов, після яких дій), Чому (чому це важливо, який очікуваний результат). Відповіді на ці питання дають повну картину проблеми.

Інший підхід – використання шаблонів. Багато компаній розробляють власні шаблони баг-репортів, які містять всі необхідні поля. Тестувальнику залишається лише заповнити їх. Це гарантує, що жодна важлива інформація не буде пропущена.

Ключові аспекти організації роботи інструментів і систем відстеження дефектів

Для того щоб баг-репорти приносили користь, важливо не тільки вміти їх писати, але й мати правильно організовану систему для їх відстеження. Перший аспект – стандартизація та структурованість даних. Всі поля мають бути чітко визначені, мати однозначні значення. Наприклад, поле "Серйозність" має мати фіксований набір значень (критична, висока, середня, низька), а не довільний текст. Це дозволяє будувати звіти та аналізувати дані.

Другий аспект – автоматизація процесів та контроль якості. Система може автоматично призначати відповідальних залежно від компонента, автоматично змінювати статуси, надсилати сповіщення, перевіряти на дублікати. Це зменшує ручну роботу і прискорює процес.

Третій аспект – забезпечення прозорості та зворотного зв’язку. Всі учасники процесу мають бачити, на якому етапі знаходиться кожен дефект, хто за нього відповідає, які коментарі залишаються. Це створює довіру та відповідальність.

Четвертий аспект – метрики ефективності та аналітика. Система має дозволяти будувати звіти за різними параметрами: скільки багів знайдено, скільки виправлено, середній час виправлення, найбільш проблемні модулі. Це допомагає керівництву приймати обґрунтовані рішення.

П’ятий аспект – гнучкість та адаптивність до процесів розробки. Система має дозволяти налаштовувати робочі процеси (workflows) відповідно до того, як саме працює команда, а не змушувати команду підлаштовуватися під систему.

Метрики та показники для оцінки ефективності звітів про дефекти

Як оцінити, наскільки якісними є баг-репорти, які створює команда? Існують певні метрики, які можна відстежувати.

Метрики повноти та якості інформації оцінюють, чи містять звіти всю необхідну інформацію. Це може бути суб'єктивна оцінка під час рев'ю або об'єктивна перевірка наявності обов'язкових полів.

Метрики оперативності реагування вимірюють, як швидко команда реагує на знайдені дефекти. Середній час вирішення дефекту (Mean Time to Resolution, MTTR) показує, скільки часу в середньому проходить від створення баг-репорту до його закриття. Час до першої реакції показує, як швидко розробник або менеджер звертає увагу на новий звіт.

Метрики коректності та повторюваності оцінюють, наскільки правильно описані дефекти. Коефіцієнт повторного відкриття дефектів (Defect Reopen Rate) показує, яка частка багів після позначки "виправлено" знову відкривається тестувальниками. Високий коефіцієнт свідчить про проблеми з якістю виправлень або з комунікацією. Кількість дублікатів показує, як часто різні тестувальники реєструють одну й ту саму проблему.

Метрики впливу звітів на загальний процес оцінюють, наскільки ефективно тестування виявляє дефекти. Defect Detection Efficiency (DDE) – це частка дефектів, знайдених під час тестування, від загальної кількості дефектів (знайдених під час тестування + знайдених користувачами після релізу). Defect Leakage – це кількість дефектів, які "протекли" в продуктивне середовище і були знайдені користувачами. Чим менше цей показник, тим краще.

Що кажуть стандарти про звіти?

Професійне тестування має міжнародні стандарти, які регламентують, зокрема, і звітність про дефекти. IEEE 829 – це стандарт тестової документації, який описує, які документи мають створюватися в процесі тестування, включаючи звіти про проблеми. Він визначає структуру таких звітів, обов'язкові розділи та їх вміст.

ISO/IEC/IEEE 29119 – це набір стандартів, що охоплюють всі аспекти тестування програмного забезпечення. Частина 29119-3 стосується тестової документації, включаючи звіти про дефекти. Стандарти визначають, яка інформація має бути зафіксована, щоб забезпечити повноту, точність і простежуваність. Хоча не всі компанії формально дотримуються цих стандартів, вони є корисною основою і джерелом кращих практик.

Адаптування звітування про дефекти до специфіки команди

Важливо розуміти, що не існує єдиного універсального рецепту ідеального баг-репорту, який підходить для всіх команд і всіх проектів. Те, що добре працює у великій корпорації з жорсткими процесами, може бути надмірним для невеликого стартапу. Тому процес звітування має адаптуватися до специфіки конкретної команди.

У невеликій команді, де тестувальники та розробники сидять в одній кімнаті і можуть постійно спілкуватися, баг-репорти можуть бути менш формальними. Головне – зафіксувати суть проблеми, а деталі можна

обговорити усно. У великій розподіленій команді, де люди працюють у різних часових поясах, баг-репорти мають бути максимально повними і самодостатніми, оскільки наступна можливість усно обговорити проблему може з’явитися лише через добу.

Важливо збирати зворотний зв’язок від розробників: чи зрозумілі їм баг-репорти, чи вистачає їм інформації, що можна покращити. На основі цього зворотного зв’язку процес звітування має постійно еволюціонувати, стаючи більш ефективним і зручним для всієї команди.

Висновок

Якісний звіт про дефект – це не просто формальність, а ключовий інструмент комунікації, який визначає ефективність всієї команди. Від того, наскільки добре тестувальник вміє описувати знайдені проблеми, залежить швидкість їх виправлення, рівень довіри між членами команди і, зрештою, якість кінцевого продукту. Хороший баг-репорт має бути зрозумілим, повним, однозначним, містити всі необхідні деталі – заголовок, кроки, фактичний та очікуваний результати, середовище, вкладення. Він має створюватися за принципом "один звіт – один дефект" і відповідати прийнятим у команді стандартам.

Уникання типових помилок, використання стандартних методик, розуміння метрик та адаптація процесів до потреб команди – все це ознаки професійного підходу до тестування. На наступних лекціях ми перейдемо до вивчення технік тест-дизайну, які допоможуть вам створювати ефективні тести і, відповідно, знаходити більше якісних багів, які ви вже вмітимете правильно документувати.

Питання для обговорення

1. Чому заголовок баг-репорту вважається одним із найважливіших його елементів? Наведіть приклад поганого (неінформативного) та хорошого заголовка для однієї й тієї ж помилки.
2. Проаналізуйте типові помилки при написанні звітів про дефекти. Які три помилки, на вашу думку, є найкритичнішими і чому?
3. Поясніть важливість принципу "Окремі звіти для кожного нового дефекту". Чому не можна створювати один звіт, у якому описано кілька різних проблем?
4. Як ви розумієте вимогу "відповідність прийнятим шаблонам оформлення"? Чи можна вважати це просто даниною бюрократії, чи в цьому є практичний сенс?
5. Чому в баг-репорті обов’язково потрібно вказувати не лише фактичний, але й очікуваний результат? Хіба не очевидно, як має працювати програма?
6. Яку роль відіграють скріншоти та відео в баг-репорті? Чи можуть вони замінити детальний текстовий опис кроків?
7. Що таке метрика Defect Reopen Rate (коефіцієнт повторного відкриття дефектів)? Про що може свідчити високе значення цієї метрики?

8. Як ви розумієте поняття Defect Detection Efficiency (DDE)? Чому ця метрика важлива для оцінки ефективності роботи команди тестування?

9. Як стандарти IEEE 829 та ISO/IEC/IEEE 29119 впливають на практику написання звітів про дефекти? Чи обов’язково їх дотримуватися в невеликих комерційних проектах?

10. Чому важливо адаптувати процес звітування про дефекти до специфіки конкретної команди та проекту? Чи можна створити універсальний шаблон баг-репорту, який підходить для всіх випадків?

Лекція 10
Вступ до технік тест-дизайну

Вступ

Ми поступово наближаємося до серцевини практичного тестування. Протягом попередніх лекцій ми вивчили життєвий цикл розробки, методології, види тестування, навчилися працювати з вимогами, створювати тестову документацію та писати якісні баг-репорти. Сьогодні ми робимо наступний важливий крок – знайомимося з техніками тест-дизайну. Що це таке? Тест-дизайн – це процес створення тестів, які максимально ефективно виявляють дефекти при мінімальних витратах ресурсів. Це перехід від інтуїтивного, "як підкаже досвід", підходу до системного, науково обґрунтованого методу. Замість того щоб просто "тицяти в екран" і сподіватися на удачу, тестувальник, який володіє техніками тест-дизайну, свідомо обирає, які тести створити, щоб з найбільшою ймовірністю знайти приховані проблеми. Сьогодні ми розглянемо основні техніки, розділені на три великі групи: техніки чорного ящика, техніки білого ящика та техніки, засновані на досвіді, а також навчимося обирати правильну техніку для конкретної ситуації.

Тест-дизайн: основні завдання і мета

Перш ніж заглиблюватися в конкретні техніки, важливо зрозуміти, навіщо вони взагалі потрібні. Уявіть собі програму, яка приймає від користувача кілька параметрів, кожен з яких може мати десятки можливих значень. Кількість усіх можливих комбінацій може обчислюватися мільйонами. Фізично неможливо перевірити їх усі. Тест-дизайн дає відповідь на питання: які саме комбінації варто перевірити, щоб з високою ймовірністю виявити дефекти, витративши прийнятний час? Він дозволяє оптимізувати процес тестування, зосередитися на найважливіших, найризикованіших місцях і уникнути марної роботи.

Порівняймо інтуїтивне тестування, яке часто використовують початківці, із системним тестуванням, заснованим на техніках тест-дизайну. Інтуїтивне тестування – це створення тестів випадково, на основі особистого досвіду або просто "як прийде в голову". Воно має низку недоліків: покриття нерівномірне, одні функції перевіряються багато разів, а інші можуть бути взагалі пропущені; документація часто відсутня, тому через місяць неможливо згадати, що саме і як перевірялося; відтворюваність низька – різні тестувальники виконують тести по-різному, отримуючи різні результати; автоматизація таких тестів ускладнена або неможлива.

Системне тестування, засноване на техніках тест-дизайну, позбавлене цих недоліків. Тести створюються структуровано, за чіткими правилами. Покриття є повним і контрольованим – ми точно знаємо, які класи вхідних даних, які гілки коду, які комбінації умов перевірені. Документація ведеться, тому тести

можна відтворити через будь-який час. Автоматизація таких тестів є природною і легко реалізується. Саме до системного підходу ми і будемо прагнути.

Основні техніки тест-дизайну: класифікація

Усі техніки тест-дизайну можна розділити на три великі групи. Перша група – техніки, засновані на специфікації, або техніки "чорного ящика". Вони використовуються, коли внутрішня структура програми невідома або неважлива, а тести створюються на основі вимог, специфікацій, описів функціональності. Друга група – техніки, засновані на структурі, або техніки "білого ящика". Вони використовуються, коли тестувальник має доступ до коду і будує тести на основі його логіки, перевіряючи різні шляхи виконання. Третя група – техніки, засновані на досвіді. Вони спираються на інтуїцію, знання та досвід тестувальника, на його розуміння того, де зазвичай ховаються помилки.

Техніки чорного ящика

Почнемо з технік чорного ящика, оскільки вони є основними в роботі мануального тестувальника.

Еквівалентне розбиття (Equivalence Partitioning)

Це одна з найпростіших і найпотужніших технік. Її суть полягає в тому, що множина всіх можливих вхідних даних розбивається на класи еквівалентності – групи значень, для яких поведінка програми очікується однаковою. Якщо програма працює правильно для одного значення з класу, то вона, швидше за все, працюватиме правильно і для всіх інших значень цього класу. І навпаки, якщо вона помиляється на одному значенні, то помилятиметься на всіх. Тому достатньо перевірити лише одне, представницьке значення з кожного класу.

Розглянемо приклад. У вимогах сказано, що вік користувача для реєстрації має бути від 18 до 65 років включно. Ми маємо три класи еквівалентності: клас валідних значень (від 18 до 65), клас невалідних значень менших за допустимі (менше 18) та клас невалідних значень більших за допустимі (більше 65). Для тестування ми оберемо по одному представнику з кожного класу: наприклад, 25 (валідний), 15 (невалідний, менший) і 70 (невалідний, більший). Якщо програма правильно прийме вік 25 і правильно відхилить вік 15 та 70, ми можемо з високою ймовірністю вважати, що вона буде правильно працювати для будь-якого віку в цих діапазонах.

Аналіз граничних значень (Boundary Value Analysis, BVA)

Ця техніка є логічним доповненням до еквівалентного розбиття. Практика показує, що помилки найчастіше виникають саме на границях допустимих

діапазонів, а не в їх середині. Тому аналіз граничних значень фокусується на перевірці значень безпосередньо на межах, а також одразу за межами.

Для нашого прикладу з віком від 18 до 65 граничними значеннями будуть: сама нижня межа (18), значення одразу нижче нижньої межі (17), значення одразу вище нижньої межі (19), сама верхня межа (65), значення одразу нижче верхньої межі (64) та значення одразу вище верхньої межі (66). Зверніть увагу: значення 17 і 66 належать до невалідних класів, але вони є граничними, тому їх перевірка особливо важлива.

Алгоритм застосування BVA простий: визначити допустимі діапазони, виділити граничні значення (нижня межа, верхня межа, безпосередньо нижче і безпосередньо вище кожної межі) і створити тест-кейси для кожного з них. Часто цю техніку використовують разом з еквівалентним розбиттям, оскільки вони добре доповнюють одна одну.

Тестування на основі таблиць прийняття рішень (Decision Table Testing)

Ця техніка використовується для перевірки складної бізнес-логіки, яка залежить від комбінацій різних умов. Наприклад, правила надання знижки можуть залежати від суми покупки, наявності дисконтної картки, дня тижня та статусу клієнта. Кількість можливих комбінацій може бути значною, і таблиця рішень допомагає систематизувати їх.

Алгоритм створення таблиці рішень такий. Спочатку виділяються всі умови (входи), що впливають на логіку. Для кожної умови визначаються можливі значення (найчастіше "так" або "ні", але може бути й більше варіантів). Далі виводяться всі можливі комбінації цих умов. Якщо умов n, то максимальна кількість комбінацій – 2 в степені n. Для кожної комбінації вказується очікувана дія або вихідне значення. Після цього таблицю можна спростити, видаливши дублікати або об’єднавши комбінації, що призводять до однакової дії. На основі отриманої таблиці будуються тест-кейси для кожного унікального сценарію.

Структура таблиці проста: рядки – це умови та дії, стовпці – окремі випадки (комбінації умов). На перетині вказуються значення умов (T – true, F – false) та позначки, які дії виконуються.

Тестування переходів станів (State Transition Testing)

Ця техніка застосовується, коли система може перебувати в різних станах, і перехід між станами відбувається під впливом певних подій. Класичні приклади – банкомат (стани: очікування картки, очікування PIN, очікування вибору операції тощо), стан замовлення в інтернет-магазині (нове, сплачене, відправлене, доставлене, скасоване), ліфт, діалогові вікна.

Для застосування цієї техніки будується діаграма станів, де вказуються всі можливі стани системи та події, що спричиняють переходи між ними. Потім створюються тест-кейси, які перевіряють кожен перехід: як валідні (наприклад, зі стану "очікування PIN" при введенні правильного PIN перейти в стан

"очікування вибору операції"), так і невалідні (наприклад, при введенні неправильного PIN залишитися в стані "очікування PIN" і збільшити лічильник спроб). Важливо перевірити також рідкісні або критичні переходи, наприклад, блокування картки після трьох невдалих спроб.

Тестування на основі використання (Use Case Testing)

Ця техніка базується на сценаріях використання системи реальними користувачами. Use case описує, як актор (користувач або інша система) взаємодіє з системою для досягнення певної мети. Наприклад, use case "Оформлення замовлення" може включати такі кроки: пошук товару, додавання до кошика, перехід до оформлення, введення адреси доставки, вибір способу оплати, підтвердження замовлення.

Тестування на основі use case дозволяє перевірити, чи підтримує система реальні бізнес-процеси, чи зручна вона для користувача, чи всі кроки працюють коректно в їх природній послідовності. Це доповнення до функціонального тестування, яке часто перевіряє окремі функції ізольовано. Use case тести показують, як ці функції працюють разом у реальному житті.

Техніки білого ящика

Тепер перейдемо до технік, що базуються на аналізі внутрішньої структури коду. Вони використовуються переважно розробниками під час модульного тестування, але можуть бути корисні й тестувальникам-автоматизаторам для написання інтеграційних тестів.

Покриття інструкцій (Statement Coverage)

Це найпростіша техніка білого ящика. Вона вимагає, щоб кожна інструкція в коді (кожен рядок, кожна команда) була виконана хоча б один раз під час тестування. Наприклад, у коді є умовний оператор if. Покриття інструкцій буде досягнуто, якщо ми виконаємо код так, що потрапимо всередину цього if, але не обов’язково перевіримо гілку else. Це мінімальний рівень покриття, який часто вважається недостатнім.

Покриття гілок (Branch Coverage)

Ця техніка вимагає, щоб були виконані всі можливі гілки коду. Для умовного оператора if це означає, що ми маємо перевірити і випадок, коли умова істинна (гілка then), і випадок, коли умова хибна (гілка else). Покриття гілок є більш суворим критерієм, ніж покриття інструкцій, і дозволяє виявити помилки, пов’язані з непередбаченою поведінкою в одній з гілок.

Покриття шляхів (Path Coverage)

Це найпотужніша, але й найскладніша техніка. Вона вимагає, щоб були виконані всі унікальні шляхи виконання коду. Якщо в коді є кілька умовних операторів, кількість можливих шляхів може зростати експоненційно. Наприклад, два послідовних оператори if створюють чотири можливі шляхи: (T,T), (T,F), (F,T), (F,F). На практиці досягти 100% покриття шляхів часто неможливо через величезну кількість варіантів, особливо якщо в коді є цикли. Тому ця техніка застосовується вибірково для найкритичніших ділянок коду.

Покриття умов (Condition Coverage) та комбіноване покриття

Ці техніки аналізують логічні умови всередині складених виразів. Наприклад, в умові (A && B) просте покриття гілок не гарантує, що ми перевірили всі можливі значення A та B. Покриття умов вимагає, щоб кожна з умов (A і B) прийняла як значення true, так і false хоча б один раз. Комбіноване покриття вимагає перевірки всіх можливих комбінацій значень умов (T,T; T,F; F,T; F,F).

Техніки, засновані на досвіді

Ці техніки не мають строгих математичних формул, але вони надзвичайно ефективні в руках досвідченого тестувальника.

Ad-hoc тестування

Це неформальне тестування без чітких сценаріїв і плану. Тестувальник просто "грається" з додатком, натискає на різні кнопки, вводить різні дані, намагаючись знайти щось незвичайне. Це хороший спосіб швидко перевірити стабільність програми, але покладатися тільки на нього не можна через низьку повторюваність і неповне покриття.

Дослідницьке тестування (Exploratory testing)

Це більш структурований підхід, ніж ad-hoc. Дослідницьке тестування – це одночасне вивчення продукту, проектування тестів та їх виконання. Тестувальник не має наперед визначеного сценарію, але він діє цілеспрямовано, керуючись своїми знаннями про продукт, гіпотезами про можливі проблеми та результатами попередніх дій. Він постійно ставить питання: "А що буде, якщо я зроблю ось так? А якщо ось так?" і одразу перевіряє. Результати дослідницького тестування можна документувати у вигляді ментальних карт або нотаток.

Припущення про помилку (Error guessing)

Ця техніка базується на досвіді тестувальника та його знанні типових помилок. Тестувальник передбачає, де можуть ховатися проблеми, і створює

тести саме для цих місць. Наприклад, знаючи, що розробники часто забувають перевіряти порожні поля, тестувальник обов'язково перевірить, що станеться, якщо залишити поле порожнім. Знаючи про ризик SQL-ін'єкцій, він спробує ввести в поле пошуку запит на кшталт ' OR '1'='1. З часом у кожного тестувальника формується власний набір "підозрілих місць", які він перевіряє насамперед.

Парне тестування (Pairwise Testing)

Окремо варто розглянути техніку парного тестування, яка займає проміжне місце між техніками чорного ящика та комбінаторними методами. Вона використовується, коли є багато параметрів з кількома значеннями кожен, і перевірити всі комбінації неможливо. Ідея парного тестування базується на спостереженні, що більшість дефектів викликані взаємодією не більше двох параметрів. Тому достатньо перевірити всі можливі пари значень параметрів, а не всі комбінації повністю.

Алгоритм простий: визначити параметри та їх допустимі значення, побудувати всі можливі пари значень, а потім створити мінімальний набір тестів, у якому кожна пара зустрічається хоча б один раз. Це дає драматичне скорочення кількості тестів порівняно з повним перебором.

Розглянемо приклад. Тестуємо конфігурацію: браузер (Chrome, Firefox), операційна система (Windows, macOS), мова (UA, EN). Повний перебір дав би 2*2*2 = 8 тестів. Парне тестування дозволяє отримати всього 4 тести, які покривають всі можливі пари значень:

Тест 1: Chrome, Windows, UA
Тест 2: Chrome, macOS, EN
Тест 3: Firefox, Windows, EN
Тест 4: Firefox, macOS, UA

Перевіримо: пара (Chrome, Windows) є в тесті 1, (Chrome, macOS) – в тесті 2, (Chrome, UA) – в тесті 1, (Chrome, EN) – в тесті 2, і так далі. Усі 12 можливих пар покриті. Це надзвичайно ефективна техніка для конфігураційного тестування, тестування API з параметрами, тестування складних форм.

Статичні техніки тестування

Окрему групу становлять статичні техніки, які не потребують виконання коду. Вони поділяються на рецензування (reviews) та статичний аналіз.

Рецензування – це ручна перевірка артефактів проекту (вимог, дизайну, коду, тест-кейсів) з метою виявлення дефектів. Існують різні види рецензування: walkthrough (неофіційний перегляд документа за участю автора), peer review (взаємна перевірка між колегами), inspection (найформальніший процес з чітко визначеними ролями, сценаріями та журналами дефектів).

Статичний аналіз – це автоматизована перевірка коду за допомогою спеціальних інструментів (лінтерів, аналізаторів). Вони знаходять потенційні помилки: невикористані змінні, недосяжний код, порушення стандартів кодування, потенційні вразливості. Метричний аналіз вимірює складність коду, довжину функцій, кількість гілок – це допомагає визначити, які частини коду є найскладнішими і потребують додаткової уваги.

Вибір техніки для конкретного сценарію

Тепер, коли ми знаємо основні техніки, постає питання: яку з них обрати в конкретній ситуації? Універсальної відповіді немає, але є загальні рекомендації, які допомагають зробити правильний вибір.

Перш за все, слід враховувати тип артефакту, що тестується. Для вимог найкраще підходять статичні техніки – рецензування. Для коду – статичний аналіз та техніки білого ящика. Для UI – техніки чорного ящика та дослідницьке тестування.

Далі – тип системи або функціональності. Якщо система має яскраво виражену станову поведінку (банкомат, замовлення), варто використовувати діаграми переходів станів. Якщо є багато комбінацій параметрів (конфігурації, фільтри), стане в нагоді парне тестування. Якщо є чітко визначені діапазони значень – еквівалентне розбиття та аналіз граничних значень. Якщо логіка залежить від багатьох умов – таблиці прийняття рішень.

Важливо враховувати доступні ресурси: час, людей, інструменти. Формальні техніки потребують більше часу на підготовку, але дають краще покриття. Дослідницьке тестування дозволяє швидко отримати результати, але покриття може бути нерівномірним.

Рівень ризику також впливає на вибір. Критичні модулі (безпека, платежі) мають тестуватися з використанням найпотужніших технік, включаючи комбіноване покриття умов і повне покриття шляхів. Для менш критичних модулів можна обмежитися простішими техніками.

І нарешті, рівень деталізації документації. Якщо є чіткі, детальні вимоги, можна застосовувати формальні техніки чорного ящика. Якщо вимоги розмиті або відсутні, доведеться покладатися на досвідчені евристики та дослідницьке тестування.

Зведемо рекомендації в таблицю:

Ситуація	Рекомендовані техніки
Чітко визначені діапазони значень	Еквівалентне розбиття, аналіз граничних значень
Комбінації параметрів	Парне тестування, комбінаторні техніки
Реакція системи на зміну стану	Діаграми переходів станів
Бізнес-логіка з умовами	Таблиця прийняття рішень
Тестування взаємодії користувача з	Use Case, сценарне тестування

системою	
Невизначеність, нові функції	Дослідницьке тестування, чек-листи
Автоматизація, перевірка коду	Статичний аналіз, code review, тестування API

Висновок

Техніки тест-дизайну – це потужний арсенал інструментів, який дозволяє тестувальнику перейти від інтуїтивного "тицяння" до системного, науково обґрунтованого підходу. Вони допомагають оптимізувати процес тестування, забезпечити максимальне покриття при мінімальних витратах, зробити тести повторюваними, документованими та придатними для автоматизації.

Ми розглянули основні техніки: еквівалентне розбиття, аналіз граничних значень, таблиці прийняття рішень, діаграми переходів станів, use case тестування, техніки білого ящика (покриття інструкцій, гілок, шляхів, умов), техніки, засновані на досвіді (ad-hoc, дослідницьке тестування, припущення про помилку), парне тестування та статичні техніки. Кожна з них має свою сферу застосування, свої переваги та обмеження.

Вміння обирати правильну техніку для конкретної ситуації приходить з досвідом, але знання теорії є необхідною базою. На наступних лекціях ми детальніше розглянемо деякі з цих технік, навчимося застосовувати їх на практиці та розв’язувати типові задачі тест-дизайну.

Питання для обговорення

1. Поясніть різницю між інтуїтивним та системним підходом до створення тестів. Чому системний підхід вважається більш професійним, незважаючи на те, що він вимагає більше часу на планування?
2. Порівняйте техніки еквівалентного розбиття (Equivalence Partitioning) та аналізу граничних значень (Boundary Value Analysis). Чому їх часто використовують разом, а не окремо?
3. Для яких типів задач найкраще підходить техніка таблиць прийняття рішень (Decision Table Testing)? Наведіть приклад бізнес-логіки, яку зручно тестувати саме за допомогою цієї техніки.
4. Як ви розумієте суть техніки тестування переходів станів (State Transition Testing)? Наведіть приклад системи, яка має яскраво виражену станову поведінку.
5. Чому покриття інструкцій (Statement Coverage) вважається недостатнім критерієм якості тестування коду? Які ризики залишаються при 100% покритті інструкцій?
6. Порівняйте техніки, засновані на досвіді (error guessing, exploratory testing), з формальними техніками чорного ящика. Чи можна вважати їх менш професійними?
7. Що таке парне тестування (pairwise testing) і для вирішення якої проблеми воно було створене? Чому воно особливо ефективне для конфігураційного тестування?

8. Як статичні техніки тестування (рев’ю, статичний аналіз коду) допомагають виявляти дефекти ще до запуску програми? Чому це вигідніше, ніж динамічне тестування?

9. Проаналізуйте ситуацію: ви тестуєте новий функціонал, вимоги до якого є дуже розмитими і постійно змінюються. Які техніки тест-дизайну ви оберете і чому?

10. Як ви розумієте твердження: "Тест-дизайн – це не просто набір технік, а спосіб мислення"? Що це означає на практиці?

Лекція 11
Техніки тест-дизайну. Поглиблене вивчення комбінаторних методів, діаграм станів та випадків використання

Вступ

На минулій лекції ми розпочали знайомство з техніками тест-дизайну, розглянули їхню класифікацію та основні представники кожної групи. Сьогодні ми продовжимо цю тему, але зосередимося на більш детальному вивченні окремих технік, які мають особливе значення для практичного тестування. Ми глибше розберемо парне тестування як метод боротьби з "комбінаційним вибухом", детально розглянемо побудову діаграм переходів станів та діаграм випадків використання, навчимося створювати на їх основі тестові сценарії. Також ми поговоримо про те, як тест-дизайн впливає на якість тестування, яких помилок слід уникати при виборі технік, і як інтегрувати ці техніки в різні види тестування, зокрема в регресійне. Наприкінці лекції ми порівняємо різні техніки за їхніми сильними та слабкими сторонами, а також розглянемо інструменти, які допомагають у створенні відповідних діаграм.

Вплив тест-дизайну на якість тестування

Перш ніж заглиблюватися в деталі окремих технік, важливо усвідомити, який саме вплив має грамотний тест-дизайн на загальну якість процесу тестування. Це не просто академічне знання, а практичний інструмент, який трансформує роботу тестувальника.

Перший і найочевидніший аспект – це повнота покриття. Застосування технік тест-дизайну дозволяє забезпечити максимальне охоплення функціоналу, різноманітних станів системи та можливих сценаріїв використання. Замість того щоб тестувати "як бог на душу покладе", тестувальник свідомо перевіряє всі класи еквівалентності, всі граничні значення, всі важливі комбінації параметрів, всі переходи між станами.

Другий аспект – точність. Добре спроектовані тести мінімізують кількість помилкових позитивних результатів (коли тест падає через помилку в самому тесті, а не в продукті) та помилкових негативних результатів (коли дефект є, але тест його не виявляє). Чітке визначення очікуваних результатів для кожного тесту робить процес оцінки однозначним.

Третій аспект – відтворюваність. Техніки тест-дизайну вимагають чіткого, структурованого опису тестів, що робить їх легко повторюваними будь-яким членом команди в будь-який час. Це критично важливо для регресійного тестування та для передачі знань у команді.

Четвертий аспект – релевантність. Техніки тест-дизайну орієнтовані на реальні ситуації та ризики. Вони допомагають зосередитися на тому, що дійсно важливо для користувача, а не на абстрактних теоретичних випадках.

П’ятий аспект – пріоритезація. Розуміння різних технік дозволяє свідомо обирати, які тести створити в першу чергу, зосереджуючись на критичних шляхах та найбільш ризикованих місцях.

Шостий аспект – придатність до автоматизації. Добре структуровані сценарії, створені на основі технік тест-дизайну, набагато легше автоматизувати, ніж набір випадкових ручних тестів.

І нарешті, сьомий аспект – аналіз дефектів. Коли тести чітко пов’язані з вимогами та модулями (за допомогою матриць трасування), знайдений дефект легко простежити до його джерела, що прискорює його виправлення.

Приклад ефективного застосування технік

Розглянемо гіпотетичний приклад, який демонструє ефективність застосування технік тест-дизайну. Уявімо, що ми тестуємо форму реєстрації з полями: вік (ціле число від 18 до 65), стать (чоловіча, жіноча, інше), країна (випадаючий список з 50 країн). Інтуїтивний підхід міг би призвести до створення, скажімо, 20-30 тестів, обраних випадково. Але чи будуть вони покривати всі важливі випадки?

Застосувавши техніки тест-дизайну, ми діємо системно. Для поля "вік" застосовуємо еквівалентне розбиття та аналіз граничних значень. Отримуємо тестові значення: 17, 18, 19, 30, 64, 65, 66. Для поля "стать" – всі три варіанти. Для поля "країна" – еквівалентне розбиття дає нам, скажімо, три представники: одну країну з початку списку, одну з середини, одну з кінця. Але ми також маємо врахувати комбінації цих параметрів. Застосувавши парне тестування, ми можемо отримати набір з 10-15 тестів, які покривають всі важливі пари значень, що значно ефективніше за повний перебір (3*50*7 = 1050 комбінацій). Цей приклад наочно показує, як техніки тест-дизайну дозволяють досягти високого покриття при обмеженій кількості тестів.

Помилки у виборі технік тестування

Навіть знаючи техніки, тестувальники можуть припускатися помилок у їх виборі. Найпоширеніша помилка – використання лише однієї, "улюбленої" техніки для всіх випадків. Не можна тестувати складну бізнес-логіку виключно еквівалентним розбиттям, ігноруючи таблиці рішень, або тестувати станову поведінку без діаграм переходів. Кожна техніка має свою сферу застосування.

Інша помилка – ігнорування комбінацій параметрів. Тестувальники перевіряють кожен параметр окремо, але не перевіряють, як вони взаємодіють. Це призводить до пропуску дефектів, які виникають лише при певних комбінаціях значень.

Третя помилка – надмірна деталізація там, де вона не потрібна. Не варто будувати повну таблицю рішень для тривіальної логіки, яка перевіряється двома-трьома тестами. Це марна трата часу.

Четверта помилка – відсутність пріоритезації. Тестувальник намагається застосувати всі техніки до всіх модулів, не враховуючи, що критичні модулі потребують глибшого покриття, ніж другорядні.

П’ята помилка – застосування технік білого ящика там, де вони недоречні, наприклад, до вимог або дизайну. І навпаки, ігнорування статичних технік, які можуть виявити дефекти ще до написання коду.

Тест-дизайн у регресійному тестуванні

Особливого значення техніки тест-дизайну набувають у контексті регресійного тестування. Регресійні набори тестів мають тенденцію з часом розростатися, стаючи важкими для підтримки та виконання. Саме тут техніки тест-дизайну допомагають оптимізувати набір, зберігаючи при цьому високе покриття.

По-перше, при створенні регресійного набору ми маємо обрати техніки, які дають максимальне покриття при мінімальній кількості тестів. Парне тестування часто є ідеальним вибором для конфігураційного регресійного тестування. Еквівалентне розбиття допомагає уникнути дублювання тестів, які перевіряють одне й те саме.

По-друге, техніки тест-дизайну допомагають підтримувати регресійний набір в актуальному стані. Коли з’являється нова функціональність, ми свідомо застосовуємо відповідні техніки для створення нових тестів, які додаються до регресійного набору.

По-третє, розуміння технік дозволяє аналізувати існуючий регресійний набір на предмет прогалин. Чи всі класи еквівалентності покриті? Чи всі важливі переходи між станами перевірені? Чи всі пари параметрів враховані? Такий аналіз допомагає постійно покращувати якість регресійного тестування.

Парне тестування (Pairwise Testing): поглиблений розгляд

На минулій лекції ми коротко ознайомилися з парним тестуванням. Сьогодні розглянемо його детальніше, оскільки це одна з найпотужніших технік для боротьби з "комбінаційним вибухом" – ситуацією, коли кількість можливих комбінацій параметрів стає астрономічною.

Парне тестування базується на спостереженні, що більшість дефектів викликані взаємодією не більше двох параметрів. Дефекти, які вимагають одночасної взаємодії трьох і більше параметрів, трапляються значно рідше. Тому, перевіривши всі можливі пари значень, ми з високою ймовірністю виявимо переважну більшість комбінаційних дефектів.

Алгоритм парного тестування складається з кількох кроків. Спочатку ми визначаємо всі параметри, які впливають на роботу системи, та всі можливі значення для кожного параметра. Потім ми генеруємо всі можливі пари значень. Далі ми будуємо мінімальний набір тестів (комбінацій значень всіх параметрів) такий, що кожна згенерована пара зустрічається хоча б в одному тесті.

У презентації наведено приклад з трьома параметрами A, B, C, кожен з яких може приймати значення Yes або No. Повний перебір дав би 2*2*2 = 8 тестів. Парне тестування дозволяє отримати 4 тести, які покривають всі можливі пари. Таблиця в презентації ілюструє, як саме пари (A,B), (A,C) та (B,C) покриваються різними тестами.

Парне тестування особливо ефективне для конфігураційного тестування (різні браузери, ОС, версії), для тестування API з багатьма параметрами, для тестування складних форм з багатьма полями. Існують спеціальні інструменти (наприклад, PICT від Microsoft, AllPairs), які автоматизують генерацію наборів тестів за методом pairwise.

Діаграми переходів станів (State Transition Diagram)

Перейдемо до детального розгляду діаграм переходів станів. Це графічне представлення того, як система змінює свій стан у відповідь на різні події. Основні компоненти такої діаграми: стан (state) – це поточне положення системи (наприклад, "Авторизований", "Очікування підтвердження", "Виконано"); подія (event) – це те, що відбувається і спричиняє перехід (дія користувача, сигнал від системи, спрацювання таймера); перехід (transition) – це зміна від одного стану до іншого у відповідь на подію; початковий стан позначається стрілкою без початку; кінцевий стан позначається подвійним кружечком.

У презентації наведено приклад для банкомату. Початковий стан – "Очікування логіну". При введенні коректних даних відбувається перехід до стану "Авторизований". При введенні невірного пароля – перехід до стану "Попередження 1". З цього стану при повторному невірному паролі – перехід до "Попередження 2", а з нього при ще одній невдалій спробі – перехід до стану "Блокування". Це простий, але дуже наочний приклад.

Інший приклад – стан кошика в інтернет-магазині. Початковий стан – "Кошик порожній". Подія "Додано товар" переводить систему в стан "Товар у кошику". З цього стану подія "Натиснуто ‘Оформити замовлення’" переводить в стан "Очікування оплати". Після підтвердження оплати – стан "Замовлення сформовано". А з цього стану можна перейти в стан "Замовлення скасовано" за подією "Натиснуто ‘Скасувати’".

Для створення тестів на основі діаграми станів ми маємо перевірити кожен перехід: як валідні (наприклад, з коректними даними), так і невалідні (спроба переходу, який не дозволений). Також важливо перевірити, що система не може опинитися в невизначеному стані, і що з кожного стану можна вийти (або принаймні є чітко визначена поведінка).

Діаграми випадків використання (Use Case Diagram)

Інший важливий тип діаграм – діаграми випадків використання. Вони описують функціональні вимоги до системи з точки зору зовнішніх акторів. Основні елементи: актор (actor) – це зовнішній учасник, який взаємодіє з

системою (користувач, адміністратор, зовнішня система); варіант використання (use case) – це конкретна функція або поведінка, яку система виконує на запит актора; межа системи (system boundary) – прямокутник, який охоплює всі варіанти використання, що належать системі; зв'язки – лінії, що з'єднують акторів з варіантами використання.

Особливу увагу слід звернути на типи зв'язків між варіантами використання. Зв'язок "include" (включення) позначає, що один варіант використання завжди включає інший як обов'язкову частину. Наприклад, варіант використання "Оформити замовлення" може включати "Авторизуватися", якщо для оформлення замовлення потрібна авторизація. Зв'язок "extend" (розширення) позначає необов'язковий або додатковий сценарій, який виконується при певних умовах. Наприклад, "Оформити замовлення" може розширюватися варіантом "Застосувати промокод" – це додаткова опція, яка не є обов'язковою.

Сценарій використання для тестування (Use Case Scenario)

На основі діаграми випадків використання створюються конкретні сценарії для тестування. Сценарій використання – це деталізований опис одного конкретного шляху реалізації варіанта використання.

Структура сценарію включає: унікальну назву та ідентифікатор; акторів, які беруть участь; попередні умови – що має бути виконано до початку сценарію; основний потік подій – покроковий опис взаємодії, що веде до успіху; альтернативні потоки – відгалуження, якщо дії йдуть не за планом; післяумови – стан системи після завершення сценарію; виключення та помилки – що відбувається у випадку помилки.

У презентації наведено приклад сценарію "Успішна авторизація користувача". Попередні умови: користувач зареєстрований, має логін і пароль. Основний потік: відкрити сторінку входу, ввести логін і пароль, натиснути кнопку "Увійти", система перевіряє дані, користувач переходить до головної сторінки акаунту. Альтернативні потоки: неправильний пароль (система показує повідомлення про помилку), порожні поля (система видає помилку валідації). Післяумови: користувач авторизований, сесія створена.

Такі сценарії є основою для створення конкретних тест-кейсів. Вони допомагають переконатися, що система підтримує всі важливі бізнес-процеси.

Порівняння технік тест-дизайну

У презентації наведено дві таблиці порівняння різних технік за їхніми сильними та слабкими сторонами, а також рекомендаціями щодо застосування. Розглянемо основні висновки.

Еквівалентне розбиття є простим і ефективним для зменшення кількості тестів, але не покриває граничні значення. Тому його слід використовувати разом з аналізом граничних значень, який добре виявляє помилки на межах діапазону, але не охоплює середину.

Таблиці прийняття рішень чудово візуалізують логіку і дають повне покриття варіантів, але ускладнюються при великій кількості умов. Вони незамінні для систем зі складною логікою типу "якщо-то".

Парне тестування ефективно зменшує комбінаційну вибуховість і добре підходить для конфігураційного тестування, але не гарантує виявлення всіх дефектів, особливо тих, що залежать від взаємодії трьох і більше параметрів.

Діаграми переходів станів добре описують поведінку системи і дозволяють знаходити нелегальні переходи, але складні в побудові для великих систем. Вони ідеальні для тестування життєвих циклів сутностей, workflow, авторизацій.

Use Case сценарії орієнтовані на бізнес-логіку і відображають реальні сценарії використання, але не деталізують внутрішню логіку системи. Вони найкраще підходять для end-to-end та функціонального тестування.

Статичне тестування дозволяє виявляти дефекти до запуску коду, є швидким і економним, але не виявляє динамічні дефекти і залежить від навичок рецензента.

Яку діаграму обрати?

Окреме питання – вибір між Use Case діаграмами та State Transition діаграмами. Вони мають різне призначення. Use Case діаграми фокусуються на зовнішній взаємодії з системою – що робить користувач. Вони орієнтовані на бізнес-процеси та функціональність, відображають точку зору користувача і використовуються для end-to-end та функціонального тестування. Вони відповідають на питання "що система повинна робити?".

State Transition діаграми, навпаки, фокусуються на внутрішньому стані системи – як вона реагує на події. Вони орієнтовані на поведінку системи, відображають точку зору самої системи і використовуються для тестування динамічної поведінки, станів та послідовності подій. Вони відповідають на питання "як система реагує на зміну подій або вхідних даних?".

У реальному проекті використовуються обидва типи діаграм, оскільки вони доповнюють одна одну, даючи повну картину системи.

Інструменти для побудови діаграм

Для створення діаграм існує багато інструментів, кожен з яких має свої переваги та недоліки. У презентації наведено порівняльну таблицю.

Lucidchart – це онлайн-інструмент з дуже зручним інтерфейсом, шаблонами UML та можливістю співпраці в реальному часі. Він працює за моделлю freemium: безкоштовна версія має обмеження. Він найкраще підходить для створення UML, BPMN, State Transition діаграм та flowchart.

Draw.io (тепер diagrams.net) – це безкоштовний інструмент, який може працювати онлайн або як десктопний додаток. Він інтегрується з Google Drive, GitHub, працює офлайн. Його недоліки – менш привабливий дизайн і менше шаблонів, але для більшості задач він цілком достатній.

Creately – ще один онлайн-інструмент з шаблонами UML та бізнес-діаграм, з можливістю колаборації. Він також freemium, але платна версія має багато обмежень, і іноді працює повільно.

PlantUML – це унікальний інструмент, який генерує діаграми з текстового опису. Це дозволяє автоматизувати створення діаграм, інтегрувати їх з CI/CD, зберігати в репозиторіях разом з кодом. Недолік – потрібно знати спеціальний синтаксис, і менше можливостей для візуальної кастомізації. Він ідеально підходить для автоматизації та документації в репозиторіях.

Вибір інструменту залежить від потреб команди, бюджету та особистих уподобань.

Висновок

Сьогодні ми поглибили наші знання про техніки тест-дизайну, розглянувши детально парне тестування, діаграми переходів станів та діаграми випадків використання. Ми побачили, як ці техніки впливають на якість тестування, забезпечуючи повноту, точність, відтворюваність та релевантність тестів. Ми навчилися уникати типових помилок при виборі технік і застосовувати їх у регресійному тестуванні. Ми також порівняли різні техніки за їхніми сильними та слабкими сторонами, що допоможе нам у майбутньому свідомо обирати найкращий інструмент для кожної конкретної ситуації. Нарешті, ми ознайомилися з інструментами, які полегшують створення відповідних діаграм. Техніки тест-дизайну – це не просто теорія, а потужний практичний інструмент, який робить тестування системним, ефективним і професійним. На наступних заняттях ми продовжимо застосовувати ці знання на практиці, розв’язуючи конкретні задачі та створюючи тестові набори для реальних сценаріїв.

Питання для обговорення

1. Чому парне тестування (pairwise testing) не гарантує виявлення всіх дефектів, пов’язаних із взаємодією параметрів? Які дефекти воно може пропустити?
2. Поясніть різницю між діаграмою переходів станів (State Transition Diagram) та діаграмою випадків використання (Use Case Diagram). У яких випадках застосовується кожна з них?
3. Проаналізуйте структуру сценарію використання для тестування (Use Case Scenario). Чому важливо описувати не лише основний, але й альтернативні потоки подій?
4. Наведіть приклад зв’язку "include" та "extend" між варіантами використання. У чому принципова різниця між цими типами зв’язків?
5. Як діаграми переходів станів допомагають знаходити "нелегальні" переходи або відсутність обробки помилкових ситуацій у програмі?
6. Порівняйте сильні та слабкі сторони таблиць прийняття рішень та парного тестування. Для яких типів задач кожна з цих технік є оптимальною?

7. Чому діаграми випадків використання вважаються орієнтованими на бізнес-процеси, а діаграми переходів станів – на поведінку системи? Як це впливає на вибір техніки тестування?

8. Які інструменти для побудови діаграм ви знаєте? Які переваги та недоліки текстових інструментів (наприклад, PlantUML) порівняно з графічними (Lucidchart, Draw.io)?

9. Уявіть, що ви тестуєте систему бронювання авіаквитків. Які техніки тест-дизайну ви застосуєте для перевірки логіки пошуку рейсів, а які – для перевірки процесу оплати?

10. Як ви розумієте твердження: "Жодна техніка тест-дизайну не є універсальною, і професійний тестувальник має володіти всім арсеналом"? Чому важливо знати різні техніки, а не одну-дві?

Лекція 12
Тестування API: від основ до практичного застосування

Вступ

Протягом попередніх лекцій ми вивчали тестування на рівні користувацького інтерфейсу – ми натискали кнопки, заповнювали форми, перевіряли відображення елементів. Однак сучасна архітектура програмного забезпечення рідко обмежується одним монолітним додатком. Сьогодні ми живемо у світі мікросервісів, хмарних обчислень та інтеграцій з десятками зовнішніх систем. У центрі цієї складної екосистеми знаходяться API – інтерфейси програмування додатків, які забезпечують взаємодію між різними компонентами. Тестування API є критично важливим, оскільки дозволяє виявляти дефекти на ранніх стадіях, перевіряти бізнес-логіку без прив'язки до графічного інтерфейсу, забезпечувати безпеку та продуктивність системи. Сьогодні ми розглянемо основи клієнт-серверної архітектури, протокол HTTP, формати обміну даними, а також навчимося працювати з Postman – найпопулярнішим інструментом для тестування API.

Роль API у взаємодії між компонентами

Що таке API? Application Programming Interface – це набір правил та механізмів, за допомогою яких одна програма може взаємодіяти з іншою. Уявіть собі ресторан: ви, як відвідувач, не заходите на кухню і не готуєте їжу самостійно. Ви користуєтеся меню – це і є інтерфейс. Ви робите замовлення, а офіціант передає його на кухню. Через деякий час вам приносять готову страву. API працює схожим чином: клієнтська програма (наприклад, мобільний додаток) надсилає запит до сервера, використовуючи визначений інтерфейс, і отримує відповідь з необхідними даними.

У сучасній архітектурі API виконують роль "клею", що з'єднує різні частини системи. Фронтенд (веб-сторінка або мобільний додаток) спілкується з бекендом через API. Бекенд, у свою чергу, може звертатися до API інших сервісів – платіжних систем, баз даних, зовнішніх постачальників послуг. Така модульність дозволяє розробляти, тестувати та масштабувати компоненти незалежно один від одного.

Для тестувальника API відкриває унікальні можливості. Тестування на рівні API є швидшим і стабільнішим, ніж тестування через UI. Воно дозволяє перевіряти бізнес-логіку без зайвих накладних витрат, легко автоматизується і може бути інтегроване в CI/CD-процеси. Крім того, API-тести виконуються на більш ранніх етапах розробки, що дозволяє виявляти дефекти до того, як вони потраплять у користувацький інтерфейс.

Основи клієнт-серверної архітектури

Клієнт-серверна архітектура – це модель взаємодії, в якій клієнт надсилає запити, а сервер їх обробляє і повертає відповіді. Клієнтом може бути веб-браузер, мобільний додаток, десктопна програма або навіть інший сервер. Сервер – це програма, яка постійно очікує на запити, обробляє їх і надає необхідні ресурси.

У контексті веб-розробки найпоширенішою є архітектура на основі HTTP. Клієнт надсилає HTTP-запит на певну URL-адресу, використовуючи один з методів HTTP. Сервер обробляє запит і повертає HTTP-відповідь, яка містить статус виконання та, можливо, дані у певному форматі. Наприклад, GET-запит до https://example.com/api/products може повернути список продуктів у форматі JSON.

Протоколи HTTP/HTTPS

HTTP (Hypertext Transfer Protocol) – це протокол прикладного рівня, який є основою обміну даними в Всесвітній павутині. HTTPS (HTTP Secure) – це захищена версія протоколу, яка використовує шифрування TLS/SSL для забезпечення конфіденційності та цілісності даних.

Ключовим поняттям HTTP є методи запитів, які визначають операцію, яку клієнт хоче виконати над ресурсом. Найпоширеніші методи: GET використовується для отримання ресурсу без зміни даних на сервері; POST – для створення нового ресурсу; PUT – для повної заміни існуючого ресурсу; PATCH – для часткового оновлення ресурсу; DELETE – для видалення ресурсу.

Важливою характеристикою методів є ідемпотентність. Ідемпотентний метод – це метод, який при багаторазовому виконанні з однаковими параметрами дає той самий результат, що й при одноразовому виконанні, і не викликає додаткових побічних ефектів. Наприклад, GET, PUT і DELETE є ідемпотентними. Якщо ви відправите DELETE-запит двічі, після першого видалення ресурс вже буде відсутній, але другий запит просто поверне помилку, не створивши нового побічного ефекту. POST, навпаки, не є ідемпотентним – повторний POST-запит створить новий ресурс (наприклад, дублікат замовлення).

У презентації наведено таблицю, яка показує відповідність HTTP-методів операціям CRUD (Create, Read, Update, Delete). POST відповідає Create, GET – Read, PUT/PATCH – Update, DELETE – Delete. Це стандартна модель для RESTful API.

Структура та призначення JSON

JSON (JavaScript Object Notation) – це легкий, текстовий формат обміну даними, який став стандартом де-факто для сучасних веб-API. Він заснований на синтаксисі об’єктів JavaScript, але є мовонезалежним і підтримується практично всіма мовами програмування.

JSON підтримує кілька типів даних: рядки (завжди в подвійних лапках), числа (цілі або з плаваючою точкою), логічні значення (true або false), null (відсутність значення), об’єкти (невпорядковані колекції пар ключ-значення в фігурних дужках) та масиви (упорядковані списки значень у квадратних дужках).

Розглянемо приклад POST-запиту, який створює нового користувача:

```
POST /api/users HTTP/1.1
Content-Type: application/json

{
"username": "test_user",
"email": "test@example.com",
"active": true
}
```

Тіло запиту містить JSON-об’єкт з трьома полями: username (рядок), email (рядок) та active (логічне значення). Заголовок Content-Type повідомляє серверу, що тіло запиту містить дані у форматі JSON.

Формат JSON vs. XML

До появи JSON стандартним форматом обміну даними був XML (eXtensible Markup Language). Хоча XML все ще використовується в деяких системах (особливо в корпоративному середовищі), JSON став набагато популярнішим завдяки своїй простоті та легкості.

Порівняймо основні відмінності. JSON є форматом обміну даними, орієнтованим на дані, тоді як XML є мовою розмітки, орієнтованою на документи. JSON має простіший, більш читабельний синтаксис і менший обсяг, що прискорює передачу даних. JSON нативно підтримує такі типи даних, як числа, логічні значення та масиви, тоді як в XML все є текстом, і ці типи доводиться емулювати.

Подивімося на приклад одного й того ж об’єкта в JSON та XML:

```
JSON:
{
"product": {
"id": 1,
"name": "VR Headset",
"price": 299.99
}
}

XML:
<product>
```

```
<id>1</id>
<name>VR Headset</name>
<price>299.99</price>
</product>
```

JSON є більш компактним і легшим для сприйняття людиною. Крім того, JSON має нативну підтримку масивів, тоді як в XML масиви емулюються через повторювані теги. Це робить JSON природним вибором для сучасних веб-API.

Створення JSON у різних мовах програмування

У презентації наведено приклади створення JSON у Python та JavaScript, що демонструє універсальність формату.

У Python ми створюємо словник (dict) з необхідними даними, а потім використовуємо модуль json для запису цього словника у файл. Параметр indent=2 забезпечує читабельне форматування з відступами.

```
import json

user = {
"id": 102,
"name": "Ivan Shevchenko",
"email": "ivan@example.com"
}

with open('user.json', 'w', encoding='utf-8') as f:
json.dump(user, f, indent=2)
```

У JavaScript (Node.js) ми створюємо об'єкт, а потім використовуємо JSON.stringify() для перетворення його в рядок JSON. Параметри null та 2 також забезпечують форматування.

```
const fs = require('fs');

const user = {
id: 103,
name: "Oksana Koval",
email: "oksana@example.com"
};

fs.writeFileSync('user.json', JSON.stringify(user, null, 2));
```

Ці приклади показують, як легко працювати з JSON у різних мовах, що робить його ідеальним вибором для міжмовної взаємодії.

Вступ до Postman

Postman – це найпопулярніший інструмент для тестування API, який поєднує в собі зручний графічний інтерфейс, потужні можливості для

створення та організації запитів, автоматизації тестування та документування API. Для тестувальника Postman є незамінним помічником у щоденній роботі.

Розглянемо основні компоненти інтерфейсу Postman. Request Builder – це область, де ви будуєте запит: обираєте метод (GET, POST тощо), вводите URL, додаєте параметри, заголовки та тіло запиту. Tabs – кожен запит відкривається в окремій вкладці, що дозволяє працювати з кількома запитами одночасно. Response Viewer – після відправки запиту тут відображається відповідь сервера: тіло відповіді, статус, час виконання, заголовки. Sidebar – це структурована панель, де ви можете створювати колекції запитів, переглядати історію, керувати середовищами. Collections – це групи збережених запитів, організовані за темами або проектами. Environments – це набори змінних (наприклад, base URL, токени), які можна перемикати залежно від середовища (dev, test, prod). Console – це дебаг-консоль для перегляду логів та console.log()-виводів, що допомагає при налагодженні складних запитів.

Перевірка відповідей сервера: коди статусу HTTP

Кожна HTTP-відповідь містить тризначний код статусу, який інформує клієнта про результат виконання запиту. Коди згруповані в п’ять класів.

Клас 1xx (100-199) – інформаційні коди, які повідомляють, що запит прийнято і очікується подальша дія. Вони рідко зустрічаються в повсякденному тестуванні.

Клас 2xx (200-299) – успішне виконання запиту. Найпоширеніші: 200 OK (запит успішно виконано, є відповідь), 201 Created (ресурс створено, зазвичай після POST-запиту), 204 No Content (успіх, але тіло відповіді відсутнє, часто використовується після DELETE).

Клас 3xx (300-399) – перенаправлення. Клієнту потрібно виконати додаткові дії для завершення запиту, наприклад, перейти за іншою адресою.

Клас 4xx (400-499) – помилки на стороні клієнта. Це означає, що запит був сформований неправильно або містить некоректні дані. Найпоширеніші: 400 Bad Request (некоректний запит, синтаксична помилка, відсутні обов’язкові параметри), 401 Unauthorized (потрібна авторизація), 403 Forbidden (доступ заборонено, навіть після авторизації), 404 Not Found (ресурс не знайдено), 405 Method Not Allowed (неприпустимий HTTP-метод), 429 Too Many Requests (перевищено ліміт запитів).

Клас 5xx (500-599) – помилки на стороні сервера. Це означає, що сервер не зміг обробити запит через внутрішню проблему. Найпоширеніші: 500 Internal Server Error (загальна помилка сервера), 502 Bad Gateway (некоректна відповідь від проміжного сервера), 503 Service Unavailable (сервіс недоступний, зазвичай через технічне обслуговування або перевантаження), 504 Gateway Timeout (тайм-аут при відповіді від проміжного сервера).

Для тестувальника розуміння кодів статусу є обов’язковим. Вони допомагають швидко діагностувати проблему: чи то помилка в запиті (4xx), чи то проблема на сервері (5xx).

Використання environment variables у Postman

Однією з найпотужніших функцій Postman є змінні середовища (environment variables). Вони дозволяють параметризувати запити, роблячи їх гнучкими та легкими в підтримці. Замість того щоб жорстко прописувати URL-адресу або токен авторизації в кожному запиті, ви використовуєте змінну, наприклад {{base_url}} або {{auth_token}}. Значення цих змінних визначаються в поточному середовищі.

Postman підтримує кілька типів змінних. Global змінні доступні у всіх середовищах та колекціях. Environment змінні прив'язані до певного середовища (наприклад, dev, test, prod) – перемикаючи середовище, ви автоматично змінюєте значення всіх змінних. Collection змінні прив'язані до певної колекції запитів. Local змінні діють тільки в межах одного запиту або скрипту. Data змінні імпортуються через CSV або JSON під час запуску тестів через Collection Runner.

Така система змінних дозволяє, наприклад, мати один набір запитів, які можна запускати на різних середовищах просто перемикаючи профіль. Це особливо цінно при тестуванні, коли ви хочете перевірити API на dev-сервері, а потім ті самі тести на stage-сервері.

Автоматизація запитів у Postman (колекції)

Колекції в Postman – це спосіб організувати пов'язані запити в групи. Вони є основою для автоматизації тестування API.

Переваги використання колекцій численні. По-перше, повторне використання: ви можете запускати ті самі тести з різними даними, використовуючи різні середовища або імпортуючи тестові дані з файлів. По-друге, організованість: запити структуровані, згруповані логічно, що полегшує навігацію та підтримку. По-третє, автоматизація: колекції легко запускати вручну, а також через командний рядок за допомогою Newman, що дозволяє інтегрувати API-тести в CI/CD-процеси. По-четверте, масштабування: колекції добре підходять для тестування великих REST API з сотнями ендпоінтів. По-п'яте, документація: колекції можна перетворити в інтерактивну API-документацію, яка буде завжди синхронізована з тестами.

Робота з API-документацією

Для ефективного тестування API необхідно вміти читати та розуміти API-документацію. Хороша документація містить кілька ключових компонентів. Базовий URL – наприклад, https://api.example.com/v1 – це коренева адреса, до якої додаються шляхи до конкретних ресурсів. Методи запитів – GET, POST, PUT, DELETE тощо – для кожного ендпоінта. Ендпоінти – шляхи до ресурсів, наприклад /users або /users/{id}. Параметри – опис всіх можливих параметрів: query-параметрів (в URL після знаку питання), path-параметрів (частина URL, як {id} вище), header-параметрів (в заголовках запиту), body-параметрів (в тілі

запиту для POST/PUT). Приклади запитів – JSON-приклади тіла запиту, заголовків, відповідей. Коди статусів – очікувані коди для різних сценаріїв (200, 201, 400, 404 тощо). Аутентифікація – опис того, як саме потрібно автентифікуватися (OAuth2, API-ключі, JWT). Опис помилок – визначення помилок із кодами та поясненнями, що допомагає розуміти, чому запит не виконався.

Вміння працювати з документацією є ключовою навичкою тестувальника API. Часто документація є єдиним джерелом інформації про те, як правильно використовувати API.

Основи тестування API для продуктивності

Окрім функціонального тестування, API підлягають тестуванню продуктивності. Існує кілька видів тестування продуктивності, кожен з яких має свою мету.

Load Testing (тестування навантаженням) перевіряє, як API працює при очікуваній кількості користувачів. Ми створюємо навантаження, яке імітує нормальний робочий день, і вимірюємо час відповіді, кількість помилок, використання ресурсів.

Stress Testing (стрес-тестування) перевіряє, що станеться, якщо навантаження значно перевищить норму. Мета – знайти межу, після якої API починає деградувати, і зрозуміти, як він відновлюється після падіння.

Spike Testing перевіряє реакцію API на раптове, різке зростання трафіку. Це важливо для систем, які можуть зазнавати різких сплесків активності.

Soak Testing (тестування витривалості) перевіряє, як API функціонує протягом тривалого часу без збоїв. Це допомагає виявити проблеми з витоком пам’яті або поступовою деградацією продуктивності.

Scalability Testing перевіряє, чи може API масштабуватись при зростанні навантаження. Наприклад, чи допомагає додавання нових серверів обробляти більше запитів.

При тестуванні продуктивності ми відстежуємо кілька ключових метрик. Response Time – час від моменту запиту до отримання повної відповіді. Latency – затримка передачі даних, пов’язана з мережею. Error Rate – частка невдалих запитів (наприклад, з кодами 5xx). Throughput – кількість запитів за секунду (RPS). CPU/Memory Usage – ресурси, що використовуються сервером.

Висновок

Тестування API є невід’ємною частиною сучасної розробки програмного забезпечення. Воно дозволяє перевіряти бізнес-логіку, безпеку та продуктивність системи на ранніх етапах, незалежно від графічного інтерфейсу. Розуміння основ клієнт-серверної архітектури, протоколу HTTP, методів запитів, кодів статусу та форматів обміну даними є обов’язковим для будь-якого тестувальника.

Postman надає потужний, але зручний інструментарій для роботи з API. Вміння створювати запити, організовувати їх у колекції, використовувати змінні середовища, автоматизувати тести та аналізувати результати – це навички, які високо цінуються на ринку праці.

Нарешті, розуміння тестування продуктивності API дозволяє оцінювати не тільки коректність роботи системи, але й її здатність витримувати реальні навантаження. Це особливо важливо для високонавантажених систем, де навіть мілісекунди затримки можуть коштувати мільйонів.

На наступних заняттях ми поглибимо наші знання про тестування API, розглянемо написання автоматизованих тестів у Postman, інтеграцію з CI/CD, а також познайомимося з іншими інструментами, такими як Swagger та SoapUI.

Питання для обговорення

1. Поясніть роль API у взаємодії між різними компонентами сучасних програмних систем. Чому тестування на рівні API вважається більш надійним і швидким, ніж тестування через UI?
2. Порівняйте основні методи HTTP-запитів: GET, POST, PUT, PATCH, DELETE. Який з них не є ідемпотентним і чому це важливо?
3. Які переваги має формат JSON перед XML для обміну даними в сучасних веб-API? Чи є ситуації, коли XML все ще є кращим вибором?
4. Проаналізуйте класи HTTP-статусів (2xx, 4xx, 5xx). Яку інформацію для діагностики проблеми дає тестувальнику код відповіді 404, а яку – код 500?
5. Як використання змінних середовища (environment variables) у Postman допомагає організувати процес тестування API на різних середовищах (dev, test, prod)?
6. Чому колекції Postman вважаються зручним інструментом не лише для ручного тестування, але й для автоматизації та документування API?
7. Які компоненти має містити якісна API-документація, щоб тестувальник міг ефективно з нею працювати?
8. Поясніть різницю між навантажувальним тестуванням (load testing) та стрес-тестуванням (stress testing) API. Які цілі переслідує кожен з цих видів тестування?
9. Як ви розумієте метрики Response Time, Latency, Error Rate та Throughput? Про що може свідчити високий Error Rate при стабільному Response Time?
10. Уявіть, що вам потрібно протестувати API інтернет-магазину. Які основні сценарії ви перевірите в першу чергу, і які HTTP-методи для цього використаєте?

ПІСЛЯМОВА

Шановний читачу! Ось ми й дійшли до завершення нашого спільного шляху дванадцятьма лекціями, присвяченими тестуванню програмного забезпечення. Сподіваюся, що цей посібник став для вас не просто збіркою теоретичних матеріалів, а справжнім дороговказом у професію, яка сьогодні є однією з найдинамічніших, найцікавіших і найзатребуваніших у світі інформаційних технологій.

Ми розпочали з найголовнішого – з розуміння того, чому тестування взагалі потрібне і яку роль воно відіграє в сучасному світі. Ми побачили, що якість програмного забезпечення – це не абстрактне поняття, а конкретна, вимірювана характеристика, яка впливає на безпеку, зручність, надійність та успішність продукту. Ми дізналися, що тестувальник – це не просто "шукач багів", а повноцінний член команди, який бере участь у створенні продукту з перших днів і до самого релізу.

Протягом курсу ми поступово заглиблювалися в різні аспекти професії. Вивчили життєвий цикл розробки та методології, які визначають, як саме працює команда. Навчилися аналізувати вимоги – адже неможливо якісно протестувати те, чого не розумієш. Опанували класифікацію видів тестування, щоб розуміти, який саме інструмент застосовувати в кожній конкретній ситуації. Навчилися створювати тестову документацію – чек-листи, тест-кейси, баг-репорти – і зрозуміли, чому якісна документація є ознакою професіоналізму.

Особливу увагу ми приділили технікам тест-дизайну. Саме вони відрізняють інтуїтивне, "любительське" тестування від справжнього професійного підходу. Еквівалентне розбиття, аналіз граничних значень, таблиці прийняття рішень, діаграми переходів станів, парне тестування – ці техніки дозволяють досягати максимального покриття при мінімальних витратах, робити процес тестування системним, керованим і передбачуваним.

Насамкінець ми зазирнули у світ API-тестування – сферу, яка сьогодні розвивається особливо стрімко і відкриває перед тестувальниками нові горизонти. Ми познайомилися з Postman, навчилися працювати з HTTP-запитами, аналізувати коди статусів, розуміти структуру JSON.

Але найголовніше, що я хотів би, аби ви винесли з цього курсу – це не просто набір фактів і технік, а особливий спосіб мислення. Мислення тестувальника – це постійне прагнення ставити під сумнів, шукати приховане, передбачати непередбачуване. Це вміння дивитися на продукт очима користувача, аналізувати, де саме може критися проблема, і не заспокоюватися, доки не знайдеш відповідь.

Пам’ятайте: справжній професіонал ніколи не припиняє вчитися. Технології змінюються, з’являються нові інструменти, нові методології, нові види тестування. Те, що сьогодні є передовим, завтра може стати застарілим. Тому найважливіша навичка, яку ви маєте розвивати – це вміння самостійно опановувати нове, адаптуватися до змін і постійно вдосконалюватися.

Щиро дякуємо вам за те, що ви пройшли цей шлях разом з нами. Сподіваємось, що знання, отримані під час вивчення цього курсу, стануть міцним фундаментом для вашої успішної кар’єри в ІТ. Бажаємо вам цікавих проектів, розумних колег, вдячних користувачів і, звісно, якомога менше критичних багів!

Пам’ятайте: якість починається з вас. Саме ви творите світ, у якому технології служать людям, а не завдають їм клопоту. Пишайтеся своєю професією і розвивайте її!

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Авраменко А. С., Авраменко В. С., Косенюк Г. В. Тестування програмного забезпечення : навчальний посібник. Черкаси : ЧНУ імені Богдана Хмельницького, 2017. 284 с.URL : https://eprints.cdu.edu.ua/1482/1/testyvan.pdf
2. Золотухіна О. А., Негоденко О. В., Резник С. Ю., Разіна С. Я. Якість та тестування інформаційних систем : навчальний посібник. Київ : ННІІТ ДУТ, 2020. 128 с.
3. Повне керівництво з тестування графічного інтерфейсу: Підручник з тестування інтерфейсу користувача. URL: https://uk.myservername.com/gui-testing-tutorial
4. Смагіна О. О., Переяславська С. О. Якість програмного забезпечення та тестування : навч. посіб. до вивчення дисц. для студ. спец. 121 – «Інженерія програмного забезпечення» / Держ. закл. «Луган. нац. ун-т імені Тараса Шевченка». Старобільськ : ДЗ «ЛНУ імені Тараса Шевченка», 2021. 286 с. URL: https://dspace.luguniv.edu.ua/xmlui/bitstream/handle/123456789/7508/2021.pdf?sequence=5&isAllowed=y
5. Трофименко О. Г., Дика А. І. Тестування та забезпечення якості програмних систем : навч. посіб. для підгот. здобув. вищої освіти галузі знань 12 «Інформаційні технології». Одеса : Фенікс, 2024. 195 с. URL: https://doi.org/10.32837/11300.27717
6. Цибульник С. О., Барандич К. С. Технології розроблення програмного забезпечення. Частина 1. Життєвий цикл програмного забезпечення : підручник. Київ : КПІ ім. Ігоря Сікорського 2022. 270 с. URL: https://ela.kpi.ua/bitstream/123456789/50623/1/TRPZ_Ch1_ZhTsPZ.pdf
7. Якість програмного забезпечення та тестування: базовий курс : навчальний посібник / за ред. С. Я. Крепич, І. Я. Співак. Тернопіль : ФОП Паляниця В.А., 2020. 478с. URL: https://surl.lu/gdmifk

Навчальне видання

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Конспект лекцій

Укладачі: **Пархоменко** Олександр Юрійович
Тищенко Світлана Іванівна
Ємельянов Святослав Ігорович
Жебко Олександр Олегович
Богатєнкова Олександра Євгенівна

Формат 60х84 1/16. Ум. друк. арк. 4,0.
Тираж 20 прим. Зам. № ____

Надруковано у видавничому відділі
Миколаївського національного аграрного університету
54008, м. Миколаїв, вул. Георгія Гонгадзе, 9

Свідоцтво суб'єкта видавничої справи ДК № 4490 від 20.02.2013 р.