Лекція 1
Основні поняття та визначення предметної області UX- тестування

1.1. Суть людино-орієнтованого проектування

Питання про роль та місце психології у програмуванні стали виникати багато років тому, при становленні програмування як професійної діяльності. Наскільки справедливо розглядати розробку програмних продуктів лише з погляду обчислювальної математики?

В даний час переважає тверде переконання, що програмування – це рід людської діяльності, де люди створюють програми виключно для людей. Комп'ютер використовується лише як інструмент. Звідси випливає, що розробка програмного забезпечення, зокрема мультимедійних продуктів, має розглядатися з позицій людино-машинної системи.

Таким чином *людино-орієнтоване проектування (Human Centered Design, HCD) — це підхід до розробки різних продуктів та систем, який на перше місце ставить користувача з його цілями, завданнями, обмеженнями та контекстом роботи.*

Формальний опис HCD наведено в стандарті ISO 9241-210. Стандарт описує, як використовувати HCD для створення комп'ютеризованих інтерактивних систем. Приклади таких систем - інтернет-сайти, комп'ютерні програми та мобільні програми, інтерфейси банкоматів та терміналів самообслуговування, а також самі пристрої (комп'ютери, телефони, банкомати тощо). Однак насправді сфера застосування HCD набагато ширша, ніж зазначено у стандарті. HCD підходить для створення практично будь-яких продуктів, сервісів чи послуг.

Процес HCD складається з п'яти етапів (рис. 1.1), причому етапи 2-5 складають цикл:

1. *Планування процесу людино-орієнтованого проектування*: вибір команди проекту та активностей, що відбуватимуться на кожному етапі.

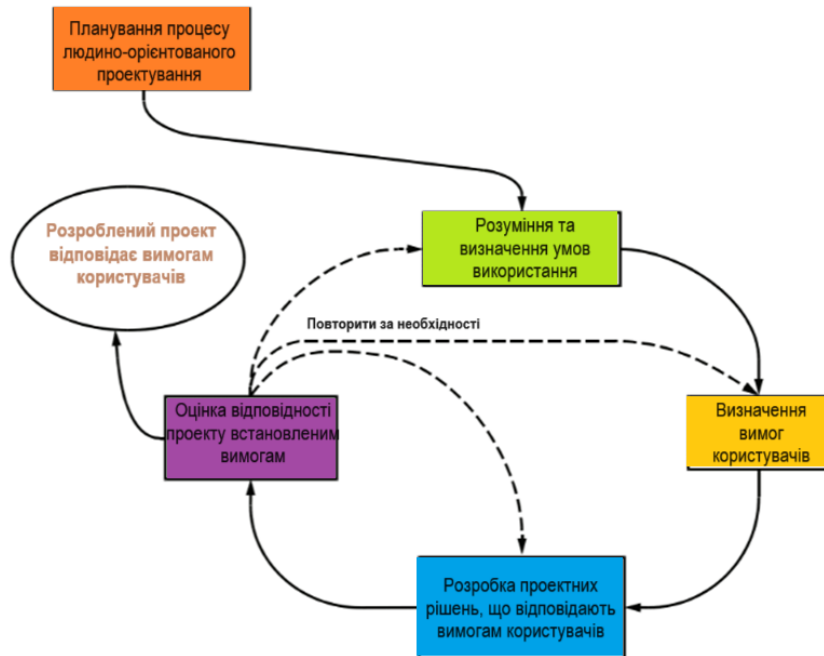

Рисунок 1.1. Процес HCD (Human Centered Design)

2. *Розуміння та визначення умов використання*: збір інформації про мету, завдання, обмеження та контекст роботи користувачів.

3. *Визначення вимог користувачів*: узагальнення отриманої на попередньому етапі інформації та формулювання вимог користувача до продукту.

4. *Розробка проектних рішень, що відповідають вимогам користувачів*: створення макетів, прототипів та різних версій дизайну відповідно до вимог користувача.

5. *Оцінка відповідності проекту встановленим вимогам*: оцінка створених рішень силами експертів або із залученням реальних користувачів. Якщо оцінка виявляє будь-які проблеми, то вимоги користувача необхідно уточнити і доопрацювати макет, прототип або дизайн продукту відповідно до нових даних.

Стандарт ISO 9241-210 ставить у центр процесу розробки користувальницькі вимоги. Однак концентрація виключно на користувачах може призвести до того, що продукт виявиться невигідним для бізнесу або важким з технічної точки зору. Тому для створення успішного продукту треба враховувати також бізнес-вимоги та технічну складову (рис. 1.2).

Рисунок 1.2. Структура людино-орієнтованого проектування

Як видно із рис. 1.2, користувачів, бізнес та технології поєднує UX-технології, а точніше UX-тестування.

UX-тестування – комплекс заходів, спрямованих на виявлення будь-яких проблемних місць на вашому ресурсі: чи достатньо зрозумілий, логічний, зручний, чи правильно працюють всі його технічні елементи.

Результатом грамотного UX-тестування є перелік рекомендацій, що і яким чином потрібно змінити, щоб підвищити кількість конверсій та перетворити відвідувачів сайту на його постійних та відданих користувачів.

Тестування виявляє великі та дрібні проблеми інтерфейсу, кожна з яких відсіває ваших потенційних покупців.

Також UX-тестування показує, наскільки зрозумілий покупцям ваш інтерфейс, чи використовують вони його так, як ви задумали, чи зовсім іншим чином. Отже, показують, яким чином потрібно змінити *user flow* на сайті, щоб користувачам було зручно.

UX-тестування потрібно, якщо ви хочете перевірити існуючий інтерфейс на зручність користувальницьких сценаріїв, відзначити всі "проблемні" місця та покращити їх. Дуже важливо проводити його наступним ресурсам:

- інтернет-магазинам, які хочуть збільшити прибуток, забезпечити зростання продажу (на будь-якому етапі їхньої роботи);
- проектам, які перебувають у розробці, щоб на початковому етапі зробити ресурс ефективним;
- сайтам, які мають великий трафік, але низькі показники конверсій;
- для мобільних додатків, у яких відсоток завантажень значно перевищує реальне використання;
- порталам з великою кількістю корисних функцій, які користувачі не використовують з невідомих причин і т.д.

Якщо ваш сайт не виконує покладену на нього функцію, якщо ви розумієте, що велика кількість користувачів просто йде, так і не дійшовши до мети (такою метою може бути не тільки покупка, але й знаходження потрібного товару, відповіді на їх питання, бронювання житла і т.д.), UX-тестування дасть відповідь на питання, чому так відбувається.

1.2. Що таке тестування та звідки воно з'явилося

Насамперед дамо визначення тестування ПЗ, щоб чіткіше розуміти, про що піде мова.

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

Протягом десятиліть розвитку розробки програмного забезпечення до питань тестування та забезпечення якості підходили дуже і дуже по-різному. Можна виділити кілька основних «епох тестування».

У 50–60-х роках минулого століття процес тестування був гранично формалізований, відокремлений від процесу безпосередньої розробки ПЗ та «математизований». Фактично тестування було швидше налагодженням програм. Існувала концепція так званого «вичерпного тестування» - перевірки всіх можливих шляхів виконання коду з усіма

можливими вхідними даними. Проте дуже швидко було з'ясовано, що вичерпне тестування неможливе, оскільки кількість можливих шляхів та вхідних даних дуже велика, а також за такого підходу складно знайти проблеми в документації.

Уявіть, що ваша програма за трьома введеними цілими числами визначає, чи може існувати трикутник з такими довжинами сторін. Припустимо, що ваша програма виконується в якомусь ізольованому ідеальному середовищі, і вам залишилося перевірити коректність її роботи на трьох 8-байтових знакових цілих числах. Ви використовуєте автоматизацію і комп'ютер може провести 100 мільйонів перевірок в секунду. Скільки візьме часу перевірка всіх варіантів?

А чи замислилися ви, як підготувати для цього тесту перевірочні дані (на основі яких можна визначити, чи правильно спрацювала програма у кожному конкретному випадку)?

У 70-х роках фактично народилися дві фундаментальні ідеї тестування: тестування спочатку розглядалося як процес доказу працездатності програми в деяких заданих умовах, а потім суворо навпаки: як процес доказу непрацездатності програми в деяких заданих умовах. Це внутрішнє протиріччя яке зникло з часом, а й у наші дні багатьма авторами цілком справедливо відзначається як дві взаємодоповнюючі мети тестування.

Зазначимо, що «процес доказу непрацездатності програми» цінується трохи більше, оскільки не дозволяє заплющувати очі на виявлені проблеми.

Швидше за все, саме з цих міркувань відбувається неправильне розуміння того, що негативні тест-кейси мають закінчуватися виникненням збоїв та відмов у додатку. Ні це не так. Негативні тест-кейси намагаються викликати збої та відмови, але застосунок, що коректна працює, витримує це випробування і продовжує працювати правильно. Також зазначимо, що очікуваним результатом негативних тест-кейсів є саме коректна поведінка програми, а самі негативні тест-кейс вважаються пройденими успішно, якщо їм не вдалося «поламати» застосунок.

У 80-х роках відбулася ключова зміна місця тестування в розробці ПЗ: замість однієї з фінальних стадій створення проекту, тестування стало застосовуватися протягом усього циклу розробки, що дозволило

в багатьох випадках не тільки швидко виявляти і усувати проблеми, але навіть передбачати і запобігати їх поява.

В цей же період відзначено бурхливий розвиток та формалізація методології тестування та поява перших елементарних спроб автоматизувати тестування.

У 90-х роках відбувся перехід від тестування як такого до все осяжнішого процесу, який називається «забезпечення якості». Цей етап охоплює весь цикл розробки ПЗ і зачіпає процеси планування, проектування, створення, виконання та підтримку наявних тест-кейсів та тестових оточень. Тестування вийшло на якісно новий рівень, який природно призвів до подальшого розвитку методології, появи досить потужних інструментів управління процесом тестування та інструментальних засобів автоматизації тестування, вже цілком схожих на своїх нинішніх нащадків.

У нульові роки нинішнього століття розвиток тестування продовжувався в контексті пошуку нових шляхів, методології, технік і підходів до забезпечення якості. Серйозний вплив на розуміння тестування справила поява гнучких методологій розробки і таких підходів, як «розробка під управлінням тестуванням». Автоматизація тестування вже сприймалася як звичайна невід'ємна частина більшості проектів, а також стали популярні ідеї про те, що на чільне місце тестування слід ставити не відповідність програми вимогам, а її здатність надати кінцевому користувачеві можливість ефективно вирішувати свої завдання.

1. 3. Компетентності необхідні проведення тестування ПЗ

Якщо пошукати інформацію за ключовими фразами з назви цього розділу, можна знайти безліч абсолютно суперечливих відповідей. І тут у першу чергу річь у тому, що автори більшості «посадових обов'язків» приписують всій професії якийсь перебільшений набір характеристик окремих її представників.

Якщо виразити образно головну мету тестувальника, то вона звучатиме так: «розуміти, що зараз необхідно проекту, чи отримує проект це необхідне належним чином, і якщо ні, як змінити ситуацію на краще».

Тож які ж технічні навички потрібні, щоб успішно почати тестувати програмні продукти?

1) Знання іноземних мов. Можете вважати це аксіомою: "немає знання англійської - немає кар'єри в IT". Інші іноземні мови теж вітаються, але англійська первинна.

2) Впевнене володіння комп'ютером на рівні по-справжньому просунутого користувача та бажання постійно розвиватися у цій галузі. Чи можете уявити собі професійного кухаря, який не може посмажити картоплю (не «не зобов'язаний», а «не вміє в принципі»)? Виглядає дивно? Не менш дивно виглядає «IT'шник» нездатний набрати осудно відформатований текст, скопіювати файл по мережі, розгорнути віртуальну машину або виконати будь-яку іншу повсякденну рутинну дію.

3) Програмування. Воно на порядок спрощує життя будь-якому IT'шнику (і тестувальнику в першу чергу). Чи можна тестувати без знання програмування? Так можна. Чи можна це робити по-справжньому добре? Ні. І зараз найголовніше (майже релігійно-філософське) питання: яку мову програмування вивчати? C/C++/C#, Java, PHP, JavaScript, Python, Ruby і т.д. Починати треба з того, на чому написаний ваш проект. Якщо проекту поки що немає, починайте з JavaScript (на даний момент найуніверсальніше рішення).

4) Бази даних та мова SQL. Тут від тестувальника теж не потрібна кваліфікація на рівні вузьких фахівців, але мінімальні навички роботи з найбільш поширеними СУБД та вміння писати прості запити можна вважати обов'язковими.

4) Розуміння принципів роботи мереж та операційних систем. Хоча б на мінімальному рівні, що дозволяє провести діагностику проблеми та вирішити її самотужки, якщо це можливо.

5) Розуміння принципів роботи веб-додатків та мобільних додатків. У наші дні майже все пишеться саме у вигляді таких програм, і розуміння відповідних технологій стає обов'язковим для ефективного тестування.

1.4. Процеси тестування та розробки ПЗ

1.4.1. Моделі розробки ПЗ. Щоб краще розібратися в тому, як тестування співвідноситься з програмуванням та іншими видами проектної діяльності, спочатку розглянемо самі основи моделі розробки ПЗ (як частина життєвого циклу). При цьому відразу підкреслимо, що розробка програмного забезпечення є лише частиною життєвого циклу програмного забезпечення, і тут ми говоримо саме про розробку.

Модель розробки ПЗ (Software Development Model, SDM) — структура, що систематизує різні види проектної діяльності, їхню взаємодію та послідовність у процесі розробки ПЗ.

Вибір тієї чи іншої моделі залежить від масштабу та складності проекту, предметної галузі, доступних ресурсів та безлічі інших факторів.

Вибір моделі розробки програмного забезпечення серйозно впливає на процес тестування, визначаючи вибір стратегії, розклад, необхідні ресурси і т. п.

Моделей розробки програмного забезпечення багато, але в загальному випадку класичними можна вважати водопадну, V-подібну, ітераційну інкрементальну, спіральну та гнучку.

Знати і розуміти моделі розробки ПЗ потрібно для того, щоб вже з перших днів роботи усвідомлювати, що відбувається навколо, що, навіщо і чому ви робите. Багато тестувальників-початківців відзначають, що відчуття безглуздості того, що відбувається, відвідує їх, навіть якщо поточні завдання цікаві. Чим повніше ви представлятимете картину того, що відбувається на проекті, тим ясніше вам буде видно ваш власний внесок у загальну справу і сенс того, чим ви займаєтеся.

Ще одна важлива річ, яку слід розуміти, полягає в тому, що жодна модель не є догмою чи універсальним рішенням. Нема ідеальної моделі. Є та, яка гірша чи найкраще підходить для конкретного проекту, конкретної команди, конкретних умов.

Водоспадна модель сьогодні представляє швидше історичний інтерес, так як у сучасних проектах практично не застосовується. Вона передбачає одноразове виконання кожної з фаз проекту, які, своєю чергою, суворо слідують друг за одним (рис. 1.3). Дуже спрощено можна

сказати, що в рамках цієї моделі будь-якої миті часу команді «видна» лише попередня і наступна фаза. У реальній розробці ПЗ доводиться «бачити весь проект цілком» і повертатися до попередніх фаз, щоб виправити недоробки або щось уточнити.

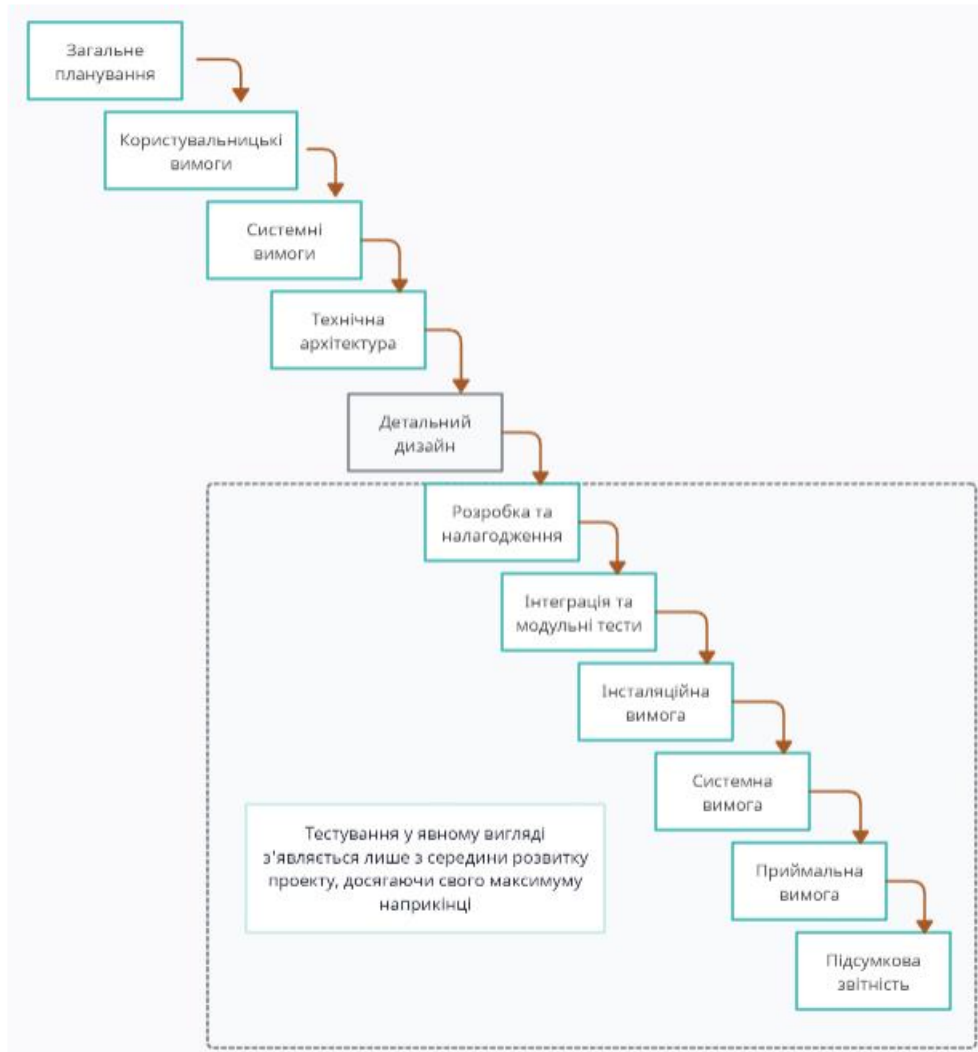

Рисунок 1.3. - Водоспадна модель розробки ПЗ

До недоліків водоспадної моделі прийнято відносити той факт, що участь користувачів ПЗ у ній або не передбачено взагалі, або передбачено лише побічно на стадії одноразового збору вимог. З точки зору тестування ця модель погана тим, що тестування в явному вигляді з'являється тут лише з середини розвитку проекту, досягаючи свого максимуму в самому кінці.

Проте водоспадна модель часто інтуїтивно застосовується під час виконання простих завдань, та її недоліки послужили чудовим

відправним пунктом до створення нових моделей. Також ця модель у дещо удосконаленому вигляді використовується на великих проектах, в яких вимоги дуже стабільні і можуть бути добре сформульовані на початку проекту (аерокосмічна область, медичне ПЗ тощо).

V-подібна модель є логічним розвитком водоспаду. Можна помітити (рис. 1.4), що у випадку як водоспадна, і V-подібна моделі життєвого циклу ПЗ можуть містити той самий набір стадій, але важлива відмінність у тому, як ця інформація використовується у процесі реалізації проекту.

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

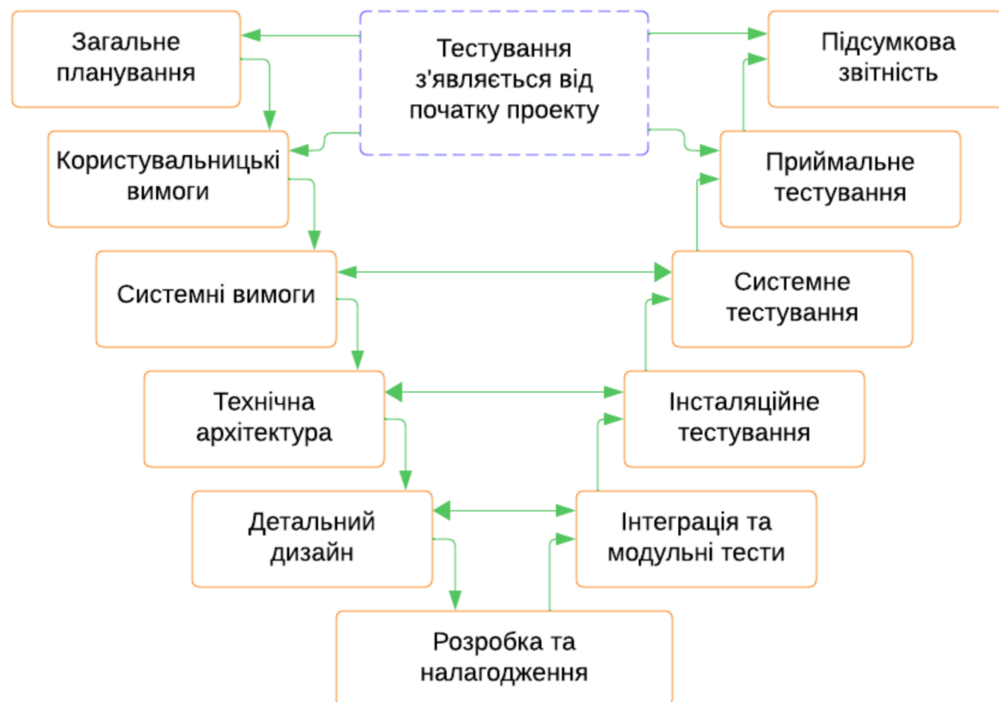

Рисунок 1.4. - V-подібна модель розробки ПЗ

Ітераційна інкрементальна модель є фундаментальною основою сучасного підходу розробки ПЗ. Як випливає з назви моделі, їй властива певна двоїстість:

по-перше, з погляду життєвого циклу модель є ітераційною, так як передбачає багаторазове повторення тих самих стадій;

по-друге,·з погляду розвитку продукту (прирощення його корисних функцій) модель є інкрементальною.

Ключовою особливістю даної моделі є розбиття проекту на відносно невеликі проміжки (ітерації), кожен з яких у загальному випадку може включати всі класичні стадії, властиві водоспадній і V-подібної моделям (рис. 1.5). Підсумком ітерації є збільшення (інкремент) функціональності продукту, виражене у проміжному білді .

Рисунок 1.5. - Ітераційна інкрементальна модель розробки ПЗ

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

У багатьох випадках допускається розпаралелювання окремих стадій усередині ітерації та активне доопрацювання з метою усунення недоліків, виявлених на будь-якій з попередніх стадій.

Ітераційна інкрементальна модель дуже добре зарекомендувала себе на об'ємних та складних проектах, які виконують великі команди

протягом тривалих термінів. Однак до основних недоліків цієї моделі часто відносять високі накладні витрати, викликані високою «бюрократизованістю» та загальною громіздкістю моделі.

Спіральна модель є окремим випадком ітераційної інкрементальної моделі, в якій особлива увага приділяється управлінню ризиками, що особливо впливають на організацію процесу розробки проекту та контрольні точки.

Схематично сутність спіральної моделі представлена на рис. 1.6.

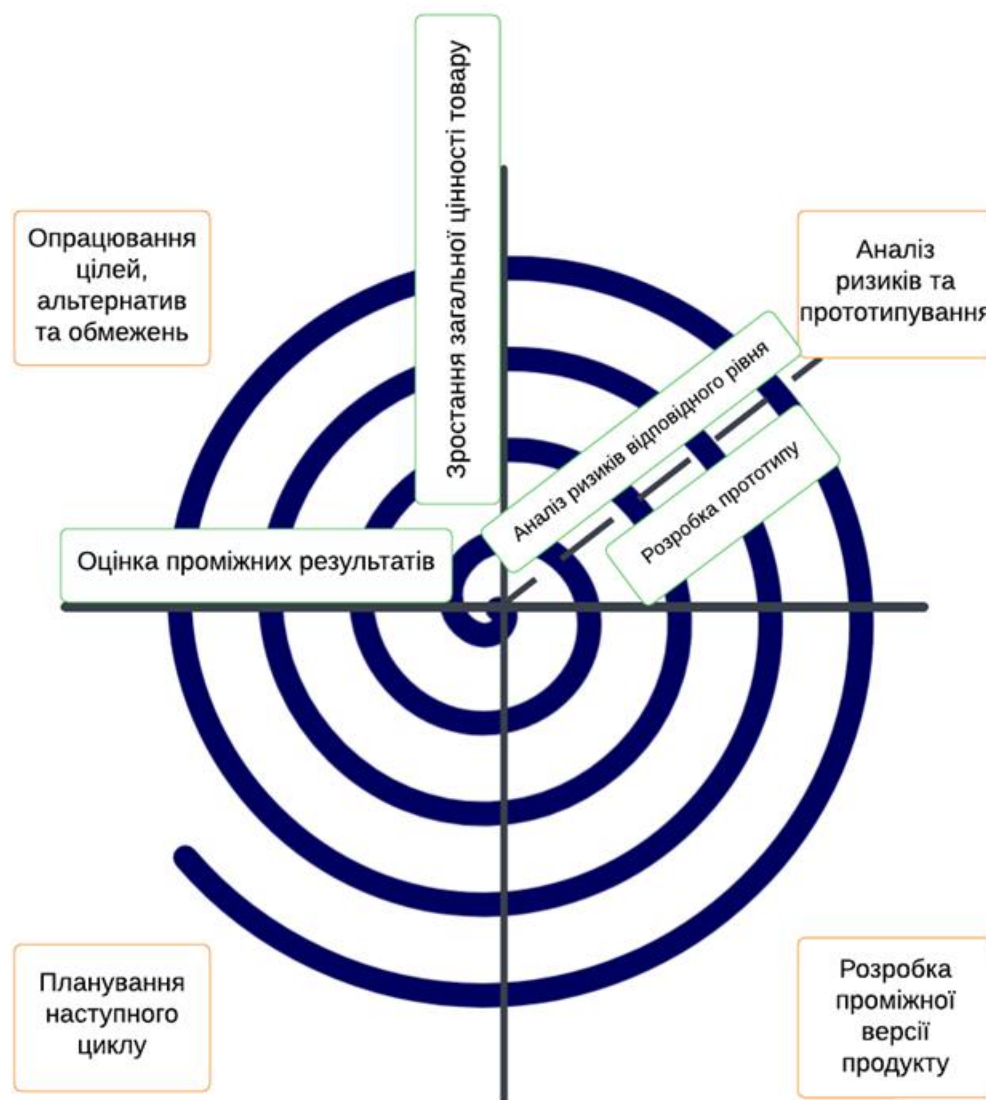

Рисунок 1.6. - Спіральна модель розробки ПЗ

Тут явно виділено чотири ключові фази:

1. опрацювання цілей, альтернатив та обмежень;
2. аналіз ризиків та прототипування;
3. розробка (проміжної версії) продукту;
4. планування наступного циклу.

З точки зору тестування та управління якістю підвищена увага ризикам є відчутною перевагою при використанні спіральної моделі для розробки концептуальних проектів, в яких вимоги є природно складними та нестабільними (можуть багаторазово змінюватися в процесі виконання проекту).

Гнучка модель (рис. 1.7) є сукупність різних підходів до розробки ПЗ і базується на так званому «agile -маніфесте»:

- 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 проходження початкового плану.

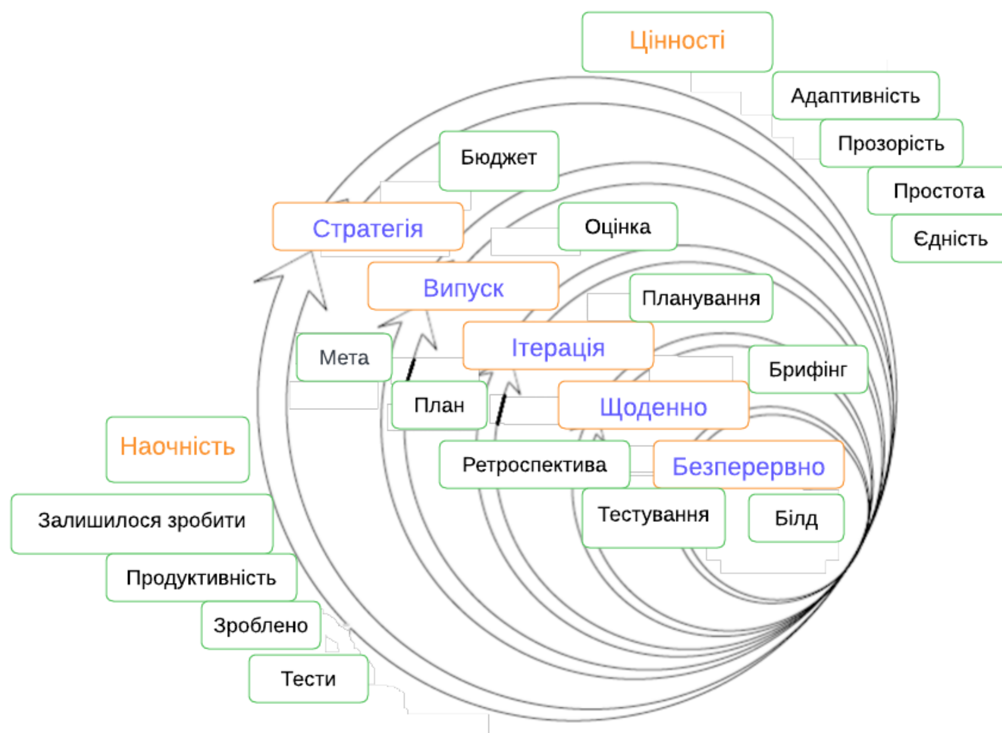

Рисунок 1.7 - Суть гнучкої моделі розробки ПЗ

Як неважко здогадатися, покладені в основу гнучкої моделі підходи є логічним розвитком і продовженням всього того, що було за десятиліття створено та випробувано у водоспадній, V-подібній, ітераційній інкрементальній, спіральній та інших моделях. Причому тут вперше було досягнуто відчутного результату у зниженні бюрократичної складової та максимальної адаптації процесу розробки ПЗ до миттєвих змін ринку та вимог замовника.

Головним недоліком гнучкої моделі вважається складність її застосування до великих проектів, а також часто помилкове впровадження її підходів, викликане не порозумінням фундаментальних принципів моделі. Проте можна стверджувати, що дедалі більше проектів починають використовувати гнучку модель розробки.

Коротко можна виразити суть моделей розробки програмного забезпечення таблицею 1.1.

Таблиця 1.1

Порівняння моделей розробки ПЗ

Модель	Переваги	Недоліки	Тестування
1	2	3	4
Водоспадна	У кожній стадії є точний результат, що перевіряється. Кожен момент часу команда виконує один вид роботи. Добре працює для невеликих завдань.	Повна нездатність адаптувати проект до змін у вимогах. Вкрай пізніше виробництво працюючого товару.	Із середини проекту.
V-подібна	У кожній стадії є точний результат, що перевіряється. Увага тестування приділяється з першої ж стадії. Добре працює для проектів із стабільними вимогами.	Недостатня гнучкість та адаптованість. Відсутнє раннє прототипування. Складність усунення проблем, пропущених ранніх стадіях розвитку проекту.	На переходах між стадіями.

Продовження табл. 1.1

Ітераційна інкрементальна	Досить раннє прототипування. Простота керування ітераціями. Декомпозиція проекту на керовані ітерації.	Недостатня гнучкість усередині ітерацій. Складність усунення проблем, пропущених ранніх стадіях розвитку проекту.	У певні моменти ітерації. Повторне тестування (після доопрацювання) вже перевіреного раніше.
Спіральна	Глибокий аналіз ризиків. Підходить для великих проектів. Досить раннє прототипування.	Високі накладні витрати. Складність застосування для невеликих проектів. Висока залежність успіху від якості аналізу ризиків.	У певні моменти ітерації. Повторне тестування (після доопрацювання) вже перевіреного раніше.
Гнучка	Максимальне залучення замовника. Багато роботи із вимогами. Тісна інтеграція тестування та розробки. Мінімізація документації.	Складність реалізації великих проектів. Складність побудови стабільних процесів.	У певні моменти ітерацій та у будь-який необхідний момент.

1.4.2. Життєвий цикл тестування. Наслідуючи загальну логіку ітеративності, що переважає у всіх сучасних моделях розробки ПЗ, життєвий цикл тестування також виражається замкнутою послідовністю дій (рис. 1.8).

Важливо розуміти, що довжина такої ітерації (і, відповідно, ступінь подробиці кожної стадії) може змінюватись у найширшому діапазоні – від одиниць годин до десятків місяців. Як правило, якщо йдеться про тривалий проміжок часу, він розбивається на безліч відносно коротких ітерацій, але сам при цьому «тяжіє» до тієї чи іншої стадії в кожний

момент часу (наприклад, на початку проекту більше планування, наприкінці — більше звітності).

Рисунок 1.8 - Життєвий цикл тестування

Також ще раз підкреслимо, що наведена схема не догма, і ви легко можете знайти альтернативи, але загальна суть і ключові принципи залишаються незмінними. Їх і розглянемо.

Стадія 1 (загальне планування та аналіз вимог) об'єктивно необхідна як мінімум для того, щоб мати відповідь на такі питання:

- що нам належить тестувати;
- як багато буде роботи;
- які є складнощі;
- чи все необхідне у нас є і т.п.

Зазвичай, отримати відповіді ці питання неможливо без аналізу вимог, так як саме вимоги є первинним джерелом відповідей.

Стадія 2 (уточнення критеріїв приймання) дозволяє сформулювати або уточнити метрики та ознаки можливості або

необхідності початку тестування, зупинення та відновлення тестування, завершення або припинення тестування.

Стадія 3 (уточнення стратегії тестування) це ще одне звернення до планування, але вже на локальному рівні: розглядають та уточнюють ті частини стратегії тестування, які актуальні для поточної ітерації.

Стадія 4 (розробка тест-кейсів) присвячена розробці, перегляду, уточненню, доопрацюванню, переробці та іншим діям із тест-кейсами, тестовими сценаріями та іншими артефактами, які будуть використовуватись при безпосередньому виконанні тестування.

Стадія 5 (виконання тест-кейсів) та *стадія* 6 (фіксація знайдених дефектів) тісно пов'язані між собою та фактично виконуються паралельно: дефекти фіксуються відразу за фактом їх виявлення у процесі виконання тест-кейсів. Однак найчастіше після виконання всіх тест-кейсів та написання всіх звітів про знайдені дефекти проводиться явно виділена стадія уточнення, на якій всі звіти про дефекти розглядаються повторно з метою формування єдиного розуміння проблеми та уточнення таких характеристик дефекту, як важливість та терміновість.

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

У життєвому циклі тестування п'ять із восьми стадій так чи інакше пов'язані з управлінням проектами, розгляд якого не входить до програми даної дисципліни. А зараз ми переходимо до ключових навичок та основних видів діяльності тестувальників та почнемо з роботи з документацією.

1.4.3. Основні принципи тестування. Тестування показує наявність дефектів, а не їх відсутність. Дуже складно виявити щось, до чого ми не знаємо — ні «де воно», ні «як воно виглядає», ні навіть «чи є воно взагалі». Це частково нагадує спроби «згадати, чи я не забув щось».

У силу того факту, що не існує фізичної можливості перевірити поведінку складного програмного продукту у всіх можливих ситуаціях та умовах, тестування не може гарантувати, що в тій чи іншій ситуації, при збігу тих чи інших обставин, дефект не виникне.

Що тестування може — це використовувати колосальний набір технік, підходів, інструментів і рішень для того, щоб перевірити найбільш ймовірні, затребувані ситуації та виявити дефекти при їх виникненні.

Такі дефекти будуть усунуті, що відчутно підвищить якість продукту, але, як і раніше, не гарантує виникнення проблем у решті, не перевірених ситуаціях та умовах. Отже, основні принципи тестування таки:

Вичерпне тестування неможливе.

Вичерпне тестування теорії покликане перевірити застосунок з усіма можливими комбінаціями всіх можливих вхідних даних у всіх можливих умовах виконання. Але щойно було підкреслено у попередньому принципі — це неможливо фізично.

Навіть для одного простого поля для введення імені користувача може існувати близько 2 432 позитивних перевірок і нескінченна кількість негативних перевірок.

Тому немає жодного шансу протестувати програмний продукт повністю, «вичерпно».

Однак, з цього не випливає, що тестування не є ефективним. Вдумливий аналіз вимог, облік ризиків, розстановка пріоритетів, аналіз предметної галузі, моделювання, робота з кінцевими користувачами, застосування спеціальних технік тестування – ці та багато інших підходів дозволяють виявити ті галузі або умови експлуатації продукту, які потребують особливо ретельної перевірки.

І оскільки тут обсяг роботи незрівнянно менший – таке тестування вже не просто можливе, а й виконується на щоденній основі.

Тестування тим ефективніше, що раніше воно виконується.

Цей принцип закликає не відкладати тестування на потім і на останній момент. Звичайно, надмірно раннє тестування може виявитися неефективним і навіть призвести до необхідності повторно виконувати великий обсяг роботи, але розпочате вчасно (без зволікання) тестування дає найбільший ефект.

Раннє тестування допомагає усунути чи скоротити дорогі зміни. Цей принцип має прекрасну аналогію зі звичайного повсякденного життя. Уявіть, що ви збираєтесь у поїздку та продумуєте список речей, які необхідно взяти із собою.

На стадії обдумування додати, змінити, видалити будь-який пункт у цьому списку нічого не варто. На стадії поїздки магазинами для закупівлі необхідного, недоробки у списку вже можуть призвести до необхідності повторної поїздки до магазину. На стадії відправлення на місце призначення недоробки у списку речей явно призведуть до відчутної втрати нервів, часу та грошей. А якщо фатальний недолік списку речей з'ясується тільки після прибуття, може виявитися, що вся поїздка втратила сенс.

Кластеризація дефектів

Дефекти не виникають "просто так". І вже тим більше «просто так» не з'являється багато дефектів у якійсь «проблемній» області програми (не дарма вона і називається «проблемною»).

Можливо, тут використовується якась нова чи складна розробка. Можливо, тут застосунку доводиться працювати у несприятливих умовах чи взаємодіяти із зовнішніми ненадійними компонентами. Або так вийшло, що відповідну частину вимог не було опрацьовано належним чином. Або зовсім (на жаль, буває і таке) за реалізацію цієї частини програми відповідали недостатньо відповідальні або недостатньо компетентні люди.

У будь-якому випадку «групування» дефектів за якоюсь явною ознакою є гарним приводом для продовження дослідження даної галузі програмного продукту: швидше за все, саме тут буде виявлено ще більше дефектів.

Так, виявлення подібних тенденцій до кластеризації (і особливо пошук глобальної першопричини) часто вимагає від тестувальників певних знань та досвіду, але якщо такий «кластер» виявлено — це дозволяє відчутно мінімізувати зусилля і при цьому істотно підвищити якість програми.

Парадокс пестициду

Назва цього принципу походить від загальновідомого явища в сільському господарстві: якщо довго розпорошувати той самий пестицид на посіви, у комах незабаром виробляється імунітет, що робить пестицид неефективним.

Те саме вірно і для тестування програмного забезпечення, де парадокс пестициду проявляється у повторенні одних і тих самих (або просто однотипних) перевірок знову і знову: згодом ці перевірки перестануть виявляти нові дефекти.

Щоб подолати парадокс пестициду, необхідно регулярно переглядати та оновлювати тест-кейси, урізноманітнити підходи до тестування, застосовувати різні техніки тестування, дивитися на ситуацію «свіжим поглядом» (можливо із залученням тих учасників команди, які раніше не працювали з даною областю програмного продукту).

Тестування залежить від контексту

Погодьтеся, ви по-різному підходитимете до приготування «чого-небудь перекусити для себе» і до організації сімейної вечері з якогось дуже урочистого приводу.

У тестуванні логіка та ж: програмні продукти можуть відноситися до різних предметних областей, бути побудовані з використанням різних технологій, використовуватися для вирішення більш менш «відповідальних» завдань і т. п. — все це та багато іншого впливає на те, як має бути організовано процес тестування.

Набір характеристик програмного продукту впливає на глибину тестування, використовуваний набір технік та інструментів, принципи організації роботи тестувальників тощо.

Основна ідея цього принципу полягає в тому, що неможливо виробити якийсь «універсальний підхід до тестування» на всі випадки

життя, і навіть бездумне копіювання підходів до тестування з одних проектів на інші часто не закінчується нічим добрим.

Якщо ж брати до уваги як загальні, так і унікальні властивості поточного проекту та вибудовувати тестування відповідним чином, воно виявляється найбільш ефективним та результативним.

Відсутність дефектів – не самоціль.

Уявіть собі, що ви купили комусь апельсин. Ідеальний. Найкращий в світі. Гідний стати зразком апельсинів на всі часи. Але той, кому ви купували цей апельсин, розчарований — адже він просив грейпфрут.

Так і програмний продукт повинен не тільки бути позбавлений дефектів настільки, наскільки це можливо, але й задовольняти вимоги замовника і кінцевих користувачів - інакше він стане непридатним для використання.

Варто зазначити, що нерідко порушення цього принципу полягає в недостатньому опрацюванні та реалізації функціональних вимог до продукту, що тягне за собою справедливу критику з боку кінцевих користувачів та загальне падіння популярності продукту.

Якщо об'єднати цей принцип із попереднім, виходить: саме розуміння контексту продукту та потреб користувачів дозволяє тестувальникам вибрати найкращу стратегію та досягти найкращого результату.

Незважаючи на те, що дані принципи тестування власними силами не є магічною гарантією успіху, їх розуміння має дозволити краще сприйняти та засвоїти поданий далі матеріал.

1.5. Тестування документації та вимог

1.5.1. Що таке «вимоги». Як ми тільки що розглянули в матеріалі, присвяченій життєвому циклу тестування, все так чи інакше починається з документації та вимог.

Вимога — опис того, які функції та з дотриманням яких умов має виконувати застосунок у процесі вирішення корисного для користувача завдання.

Невеликий історичний відступ: якщо пошукати визначення вимог у літературі 10-20-30-річної давності, то можна помітити, що спочатку про користувачів, їх завдання та корисні для них властивості застосунку у визначенні вимоги не було сказано. Користувач виступав якоюсь абстрактною фігурою, яка не має відношення до застосунку. В даний час такий підхід неприпустимий, тому що він не тільки призводить до комерційного провалу продукту на ринку, а й багаторазово підвищує витрати на розробку та тестування.

1.5.2. Важливість вимог. Вимоги є відправною точкою визначення того, що проектна команда буде проектувати, реалізовувати і тестувати. Елементарна логіка говорить нам, якщо у вимогах щось «не те», то й реалізовано буде «не те», тобто колосальна робота безлічі людей буде виконана марно.

Брайан Хенкс, описуючи важливість вимог, підкреслює, що вони:

- дозволяють зрозуміти, що з дотриманням яких умов система повинна робити;
- надають можливість оцінити масштаб змін та керувати змінами;
- є основою формування плану проекту (зокрема плану тестування);
- допомагають запобігати чи вирішувати конфліктні ситуації;
- спрощують розміщення пріоритетів у наборі завдань.
- дозволяють об'єктивно оцінити ступінь прогресу у розробці проекту.

Незалежно від того, яка модель розробки ПЗ використовується на проекті, чим пізніше буде виявлено проблему, тим складніше і дорожче буде її вирішення. А на самому початку («водоспаду», «спуску по літері v», «ітерації», «витку спіралі») йде планування та робота з вимогами.

Якщо проблема в вимогах буде з'ясована на цій стадії, її вирішення може звестися до виправлення кількох слів у тексті, тоді як недоробка, викликана пропущеною проблемою в вимогах і виявлена на стадії експлуатації, може навіть повністю знищити проект.

Спробуємо проілюструвати цю саму думку на простому прикладі. Допустимо, ви з друзями складаєте список покупок перед поїздкою до гіпермаркету. Ви поїдете купувати, а друзі чекають на вас вдома. Скільки «коштує» дописати, викреслити або змінити пару пунктів, поки

ви тільки складаєте список? Анітрохи. Якщо думка про недосконалість списку наздогнала вас на шляху до гіпермаркету, вже доведеться дзвонити (дешево, але не безкоштовно). Якщо ви зрозуміли, що у списку «щось не те» в черзі на касу, доведеться повертатися до торгового залу та витрачати час. Якщо проблема з'ясувалась по дорозі додому або навіть вдома, доведеться повернутися до гіпермаркету. І, нарешті, клінічний випадок: у списку спочатку було щось зовсім неправильне (наприклад, «100 кг цукерок — і все»), поїздка здійснена, всі гроші витрачені, цукерки привезені і тільки тут з'ясовується, що «ну ми ж пожартували».

Ще одним аргументом на користь тестування вимог є те, що, за різними оцінками, вони зароджуються від ½ до ¾ всіх проблем з програмним забезпеченням. У результаті є ризик, що вийде так як показано на рис. 1.9.

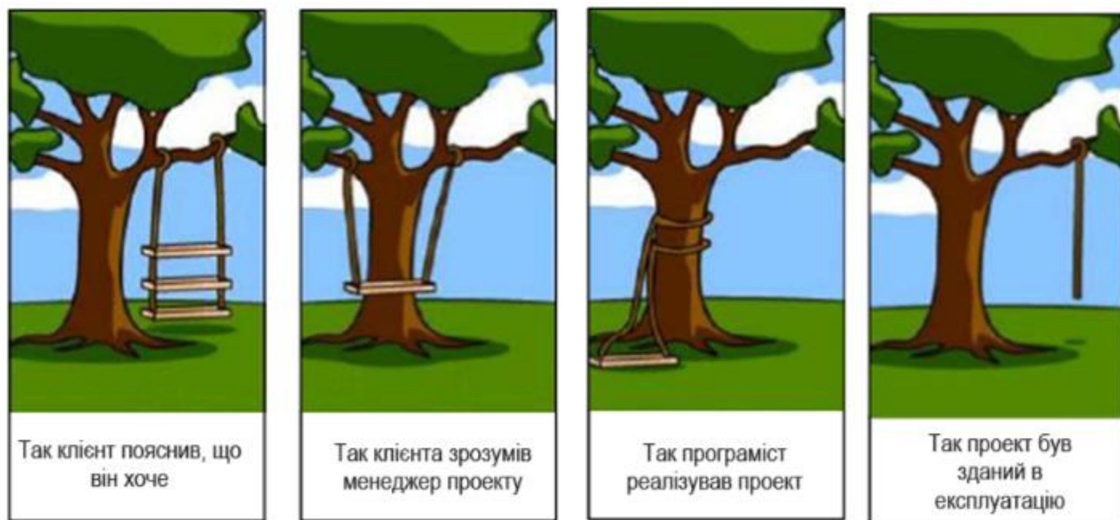

Рисунок 1.9 — Типовий проект із поганими вимогами

Оскільки ми постійно говоримо «документація та вимоги», а не просто «вимоги», то варто розглянути перелік документації, яка повинна піддаватися тестуванню у процесі розробки ПЗ (хоча далі ми концентруватимемося саме на вимогах).

Документацію можна розділити на два великих виду залежно від часу та місця її використання.

Продуктна документація використовується проектною командою під час розробки та підтримки продукту. Вона включає:

- план проекту і навіть тестовий план;
- вимоги до програмного продукту та функціональні специфікації;
- архітектуру та дизайн;
- тест-кейси та набори тест-кейсів;
- технічні специфікації, такі як схеми баз даних, опис алгоритмів, інтерфейсів і т.д.

Проектна документація включає як продуктну, так і деякі додаткові види документації і використовується не тільки на стадії розробки, але і на більш ранніх і пізніх стадіях (наприклад, на стадії впровадження та експлуатації). Вона включає:

- користувальницьку та супровідну документацію, таку як вбудована допомога, посібник зі встановлення та використання, ліцензійні угоди тощо;
- маркетингову документацію, яку представники розробника чи замовника використовують як у початкових етапах (для уточнення суті та концепції проекту), і на фінальних етапах розвитку проекту (для просування продукту над ринком).

У деяких класифікаціях частину документів із продуктної документації можна перелічити у проектної документації — це цілком нормально, так як поняття проектної документації за визначенням є ширшим.

Ступінь важливості та глибина тестування того чи іншого виду документації і навіть окремого документа визначається великою кількістю факторів, але незмінним залишається загальний принцип: все, що ми створюємо в процесі розробки проекту (навіть малюнки маркером на дошці, аркуші, листування у скайпі), можна вважати документацію і так чи інакше піддавати тестуванню (наприклад, вичитування листа перед відправкою - це теж свого роду тестування документації).

1.5.3. Джерела та шляхи виявлення вимог. Вимоги розпочинають своє життя на стороні замовника. Їх збирання та виявлення здійснюються за допомогою наступних основних технік (рис. 1.10).

Рисунок 1.10 — Основні техніки збору та виявлення вимог

Інтерв'ю. Найуніверсальніший шлях виявлення вимог, що полягає у спілкуванні проектного спеціаліста (як правило, спеціаліста з бізнес-аналізу) та представника замовника (або експерта, користувача тощо). Інтерв'ю може відбуватися у класичному розумінні цього слова (розмова у вигляді «питання-відповідь»), у вигляді листування тощо. Головним тут є те, що ключовими фігурами виступають двоє — інтерв'юйований та інтерв'юер (хоча це й не виключає наявності «аудиторії слухачів», наприклад, у вигляді осіб, які поставлені в копію листування).

Робота з фокусними групами. Може виступати як варіант «розширеного інтерв'ю», де джерелом інформації є не одна особа, а група осіб (як правило, що представляють цільову аудиторію, та/або мають важливу для проекту інформацію, та/або уповноважених приймати важливі для проекту рішення).

Анкетування. Цей варіант виявлення вимог викликає багато суперечок, так як за неправильної реалізації може призвести до нульового результату при об'ємних витратах. У той же час при правильній організації анкетування дозволяє автоматично зібрати та обробити величезну кількість відповідей від величезної кількості респондентів. Ключовим фактором успіху є правильне складання анкети, правильний вибір аудиторії та правильне піднесення анкети.

Семінари та мозковий штурм. Семінари дозволяють групі людей дуже швидко обмінятися інформацією (і наочно продемонструвати ті чи

інші ідеї), а також добре поєднуються з інтерв'ю, анкетуванням, прототипуванням та моделюванням — у тому числі для обговорення результатів та формування висновків та рішень. Мозковий штурм може проводитись і як частина семінару, і як окремий вид діяльності. Він дозволяє за мінімальний час згенерувати велику кількість ідей, які надалі можна не поспішаючи розглянути з точки зору їх використання для розвитку проекту.

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

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

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

Моделювання процесів та взаємодій. Може застосовуватися як до «бізнес-процесів та взаємодій» (наприклад: «договір на закупівлю

формується відділом закупівель, візується бухгалтерією та юридичним відділом…»), так і до «технічних процесів та взаємодій» (наприклад: «платіжне доручення генерується модулем “Бухгалтерія ”, шифрується модулем “Безпека” та передається на збереження в модуль “Сховище”»). Ця техніка вимагає високої кваліфікації фахівця з бізнес-аналізу, так як пов'язана з обробкою великого обсягу складної (і часто погано структурованої) інформації.

Самостійний опис. Є не так технікою виявлення вимог, як технікою їх фіксації та формалізації. Дуже складно (і навіть не можна!) намагатися самому «вигадати вимоги за замовника», але у спокійній обстановці можна самостійно опрацювати зібрану інформацію та обережно оформити її для подальшого обговорення та уточнення.

1.5.4 . Рівні та типи вимог. Форма подання, ступінь деталізації та перелік корисних властивостей вимог залежать від рівнів та типів вимог, які схематично представлені на рис. 1.11.

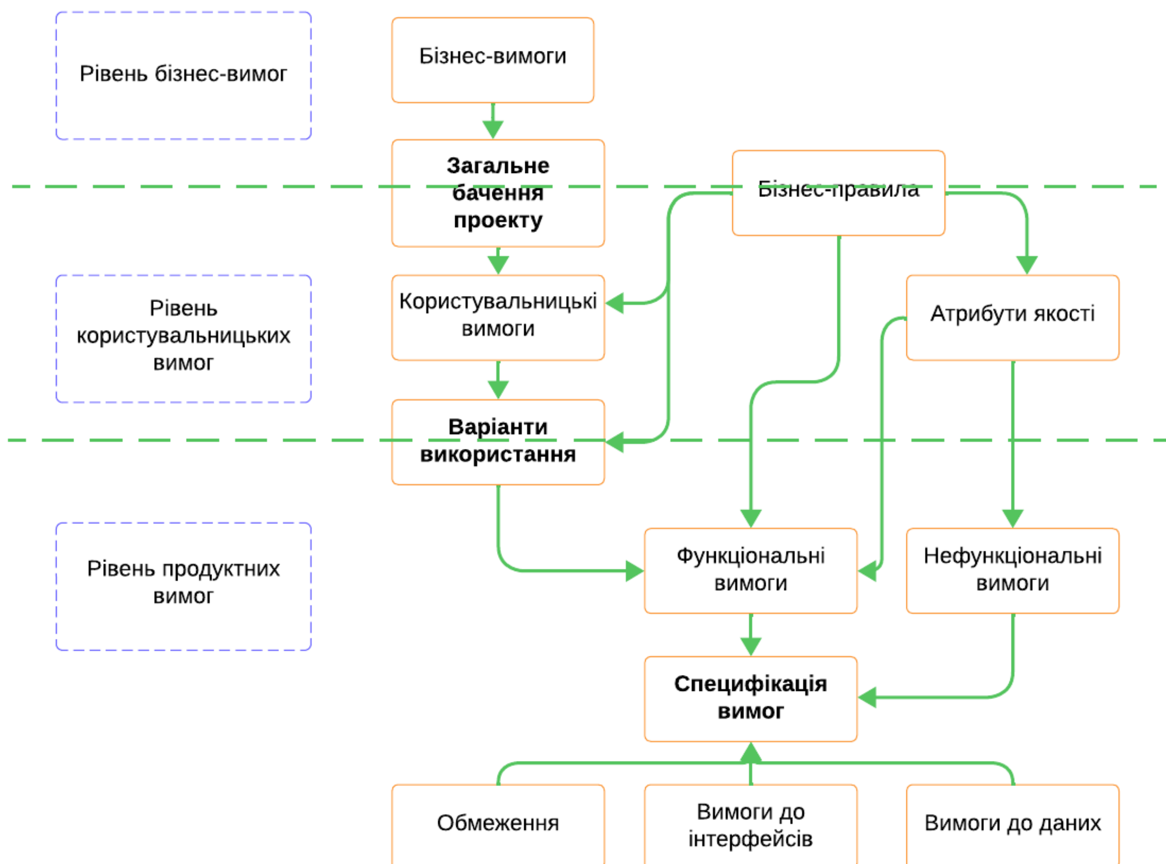

Рисунок 1.11 – Рівні та типи вимог

Бізнес-вимоги виражають мету, заради якої розробляється продукт (навіщо взагалі він потрібен, яка від нього очікується користь, як замовник за його допомогою отримуватиме прибуток). Результатом виявлення вимог цьому рівні є загальне бачення — документ, який, зазвичай, представлений простим текстом і таблицями. Тут немає деталізації поведінки системи та інших технічних характеристик, але цілком можуть бути визначені пріоритети бізнес-завдань, що вирішуються, ризики і т.п.

Декілька простих, ізольованих від контексту та один від одного прикладів бізнес-вимог:

Потрібен інструмент, що в реальному часі відображає найбільш вигідний курс купівлі та продажу валюти.

Необхідно вдвічі-втричі підвищити кількість заявок, що обробляються одним оператором за зміну.

Потрібно автоматизувати процес виписки товарно-транспортних накладних з урахуванням договорів.

Користувацькі вимоги описують завдання, які користувач може виконувати за допомогою системи, що розробляється (реакцію системи на дії користувача, сценарії роботи користувача). Оскільки тут з'являється опис поведінки системи, вимоги цього рівня можуть бути використані для оцінки обсягу робіт, вартості проекту, часу розробки і т.п. Користувацькі вимоги оформляються у вигляді варіантів використання, історій користувача, користувальницьких сценаріїв.

Декілька простих, ізольованих від контексту та один від одного прикладів користувальницьких вимог:

При першому вході користувача до системи має відображатися ліцензійна угода.

Адміністратор повинен мати можливість переглядати список усіх користувачів, які працюють на даний момент у системі.

При першому збереженні нової статті, система повинна видавати запит на збереження у вигляді чернетки або публікацію.

Бізнес-правила описують особливості прийнятих у предметній галузі (або безпосередньо у замовника) процесів, обмежень та інших

правил. Ці правила можуть належати до бізнес-процесів, правил роботи співробітників, нюансів роботи ПЗ тощо.

Декілька простих, ізольованих від контексту та один від одного прикладів бізнес-правил:

Жодний документ, переглянутий відвідувачами сайту хоча б один раз, не може бути відредагований або видалений.

Публікація статті можлива лише після затвердження головним редактором.

Підключення до системи ззовні офісу заборонено у неробочий час.

Атрибути якості розширюють собою нефункціональні вимоги і на рівні вимог користувача можуть бути представлені у вигляді опису ключових для проекту показників якості (властивостей продукту, не пов'язаних з функціональністю, але є важливими для досягнення цілей створення продукту - продуктивність, масштабованість, відновлюваність). Атрибутів якості дуже багато, але для будь-якого проекту реально важливими є лише деяке їхнє підмножина.

Декілька простих, ізольованих від контексту та один від одного прикладів атрибутів якості:

Максимальний час готовності системи до виконання нової команди після скасування попередньої не може перевищувати однієї секунди.

Внесені до тексту статті зміни не повинні бути втрачені при порушенні з'єднання між клієнтом та сервером.

Програма повинна підтримувати додавання довільної кількості неієрогліфічних мов інтерфейсу.

Функціональні вимоги описують поведінку системи, тобто її дії (обчислення, перетворення, перевірки, обробку тощо.). У контексті проектування функціональні вимоги переважно впливають на дизайн системи.

Варто пам'ятати, що до поведінки системи відноситься не тільки те, що система повинна робити, а й те, що вона не повинна робити (наприклад: «застосунок не повинен вивантажувати з оперативної

пам'яті фонові документи протягом 30 хвилин з моменту виконання з ними останньої операції»).

Декілька простих, ізольованих від контексту та один від одного прикладів функціональних вимог:

У процесі інсталяції програма повинна перевіряти залишок вільного місця на цільовому носії.

Система повинна автоматично виконувати резервне копіювання даних щодня у вказаний час.

Електронна адреса користувача, що вводиться під час реєстрації, повинна бути перевірена на відповідність вимогам RFC822.

Нефункціональні вимоги описують властивості системи (зручність використання, безпека, надійність, розширюваність і т. п.), якими вона повинна мати при реалізації своєї поведінки. Тут наводиться більш технічний та детальний опис атрибутів якості. У контексті проектування дисфункції переважно впливають на архітектуру системи.

Декілька простих, ізольованих від контексту та один від одного прикладів нефункціональних вимог:

При одночасній безперервній роботі з системою 1000 користувачів мінімальний час між виникненням збоїв повинен бути більшим або рівним 100 годин.

За жодних умов загальний обсяг пам'яті, що використовується додатком, не може перевищувати 2 ГБ.

Розмір шрифту для будь-якого напису на екрані має підтримувати налаштування в діапазоні від 5 до 15 пунктів.

Наступні вимоги в загальному випадку можуть бути віднесені до нефункціональних, проте їх часто виділяють в окремі підгрупи. Тут для простоти розглянуті лише три таких підгрупи, але їх може бути і набагато більше; як правило, вони походять з атрибутів якості, але високий ступінь деталізації дозволяє віднести їх до рівня вимог продукту.

Обмеження є чинники, що обмежують вибір способів і засобів (зокрема інструментів) реалізації продукту.

Декілька простих, ізольованих від контексту та один від одного прикладів обмежень:

Всі елементи інтерфейсу повинні відображатися без прокручування при роздільній здатності екрана від 800x600 до 1920x1080.

Не допускається використання *Flash* під час реалізації клієнтської частини програми.

Програма повинна зберігати здатність реалізовувати функції з рівнем важливості «критичний» за відсутності у клієнта підтримки *JavaScript*.

Вимоги до інтерфейсів описують особливості взаємодії системи, що розробляється, з іншими системами та операційним середовищем.

Декілька простих, ізольованих від контексту та один від одного прикладів вимог до інтерфейсів:

Обмін даними між клієнтською та серверною частинами програми під час здійснення фонових AJAX-запитів має бути реалізований у форматі JSON.

Протоколування подій має проводитись у журналі подій операційної системи.

З'єднання з поштовим сервером має виконуватися згідно з RFC3207.

Вимоги до даних описують структури даних (і самі дані), що є невід'ємною частиною системи, що розробляється. Часто сюди відносять опис бази даних та особливостей її використання.

Декілька простих, ізольованих від контексту та один від одного прикладів вимог до даних:

Всі дані системи, за винятком документів користувача, повинні зберігатися в БД під керуванням СУБД MySQL, користувальницькі документи повинні зберігатися в БД під керуванням СУБД MongoDB .

Інформація про касові транзакції за поточний місяць повинна зберігатися в операційній таблиці, а після закінчення місяця переноситись до архівної.

Для прискорення операцій пошуку за текстом статей та оглядів мають бути передбачені повнотекстові індекси на відповідних полях таблиць.

Специфікація вимог поєднує в собі опис усіх вимог рівня продукту і може бути вельми об'ємним документом (сотні і тисячі сторінок).

Оскільки вимог може бути дуже багато, а їх доводиться не лише один раз написати та узгодити між собою, а й постійно оновлювати, роботу проектної команди з управління вимогами значно полегшують відповідні інструментальні засоби.

1.1.5. Властивості якісних вимог. У процесі тестування вимог перевіряється їхня відповідність певному набору властивостей (рис. 1.12).

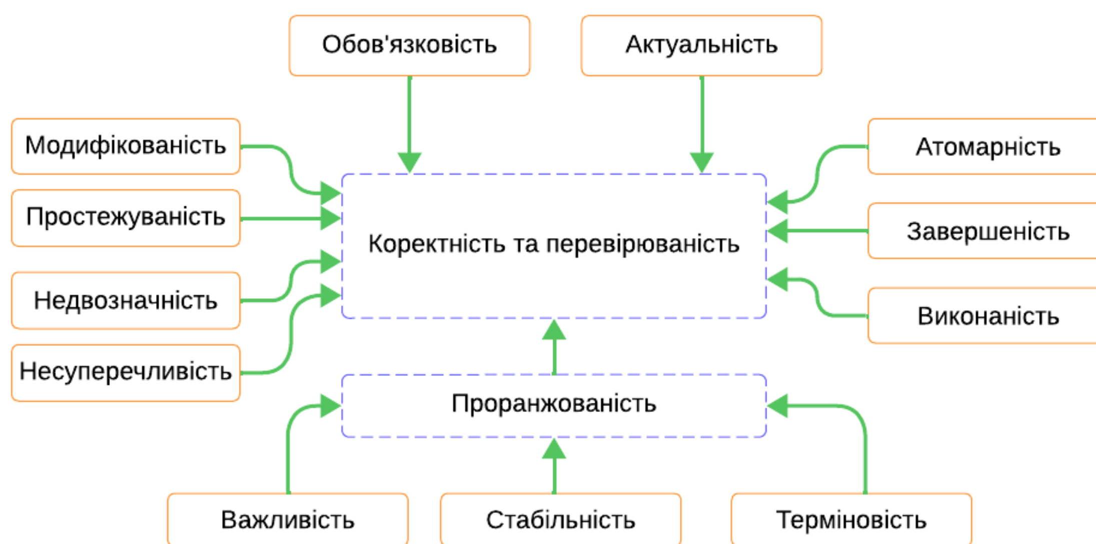

Рисунок 1.12 - Властивості якісної вимоги

Завершеність. Вимога є повної і закінченої з погляду подання у ній всієї необхідної інформації, ніщо не пропущена з міркувань «це і так всім зрозуміло». Типові проблеми із завершеністю:

Відсутні нефункціональні складові вимоги або посилання на відповідні нефункціональні вимоги (наприклад: "паролі повинні зберігатися в зашифрованому вигляді" – який алгоритм шифрування?).

Вказано лише частину деякого перерахування (наприклад: «експорт здійснюється у формати PDF, PNG тощо» – що ми повинні розуміти під «тощо»?).

Наведені посилання є неоднозначними (наприклад: «див. вище» замість «див. розділ 123.45.b»).

Атомарність, одиничність. Вимога є атомарною, якщо її не можна розбити на окремі вимоги без втрати завершеності і вона описує одну і туж ситуацію. Типові проблеми з атомарністю:

В одній вимогі фактично міститься кілька незалежних (наприклад: «кнопка “Restart” не повинна відображатися при зупиненому сервісі, вікно “Log” має вміщувати не менше 20-ти записів про останні дії користувача» — тут навіщось в одній пропозиції описані абсолютно різні елементи інтерфейсу в різних контекстах).

Вимога допускає різночитання в силу граматичних особливостей мови (наприклад: «якщо користувач підтверджує замовлення та редагує замовлення або відкладає замовлення, повинен видаватися запит на оплату» — тут описані три різні випадки, і цю вимогу варто розбити на три окремі, щоб уникнути плутанини). Таке порушення атомарності часто тягне у себе виникнення суперечливості.

В одній вимогі об'єднано опис кількох незалежних ситуацій (наприклад: «коли користувач входить до системи, йому має відображатися привітання; коли користувач увійшов до системи, має відображатися ім'я користувача; коли користувач виходить із системи, має відображатися прощання» — всі ці три ситуації заслуговують того, щоб бути описаними окремими і більш детальними вимогами).

Несуперечність, послідовність. Вимога не повинна містити внутрішніх протиріч та протиріч іншим вимогам та документам. Типові проблеми з несуперечністю:

Суперечності всередині однієї вимоги (наприклад: «після успішного входу в систему користувача, який не має права входити в систему…» — тоді як він успішно увійшов до системи, якщо не мав такого права?)

Суперечності між двома і більше вимогами, між таблицею та текстом, малюнком та текстом, вимогою та прототипами тощо.

(наприклад: «712.a Кнопка “Close” завжди має бути червоною» і «364.x Кнопка “Close” завжди має бути синьою» — так все ж таки червоною чи синьою?)

Використання неправильної термінології або використання різних термінів для позначення одного і того ж об'єкта або явища (наприклад: «у випадку, якщо роздільна здатність вікна становить менше 800x600…» — роздільна здатність є у екрана, у вікна є розмір).

Недвозначність. Вимога повинна бути описана без використання жаргону, неочевидних абревіатур та розпливчастих формулювань, повинна допускати лише однозначне об'єктивне розуміння та бути атомарною в плані неможливості різного трактування поєднання окремих фраз. Типові проблеми з недвозначністю:

Використання термінів або фраз, що допускають суб'єктивне тлумачення (наприклад: «застосунок повинен підтримувати передачу великих обсягів даних» — наскільки «великих»?) Ось лише невеликий перелік слів і виразів, які можна вважати вірними ознаками двозначності: адекватно, бути здатним, легко забезпечувати як мінімум, бути здатним, ефективно, своєчасно, застосовно, якщо можливо й інші їм подібні.

Ось перебільшений приклад вимоги, що звучить дуже красиво, але абсолютно нереалізована і незрозуміло: «У разі необхідності оптимізації передачі великих файлів система повинна ефективно використовувати мінімум оперативної пам'яті, якщо це можливо».

Використання неочевидних або двозначних абревіатур без розшифровки (наприклад: «доступ до ФС здійснюється за допомогою системи прозорого шифрування» та «ФС надає можливість фіксувати повідомлення в їх поточному стані зі збереженням історії всіх змін» — ФС тут позначає файлову систему? Точно? А не який- або «Фіксатор Повідомлень»?)

Формулювання вимог з міркувань, що щось має бути всім очевидним (наприклад: «Система конвертує вхідний файл із формату PDF у вихідний файл формату PNG») і при цьому автор вважає цілком очевидним, що імена файлів система отримує з командного рядка, а багатосторінковий PDF конвертується кілька PNG-файлів, до імен яких

додається «page-1», «page-2» і т. п.). Ця проблема перегукується із порушенням коректності.

Виконаність. Вимога має бути технологічно здійсненною та реалізованою в рамках бюджету та термінів розробки проекту. Типові проблеми з здійсненністю:

Так зване «озолочення» – вимоги, які вкрай довго або дорого реалізуються і при цьому практично не приносять користі кінцевим користувачам (наприклад: «налаштування параметрів для підключення до бази даних має підтримувати розпізнавання символів з жестів, отриманих з пристроїв тривимірного введення»).

Технічно нереалізовані на сучасному рівні розвитку технологій вимоги (наприклад: «аналіз договорів повинен виконуватися із застосуванням штучного інтелекту, який виноситиме однозначний коректний висновок про ступінь вигоди від укладання договору»).

У принципі вимоги, що не реалізуються (наприклад: «система пошуку повинна заздалегідь передбачати всі можливі варіанти пошукових запитів і кешувати їх результати»).

Обов'язковість, потрібність та актуальність. Якщо вимога не є обов'язковою для реалізації, вона повинна бути просто виключена з набору вимог. Якщо вимога потрібна, але «не дуже важлива», для вказівки цього факту використовується вказівка на пріоритет. Також виключені (або перероблені) мають бути вимоги, що втратили актуальність. Типові проблеми з обов'язковістю та актуальністю:

Вимога була додана «про всяк випадок», хоча реальної потреби у ньому був і немає.

Вимоги виставлено неправильні значення пріоритету за критеріями важливості чи терміновості.

Вимога застаріла, але не була перероблена чи видалена.

Простежуваність. Простежуваність буває вертикальною та горизонтальною. Вертикальна дозволяє співвідносити між собою вимоги різних рівнях вимог, горизонтальна дозволяє співвідносити вимогу з тест-планом, тест-кейсами, архітектурними рішеннями тощо.

Для забезпечення простежуваності часто використовуються спеціальні інструменти управління вимогами або матриці простежуваності. Типові проблеми із простежуваністю:

Вимоги не пронумеровані, не структуровані, немає змісту, немає працюючих перехресних посилань.

При розробці вимог не були використані інструменти та техніки управління вимогами.

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

Модифікованість. Ця властивість характеризує простоту внесення змін до окремих вимог і набір вимог. Можна говорити про наявність модифікованості в тому випадку, якщо при доопрацюванні вимог потрібну інформацію легко знайти, а її зміна не призводить до порушення інших описаних у цьому переліку властивостей. Типові проблеми з модифікованістю:

Вимоги неатомарні і непростежувані, тому їх зміна з високою ймовірністю породжує суперечливість.

Вимоги спочатку суперечливі. У такій ситуації внесення змін (не пов'язаних з усуненням суперечливості) лише посилює ситуацію, збільшуючи суперечливість та знижуючи простежуваність.

Вимоги представлені у незручній для обробки формі (наприклад, не використані інструменти управління вимогами, і в результаті команді доводиться працювати з десятками великих текстових документів).

Проранжованість за важливістю, стабільністю, терміновістю . Важливість характеризує залежність успіху проекту від успіху реалізації вимоги. Стабільність характеризує ймовірність того, що в найближчому майбутньому в вимогу не буде внесено жодних змін. Терміновість визначає розподіл у часі зусиль проектної команди щодо реалізації тієї чи іншої вимоги.

Типові проблеми з проранжованістю полягають у її відсутності чи неправильній реалізації та призводять до наступних наслідків.

Проблеми з проранжованістю за важливістю підвищують ризик неправильного розподілу зусиль проектної команди, спрямування зусиль на другорядні завдання та кінцевого провалу проекту через

нездатність продукту виконувати ключові завдання з дотриманням ключових умов.

Проблеми з проранжованістю за стабільністю підвищують ризик виконання безглуздої роботи з удосконалення, реалізації та тестування вимог, які найближчим часом можуть зазнати кардинальних змін (аж до повної втрати актуальності).

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

Коректність та перевірюваність. Фактично ці властивості випливають із дотримання всіх перелічених вище (або можна сказати, що вони не виконуються, якщо порушено хоча б одне з вище перелічених). На додаток можна відзначити, що проверяемость передбачає можливість створення об'єктивного тест-кейсу, що однозначно показує, що вимога реалізована вірно і поведінка застосунку точно відповідає вимогі. До типових проблем із коректністю також можна віднести:

друкарські помилки (особливо небезпечні друкарські помилки в абревіатурах, що перетворюють одну осмислену абревіатуру в іншу також осмислену, але яка не має відношення до якогось контексту; такі друкарські помилки вкрай складно помітити);

наявність неаргументованих вимог до дизайну та архітектури;

погане оформлення тексту та супутньої графічної інформації, граматичні, пунктуаційні та інші помилки у тексті;

неправильний рівень деталізації (наприклад, надто глибока деталізація вимоги на рівні бізнес-вимог або недостатня деталізація на рівні вимог до продукту);

вимоги до користувача, а не до додатка (наприклад: «користувач повинен мати можливість надіслати повідомлення» — на жаль, ми не можемо впливати на стан користувача).

1.5.6. Техніки тестування вимог. Тестування документації та вимог відноситься до розряду нефункціонального тестування. Основні техніки такого тестування у тих вимог такі.

Взаємний перегляд. Взаємний перегляд («рецензування») є однією з технік тестування вимог, що найбільш активно використовуються, і може бути представлений в одній з трьох наступних форм (у міру наростання його складності і ціни):

1. Побіжний перегляд може виражатися як у показі автором своєї роботи колегам з метою створення спільного розуміння та отримання зворотного зв'язку, так і в простому обміні результатами роботи між двома та більше авторами для того, щоб колега висловив свої питання та зауваження. Це найшвидший, найдешевший і найчастіше використовуваний вид перегляду. Для запам'ятовування: аналог швидкого перегляду — це ситуація, коли ви в школі з однокласниками перевіряли перед здаванням твору один одного, щоб знайти описки та помилки.

2. Технічний перегляд виконується групою спеціалістів. В ідеальній ситуації кожен фахівець має представляти свою галузь знань. Тестований продукт не може вважатися досить якісним, поки хоча б у одного переглядача залишаються зауваження. Для запам'ятовування: аналог технічного перегляду — ситуація, коли якийсь договір візує юридичний відділ, бухгалтерія тощо.

3. Формальна інспекція є структурованим, систематизованим і документованим підходом до аналізу документації. Для його виконання залучається велика кількість фахівців, саме виконання займає чимало часу, і тому цей варіант перегляду використовується досить рідко (як правило, при отриманні на супровід та доопрацювання проекту, створенням якого раніше займалася інша компанія). Для запам'ятовування: аналог формальної інспекції - це ситуація генерального прибирання квартири (включаючи вміст усіх шаф, холодильника, комори тощо).

Запитання. Наступною очевидною технікою тестування та підвищення якості вимог є (повторне) використання технік виявлення вимог, а також (як окремий вид діяльності) – питання. Якщо хоч щось у вимогах викликає у вас нерозуміння чи підозра — запитуйте. Можна спитати представників замовника, можна звернутися до довідкової інформації. З багатьох питань можна звернутися до досвідченіших колег за умови, що вони мають відповідну інформацію, раніше отриману

від замовника. Головне, щоб ваше питання було сформульоване таким чином, щоб отримана відповідь дозволила покращити вимоги.

Оскільки тут тестувальники-початківці допускають безліч помилок, розглянемо докладніше. У табл. 1.2 наведено кілька погано сформульованих вимог, а також прикладів поганих та добрих питань. Погані питання провокують бездумні відповіді, які містять корисної інформації.

Таблиця 1.2

Приклад поганих та добрих питань до вимог

Погана вимога	Погані питання	Хороші питання
1	2	3
"Додаток повинен швидко запускатися".	"Наскільки швидко?" (На це ви ризикуєте отримати відповіді у стилі «дуже швидко», «максимально швидко», « нууу … просто швидко»). "А якщо не вийде швидко?" (Цим ви ризикуєте просто здивувати або навіть роздратувати замовника.) "Завжди?" («Так, завжди». Хм, а ви чекали іншої відповіді?)	«Який максимально допустимий час запуску програми, на якому обладнанні та за якої завантаженості цього обладнання операційною системою та іншими додатками? На досягнення яких цілей впливає швидкість запуску програми? Чи дозволяється фонове завантаження окремих компонентів програми? Що є критерієм того, що програма закінчила запуск?»
«Опційно має підтримуватись експорт документів у формат PDF».	«Будь-яких документів?» (Відповіді «так, будь-яких» чи «ні, тільки відкритих» вам все одно не допоможуть.) «У PDF якої версії повинен вироблятися експорт?» (Сам собою питання хороше, але він не дає зрозуміти, що мало на увазі під «опціонально».) «Навіщо?» («Потрібно!» Саме так хочеться відповісти, якщо питання не розкрите повністю.)	«Наскільки можливість експорту до PDF важлива? Як часто, ким і з якою метою вона використовуватиметься? Чи є PDF єдиним допустимим форматом для цих цілей, чи є альтернативи? Чи дозволяється використання зовнішніх утиліт (наприклад, віртуальних PDF-принтерів) для експорту документів у PDF?»

Продовження табл. 1.2

1	2	3
"Якщо дата події не вказана, вона вибирається автоматично".	"А якщо вказана?" (То вона вказана. Логічно, чи не так?) "А якщо дату неможливо вибрати автоматично?" (Саме питання цікаве, але без пояснення причин неможливості звучить як знущання.) "А якщо у події немає дати?" (Тут автор питання, швидше за все, хотів уточнити, чи обов'язково це поле для заповнення. Але із самої вимоги видно, що обов'язково: якщо воно не заповнене людиною, його має заповнити комп'ютер.)	«Можливо, йшлося про те, що дата генерується автоматично, а не вибирається? Якщо так, то яким алгоритмом вона генерується? Якщо ні, з якого набору вибирається дата і як генерується цей набір? PS Можливо, чи варто використовувати поточну дату?»

Тест-кейси та чек-листи. Ми пам'ятаємо, що хороша вимога є перевіреною, а отже, повинні існувати об'єктивні способи визначення того, чи правильно реалізовано вимогу. Продумування чек-листів чи навіть повноцінних тест-кейсів у процесі аналізу вимог дозволяє нам визначити, наскільки вимогу перевіряємо. Якщо ви можете швидко вигадати кілька пунктів чек-листа, це ще не ознака того, що з вимогою все добре (наприклад, воно може суперечити якимось іншим вимогам). Але якщо жодних ідей щодо тестування вимоги на думку не спадає — це тривожний знак.

Рекомендується спочатку переконатися, що ви розумієте вимогу (зокрема прочитати сусідні вимоги, поставити питання колегам тощо). Також можна поки що відкласти роботу з цією конкретною вимогою і повернутися до неї пізніше — можливо, аналіз інших вимог дозволить вам краще зрозуміти і це конкретне. Але якщо ніщо не допомагає, швидше за все, з вимогою щось не так.

Заради справедливості слід зазначити, що на початковому етапі опрацювання вимог такі випадки трапляються дуже часто — вимоги

сформовані дуже поверхово, розпливчасто і явно потребують доопрацювання, тобто тут немає необхідності проводити складний аналіз, щоб констатувати неперевірюваність вимоги.

На стадії ж, коли вимоги вже добре сформульовані та протестовані, ви можете продовжувати використовувати цю техніку, поєднуючи розробку тест-кейсів та додаткове тестування вимог.

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

Малюнки (графічне уявлення). Щоб побачити загальну картину вимог цілком, дуже зручно використовувати малюнки, схеми, діаграми, інтелект-карти тощо. Графічне подання зручне одночасно своєю наочністю і стислою (наприклад, UML-схема бази даних, що займає один екран, може бути описана кількома десятками сторінок тексту). На малюнку дуже легко помітити, що якісь елементи «не стикуються», де чогось не вистачає і т. п. Якщо ви для графічного подання вимог використовуватимете загальноприйняту нотацію (наприклад, вже згаданий UML), ви отримаєте додаткові переваги: вашу схему зможуть легко розуміти і доопрацьовувати колеги, а в результаті може вийти гарний додаток до текстової форми подання вимог.

Прототипування. Можна сміливо сказати, що прототипування часто є наслідком створення графічного уявлення та аналізу поведінки системи. З використанням спеціальних інструментів можна дуже швидко зробити нариси інтерфейсів користувача, оцінити застосовність тих чи інших рішень і навіть створити не просто «прототип заради прототипу», а заготівлю для подальшої розробки, якщо виявиться, що

реалізоване в прототипі (можливо, з невеликими доробками) влаштовує замовника.

1.5.7. Приклад аналізу та тестування вимог. Оскільки наше завдання полягає в тому, щоб сформувати розуміння логіки аналізу та тестування вимог, ми розглядатимемо гранично короткий та простий їх набір.

Припустимо, що у якогось клієнта є проблема: текстові файли, що надходять у величезній кількості його співробітникам, приходять у різних кодуваннях, і співробітники витрачають багато часу на перекодування («ручний підбір кодування»). Відповідно, він хотів би мати інструмент, що дозволяє автоматично приводити кодування всіх текстових файлів до якоїсь однієї. Отже, у світ з'являється проект з кодовою назвою «Конвертер файлів».

Рівень бізнес-вимог. Бізнес-вимоги спочатку можуть мати такий вигляд: «Необхідний інструмент для автоматичного приведення кодувань текстових документів до одного».

Тут ми можемо поставити багато запитань. Для зручності наведемо як самі питання, і передбачувані відповіді клієнта (табл. 1.3).

Таблиця 1.3

Формулювання бізнес-вимог

Запитання	Відповіді
У яких форматах представлені текстові документи (звичайний текст, HTML, MD, щось інше)?	Поняття не маю, я в цьому не розумію.
В які кодування приходять вихідні документи?	В різних.
На яке кодування потрібно перетворити документи?	У найзручнішу та універсальну.
Якими мовами написаний текст у документах?	Російська та англійська.
Звідки і як надходять текстові документи (поштою, сайтами, мережею, якось інакше)?	Це не важливо. Надходять звідусіль, але ми складаємо їх в одну папку на диску, нам так зручно.
Який максимальний обсяг документа?	Пара десятків сторінок.
Як часто з'являються нові документи (наприклад, скільки максимум документів може надійти за годину)?	200–300 за годину.
За допомогою чого працівники переглядають документи?	Notepad ++.

Навіть таких питань і відповідей достатньо, щоб переформулювати бізнес-вимоги в такий спосіб (зверніть увагу, що багато питань було поставлено на майбутнє і не призвело до появи в бізнес-вимогах зайвої технічної деталізації).

Суть проекту: розробка інструменту, що усуває проблему множинності кодувань у текстових документах, що у локальному дисковому сховищі.

Мета проекту:

Виключення необхідності ручного підбору коду текстових документів.

Скорочення часу роботи з текстовим документом на величину, необхідну ручного підбору кодування.

Метрики досягнення цілей:

Повна автоматизація визначення та перетворення кодування текстового документа до заданого.

Скорочення часу обробки текстового документа в середньому на 1–2 хвилини на документ за рахунок усунення ручного підбору кодування.

Ризики:

Висока технічна складність безпомилкового визначення вихідного кодування текстового документа.

Чому ми вирішили, що середній час на вибір кодування становить 1–2 хвилини? Ми провели спостереження. Також ми пам'ятаємо відповіді замовника на запитання про вихідні формати документів, вихідні та кінцеві кодування (замовник чесно сказав, що не знає відповіді), а тому ми попросили його надати нам доступ до сховища документів та з'ясували:

- Вихідні формати: plain text , HTML, MD.
- Вихідні кодування: CP1251, UTF8, CP866, KOI8R.
- Цільове кодування: UTF8.

На цьому етапі цілком можемо вирішити, що варто зайнятися деталізацією вимог більш низьких рівнях, так як питання, що з'явилися там, дозволять нам повернутися до бізнес-вимог і поліпшити їх, якщо в цьому виникне необхідність.

Рівень користувальницьких вимог. Настав час зайнятися рівнем потреб користувача. Проект у нас дещо специфічний — результатами роботи програмного засобу користуватиметься велика кількість людей, але сам програмний засіб при цьому вони не використовуватимуть (воно буде виконувати свою роботу «само по собі» — запущене на сервері зі сховищем документів). Тому під користувачем тут ми розумітимемо людину, яка налаштовує роботу програми на сервері.

У процесі аналізу текст вимог набуде такого вигляду.

Системні характеристики

СХ-1: Додаток є консольним.

СХ-2: Для роботи програма використовує інтерпретатор PHP.

Яка мінімальна версія інтерпретатора PHP підтримується програмою? (5.5.x)

Чи існує певна специфіка налаштування інтерпретатора PHP для коректної роботи програми? (Напевно, має працювати mbstring .)

Чи наполягаєте ви на реалізації програми саме на PHP? Якщо так, то чому. (Так, тільки PHP. У нас є співробітник, який його знає.)

Чи повинна бути описана в посібнику користувача процедура встановлення та налаштування інтерпретатора PHP? (Ні.)

СХ-3: Додаток є кросплатформним.

Які ОС мають підтримуватись? (Будь-яка, де працює PHP.)

У чому взагалі мета кроссплатформенності? (Ми ще не знаємо, на чому працюватиме сервер.)

Користувальницькі вимоги

ПТ-1: Запуск та зупинення програми.

ПТ-1.1: Запуск програми здійснюється з консолі командою PHP *(можливо, тут помилка: має бути php (у нижньому регістрі)) (Так, ОК.)* converter.php параметри.

Які параметри передаються скрипту під час запуску? (Каталог із вихідними файлами, каталог із кінцевими файлами.)

Яка реакція скрипту на:

· *Відсутність параметрів ; (Пише хелп .)*

· *Неправильна кількість параметрів; (Пише Хелп і пояснює, що не так.)*

· *Неправильні значення кожного з параметрів. (Пише Хелп і пояснює, що не так.)*

ПТ-1.2: Зупинення програми здійснюється виконанням команди Ctrl+C *(пропонуємо доповнити цей вираз фразою «у вікні консолі, з якого було запущено додаток») (Так, ОК.).*

ПТ-2: Налаштування програми.

ПТ-2.1: Конфігурація програми зводиться до вказівок шляхів у файловій системі.

Шляхів до чого? (Каталог із вихідними файлами, каталог із кінцевими файлами .)

ПТ-2.2: Цільовим кодуванням є UTF8.

Чи передбачається інше цільове кодування, чи UTF8 використовується як цільова завжди? (Тільки UTF8, інших не треба.)

ПТ-3: Перегляд журналу роботи програми.

ПТ-3.1: У процесі роботи програма повинна виводити журнал своєї роботи в консоль та лог-файл.

Який формат журналу? (Дата-час, що і з чим робили, що вийшло. Гляньте в логі апача, там нормально написано.)

Чи відрізняються формати журналу для консолі та лог-файлу? (Ні.)

Як визначається ім'я файлу лог-файлу? (Третій параметр при запуску. Якщо не вказано - нехай буде converter.log поряд з php - скриптом.)

ПТ-3.2: При першому запуску програми лог-файл створюється, а за наступних – дописується.

Як додаток розрізняє свій перший та наступні запуски? (Ніяк.)

Якою є реакція програми на відсутність лог-файлу у випадку, якщо це не перший запуск? (Створює. Тут ідея в тому, щоб воно не затирало старе ліг - і все.)

Бізнес-правила

БП-1: Джерело та приймач файлів

БП-1.1: Каталоги, що є джерелом вихідних та приймачем кінцевих файлів, не повинні збігатися.

Яка реакція програми у разі збігу цих каталогів? (Пише Хелп і пояснює, що не так.)

БП-1.2: Каталог, що є приймачем кінцевих файлів, не може бути підкаталогом джерела *(пропонуємо замінити слово "джерела" на фразу "каталогу, що є джерелом вихідних файлів"). (Добре нехай буде так.)*

Атрибути якості

АК-1: Продуктивність

АК-1.1: Програма повинна забезпечувати швидкість обробки даних 5 МБ/сек.

За яких технічних характеристик системи? (i7, 4GB RAM)

АК-2: Стійкість до вхідних даних

АК-2.1: Програма має обробляти вхідні файли розміром до 50 МБ включно.

Якою є реакція програми на файли, розмір яких перевищує 50 МБ? (Не чіпає.)

АК-2.2: Якщо вхідний файл не є текстовим, програма повинна зробити обробку.

Обробку чого має зробити додаток? (Цього файлу. Не важливо, що станеться з файлом, аби скрипт не помер.)

Тут є кілька важливих моментів, на які варто звернути увагу:

Відповіді замовника можуть бути менш структурованими та послідовними, ніж наші запитання. Це нормально. Він може дозволити собі таке, ми ні.

Відповіді замовника можуть містити суперечності (у нашому прикладі спочатку замовник писав, що параметрами, що передаються з командного рядка, є лише два імені каталогу, а потім сказав, що там вказується ім'я лог-файлу). Це теж нормально, так як замовник міг щось забути або переплутати. Наше завдання — звести ці суперечливі дані докупи (якщо це можливо) і поставити уточнюючі питання (якщо це необхідно).

Якщо з нами спілкується технічний фахівець, у його відповідях цілком можуть проскакувати технічні жаргонізми (як "хелп" у нашому прикладі). Не треба перепитувати його про те, що це таке, якщо жаргонізм має однозначне загальноприйняте значення, але при доопрацюванні тексту наше завдання — написати таку саму строгу технічну мову. Якщо жаргонізм все ж таки незрозумілий — тоді краще

запитати (так, «хелп» — це лише коротка допомога, що виводиться консольними додатками як підказка про те, як їх використовувати).

Рівень продуктних вимог.

Застосуємо так званий «самостійний опис» та покращимо вимоги. Оскільки ми отримали багато специфічної технічної інформації, можна паралельно писати повноцінну специфікацію вимог. У багатьох випадках, коли для оформлення вимог використовується простий текст, для зручності формується єдиний документ, який інтегрує в собі як вимоги користувача, так і детальні специфікації. Тепер вимоги набувають такого вигляду.

Системні характеристики

СХ-1: Додаток є консольним.

СХ-2: Додаток розробляється мовою програмування PHP (причину вибору мови PHP відображено в пункті О-1 розділу «Обмеження», особливості та важливі налаштування інтерпретатора PHP відображено в пункті ДС-1 розділу «Детальні специфікації»).

СХ-3: Програма є кросплатформною з урахуванням пункту О-4 розділу «Обмеження».

Користувальницькі вимоги

ПТ-1: Запуск та зупинення програми.

ПТ-1.1: Запуск програми провадиться з консолі командою «php converter.php SOURCE_DIR DESTINATION_DIR [LOG_FILE_NAME]» (опис параметрів наведено в розділі ДС-2.1, реакцію на помилки при вказівці параметрів наведено в розділах ДС-2.2, ДС-2.3, ДС-2.4).

ПТ-1.2: Зупинення програми здійснюється виконанням команди Ctrl+C у вікні консолі, з якого було запущено програму.

ПТ-2: Налаштування програми.

ПТ-2.1: Налаштування програми зводиться до вказівки параметрів командного рядка (див. ДС-2).

ПТ-2.2: Цільовим кодуванням перетворення текстів є кодування UTF8 (також див. О-5).

ПТ-3: Перегляд журналу роботи програми.

ПТ-3.1: У процесі роботи додаток повинен виводити журнал своєї роботи в консоль та лог-файл (див. ДС-4), ім'я якого визначається правилами, зазначеними у ДС-2.1.

ПТ-3.2: Формат журналу роботи та лог файлу вказаний у ДС-4.1, а реакція програми на наявність або відсутність лог-файлу вказана у ДС-4.2 та ДС-4.3 відповідно.

Бізнес-правила

БП-1: Джерело та приймач файлів

БП-1.1: Каталоги, що є джерелом вихідних та приймачем кінцевих файлів, не повинні збігатися (див. також ДС-2.1 та ДС-3.2).

БП-1.2: Каталог, що є приймачем кінцевих файлів, не може перебувати всередині каталогу, що є джерелом вихідних файлів або його підкаталогів (див. також ДС-2.1 та ДС-3.2).

Атрибути якості

АК-1: Продуктивність

АК-1.1: Додаток повинен забезпечувати швидкість обробки даних не менше 5 МБ/с на апаратному забезпеченні, еквівалентному наступному: процесор i7, 4 ГБ оперативної пам'яті, середня швидкість читання/запису на диск 30 МБ/с. Також див. О-6.

АК-2: Стійкість до вхідних даних

АК-2.1: Вимоги щодо форматів файлів, що обробляються, викладені в ДС-5.1.

АК-2.2: Вимоги щодо розмірів файлів, що обробляються, викладені в ДС-5.2.

АК-2.3: Поведінка програми в ситуації обробки файлів з порушеннями формату визначена у ДС-5.3.

Обмеження

О-1: Додаток розробляється мовою програмування PHP, використання якого обумовлено можливістю замовника здійснювати підтримку програми силами власного IT-відділу.

О-2: Обмеження щодо версії та налаштувань інтерпретатора PHP відображені у пункті ДС-1 розділу «Детальні специфікації».

В-3: Процедури встановлення та налаштування інтерпретатора PHP виходять за рамки даного проекту і не описуються в документації.

О-4: Кросплатформні можливості програми зводяться до здатності працювати під ОС сімейства Windows та Linux , що підтримують роботу інтерпретатора PHP версії, вказаної в ДС-1.1.

О-5: Цільове кодування UTF8 є жорстко заданим, і його зміна в процесі експлуатації програми не передбачено.

О-6: Допускається невиконання АК-1.1 у випадку, якщо неможливість забезпечити заявлену продуктивність обумовлена зовнішніми причинами (наприклад, технічними проблемами на сервері замовника).

Детальні специфікації

ДС-1: Інтерпретатор PHP

ДС-1.1: Мінімальна версія - 5.5.

ДС-1.2: Для роботи програми має бути встановлене та включене розширення mbstring .

ДС-2: Параметри командного рядка

ДС-2.1: При запуску програми воно отримує з командного рядка три параметри:

SOURCE_DIR - обов'язковий параметр, що визначає шлях до каталогу з файлами, які необхідно обробити;

DESTINATION_DIR — обов'язковий параметр, що визначає шлях до каталогу, в який необхідно помістити оброблені файли (цей каталог не може знаходитися всередині каталогу SOURCE_DIR або в його підкаталогах (див. БП-1.1 та БП-1.2));

LOG_FILE_NAME — необов'язковий параметр, що визначає повне ім'я лог-файлу (за замовчуванням лог-файл з ім'ям «converter.log» розміщується тим же шляхом, яким знаходиться файл скрипта converter.php);

ДС-2.2: Якщо вказати недостатню кількість параметрів командного рядка, програма повинна завершити роботу, видавши повідомлення про використання (ДС-3.1).

ДС-2.3: У разі надмірної кількості параметрів командного рядка додаток повинен ігнорувати всі параметри командного рядка, крім зазначених у пункті ДС-2.1.

ДС-2.4: Якщо вказати неправильне значення будь-якого з параметрів командного рядка, програма повинна завершити роботу, видавши повідомлення про використання (ДС-3.1), а також повідомивши ім'я невірно вказаного параметра, його значення та суть помилки (див. ДС-3.2).

ДС-3: Повідомлення

ДС-3.1: Повідомлення про використання: "USAGE converter.php SOURCE_DIR DESTINATION_DIR LOG_FILE_NAME".

ДС-3.2: Повідомлення про помилки: Directory not exists or inaccessible .Destination dir may not reside within source dir tree . Wrong file name or inaccessible path .

ДС-4: Журнал роботи

ДС-4.1: Формат журналу роботи однаковий для відображення в консолі та запису в лог-файл: YYYY-MM-DD HH:II:SS имя_операции параметри_операції результат_операції .

ДС-4.2: Якщо лог-файл відсутній, повинен бути створений новий пустий лог-файл.

ДС-4.3: Якщо лог-файл вже існує, має відбуватися додавання нових записів у його кінець.

ДС-5: Формати та розміри файлів

ДС-5.1: Додаток повинен обробляти текстові файли російською та англійською мовами в наступних кодуваннях: WIN1251, CP866, KOI8R.

Оброблювані файли можуть бути представлені в наступних форматах, які визначаються розширеннями файлів:

- Plain Text (TXT);
- Hyper Text Markup Language Document (HTML);
- Mark Down Document (MD).

ДС-5.2: Програма має обробляти файли розміром до 50 МБ (включно), ігноруючи будь-який файл, розмір якого перевищує 50 МБ.

ДС-5.3: Якщо файл з розширенням ДС-5.1 містить у собі дані, що не відповідають формату файлу, допускається пошкодження таких даних.

Отже, ми отримали набір вимог, з якими цілком можна працювати. Він не ідеальний (і ніколи ви не зустрінете ідеальних вимог), але він цілком придатний для того, щоб розробники змогли реалізувати додаток, а тестувальники протестувати його.