

Ірина СПІВАК, Світлана КРЕПИЧ



Якість програмного забезпечення в контексті аналізу вимог

Підручник



Тернопіль, 2025

УДК 004.05

*Затверджено Вченою радою
Західноукраїнського національного університету
(протокол № 7 від 25 березня 2026 р.) та
присвячено 60-річчю Західноукраїнського національного університету*

Рецензенти:

Ігор БАРАН - кандидат техн. наук, доцент, декан факультету
комп'ютерно-інформаційних систем і програмної інженерії
Тернопільського національного технічного університету ім. І.Пулюя.
Юрій ФРАНКО - кандидат техн. наук, доцент, завідувач кафедри
комп'ютерних технологій Тернопільського національного
педагогічного університету ім. В.Гнатюка.
Сергій КУТУЗОВ - Керівник Тернопільської філії ТзОВ «Елекс»

Співак І.Я., Крепич С.Я. Якість програмного забезпечення в
контексті аналізу вимог: Підручник. Тернопіль: ВПЦ
«Університетська думка», 2026. – 318с.

ISBN 978-966-654-827-9

Підручник «Якість програмного забезпечення в контексті аналізу вимог» присвячений розгляду теоретичних і практичних аспектів забезпечення якості програмного забезпечення на етапі аналізу та формування вимог. В підручнику висвітлено роль інженерії вимог у процесі розроблення програмного забезпечення, розкрито підходи до виявлення, структурування, документування та перевірки вимог, а також їх вплив на формування якісного програмного продукту. Особливу увагу приділено сучасним методам аналізу вимог, використанню моделей, діаграм і специфікацій, а також взаємодії між зацікавленими сторонами під час формування вимог до системи.

ISBN 978-966-654-827-9

**Співак І.Я., Крепич С.Я., 2026
ЗУНУ, 2026**

ЗМІСТ

	<i>ПЕРЕДМОВА</i>	6
1	<i>БІЗНЕС АНАЛІЗ В ІТ ТА ОСНОВИ РОБОТИ З ВИМОГАМИ</i>	8
1.1	Аналіз вимог в процесі розроблення програмного забезпечення	8
1.2	Проблеми встановлення вимог до програмного забезпечення	12
1.3	Інженерія вимог до програмного забезпечення	16
1.4	Основні кроки процедури аналізу вимог	18
1.5	Проблеми аналізу вимог до програмного забезпечення	26
	<i>Питання для самоперевірки</i>	31
2	<i>ВИМОГИ. ЯКІСТЬ ВИМОГ. ДОКУМЕНТУВАННЯ ВИМОГ</i>	34
2.1	Поняття про вимоги до програмного забезпечення	34
2.2	Категоризація вимог до програмного забезпечення	37
2.3	Функціональні та нефункціональні вимоги	41
2.4	Джерела та методи виявлення вимог до програмного забезпечення	44
2.5	Життєвий цикл вимог до програмного забезпечення	47
2.6	Якість вимог до програмного забезпечення. «Будинок» якості вимог	49
2.7	Документування вимог до програмного забезпечення	63
	<i>Питання для самоперевірки</i>	78
3	<i>МОДЕЛЮВАННЯ ВИМОГ: UML, BPMN</i>	81
3.1	Сутність моделювання вимог до програмного забезпечення	81
3.2	<i>Use Case Diagram</i> – варіанти використання	83
3.3	<i>State Diagram</i> – зміни станів системи	90
3.4	<i>Activity Diagram</i> – діаграма діяльності	95
3.5	<i>Sequence Diagram</i> – послідовність дій	101
3.6	Нотація моделювання бізнес-процесів (<i>BPMN</i> – діаграми)	108
3.7	<i>UML</i> чи <i>BPMN</i> ?	114

	<i>Питання для самоперевірки</i>	115
4	ПРІОРИТЕЗАЦІЯ ФУНКЦІЙ ТА ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	118
4.1	Методологія визначення пріоритетів	118
4.2	Кількісні методи пріоритезації	119
4.3	Якісні методи пріоритезації	138
	<i>Питання для самоперевірки</i>	159
5	РОЛЬОВА МОДЕЛЬ КОРИСТУВАЧІВ: User Story, Backlog	161
5.1	Що ж таке історії користувачів та їх роль в процесі аналізу вимог	161
5.2	Backlog та декомпозиція User Stories	168
5.3	Як писати ефективні User Stories	176
5.4	Критерії прийняття та завершеності User Stories	185
5.5	User Story Mapping: від ідеї до релізу	192
5.6	Правила написання історій користувачів та критеріїв прийняття	205
	<i>Питання для самоперевірки</i>	214
6	ПРОТОТИПИ ЯК ЗАСІБ ЗМЕНШЕННЯ РИЗИКІВ	216
6.1.	Що таке прототип програмного продукту і для чого він потрібний?	217
6.2.	Типи прототипів програмного забезпечення	221
6.3.	Ризики прототипування	235
6.4.	Фактори успіху прототипування	237
6.5.	Інструменти створення електронних прототипів	239
	<i>Питання для самоперевірки</i>	248
7	ТРАСОВАНІСТЬ ТА ВАЛІДАЦІЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	250
7.1.	Встановлення зв'язків між вимогами	250
7.2.	Відображення елементарних зв'язків між вимогами	256
7.3.	Матриця покриття та матриця відстеження вимог	260
7.4.	Валідація вимог до програмного забезпечення	265
	<i>Питання для самоперевірки</i>	269

8	<i>ПОГОДЖЕННЯ ТА УПРАВЛІННЯ ВИМОГАМИ</i>	271
8.1.	Погодження вимог до програмного забезпечення	271
8.2.	Перегляд вимог до програмного забезпечення	275
8.3.	Управління змінами вимог	284
8.4.	Управління ризиками	290
	<i>Питання для самоперевірки</i>	297
	<i>ГЛОСАРІЙ ПРОФЕСІЙНИХ ТЕРМІНІВ</i>	300
	<i>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ</i>	306

ЯКІСТЬ ПЗ

ПЕРЕДМОВА

Сучасний розвиток інформаційних технологій супроводжується постійним зростанням складності програмних систем, що, у свою чергу, підвищує вимоги до якості їх проєктування та розроблення. Одним із ключових факторів успіху будь-якого ІТ-проєкту є чітке розуміння потреб користувачів і бізнесу, а також правильне формулювання, аналіз і управління вимогами до програмного забезпечення. Саме тому бізнес-аналіз та інженерія вимог займають центральне місце у сучасній практиці розроблення програмних продуктів.

Підручник «Якість програмного забезпечення в контексті аналізу вимог» присвячений системному викладенню теоретичних основ і практичних підходів до аналізу, формування, документування та управління вимогами в ІТ-проєктах. У ньому розглядаються ключові поняття бізнес-аналізу, процеси інженерії вимог, проблеми встановлення та аналізу вимог до програмного забезпечення, а також сучасні методи їх вирішення.

Особливу увагу приділено класифікації вимог, їх якості та способам забезпечення цієї якості. Розглянуто життєвий цикл вимог, джерела та методи їх виявлення, а також підходи до документування, включаючи створення специфікації вимог до програмного забезпечення. Значне місце відведено моделюванню вимог за допомогою нотацій UML та BPMN, що дозволяє підвищити зрозумілість і формалізованість опису системи.

Підручник також охоплює сучасні підходи до роботи з вимогами в гнучких методологіях розробки, зокрема використання *User Stories*, *Backlog*, технік пріоритезації та *User Story Mapping*. Окремі розділи присвячені прототипуванню як засобу зменшення ризиків, забезпеченню трасованості вимог, їх валідації та верифікації, а також процесам погодження, управління змінами та ризиками.

Матеріал структуровано таким чином, щоб забезпечити логічну послідовність вивчення - від базових понять до практичних інструментів і методів. Кожний розділ супроводжується питаннями для самоперевірки, що сприяють закріпленню знань та розвитку аналітичного мислення.

Підручник «Якість програмного забезпечення в контексті аналізу вимог» буде корисним для студентів спеціальностей з інформаційних технологій, бізнес-аналітики, інженерії програмного забезпечення, а також для бізнес-аналітиків-початківців, які прагнуть систематизувати знання та підвищити ефективність роботи з вимогами.

ЯКІСТЬ ПЗ

ТЕМА 1

БІЗНЕС-АНАЛІЗ В ІТ ТА ОСНОВИ РОБОТИ З ВИМОГАМИ

1.1. Аналіз вимог в процесі розроблення програмного забезпечення

Процедура аналізу вимог до програмного забезпечення полягає у визначенні потреб та умов, які висувають різні Замовники щодо можливостей нового програмного продукту чи його модифікації. При цьому потрібно враховувати можливі суперечності між різними вимогами та вирішувати будь-які конфліктні ситуації між Замовниками програмного забезпечення (далі Замовник) та Виконавцями робіт із розроблення програмного забезпечення (далі Виконавець).

Аналіз вимог до програмного забезпечення є критичною стадією реалізації програмного проекту, від якісного виконання якої залежить успішна його реалізація.

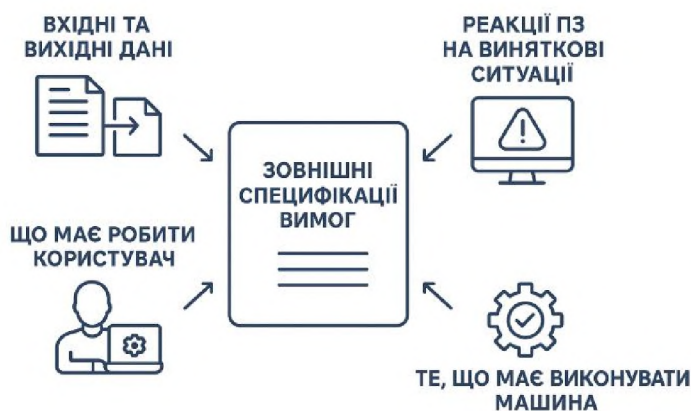
Вимоги мають бути задокументованими, вимірними, протестованими, пов'язаними з бізнес-потребами, а також описані з рівнем деталізації, достатнім для проектування програмного продукту.

Аналізують вимоги до програмного забезпечення за архітектурними, структурними, поведінковими, функціональними та експлуатаційними характеристиками нового чи модифікованого програмного продукту.

Аналіз вимог до архітектури програмного забезпечення та їх документування значно спрощує процес комунікації між зацікавленими сторонами, дає змогу зафіксувати прийняті на ранніх етапах проектування рішення про якісний дизайн системи, а також дає можливість використовувати компоненти цього дизайну і шаблони проектування повторно в інших програмних проектах.

Аналіз вимог до структури окремих програмних компонент і організації файлової системи для міжпрограмного обміну даними стосується переважно підготовки програмної документації. Розрізняють зовнішню програмну документацію, яка узгоджується із Замовником, і проміжну - внутрішню документацію до програмного проекту.

Під час складання програмної документації спочатку розробляють зовнішні, а потім - внутрішні специфікації вимог.



Зовнішні специфікації вимог містять усе те, що побачив би користувач під час отримання готового програмного продукту. Аналіз зовнішньої специфікації вимог залежить від тривалості та стадій життєвого циклу

програмного забезпечення.

Внутрішні специфікації вимог подають разом із описом архітектури програмного комплексу і внутрішньої структури окремих програмних компонент.



Вимоги користувачів до програмного забезпечення подають у вигляді тверджень про їхні потреби, виражені користувачем чи групою осіб, зацікавлених у якісному створенні майбутнього програмного продукту та в успішному його використанні. Вони стосуються багатьох особливостей функціонування програмного забезпечення, зокрема поведінкових. Загалом

поведінкові вимоги користувачів до програмного забезпечення можна поділити на два типи:

- ✓ функціональні вимоги, які встановлюють, що саме має бути реалізовано в програмному забезпеченні;
- ✓ вимоги до програмної продуктивності, які встановлюють, наскільки добре це має бути виконано.

Функціональні вимоги до програмного забезпечення описують внутрішню роботу програмного продукту, його поведінку у випадках калькулювання даних, маніпулювання даними, обробку даних та інші специфічні функції, які він має виконувати. На відміну від нефункціональних вимог, які встановлюють, яким має бути програмний продукт, функціональні вимоги встановлюють, що продукт має робити.

Функціональні вимоги до програмного забезпечення мають бути з'ясовані на першій стадії процесу розроблення програмного забезпечення, тобто на етапі аналізу вимог.

Нефункціональні вимоги (експлуатаційні) до програмного забезпечення задають критерії оцінювання якості його роботи. На відміну від функціональних вимог, які встановлюють, що програмний продукт має робити, нефункціональні вимоги встановлюють, яким продукт має бути.

Нефункціональні вимоги мають бути з'ясовані також на етапі аналізу вимог до програмного забезпечення.

За призначенням нефункціональні вимоги до програмного забезпечення спрямовані на покращення безпеки програмного забезпечення, його надійності, швидкодії, зручності у використанні тощо, а також на вдосконалення (масштабування, відновлюваність, тощо) властивостей програмного забезпечення.

Тепер розберемося з професіями в ІТ, хто з них має і яке відношення до виявлення та аналізу вимог. Якщо із розробниками (*Developer, Dev*) і тестувальниками (*Quality Assurance, QA*) зрозуміло - одні продукт створюють на основі вимог до програмного забезпечення, а інші тестують правильність цієї роботи, то з власником продукту (*Product Owner, PO*), менеджером проєкту (*Project Manager* або *Product Manager, PM*) та бізнес-аналітиком (*Business Analyst, BA*) все не настільки однозначно.

Менеджер проєкту – це стратегічна роль в ІТ-компанії, яка існує незалежно від конкретної методології, яка реалізується компанією (*Agile, Scrum, SAFe* тощо). Ключовими обов'язками менеджера проєкту є формування бачення продукту (*Vision*), визначення функцій продукту (*Features*) та відповідальність за їх реалізацію (*Program Backlog*), аналіз ринку та конкурентів, визначення цільової аудиторії, створення бізнес-моделі та її монетизація, взаємодія зі стейкхолдерами та керівництвом. Менеджер проєкту працює на рівні декількох команд.

Власник продукту – це тактична роль, яка походить із фреймворку *Scrum*. Тут основний фокус направлений на максимізацію цінності продукту через управління беклогом. Тому до ключових обов'язків належить формування та пріоритезація беклогу (*Product Backlog*), розбиття функцій на користувацькі історії, чітке формулювання вимог, комунікація з командою розроблення програмного забезпечення, прийняття результатів спринтів та уточнення вимог під час розроблення. Власник продукту це «виконавчий представник» продуктового бачення в межах однієї команди.

В той же час різниця між власником продукту і бізнес-аналітиком полягає у рівні відповідальності, прийнятті рішень та фокусі роботи з вимогами.

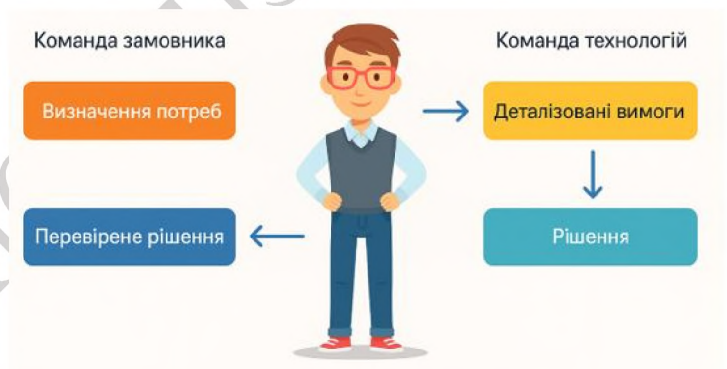
Якщо власник продукту формує і пріоритезує беклог продукту і визначає, що розробляти і у якій послідовності, то бізнес-аналітик виявляє, аналізує і

формалізує вимоги, з яких потім буде створюватись беклог. Тому ключовими функціями бізнес-аналітика є збір і аналіз вимог, проведення воркшопів вимог, моделювання бізнес-процесів, опис функціональних та нефункціональних вимог та узгодження вимог між сторонами розроблення програмного забезпечення.

Виходячи з вище сказаного, менеджер проекту відповідає за продукт, який і чому необхідно створити, власник продукту говорить яку саме функцію цього продукту необхідно реалізувати першою у спринті, а бізнес-аналітик надає детально описані вимоги, щоб команда її правильно реалізувала.

Якщо говорити в контексті ІТ-компаній України, то досить часто в невеликих командах ці ролі може виконувати одна людина, але у великих продуктах такі ролі чітко розділені.

Надалі будемо розглядати бізнес-аналітика як особу, яка поєднує всі функціональні обов'язки, пов'язані з процесом інженерії вимог, тобто є зв'язною ланкою між командою Замовника та командою Виконавця.



1.2. Проблеми встановлення вимог до програмного забезпечення

Інтенсивне освоєння сучасних інформаційних технологій у різних сферах застосування призвело до того, що розроблення актуального та ефективного програмного забезпечення є основною причиною появи нових програмних продуктів. Ця тенденція в інженерії програмного забезпечення зумовлена трьома ключовими моментами:

- ✓ можливістю побудови складного програмного забезпечення. Найскладніші системи можуть містити програмне забезпечення, яке дуже часто глибоко інтегроване в компоненти систем вищого рівня;

- ✓ швидким поширенням програмного забезпечення серед його користувачів. Сьогодні будь-яка компанія може задумати новий програмний продукт, замовити його розроблення у вигляді комерційного програмного забезпечення та неймовірно швидко поширити мережею Інтернет;

- ✓ використання готових (так званих «*Off-the-shelf*») програмних модулів. Сьогодні програмне забезпечення може складатись з готових модулів як власного виготовлення, так і придбаних разом з технологіями його розроблення.

Для задоволення таких можливостей у Виконавців робіт із розроблення програмного забезпечення з'являється нагальна потреба не тільки в скороченні циклу розроблення нового програмного продукту, але й у тривалості його виходу на ринок до потенційних споживачів. Однак сама тривалість виходу на ринок не дає достатньої переваги будь-яким ІТ-компаніям серед множини конкурентів. Тому стратегічною метою стає дотримання термінів виходу на ринок саме «правильного» програмного продукту і забезпечення його комерційної якості.

Встановлення вимог до програмного забезпечення - саме те, що дає змогу Виконавцям отримати наочне і узгоджене уявлення Замовників про якісний програмний продукт.

Наявність вимог до програмного забезпечення є основою для реалізації будь-якого програмного проекту. Вони визначають ті потреби зацікавлених сторін (Замовників, користувачів, споживачів, Виконавців, розробників тощо), які є для них необхідними, а також той функціонал, який програмне забезпечення має згодом мати, щоб повністю задовольнити потреби цих сторін.

Наявність вимог до програмного забезпечення є базою для:

- ✓ планування послідовності виконання робіт для реалізації програмного проекту;

- ✓ управління ризиками розроблення програмного забезпечення внаслідок появи потенційних проблем;

- ✓ поточного та приймального тестування програмного забезпечення;

- ✓ узгодження компромісів щодо реалізації програмного забезпечення;

- ✓ управління змінами під час реалізації програмного забезпечення.

Найбільш традиційні проблеми, через які «провалюються» програмні проекти, є зовсім не технічними. Далі по тексту наведено основні причини провалів програмних проектів, отримані на основі опитування. Демонстрація причин відбувається по пріоритетах (зверху до низу) виникнення проблем, а курсивом відмічені причини, які пов'язані безпосередньо з вимогами до програмного забезпечення:

- ✓ *неповнота вимог до програмного забезпечення від Замовників;*

- ✓ *недостатнє залучення користувачів для зв'язування вимог до програмного забезпечення;*

- ✓ недостатній обсяг виділених ресурсів для виконання проектних робіт;

- ✓ *нереалістичні очікування користувачів від експлуатації програмного забезпечення;*

- ✓ недостатня підтримка Виконавців з боку керівництва ІТ-компанії;

- ✓ часта зміна вимог/специфікацій вимог до програмного забезпечення;

- ✓ недостатнє планування послідовності виконання робіт для реалізації програмного проекту;

- ✓ втрата Замовником потреби в програмному забезпеченні;

- ✓ недостатній проектний менеджмент реалізації програмного забезпечення;

- ✓ технологічна неграмотність персоналу розроблення програмного забезпечення;
- ✓ інші чинники.

Перераховані причини можна поділити на три основні категорії:

✓ *вимоги до програмного забезпечення*: поверхнево організований їх збір та недостатньо їх проаналізовано, погано сформульовані вимоги і слабо пов'язані із запитами і потребами зацікавлених сторін, дуже швидко змінюються вимоги або змінюються без потреб, а також нереалістичні очікування користувачів від можливостей майбутнього програмного забезпечення;

✓ *проблеми, пов'язані з недостатнім обсягом виділених ресурсів*: часткова відсутність коштів та кваліфікованого персоналу, недостатня підтримка персоналу з боку керівництва, недостатній проектний менеджмент, і в загальному погане управління вимогами до програмного забезпечення;

✓ *здатність управління проектами*: недолугість такого управління виражається у впливі керівництва на перші дві категорії.

Всі ці причини можна успішно усунути шляхом зміни підходів до управління програмними проектами. Далі наведені чинники, що сприяють успіху реалізації програмних проектів (курсивом виділені найефективніші):

✓ *залучення користувачів до виявлення вимог до програмного забезпечення*;

✓ підтримка керівництва ІТ-компанії Виконавця робіт із розроблення програмного забезпечення;

✓ *чітка і зрозуміла постановка вимог до програмного забезпечення*;

✓ *чітке планування організації виконання проектних робіт*;

✓ *реалістичні очікування користувачів від експлуатації програмного забезпечення*;

✓ часті контрольні точки у процесі реалізації програмного проекту;

- ✓ компетентна команда розробників програмного забезпечення;
- ✓ володіння вимогами до програмного забезпечення та здатність управління ними;
- ✓ зрозумілий проектний менеджмент для зацікавлених сторін;
- ✓ технологічна грамотність персоналу зацікавлених сторін;
- ✓ інші чинники.

1.3. Інженерія вимог до програмного забезпечення

Термін «інженерія вимог до програмного забезпечення» походить від такого поняття, як «специфікація вимог». Водночас, термін «аналіз вимог до програмного забезпечення» тісно пов'язаний з поняттям «системний аналіз предметної області», який проводять після збору вимог, узгодження їх із Замовником та детального документування.

Інженерія вимог - це сукупність процесів, що охоплюють виявлення, аналіз, специфікацію, перевірку та управління вимогами до програмного забезпечення. Мета інженерії вимог полягає в тому, щоб створити узгоджений, повний і реалістичний набір вимог, який забезпечить успішне розроблення програмного продукту.

Хронологія, наведена на рисунку 1.1, відображає основні етапи процесу інженерії вимог до програмного забезпечення: від початкового збору інформації про майбутній продукт і формування вимог до нього до подальшого управління цими вимогами.

Збір вимог. На цьому етапі відбувається взаємодія із Замовником, користувачами та іншими зацікавленими сторонами для виявлення їхніх потреб, очікувань і проблем, які має вирішити майбутній продукт з метою сформулювати початковий набір вимог, що відображає потреби користувачів.

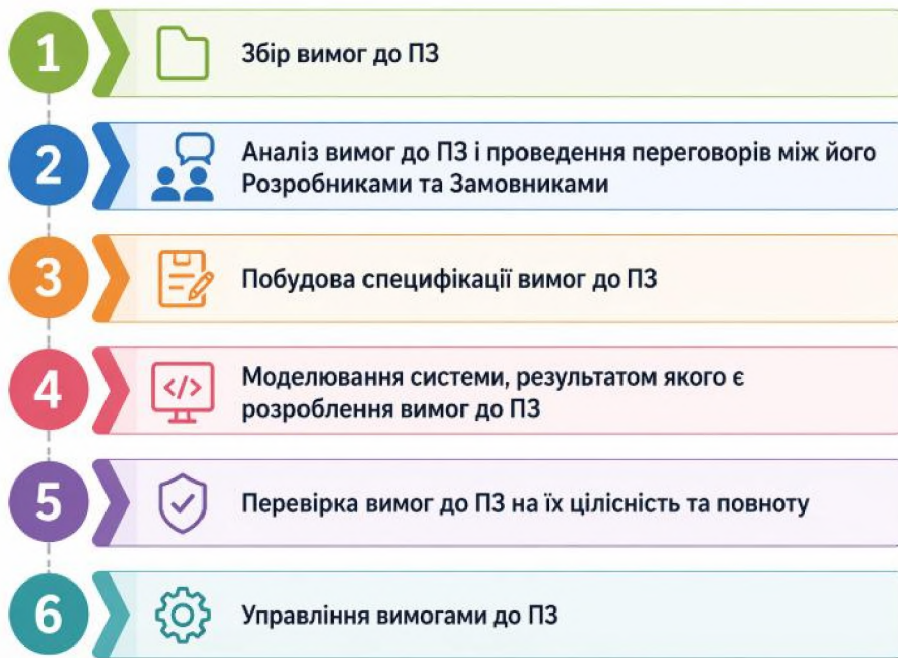


Рис.1.1. Хронологічні етапи інженерії вимог до програмного забезпечення

Аналіз вимог і проведення переговорів. Зібрані вимоги підлягають аналізу, уточненню та узгодженню. На цьому етапі виявляються суперечності, дублювання, неузгодженості чи нереалістичні очікування, шляхом проведення переговорів між Виконавцем та Замовником для досягнення компромісу між бажаним і можливим.

Побудова специфікації вимог. Після узгодження, вимоги документуються у вигляді специфікації вимог до програмного забезпечення (*Software Requirements Specification, SRS*), тобто формується документ, який містить детальний опис функціональних і нефункціональних вимог, сценарії використання, діаграми та інші моделі.

Моделювання системи. Завдання даного етапу полягає у створенні моделей, що відображають структуру, поведінку та взаємодію системи шляхом застосування *UML*-діаграм, *BPMN*-схем, прототипів інтерфейсів тощо. Моделювання допомагає перевірити логіку роботи системи, знайти приховані помилки у вимогах і підготувати основу для подальшого проектування.

Перевірка вимог. Проводиться верифікація та валідація вимог для оцінки їх якості, а саме чи є вимоги повними, несуперечливими, досяжними, зрозумілими та перевірюваними.

Управління вимогами. Під час розроблення програмного забезпечення вимоги можуть змінюватися, тому необхідно забезпечити контроль за їх актуальністю, відстежуваністю та зміною версій. Управління вимогами включає моніторинг змін, оновлення документації, комунікацію з командою та підтримку узгодженості між версіями вимог і реалізованим кодом. Цей процес триває протягом усього життєвого циклу системи.

В деяких моделях життєвого циклу програмного забезпечення хронологія етапів інженерії вимог починається з вивчення його здійсненності, результатом якого є обґрунтований висновок компетентних аналітиків. Якщо висновок припускає, що програмне забезпечення можна реалізувати, то починають аналізувати вимоги до неї, і навпаки.

Важливим завданням при аналізі вимог є встановлення пріоритетності цих самих вимог, оскільки вимоги, висунуті різними зацікавленими сторонами, у реальній системі можуть конфліктувати між собою. Окрім цього, кожна з вимог потребує для свого втілення певних ресурсів, надання яких може залежати також від визначеного для неї пріоритету.

Ще одним важливим завданням аналізу вимог є передбачення здатності програмного забезпечення до можливих змін його компонент та забезпечення можливостей внесення змін без істотного перегляду всієї системи.

1.4. Основні кроки процедури аналізу вимог

Крок 1. *Визначення зацікавлених сторін.* З'ясування переліку зацікавлених сторін, встановлення їхніх потреб та інтенсивності використання

майбутнього програмного продукту часто застосовують для опису можливостей його застосування. При цьому зацікавлені сторони (особи чи організації) реально претендують на придбання програмного продукту, який може безпосередньо чи опосередковано впливати на їхню бізнес-діяльність.

До зацікавлених сторін можуть належати:

- ✓ ті, хто здійснює управління програмним забезпеченням - звичайні та обслуговуючі оператори;

- ✓ ті, хто отримає вигоди від застосування програмного забезпечення - функціональні, фінансові, політичні та соціальні бенефіціари;

- ✓ ті, хто бере участь у фінансуванні процесу розроблення програмного забезпечення або його придбанні;

- ✓ особи та організації, які проти розроблення програмного забезпечення – так звані негативні зацікавлені сторони;

- ✓ особи та організації, що відповідають за розроблення програмного забезпечення, які будуть взаємодіяти з ним.

Крок 2. *Інтерв'ю із зацікавленими сторонами.* Проведення інтерв'ю з представниками від зацікавлених сторін є традиційним підходом, що використовується під час аналізу вимог до програмного забезпечення. Таке уточнення

вимог є одним із шляхів отримання цілісного та достовірного знання.



Крок 3. *Спільні сесії розроблення вимог до програмного забезпечення.* Вимоги до програмного забезпечення часто мають крос-функціональні наслідки, які не завжди відомі окремим представникам від зацікавлених сторін,



тому їх часто пропускають чи неправильно описують під час проведення інтерв'ю з респондентами. Такі негативні наслідки можна виявити під час проведення сесій між зацікавленими сторонами у контрольованому середовищі, роботу якого стимулює кваліфікований посередник - модератор. При цьому зацікавлені сторони беруть участь у дискусії, внаслідок проведення якої вдається виявити деякі крос-функціональні вимоги до програмного забезпечення, проаналізувати їхні деталі та, що найважливіше, розкрити їхні наслідки.

Крок 4. *Набори вимог до програмного забезпечення в стилі контракту.* Традиційним способом документування вимог до програмного забезпечення є формування їх переліку в стилі контракту. У ІТ-сфері не прижилася практика розробки договору під кожного клієнта, тому ІТ-юристи розробляють універсальний темплейт, а деталі кожного окремого проєкту, або його стадій, описують у технічних завданнях, оскільки в складному програмному забезпеченні такий набір вимог може займати сотні сторінок. Головними перевагами документування вимог до програмного забезпечення в стилі контракту є:



- ✓ контракт надає чіткий попунктовий набір вимог до програмного забезпечення, конкретне формулювання та однозначне трактування;

✓ є контрактом між Замовниками (спонсорами) програмного продукту та його Виконавцем.

Але в той же час контракт вимог до програмного забезпечення має ряд недоліків:

- ✗ такі набори вимог містять сотні сторінок, внаслідок чого ці вимоги майже неможливо повністю осмислити й отримати повне розуміння роботи програмного забезпечення;

- ✗ такі набори абстрагують усі вимоги до програмного забезпечення, тому в них є мало контексту;

- ✗ такі набори вимог створюють фальшиве відчуття взаємного порозуміння між Замовниками та Виконавцями;

- ✗ набори вимог у стилі контракту дають зацікавленим сторонам фальшиве відчуття безпеки в тому, що Виконавці змушені будуть досягти тих висот, які вони замовили оскільки через свою природу, в таких наборах вимог часто упускаються критичні вимоги, які проявляються пізніше в процесі розроблення програмного забезпечення. Також Виконавці часто використовують такі відкриті вимоги для того, щоб переглянути умови договору на свою користь;

- ✗ такі набори вимог не зовсім допомагають Виконавцям в ефективному його проектуванні.

Альтернативою до великих, попередньо описаних наборів вимог до програмного забезпечення є гнучке розроблення програмного забезпечення, яке використовує історії користувачів для опису вимог про їхню діяльність у повсякденних термінах.

Крок 5. *Вимірювані цілі*. Найкращі практики (проектанти, архітектори та розробники програмного забезпечення) використовують сформований набір вимог як підказку до етапу проектування архітектури та власне розроблення програмного забезпечення.

При цьому вони можуть запитувати Замовника стосовно предметної області «Чому?» доти, поки не стане зрозумілою справжня бізнес-ціль програмного продукту. Замовники програмного забезпечення та його Виконавці можуть скласти спеціальні тести, які дають їм змогу виміряти рівень досягнення поставлених цілей. Зазвичай такі цілі змінюються повільніше, ніж довгий перелік конкретних, але невимірних вимог. Як тільки невеликий набір критичних, але вимірних цілей буде встановлено, то швидке прототипування та короткі ітеративні етапи розроблення програмного забезпечення часто можуть приносити значно більшу користь для Замовника, інколи навіть задовго до середини реалізації програмного проекту.



Крок 6. *Прототипи - макети програмного забезпечення.* Технологію прототипування програмного забезпечення розглядають як найкраще рішення проблеми аналізу вимог до програмного забезпечення. Прототипи програмного забезпечення, тобто його макети, дають змогу розробникам візуалізувати ще не створений програмний продукт та значно спростити процедуру прийняття архітектурних рішень. Водночас прототипи допомагають майбутнім користувачам уявити, як саме виглядатиме готовий програмний продукт.



Однак протягом останнього десятиліття прототипування, яке себе зарекомендувало як корисна методика аналізу вимог до програмного забезпечення, створило і певні проблеми під час визначення та аналізу вимог, а саме:

✓ як тільки Замовники бачать прототип програмного продукту, то вони перестають розуміти, чому Виконавці не зможуть завершити програмний проєкт протягом деякого часу;

✓ Виконавці часто думають, що змушені будуть використовувати «склеєний» до купи код прототипу в реальному програмному забезпеченні через страх «втратити» час, тобто почати заново;

✓ загалом, прототипи допомагають Виконавцям у прийнятті архітектурних рішень щодо проектування інтерфейсу користувача, однак прототипи не завжди можуть вказати їм, які вимоги були спочатку в Замовника;

✓ Виконавці робіт із розроблення програмного забезпечення і його кінцеві користувачі можуть занадто довго зосереджуватись на побудові відповідних інтерфейсів і занадто мало власне на розробленні продукту, який має реалізовувати певні бізнес-процеси;

✓ прототипи добре придатні для розроблення інтерфейсів користувача, які відображають реальні події на екрані, але не дуже корисні для реалізації пакетних чи асинхронних процесів, які можуть містити складні розрахунки чи цілеспрямовані запити для пошуку даних.

Крок 7. *Прецеденти - технологія документування вимог до програмного забезпечення.* Прецеденти дають змогу задокументувати потенційні вимоги і до нового, і для модифікованого програмного забезпечення. Кожен прецедент надає один чи більше сценаріїв, які відображають те, як програмне забезпечення має взаємодіяти з користувачем чи іншим програмним забезпеченням, щоб досягти конкретної бізнес-цілі. У прецедентах зазвичай уникають технічного жаргону, віддаючи перевагу мові безпосереднього користувача чи досвідченого експерта в предметній області.

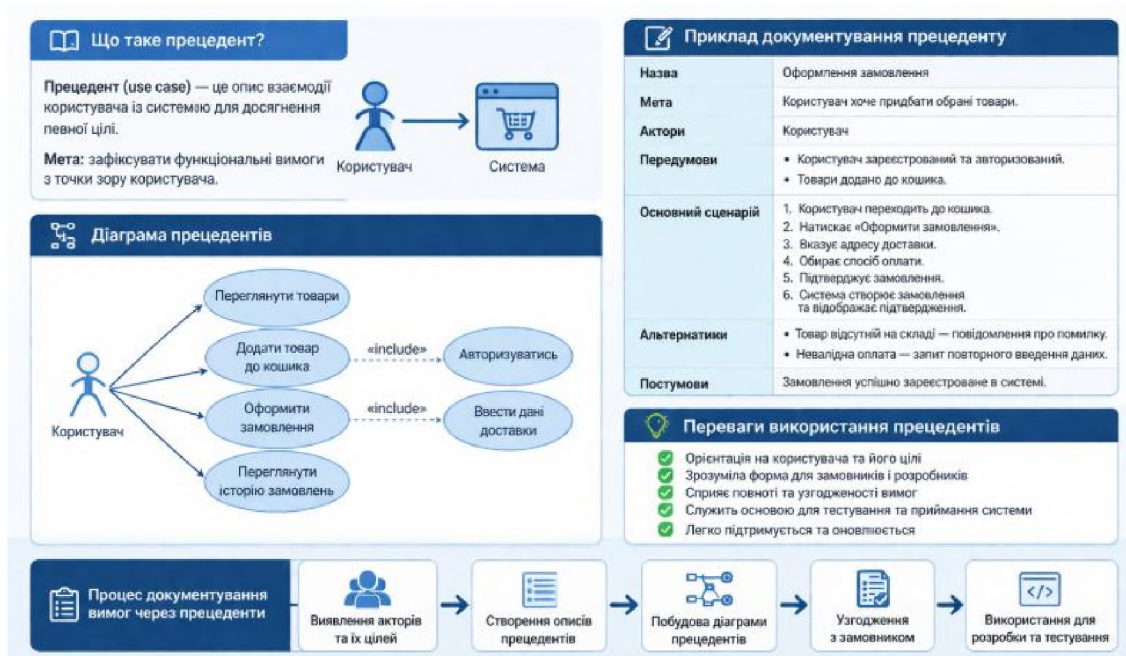


Рис. 1.2. Технологія документування вимог до програмного забезпечення з використанням прецедентів

Для початківців часто прецеденти видаються оманливо простими інструментами для опису поведінки програмного забезпечення. Хоча прецедент і містить текстовий опис усіх способів, якими користувачі можуть працювати з програмним забезпеченням, однак вони не описують жодних внутрішніх процесів у ньому, а також не пояснюють, як саме програмне забезпечення має бути реалізоване. Прецеденти також відомі як функціональні вимоги до програмного забезпечення.

Крок 8. *Специфікація вимог до програмного забезпечення.* Специфікація вимог до програмного забезпечення - це повний опис поведінки програмного забезпечення, яке потрібно розробити; вона містить набір прецедентів, які описують всі взаємодії користувача з програмним забезпеченням.



Окрім прецедентів, специфікація вимог також містить нефункціональні (додаткові) вимоги, які накладають обмеження на програмний продукт чи його реалізацію.

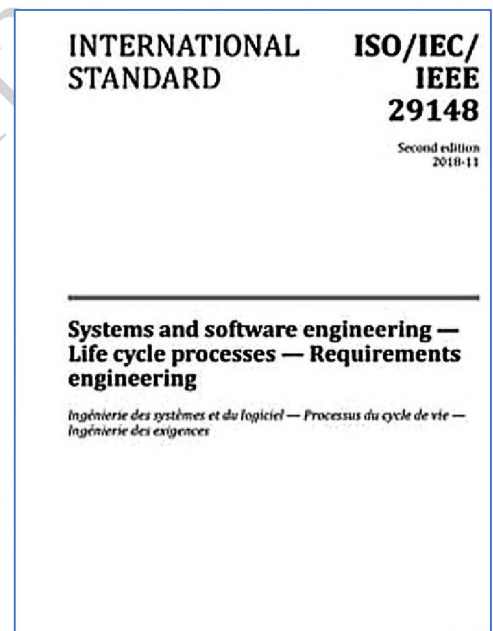
На сьогодні актуальним міжнародним стандартом для написання специфікації вимог є ISO/IEC/IEEE 29148:2018 (*Systems and software engineering - Life cycle processes - Requirements engineering*). Це спільний стандарт ISO, IEC та IEEE, який визначає процеси інженерії вимог і структуру документів специфікації вимог до систем та програмного забезпечення. В Україні прийнято національний аналог: ДСТУ ISO/IEC/IEEE 29148:2025 (ідентичний міжнародному стандарту ISO/IEC/IEEE 29148:2018).

Стандарт ISO/IEC/IEEE 29148 описує повний процес інженерії вимог (*Requirements Engineering*), визначає характеристики якісних вимог, встановлює структуру документів специфікації вимог та регламентує управління вимогами протягом життєвого циклу системи.

Він також об'єднав і фактично замінив старі стандарти, зокрема IEEE 830-1998 (*Software Requirements Specification*), IEEE 1233-1998 (*System Requirements Specification*) та IEEE 1362-1998 (*Concept of Operations*). Хоча стандарт IEEE 830 наразі вважається застарілим, але його структура досить часто все ще використовується як шаблон.

ISO/IEC/IEEE 29148 передбачає два типи специфікацій: специфікація системних вимог (*System Requirements Specification, SyRS*) та специфікація вимог до ПЗ (*Software Requirements Specification, SRS*).

Основні положення стандарту ISO/IEC/IEEE 29148 пов'язані з формуванням цілісної методології роботи з вимогами протягом усього життєвого



циклу системи або програмного забезпечення. Стандарт розглядає інженерію вимог як системний і безперервний процес, який починається на етапі формування концепції системи та продовжується під час її розроблення, впровадження і подальшої експлуатації. Його головна мета полягає у забезпеченні того, щоб вимоги були чітко визначені, узгоджені між усіма зацікавленими сторонами та коректно реалізовані у кінцевому продукті.

Важливим аспектом стандарту є формалізація вимог у вигляді стандартизованої документації. Документи специфікації вимог виконують роль основного джерела інформації для подальших етапів проєктування та розроблення системи. Стандарт визначає загальні принципи формування таких документів, їх структуру та зміст, а також вимоги до якості опису. Особлива увага приділяється забезпеченню однозначності формулювань, повноти опису та можливості перевірки кожної вимоги. Це дозволяє уникнути неоднозначного трактування вимог і забезпечити їх коректну реалізацію у програмному продукті.

1.5. Проблеми аналізу вимог до програмного забезпечення

Аналіз вимог до програмного забезпечення є одним із найвідповідальніших етапів життєвого циклу розроблення програмного забезпечення, оскільки саме від його якості залежить коректність проєктування, реалізації та подальшої експлуатації системи. Однак на практиці цей процес супроводжується низкою проблем, які зумовлені як людськими факторами, так і технічними чи організаційними обмеженнями (див.Рис.1.3).



Рис. 1.3. Джерела проблем аналізу вимог

Проблеми із зацікавленою стороною

Зацікавлені сторони - це особи, групи осіб або організації, які мають інтерес до певного проєкту, продукту, системи чи процесу, можуть впливати на нього або зазнавати впливу від його результатів. У контексті розроблення програмного забезпечення зацікавлені сторони відіграють ключову роль у формуванні, уточненні та погодженні вимог до програмного забезпечення.

До зацікавлених сторін можуть належати різні категорії учасників проєкту. Насамперед це Замовник, який ініціює створення системи та визначає її бізнес-цілі. Важливу роль відіграють кінцеві користувачі, оскільки саме вони безпосередньо працюватимуть із програмним продуктом і можуть оцінити його з точки зору зручності та функціональності. Також до зацікавлених сторін відносять керівників організацій, бізнес-аналітиків, розробників, тестувальників, адміністраторів систем, регуляторні органи, а інколи навіть партнерів або постачальників, якщо їх діяльність пов'язана з функціонуванням системи.

Важливою характеристикою зацікавлених сторін є різноманітність їхніх інтересів. Кожна група може мати власні очікування щодо майбутньої системи,

наприклад, Замовник зазвичай орієнтується на досягнення бізнес-результатів, користувачі - на зручність і ефективність роботи з системою, а розробники - на технічну реалізацію вимог. У зв'язку з цим у процесі аналізу вимог виникає необхідність узгодження різних, іноді суперечливих, очікувань, оскільки саме вони часто стають джерелом основних труднощів під час аналізу вимог:

- ✘ *нечітке розуміння потреб* (користувачі не завжди знають, що саме їм потрібно від програмного продукту, або не можуть сформулювати свої вимоги у зрозумілій формі);

- ✘ *відсутність участі у зборі вимог* (Замовники рідко долучаються до створення письмової специфікації, обмежуючись усними домовленостями);

- ✘ *постійна зміна вимог* (після узгодження графіка робіт і бюджету Замовник вносить нові побажання, що призводить до порушення строків і перевитрат ресурсів);

- ✘ *повільна комунікація* (обмін інформацією між Замовником і Виконавцем відбувається із затримками, що ускладнює процес узгодження);

- ✘ *низька технічна грамотність* (частина користувачів не володіє базовими знаннями про ІТ, що ускладнює розуміння можливостей програмного продукту);

- ✘ *нерозуміння процесу розроблення* (Замовники не завжди уявляють послідовність етапів створення програмного забезпечення, через що очікують швидких результатів або змін без урахування наслідків);

- ✘ *недостатня обізнаність із сучасними технологіями* (це обмежує здатність користувачів адекватно оцінювати можливості реалізації своїх ідей).

Як наслідок - неточність, неповнота або суперечливість вимог, що знижує якість кінцевого продукту.

Проблеми з командою Виконавця

Не менше труднощів виникає й серед команди Виконавця робіт із розроблення програмного забезпечення:

- ✘ *неправильна інтерпретація вимог* (Виконавець може по своєму тлумачити описані вимоги, особливо якщо вони нечіткі або не стандартизовані);

- ✘ *брак комунікації між аналітиками та розробниками* (відсутність зворотного зв'язку призводить до втрати контексту або помилкових рішень);

- ✘ *фокус лише на технічній частині* (ігнорування бізнес-цілей призводить до створення функцій, які не приносять користі користувачам);

- ✘ *недостатня перевірка вимог* (у процесі розроблення програмного забезпечення вимоги часто не проходять належної валідації та тестування на узгодженість);

- ✘ *складність у відстеженні змін* (без систем управління вимогами розробники не завжди швидко можуть адаптуватися до оновлень у документації).

Як наслідок - помилки на етапі проектування, перевитрати часу, створення функціоналу, який не відповідає очікуванням користувачів.

Як видно, проблеми аналізу вимог до програмного забезпечення найчастіше виникають через нестачу комунікації, невизначеність очікувань і брак єдиного бачення продукту між усіма учасниками проекту.

Такі технології, запроваджені ще в 1990-ті роки, як прототипування, *UML* (*Unified Modeling Language*), прецеденти та гнучке розроблення програмного забезпечення є ефективним вирішенням визначених проблем.

Одночасно, на IT-ринку існують інструменти, призначені для симуляції роботи програмного забезпечення та його опису. Такі інструменти поділяються на дві основні категорії: симулятори/засоби тестування (для симуляції роботи) та засоби моделювання/прототипування (для опису програмного забезпечення). Ці інструменти створені як місток через комунікаційний розрив між користувачами

програмного забезпечення та ІТ-компаніями. Вони дають змогу програмним продуктам бути «випробуваними ринком» перед тим, як з'явиться перший програмний код.

Інструменти для симуляції роботи ПЗ

Інструменти для симуляції роботи ПЗ допомагають відтворити, імітувати або перевірити поведінку програмного забезпечення або системи без необхідності розгортання в реальному середовищі:

- ✓ симулятори (*Simulators*) призначені для імітації фізичного середовища, пристроїв або складних систем, з якими взаємодіє програмне забезпечення (наприклад, симулятори роботи мереж, мікроконтролерів, мобільних пристроїв);

- ✓ моки/стаби (*Mocks/Stubs*) використовуються в процесі тестування (особливо юніт-тестування), щоб імітувати поведінку залежних компонентів або сервісів, які поки що не готові або складні для інтеграції;

- ✓ віртуальні машини (*VM*) та контейнери (наприклад, *Docker*) створюють ізольоване середовище, що імітує реальний сервер або операційну систему для розробки та тестування (хоча не є симуляцією в прямому сенсі).

Інструменти для опису ПЗ

Інструменти для опису ПЗ використовують для моделювання, документування та візуалізації архітектури, вимог, процесів та структури програмного забезпечення:

- ✓ *UML (Unified Modeling Language)* стандартизована мова для візуалізації, специфікації, конструювання та документування артефактів програмних систем;

- ✓ *BPMN (Business Process Model and Notation)* застосовується для графічного опису бізнес-процесів;

- ✓ *SysML (Systems Modeling Language)* – це розширення *UML*, що застосовується для системного інжинірингу;

✓ CASE-засоби (*Computer-Aided Software Engineering*) - програмні інструменти, що автоматизують деякі етапи життєвого циклу програмного забезпечення, включаючи моделювання;

✓ інструменти для створення прототипів, які дозволяють створювати інтерактивні моделі (макети) інтерфейсу користувача для візуалізації та узгодження функціональності (*Figma, Sketch, Azure* тощо), візуальної демонстрації, як виглядатиме та працюватиме продукт;

✓ інструменти управління вимогами використовують для опису, відстеження та управління вимогами до програмного забезпечення (*Jira* (частково), *Confluence, DOORS*).

Отже, використання інструментальних засобів для формалізованого опису та симуляції роботи програмної системи є доцільним для підвищення точності формулювання вимог, виявлення суперечностей і уточнення функціональної поведінки програмного продукту на ранніх етапах розроблення. Застосування відповідних інструментів сприяє покращенню комунікації між зацікавленими сторонами та зменшенню ризиків помилок під час проєктування і реалізації програмного забезпечення.

Питання для самоперевірки



1. У чому полягає зміст процедури аналізу вимог до програмного забезпечення?
2. Чому аналіз вимог вважається критичною стадією програмного проєкту?
3. Які характеристики повинні мати якісно сформульовані вимоги?
4. За якими характеристиками здійснюється аналіз вимог (архітектурними, структурними, поведінковими тощо)?
5. У чому полягає різниця між зовнішньою та внутрішньою специфікаціями вимог?

6. Яка послідовність розроблення програмної документації передбачена в тексті?
7. Що включає зовнішня специфікація вимог?
8. Яку інформацію містить внутрішня специфікація вимог?
9. Як у тексті визначаються вимоги користувачів?
10. На які два типи поділяються поведінкові вимоги?
11. У чому полягає відмінність між функціональними та нефункціональними вимогами?
12. Чому функціональні вимоги мають з'ясовуватися на першій стадії розроблення?
13. Яку роль відіграють нефункціональні вимоги в оцінюванні якості програмного продукту?
14. Які властивості програмного забезпечення охоплюють нефункціональні вимоги?
15. Які основні обов'язки *Product Owner* у команді розробки?
16. Чим відрізняється роль *Project Manager* від *Product Owner*?
17. Які ключові функції виконує *Business Analyst* у процесі розробки ПЗ?
18. Хто відповідає за формування та пріоритизацію *backlog*: РО, РМ чи ВА?
19. Як взаємодіють між собою *Product Owner*, *Project Manager* і *Business Analyst* у рамках одного проєкту?
20. Яка роль відповідає за управління ризиками та строками проєкту?
21. Хто з ролей найбільше залучений до збору та аналізу вимог від стейкхолдерів?
22. У чому полягає різниця між бізнес-вимогами та продукт-вимогами, і хто за них відповідає?
23. Яка роль виступає основним комунікатором між замовником і командою розробки?

24. Як розподіляється відповідальність між РО, РМ та ВА у випадку зміни вимог під час проєкту?
25. Які чинники спричинили зростання складності сучасного програмного забезпечення?
26. Яку роль відіграють вимоги у плануванні проєктних робіт?
27. Як вимоги пов'язані з управлінням ризиками?
28. Які причини провалу проєктів безпосередньо пов'язані з вимогами?
29. Чому недостатнє залучення користувачів призводить до проблем?
30. Як часта зміна вимог впливає на реалізацію проєкту?
31. Які категорії проблем виділено в тексті?
32. Які чинники сприяють успіху програмних проєктів?
33. Як визначається поняття «інженерія вимог»?
34. Які етапи входять до процесу інженерії вимог?
35. У чому полягає зміст етапу збору вимог?
36. Яке значення має моделювання системи під час аналізу вимог?
37. Яку роль відіграє пріоритезація вимог?
38. Чому визначення зацікавлених сторін є першим кроком аналізу вимог?
39. Які категорії зацікавлених сторін наведено в тексті?
40. Яке значення мають спільні сесії розроблення вимог?
41. Які переваги та недоліки документування вимог у стилі контракту?
42. Чому вимірювані цілі є важливими для реалізації проєкту?
43. Які переваги та ризики використання прототипів?
44. Які інструменти застосовуються для моделювання та управління вимогами?

ТЕМА 2

ВИМОГИ. ЯКІСТЬ ВИМОГ. ДОКУМЕНТУВАННЯ ВИМОГ

2.1. Поняття про вимоги до програмного забезпечення

Вимоги до програмного забезпечення - це формалізований опис функцій, характеристик і обмежень, які визначають, що саме має робити програмна система та за яких умов вона повинна працювати.

Іншими словами, вимоги - це очікування Замовника та користувачів, виражені в технічній формі, зрозумілій для розробників, тестувальників і всіх учасників проєкту.

Вимоги до програмного забезпечення є основою для:

- ✓ планування процесу розроблення програмного забезпечення;
- ✓ проєктування архітектури та вибору технологій;
- ✓ розроблення й тестування продукту;
- ✓ оцінювання якості та приймання готової системи Замовником.

Якість програмного продукту напряду залежить від якості сформульованих вимог.

Ієрархія вимог до ПЗ

Вимоги до програмного забезпечення поділяють на кілька рівнів ієрархії (див.Рис.2.1), що відображають поступовий перехід від стратегічних бізнес-цілей до конкретних технічних характеристик системи.

Такий поділ допомагає забезпечити узгодженість між потребами Замовника і реалізованим продуктом.

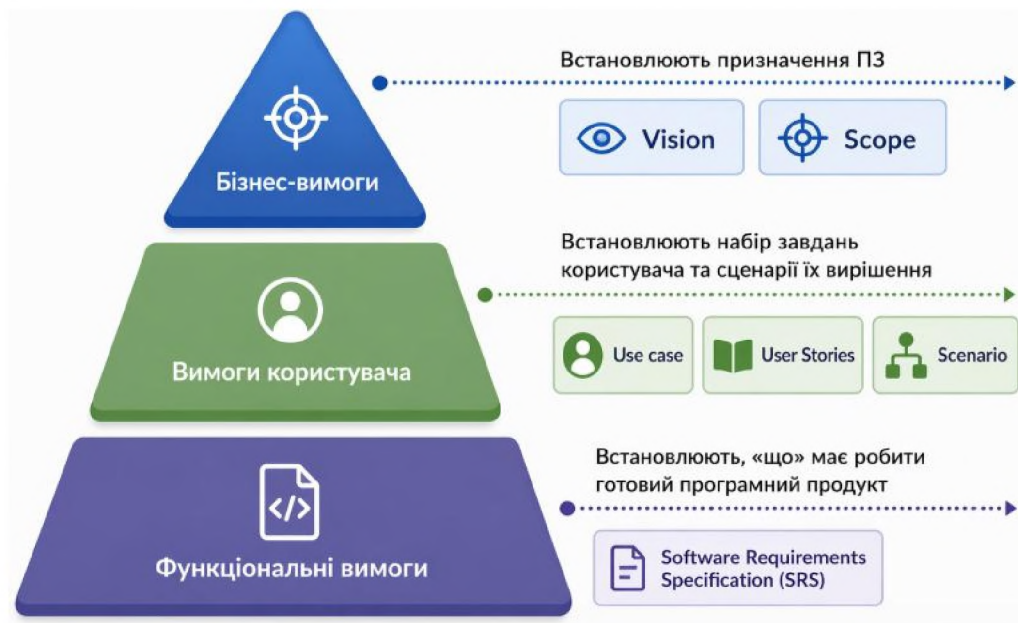


Рис.2.1. Види вимог до ПЗ за ієрархічними рівнями

Бізнес-вимоги (Business Requirements) визначають цілі та очікувані вигоди від системи; детально описуються у документі про бачення проекту та документі про межі проекту.

Вимоги користувача (User Requirements) описують задачі, які користувачі зможуть виконувати за допомогою системи; можуть мати вигляд тверджень, варіантів використання, історій користувача, сценаріїв взаємодії тощо.

Вимоги системного рівня (System Requirements) конкретизують, як система має реалізовувати задачі користувача; детально описуються у специфікації вимог до програмного забезпечення.

Рисунок 2.2. ілюструє взаємозв'язок між різними рівнями та типами вимог до програмного забезпечення, а також документами, у яких вони фіксуються. Схема на рисунку відображає ієрархічний перехід від бізнес-цілей до системних вимог. Вона поділена на кілька рівнів, які показують послідовність формування вимог - від стратегічного рівня до технічного. На рисунку суцільні лінії означають «містяться у», а пунктирні – «впливають на».

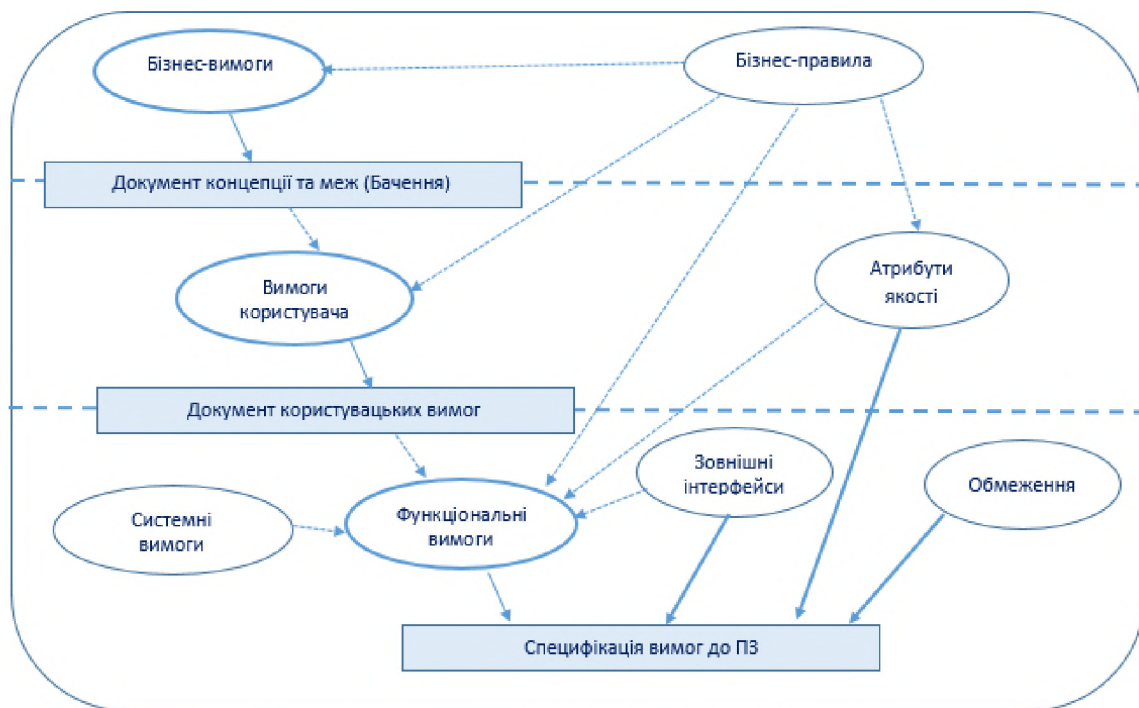


Рис. 2.2. Взаємозв'язки різних типів вимог

Верхній рівень – це бізнес-вимоги, які визначають основні цілі організації або проєкту і дають відповідь на головні питання: «Чому створюється система?» та «Яку проблему вона має вирішувати?»

Бізнес-вимоги пов'язані з бізнес-правилами, які задають загальні принципи діяльності компанії (нормативи, стандарти, політики). На основі цих елементів створюються документи концепції (бачення) та меж проєкту, що узагальнюють призначення продукту.

Середній рівень – це вимоги користувача. З бізнес-вимог впливають вимоги користувача, а саме - опис того, що саме користувач хоче отримати від системи. Вони оформлюються у документі користувацьких вимог. Ці вимоги слугують містком між бізнес-очікуваннями та технічними характеристиками системи.

Нижній рівень – це системні вимоги. На цьому рівні конкретизуються функціональні вимоги (що система робить) і пов'язані з ними:

- ✓ системні вимоги, які описують взаємодію компонентів і середовища;
- ✓ зовнішні інтерфейси, які визначають зв'язок із користувачами або іншими системами;
- ✓ атрибути якості, які характеризують нефункціональні властивості системи (швидкодія, безпека, надійність);
- ✓ обмеження, які задають рамки розроблення (технологічні, організаційні чи часові).

Усі ці елементи формують специфікацію вимог до програмного забезпечення - головний документ для розробників і тестувальників.

Отже, схема показує, як від загального уявлення про продукт Замовника відбувається перехід до формального опису вимог, необхідних для Виконавця. Це логічний шлях перетворення потреб Замовника на чітко визначені вимоги, що становлять основу технічної документації.

2.2. Категоризація вимог до програмного забезпечення

Вимоги до програмного забезпечення можна категоризувати декількома способами. Ми ж розглянемо звичайну категоризацію вимог до програмного забезпечення, яка стосується технічного менеджменту.

Категорії вимог до ПЗ

Вимоги користувача (User Requirements) - це опис того, що саме користувач очікує від програмного забезпечення, які функції, можливості або сервіси йому потрібні для досягнення власних цілей. Вони відображають потреби, задачі та очікування кінцевих користувачів системи, не вдаючись у технічні деталі реалізації. Їх основне завдання - зафіксувати, що саме система має забезпечувати для користувача, щоб задовольнити його потреби. Вимоги користувача формують очікування користувачів у доступній формі, узгоджують функціональність

майбутнього продукту з бізнес-цілями, створюють основу для подальшої розробки технічної специфікації.

Операційні вимоги (Operational Requirements) - це вимоги, які визначають умови функціонування програмного забезпечення в реальному середовищі експлуатації. Вони описують, як саме система повинна працювати під час використання, з урахуванням технічних, організаційних і користувацьких аспектів. Іншими словами, це вимоги, що забезпечують ефективну, безпечну й стабільну роботу програмного забезпечення після його впровадження. Операційні вимоги встановлюють умови використання системи в робочому середовищі, визначають вимоги до технічної інфраструктури, обслуговування, резервного копіювання, допомагають забезпечити безперервність роботи, доступність і надійність ПЗ.

Архітектурні вимоги (Architectural Requirements) - це вимоги, які визначають структуру, компоненти, взаємозв'язки та принципи побудови програмної системи. Вони описують як має бути організоване програмне забезпечення на рівні архітектури, щоб забезпечити виконання всіх функціональних і нефункціональних вимог. Архітектурні вимоги задають рамки для проєктування системи і впливають на прийняття технічних рішень, вибір технологій, платформ і способів інтеграції. Архітектурні вимоги визначають структуру програмного забезпечення (модулі, підсистеми, інтерфейси), забезпечують масштабованість, продуктивність і надійність системи, задають обмеження на використання технологій, мов програмування, баз даних, формують основу для розроблення архітектурного проєкту.

Структурні вимоги (Structural Requirements) - це вимоги, які визначають внутрішню будову програмної системи, її компоненти, модулі, підсистеми та взаємозв'язки між ними. Вони описують архітектурну структуру програмного забезпечення і слугують основою для проєктування, розроблення та подальшої

підтримки системи. Структурні вимоги визначають як система поділяється на логічні та фізичні частини, яким чином ці частини взаємодіють між собою, які модулі, сервіси, інтерфейси та бази даних повинні бути реалізовані, як забезпечується гнучкість, масштабованість і повторне використання компонентів.

Функціональні вимоги (Functional Requirements) - це вимоги, які визначають що саме повинна робити система, тобто описують функції, дії, операції та реакції системи у відповідь на певні події або запити користувачів. Ці вимоги фіксують очікувану поведінку програмного забезпечення, визначають, як система взаємодіє з користувачем, іншими системами та своїм середовищем. Функціональні вимоги формують основу для проектування архітектури і модулів програмного забезпечення, визначають основну функціональність системи, є базою для тестування (перевірки правильності роботи програми), забезпечують зрозуміле узгодження між Замовником і Виконавцем щодо очікуваних результатів роботи системи.

Нефункціональні вимоги (Non-functional Requirements) - це вимоги, які описують якість роботи системи, а не її функції. Вони визначають як саме система виконує свої функції, тобто характеризують ефективність, надійність, безпеку, зручність, продуктивність тощо. Нefункціональні вимоги встановлюють критерії якості програмного продукту, забезпечують стабільність, безпеку і комфортність використання системи, є основою для оцінювання продуктивності, доступності та зручності інтерфейсу, впливають на архітектурні та технологічні рішення під час розроблення.

Вимоги до продуктивності програмного забезпечення (Performance Requirements) - це нефункціональні вимоги, які визначають ефективність роботи системи в реальних умовах експлуатації. Вони описують, наскільки швидко, стабільно та ефективно система виконує свої функції при певному

навантаженні. Вимоги до продуктивності потрібні для забезпечення відповідності очікуванням користувачів щодо швидкодії, оптимізації ресурсів (пам'яті, процесора, мережевих каналів), гарантування стабільної роботи системи при зростанні кількості користувачів чи даних, створення умов для тестування продуктивності.

Вимоги дизайну (Design Requirements) - це вимоги, які визначають зовнішній вигляд, структуру та зручність використання програмного забезпечення, а також принципи побудови інтерфейсу користувача (*UI*) і досвіду взаємодії (*UX*). Вони встановлюють *як саме має виглядати і поводитись система з точки зору користувача*, тобто описують естетичні, ергономічні та поведінкові характеристики інтерфейсу. Вимоги дизайну формують єдиний стиль і логіку інтерфейсу користувача, зручність, інтуїтивність і доступність у взаємодії з системою, узгодженість між екранами, кнопками, кольорами, шрифтами, відповідність стандартам доступності та корпоративного стилю.

У межах інженерії вимог декомпозиція системи передбачає не лише структурний поділ на підсистеми та компоненти, але й логічний розподіл вимог між різними рівнями архітектури. У цьому контексті особливого значення набувають поняття успадкованих та розподілених вимог. Ці категорії застосовуються у системній інженерії, архітектурному проєктуванні складних програмних комплексів, а також у стандартах, що регламентують специфікацію вимог.

Успадковані вимоги - це вимоги, які не були явно сформульовані користувачем або Замовником, але *впливають із зовнішніх умов, стандартів, обмежень або архітектури системи*. Вони успадковуються від вищого рівня: бізнес-вимог, організаційних політик, законодавства, стандартів безпеки чи сумісності. Іншими словами, це вимоги, що автоматично переходять до нового продукту з попередніх рішень або контексту використання.

Розподілені вимоги - це вимоги, які виникають у системах, що складаються з кількох взаємопов'язаних компонентів або модулів, розташованих у різних середовищах (фізичних або логічних) та визначають правила, умови та механізми взаємодії між цими компонентами, щоб система працювала як єдине ціле. Іншими словами, розподілені вимоги *описують, як частини розподіленої системи повинні обмінюватися даними, координувати дії та забезпечувати цілісність інформації*. Розподілені вимоги встановлюють принципи взаємодії компонентів, що функціонують на різних вузлах мережі, вимоги до передачі даних, синхронізації, безпеки та надійності, умови продуктивності й масштабованості в розподіленому середовищі. Вони особливо важливі для вебсервісів, хмарних платформ, корпоративних систем, мікросервісної архітектури.

2.3. Функціональні та нефункціональні вимоги

Розберемося, що таке функціональні вимоги за допомогою простого прикладу: у проєкті створення автомобіля функціональною вимогою системи об'ємного огляду може бути «Задня камера повинна виявляти загрозу», а нефункціональною вимогою тут може бути «Як швидко повинно відображатися попередження для користувача, коли датчики камери виявляють загрозу?».

Функціональні
вимоги ПЗ

Функціональні вимоги говорять про конкретний результат системи, коли користувач виконує над ними завдання. З іншого боку, нефункціональна вимога надає загальну поведінку системи або її компоненти, а не функцію.

Бізнес-аналітик формулює функціональні вимоги на основі цілей і сценаріїв користувачів, моделює вимоги у вигляді діаграм варіантів використання, перевіряє, чи кожна вимога є необхідною і перевірюваною,

узгоджує формулювання із Замовником і командою Виконавця. Основні типи функціональних вимог наведені в таблиці 2.1.

Таблиця 2.1 - Основні типи функціональних вимог

Тип функціональної вимоги	Опис	Приклад
Вимоги до введення даних	Визначають, яку інформацію користувач може вводити у систему.	Поле для логіну й пароля; форма заповнення заявки.
Вимоги до обробки даних	Описують, як система має обробляти введену інформацію.	Обчислення податку, конвертація валюти, фільтрація даних.
Вимоги до виведення даних	Визначають, які результати система повинна повертати користувачу.	Звіт про продажі, квитанція, повідомлення про помилку.
Вимоги до бізнес-логіки	Регламентують правила прийняття рішень у системі.	Знижка застосовується лише для зареєстрованих користувачів.
Вимоги до користувацьких ролей і доступу	Визначають, які дії дозволено різним користувачам.	Адміністратор може видаляти записи, користувач - лише переглядати.
Вимоги до інтеграції	Описують взаємодію системи з іншими системами або сервісами.	Інтеграція з банківським <i>API</i> , платіжним шлюзом або <i>CRM</i> .
Вимоги до повідомлень і сповіщень	Регламентують, які події викликають надсилання повідомлень.	Система надсилає листа після реєстрації користувача.
Вимоги до обробки помилок	Визначають, як система має реагувати на помилки користувача або збої.	Відображення повідомлення «Невірний пароль», логування помилок.
Вимоги до безпеки доступу	Часто належать до функціональних, якщо описують дію (а не властивість).	Авторизація через двофакторну перевірку.
Вимоги до звітності та аналітики	Описують функції формування статистичних або управлінських звітів.	Звіт про активність користувачів за день, тиждень, місяць.

Нефункціональні
вимоги ПЗ

Призначення нефункціональних вимог - це забезпечення зручності, швидкості, безпеки, надійності програмного забезпечення, формування очікуваної якості системи, вплив на архітектурні рішення та технологічний вибір,

визначення критеріїв приймання продукту. Основні типи нефункціональних вимог наведені в таблиці 2.2.

Таблиця 2.2 - Основні типи нефункціональних вимог

Тип нефункціональної вимоги	Опис	Приклад
Продуктивність (<i>Performance</i>)	Визначає швидкодію системи, час відгуку, пропускну здатність.	Система повинна обробляти 100 запитів за секунду; час відповіді ≤ 2 с.
Надійність (<i>Reliability</i>)	Визначає стабільність роботи системи, здатність до відновлення.	Система має працювати без відмов 99,9% часу.
Безпека (<i>Security</i>)	Регламентує захист даних і доступу.	Доступ лише після двофакторної автентифікації.
Зручність використання (<i>Usability</i>)	Визначає, наскільки система проста у використанні.	Новий користувач має опанувати інтерфейс ≤ 15 хвилин.
Масштабованість (<i>Scalability</i>)	Визначає здатність системи збільшувати навантаження без втрати продуктивності.	Підтримка до 10 000 одночасних користувачів.
Супровідність (<i>Maintainability</i>)	Легкість внесення змін, виправлення помилок, оновлення системи.	Код повинен бути документований; середній час виправлення ≤ 4 год.
Портативність (<i>Portability</i>)	Здатність системи працювати на різних платформах і ОС.	ПЗ має запускатися на <i>Windows</i> , <i>Linux</i> і <i>macOS</i> .
Сумісність (<i>Compatibility</i>)	Визначає взаємодію з іншими системами, версіями або протоколами.	Підтримка <i>REST API</i> , <i>XML</i> та <i>JSON</i> форматів.
Доступність (<i>Availability</i>)	Частка часу, протягом якого система функціонує.	Система доступна 24/7, час простою ≤ 5 год/рік.
Тестованість (<i>Testability</i>)	Наскільки легко перевірити функції системи.	Всі функції мають бути перевірені автоматизованими тестами.
Модифікованість (<i>Modifiability</i>)	Здатність системи до змін без порушення роботи.	Додавання нового модуля не повинно впливати на інші.
Відстежуваність (<i>Traceability</i>)	Можливість простежити вимогу від джерела до реалізації.	Кожна вимога має унікальний ідентифікатор.
Юридичні та нормативні (<i>Compliance</i>)	Відповідність законам, стандартам або регламентам.	Система повинна відповідати <i>GDPR</i> або <i>ISO</i> .

На рисунку 2.3 наведена класифікація нефункціональних вимог за сферою їхнього впливу та характером обмежень, які вони накладають на програмне забезпечення.

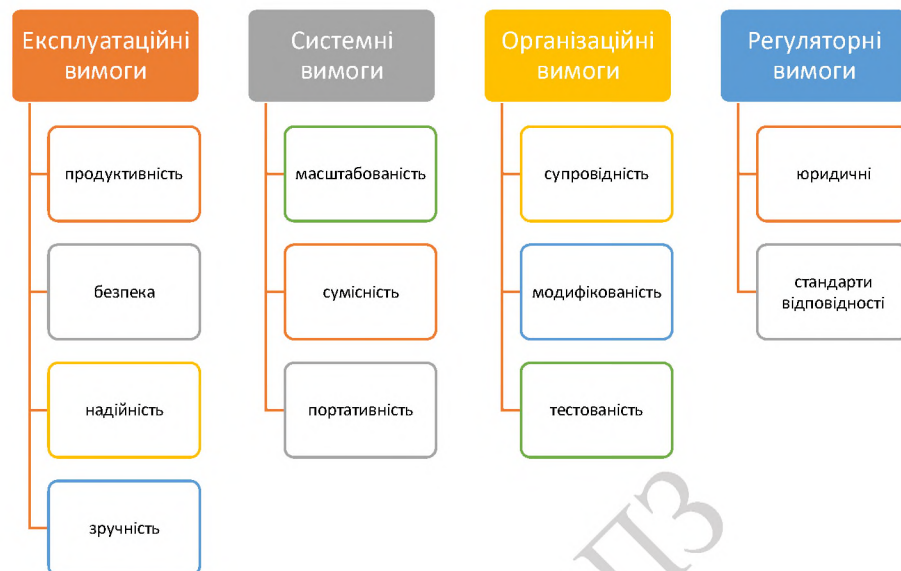


Рис.2.3. Класифікація нефункціональних вимог

Бізнес-аналітик виявляє нефункціональні вимоги під час обговорення очікувань користувачів, формулює вимірювані критерії якості (наприклад, час, відсотки, кількість), перевіряє, чи всі нефункціональні вимоги узгоджені з бізнес-цілями, документує їх у специфікації вимог у розділі «*Quality Attributes*».

Нефункціональні вимоги визначають якість, надійність та ефективність системи. Вони не описують поведінку програмного забезпечення, але впливають на архітектуру, дизайн, інфраструктуру і досвід користувачів.

2.4. Джерела та методи виявлення вимог до програмного забезпечення

Виявлення вимог - це перший етап інженерії вимог, під час якого збирається, уточнюється та документується інформація про потреби користувачів і цілі системи з метою зрозуміти, що саме потрібно Замовнику та

кінцевим користувачам, щоб майбутнє програмне забезпечення дійсно вирішувало їхні проблеми.

Джерела вимог - це особи, документи або системи, з яких бізнес-аналітик отримує інформацію про майбутній продукт. Найважливіші категорії джерел виявлення вимог до програмного забезпечення подані нижче в таблиці 2.3.

Таблиця 2.3 - Категорії джерел виявлення вимог до ПЗ

Категорія	Опис	Приклади
Зацікавлені сторони	Люди або групи, які впливають на розробку системи чи користуються нею.	Замовник, кінцеві користувачі, менеджери, технічні експерти.
Бізнес-процеси	Опис поточних операцій, які система має автоматизувати або вдосконалити.	Схеми <i>BPMN</i> , регламенти, посадові інструкції.
Існуючі системи	Програмні продукти, які потрібно замінити, інтегрувати чи доповнити.	Поточна <i>CRM</i> , бухгалтерська програма.
Документація	Регламентуючі, технічні, нормативні документи.	ТЗ, стандарти <i>ISO</i> , політики безпеки.
Законодавство та нормативи	Вимоги, обов'язкові до виконання за законом.	<i>GDPR</i> , <i>ISO/IEC</i> , державні стандарти.
Аналіз конкурентів	Дослідження рішень, які вже існують на ринку.	Продукти-конкуренти, публічні <i>API</i> , демо-версії.
ІТ-інфраструктура	Середовище, у якому буде працювати система.	Сервери, мережеве обладнання, операційні системи.



Методи виявлення вимог поділяються на традиційні (збір інформації безпосередньо від людей) та аналітичні (на основі дослідження даних і систем). Найпоширеніші методи виявлення вимог до програмного забезпечення:

- ✓ *інтерв'ювання* (індивідуальні або групові бесіди із зацікавленими сторонами, які дозволяють отримати глибоке розуміння потреб користувачів);
- ✓ *анкетування* (використовується, коли потрібно отримати думку великої кількості користувачів, а також може надавати кількісну оцінку очікувань і проблем);
- ✓ *спостереження* (бізнес-аналітик спостерігає за користувачем у процесі роботи і це дозволяє виявити приховані вимоги, про які користувач може не згадати);
- ✓ *семінари та воркшопи* (спільні зустрічі користувачів, розробників і аналітиків, що підходить для швидкого погодження вимог);
- ✓ *аналіз документів* (вивчення наявних документів, інструкцій, договорів, регламентів);
- ✓ *прототипування* (створення простого макету інтерфейсу або системи для уточнення очікувань користувачів);
- ✓ *мозковий штурм* (колективне генерування ідей для пошуку можливих функцій чи рішень, що ефективно на ранніх етапах, коли система ще не визначена чітко);
- ✓ *аналіз сценаріїв використання* (побудова сценаріїв взаємодії користувача з системою, що дозволяє конкретно описати очікувану поведінку системи);
- ✓ *аналіз конкурентних систем* (порівняння з аналогічними продуктами для виявлення корисних функцій або недоліків).

Під час процесу виявлення вимог визначаються релевантні джерела інформації, підбираються методи виявлення залежно від типу користувачів і складності проєкту, аналізуються, узагальнюються та формалізуються вимоги у вигляді документів.

2.5. Життєвий цикл вимог до програмного забезпечення

Життєвий цикл вимог до програмного забезпечення (Requirements Life Cycle, RLC) - це послідовність процесів, через які проходять вимоги від моменту їх виникнення до завершення життєвого циклу програмного продукту. Його мета – це забезпечити якісне управління вимогами, підтримуючи їхню актуальність, відстежуваність та узгодженість протягом усього проєкту.

Виявлення та збір вимог. Це початковий етап, на якому бізнес-аналітик збирає інформацію про потреби користувачів, бізнес-цілі та очікування Замовника з метою визначення, що саме потрібно системі робити, а не як саме це реалізувати. Як результат: початковий перелік вимог і бізнес-цілей, сформульований у вигляді чернетки або звіту про потреби.



Аналіз і погодження вимог. На цьому етапі вимоги уточнюються, структуруються та узгоджуються між усіма зацікавленими сторонами, перевіряється їх повнота, несуперечливість і здійсненність за допомогою класифікації вимог (бізнесові, користувацькі, системні), визначаються пріоритети, виявляться суперечності, шляхом проведення переговорів між Замовником і Виконавцем. Як результат: узгоджений список вимог, готовий до формалізації в документації.

Специфікація вимог. Усі погоджені вимоги оформлюються у вигляді офіційного документа - Специфікації вимог до програмного забезпечення, який стане основою для роботи проєктувальників, розробників і тестувальників.

Перевірка та валідація вимог. На цьому етапі здійснюється оцінка якості вимог: чи вони зрозумілі, перевірювані, узгоджені та відповідають потребам користувачів; перевірка: чи правильно сформульовані вимоги, чи не містять вони помилок або двозначностей та валідація: чи відповідають вимоги реальним очікуванням Замовника. Як результат: підтверджений, перевірений набір вимог, готовий до реалізації.

Управління вимогами. Під час розроблення програмного забезпечення вимоги можуть змінюватися, тому їх необхідно контролювати з метою забезпечення відстежуваності, актуальності та узгодженості вимог протягом усього життєвого циклу продукту.

Супровід вимог. Після реалізації програмного забезпечення вимоги не зникають. Вони можуть оновлюватися, уточнюватися або розширюватися у процесі підтримки системи. Завдання бізнес-аналітика на цьому етапі - перевіряти відповідність реалізованого функціоналу вимогам, збирати нові потреби користувачів після впровадження, оновлювати специфікацію вимог для нових версій продукту.

Життєвий цикл вимог тісно інтегрований із життєвим циклом розробки програмного забезпечення (*Software Development Life Cycle, SDLC*). Життєвий цикл вимог тісно інтегрований із життєвим циклом розробки програмного забезпечення, оскільки вимоги супроводжують продукт на всіх його етапах. Вони виникають під час аналізу, уточнюються та деталізуються на етапі проєктування, реалізуються під час розробки, перевіряються у процесі тестування та постійно оновлюються під час супроводу системи. Таким чином, вимоги не є статичними, вони еволюціонують разом із продуктом, забезпечуючи його відповідність потребам користувачів і бізнесу.



Рис. 2.4. Інтеграція життєвого циклу вимог у життєвий цикл розробки програмного забезпечення

Таким чином, вимоги живуть протягом усього життєвого циклу програмного продукту, забезпечуючи зв'язок між бізнес-цілями та реалізованим рішенням.

2.6. Якість вимог до програмного забезпечення. «Будинок» якості вимог

Якість вимог визначає успіх проєкту, адже саме від точності їх формулювання залежить чи правильно буде реалізовано програмне забезпечення.

Якісна вимога - це вимога, яка є зрозумілою, однозначною, повною, перевірюваною та здійсненою, тобто такою, що може бути реалізована Виконавцем і зрозуміла Замовнику.

Характеристики якості вимог до програмного забезпечення по-різному визначено в різних джерелах. Однак, деякі характеристики якості вимог є загальновизнаними, пояснення їх змісту та приклад застосування на практиці наведено в таблиці 2.4.

Бізнес-аналітик забезпечує якість вимог через перевірку вимог за кожним критерієм, узгодження формулювань між Замовником і Виконавцем, ведення

матриці якості вимог або матриці трасування, документування чітких, вимірюваних та логічно узгоджених вимог.

Таблиця 2.4 - Типові характеристики якості вимог до ПЗ

Характеристика	Сутність	Приклад або пояснення
Зрозумілість (Clarity)	Вимога чітко сформульована, без двозначностей.	✗ «Система повинна швидко працювати» ✓ «Система повинна обробляти запит ≤ 2 сек»
Однозначність (Unambiguity)	Всі зацікавлені сторони однаково розуміють зміст вимоги.	Термін «користувач» має означати конкретну роль, наприклад, «адміністратор системи».
Повнота (Completeness)	Вимога описує всі необхідні умови та сценарії.	Вимога до авторизації має включати вхід, відновлення пароля, вихід.
Несуперечливість (Consistency)	Вимоги не суперечать одна одній і не дублюються.	Одна вимога не може вимагати «тільки адміністратор може змінювати дані», а інша «будь-який користувач може редагувати».
Перевірюваність (Verifiability)	Можна об'єктивно перевірити чи виконана вимога.	Вимога має містити критерій оцінювання (час, відсоток, кількість).
Здійсненність (Feasibility)	Реалізація вимоги можлива в межах ресурсів, технологій і часу.	Неякісна вимога: «Система має перекладати текст миттєво без помилок».
Відстежуваність (Traceability)	Вимогу можна пов'язати з її джерелом (ціллю, користувачем, тестом).	Кожна вимога має свій ідентифікатор.
Актуальність (Relevance)	Вимога є необхідною і відповідає бізнес-цілям проєкту.	У вимогах не повинно бути непотрібних функцій.
Стабільність (Stability)	Вимога не змінюється часто без суттєвої причини.	Стабільні вимоги - це ті, що затверджені і мають низький ризик зміни.
Пріоритетність (Prioritization)	Кожна вимога має визначену важливість для реалізації.	Наприклад: «Критично», «Бажано», «Необов'язково».

Матриця
якості вимог

Матриця якості вимог (Requirements Quality Matrix, RQM) - це інструмент контролю, який використовується для оцінювання якості сформульованих вимог до програмного

забезпечення; допомагає перевірити, наскільки кожна вимога відповідає ключовим характеристикам якості.

Матриця якості дозволяє виявити слабкі або неякісно сформульовані вимоги, забезпечити єдині стандарти оцінювання вимог у проєкті, відстежувати покращення якості після доопрацювання вимог, служити доказом якості вимог під час аудиту або приймання документації.

Кожна характеристика вимоги може оцінюватися за шкалою:

 Так / Ні або  1–5 балів.

Загальна якість вимоги розраховується як середнє значення або частка виконаних критеріїв.

Якщо вимога не відповідає більш ніж двом критеріям - її потрібно переписати або уточнити.

В матриці можуть використовуватися такі критерії:

- ✓ зрозумілість (чи вимога написана просто і без двозначностей);
- ✓ повнота (чи вона містить усі необхідні умови);
- ✓ несуперечливість (чи не конфліктує з іншими вимогами);
- ✓ перевірюваність (чи можна перевірити її реалізацію);
- ✓ здійсненність (чи є вона реальною для виконання);
- ✓ відстежуваність (чи можна пов'язати вимогу з джерелом або тестом).

Бізнес-аналітик формує матрицю якості вимог у процесі аналізу, проводить рев'ю (перевірку) вимог разом із командою, фіксує зауваження, рекомендації та статус кожної вимоги, забезпечує постійне вдосконалення якості вимог протягом життєвого циклу проєкту.

Ідентифікатор вимоги	Формулювання вимоги	Зрозумілість	Повнота	Несуперечливість	Перевірюваність	Здійсненість	Коментар / Дії
REQ-01	Система повинна дозволяти користувачу входити через логін і пароль.	✓	✓	✓	✓	✓	Вимога чітка і перевірювана
REQ-02	Система повинна працювати швидко.	✗	✗	✓	✗	✓	Конкретизувати критерій «швидко»
REQ-03	Система повинна надсилати звіт керівнику кожного понеділка.	✓	✓	✓	✓	✓	Прийнято
REQ-04	Інтерфейс має бути зрозумілим.	✓	✗	✓	✗	✓	Додати показники зручності (наприклад, 3 кліки)

Рис.2.5. Приклад структури матриці якості вимог

Матриця трасування вимог

Матриця трасування вимог (*Requirements Traceability Matrix, RTM*) - це документ або таблиця, яка забезпечує відстежуваність зв'язків між вимогами та іншими артефактами проекту, такими як бізнес-цілі, тестові сценарії, модулі коду, проєктна документація тощо. Її головна мета – це переконатися, що всі вимоги реалізовані, перевірені й не залишилися поза увагою під час розробки програмного забезпечення.

Матриця трасування використовується для контролю, що всі вимоги реалізовані в продукті, виявлення пропущених або зайвих функцій, перевірки повноти тестування, аналізу впливу змін у вимогах, забезпечення прозорості взаємозв'язків між етапами життєвого циклу розробки.

Найчастіше матриця трасування має табличний вигляд, де по рядках і стовпцях відображено взаємозв'язки між елементами.

№	Ідентифікатор вимоги	Опис вимоги	Джерело / Бізнес-ціль	Модуль / Компонент	Тест-кейс	Статус реалізації
1	FR-01	Користувач може увійти в систему.	Бізнес-ціль BG-01	Модуль «Авторизація»	ТС-01, ТС-02	Реалізовано
2	FR-02	Система надсилає звіт керівнику щотижня.	Бізнес-ціль BG-03	Модуль «Звітність»	ТС-03	У процесі
3	NFR-01	Система має відповідати <i>GDPR</i> .	Законодавство	Усі модулі	ТС-04	Перевіряється
4	FR-03	Користувач може відновити пароль через e-mail.	Бізнес-ціль BG-01	Модуль «Авторизація»	ТС-05	Реалізовано

Рис. 2.6. Приклад структури матриці трасування вимог

Кожна вимога має бути простежуваною, тобто можна дізнатись звідки вона походить, як реалізована та як перевірена. Це дозволяє відповісти на ключові запитання:

- ✓ Чи всі бізнес-вимоги покриті системними та функціональними вимогами?
- ✓ Чи всі реалізовані функції мають джерело у вимогах?
- ✓ Чи всі вимоги протестовано?

Трасування вимог - це процес відстеження зв'язків між вимогами та іншими артефактами життєвого циклу програмного забезпечення, такими як бізнес-цілі, користувацькі потреби, проєктні рішення, тести, реалізовані модулі системи. Мета трасування полягає в забезпеченні повної простежуваності вимог, щоб на кожному етапі розроблення можна було підтвердити, що кожна вимога має джерело, реалізацію і тестову перевірку.



Рис. 2.7. Трасування вимог – повна простежуваність вимог

Типи трасування вимог визначають способи встановлення та підтримки зв'язків між різними елементами вимог і артефактами розробки. Вони відображають напрямок і характер цих зв'язків. Розуміння різних типів трасування дозволяє ефективніше контролювати узгодженість вимог і повноту їх реалізації.

Таблиця 2.5 - Типи трасування

Тип трасування	Напрямок зв'язку	Призначення
Пряме (<i>Forward Traceability</i>)	Від вимоги → до реалізації та тесту	Перевірка, що кожна вимога реалізована і протестована
Зворотне (<i>Backward Traceability</i>)	Від тесту чи коду → до вимоги	Перевірка, що не реалізовано нічого зайвого
Двонаправне (<i>Bidirectional Traceability</i>)	В обидва боки	Повний контроль відповідності між вимогами, реалізацією й тестуванням

Бізнес-аналітик створює і підтримує актуальність матриці трасування, забезпечує зв'язок між вимогами, тестами, модулями і бізнес-цілями,

використовує матрицю для контролю впливу змін, співпрацює з *QA*-фахівцями для перевірки покриття вимог тестами.

«Будинок» якості вимог

«Будинок» якості вимог (*House of Quality, HoQ*) - це матриця, що демонструє взаємозв'язок між потребами користувачів (що потрібно) та технічними або системними характеристиками (як це буде реалізовано). Ця модель походить із методології *QFD (Quality Function Deployment)* - розгортання функції якості, яку часто застосовують при розробленні програмного забезпечення.

«Будинок» якості - це графічна техніка, яка ілюструє зв'язок між вимогами користувачів і атрибутами продукту або процесу. Дана схема названа за її представленням у вигляді будинку, який складається з декількох таблиць і матриць, які аналізують набори даних відповідно до мети розгортання функції якості.

Існує багато версій «Будинку якості», але ми розглянемо одну з найпопулярніших версій, зосередивши увагу на основних будівельних блоках.

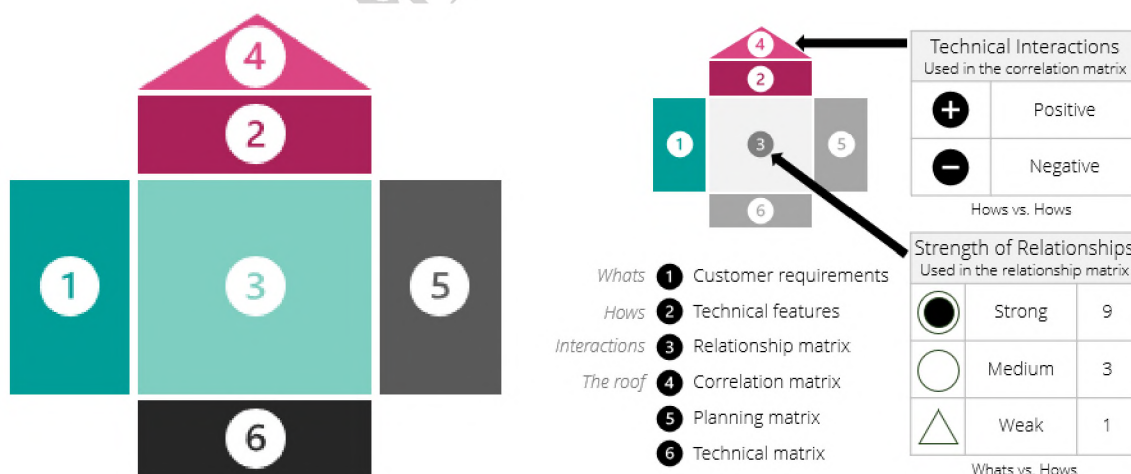


Рис.2.8. Ілюстрація діаграми «Будинку» якості

Вимоги клієнта («the Whats») представляють те, що клієнти потребують в продукті або процесі. Ці потреби можуть бути класифіковані для ясності, і їх відносна важливість може бути оцінена через відгуки клієнтів або дослідження ринку.

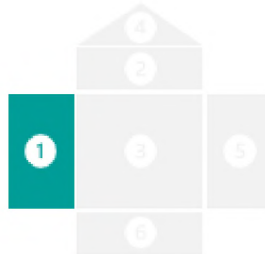


Рис.2.9. Блок 1 – Вимоги клієнта (Customer Requirements)

Технічні характеристики («the Hows») представляють, як компанія збирається перевести потреби користувачів в атрибути продукту та процесу. Вони включають в себе конкретні функції або елементи дизайну, необхідні для задоволення вимог Замовника.

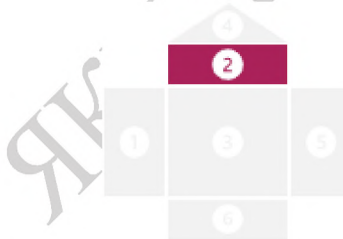


Рис. 2.10. Блок 2 – Технічні характеристики (Technical Features)

Матриця відношень - це центральна частина «Будинку», де оцінюється існування та міцність відношень між кожною вимогою Замовника та технічною особливістю.

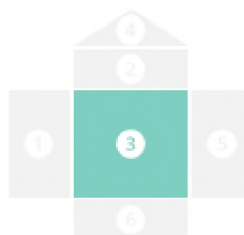


Рис.2.11. Блок 3 - Матриця відношень (Relationship Matrix)

Кореляційна матриця («the Roof») представляє взаємодію між парами технічних характеристик, включаючи як взаємодоповнюючі, так і конфліктні відношення, оскільки зміни в одній парі відношень впливають на зміни в іншій.

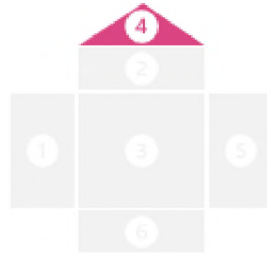


Рис.2.12. Блок 4 - Кореляційна матриця (Correlation Matrix)

Матриця планування використовує рейтинги продуктивності продуктів або процесів та рейтинги важливості клієнтів для планування поліпшень, орієнтованих на потреби клієнтів. Це також дозволяє оцінювати конкуруючі продукти або процеси.

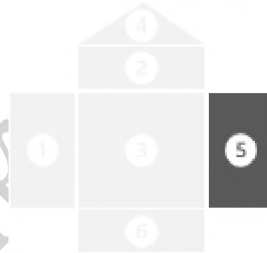


Рис.2.13. Блок 5 - Матриця планування (Planning Matrix)

Технічна матриця - це розрахунок зважених балів за технічні характеристики для визначення їх пріоритетів. Це також дозволяє порівняти технічні характеристики з конкуруючими продуктами або процесами.

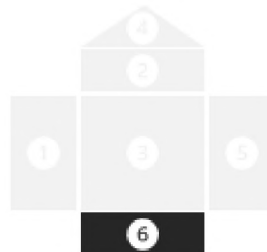


Рис.2.14. Блок 6 - Технічна матриця (Technical Matrix)

Кроки, які необхідно здійснити бізнес-аналітику, щоб побудувати «Будинок» якості наступні:

- ✓ разом з проектною командою, перерахувати вимоги Замовника з їх відносною важливістю у лівій стороні «Будинку»;
- ✓ перерахувати технічні характеристики вздовж верхньої частини «Будинку»;
- ✓ оцінити взаємозв'язок між кожною вимогою клієнта та технічною особливістю в матриці відношень;
- ✓ у кореляційній матриці оцінити потенційний вплив досягнення однієї технічної ознаки на іншу;
- ✓ у матриці планування спланувати покращення на основі оцінок продуктивності продуктів (або процесів) та оцінок важливості клієнтів;
- ✓ у технічній матриці підсумувати кожен стовпець, щоб визначити пріоритетну технічну особливість або характеристики якості.

Customer requirements		Technical features					
	Weight	Adaptive cruise control	0 – 60 time	Size of seats	Liters per 100km	Climate control systems	Cost for a basic service
Comfortable front seating	7			△		△	
Able to fit three adults in the back seat	8			●			
Includes advanced technology features	5	○				○	
Equipped with various safety features	7	●					△
Good acceleration for safe overtaking	4	△	●		△		
Economical to run	9		△		○		●
Technical priority		82	45	79	31	22	88

Рис.2.15. Приклад аналізу вимог клієнта при купівлі автомобіля, створений виробником автомобіля

Зауважимо, що даний приклад є простим, загальним і використовує лише одну матрицю (матрицю відношень) для демонстрації ключових елементів. В реальних умовах цей приклад буде більш всеосяжним і включатиме додаткові вимоги клієнтів та технічні особливості. На практиці, досить часто достатньо

перевести вимоги Замовника в технічні особливості, як зазначено в прикладі. Однак для критично важливих застосунків може знадобитися більш ретельний підхід.

Слід звернути увагу, що напрямок вдосконалення (*Direction of Improvement*) вказує на збільшення або зменшення позитивного впливу на задоволеність клієнтів конкретної технічної характеристики.

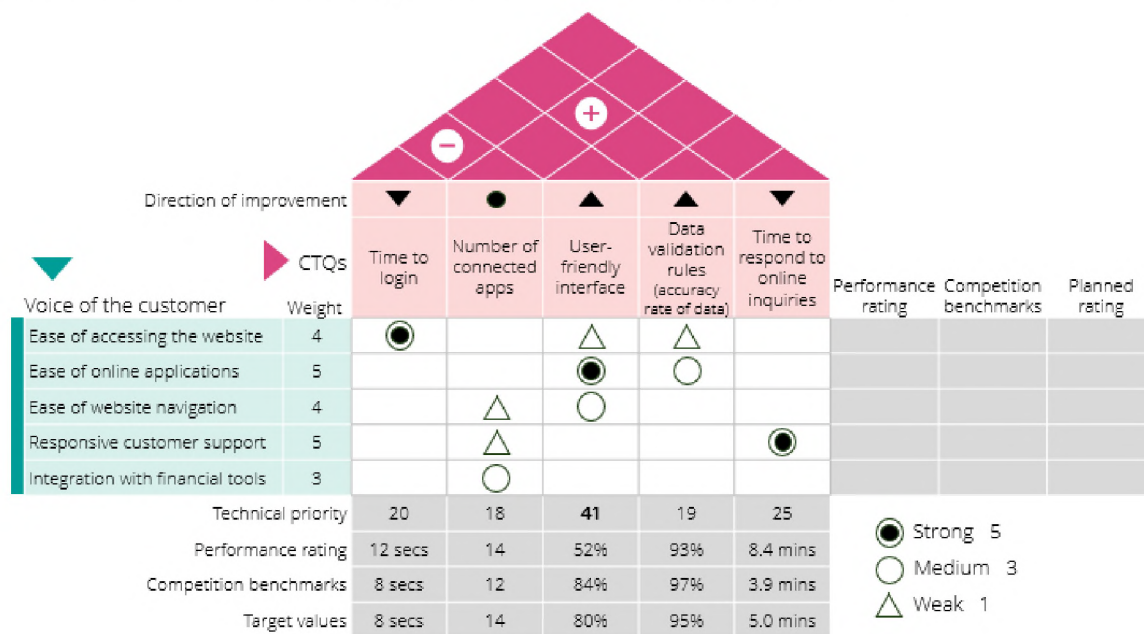


Рис. 2.16. Приклад частково завершеного процесу аналізу вимог для порталу онлайн-банкінгу

Щоб допомогти в розумінні, структуруванні та визначенні пріоритетів вимог клієнтів та технічних особливостей, разом з «Будинком» якості працює цілий ряд інших інструментів, таких як аналіз *Kano*, дерева критичних до якості характеристик та діаграми спорідненості.

Аналіз Kano (Kano analysis) - це метод класифікації вимог користувачів за рівнем їхнього впливу на задоволення клієнтів (детальніше буде розглянуто в темі «Пріоритезація функцій та вимог до програмного забезпечення»).

Дерево якості вимог

«Дерево» критичних до якості характеристик (*Quality Attribute Tree*) - це аналітична модель, яка структурує нефункціональні вимоги до програмного забезпечення, зокрема ті, що визначають його якість. «Дерево» допомагає візуалізувати, уточнювати та пріоритезувати вимоги, пов'язані з якістю системи, наприклад, продуктивність, безпеку, масштабованість, надійність тощо.

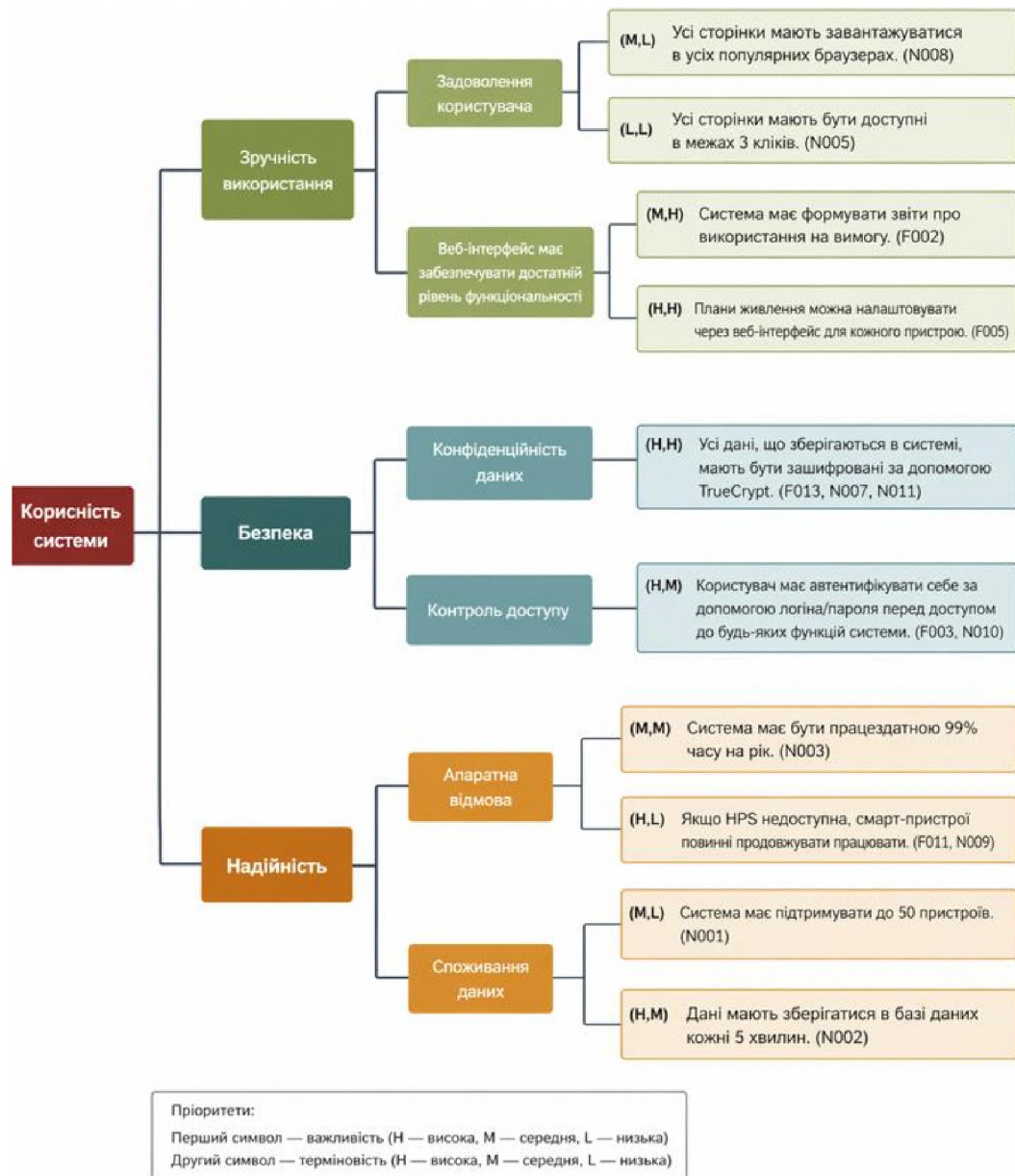


Рис. 2.17. Структурування нефункціональних вимог за допомогою «дерева» критичних до якості характеристик

«Дерево» якості використовується для ідентифікації ключових атрибутів якості системи, розбиття загальних понять («система має бути надійною») на конкретні, вимірювані характеристики, встановлення пріоритетів між різними атрибутами якості, підтримки архітектурних рішень, які впливають на якість програмного забезпечення.

«Дерево» будується ієрархічно - від загальних категорій до конкретних показників.

Таблиця 2.6 - Рівні дерева якості

Рівень дерева	Зміст	Приклад
Корінь	Загальна властивість — Якість системи	Якість програмного забезпечення
Гілки 1-го рівня	Атрибути якості	Продуктивність, Надійність, Безпека
Гілки 2-го рівня	Конкретизація атрибутів	Час відгуку, Захист даних, Відновлення після збою
Листки дерева	Вимірювані показники	«Час відгуку ≤ 2 с», «Шифрування AES-256»

Бізнес-аналітик визначає які саме атрибути якості є критичними для продукту, будує дерево разом з архітекторами й Замовником, формулює кількісні критерії якості для перевірки реалізації, використовує дерево як основу для обговорення компромісів між вимогами (наприклад, швидкість vs безпека).

Діаграма
спорідненості

Діаграма спорідненості (Affinity Diagram) - це інструмент для групування великої кількості ідей, фактів або вимог, які здаються різнорідними, але мають прихований взаємозв'язок. Метод дозволяє структурувати інформацію, отриману під час інтерв'ю, мозкових штурмів або збору вимог, і виявити ключові теми, напрями або проблеми. Метод розроблено Джиро Кавакіто (тому іноді його називають *КЛ-методом*) з метою організувати неструктуровані ідеї, думки або вимоги, виявити

логічні зв'язки між елементами, сформувати узагальнені категорії для подальшого аналізу, сприяти командному розумінню проблеми або продукту.

KJ-метод застосовується на етапі збору та аналізу вимог, після мозкового штурму (щоб упорядкувати ідеї), для класифікації функціональних і нефункціональних вимог, для визначення пріоритетів і проблем користувачів.

Побудова діаграми спорідненості відбувається в декілька етапів:

1. *збір даних*: зібрати всі ідеї, вимоги або твердження (наприклад, на стікерах або картках);
2. *візуальне розташування*: розкласти всі елементи на дошці або екрані;
3. *групування за змістом*: об'єднати схожі за змістом ідеї у групи (без назв на цьому етапі);
4. *найменування груп*: для кожної групи придумати узагальнюючу назву (тему, категорію);
5. *побудова ієрархії*: зв'язати групи в логічну структуру (від загального до конкретного);
6. *обговорення результату*: узгодити, чи всі зв'язки логічні, і за потреби відредагувати.

Діаграми спорідненості є ефективним методом для початкового етапу аналізу вимог. Вони допомагають перетворити хаотичний набір думок користувачів на чітко структурований перелік тематичних груп, що формують основу майбутніх функціональних і нефункціональних вимог.

Нехай маємо зібрані ідеї від користувачів. Після групування отримуємо категорії та приклади ідей і три основні напрямки для формування вимог (див.Рис.2.18).



Рис.2.18. Приклад діаграми спорідненості (у контексті збору вимог до програмного забезпечення)

Бізнес-аналітик фасилітує сесію побудови діаграми, допомагає знайти логічні зв'язки між вимогами, формує узагальнені категорії для подальшого аналізу і документування.

2.7. Документування вимог до програмного забезпечення

Зазвичай вимоги до програмного забезпечення використовують як засіб комунікації між різними зацікавленими сторонами та членами команди його розроблення, а отже вимоги мають бути простими та зрозумілими і для звичайних користувачів, і для самих розробників програмного забезпечення. Для цього створюють такі документи:

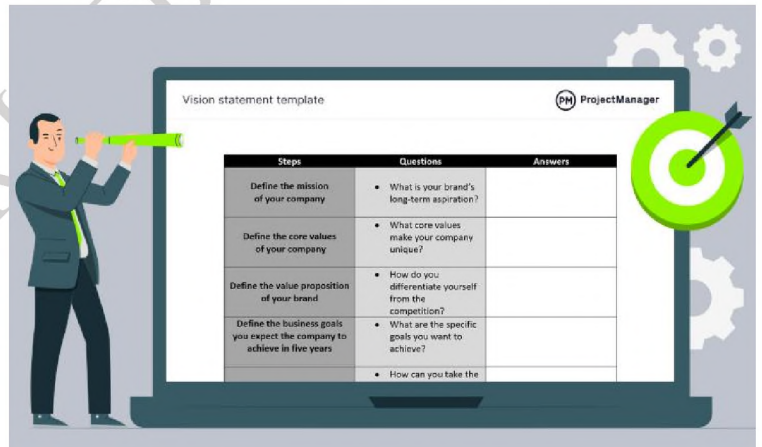


- ✓ документ про бачення програмного продукту (*Vision Document*);
- ✓ документ про межі розроблення програмного продукту (*Scope Document*);
- ✓ специфікація вимог до програмного забезпечення (*Software Requirements Specification, SRS*).

Документ про бачення *Vision Document*

Документ про бачення - це базовий аналітичний документ, який описує загальну концепцію, цілі, межі та ключові характеристики програмного продукту або системи, що планується до розроблення.

Його головна мета - сформулювати єдине бачення майбутнього продукту для всіх зацікавлених сторін: Замовника, користувачів, аналітиків, девелоперів, тестувальників і менеджерів проєкту. Він



узагальнює очікування бізнесу та кінцевих користувачів, визначає стратегічні цілі створення продукту, встановлює рамки проєкту (що буде, а що не буде реалізовано), слугує орієнтиром для подальшого аналізу вимог і розроблення технічної документації. Основні розділи документа про бачення наведені в таблиці 2.7.

Таблиця 2.7 - Основні розділи документа про бачення

Розділ	Зміст
Вступ (<i>Introduction</i>)	Мета документа, короткий опис продукту, основні учасники (stakeholders).
Бізнес-цілі (<i>Business Goals / Business Objectives</i>)	<i>Business Goals</i> - загальні, стратегічні напрями (що бізнес прагне досягти). <i>Business Objectives</i> - конкретизовані, вимірювані показники (як саме буде досягнуто мету).
Опис продукту (<i>Product Description</i>)	Загальна характеристика системи, її основні функції, контекст використання.
Цільова аудиторія (<i>Target Audience</i>)	Опис користувачів, ролей, сценаріїв використання.
Основні можливості системи (<i>Main System Features</i>)	Ключові функції, які відрізняють продукт і приносять користь користувачам.
Межі проєкту (<i>Scope</i>)	Що входить і не входить до складу продукту.
Обмеження (<i>Constraints</i>)	Технічні, організаційні або часові фактори, які можуть впливати на проєкт.
Конкурентні переваги (<i>Competitive Advantages</i>)	Порівняння з аналогічними продуктами, унікальні особливості.
Показники успіху (<i>Success Criteria</i>)	Як буде вимірюватися досягнення цілей продукту.
Майбутні напрями розвитку (<i>Future Development Directions</i>)	Потенційні можливості для вдосконалення системи в наступних версіях.

У вступі визначається контекст створення продукту, формулюється його основна ідея та обґрунтовується потреба у розробленні. Основні завдання вступу:

- ✓ окреслити мету створення системи та ключові проблеми, які вона має вирішити;
- ✓ визначити основних користувачів і зацікавлені сторони;
- ✓ сформулювати стратегічні очікування від продукту;
- ✓ описати зв'язок між цілями бізнесу та майбутнім функціоналом системи.

Типові бізнес-цілі в ІТ-проєктах:

- ✓ підвищення прибутковості бізнесу;

- ✓ оптимізація бізнес-процесів;
- ✓ підвищення якості обслуговування клієнтів;
- ✓ розширення ринкової частки;
- ✓ зменшення операційних витрат;
- ✓ підтримка цифрової трансформації бізнесу;
- ✓ підвищення конкурентоспроможності компанії.

Типові категорії цільової аудиторії в IT-проєктах:

- ✓ користувачі (*Users*) - ті, хто безпосередньо працює з продуктом;
- ✓ менеджери / бізнес-власники (*Managers / Business Owners*) - зацікавлені у бізнес-результатах;
- ✓ адміністратори систем (*System Administrators*) - відповідають за налаштування, безпеку та підтримку;
- ✓ аналітики / експерти (*Analysts / Experts*) - використовують аналітичні чи технічні функції системи;
- ✓ клієнти / споживачі (*Customers*) - кінцева аудиторія, що отримує послугу чи результат роботи системи.

Види меж у проєкті:

- ✓ *функціональні межі* (які можливості та функції буде реалізовано (і не буде)). Приклад: система обробляє замовлення, але не здійснює фінансові транзакції.
- ✓ *організаційні межі* (які підрозділи, користувачі або процеси охоплює система). Приклад: система впроваджується лише у відділі продажів, а не у всій компанії.
- ✓ *технічні межі* (які технології, платформи або інтеграції дозволені чи заборонені).
- ✓ *часові межі* (які етапи входять у поточну фазу проєкту, а які заплановано на наступні релізи).

Обмеження проєкту - це сукупність факторів, які обмежують або визначають рамки реалізації проєкту. Вони задають межі для планування, розробки й оцінки результатів.

Основні типи обмежень:

✓ часові обмеження (*Time Constraints*) - визначають терміни завершення робіт, етапів і здачі продукту. Приклад: система має бути розроблена за 6 місяців.

✓ бюджетні обмеження (*Cost Constraints*) - мають на меті утримання проєкту в межах фінансових ресурсів. Приклад: загальний бюджет не перевищує 100 000 доларів.

✓ ресурсні обмеження (*Resource Constraints*) - обмеження на людські ресурси, обладнання, програмне забезпечення. Приклад: у команді лише два розробники і один тестувальник.

✓ технологічні обмеження (*Technical Constraints*) - вимоги до середовища розробки, технологій, сумісності тощо. Приклад: система має працювати лише на Android.

✓ організаційні обмеження (*Organizational Constraints*) - пов'язані з політикою компанії, процедурами, графіками роботи. Приклад: впровадження дозволено тільки у внутрішній мережі організації.

✓ юридичні та нормативні обмеження (*Legal and Regulatory Constraints*) - відповідність законам, стандартам або індустріальним вимогам. Приклад: система має відповідати вимогам *GDPR* (*General Data Protection Regulation*) - загальний регламент про захист даних Європейського Союзу, який встановлює правила збору, оброблення, зберігання та захисту персональних даних фізичних осіб).

Досить часто виникає питання: яка різниця між межами та обмеженнями? Ці два поняття досить часто плутають, хоча вони мають різну роль у бізнес-аналізі та керуванні вимогами. Дамо чітке пояснення різниці.

Межі визначають, що входить і що не входить у проєкт. Це «лінія», яка відокремлює область відповідальності команди від зовнішніх факторів даючи відповіді на питання:

- ✓ що буде зроблено в межах проєкту?
- ✓ що не буде реалізовано у цій версії системи?
- ✓ який обсяг робіт і функціоналу входить у проєкт?

Приклад меж проєкту

Входить у проєкт:

- ✓ створення веб-застосунку для управління студентськими проєктами;
- ✓ реєстрація та авторизація користувачів (адміністратор, викладач, студент);
- ✓ створення курсів, призначення завдань, формування звітів;
- ✓ адміністрування користувачів і контроль доступу.

Не входить у проєкт:

- ✗ мобільна версія або мобільний застосунок;
- ✗ інтеграція з зовнішніми системами (*Moodle, Google Classroom*);
- ✗ модуль фінансів чи оплати;
- ✗ використання штучного інтелекту або аналітики.

Обмеження проєкту визначають фактори, які впливають на те, як проєкт може бути виконаний. Вони не визначають обсяг, але накладають умови або обмеження на процес реалізації, тобто дають відповіді на головні питання:

- ✓ за яких умов команда повинна працювати?
- ✓ які є ресурсні, часові, технологічні чи фінансові рамки?

Приклад обмежень проекту

🕒

Проект потрібно завершити за 3 місяці.

💰

Бюджет - не більше 50 00 доларів.

💻

Технологія - тільки .NET і MS SOL Server.

Порівняння меж (що робимо) і обмежень (як і в яких умовах) наведено в таблиці 2.8.

Таблиця 2.8 - Порівняння меж і обмежень проекту

Ознака	Межі проєкту	Обмеження проєкту
Визначають	Що входить / не входить у проєкт	Умови, в яких проєкт виконується
Впливають на	Обсяг робіт	Хід виконання
Типові приклади	Функціонал, який буде реалізовано	Бюджет, терміни, технології
Мета	Задати рамки змісту	Задати рамки можливостей
Документуються в	Vision / Scope	Project Charter, SRS

Бізнес-аналітик відповідає за збір і структурування вихідних даних для документа про бачення, формування опису бізнес-проблеми та очікуваних вигід, узгодження бачення продукту з усіма зацікавленими сторонами, забезпечення узгодженості між документом про бачення, бізнес-вимогами та специфікацією вимог.

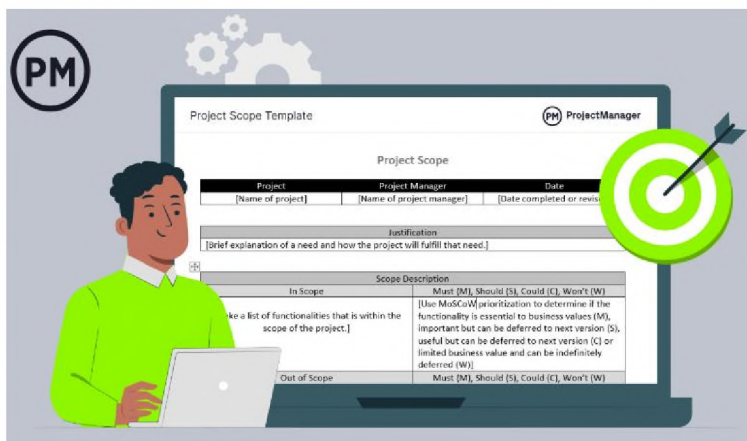
Документ про бачення є першим кроком у процесі інженерії вимог, оскільки визначає напрямок і межі майбутнього проєкту, запобігає розбіжностям у розумінні цілей між командами розроблення ПЗ, служить базою для створення специфікації вимог. Він є стратегічним документом, який фіксує спільне розуміння того, що створюється, для кого і навіщо. Він визначає фундамент для

подальшої розробки вимог і забезпечує узгодженість усіх рішень протягом життєвого циклу проєкту.

Документ про межі *Scope Document*

Документ про межі - це аналітичний документ, у якому описано межі, обсяг і зміст майбутнього проєкту або системи, що розробляється.

Його головна мета - чітко визначити, що входить і не входить до сфери проєкту, щоб усі зацікавлені сторони мали однакове розуміння обсягу роботи, очікуваних результатів і відповідальності.



Документ про межі допомагає встановити чіткі рамки проєкту (*scope boundaries*), узгодити обсяг робіт між Замовником і Виконавцем, запобігти «розповзанню» вимог (*scope creep*), створити основу для планування ресурсів, термінів і вартості проєкту, визначити критерії завершення (що вважається результатом виконання проєкту). Основні розділи документа про межі наведені в таблиці 2.9.

Таблиця 2.9 - Основні складові документа про межі

Розділ	Зміст
Вступ (<i>Introduction</i>)	Мета документа, контекст проєкту, короткий опис системи.
Цілі проєкту (<i>Project Goals / Project Objectives</i>)	<i>Project Goals</i> - загальні, стратегічні цілі (що хочемо досягти). <i>Project Objectives</i> - конкретні, вимірювані результати (як цього досягнемо).
Межі проєкту (<i>Scope</i>)	Чітке визначення, які функції, модулі, процеси входять до сфери проєкту, а які — ні.
Основні результати (<i>Main Outcomes</i>)	Конкретні продукти, документи або компоненти, які мають бути створені.
Обмеження (<i>Constraints</i>)	Чинники, які обмежують реалізацію (бюджет, час, технології, стандарти).

<i>Розділ</i>	<i>Зміст</i>
Припущення (<i>Assumptions</i>)	Умови, які вважаються істинними для планування, але можуть змінитися.
Залежності (<i>Dependencies</i>)	Взаємозв'язки з іншими проектами, системами або процесами.
Критерії успіху (<i>Success Criteria</i>)	Як визначається, що проєкт виконано успішно.
Зацікавлені сторони (<i>Stakeholders</i>)	Перелік ключових осіб або груп, відповідальних за узгодження і затвердження меж.
Погодження і затвердження (<i>Approval and Sign-off</i>)	Підписи або електронне підтвердження схвалення документа всіма сторонами.

Опис вступу, цілей, меж, обмежень та основних результатів документу про межі відповідає їх опису в документі про бачення.

Припущення - це твердження, які вважаються істинними на момент планування або аналізу проєкту, але не мають гарантованого підтвердження. Вони допомагають продовжувати роботу, коли інформація ще неповна, і стають основою для прийняття рішень або оцінки ризиків. Наприклад, припускається, що «Всі користувачі матимуть доступ до інтернету». Це не перевірений факт, але на ньому базується архітектурне рішення.

Формулювання припущень дозволяє уникнути непорозумінь між Замовником і командою Виконавця, своєчасно виявити ризики (якщо припущення виявиться хибним), визначити залежності між компонентами або етапами проєкту, забезпечити прозорість очікувань усіх сторін. Типові приклади припущень при розробленні програмного забезпечення наведені в таблиці 2.10.

Залежності - це зв'язки між завданнями, модулями або етапами проєкту, коли виконання однієї частини роботи залежить від завершення іншої. У контексті вимог до розроблення програмного забезпечення - це умови, зовнішні фактори або компоненти, без яких система не може функціонувати або розвиватись. Типи залежностей наведені в таблиці 2.11.

Таблиця 2.10 - Приклади припущень при розробленні ПЗ

<i>Категорія</i>	<i>Приклад припущення</i>
Технічні	Усі користувачі матимуть стабільне інтернет-з'єднання.
Організаційні	Замовник надаватиме зворотний зв'язок не пізніше ніж за 2 робочі дні.
Бізнесові	Попит на продукт зростатиме протягом першого року після запуску.
Функціональні	Користувачі матимуть базові навички роботи з ПК.
Інфраструктурні	Система розгортатиметься на наявних серверах компанії.

Бізнес-аналітик бере участь у створенні документа про межі аналізуючи бізнес-цілі та визначає, що є «всередині» і «поза межами» проекту, узгоджує межі з усіма зацікавленими сторонами, документує припущення, обмеження та залежності. Документ про межі фактично є контрактом очікувань між Замовником і командою Виконавця, який забезпечує основу для оцінки витрат і термінів, підтримує прозорість і контроль над процесом реалізації.

Таблиця 2.11 - Типи залежностей при розробленні програмного забезпечення

<i>Тип залежності</i>	<i>Опис</i>	<i>Приклад</i>
Внутрішні (<i>Internal Dependencies</i>)	Залежності між завданнями в межах одного проекту.	Тестування починається після завершення розробки.
Зовнішні (<i>External Dependencies</i>)	Пов'язані з іншими командами, організаціями чи системами.	Інтеграція можлива лише після отримання API від партнера.
Функціональні (<i>Functional Dependencies</i>)	Одна функція неможлива без іншої.	Звіти генеруються тільки після збору даних.
Технічні (<i>Technical Dependencies</i>)	Вимагають певних технологій, середовищ або бібліотек.	Програмне забезпечення працює лише з певною версією операційної системи.
Ресурсні (<i>Resource Dependencies</i>)	Потребують доступності ресурсів — людей, обладнання, фінансів.	Розробку можна почати після призначення команди.

Відмінність між документом про бачення і документом про межі наведено в таблиці 2.12.

Таблиця 2.12 - Порівняння Vision і Scope

Ознака	Документ про бачення (Vision)	Документ про межі (Scope)
Призначення	Визначає, що і навіщо створюється	Визначає, що саме буде зроблено, а що - ні
Фокус	На стратегічних цілях і очікуваних вигодах	На деталях реалізації і межах проєкту
Рівень деталізації	Високий рівень узагальнення	Деталізований опис обсягу робіт
Використання	Початковий етап планування	Планування, контроль і управління змінами

Специфікація вимог до ПЗ *Software Requirements Specification*

Специфікація вимог до програмного забезпечення (*Software Requirements Specification, SRS*) - це офіційний документ, який детально описує всі вимоги до майбутньої програмної системи. Його мета - зафіксувати що саме має виконувати система, за яких умов і з якими обмеженнями, щоб уникнути непорозумінь між Замовником, аналітиком, розробником і тестувальником.

Специфікація вимог до програмного забезпечення - це «контракт» між Замовником і Виконавцем, який визначає, що система повинна робити і як має поводитися, але не описує, як її реалізувати технічно.

Специфікація вимог до програмного забезпечення є ключовим результатом етапу аналізу вимог і використовується:

- ✓ як основа для проєктування та реалізації системи;
- ✓ як орієнтир для тестування (перевірка, чи відповідає система описаним вимогам);
- ✓ як договір між Замовником і командою Виконавця;
- ✓ як база для управління змінами вимог у процесі розроблення.



Вступ

Призначення документа

Документ бачення описує концепцію створення вебзастосунку для управління студентськими проєктами. Система має забезпечити зручний облік завдань, комунікацію між викладачами та студентами, а також автоматизацію оцінювання результатів. Основна мета проєкту - підвищити ефективність навчального процесу та спростити взаємодію між усіма учасниками.

Передумови створення

У сучасних навчальних закладах виникає потреба в ефективному управлінні студентськими проєктами. Існуючі інструменти не забезпечують гнучкої взаємодії між викладачами та студентами, що ускладнює контроль за виконанням завдань.

Область застосування

Система призначена для викладачів, студентів і координаторів навчальних програм в університетах.

Цілі проєкту

Надати інструмент для планування, моніторингу та оцінювання студентських проєктів.

Забезпечити прозорість виконання завдань і взаємодії між викладачем та студентом.

Оптимізувати облік результатів і автоматизувати звітність.

Зацікавлені сторони

Замовник. Очікування: отримання зручного інструменту управління навчальним процесом. Дії: формує вимоги, затверджує результати.

Викладач. Очікування: контроль за виконанням студентських проєктів. Дії: використовує систему, формує звіти.

Студент. Очікування: отримання завдань, подання результатів. Дії: активний користувач.

Адміністратор. Очікування: технічна підтримка, контроль доступів. Дії: забезпечує працездатність системи.

Опис продукту

Тип продукту: веб-застосунок.

Основна функціональність: створення курсів, призначення завдань, оцінювання результатів.

Користувацький інтерфейс: панель викладача, панель студента, аналітична звітність.

Очікуваний результат: спрощення управління навчальними проєктами та підвищення прозорості комунікації.

Цільова аудиторія

Викладачі, які керують студентськими проєктами.

Студенти, що виконують курсові чи дипломні роботи.

Академічна адміністрація університету.

Основні можливості системи

1. Реєстрація та авторизація користувачів різних ролей.
2. Створення та редагування навчальних курсів.
3. Призначення завдань студентам.
4. Завантаження виконаних робіт.
5. Автоматичне формування звітів і статистики.

Основні результати

1. Розроблений прототип веб-застосунку.
2. Документація користувача та адміністратора.
3. База даних із тестовими записами.
4. Тестовий звіт із перевіркою функціональності.

Межі проєкту

Входить у проєкт:

веб-застосунок із можливістю створення курсів, завдань і звітів;
система аутентифікації користувачів;
модуль звітності.

Не входить у проєкт:

мобільна версія застосунку;
інтеграція з LMS або зовнішніми базами даних.

Обмеження проєкту

Бюджет - обмежений навчальним фінансуванням.

Термін розробки - 4 місяці.

Використання лише безкоштовних технологій (наприклад, *Python*).

Конкурентні переваги

Простий інтерфейс і швидке впровадження.

Орієнтація на навчальні процеси.

Можливість кастомізації під конкретні потреби кафедри або факультету.

Майбутні напрями розвитку

Створення мобільної версії застосунку.

Інтеграція з LMS (наприклад, *Moodle*, *Google Classroom*).

Розширення аналітичних звітів і дашбордів.

Припущення

Користувачі мають базові навички роботи з веб-інтерфейсами.

Доступ до системи здійснюється через сучасні браузерери.

Кількість одночасних користувачів не перевищує 100.

Залежності

Залежність від наявності стабільного інтернет-з'єднання.

Використання відкритого програмного середовища (наприклад, *Ljango*).

Залежність від серверної інфраструктури університету.

Специфікація вимог до програмного забезпечення створюється бізнес-аналітиком і використовується розробниками - для проєктування та

програмування, тестувальниками - для створення тест-кейсів, Замовниками - для контролю відповідності кінцевого продукту очікуванням, менеджерами - для оцінки обсягу робіт і ресурсів.

Структура специфікації вимог до програмного забезпечення (*SRS*) відповідно до стандарту ISO/IEC/IEEE 29148 визначає рекомендовану організацію документа, який описує всі функціональні та нефункціональні вимоги до системи. Нижче, на рисунку 2.19, подано типову структуру *SRS*, рекомендовану стандартом ISO/IEC/IEEE 29148.

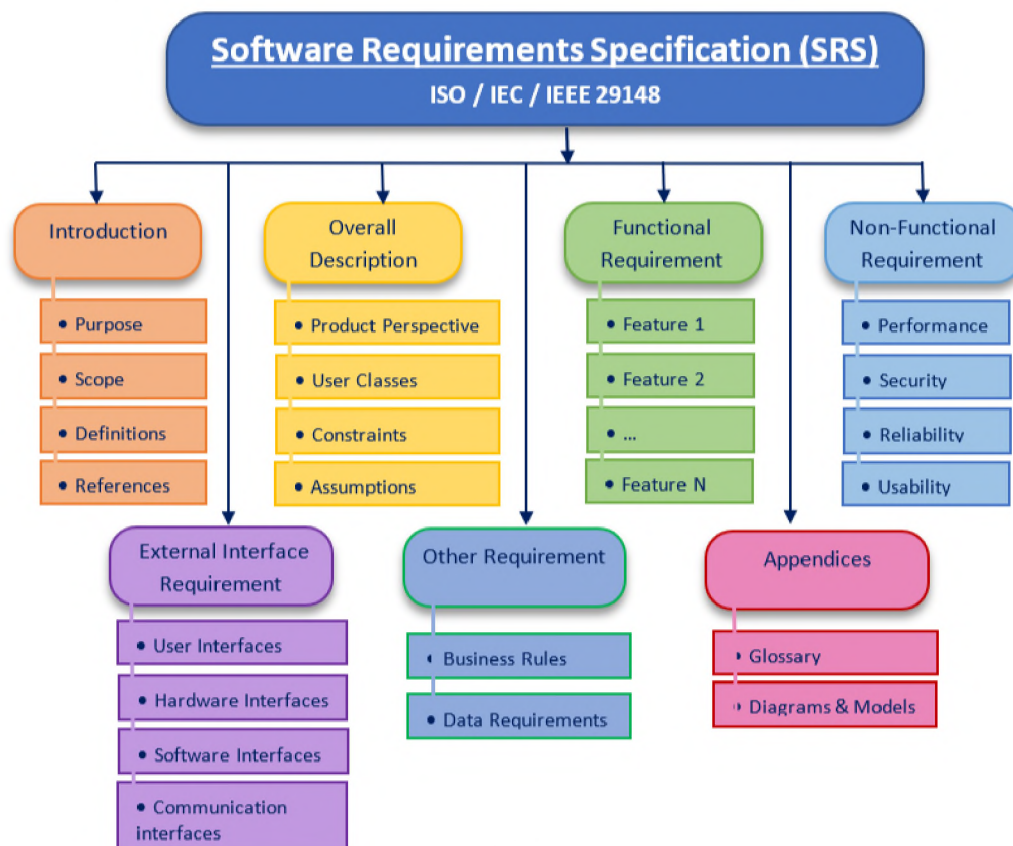


Рис.2.19. Базова структура специфікації вимог до програмного забезпечення, рекомендована стандартом ISO/IEC/IEEE 29148

Вступ (*Introduction*). У цьому розділі подається загальна інформація про документ і систему.

- ✓ Мета (*Purpose*) - опис призначення документа *SRS* та його цільову аудиторію.
- ✓ Область застосування системи (*Scope*) - короткий опис програмного продукту, бізнес-цілей і проблем, які система повинна вирішувати.
- ✓ Визначення (*Definitions*) - перелік термінів, скорочень і їх пояснень.
- ✓ Посилання (*References*) - документи, стандарти, нормативні матеріали та інші джерела, які використовуються при розробленні вимог.

Загальний опис системи (*Overall Description*). У цьому розділі описується контекст системи, її взаємодія із середовищем і основні характеристики.

- ✓ Перспектива продукту (*Product Perspective*) - місце програмної системи в загальній IT-інфраструктурі або бізнес-процесі.
- ✓ Функції продукту (*Product Functions*) - загальний перелік основних функцій системи.
- ✓ Класи користувачів та їх характеристики (*User Characteristics*) - опис різних типів користувачів системи та їхніх ролей.
- ✓ Обмеження (*Constraints*) - технічні, організаційні, правові або стандартні обмеження.
- ✓ Припущення та залежності (*Assumptions and Dependencies*) - фактори, від яких залежить реалізація системи.

Специфічні вимоги (*Specific Requirements*). Це основний розділ документа, який містить детальний опис усіх вимог.

Функціональні вимоги (*Functional Requirements*) - опис функцій, які повинна виконувати система.

- ✓ Вимоги до зовнішніх інтерфейсів (*External Interface Requirements*) - інтерфейс користувача (*UI*), апаратні інтерфейси, програмні інтерфейси, комунікаційні інтерфейси.

✓ Нефункціональні вимоги (*Non-Functional Requirements*) – продуктивність, безпека, надійність, масштабованість, зручність використання, сумісність.

✓ Логічна структура даних (*Logical Database Requirements*) - опис даних, які обробляє система.

✓ Бізнес-правила (*Business Rules*) - правила, які регламентують роботу системи.

Додатки (*Appendices*) містять допоміжну інформацію.

✓ Глосарій - розширений перелік термінів.

✓ Моделі та діаграми - *UML*-діаграми, *Use Case Diagram*, *User Story Map*, *BPMN*-діаграми.

Стандарт ISO/IEC/IEEE 29148 не вимагає жорстко фіксованої структури документа, але пропонує рекомендовану модель *SRS*, яка може адаптуватися під конкретний проєкт, методологію або домен.

Питання для самоперевірки



1. Що таке вимога у контексті розробки програмного забезпечення?
2. Чому вимоги є основою для всіх наступних етапів життєвого циклу програмного забезпечення?
3. Яким чином бізнес-цілі впливають на формування вимог у проєкті?
4. Чому вимоги мають описувати що потрібно, а не як це реалізувати?
5. Які можуть бути наслідки неправильно сформульованих вимог?
6. Чому вимоги повинні бути узгоджені зі стейкхолдерами?
7. Чому важливо формувати вимоги мовою, зрозумілою як бізнесу, так і технічній команді?

8. Чому вимоги розглядаються як контракт між бізнесом і технічною командою?
9. Чому важливо відрізнити вимоги високого та низького рівня?
10. Яка різниця між бізнес-вимогами та вимогами користувача?
11. Що таке системні (system requirements) вимоги?
12. Чим функціональні вимоги відрізняються від нефункціональних?
13. Які типи нефункціональних вимог зазвичай виділяють?
14. До якої категорії відносять вимоги до продуктивності системи?
15. Чому бізнес-вимоги не можуть замінити функціональні?
16. Як вимоги користувача пов'язуються з системними вимогами?
17. Чому вимоги стейкхолдерів можуть входити до окремої категорії?
18. Які вимоги найчастіше змінюються у процесі розробки і чому?
19. Чому важливо чітко розділяти категорії вимог у документації?
20. Які основні джерела інформації для виявлення вимог?
21. У чому переваги інтерв'ю як методу збору вимог?
22. Чому спостереження (observation) може бути надійнішим за опитування?
23. Як аналіз документів допомагає у зборі вимог?
24. Що таке воркшоп зі стейкхолдерами та для чого він використовується?
25. Чому прототипування є ефективним джерелом уточнення вимог?
26. Як аналіз поточних бізнес-процесів допомагає виявити приховані вимоги?
27. Чим корисні опитування для масового збору думок користувачів?
28. Чому інколи важливо проводити аналіз конкурентних продуктів?
29. Чому вимоги мають бути однозначними?
30. Чому важливо, щоб вимога була перевіряльною?
31. Яким чином нечіткі межі вимоги можуть призвести до помилок у розробці?
32. Як суперечливі вимоги впливають на планування та оцінювання проєкту?
33. Чому вимоги мають бути реалістичними та здійсненними?
34. Яким чином якість вимог впливає на якість кінцевого продукту?

35. Чому важливо уникати суб'єктивних формулювань у вимогах?
36. Які техніки перевірки якості вимог використовуються найчастіше?
37. Як якісна вимога підтримує кращий процес тестування?
38. Що таке «будинок якості» у контексті аналізу вимог?
39. Роль матриці «вимоги–характеристики продукту» у «будинку якості»?
40. Чому метод *HoQ* допомагає у пріоритезації вимог?
41. Як метод *HoQ* пов'язує голос користувача з технічними параметрами?
42. Які етапи включає побудова «будинку якості» для програмного забезпечення?
43. Чому важливо документувати вимоги формально, а не лише усно?
44. Які типи документів використовуються в процесі документування вимог?
45. Яку роль відіграють діаграми під час документування вимог?
46. Чому контроль версій є критично важливим у документації вимог?
47. Які переваги надає використання шаблонів під час документування вимог?
48. Що таке *Vision Document* і яку роль він відіграє у процесі розроблення програмного забезпечення?
49. Які основні розділи зазвичай містить документ про бачення системи?
50. У чому полягає різниця між документом про бачення та документом про межі системи?
51. Яке призначення *Scope Document* у процесі аналізу вимог?
52. Яку інформацію щодо зацікавлених сторін та користувачів системи зазвичай фіксують у документі про бачення?
53. Що таке межі системи та чому їх важливо визначити на ранніх етапах розроблення ПЗ?
54. Які типи вимог можуть бути визначені на етапі формування документа про межі системи?
55. Яке призначення *Software Requirements Specification* у життєвому циклі програмного забезпечення?

Тема 3

МОДЕЛЮВАННЯ ВИМОГ: *UML, BPMN*

3.1. Сутність моделювання вимог до програмного забезпечення

Моделювання вимог - це один із ключових етапів процесу інженерії вимог, метою якого є графічне або формалізоване представлення вимог до програмного забезпечення, що дозволяє зрозуміти структуру, поведінку й функціонування системи до її розроблення.

Моделі вимог допомагають зменшити неоднозначність, покращити комунікацію між Замовником, бізнес-аналітиком і командою Виконавця, а також створити базу для подальшого проєктування та тестування системи.

Моделювання вимог - це процес створення абстрактних представлень майбутньої системи, які описують:

- ✓ які дані обробляються;
- ✓ які дії виконуються;
- ✓ як користувач взаємодіє із системою;
- ✓ як система реагує на події;
- ✓ які обмеження впливають на її поведінку.

Модель вимог - це візуальне або структурне подання вимог, яке може бути графічним (діаграма, схема, модель даних) або текстовим (таблиця сценаріїв, описи випадків використання).

Цілі моделювання вимог:

- ✓ уточнення вимог (допомагає виявити суперечності, дублювання або прогалини у вимогах);
- ✓ комунікація (спрощує спільне розуміння вимог усіма зацікавленими сторонами);

- ✓ аналіз системи (дозволяє оцінити взаємозв'язки, потоки інформації та логіку роботи системи);
- ✓ підготовка до проектування (формує основу для розроблення архітектури);
- ✓ підтримка трасування вимог (забезпечує можливість відстеження вимог до конкретних компонентів системи).

Моделювання систем, а відповідно і вимог до неї - складний багатоетапний процес, у якому беруть участь фахівці різного профілю і кваліфікації. Тому одна з головних проблем моделювання - забезпечення спільної мови фахівців, яка дозволить однозначно, чітко і зрозуміло сформулювати основні концепції проекту. Така мова була створена в ході розв'язання задач управління проектами і автоматизації розроблення програмного забезпечення, але її основні засоби можуть бути застосовані для моделювання програмних систем. Тому ця мова отримала назву *UML (Universal Modeling Language)*. Зрозумілість і прозорість проектів, які описані мовою *UML* забезпечується використанням графічних засобів - так званих *UML-діаграм (UML містить понад 14 типів діаграм)*. Для моделювання вимог найчастіше застосовуються:

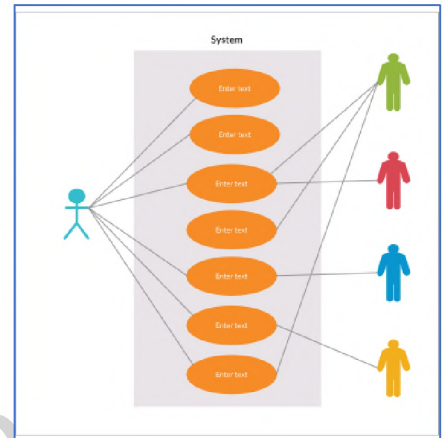
- ✓ *Use Case Diagram* - варіанти використання;
- ✓ *State Diagram* - зміни станів системи;
- ✓ *Activity Diagram* - діаграма діяльності;
- ✓ *Sequence Diagram* - послідовність дій.

UML дозволяє створювати єдину систему моделей, яку можна пов'язати з архітектурою програмного забезпечення, базами даних і тестами.

3.2. Use Case Diagram - варіанти використання

Діаграма варіантів використання (Use Case Diagram) - це вихідна концептуальна модель системи в процесі її проектування і розробки, на якій зображуються відношення між зовнішніми агентами і варіантами використання.

Діаграма варіантів використання дозволяє уявити типи ролей акторів та їх взаємодію із системою, проте не показує порядок виконання кроків. Зображує функціональні вимоги (те, що система може зробити) з точки зору користувача та може описуватись текстом або у вигляді діаграми.



Діаграми варіантів використання складаються з 4 об'єктів: актор, прецедент (варіант використання), система, зв'язок.

Нотація діаграми варіантів використання

Актор. Поняття когось, або чогось, що взаємодіє з системою, але не належить до неї (знаходиться за межами системи). Зазвичай позначається у вигляді стилізованого чоловічка. Рідше у вигляді прямокутника класу з написом «actor».

Усіх акторів умовно можна розділити на *первинних* (ті, що ініціюють взаємодію з системою) та *вторинних* (ті, що реагують на взаємодію). Первинні зазвичай зображуються зліва, а вторинні - справа від системи.

Оскільки будь-яка система може взаємодіяти не лише з людьми, актор не завжди позначає користувача-людину. Він може позначати іншу систему або пристрій з яким взаємодіє.

Часто виникає плутанина між поняттями актор та користувач:

✓ *актор* – це поняття, що представляє клас користувачів (узагальнення групи користувачів), а не конкретного користувача, та може поєднувати в собі декілька ролей (наприклад, актор - працівник компанії може мати ролі інженер, менеджер, директор);

✓ *користувач* – це тип актора або його конкретна реалізація. Декілька користувачів можуть грати одну роль, тобто бути одним актором (наприклад Іван та Роман – студенти). Також один користувач може належати до різних акторів, тобто виконувати різні ролі (наприклад, в системі університету актор може бути викладачем в одному випадку, і науковим керівником в іншому).

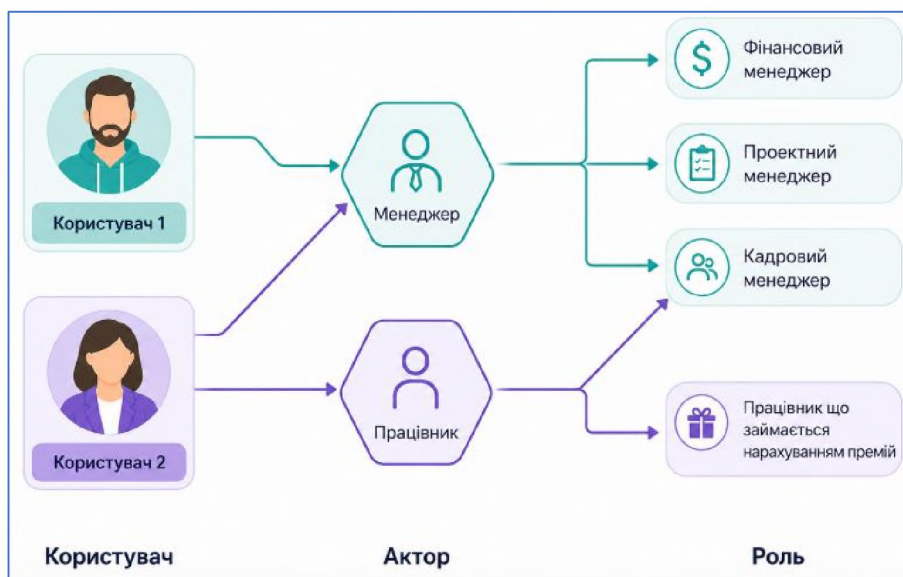


Рис.3.1. Різниця між актором, користувачем і роллю

Варіант використання (прецедент). Прецеденти визначають очікувану поведінку системи, представляють набір можливих функцій, дій або завдань; зазвичай зображується у вигляді еліпса з назвою дії (дієсловом) у ньому. Прецедент вказує, що має трапитись, проте не відповідає на запитання, як це має статись (наприклад, в системі банківського додатка прецедентами можуть бути перевірити баланс, переказати кошти, оплатити рахунок).

Кожен варіант використання відповідає окремому сервісу, який надає

модельована система по запиту зовнішнього агента, тобто визначає один із способів використання системи. Сервіс, який ініціалізувався по запиту зовнішнього агента, має бути закінченою послідовністю дій, а це означає, що після того, як система закінчить обробку запиту зовнішнього агента, вона повинна повернутися у вихідний стан, в якому знову готова до виконання наступних запитів.

Діаграма варіантів використання містить кінцеву множину прецедентів, які в цілому повинні визначати всі можливі сторони очікуваної поведінки системи.

Зовнішній агент - є будь-який зовнішній по відношенню до модельованої системи елемент, який взаємодіє з системою і використовує її функціональні можливості (це може бути людина, технічний пристрій, програма або будь-яка інша система, яка служить джерелом дії на модельовану систему). Варіант використання визначає набір дій, що здійснюється системою при діалозі із зовнішнім агентом.

Система – це те, що моделюється (це може бути сайт, мобільний додаток або навіть модуль програмного забезпечення), зазвичай зображується у вигляді прямокутника з назвою системи у верхній частині.

Зв'язки (відношення). Усі актори системи мають бути пов'язані з прецедентом. Проте не всі прецеденти повинні бути пов'язані з акторами.

У діаграмі варіантів використання існує 4 типи відношень (зв'язків):

1. *Асоціація (Association).* Звичайний зв'язок актора та прецеденту; зазвичай позначається суцільною лінією без напису (стереотипу). Незалежно від типу зв'язку, будь-який актор повинен бути пов'язаний принаймні з одним (можна з декількома) варіантом використання. Кілька акторів можуть бути пов'язані з одним варіантом використання.

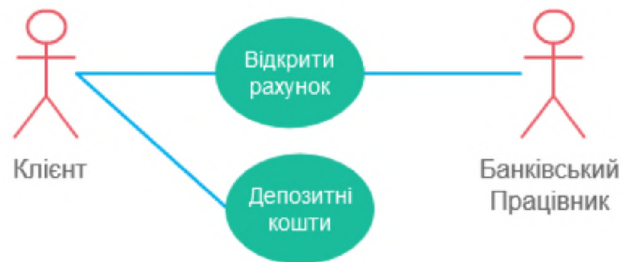


Рис.3.2. Приклад відношення «асоціація»

1. *Розширення (Extend)*. Зв'язок, який показує додаткову функціональність або можливий, не обов'язковий варіант поведінки системи. Базовий прецедент має сенс сам по собі, він не залежить від розширюючого і може існувати без нього. Зазвичай відношення «розширення» позначається пунктирною лінією зі звичайним вказівником та стереотипом (написом) «*extend*», що вказує на базовий прецедент. Розширюючий прецедент активується лише за виконання умови (наприклад, прецедент «хибний пароль» можливий лише при введенні невірного паролю).



Рис.3.3. Приклад відношення «розширення» між прецедентами

При використанні відношення «розширення» базовий варіант може надавати точки розширення (*Extension Points*) для розширюючого прецеденту (за бажанням розробника моделі).



Рис.3.4. Приклад відношення «розширення» з точками розширення

2. *Включення (Include)*. Зв'язок, який показує, що поведінка одного прецеденту включається як складовий компонент у послідовність поведінки іншого прецеденту та ілюструє, що саме використовує базовий варіант для виконання операції. На відміну від відношення «розширення», дочірній варіант у відношенні «включення» має бути обов'язковим для базового. Відношення «включення» використовується для уникнення дублювання однакових прецедентів та додає функціональність, не вказану в базовому. Зазвичай відношення «включення» позначається пунктирною лінією зі стрілкою та стереотипом «include», що вказує на включений варіант.

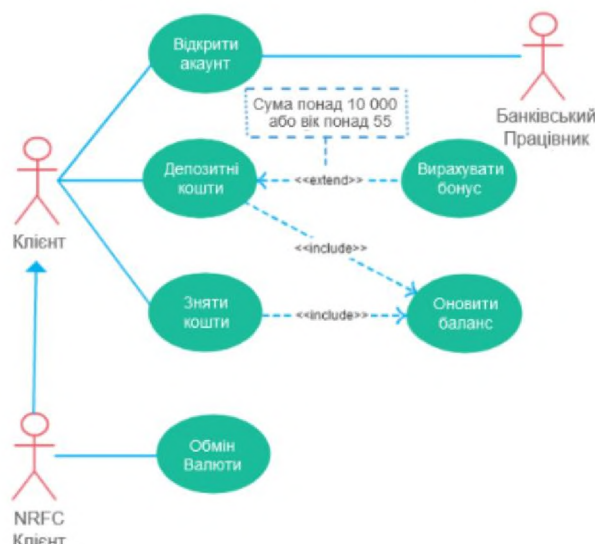


Рис.3.5. Приклад відношення «включення» між прецедентами

4. *Узагальнення (Generalization)*. Відношення «узагальнення» (успадкування, генералізація) це батьківсько-дочірні відношення. «Узагальнення» зазвичай позначається суцільною лінією з трикутним, не зафарбованим вказівником, що вказує на прецедент-предок. Властивості відношення «узагальнення»:

- ✓ дочірні прецеденти мають всі властивості предків;
- ✓ для одного предка може існувати декілька дочірніх прецедентів;
- ✓ може бути кілька батьків (множинне успадкування).

Узагальнення актора. В «узагальненні» актора один актор може успадкувати роль іншого. Нащадок успадковує всі варіанти використання предка, проте може мати і власні унікальні варіанти.

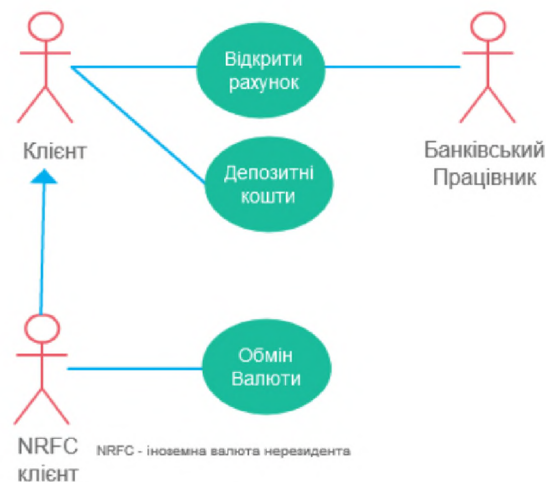


Рис.3.6. Приклад відношення «узагальнення» актора

Узагальнення варіанту використання схоже на узагальнення актора. В цьому випадку поведінка предка успадковується нащадком. Використовується, коли існує спільна поведінка між двома варіантами використання (наприклад, варіанти «Оплата банківською карткою», «Оплата через Google Pay» та «Оплата сертифікатом» можна узагальнити до «Оплати товар»).

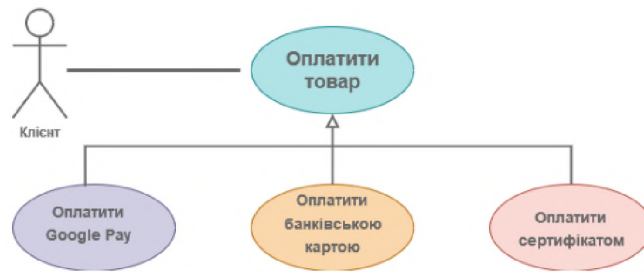


Рис.3.7. Приклад відношення «узагальнення» прецеденту

Випадки застосування Use Case Diagram

Основні випадки, коли варто застосовувати діаграму варіантів використання:

на початкових етапах проєкту

✓ коли потрібно швидко окреслити межі системи, її головні функції та взаємодію із зовнішнім середовищем.

✓ допомагає визначити обсяг роботи і сформувати бачення проєкту.

під час збору та аналізу вимог

✓ бізнес-аналітики використовують її для узгодження очікувань із замовником;

✓ зручно виявляти основні сценарії використання системи, які згодом деталізуються.

для комунікації між учасниками команди

✓ надає спільну візуальну мову між Замовником, розробниками, тестувальниками;

✓ спрощує обговорення, адже зрозуміла навіть для не технічних фахівців.

для опису функціональності системи на високому рівні

✓ показує, які саме послуги або функції система надає користувачам;

- ✓ дозволяє виділити головні ролі користувачів і взаємозв'язки між випадками використання.

для підготовки технічної документації

- ✓ діаграма стає частиною документів, як-от «*Vision Document*» або «*Software Requirements Specification*»;

- ✓ може бути основою для подальшого моделювання (наприклад, створення діаграм послідовності або діяльності).

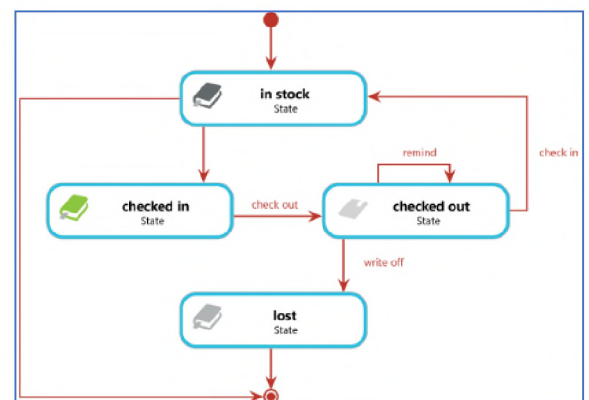
для тестування

- ✓ допомагає створювати тест-кейси на основі сценаріїв використання;
- ✓ забезпечує повноту перевірки всіх основних функцій системи.

3.3. State Diagram - зміни станів системи

Діаграма станів (State Diagram) - це графічне відображення різних станів, у яких може перебувати об'єкт або система, та переходів між цими станами у відповідь на події чи умови. Вона надає узагальнене уявлення про поведінку системи в часі.

Діаграма станів у *UML* - це графічне представлення різних станів об'єкта або системи та переходів між ними, які відбуваються у відповідь на певні події або умови. Діаграми станів використовуються для моделювання поведінки об'єктів чи систем у часовому вимірі. Вони особливо корисні для зображення складних систем, програмних застосунків і систем управління.



Стан системи - це сукупність внутрішніх характеристик (параметрів, змінних, умов), які визначають її поведінку у певний момент часу. Інакше кажучи, стан описує, що саме зараз відбувається із системою, або в якому режимі вона перебуває. На діаграмі стан зображається прямокутником із заокругленими вершинами. Цей прямокутник, у свою чергу, може бути розділений на дві секції горизонтальною лінією. Якщо вказана лише одна секція, то в ній записується лише ім'я стану. Інакше, в першій з них записується ім'я стану, а в другій - список деяких внутрішніх дій або переходів в даному стані.

Ім'я повинне завжди записуватися з великої літери. Оскільки стан системи є частиною процесу її функціонування, рекомендується як ім'я використовувати дієслова в теперішньому часі або відповідні дієприкметники.

Аутентифікація клієнта

entry / отримання пароля
do / перевірка пароля
exit / показати меню опцій

Вхідна дія (entry) - дія, яка виконується у момент переходу в даний стан. Позначається за допомогою ключового слова - мітки дії «entry».

Дія виходу (exit) - дія, яка виконується при виході з даного стану. Позначається за допомогою ключового слова - мітки дії «exit».

Внутрішня діяльність (do activity) - позначається за допомогою ключового слова - мітки діяльності «do», що виконується протягом всього часу, поки об'єкт знаходиться в даному стані, або до тих пір, поки не буде перервано зовнішньою подією. При нормальному завершенні внутрішньої діяльності генерується відповідна подія.

Перехід (transition) - здійснюється при настанні деякої події: закінчення виконання діяльності (*do activity*), отримання об'єктом повідомлення або прийом сигналу. На переході вказується ім'я події, а також дії, які виконуються об'єктом у відповідь на зовнішні події при переході з одного стану в інший.

Перехід може бути направлений в той же стан, з якого він виходить. В цьому випадку його називають переходом в себе. Цей перехід зображається петлею із стрілкою і відрізняється від внутрішнього переходу.

На діаграмі станів перехід зображається суцільною лінією із стрілкою, яка виходить з вихідного стану і направлена в цільовий стан. Кожен перехід може бути помічений рядком тексту.

Початковий стан (start state) - в мові UML позначається у вигляді зафарбованого кружка, з якого може лише виходити стрілка-перехід. Кожна діаграма станів повинна мати єдиний початковий стан.

Кінцевий стан (final state) - в мові UML позначається у вигляді зафарбованого кружка, поміщеного в коло, в яке може лише входити стрілка-перехід. Кожна діаграма станів може мати один або декілька кінцевих станів.

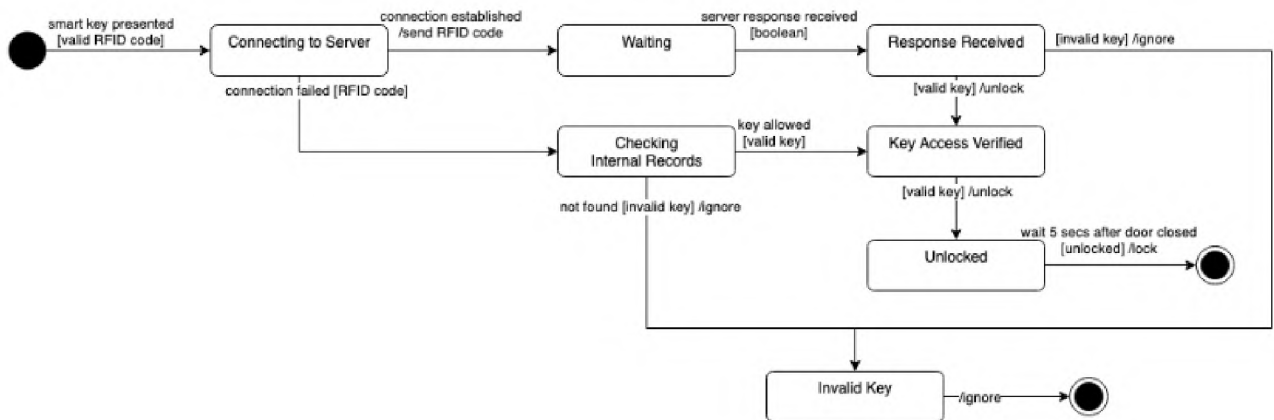


Рис. 3.8. Приклад діаграми станів процесу «Верифікації»

Валідний стан (valid state) - це допустимий або коректний стан системи, у якому всі її внутрішні дані, параметри та зв'язки узгоджені між собою і не порушують встановлених правил функціонування.

Система може перебувати у багатьох станах, але не всі вони є валідними. Валідний стан - це такий, що:

- ✓ відповідає логіці та правилам бізнес-процесу або моделі;
- ✓ не суперечить обмеженням (структурним, логічним, часовим, ресурсним);
- ✓ може бути реально досягнутий у межах нормального функціонування системи.

На діаграмі станів зазвичай відображають лише валідні стани, а переходи між ними мають умови, які гарантують правильність.

Приклад валідного та невалідного станів

Інтернет-магазин:

- ✓ Валідний стан: товар у кошику, є достатня кількість на складі, покупець авторизований.
- ✗ Невалідний стан: оформлення замовлення без вибраного товару або без авторизації.

Банківська система:

- ✓ Валідний стан: баланс рахунку ≥ 0 .
- ✗ Невалідний стан: від'ємний баланс при забороні овердрафту.

Система бронювання:

- ✓ Валідний стан: квиток заброньований і сплачений.
- ✗ Невалідний стан: квиток одночасно має статуси «сплачений» і «скасований».

Складений стан називають також станом-комполитом. Вкладені стани виступають по відношенню до складеного стану як підстани (*substate*). І хоча між ними має місце відношення композиції, графічно всі вершини діаграми, які відповідають вкладеним станам, зображаються усередині символу складеного стану. В цьому випадку розміри графічного символу складеного стану збільшуються щоб вміщати в себе всі підстани.

Складений стан може містити або декілька послідовних підстанів, або декілька паралельних підстанів. Кожний стан-комполит може уточнюватися лише одним з цих способів. При цьому будь-який з підстанів, у свою чергу, може бути станом-комполитом і містити усередині себе інші вкладені підстани.

Кількість рівнів вкладеності станів в мові UML не фіксована.

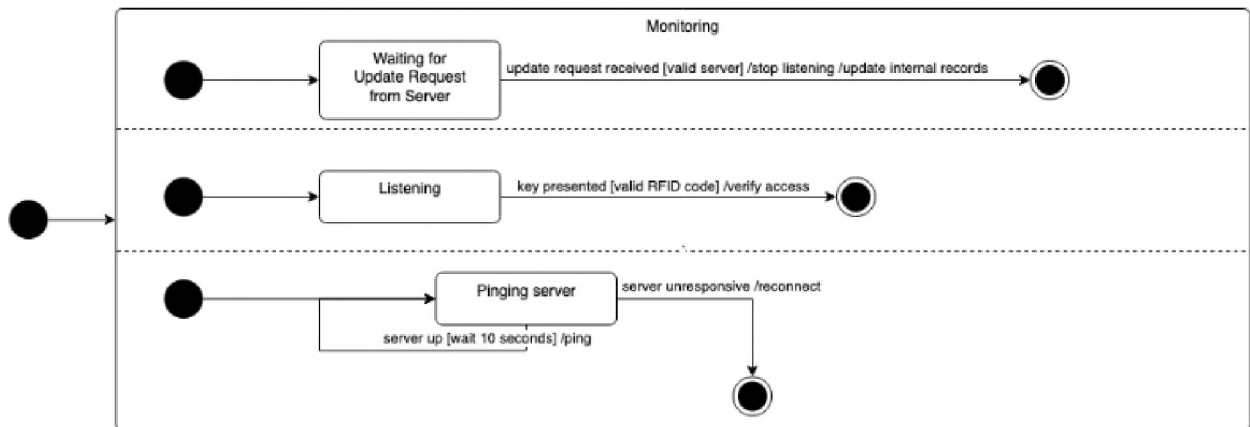


Рис.3.9. Ілюстрація зображення вкладеного стану

Випадки застосування State Diagram

Наведемо декілька причин, коли й навіщо доцільно використовувати діаграму станів саме під час аналізу вимог:

- ✓ для виявлення динамічної поведінки системи (під час аналізу вимог бізнес-аналітик повинен зрозуміти, як система поводить себе у відповідь на дії користувача чи зовнішні події). І тут діаграма станів допомагає описати поведінкові сценарії, які неможливо передати лише словами, виявити умови, що впливають на переходи між станами, уточнити вимоги, які стосуються реакції системи на події (натискання кнопки, отримання повідомлення, помилка тощо). Така діаграма дає змогу бізнес-аналітику чітко описати, що саме має відбуватися при кожній події - наприклад, при поверненні платежу або закінченні часу очікування.

- ✓ для виявлення пропущених або суперечливих вимог. Коли створюється діаграма станів, часто з'ясовується, що деякі переходи неможливі або нелогічні, відсутні описи поведінки у певному стані, не всі події мають визначену реакцію системи. Таким чином, діаграма допомагає уточнити та виправити прогалини у вимогах ще до етапу проєктування.

- ✓ для комунікації між аналітиками, Замовником і розробниками.

Діаграма станів наочно дозволяє легко пояснити, як поводить себе система в часі. Вона зрозуміла навіть для не технічних стейкхолдерів, тому її часто включають до документа «*Software Requirements Specification*» або «*Vision Document*».

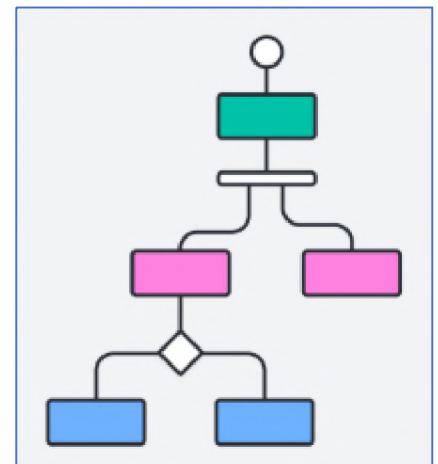
- ✓ для аналізу сценаріїв використання. Кожен *use case* може мати власну діаграму станів, якщо він описує об'єкт або процес із чіткими змінами станів. Це допомагає деталізувати поведінку системи у кожному кроці сценарію, узгодити переходи між станами в різних *use cases*, перевірити повноту сценаріїв.

- ✓ для підготовки до тестування вимог. На основі діаграми станів можна створити тестові сценарії (*test cases*), які перевіряють усі можливі переходи між станами, коректність реакцій системи на події, обробку виключних ситуацій. Це дозволяє здійснити валідацію вимог ще до етапу реалізації.

3.4. Activity Diagram - діаграма діяльності

Діаграма діяльності (*Activity Diagram*) візуалізує процес використання та ілюструє потік повідомлень від однієї дії до іншої. Показує цілісну роботу системи та вважається розширеним варіантом блок-схем. Проте блок-схема не має стандартизованої нотації та не містить паралелізмів (паралельне виконання дій) та синхронізації.

Діаграма діяльності потрібна для опису роботи всієї системи, вона показує перехід від однієї дії до іншої, які можуть виконуватися людьми, програмними компонентами або зовнішніми по відношенню до системи компонентами. Потік керування (порядок виконання) на діаграмі переходить від однієї операції до іншої. Цей потік може бути послідовним, розгалуженим або одночасним.



Нотація діаграми діяльності

Графічна нотація діаграми діяльності багато в чому схожа на нотацію діаграми станів. Кожний стан на діаграмі діяльності відповідає виконанню деякої операції, а перехід в наступний стан відбувається лише після завершення виконання цієї операції.

Стан діяльності (activity state) - стан в графі діяльності, який служить для подання послідовності дій, що вимагають певного часу. Вихід із стану діяльності відбувається після виконання специфікованої в ньому діяльності. Стан діяльності не може мати внутрішніх переходів, оскільки він є елементарним. Графічно стани діяльності зображаються фігурою, що нагадує прямокутник, бічні сторони якого замінені опуклими дугами. Усередині цієї фігури записується ім'я стану діяльності, який має бути унікальним в межах однієї діаграми діяльності. Рекомендується як ім'я простої дії використовувати дієслово із словами пояснень.

Кожна діаграма діяльності повинна мати єдиний початковий і кінцевий стани. Вони мають такі ж позначення, як і на діаграмі станів. Саму діаграму діяльності прийнято розташовувати так, щоб дії слідували зверху вниз або зліва направо. На діаграмі діяльності допускається показувати декілька кінцевих станів.

Потік керування - з'єднує два вузли на діаграмі активності. Вузол розгалуження (вилка) — це вузол керування, який розділяє потік на кілька одночасних. Потрібен для створення декількох одночасних завдань. Вузол злиття об'єднує ці потоки.

Вузол прийняття рішень — це вузол керування, який вибирає між декількома потоками один істинний. Він схожий на оператор *if*. Умови (охоронні умови) записуються в квадратних дужках поруч з потоком.

При побудові діаграми діяльності використовуються лише нетригерні переходи, тобто такі, які відбуваються одразу після завершення діяльності або

виконання відповідної дії.

Якщо із стану дії виходить єдиний перехід, то його можна ніяк не позначати. Якщо ж таких переходів декілька, то при моделюванні послідовної діяльності запускається лише один з них. При цьому для всіх переходів, що виходять з деякого стану діяльності, повинна виконуватися вимога істинності лише одного з них. Така ситуація отримала назву розгалуження, а для її позначення застосовується спеціальний символ *рішення* (*decision*), що має форму ромба, усередині якого *немає* тексту. У цей ромб може входити лише одна стрілка від того стану дії, після виконання якого послідовність дій має бути продовжена по одній з гілок, що взаємно виключають одна одну. Прийнято вхідну стрілку приєднувати до верхньої або лівої вершини символу рішення. Стрілок, що виходять, може бути дві або більше, але для кожної з них явно вказується відповідна сторожова умова у квадратних дужках.

Для графічного об'єднання альтернативних гілок на діаграмі діяльності рекомендується також використовувати аналогічний символ у формі ромба, який в цьому випадку називають *з'єднанням* (*merge*). Вхідних стрілок у символу з'єднання може бути декілька. Виходити з ромба з'єднання може лише одна стрілка, при цьому жодна стрілка не повинна містити сторожових умов.

У діаграмах діяльності також використовується спеціальний символ для розділення і злиття паралельних обчислень або потоків дій. Це пряма риска, аналогічна позначенню паралельних переходів для діаграм станів. На діаграмах діяльності така риска зображається відрізком горизонтальної, рідше - вертикальної, лінії, товщина якої більша за лінії простих переходів. При цьому розділення має один вхідний перехід і декілька вихідних, а

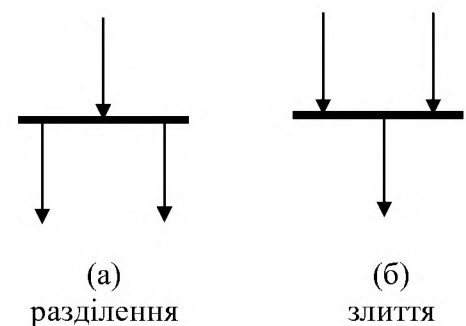


Рис. 3.10. Графічне позначення розділення та злиття паралельних потоків дій

злиття, навпаки, має декілька вхідних переходів і один вихідний (див. Рис. 3.10). Паралельні переходи на діаграмі діяльності можна зображувати в подовженій формі, а вхідні і вихідні переходи вертикальними стрілками.

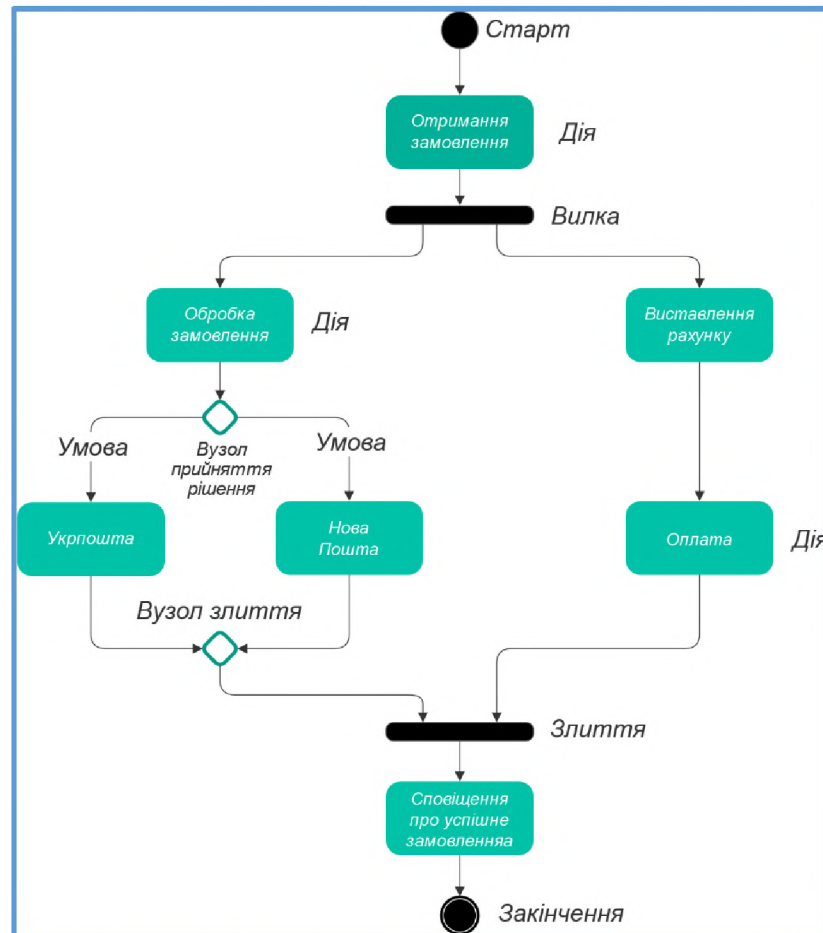


Рис. 3.11. Схематичне представлення діаграми діяльності

Діаграми діяльності можуть бути використані не лише для специфікації алгоритмів окремих підсистем, але і для моделювання взаємодії процесів в різних підсистемах. Це особливо актуально для моделювання процесів у апаратних засобах складних систем, де більшість блоків працюють одночасно, у розподілених системах, на підприємствах, які складаються з багатьох підрозділів, тощо. Для моделювання цих особливостей в мові UML запропонована спеціальна конструкція, що отримала назву «доріжки».

Доріжка (swimlane) - графічна область діаграми діяльності, що містить

елементи моделі, відповідальність за виконання яких належить окремим підсистемам. При цьому всі стани на діаграмі діяльності діляться на групи, розмежовані вертикальними або горизонтальними лініями. Це спосіб групування дій, які виконуються одним актором, або декількома акторами в одному потоці. Не слід додавати на діаграму більш як 5 доріжок одночасно. Розташовувати доріжки необхідно у логічному порядку, а їх назви (посади або актори) явно вказувати зверху або зліва доріжки. Пересікати лінію доріжки можуть лише переходи, які в цьому випадку позначають вихід або вхід потоку управління. Порядок розташування доріжок визначається міркуваннями зручності. На рисунку 3.12 наведений короткий приклад побудови діаграми діяльності з використанням доріжок.

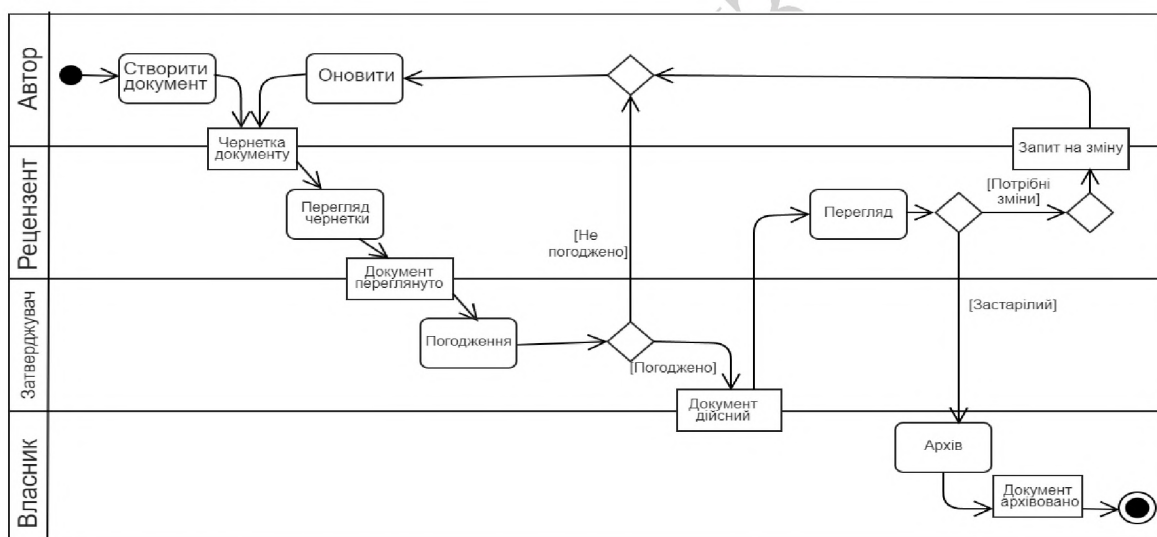


Рис.3.12. Діаграма діяльності для процесу «Робота з документом»

Випадки
застосування
Activity Diagram

Діаграма діяльності є одним із найефективніших інструментів аналізу вимог, особливо коли потрібно зрозуміти або описати послідовність дій, робочі процеси та логіку виконання сценаріїв у системі.

Наведемо доцільність використання діаграм діяльності під час аналізу вимог.

✓ *для опису бізнес-процесів і потоків робіт.* На етапі аналізу вимог діаграма діяльності допомагає візуалізувати послідовність дій користувача і системи, показати логіку бізнес-процесу від початку до кінця, визначити точки прийняття рішень, цикли, паралельні дії. Це дозволяє бізнес-аналітику і Замовнику однаково бачити, як система має працювати.

✓ *коли потрібно деталізувати сценарії використання.* Після створення діаграм випадків використання, діаграма діяльності допомагає розкрити, як саме реалізується кожен сценарій крок за кроком. Це полегшує розуміння логіки процесів і забезпечує повноту вимог.

✓ *для моделювання взаємодії користувача з системою.* Діаграма діяльності зручна, коли потрібно показати чергування дій користувача і системи, відобразити гілки умовного виконання («якщо/інакше»), узгодити інтерфейсні вимоги та реакцію системи на дії користувача.

✓ *для виявлення «вузьких місць» і неефективностей у процесі.* Під час аналізу вимог діаграма діяльності допомагає знайти дублювання дій, зайві перевірки чи етапи, залежності між процесами, моменти, де можна автоматизувати дії. Це важливо при підготовці бізнес-моделі або процесів оптимізації.

✓ *для опису паралельних або складних процесів.* Діаграма діяльності може відображати паралельне виконання кількох дій, асинхронну взаємодію між підрозділами або системами, обмін інформацією між підпроцесами.

✓ *для комунікації між аналітиками, Замовником і командою розробки.* Ця діаграма зрозуміла усім учасникам процесу, оскільки вона показує логіку дій у візуальній формі, не потребує знань програмування, може бути частиною документів «*Vision Document*» та «*Software Requirements Specification*».

✓ *для підготовки тестових сценаріїв.* На основі діаграми діяльності легко побудувати тест-кейси (кожен шлях через діаграму – це окремий сценарій),

варіанти виконання (гілки умовного вибору), тестування граничних ситуацій (цикли, паралельність, помилки).

Отже, на етапі аналізу вимог діаграма діяльності - це «інструмент логіки процесів», а діаграма станів – «інструмент поведінки системи». Їх часто використовують разом, щоб повністю охопити як послідовність дій, так і зміни станів.

3.5. *Sequence Diagram* - послідовність дій

Діаграма послідовності дій (*Sequence Diagram*) - це тип *UML*-діаграми, який відображає взаємодію між об'єктами системи у хронологічному порядку. Вона показує, які об'єкти беруть участь у сценарії та в якому порядку вони обмінюються повідомленнями. Діаграма послідовності використовується для:

- ✓ відображення послідовності викликів методів або повідомлень у межах певного сценарію;
- ✓ моделювання сценаріїв використання;
- ✓ проектування архітектури системи та взаємодії компонентів;
- ✓ аналізу часу виконання процесів і визначення залежностей між діями.

Нотація діаграми послідовності дій

Основними елементами діаграми послідовності дій є:

- ✓ *актор (Actor)* - зовнішній користувач або система, що ініціює взаємодію із системою, використовується, коли конкретна діаграма послідовності належить варіанту використання;
- ✓ *об'єкт (Object)* - елемент системи, який бере участь у сценарії;
- ✓ *лінія життя (Lifeline)* - вертикальна пунктирна лінія, що позначає існування об'єкта протягом часу;

- ✓ *повідомлення (Message)* - горизонтальна стрілка, яка показує передачу даних або виклик методу між об'єктами;
- ✓ *смуга активації (Activation strip)* - вертикальний прямокутник на лінії життя, який відображає період, коли об'єкт виконує дію або обробляє подію;
- ✓ *повернення повідомлення (Return message)* - пунктирна стрілка, що позначає повернення результату виконання дії.
- ✓ *фрагменти керування (Interaction fragments)* - спеціальні області, що показують логічні структури, наприклад *Loop* (повторення дій), *Alt* (альтернативні варіанти виконання (*if-else*)), *Opt* (умовне виконання).

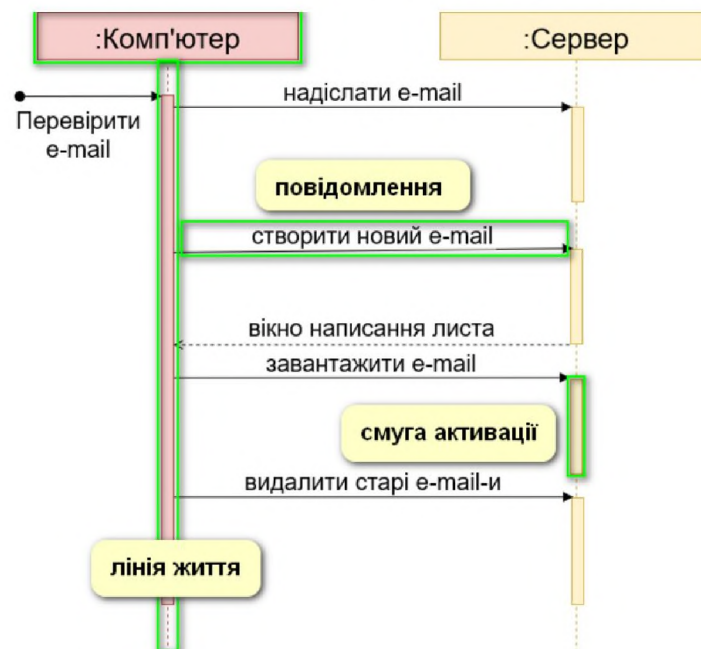


Рис. 3.13. Загальні елементи діаграми послідовності дій

Лінія життя починається з об'єкта-прямокутника (голова) та зображується вертикальною пунктирною лінією (стеблом). Вона служить для позначення періоду часу, протягом якого об'єкт існує в системі. Якщо об'єкт існує в системі постійно, то його лінія життя повинна продовжуватися по всій площині діаграми зверху донизу. Зазвичай об'єкти перераховуються зліва направо.

Смуга активації - тонкий прямокутник на лінії життя, протягом якого елемент виконує операцію. Довжина прямокутника вказує на тривалість перебування об'єктів в активному режимі.

Повідомлення (виклики) з'являються в послідовному порядку на лінії життя. Повідомлення зображується за допомогою стрілок. Початок стрілки завжди торкається лінії життя відправника та лінії життя об'єкта, що приймає повідомлення. Підпис може знаходитись над або всередині стрілки повідомлення. Для зручності перед повідомленням можна проставляти нумерацію дій.

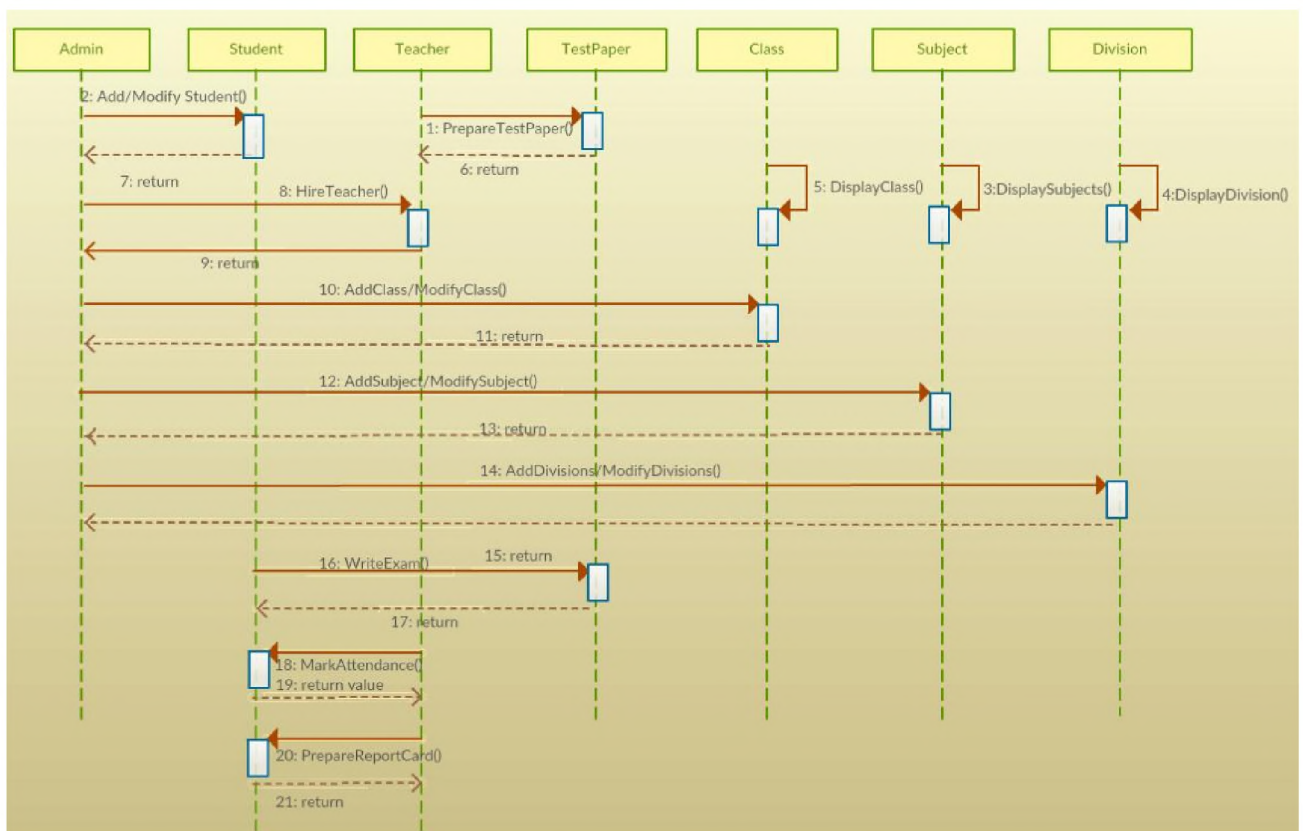


Рис.3.14. Приклад діаграми послідовності дій системи управління школою

Повідомлення можна розділити на такі категорії:

✓ *синхронне* повідомлення - очікує відповіді (лінія із зафарбованим вказівником);

- ✓ *асинхронне* повідомлення — не очікує відповіді (лінія зі звичайним вказівником);
- ✓ *рекурсивне* повідомлення - направлене до себе (починається та закінчується на одній лінії життя);
- ✓ повідомлення *повернення* (пунктирна лінія із зафарбованим вказівником) - зворотне повідомлення викликаючій стороні. Можна уникнути захаращення діаграм, вказуючи значення в самій стрілці початкового повідомлення.

Учасники не обов'язково живуть протягом часу взаємодії діаграми послідовності. Вони можуть створюватись і знищуватись у відповідності до повідомлень, які передаються.

Повідомлення «*Створення учасника*» - повідомлення, що створює нову лінію життя. Позначається пунктирною лінією, направленою до прямокутника учасника. Якщо створений учасник робить щось після його створення, слід додати вікно активації під його полем.

Повідомлення «*Видалення учасника*» - позначається хрестом в кінці лінії життя учасника. При цьому, якщо стрілка повідомлення направлена від одного учасника до іншого, значить, один учасник видаляє іншого. Якщо стрілка відсутня, учасник самознищується.

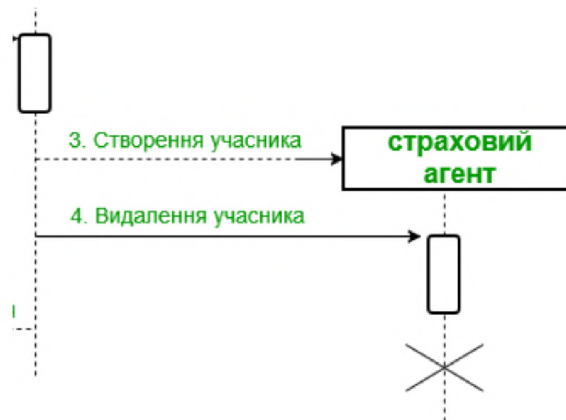


Рис. 3.15. Приклад створення та видалення учасника

Коментар - прямокутник із загнутим кутом. Може бути з'єднаний з об'єктом пунктирною лінією.

Фрагмент використовується, якщо процеси утворюють цикл або вимагають виконання умов для його закінчення. Він складається з вікна (рамки) та оператора фрагмента (напису) в п'ятикутнику зверху зліва.

Фрагмент діаграми послідовності (sd) використовується для оточення усієї діаграми. Після напису *sd* вказується назва діаграми.

Альтернативний фрагмент (alt) - показує один або декілька альтернативних сценаріїв, де виконується лише той, чия умова істинна. Для опису двох або більше альтернативних сценаріїв використовуються пунктирні лінії - операнд взаємодії. Кожен операнд має умову захисту в квадратних дужках.

Опціональний (не обов'язковий) фрагмент (opt) - виконується, лише якщо вказана умова, істинна. Він має лише одну умову і не поділяється на операнди. Використовується для опису не обов'язкового кроку робочого процесу.

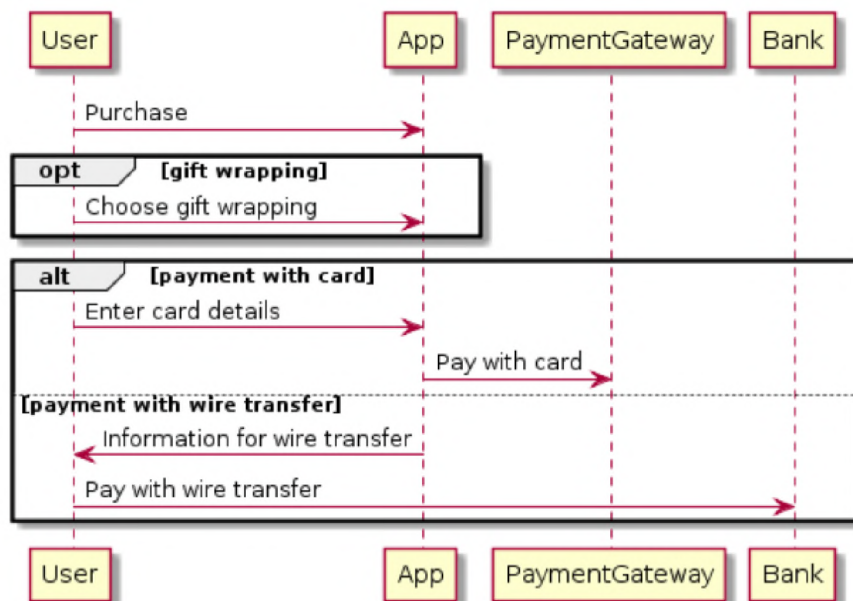


Рис. 3.16. Приклад фрагменту *opt*, щоб описати, як користувач може додати подарункове пакування та фрагменту *alt* для опису двох варіантів оплати

Фрагмент циклу (*loop*) - використовується для послідовності, що повторюється. Може мати межі ітерації, що пишуться біля назви *loop*.

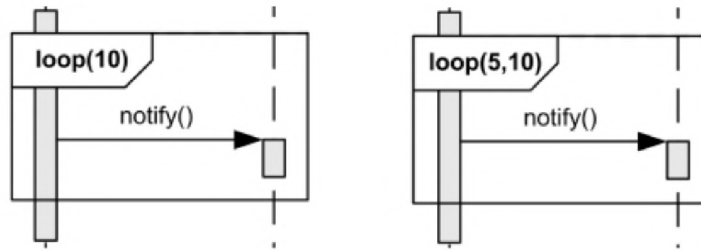


Рис. 3.17. Позначення фрагмента циклу

На рисунку 3.17 надпис *loop(10)* означає, що цикл має виконатись 10 разів, в той час надпис *loop(5,10)* встановлює межі: цикл має повторитись не менше 5 разів і не більше 10.

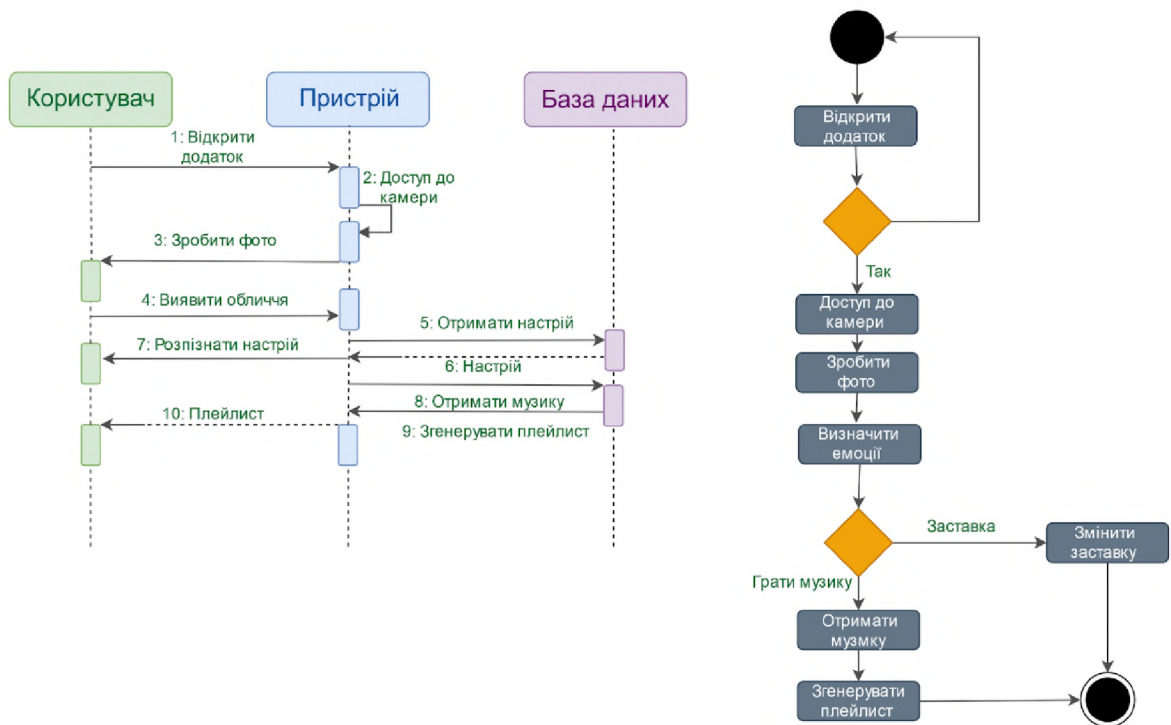


Рис.3.18. Різниця між діаграмою послідовності та діаграмою діяльності (музичний плеєр на основі емоцій)

Отже, діаграма послідовності - це візуальний інструмент, який доцільно застосовувати на етапі аналізу вимог, коли потрібно зрозуміти взаємодію між об'єктами системи у часі: хто, коли й у якому порядку виконує дії або обмінюється повідомленнями, а також для:

- ✓ *для деталізації сценаріїв використання.* Після створення діаграм випадків використання діаграма послідовності допомагає показати, як саме реалізується сценарій крок за кроком, тобто які об'єкти беруть участь у процесі, які повідомлення чи виклики передаються між ними, у якому порядку відбуваються події. Це дозволяє перетворити опис сценарію у чітку послідовність взаємодій.

- ✓ *для аналізу вимог до інтерфейсів.* У вимогах до інтеграцій діаграма послідовності дає змогу описати обмін даними між системами (запит-відповідь), уточнити формат, черговість і залежність викликів *API*, уникнути двозначностей у специфікації взаємодії. Це особливо актуально для вебсервісів, мікросервісної архітектури, інтеграції з зовнішніми системами (платіжні шлюзи, *CRM* тощо).

- ✓ *для виявлення пропущених або зайвих вимог.* Побудова діаграми послідовності допомагає побачити, чи всі потрібні кроки визначені, виявити зайві або нелогічні переходи, уточнити відповідальність кожного об'єкта чи ролі. Часто після побудови діаграми виявляється, що система не має описаної реакції на певну подію або що об'єкт виконує дії, які не входять у його логіку.

- ✓ *для комунікації між аналітиком, розробниками і Замовником.* Діаграма послідовності дуже зрозуміла візуальна форма для опису вимог, яка показує часовий порядок взаємодії між учасниками, допомагає усім стейкхолдерам бачити «живу картину процесу», спрощує узгодження вимог і майбутнього технічного дизайну.

- ✓ *для підготовки до тестування вимог.* На основі діаграми послідовності можна створювати тестові сценарії, коли кожен ланцюжок

повідомлень – це окремий тест, коли легко визначити, які модулі та дії мають бути протестовані. Це сприяє валідації логіки процесів ще до розроблення коду.

✓ *коли важлива часово-логічна послідовність.* Якщо процес вимагає строгого порядку дій, затримок або синхронізації, діаграма послідовності – незамінна, оскільки дозволяє чітко відобразити «хто, коли і кому» передає інформацію.

3.6. Нотація моделювання бізнес-процесів (BPMN-діаграми)

BPMN-діаграми (Business Process Model and Notation) активно використовують на етапі аналізу вимог, особливо у бізнес-аналізі та розробленні програмних систем, які автоматизують певні бізнес-процеси.

Основна мета аналізу вимог - зрозуміти, як працює бізнес сьогодні (*AS-IS*) і як він має працювати після впровадження системи (*TO-BE*). *BPMN-діаграми* дають змогу візуально відобразити логіку бізнес-процесів - послідовність дій, ролей, підрозділів, документів, подій, зрозуміти реальні потреби замовника, не заглиблюючись у технічні деталі та виявити неефективності, дублювання або прогалини у процесах.

BPMN-діаграми - це універсальна нотація, зрозуміла бізнес-користувачам (менеджерам, клієнтам), аналітикам (які збирають вимоги) та розробникам і тестувальникам (які реалізують систему). Вона виконує роль «спільної мови», яка допомагає узгодити вимоги і уникнути різночитань між бізнесом і технічною командою.

BPMN-діаграми допомагають визначити вхідні та вихідні дані процесів, знайти місця взаємодії користувачів із системою (де з'являються *use cases*), чітко сформулювати функціональні вимоги (які дії система має виконувати), виявити нефункціональні вимоги (час, послідовність, автоматизація, рольові обмеження).

Бізнес-аналітик створює дві моделі: *AS-IS* - як процес відбувається зараз (для виявлення проблем, зайвих дій, ручних операцій) та *TO-BE* - як він має виглядати після автоматизації або оптимізації. Це дає змогу чітко обґрунтувати цінність майбутньої системи та сформулювати вимоги на основі реальних бізнес-цілей.

Нотація *BPMN* допомагає розділити процеси між підрозділами (за допомогою *swimlanes*), показати взаємодію між різними системами або сервісами, визначити межі автоматизації - які кроки виконує людина, а які система.

BPMN-діаграма - це основа для подальших *UML*-моделей: кожен крок процесу може стати випадком використання (*use case*), потім для нього створюють діаграми послідовності, діяльності або станів, де розробники отримують чітке бачення, як система має підтримувати бізнес-логіку, а тестувальникам допомагає створювати тестові сценарії на основі потоків процесів, перевіряти, чи система підтримує усі варіанти виконання процесу (нормальні та альтернативні), здійснювати валідацію вимог із кінцевими користувачами.

Нотація *BPMN*- діаграми

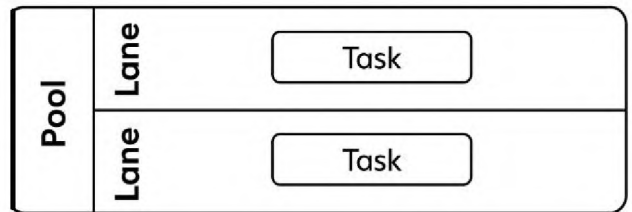
Мета нотації *BPMN* полягає у створенні чіткого зображення бізнес-процесу від його початку до кінця. Ця діаграма візуалізує послідовність бізнес-операцій, необхідних для переходу від однієї точки процесу до іншої. Однією з ключових переваг діаграми є гнучкість у створенні блок-схем різної складності - від простих до досить складних.

BPMN нотація базується на конкретних графічних елементах, які використовуються для візуалізації послідовності подій і взаємодій між різними етапами бізнес-процесу.

Ролі та зони відповідальності. У BPMN-діаграмах ролі й зони відповідальності відіграють ключову роль у визначенні учасників, їхніх відповідальностей і взаємодій в рамках бізнес-процесу.

Ролі відображають різних учасників або групи в рамках бізнес-процесу. Це може бути окрема особа, команда, відділ або зовнішня організація. На діаграмі ролі позначаються у вигляді пула (прямокутник із заокругленими кінцями) (*Pool*).

Зона відповідальності або «смуга» (*Lane*) визначає конкретну область або відділ, що несе відповідальність за виконання певних



функцій у бізнес-процесі. Lanes можуть бути частиною пула та вказувати на розподіл відповідальності між ролями. «Смуги» відображаються як горизонтальні або вертикальні розділювачі в пулі.

Задачі (Tasks) або *активності (Activities)*, що виконуються в рамках бізнес-процесу, позначаються на схемі у вигляді прямокутника з загнутими кутами.

Об'єкти потоку управління. Об'єкти потоку управління (*Flow Objects*) в нотації BPMN відображають послідовність дій, подій, рішень та умовних переходів у бізнес-процесах. Вони визначають, як дані й управління передаються від одного стану до іншого у рамках процесу.

Події (Event) - моменти, що відбуваються в процесі, такі як старт, завершення чи проміжкові. Наприклад, це може бути продаж послуги чи розмова з клієнтом. На діаграмі події позначаються кружечком або кружечком з іконкою, що вказує на тип події.

Початкова подія (Start Event) - ініціює процес, *проміжна подія (Intermediate Event)* відбувається між початком і кінцем процесу (наприклад,

отримання повідомлення, затримка), а *кінцева подія* (*End Event*) - завершує процес.

	початкові	проміжні	завершальні
	обробка	генерація	
просте			
повідомлення			
таймер			
помилка			
відміна			
компенсація			

Рис. 3.19. Позначення подій на діаграмі BPMN

Для моделювання рішень і визначення відносин між шляхами використовуються *розгалуження* або *шлюзи* (*Gateway*). На діаграмі бізнес-процесів вони позначаються у вигляді ромба з вказівкою, яка визначає умови вибору шляхів (ворота-розгалуження) або ромба з вказівкою, яка вказує, звідки об'єднуються шляхи (ворота-злиття).

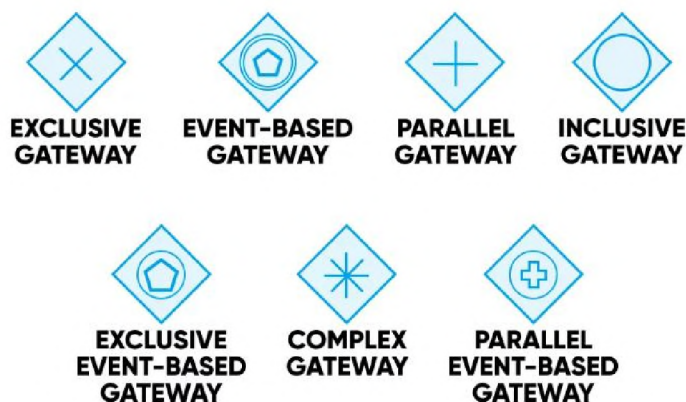


Рис.3.20. Позначення шлюзів на діаграмі BPMN

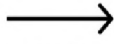
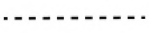
Основні типи шлюзів у BPMN:

- ✓ *ексклюзивний шлюз (Exclusive Gateway, XOR)* виконує одну гілку з кількох залежно від умов. Використовується, коли рішення взаємовиключні;
- ✓ *паралельний шлюз (Parallel Gateway, AND)* запускає усі гілки одночасно без перевірки умов. Використовується для паралельного виконання завдань;
- ✓ *інклюзивний шлюз (Inclusive Gateway, OR)* може активувати одну або кілька гілок залежно від умов. Дозволяє комбіноване виконання декількох шляхів;
- ✓ *комплексний шлюз (Complex Gateway)* використовується, коли логіка переходів складна (наприклад, необхідно завершити 2 із 3 паралельних завдань). Дає можливість задати спеціальні правила активації;
- ✓ *подієвий шлюз (Event-Based Gateway)* використовується, коли вибір гілки залежить від того, яка подія відбудеться першою. Використовується, коли потік реагує на зовнішні події.

З'єднуючі об'єкти. З'єднуючі об'єкти на діаграмі бізнес-процесів використовуються для визначення взаємозв'язків і потоків між подіями й активностями. Основні типи з'єднуючих об'єктів включають потік управління, потік повідомлень та асоціацію.

Послідовний потік (Sequence Flow) визначає порядок виконання дій та подій в бізнес-процесі. Позначається суцільною лінією із стрілкою, яка вказує напрямок потоку.

Потік повідомлень (Message Flow) визначає обмін повідомленнями між елементами бізнес-процесу, які можуть перебувати в різних ролях чи пулах. Зображується пунктирною лінією зі стрілкою, але з додатковими маркерами, що конкретизують, що це за повідомлення.

Асоціація (Association)   
 використовується для пов'язання **SEQUENCE FLOW** **MESSAGE FLOW** **ASSOCIATION FLOW**
 артефактів, текстових анотацій чи
 інших елементів з об'єктами процесу, не впливаючи на хід виконання процесу.

Артефакти. Артефакти в нотації *BPMN* використовуються для надання додаткової інформації, анотацій та допомагають розширювати розуміння процесів і забезпечувати контекстуальну інформацію.

Дані (Data Object) визначають дані, які використовуються або генеруються в бізнес-процесі (позначаються прямокутником з заголовком «*Data Object*»), *група (Group)* використовується для логічного згрупування елементів діаграми (зображуються прямокутником зі стрілкою, яка вказує на групування, а також містить ім'я групи) та *текстові анотації (Text Annotation)*, які надають додатковий опис чи пояснення елементів або зв'язків на діаграмі (позначаються лінією без стрілки, яка з'єднує елементи або артефакти). Асоціації часто використовуються для позначення взаємозв'язків без вказівки на порядок або напрямок.

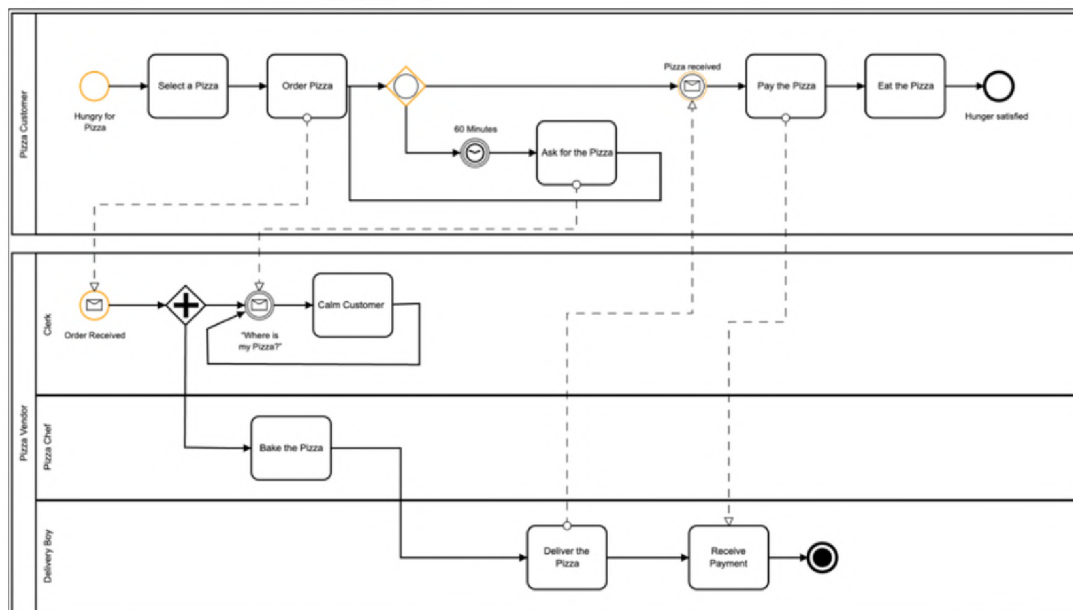


Рис.3.21. Приклад BPMN для демонстрації взаємодії різних учасників процесу

Випадки застосування BPMN-діаграм

В роботі бізнес-аналітиків нотація *BPMN* слугує не лише інструментом для створення діаграм, але й засобом комунікації між учасниками проєкту. Діаграма ефективна у відображенні різних бізнес-сценаріїв, від взаємодії клієнта з сервісом до керування проєктами. Основні випадки застосування даної нотації у бізнес-аналізі:

- ✓ візуалізація бізнес-процесів, оскільки *BPMN*-діаграми зрозумілі й легкі для використання як інструмент комунікації між різними учасниками проєкту;
- ✓ аналіз й моделювання процесів, оскільки *BPMN*-діаграми дозволяють легко виявляти покращення та оптимізаційні можливості;
- ✓ визначення бізнес-вимог, оскільки використання *BPMN*-діаграм допомагає конкретизувати бізнес-вимоги до процесів, включаючи вхідні й вихідні дані, взаємодії між учасниками й умови виконання.

3.7. *UML* чи *BPMN*?

UML і *BPMN* часто використовуються разом у проєктуванні програмних систем, але вони мають різне призначення, фокус і цільову аудиторію.

UML (уніфікована мова моделювання програмного забезпечення) описує структуру, поведінку й взаємодію системи, з метою допомогти розробникам і архітекторам створити логічну модель системи перед її реалізацією. *BPMN* (мова моделювання бізнес-процесів) описує, як бізнес виконує свої операції, а тому допомагає бізнес-аналітикам і менеджерам зрозуміти, оптимізувати та автоматизувати процеси.

UML використовується при розробленні програмного забезпечення, системному аналізі, моделюванні об'єктів, компонентів і сценаріїв використання та фокусується на системі як технічному продукті. *BPMN* використовується у

бізнес-аналізі, для моделювання бізнес-процесів, потоків роботи та ролей і фокусується на процесі як послідовності бізнес-дій.

Цільовою аудиторією *UML* є системні аналітики, архітектори, розробники, тестувальники, а *BPMN*: бізнес-аналітики, менеджери процесів, Замовники, фахівці з оптимізації бізнесу.

UML - більше технічна, ближча до програмної реалізації, а *BPMN*, навпаки, більш концептуальна, наближена до бізнес-мови.

UML часто виглядає як структурована логічна схема, близька до архітектурного креслення, а *BPMN* нагадує потокову діаграму (*flowchart*) із чітким відображенням ролей, рішень і подій.

Отож в загальному, *UML* і *BPMN* не конкурують, а доповнюють одне одного: якщо *BPMN* описує що робить бізнес, то *UML* описує як це реалізується у системі (наприклад, *BPMN* показує «як клієнт оформлює замовлення», а *UML* – «як система обробляє дані цього замовлення»). Разом вони створюють повну картину «від бізнесу до коду»: *BPMN* формує вимоги, *UML* допомагає реалізувати їх у системі.

Питання для самоперевірки



1. Що таке діаграма варіантів використання? Яка основна мета створення діаграми варіантів використання?
2. Хто такі актори в діаграмі варіантів використання? Які типи акторів існують?
3. Що таке «варіант використання» (*use case*)?
4. Як відображають зв'язки між акторами та варіантами використання?
5. Для чого використовується зв'язок *include* у діаграмі варіантів використання?

6. Для чого використовується зв'язок *extend* у діаграмі варіантів використання?
7. Що означає узагальнення між акторами у діаграмі варіантів використання?
8. Що означає узагальнення між прецедентами у діаграмі варіантів використання?
9. Які переваги використання діаграм варіантів використання при аналізі вимог?
10. Що таке діаграма станів? Для чого вона використовується в аналізі вимог?
11. Що таке стан системи? Що таке валідний стан?
12. Як відображається перехід між станами?
13. Які типи подій можуть спричинити зміну стану?
14. Що таке початковий і кінцевий стан у діаграмі?
15. Які системи доцільно моделювати за допомогою діаграми станів?
16. Як діаграма станів допомагає у виявленні вимог до поведінки системи?
17. Яка різниця між станом і дією у моделюванні?
18. Що описує діаграма діяльності? У чому її схожість із блок-схемами?
19. Які елементи є базовими для діаграми діяльності?
20. Яке призначення мають рішення?
21. Як відображаються паралельні дії в діаграмі діяльності?
22. Яке значення мають «смуги відповідальності» (swimlanes)?
23. Як діаграма діяльності допомагає у формуванні вимог користувача? Коли доцільно використовувати діаграму діяльності при аналізі вимог?
24. Яка різниця між діаграмою діяльності та діаграмою станів?
25. Як діаграма діяльності може показувати точки прийняття рішень?
26. Що таке діаграма послідовності? Яку інформацію вона відображає?
27. Що таке «життєва лінія»?
28. Як позначаються повідомлення між об'єктами?
29. Яке значення мають синхронні та асинхронні повідомлення?

30. Як відображається створення та знищення об'єкта на діаграмі?
31. У чому перевага діаграми послідовності над текстовим описом сценарію?
32. Як діаграма послідовності допомагає уточнювати вимоги?
33. Яка різниця між діаграмою послідовності та діяльності?
34. Як у діаграмі послідовності показуються альтернативні сценарії?
35. Що таке *BPMN*? Яка основна мета використання *BPMN*-діаграм?
36. Які основні типи елементів використовуються в *BPMN*?
37. Що таке події у *BPMN*?
38. Що таке дії або задачі у *BPMN*?
39. Яке призначення шлюзів у *BPMN*?
40. Що означають «*Pool*» та «*Lane*» у *BPMN*?
41. У чому полягає різниця між *BPMN* і *UML*?
42. Коли доцільно застосовувати *BPMN* при аналізі вимог? Які переваги *BPMN* для комунікації між бізнесом і технічною командою?
43. Які типи акторів можуть існувати на діаграмі варіантів використання?
44. Що означає варіант використання (*use case*) і яку інформацію він відображає про взаємодію користувача із системою?
45. Яким чином діаграма варіантів використання пов'язана з текстовим описом варіантів використання та специфікацією вимог?
46. Що таке моделювання вимог до програмного забезпечення і яке його місце в процесі розробки програмного забезпечення?
47. Які основні підходи та нотації використовуються для моделювання вимог до програмного забезпечення (наприклад, *UML*, *BPMN*)?
48. Як моделювання вимог сприяє кращому розумінню системи між замовниками, аналітиками та розробниками?

Тема 4

ПРІОРИТЕЗАЦІЯ ФУНКЦІЙ ТА ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Методологія визначення пріоритетів

Пріоритезація вимог - це процес визначення відносної важливості або черговості реалізації вимог до програмного забезпечення з метою допомогти команді зосередитися на тих вимогах, які приносять найбільшу цінність користувачам і бізнесу в межах обмежених ресурсів (часу, бюджету, персоналу).

Основні цілі, які ставить перед собою процес пріоритезації:

- ✓ забезпечити оптимальне використання ресурсів під час розробки;
- ✓ визначити мінімально життєздатний продукт (*Minimum Viable Product, MVP*);
- ✓ узгодити очікування Замовника, користувачів і Виконавця;
- ✓ сприяти поетапній реалізації вимог з найвищою цінністю.

Основні критерії оцінки вимог, які впливають на визначення їх пріоритетів:

- ✓ бізнес-цінність (наскільки вимога важлива для досягнення бізнес-цілей);
- ✓ складність реалізації (скільки ресурсів потрібно);
- ✓ вартість реалізації;
- ✓ критичність для користувача;
- ✓ терміновість.

Методологія пріоритезації продукту дає можливість перейти від нескінченної словесної аргументації до кількісних оцінок, діаграм та матриць, які безпосередньо пов'язані з відгуками клієнтів та продуктовою стратегією.

Нижче приведено перелік із 12 найбільш розповсюджених методів пріоритезації:

- ✓ *RICE*
- ✓ *ICE*
- ✓ *Value vs Effort*
- ✓ Модель Кано
- ✓ *Story mapping*
- ✓ *MoSCoW*
- ✓ *Opportunity scoring*



- ✓ *Product tree*
- ✓ *Cost of delay*
- ✓ *Buy a feature*
- ✓ *Weighted scores*
- ✓ *WSJF*

4.2. Кількісні методи пріоритезації

Кількісні методи пріоритезації - це підходи, які використовують числові оцінки, формули або матриці для об'єктивного визначення пріоритетів вимог, функцій або ідей у проєкті. На відміну від якісних (суб'єктивних) методів, вони дозволяють виміряти та порівняти цінність і зусилля реалізації кожної вимоги на основі обґрунтованих показників.

Метод
RICE

Ця система оцінки вимірює кожну функцію або ініціативу за чотирма параметрами: охоплення (*Reach*), вплив (*Impact*),

достовірність (*Confidence*) та зусилля (*Effort*). Що означає кожен із параметрів і як його можна оцінити кількісно?

Reach. Скількох користувачів торкнеться ця функція за певний період? Наприклад, користувачів за квартал, транзакцій на місяць.

Impact. Який вплив ця функція здійснить на користувача (наприклад, наскільки ця функція вплине на коефіцієнт конверсії)? Параметр впливу має певну шкалу:

- ✓ 3 - дуже високий вплив;
- ✓ 2 - високий вплив;
- ✓ 1 - середній вплив;
- ✓ 0,5 - слабкий вплив;
- ✓ 0,25 – мінімальний вплив.

Confidence. Наскільки можна бути упевненим в оцінці параметрів *Reach* і *Impact* (чи достатньо даних є, щоб підтвердити оцінку)?

- ✓ 100% - висока достовірність;
- ✓ 80% - середня достовірність;
- ✓ 50% - низька достовірність.

Effort. Скільки часу потрібно інвестувати у цю ініціативу (який обсяг ресурсів, необхідний для реалізації)? Параметр зазвичай оцінюється у людино-тижнях або людино-днях.

Всі ці фактори враховуються в одній загальній формулі, яка допомагає команді

$$\frac{\text{Reach} \times \text{Impact} \times \text{Confidence}}{\text{Effort}} = \text{RICE Score}$$

розробників уніфікувати підхід до визначення того, які ініціативи необхідно



додати до дорожньої карти продукту і визначити черговість їх реалізації. Чим більший RICE Score - тим вищий пріоритет вимоги.

Отриману оцінку можна використовувати для ранжування ініціатив будь-якого типу, щоб визначити, в якому порядку вони будуть втілюватись.

Функція (вимога)	Reach	Impact	Confidence	Effort	RICE Score
Новий екран реєстрації	800	2	90%	3	$(800 \times 2 \times 0.9) / 3 = 480$
Автоматичне оновлення профілю	400	3	80%	2	$(400 \times 3 \times 0.8) / 2 = 480$
Темна тема інтерфейсу	200	3	70%	1	$(200 \times 3 \times 0.7) / 1 = 420$

Рис.4.1. Приклад оцінювання пріоритету методом RICE



Переваги методу RICE:

- ✓ кількісний і структурований підхід;
- ✓ прозорість у прийнятті рішень;
- ✓ легкість застосування у беклогах, дорожніх картах продукту, спринтах;
- ✓ дозволяє збалансувати бізнес-цінність і затрати.



Недоліки методу RICE:

- ✗ не враховує залежності (непоодинокі випадки, коли ініціативу з високим показником RICE слід депріоритизувати на користь іншої, і командам варто сприймати цю систему оцінки швидше як модель, а не як спосіб ухвалити остаточне і безповоротне рішення);
- ✗ оцінки ніколи не точні на 100% (не завжди доступні точні дані для розрахунку Reach або Impact).

Метод
ICE

Якщо ж шукати метод швидкої пріоритизації, то модель ICE підійде набагато краще, ніж RICE, оскільки вона допомагає швидко оцінити, які ініціативи принесуть найбільшу користь при

мінімальних зусиллях (коли часу або даних недостатньо для детальної аналітики як у *RICE*). Оцінка пріоритету реалізації функції за даною моделлю формується відповідно до трьох параметрів: вплив (*Impact*), достовірність (*Confidence*) та простота (*Ease*). Що означає кожен із параметрів *ICE* і як його можна оцінити кількісно?

Impact. Який вплив на користувача здійснить впровадження цієї ініціативи? Наскільки сильно реалізація задачі вплине на користувача або бізнес? Оцінюється від 1–10 (де 10 - найвищий вплив).

$$\text{Impact} \times \text{Confidence} \times \text{Ease} = \text{ICE Score}$$

Confidence. Наскільки впевнені у своїй оцінці впливу (тобто, наскільки достовірні дані або експертна думка). Оцінюється від 1–10 (де 10 - найвищий вплив).

Ease. Наскільки легко реалізувати ініціативу: чим простіше, тим вищий бал. Оцінюється від 1–10.

Функція (вимога)	<i>Impact</i>	<i>Confidence</i>	<i>Ease</i>	<i>ICE Score</i>
Оптимізація сторінки входу	9	8	7	504
Темна тема інтерфейсу	7	7	9	441
<i>Push</i> -сповіщення	10	6	5	300

Рис.4.2. Приклад оцінювання пріоритету методом *ICE*

Модель *ICE* дає систему відносної пріоритезації, а не точних підрахунків, заснованих на даних. Щоб мінімізувати неоднорідність оцінок, необхідно переконатися, чи вірно визначається кожен параметр у системі *ICE*, а саме домовитись всередині команди, що саме означає, наприклад, *Impact* 5, *Confidence* 7, *Ease* 3.



Переваги ICE-методології:

- ✓ не потребує складних розрахунків або великої кількості даних;
- ✓ придатна для будь-якої сфери (від ІТ-продуктів до освітніх проєктів);
- ✓ легкість використання на воркшопах чи мозкових штурмах;
- ✓ підтримує командне узгодження пріоритетів.



Недоліки ICE-методології:

- ✗ висока суб'єктивність оцінок (особливо *Impact* та *Ease*);
- ✗ не враховує охоплення користувачів (як у *RICE*);
- ✗ менш точна для великих проєктів або стратегічних рішень.

Метод 100 балів *100-Dollar Test*

100-Dollar Test - це один із найпростіших і водночас ефективних кількісних методів пріоритезації вимог, який базується на оцінці відносної важливості кожної вимоги через розподіл обмеженого ресурсу (балів, «доларів», жетонів тощо).

Його часто застосовують у бізнес-аналізі та у *Agile*-командах для формування беклогу або *MVP*-плану.

Основна ідея методу полягає в тому, що кожен учасник процесу оцінювання пріоритетів функцій (Замовник, користувач, аналітик, розробник) отримує 100 «одиниць цінності» (умовних балів або доларів), які він розподіляє між усіма функціями системи відповідно до правила: «Якщо у вас обмежений бюджет (100 одиниць), на що б ви його витратили насамперед?»

Покрокова
реалізація
методу

1. *Визначення можливостей продукту.* Створюється перелік усіх функцій (вимог, епіків, історій користувачів тощо) програмного продукту, які потрібно оцінити (зазвичай це 5–20 елементів, щоб уникнути перевантаження).

2. *Розподіл «бюджету».* Кожен учасник отримує 100 балів (або доларів) і розподіляє їх між функціями залежно від власної оцінки важливості (важливі вимоги отримують більше балів, менш критичні - менше або нуль). Наприклад, учасник вважає, що

реєстрація - критично важлива (30 балів),
пошук - 25,
фільтрація - 15,
темна тема інтерфейсу - 10,
push-сповіщення — 20.
Разом: 100 балів.

3. *Підсумок результатів.* Усі оцінки об'єднуються в загальну таблицю або діаграму.

Функція (вимога)	Користувач 1	Користувач 2	Користувач 3	Σ (сума)
Реєстрація	30	25	35	90
Пошук	25	30	20	75
Фільтрація	15	20	10	45
Темна тема	10	10	5	25
Push-сповіщення	20	15	30	65

4. *Ранжування.* Розраховується загальна кількість балів для кожної функції, а потім вони сортуються за спаданням - від найвищої суми (найважливіша) до найнижчої. У прикладі вище найвищий пріоритет має функція «Реєстрація» (90 балів), а найнижчий – функція «Темна тема інтерфейсу».

5. *Інтерпретація результатів.* Топ-3 функції - кандидати на реалізацію в найближчому спринті або MVP, середні значення - для наступних ітерацій, а нижні позиції - можуть бути відкладені або виключені.

Метод не дає абсолютної оцінки, але чітко відображає колективне сприйняття пріоритетів.



Переваги методу *100-Dollar Test*:

- ✓ не потребує складних розрахунків;
- ✓ дозволяє врахувати думку всіх зацікавлених сторін;
- ✓ легко пояснити результати менеджеру або клієнту;
- ✓ можна проводити офлайн (жетони, картки) або онлайн (*Miro, Mural, Hygger, ProductPlan*).



Недоліки методу *100-Dollar Test*:

- ✗ залежить від учасників (можуть бути упередження або різні інтереси);
- ✗ не враховує складність реалізації (оцінюється лише цінність, не зусилля);
- ✗ потрібен консенсус (якщо учасники з різних груп мають різні бачення).

Тому іноді метод комбінують із іншими (наприклад, *Value vs Effort* або *Weighted Scoring*).

Метод купівлі
функції
Buy a Feature

Buy a Feature - це інтерактивний метод пріоритезації можливостей продукту, який моделює ринкову поведінку користувачів або стейкхолдерів з метою визначити, які функції продукту є для них справді цінними, і чому. Цей метод використовується не лише для кількісного ранжування функцій, а й для глибшого розуміння мотивації та очікувань користувачів.

В методі *Buy a Feature* кожна функція розглядається як товар, який можна «купити». Учасники отримують обмежену кількість умовних коштів (наприклад, \$100) і мають вирішити, які функції вони куплять. Мета застосування методу: виявити, які функції цінні настільки, що користувачі готові «витратити» свій обмежений ресурс на їх реалізацію.



1. *Визначення функцій.* Створюється список функцій майбутнього продукту.

2. *Призначення «вартості» кожній функції.*

Визначається вартість як зусилля/ресурси для розробки (дорожчі функції = складніші для реалізації).

3. *Розподіл бюджету між учасниками.* Кожен отримує фіксовану суму «коштів» (наприклад, \$100) - це створює умови обмеженого вибору.

4. *Процес купівлі.* Учасники купують функції, які хочуть бачити в продукті. Вони можуть купувати індивідуально одну дорогу функцію, або декілька дрібних, або об'єднуватися в групи, щоб зібрати на дорожчу функцію.

5. *Аналіз результатів.* Визначається, які функції «продалися», оскільки вони є найбільш пріоритетні для користувачів. Типові результати:

- ✓ функції, які купили всі - найвищий пріоритет (*must-have*);
- ✓ функції, які купили частково - середній пріоритет (*should-have*);
- ✓ функції, які ніхто не купив - низький пріоритет або неактуальні.

Після завершення «гри» можна скласти таблицю «продажів» та візуалізувати її графіком.



Переваги методу *Buy a Feature*:

- ✓ учасники активно обговорюють і аргументують свій вибір;

✓ під час дискусії з'являється глибше розуміння мотивації («чому саме ця функція важлива»);

✓ не вимагає складних обчислень;

✓ можна проводити офлайн (жетони, картки) або онлайн (*Miro, Mural, Hygger, ProductPlan*).



Недоліки методу *Buy a Feature*:

✗ рішення базуються на відчутті цінності, а не на фактичних метриках;

✗ обмежена кількість учасників (метод працює найкраще з 5–12 людьми);

✗ не показує прямо складність реалізації функції, хоча її «вартість» може це відображати.

Методи *Buy a Feature* та *100-Dollar Test*, як видно, є споріднені, але не зовсім однакові. Вони базуються на одній ідеї розподілу обмеженого ресурсу (умовних коштів чи балів) між функціями, однак відрізняються цілями, форматом та поведінковими аспектами. Порівняння цих двох методів наведено в таблиці 4.1.

Таблиця 4.1 - Порівняльна характеристика методів
Buy a Feature та *100-Dollar Test*

Характеристика	<i>Buy a Feature</i>	<i>100-Dollar Test</i>
Тип	Групово гра	Індивідуальне оцінювання
Мета	Виявити колективні пріоритети і поведінку	Зібрати кількісну статистику
Формат	Інтерактивна сесія	Табличне/онлайн-опитування
Результат	«Куплені» функції + інсайти з дискусій	Рейтинг функцій за балами

Як висновок: обидва методи базуються на ідеї обмеженого бюджету, але *Buy a Feature* - ігровий і поведінковий, а *100-Dollar Test* - аналітичний і опитувальний.

Метод
«Цінність-Зусилля»
Value vs Effort

Value vs Effort - це візуальний і кількісний метод пріоритезації, який допомагає визначити, які завдання, функції або ідеї варто реалізувати першими, виходячи з їх цінності для бізнесу/користувача та обсягу необхідних зусиль.

У методі *Value vs Effort* кожен функцію або ідею оцінюють за двома ключовими параметрами:

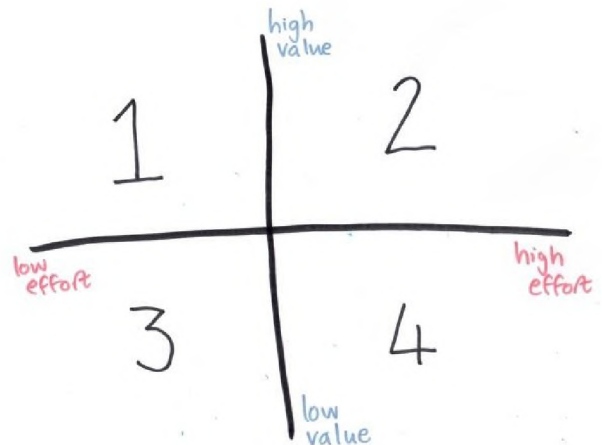
- ✓ цінність (*Value*) - наскільки реалізація функції принесе користь користувачу, бізнесу або системі (наприклад, підвищення прибутку, зручності, продуктивності, задоволення користувача тощо).

- ✓ зусилля (*Effort*) - скільки ресурсів, часу або складності потребує реалізація функції (у людино-днях, спринтах, вартості тощо).

Потім кожну можливість (функцію) системи заносять у двовимірну матрицю ($X-Y$), щоб візуально побачити, що варто робити в першу чергу. Матриця має 2 осі:

- ✓ вісь $X \rightarrow \text{Effort (зусилля)}$ → від «мало» до «багато»;

- ✓ вісь $Y \rightarrow \text{Value (цінність)}$ → від «низька» до «висока».



Для роботи з матрицею можна використовувати велику дошку і просто намалювати на ній великий знак плюс «+», задати напрямки осей *Value* та *Effort* та розміщувати задачі згідно цим критеріям.

Щоправда, все це набагато зручніше робити за допомогою готового інструменту, який пропонують багато платформ для керування продуктами (наприклад, *airfocus.com*, *Aha! Roadmaps*, *Hygger.io* тощо) (див.Рис. 4.3.-4.5).

При створенні матриці *Value vs Effort* будь-яким зручним способом, утворюється 4 квадранти, у кожному з яких визначаються різні стратегії дій (див. Табл. 4.2).

Таблиця 4.2 - 4 квадранти матриці *Value vs Effort*

Квадрант	Назва	Характеристика	Рішення
Висока цінність / Низькі зусилля	<i>Quick Wins</i> (Швидкі перемоги)	Дас значну вигоду при мінімальних витратах	Реалізувати в першу чергу
Висока цінність / Високі зусилля	<i>Major Projects</i> (Стратегічні ініціативи)	Важливі, але потребують багато ресурсів	Планувати, коли доступні ресурси
Низька цінність / Низькі зусилля	<i>Fill-ins</i> (Дрібні покращення)	Невеликі задачі, які можна робити «між справою»	Реалізувати за наявності вільного часу
Низька цінність / Високі зусилля	<i>Time Wasters</i> (Марна трата часу)	Потребують багато ресурсів, але мало користі	Уникати або відкладати



Рис.4.3. Побудова матриці *Value vs Effort* за допомогою AI-powered Product Management Software *airfocus.com*

2x2 prioritization matrix

Example

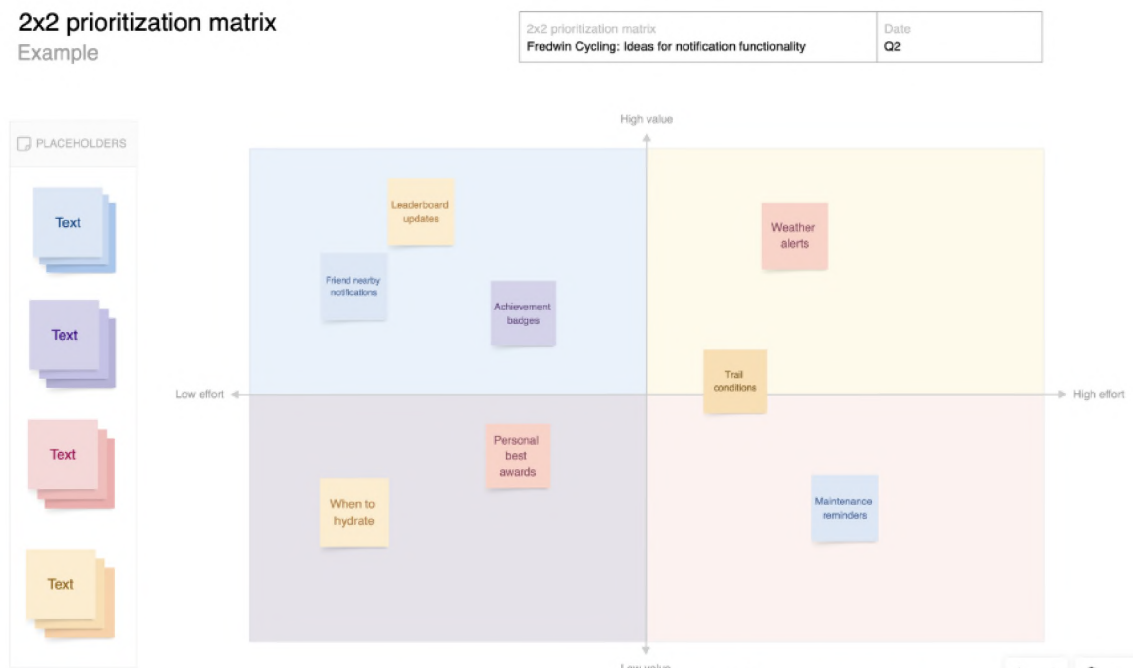


Рис. 4.4. Побудова матриці Value vs Effort за допомогою Product Management Tool Aha! Roadmaps

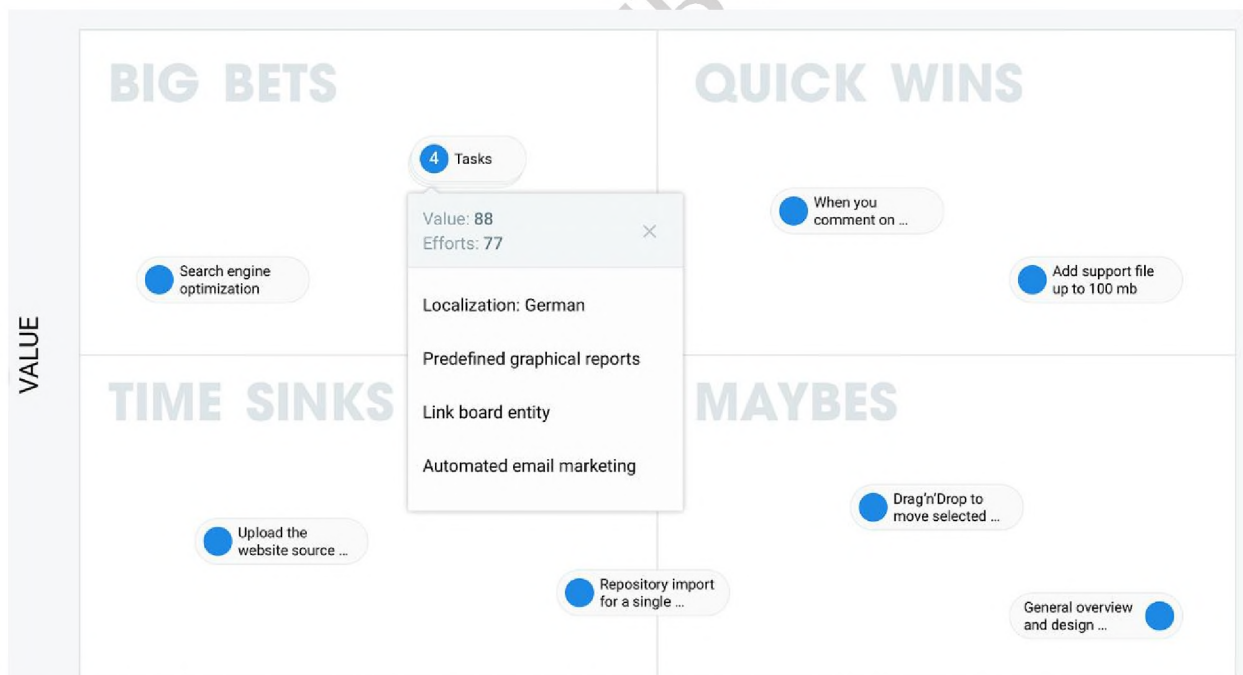


Рис. 4.5. Побудова матриці Value vs Effort за допомогою Software Development Tool for Product Makers Hygger.io



1. *Збір вимог*. Скласти перелік усіх функцій, задач або ідей, які потрібно оцінити.

2. *Визначення критеріїв оцінювання*. Визначити, як саме будете оцінюватись «цінність» та «зусилля» (наприклад, *Value* може базуватись на прирості доходу, задоволенні користувачів, стратегічній важливості, конкурентній перевазі, а *Effort* може вимірюватися через кількість людино-днів, складність реалізації, технологічні ризики).

3. *Оцінювання*. Кожну функцію, задачу або ідею необхідно оцінити за шкалою від 1 до 10 або іншими числовими показниками (наприклад, у балах чи відсотках).

4. *Побудова матриці пріоритетів вимог*. Нанести усі вимоги на координатну площину (*Value* на вісь *Y*, *Effort* на вісь *X*): кожна точка = одна функція.

5. *Аналіз матриці і ранжування вимог*. Визначити, які вимоги належать до «*Quick Wins*», а які варто відкласти або переглянути, наприклад:

Функція (вимога)	Value	Effort	Квадрант
Оптимізація пошуку	9	3	<i>Quick Win</i> (1 квадрант)
Темна тема інтерфейсу	6	2	<i>Fill-in</i> (1 квадрант)
Автоматичне резервне копіювання	8	8	<i>Major Project</i> (2 квадрант)
Інтеграція з CRM	3	9	<i>Time Waster</i> (4 квадрант)



Переваги методу *Value vs Effort*:

- ✓ легко сприймається на мітингах, воркшопах, презентаціях;
- ✓ зрозумілий навіть не фахівцям;
- ✓ можна адаптувати критерії під будь-який тип проекту;
- ✓ сприяє узгодженню між стейкхолдерами (всі бачать спільну картину).



Недоліки методу *Value vs Effort*:

- ✗ оцінки «цінності» часто залежать від думки окремих осіб;
- ✗ не враховує ризики та невизначеність;
- ✗ обмежена точність (підходить більше для стратегічного огляду, ніж для точних планів).

Метод зваженого
оцінювання
Weighted Scoring

Weighted Scoring - це найгнучкіший і найобґрунтованіший кількісний підхід до пріоритезації функцій та можливостей продукту, який дозволяє чітко пояснювати рішення перед Замовником, балансувати між бізнес-цінністю, ризиками й ресурсами та підвищити прозорість у командному прийнятті рішень.

Weighted Scoring - це кількісний метод пріоритезації, який дозволяє оцінити кожную можливість продукту за кількома критеріями з урахуванням їх відносної важливості (ваги) з метою прийняття обґрунтованих рішень про черговість їх реалізації, базуючись на цифрах, а не на інтуїції.

У методі *Weighted Scoring* кожна функція оцінюється за набором критеріїв (наприклад: бізнес-цінність, ризик, вартість, вплив на користувача), при чому кожному критерію надається вага (*weight*), тобто наскільки він важливий у загальній оцінці. Потім всі оцінки перемножуються на ваги, підсумовуються і отримується підсумковий бал (*Score*).

$$\text{Weighted Score} = \sum_{i=1}^n (W_i \times S_i)$$

Підсумковий бал складається з W_i - ваги критерію i (вираженої у відсотках або дробовій частці), S_i - оцінці функції за критерієм i (наприклад, від 1 до 5 або від 1 до 10) та n - кількості критеріїв (важливо, що сума всіх ваг = 1 (або 100%)).

Покрокова
реалізація
методу

1. *Визначення критеріїв оцінювання.* Вибираються 3–6 критеріїв, які найкраще відображають значущість функцій (вимог) у конкретному контексті (наприклад, бізнес-цінність, користувацький ефект, терміновість, ризик або невизначеність, вартість реалізації, технічна складність).

2. *Призначення ваг критеріїв.*

Кожному критерію надається вага, що відображає його відносну важливість. Наприклад, вага критерію «бізнес-цінність» - 40%, «користувацького ефекту» - 30%, «складність реалізації» - 20% та «ризик» - 10%. Разом 100%.



3. *Оцінювання.* Кожну функцію (вимогу) оцінюють за всіма критеріями за шкалою (наприклад, від 1 до 5, де 1 - дуже низько, 5 - дуже високо). Найвищі оцінки означають вищий пріоритет.

4. *Розрахунок підсумкових балів за формулою.*

Функція (вимога)	Бізнес-цінність (0.4)	Користувацький ефект (0.3)	Складність (0.2)	Ризик (0.1)	Підсумок
Реєстрація користувачів	5	4	2	3	$(5 \times 0.4) + (4 \times 0.3) + (2 \times 0.2) + (3 \times 0.1) = 3.9$
Пошук	4	5	3	2	$(4 \times 0.4) + (5 \times 0.3) + (3 \times 0.2) + (2 \times 0.1) = 3.9$
Автозбереження	3	3	2	2	$(3 \times 0.4) + (3 \times 0.3) + (2 \times 0.2) + (2 \times 0.1) = 2.7$
Інтеграція з CRM	9	8	5	5	$(9 \times 0.4) + (8 \times 0.3) + (5 \times 0.2) + (5 \times 0.1) = 7.5$
Темна тема інтерфейсу	1	2	1	1	$(1 \times 0.4) + (2 \times 0.3) + (1 \times 0.2) + (1 \times 0.1) = 1.3$



Переваги методу *Weighted Scoring*:

- ✓ рішення базуються на цифрах, а не на інтуїції;
- ✓ зрозуміло, чому певна функція (вимога) отримала саме такий пріоритет;
- ✓ можна змінювати ваги під конкретний проєкт чи етап;
- ✓ придатність для колективного обговорення (легко узгоджувати думки різних стейкхолдерів).



Недоліки методу *Weighted Scoring*:

- ✗ потребує часу для обговорення критеріїв і ваг;
- ✗ оцінки можуть бути суб'єктивними, якщо немає точних даних;
- ✗ не завжди враховує залежності (наприклад, технічні залежності).

Метод першого
зваженого
найкоротшого
завдання
WSJF

Weighted Shortest Job First (WSJF) - це один із найпотужніших кількісних способів пріоритизації в *Agile*-методології. Він допомагає командам оцінювати економічну вигоду від швидкої реалізації, приймати раціональні рішення при обмежених ресурсах, підтримувати гнучке, але обґрунтоване планування. У *SAFe* (*Agile* для великих організацій) його рекомендують як основний стандартний метод пріоритизації беклогу на рівні функцій, епіків та можливостей.

Метод *WSJF* допомагає відповісти на ключове питання: Яку роботу варто виконати першою, щоб отримати максимальний економічний ефект у мінімальний час? Для цього використовується співвідношення «вартість

$$WSJF = \frac{\text{Cost of Delay}}{\text{Job Size (Job Duration)}}$$

затримки (*Cost of Delay, CoD*) / розмір роботи (*Job Size*)». Чим більше співвідношення - тим вищий пріоритет.

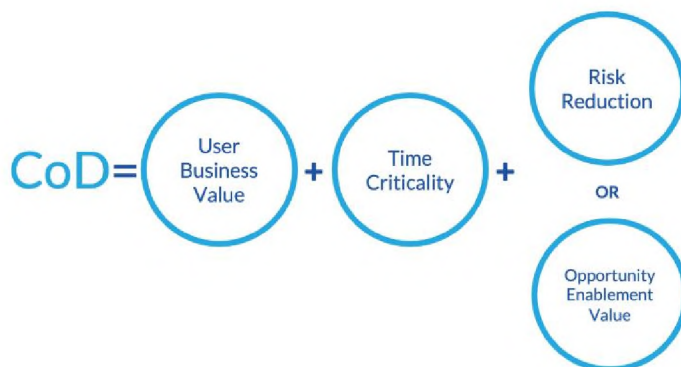
Вартість затримки (*Cost of Delay*) - оцінка втрат або упущених можливостей у разі відтермінування реалізації функції (вимоги).

Розмір роботи (*Job Size*) - обсяг ресурсів, часу, необхідних для виконання.

У *SAFe* пропонується розбивати *Cost of Delay* на три компоненти, які оцінюються окремо:



- ✓ *Business Value* (бізнес-цінність): яку вигоду або прибуток принесе функція?
- ✓ *Time Criticality* (критичність у часі): наскільки важливо зробити це зараз?
- ✓ *Risk Reduction / Opportunity Enablement* (зниження ризику або створення можливостей): чи зменшує це ризики або відкриває нові можливості?



Щоб розрахувати загальний бал *Cost of Delay*, кожен компонент показника слід оцінити за шкалою від 1 до 10. Сума трьох чисел і визначить *Cost of Delay*.

Потім потрібно визначити шкалу для оцінки *Job Size*, яку треба виконати для кожної ідеї чи ініціативи. Шкала може

відрізнятись від шкали *Cost of Delay*, головне, щоб вона була єдиною для всіх ініціатив.

Визначити цей параметр може бути важко. Він змінюватиметься залежно від наявних ресурсів, навичок команди, залежностей і зовнішніх факторів. Але команда повинна домовитися і прийти до спільної методології, за якою можна оцінювати кожну ініціативу. Ті ініціативи, які отримують найнижчий рейтинг *Job Size*, будуть в більшому пріоритеті.

Поділивши перший параметр на другий і ранжувавши ініціативи в балах від більшого до меншого – отримаємо пріоритизований список.

Функція (вимога)	<i>Business Value</i>	<i>Time Criticality</i>	<i>Risk Reduction</i>	<i>CoD</i>	<i>Job Size</i>	<i>WSJF = CoD / Job Size</i>
Новий пошук	8	7	3	18	6	3.0
Оптимізація UI	6	4	2	12	3	4.0
Інтеграція з CRM	9	8	5	22	8	2.75
Темна тема інтерфейсу	3	2	1	6	1	6.0

Найвищу оцінку методом *WSJF* має функція (вимога) «Темна тема інтерфейсу», тому вона більш пріоритетна для реалізації (навіть якщо її бізнес-цінність невелика, але вона швидко реалізується і дає миттєвий результат).



Переваги методу *WSJF*:

- ✓ кількісна обґрунтованість;
- ✓ баланс між цінністю та витратами часу;
- ✓ усуває суб'єктивність (усі параметри оцінюються за єдиною шкалою);
- ✓ підтримує Agile-планування;
- ✓ добре працює для постійно змінних беклогів, коли ресурси обмежені.



Недоліки *WSJF*:

- ✗ потребує спільного узгодження шкал і критеріїв, що може бути складним;
- ✗ вимагає досвіду у правильному визначенні «вартості затримки»;
- ✗ не враховує залежності (одна функція може блокувати іншу);
- ✗ іноді створює спокусу переоцінювати короткі, але малозначущі завдання.

Підсумкова характеристика кількісних методів пріоритезації наведена у таблиці 4.3.

Таблиця 4.3 - Характеристики кількісних методів пріоритезації

Метод	Основна ідея	Принцип оцінки	Коли застосовувати
<i>RICE</i>	Визначення пріоритету за очікуваною цінністю і ресурсами	Формула: $(Reach \times Impact \times Confidence) / Effort$	Для стратегічного планування функцій і беклогу, коли є дані
<i>ICE</i>	Швидка оцінка ідей за впливом, впевненістю та простотою реалізації	Формула: $(Impact \times Confidence \times Ease)$	Для швидкої пріоритезації у стартапах, MVP, воркшопах
<i>100-Dollar Test</i>	Розподіл обмеженого «бюджету» балів між вимогами	Кожен учасник «інвестує» 100 балів у найцінніші функції	Коли потрібно врахувати думку стейкхолдерів або користувачів
<i>Buy a Feature</i>	«Купівля» бажаних функцій за ігрові кошти	Вибір функцій на основі їх вартості й обмеженого бюджету	Для фокус-груп, UX-воркшопів, колективного визначення цінності
<i>Value vs Effort</i>	Візуальний баланс між цінністю та зусиллями	Матриця 2×2: <i>Value</i> проти <i>Effort</i>	Для визначення «швидких перемог» і планування MVP
<i>Weighted Scoring</i>	Оцінювання за кількома критеріями з різними вагами	Формула: $Score = \sum (Оцінка \times Вага)$	Для обґрунтованої пріоритизації, коли потрібно врахувати багато факторів (цінність, ризик, вартість, тощо)
<i>WSJF</i>	Максимізація цінності за мінімальний час реалізації	Формула: $Cost\ of\ Delay / Job\ Size$	Для <i>Agile</i> / <i>SAFe</i> -планування

4.3. Якісні методи пріоритезації вимог

Якісні методи не використовують математичних формул. Вони ґрунтуються на думці експертів, логіці, досвіді та взаємодії учасників, і їх основна мета – це зрозуміти, що важливо і чому.

Метод MoSCoW

MoSCoW - це якісний метод пріоритезації функцій (вимог), який полягає в класифікації їх за рівнем важливості. Назва методу - акронім від перших літер категорій: *Must*, *Should*, *Could*, *Won't*. Метод використовується в бізнес-аналізі, управлінні продуктами, проєктному менеджменті для вирішення питання: що реалізувати обов'язково, а що можна відкласти?



Must have - обов'язкові завдання - це основа проєкту, невід'ємні елементи, без яких проєкт втрачає свою суть; це завдання найвищого пріоритету, які потребують негайної уваги. Щоб визначити, чи належить вимога до цієї категорії, слід запитати: чи може проєкт рухатися вперед без виконання цього завдання? Якщо відповідь негативна, то воно підпадає під категорію *Must have* завдань. Ці завдання є важливими для досягнення цілей проєкту та отримання значущих результатів у визначені терміни.

Should have - завдання, які повинні бути - вказують на важливі, але менш термінові вимоги, ніж обов'язкові; це другорядні пріоритети, які сприяють успіху проєкту; однак команда може впоратися з ними пізніше.

Щоб визначити *Should have*, слід перевірити, чи може команда завершити роботу пізніше, не ставлячи під загрозу хід проєкту. Якщо відповідь «так», то це відповідає даній категорії. Хоча *Should have* є життєво важливими, команда не вважає їх критичними для негайного виконання проєкту та може відкласти їх, якщо необхідно.



COULD HAVE

Could have - можливі завдання - це додаткові вимоги, які можуть розширити обсяг та функціональність проєкту, але не є життєво важливими для його успіху; ці завдання виступають у ролі бонусних функцій, надаючи додаткову цінність, якщо їх врахувати. Однак їхнє виключення не суттєво вплине на результати проєкту.

Вирішуючи, чи включати *Could have* завдання чи ні, слід оцінити їхні потенційні переваги порівняно з витратами та впливом на терміни проєкту. Якщо виключення цих завдань не шкодить іншим елементам проєкту, тоді їх можна включити; в іншому випадку -пропустити.

Won't have - невизначені завдання - це завдання, явно виключені з поточного обсягу проєкту. Проєктні команди вважають ці вимоги з низьким пріоритетом і можуть відкласти їх на майбутні ітерації або проєкти, хоча вони можуть мати потенційну цінність, але не є необхідними для досягнення цілей проєкту в задані терміни. Замість того, щоб повністю виключати *Won't have*



WILL NOT HAVE

завдання, проєктні команди ставлять їх у резерв для розгляду в майбутньому, коли умови проєкту сприятимуть їх реалізації. Такий підхід дозволяє проєктним командам зосередитися на завданнях високого пріоритету та ефективно оптимізувати зусилля.

Рисунок 4.6 є наочною інфографікою *MoSCoW*, яка пояснює логіку кожної категорії.

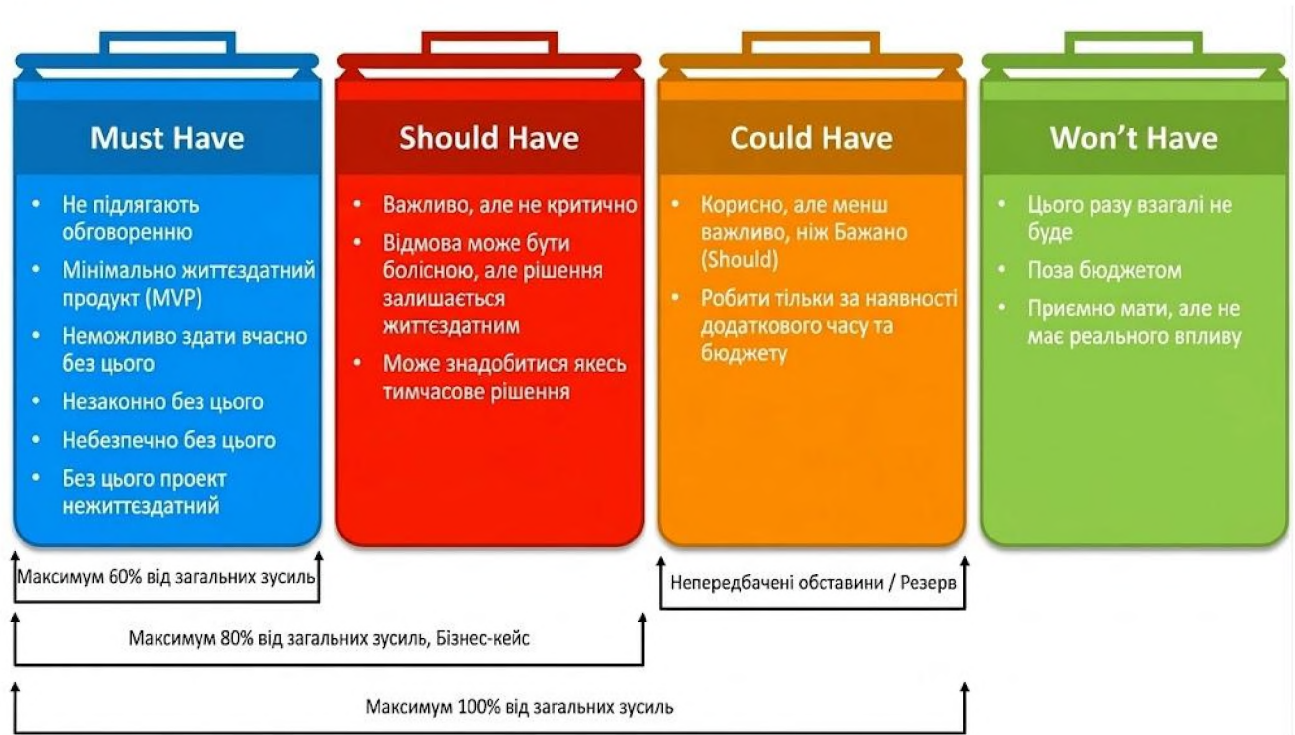


Рис.4.6. Інфографіка *MoSCoW*

Як видно з рисунка 4.6, у нижній його частині відображено орієнтовний розподіл ресурсів та загальних зусиль (*Total Effort*) між категоріями вимог відповідно до принципів методу *MoSCoW*. Зокрема, передбачається, що категорія *Must Have* повинна охоплювати не більше 60% загального обсягу робіт, оскільки саме ці вимоги формують базову функціональність системи та забезпечують можливість створення мінімально життєздатного продукту (*MVP*). Реалізація цієї частини функціоналу є обов'язковою для успішного завершення проєкту або релізу.

Категорія *Must Have* разом із *Should Have* можуть складати приблизно до 80% загального обсягу робіт, що відображає включення важливих, але не критичних функціональних можливостей. Такі вимоги значно підвищують

цінність продукту для користувачів, однак їх відсутність не робить систему повністю нефункціональною.

Решта ресурсів (до 100% загального обсягу робіт) використовується для реалізації вимог категорії *Could Have*, які виконуються лише за наявності додаткового часу, бюджету або інших ресурсів. На рисунку ця частина позначена як резерв (*contingency*), що слугує певним буфером для управління ризиками та змінами під час розроблення. Такий підхід дозволяє забезпечити пріоритетну реалізацію критично важливих функцій і водночас зберегти гнучкість у плануванні проєкту.



1. Зібрати повний список вимог (функціональні, нефункціональні, бізнес-вимоги, покращення, тощо).
2. Залучити ключових стейкхолдерів (Замовник, бізнес-аналітик, продакт-менеджер, представники користувачів, технічні експерти).
3. Обговорити і віднести кожну вимогу до однієї з категорій M/S/C/W.
4. Уточнити, чи не забагато *Must* (якщо майже все *Must*, тоді метод втрачає сенс).
5. Зафіксувати домовленість (результати пріоритизації документуються в специфікації, беклозі або плані релізу та використовуються при плануванні релізів / спринтів: спочатку реалізуються *Must*, потім *Should*, далі *Could*. *Won't* - у backlog на майбутнє).

Функція (вимога)	Категорія
 Реєстрація та вхід користувача	 Must
 Перегляд відео-лекцій	 Must
 Тести для перевірки знань	 Should
 Видача сертифіката після проходження курсу	 Should
 Коментарі під лекціями	 Could
 Мобільний офлайн-доступ до лекцій	 Won't (this time)

Рис.4.7. Приклад розподілу вимог відповідно до категорій M/S/C/W для системи онлайн-курсів

Як висновок: для MVP потрібні: реєстрація + відео, для повнішого релізу додаються тести й сертифікати, коментарі – це покращення на потім, а офлайн-доступ - за межами поточного етапу.



Переваги методу MoSCoW:

- ✓ простий і зрозумілий для Замовника і Виконавця;
- ✓ добре працює в обмежених умовах часу / бюджету;
- ✓ допомагає керувати очікуваннями стейкхолдерів («це точно буде / не буде»);
- ✓ легке використання під час воркшопів, інтерв'ю, зустрічей.



Недоліки методу MoSCoW:

- ✗ без додаткових критеріїв - усе вирішується дискусією;

✗ існує спокуса назвати все *Must*, якщо Замовник не готовий до компромісів;

✗ не дає точного числового рангу всередині категорій (наприклад, який із *Must* важливіший). Тому часто *MoSCoW* комбінують з іншими підходами (наприклад, спочатку *MoSCoW*, а потім усередині категорії застосовують *RICE*, *ICE* чи *Weighted Scoring*).

Модель *Kano*

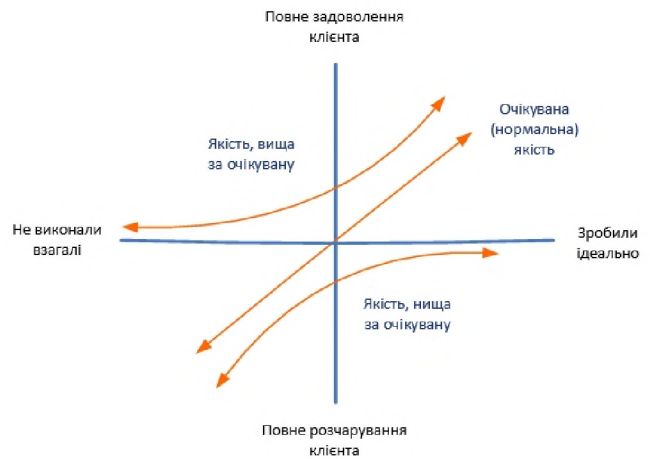
Модель *Kano* - це також якісний метод пріоритезації, який допомагає визначити, як різні функції впливають на задоволення користувачів. Вона не просто сортує функції за важливістю - вона показує емоційний ефект функцій, їхній вплив на сприйняття продукту й лояльність до нього.

Модель *Kano* є системою координат, де на горизонтальній осі відображені потреби клієнта, які можуть бути розділені на три пункти:

- ✓ обов'язкові чи базові функції (*Must-Be*) (якщо немає цієї функції, клієнт не розглядатиме продукт);
- ✓ функції продуктивності (*Satisfaction*) (чим більше інвестуєте в їх розвиток, тим вище буде рівень задоволеності користувача);
- ✓ захоплюючі функції (*Delights*) (користувач на них не очікує, але вони викликають у нього захоплення).

Ідея, яка лежить в основній моделі *Kano* полягає в тому, що чим більше часу, грошей і зусиль витрачається на те, щоб створити, впровадити та покращити функції з кожної категорії, тим вищим буде рівень задоволеності користувачів.

На горизонтальній осі зліва розміщуються потреби, які взагалі незадоволені, праворуч ті, які задоволені повністю. Шкала функціональності показує, наскільки добре реалізована функція, від «не реалізована взагалі» до «повністю реалізована». Вертикальна вісь відображає рівень задоволеності користувача.



В основному, модель *Kano* класифікує характеристики продуктів і послуг на три основні групи: «*Must-Be*», «*Satisfaction*» і «*Delights*» (див.Рис.4.8). Крім того, дві інші групи, «*Indifferent*» (байдужі) та «*Reverse*» (зворотні) - це особливості, які можуть бути розглянуті під час аналізу. Ці п'ять груп функцій засновані на тому, що клієнти кажуть, що важливо і як вони цінують продукти і послуги.

Обов'язкові функції (Must-be) - це основні функції, які характеризують розумний рівень якості, визначений Замовником (якщо цих вимог немає або їх недостатньо, то клієнти будуть незадоволені, наприклад, продукт без дефектів, своєчасне обслуговування клієнтів, лобове скло автомобіля та операційна система на ПК).

Задоволені функції (Satisfaction/Perfomance) - функції, виконання яких призводить до задоволення клієнтів, або навпаки, до невдоволення, коли вони не виконуються (чим вища і краща продуктивність, тим більше клієнт задоволений). Компанії часто використовують такі функції, щоб визначити пріоритети своїх ініціатив з покращення, наприклад, зменшення кількості псувань на виробничій лінії, швидкість відповіді на дзвінок у колл-центрі.

Захоплеві функції (Delights/Attractive) - функції, реалізація яких дасть змогу відрізнити продукт або послугу від ринкових, схожих за функціоналом (вони часто бувають несподіваними, негласними). Компанії повинні прагнути дати клієнтам більше, ніж вони очікують.

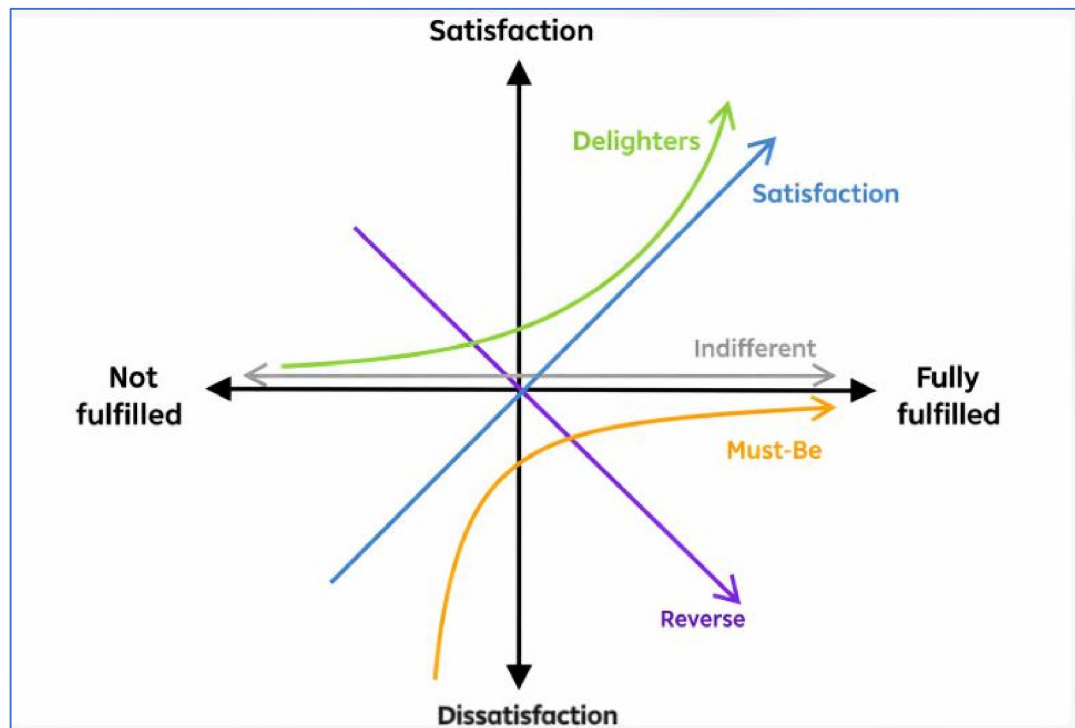


Рис.4.8. Категорії вимог Капо

Байдужі функції (Indifferent) - це функції, які своєю присутністю або відсутністю не впливають на рівень задоволеності клієнтів, наприклад, деякі утиліти в смартфоні або комп'ютері.

Зворотні функції (Reverse) - це функції, які приносять невдоволення, якщо вони присутні, наприклад, додавання занадто багато додаткових функцій може призвести до невдоволення більшості клієнтів.

Для проведення успішного процесу побудови моделі *Kano* необхідна максимальна участь клієнтів, коли це можливо. У випадках, коли безпосередня взаємодія з клієнтами неможлива, доцільне використання таких методів, як пряме спостереження за клієнтами для визначення їх пріоритетів. Якщо пряма взаємодія з клієнтами ускладнена - тоді співпраця з особами, які регулярно взаємодіють з клієнтами, такими як фахівці з продажу та представники обслуговування клієнтів, щоб генерувати необхідну інформацію.



Для того, щоб отримати від клієнтів інформацію про їхні потреби, необхідно скласти опитувальник *Kano*, заповнюючи який вони повинні оцінити кожну функцію у двох формах:

1. функціональне питання: що ви відчуєте, якщо функція є?
2. дисфункціональне питання: що ви відчуєте, якщо функції немає?

Варіанти відповідей (5 класичних):

- ✓ Мені це подобається (*Like it*)
- ✓ Я очікую це (*Expect it*)
- ✓ Мені все рівно (*Don't care*)
- ✓ Я можу з цим жити (*Live with*)
- ✓ Мені це не подобається (*Dislike*)

Приклад такого опитувальника наведений на рисунку 4.9.



Рис. 4.9. Приклад опитувальника

Поєднання відповідей визначає категорію вимог за таблицею *Kano*.

		Dysfunctional (Feature absent)				
		Like it	Expect it	Don't care	Live with	Dislike
Functional (Feature present)	Like it	Questionable	Attractive	Attractive	Attractive	Performance
	Expect it	Reverse	Questionable	Indifferent	Indifferent	Must-be
	Don't care	Reverse	Indifferent	Indifferent	Indifferent	Must-be
	Live with	Reverse	Indifferent	Indifferent	Questionable	Must-be
	Dislike	Reverse	Reverse	Reverse	Reverse	Questionable

Рис. 4.10. Таблиця *Kano*

Таблиця на рисунку 4.10 - це ядро моделі *Kano*. Вона показує, як на основі опитування можна визначити тип вимоги, а саме що користувач очікує, що його радує, що не має значення, що дратує, а що викликає протиріччя. Принцип роботи таблиці наступний: кожна клітинка показує, що означає певне поєднання реакцій користувача на присутність або відсутність функції, наприклад:

✓ *Attractive (Delighters)*: з'являються, коли користувач не очікує функцію або байдужий до її відсутності, але йому подобається, коли вона є. Це «вау-функції».

✓ *Performance (Satisfiers)*: з'являються, коли користувачу подобається наявність функції і не подобається, коли її немає (чим краще виконана функція, тим більше задоволення).

✓ *Must-be*: з'являються, коли користувач очікує функцію і негативно реагує, якщо її немає (обов'язкові вимоги).

✓ *Indifferent*: з'являються, коли користувачу байдуже чи функція є, чи ні (не впливає на задоволення потреб користувача).

✓ *Reverse*: з'являються, коли користувач не любить, коли функція є, і йому подобається, коли її немає (функція більше шкодить, ніж допомагає).

✓ *Questionable*: з'являються, коли відповіді суперечливі (наприклад, користувач одночасно «любить» і «не любить»). В такому випадку, можливо, користувач не зрозумів питання або сталася помилка.



Переваги моделі *Kano*:

- ✓ орієнтація не просто на логіку, а на емоції користувача;
- ✓ чітке відокремлення «очікуваного» від «захопливого»;
- ✓ добре працює в UX і продуктовому дизайні.



Недоліки моделі *Kano*:

- ✗ потребує правильно сформульованих питань;
- ✗ вимагає аналізу великої кількості відповідей;
- ✗ результати залежать від якості вибірки користувачів.

Метод парного порівняння *Pairwise Comparison*

Pairwise Comparison (парне порівняння) - це метод пріоритезації, у якому кожна вимога порівнюється з кожною іншою в парі, щоб визначити, яка з них важливіша. Після завершення порівнянь формується ранг вимог за числом «перемог».

Основна ідея полягає в наступному: якщо складно оцінити всі вимоги одразу, то їх простіше оцінювати парами: «Що важливіше - А чи В?» Кожна «перемога» фіксується, після чого підраховується загальна кількість перемог кожної вимоги.

Парне порівняння застосовується у випадках, коли:

- ✓ потрібно пріоритизувати 5–15 вимог;
- ✓ не вистачає даних для кількісних методів;
- ✓ потрібно швидко узгодити погляди в команді;
- ✓ важливо уникнути нав'язування суб'єктивних виборів однієї людини;
- ✓ у воркшопах зі стейкхолдерами, користувачами, Замовником.

Покрокова
реалізація
методу

1. Скласти список функцій (вимог), наприклад: А – «*User Authentication*», В – «*Intelligent Search*», С – «*Advanced Analytics*» тощо.

2. Побудувати матрицю порівнянь, а саме квадратну таблицю, де на обох осях перелік вимог.

3. Провести попарне порівняння. Для кожної пари ставиться запитання: «Що важливіше для користувача/бізнесу: функція А чи функція В?». Вибір може бути:

- ✓ А важливіша → ставимо А
- ✓ В важливіша → ставимо В

	A	B	C	D	E	F	G	H
A: <i>User Authentication</i>		A	A	A	A	A	A	A
B: <i>Intelligent Search</i>			B	D	E	B	B	B
C: <i>Advanced Analytics</i>				D	E	F	G	C
D: <i>Real-time Notifications</i>					E	D	D	D
E: <i>Workflow Automation</i>						E	E	E
F: <i>Dashboard Customization</i>							G	G
G: <i>Enhanced Security</i>								G
H: <i>Data Export</i>								
Count	7	4	1	5	6	1	4	0
Rank	1	4	5	3	2	5	4	6

4. Сформувати упорядкований список пріоритетів відповідно до підсумкового пріоритету реалізації функції в системі (*Rank*), який базується на кількості «перемог» (*Count*) функції при порівнянні.



Переваги *Pairwise Comparison*:

- ✓ зменшує складність (замість вибору з 10 елементів, порівнюється тільки 2);
- ✓ добре працює при невеликій кількості вимог;
- ✓ дає об'єктивніший результат, ніж інтуїтивне оцінювання;
- ✓ стимулює обговорення та пояснення виборів.



Недоліки *Pairwise Comparison*:

- ✗ залежить від контексту (без критеріїв рішення можуть бути суб'єктивними);
- ✗ не враховує взаємозалежності між вимогами;
- ✗ складно застосовувати в дуже великих проєктах.

Метод парного порівняння - ефективний, логічний та прозорий спосіб пріоритизації, коли елементів небагато, даних мало, потрібно швидко узгодити пріоритети та важливо виявити реальні переваги між вимогами.

Метод *Delphi*

Delphi - це ітераційний експертний метод, який застосовується для пріоритизації вимог, прогнозування та ухвалення рішень у ситуаціях невизначеності з метою отримати обґрунтований консенсус експертів без впливу авторитетів, тиску групи чи домінування окремих учасників.



Delphi базується на принципах:

- ✓ багатостадійності оцінки (кілька раундів);
- ✓ анонімності експертів (незнання хто є експертом виключає вплив на їх оцінку);
- ✓ керованого зворотного зв'язку (експерти бачать узагальнені відповіді групи);
- ✓ досягнення консенсусу (розбіжності зменшуються після кожного раунду).

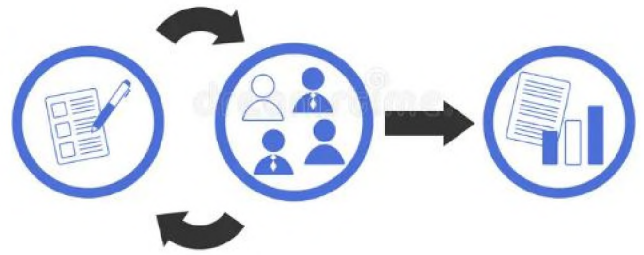
У результаті отримують більш точне та узгоджене рішення, ніж при відкритих дискусіях.

Покрокова реалізація методу

1. *Вибір експертів.* Обирають групу (7–20 осіб) із доменними знаннями, досвідом роботи з аналогічними продуктами або технічними компетенціями.
2. *Формування першого опитувальника.* У ньому експертам пропонують оцінити вимоги, визначити їхню важливість, надати коментарі чи обґрунтування. Оцінювання може бути у формі шкали (від 1 до 5, від 1 до 10), ранжування або категорій (високий, середній, низький пріоритет).

3. *Перший раунд оцінки.*

Експерти відповідають анонімно й незалежно, а потім результати їх оцінювання збираються та агрегуються.



4. *Аналіз і зворотний зв'язок.*

Організатор (модератор) готує зведення (анонімне) про середні оцінки, діапазон значень, коментарі експертів та обґрунтування високих або низьких оцінок.

5. *Другий раунд.* Експерти отримують зведення і можуть переглянути свої попередні відповіді, змінити оцінку, обґрунтувати її або залишити без змін. Часто після повторного перегляду оцінки зближуються.

6. *Повторення раундів.* Проводять 2–4 раунди, поки розкид оцінок не зменшиться, не буде досягнуто стабільності або експерти не дійдуть консенсусу.

7. *Формування остаточного рейтингу.* Після завершення усіх раундів створюється підсумкова таблиця пріоритетів, в яку вноситься рейтинг вимог, середні значення, узгодженість оцінок.



Переваги методу *Delphi*:

- ✓ усуває груповий тиск (експерти не знають, хто дав яку оцінку);
- ✓ дозволяє об'єднати знання різних спеціалістів;
- ✓ формує аргументовані рішення (коментарі експертів підсилюють результат);
- ✓ може працювати онлайн (багато інструментів підтримують анонімні опитування);
- ✓ ідеальний для складних систем (де кількість факторів велика, а даних мало).



Недоліки методу *Delphi*:

- ✗ метод може займати багато часу (до кількох тижнів);
- ✗ потребує уважної модерації;
- ✗ вимагає наявності кваліфікованих експертів;
- ✗ не підходить для «швидких» рішень.

Метод групового ранжування *Group Ranking*

Group Ranking – ще один метод якісної пріоритезації, у якому група учасників спільно формує ранжований список вимог, впорядковуючи їх від найважливіших до найменш важливих. Це інтерактивний, координований, комунікаційний метод, який дозволяє врахувати думку всієї команди або стейкхолдерів. Його головна цінність полягає у досягненні спільного рішення через структуруване обговорення.

Основна ідея методу полягає в тому, що група (аналітики, Замовник, користувачі, розробники, менеджери) разом ранжує список вимог, обговорюючи та аргументуючи вибір, поки не буде досягнуто узгодженого порядку пріоритетів. На відміну від індивідуальних методів, *Group Ranking* ґрунтується на колективному консенсусі.



Покрокова реалізація методу

1. *Формування списку вимог.* Учасники отримують попередній список вимог (це можуть бути картки, нотатки, таблиця).

2. *Індивідуальне попереднє оцінювання.* Кожен учасник, читаючи опис



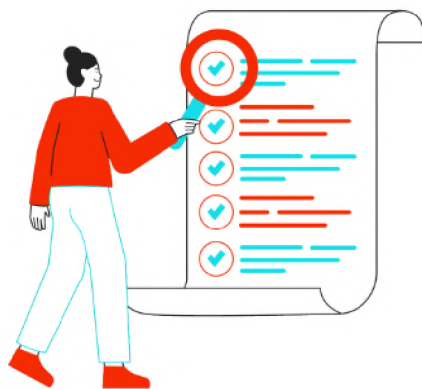
вимог, самостійно визначає свій порядок важливості та готує аргументи свого рішення (це зменшує вплив групи на початкові думки).

3. *Спільне обговорення.* Координатор пропонує обговорити вимоги по черзі, починаючи з тих, що викликають суперечки, фокусуючись на причинно-наслідкових аргументах.

4. *Побудова початкового ранжування.* Група створює першу версію впорядкованого списку, наприклад, розміщуючи вимоги вертикально чи за допомогою *drag-and-drop* у *Miro/Mural* (взаємодія з елементами на цифровій дошці, коли користувач може перетягувати об'єкти мишею з одного місця на інше).

5. *Перегляд і коригування.* Обговорення триває, доки учасники не погодяться з порядком або поки не буде знайдено компроміс, при цьому координатор фіксує причини, чому деякі вимоги рухаються вгору або вниз.

6. *Досягнення консенсусу.* Група приходить до узгодженого рішення (якщо суперечки не знімаються, тоді додатково ще використовують голосування, додаткові аргументи або правило 70/30 (більшість приймає рішення)).



7. *Документування результатів.* Підготовка фінального списку:

1 → найважливіше

2 → важливе

3 → середній пріоритет

...

N → найменш критичне

Також фіксуються логіка прийняття рішень, ризики, коментарі учасників, залежності між вимогами.

Для зручності ранжування можна використовувати вертикальний список (з можливістю перетягування карток), ранжувальну лінійку (на якій розкладаються вимоги), матрицю пріоритетів (якщо потрібно більше структури) або фізичні картки у живому воркшопі (*sticky notes* на дошці). Це допомагає команді бачити загальну картину.



Переваги *Group Ranking*:

- ✓ *прозорість* прийняття рішень (усі бачать, як формується пріоритет);
- ✓ *висока залученість* стейкхолдерів (рішення не нав'язане, воно колективне);
- ✓ врахування *різних точок зору* (аналітики, користувачі, бізнес і технічні фахівці впливають на результат);
- ✓ можна провести за 1–2 години на воркшопі;
- ✓ *зменшення ризику конфліктів*, оскільки рішення приймається спільно.



Недоліки *Group Ranking*:

- ✗ може виникати ефект домінування (сильніші учасники впливають на групу);
- ✗ не працює, якщо стейкхолдери не готові до компромісів;
- ✗ вимагає хорошого координатора;
- ✗ не завжди підходить для великих списків вимог.

Метод експертного судження *Expert Judgment*

Expert Judgment - це один із найшвидших методів пріоритизації, який дозволяє ухвалити рішення в умовах невизначеності чи нестачі даних, це метод пріоритизації, у якому рішення про важливість вимог приймаються на основі знань, досвіду та професійних оцінок експертів. Експертами можуть бути продакт-менеджери, бізнес-аналітики, системні аналітики, архітектори, технічні лідери, доменні фахівці, стейкхолдери.



Метод *Expert Judgment* ґрунтується на припущенні, що «люди з достатнім досвідом, компетентністю та доменними знаннями можуть робити якісні та обґрунтовані оцінки важливості вимог». Експерти аналізують вимоги та визначають

їхній пріоритет, керуючись:

- ✓ знанням бізнес-контексту;
- ✓ оцінкою впливу на користувача;
- ✓ розумінням технологічної складності;
- ✓ досвідом роботи в аналогічних проєктах;
- ✓ стратегічними цілями організації.

Покрокова
реалізація
методу

1. *Визначення експертів.* Обираються найбільш компетентні люди: бізнес-аналітик, менеджер проєкту, архітектор рішень, власник продукту, експерти з предметної області.

2. *Ознайомлення експертів із вимогами.* Експертам надають список вимог, контекст, обмеження, бізнес-цілі.

3. *Індивідуальна або колективна оцінка*. Експерти визначають пріоритети усно під час зустрічі або письмово у документі. *Expert Judgment* може застосовуватися в різних формах:



1) просте призначення пріоритетів (експерт визначає: *High/Medium/Low*);

2) ранжування (експерт впорядковує вимоги: 1, 2, 3, ...);

3) вага вимог (наприклад, бізнес-цінність, ризик, технічна критичність);

4) комбіновані методи (*Expert Judgment* часто використовується як надбудова над іншими підходами, такими як *RICE*, *MoSCoW*, *Kano*, *WSJF*).

4. *Узгодження* (за потреби). Якщо експертів кілька, їхні оцінки можуть бути усереднені, обговорені, зіставлені на нараді.

5. *Документування*. Фіксується фінальний пріоритет, хто надав оцінку, ключові аргументи та умови, за яких прийнято рішення.



Переваги *Expert Judgment*:

✓ *дуже швидкий* (можна отримати результат за годину чи навіть хвилини);

✓ *експертність* → *якість рішення* (рішення базується на глибокому доменному знанні);

✓ *працює при нестачі даних* (коли немає аналітики, метрик, статистики);

✓ *гнучкий* (підходить і для малих стартапів, і для великих проєктів);

✓ *ефективний у критичних сферах*.



Недоліки *Expert Judgment*:

- ✗ суб'єктивність (рішення залежить від особистої думки експерта);
- ✗ ризик упередженості (підтвердження власної думки, вплив попереднього досвіду);
- ✗ домінування авторитетів (може придушити альтернативні сценарії);
- ✗ відсутність прозорої формули (не завжди зрозуміло, чому вибрано саме ту вимогу).

Expert Judgment - це один із найгнучкіших та найпрактичніших методів пріоритезації вимог. Його цінність полягає в тому, що досвідчені фахівці здатні ухвалювати якісні рішення навіть за відсутності даних, але метод потребує обережного використання, щоб уникнути суб'єктивності та залежності від думки окремих людей.

Ідеальний метод пріоритезації повинен робити кілька обов'язкових речей:

- ✓ повинен залучати кожного члена команди, стейкхолдера, а в деяких випадках і клієнтів;
- ✓ має принести практичні результати та надати інформацію, яка допоможе покращити продуктову стратегію;
- ✓ повинен спонукати членів команди думати про те, як будь-яка ідея впливає на цілі компанії та визначати пріоритети, виходячи з цього;
- ✓ повинен позбавити від рою безглузвих і марних ідей.

Підсумкова порівняльна характеристика якісних методів пріоритезації наведена у таблиці 4.4.

Жодні методи пріоритезації не можуть замінити прийняття рішень людиною. Хороші фреймворки - це спосіб допомогти всім домовитися про

терміни або цілі продукту, це вправи, щоб зуміти прийняти якомога більше точних рішень щодо кожної фічі чи ідеї.

Таблиця 4.4 - Характеристики якісних методів пріоритизації вимог

Метод	Основна ідея	Переваги	Недоліки	Коли найкраще застосовувати
<i>MoSCoW</i>	Категоризація <i>Must/Should/Could/Won't</i>	Простота, швидкість	Суб'єктивний	<i>MVP</i> , перші релізи
<i>Kano</i>	Задоволення користувача	<i>UX</i> -фокус, виявлення «вау»-ініціатив	Потрібні опитування	<i>UX</i> , продуктове дослідження
<i>Pairwise</i>	Порівняння кожної пари вимог	Прозорість рейтингу	НЕ для великих списків вимог	Невеликі проекти, воркшопи
<i>Delphi</i>	Анонімний експертний консенсус	Об'єктивність, глибина	Тривалий процес	Великі складні системи
<i>Group Ranking</i>	Спільне групове ранжування	Залученість, швидкість	Ризик домінування	Командна фасилітація
<i>Expert Judgment</i>	Рішення експертів	Швидкість, ефективність	Суб'єктивний	Старт проекту, технічні домени

Вибір методу пріоритизації - ітеративний процес. Тільки пробуючи різні підходи, можна отримати зворотний зв'язок, виміряти ефективність і знайти той, який відповідає заданим вимогам.

Питання для самоперевірки



1. Що таке кількісні методи пріоритизації та в чому їхня основна перевага?
2. Які складові входять у формулу *RICE*?
3. У чому ключова різниця між *RICE* та *ICE*?
4. Чому *ICE* вважається швидким методом пріоритизації?
5. Що означає *Ease* у методі *ICE*?
6. Що таке *Weighted Scoring* і в чому його гнучкість?
7. Як обираються ваги критеріїв у *Weighted Scoring*?

8. Чим небезпечний неправильний підбір вагових коефіцієнтів?
9. Які квадранти містить матриця *Value vs Effort*?
10. Чому метод *Value vs Effort* є візуальним і простим?
11. Що таке *WSJF* і для чого він використовується?
12. Як обчислюється *Cost of Delay*?
13. Чим *Job Size* відрізняється від *Effort*?
14. У чому полягає зміст *100-Dollar Test*?
15. Чим відрізняється *Buy a Feature* від *100-Dollar Test*?
16. У чому полягає принцип якісних методів пріоритезації?
17. Які категорії входять у *MoSCoW*?
18. Чим *Must* відрізняється від *Should*?
19. Коли категорія *Won't* є корисною та чому?
20. У чому полягає основна ідея моделі *Kano*?
21. Які типи вимог виділяє *Kano*?
22. Чим відрізняються *Performance* та *Delighters* у *Kano*?
23. Як проводиться опитування для моделі *Kano*?
24. Коли модель *Kano* приносить найбільшу користь?
25. Що таке *Pairwise Comparison*?
26. Чому метод *Pairwise* не ефективний для великих списків?
27. Який зміст методу *Delphi* у пріоритезації? У чому перевага анонімності?
28. Чому *Delphi* застосовують у складних проєктах?
29. Скільки раундів зазвичай потрібно у *Delphi*?
30. У чому суть методу *Group Ranking*? Чому його часто використовують у воркшопах?
31. Де саме може виникнути упередженість у *Group Ranking*?
32. Що таке *Expert Judgment*? Чому даний метод може бути суб'єктивним?
33. Який якісний метод найкраще підходить для визначення *MVP* і чому?

Тема 5

РОЛЬОВА МОДЕЛЬ КОРИСТУВАЧІВ: *User Story, Backlog*

5.1. Що таке історії користувача та їх роль в процесі аналізу вимог

Користувацькі історії та варіанти використання є різними інструментами документування вимог, які використовуються в процесі розробки програмного забезпечення. Кожен із них має свою функцію. Історії користувачів використовуються для визначення функціональних вимог до системи з погляду користувача та мовою бізнесу, а варіанти використання - для більш детального опису взаємодії користувача із системою у визначених сценаріях.

Історії користувача є одним із способів представлення функціональних вимог до програмної системи у простій і зрозумілій формі. Вони не містять детального технічного опису реалізації, а концентруються на потребах користувача та очікуваному результаті взаємодії із системою. Завдяки цьому історії користувача сприяють кращому розумінню вимог між усіма зацікавленими сторонами: Замовниками, аналітиками, розробниками та тестувальниками.

У процесі аналізу вимог історії користувача виконують кілька важливих функцій. *По-перше*, вони допомагають виявляти та формалізувати потреби користувачів, оскільки орієнтовані саме на їхні цілі та очікування. *По-друге*, вони забезпечують комунікацію між учасниками проєкту, оскільки формуються зрозумілою мовою без складної технічної термінології. *По-третє*, історії користувача використовуються для пріоритизації вимог і



формування переліку робіт у процесі планування розроблення програмного забезпечення. Крім того, кожна історія користувача зазвичай доповнюється критеріями приймання, що дозволяє уточнити умови виконання вимоги та полегшує її перевірку під час тестування.

Після формування та уточнення історій користувача наступним важливим етапом є їх використання для планування і організації роботи команди розробників. Історії користувача, що відображають потреби користувачів та очікувану функціональність системи, упорядковуються, оцінюються, пріоритизуються та деталізуються відповідно до бізнес-цінності й складності реалізації. Подальша їх реалізація відбувається в межах ітераційного процесу розроблення програмного забезпечення, який організовується у вигляді спринтів у межах фреймворку *Scrum*.

Спринт (Sprint) - це міні-проект усередині великого проекту, який має чітку тривалість, свої цілі, конкретний запланований обсяг роботи та завершений результат. Це короткий, фіксований проміжок часу (найчастіше 1–4 тижні), протягом якого команда в *Agile/Scrum* планує, виконує і доставляє готовий, потенційно релізний інкремент продукту. Головне правило: тривалість спринту завжди однакова у всьому проекті.



1. *Планування спринту (Sprint Planning Meeting)*. Під час планування спринту команда обирає з переліку робіт найбільш пріоритетні історії користувача, які можуть бути реалізовані протягом визначеного часового інтервалу. Обрані історії формують перелік завдань, що підлягають виконанню протягом конкретного спринту (*Sprint Backlog*).

2. *Робота над задачами* - команда виконує користувацькі історії, задачі, тестує та усуває баги, інтегрує.

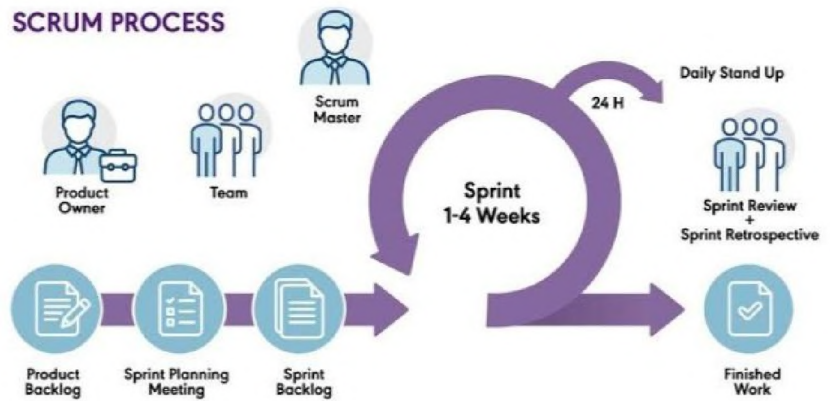
3. *Щоденні стендапи (Daily Stand Up)* -

це короткі щоденні зустрічі команди розробників, що проводяться для синхронізації роботи, обговорення поточного прогресу та виявлення можливих перешкод у виконанні завдань. Щоденний стендап зазвичай проводиться щодня у той самий час протягом спринту і має тривати не більше 15 хвилин. Основна мета цієї зустрічі полягає у координації дій членів команди та забезпеченні прозорості процесу виконання завдань. Під час стендапу учасники команди коротко інформують один одного про стан виконання робіт і можливі труднощі, а саме:

- ✓ що було зроблено з моменту попередньої зустрічі?
- ✓ що планується зробити до наступної зустрічі?
- ✓ які перешкоди або проблеми виникають під час виконання завдань?

Завдяки щоденним стендапам команда отримує можливість оперативно реагувати на проблеми, коригувати план робіт та підтримувати спільне розуміння поточного стану розробки продукту.

4. *Огляд спринту (Sprint Review)* – зустріч, яка проводиться наприкінці спринту, під час якої команда розробників демонструє результати роботи, виконаної протягом спринту, та обговорює їх із зацікавленими сторонами. Основною метою огляду спринту є представлення інкременту продукту (*Product Increment*), тобто результату роботи команди розробників, отриманий у межах одного спринту, який являє собою нову або вдосконалену функціональність



програмного продукту, що відповідає встановленим вимогам і може бути використана або продемонстрована зацікавленим сторонам. Під час цієї зустрічі команда демонструє працюючу частину системи, що дозволяє Замовникам і іншим зацікавленим сторонам оцінити досягнутий результат та надати зворотний зв'язок.

5. *Ретроспектива (Sprint Retrospective)* - це зустріч команди, яка також проводиться після завершення спринту з метою аналізу процесу роботи, виявлення проблем та визначення можливостей для його покращення. На відміну від огляду спринту, під час якого оцінюється створений інкремент продукту, ретроспектива зосереджується саме на процесі виконання роботи. Учасники команди аналізують, наскільки ефективною була організація роботи протягом спринту, які фактори сприяли успішному виконанню завдань і які труднощі виникали.

6. *Завершення спринту* - це етап підбиття підсумків роботи команди після завершення ітерації розроблення. На цьому етапі команда аналізує, які історії користувачів були успішно реалізовані та відповідають встановленим критеріям прийняття. Такі історії вважаються завершеними і формують інкремент продукту, який може бути представлений зацікавленим сторонам або підготовлений до подальшого використання.

Робота з історіями користувачів має кілька доведених переваг, тому багато гнучких команд у всьому світі використовують їх. Ось деякі з ключових причин, чому гнучкі команди надають пріоритет історіям користувачів:

✓ *поділ роботи на керовані завдання.* Однією з головних переваг історій користувачів є розбиття амбітних і величезних обсягів продуктів на менші, більш досяжні набори функцій. Історії користувачів - це просто короткий виклад цілей користувача їхніми власними словами. Вони

допомагають групам уникнути заплутування в деталях і протоколах, а також у технічних аспектах і процедурах.

✓ *організація спринта.* Історії користувачів допомагають командам визначити пріоритети, тобто які саме функції створювати в першу чергу.

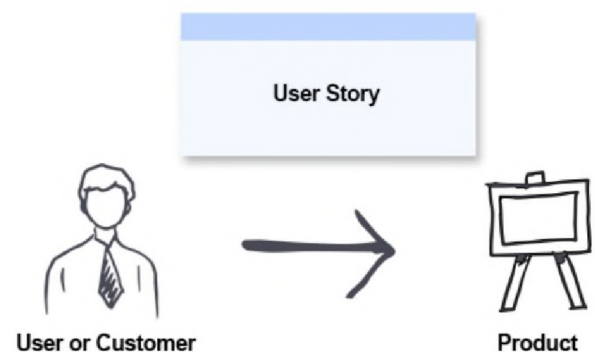
✓ *надають критерії тестування.* Історії користувачів дають

гнучким командам конкретні критерії для перевірки, оскільки кожна історія повинна мати чіткий, демонстрований результат, який гарантує, що історія користувача відповідає бажаному результату. Це допомагає командам ефективніше вимірювати продуктивність і результати під час кожного спринту.

✓ *зменшення технічного жаргону.* Відсутність в історіях користувачів технічного жаргону сприяє більш творчому та гнучкому діалогу між командами розробників і дизайнерів. За допомогою історій користувачів акцент зміщується з функцій і технічних деталей на точку зору користувача та потенційні можливості використання продукту. Нетехнічні зацікавлені сторони, які бажають знати, над якими функціями працюється, також вважають цей метод дуже корисним.

✓ *втілення в життя осіб користувачів через історії користувачів.*

Історії користувачів можуть втілити в життя особи користувачів, залучаючи до проекту цих вигаданих, але реальних персонажів. Це схоже на те, що представники тисяч або мільйонів користувачів направляють команду



розробників щодо того, що їм потрібно, а не команда просто уявляє це. Історії

користувачів можна поєднувати, щоб сформувати карти історій користувачів, продуктивну гнучку техніку, у якій усі історії організовуються для створення графіка розробки та списку пріоритетів.

Макет User Story

Історія користувача (User Story) - це короткий опис того, що повинен робити продукт, для кого він призначений і навіщо він потрібен. Зазвичай користувацька історія складається з одного або двох речень і має простий формат, щоб її легко було зрозуміти та реалізувати. На рисунку 5.1. зображена структура побудови *User Story*.

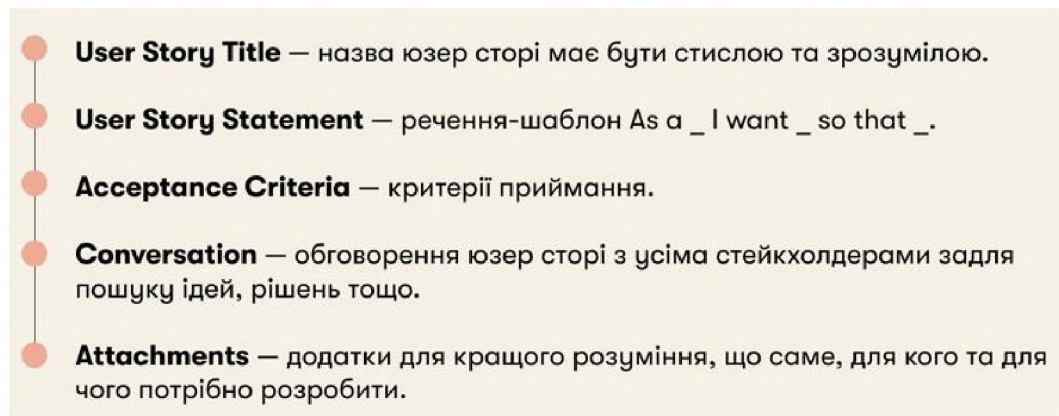
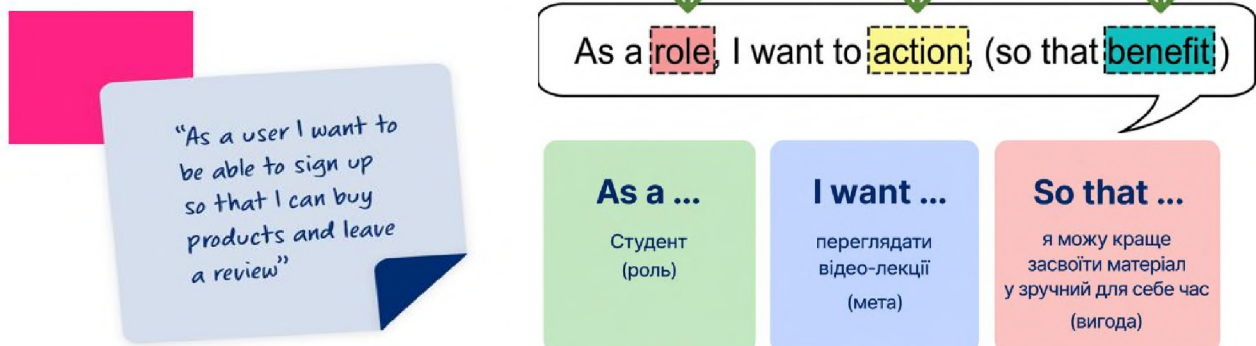


Рис.5.1. Структура User Story

Те, як гнучкі команди пишуть історії користувачів, часто різняться в різних організаціях і командах. Щоб успішно писати історії користувачів, завжди необхідно брати до уваги наступні елементи: *Користувач • Дія • Вигода*, щоб надати своїй команді повну картину того, що саме потрібно на кожному етапі розробки продукту.



За допомогою цих трьох елементів можна написати коротку історію користувача щодо функції, яка потрібна, і чому вона потрібна. Зазвичай історію користувача пишуть за такою структурою:

Крок 1 - Як... (*As a...*). Перший крок під час написання історії користувача полягає у визначенні та формулюванні ролі користувача. Цей етап задає загальний контекст, зосереджуючи увагу на тому, хто саме отримуватиме користь від реалізованої функціональності. Важливо чітко визначити тип користувача - це може бути звичайний клієнт, внутрішній працівник організації або інша зацікавлена сторона.

Крок 2 - Я хочу... (*I want...*). Другий крок розкриває основну сутність історії - потребу або бажання користувача. У цій частині описується, що саме користувач хоче зробити або якої можливості очікує від системи. Важливо зосереджуватися саме на тому, що користувач хоче отримати, а не на тому, яким чином це має бути реалізовано.

Крок 3 - Щоб... (*So that...*). Останній крок уточнює мету та цінність історії користувача. Він пояснює, чому потреба користувача є важливою і який результат або користь очікується від реалізації відповідної функціональності. Ця частина допомагає забезпечити, щоб кожна розроблена функція мала чітке обґрунтування та створювала конкретну цінність для користувачів, узгоджуючись із загальними цілями проєкту.

Після формулювання історії користувача в ній слід прописати ще так звані критерії прийняття (детальніше розглянемо нижче), у яких слід перелічити технічні характеристики, необхідні для забезпечення задоволення потреб користувача.

Для порівняння, у таблиці 5.1 наведено приклади правильно і неправильно сформульованих *User Stories*.

Таблиця 5.1 - Правильно і неправильно сформульовані User Stories

Правильна User Story	Неправильна User Story (з поясненням чому?)
Як студент, я хочу бачити результат тесту, щоб оцінити свій рівень знань.	Зробити форму результатів (немає користувача, потреби та цінності).
Як викладач, я хочу редагувати питання тесту, щоб підтримувати їх актуальність.	Система повинна дозволяти редагувати дані в БД (технічна вимога, не історія користувача).
Як користувач, я хочу відновити пароль через email, щоб увійти в акаунт.	Зробити логіку відновлення пароля (орієнтовано на розробника, а не на користувача).
Як адміністратор, я хочу переглядати список користувачів, щоб керувати доступами.	Вивести таблицю всіх користувачів (не зрозуміло, кому і навіщо це потрібно).
Як покупець, я хочу додавати товари до кошика, щоб оформити замовлення пізніше.	Кнопка «Додати» має працювати (описує елемент UI, а не цінність для користувача).
Як водій, я хочу отримувати попередження про ожеледицю, щоб уникати небезпеки.	Додати сповіщення на панель (немає контексту користувача та цілі).
Як новий користувач, я хочу пройти швидку реєстрацію, щоб почати користуватися сервісом.	Реалізувати реєстрацію за 3 кроки (це рішення, а не потреба користувача).

5.2. Backlog та декомпозиція User Stories

Беклог (Backlog) - це впорядкований список усього, що потрібно зробити в продукті, починаючи від великих функцій і завершуючи дрібними завданнями. У ньому зібрані ідеї, вимоги, задачі, покращення, баги - усе, що може принести цінність продукту.

Перелік робіт продукту (Product Backlog) - це впорядкований список усіх вимог, функцій, покращень і виправлень, необхідних для створення та подальшого розвитку програмного продукту (див.Рис.5.2). Цей перелік є основним джерелом завдань для команди розробників і постійно уточнюється та оновлюється протягом усього життєвого циклу продукту у фреймворку *Scrum*.

До переліку робіт продукту включаються різні типи елементів, зокрема це історії користувача, функціональні та нефункціональні вимоги, технічні завдання, а також виправлення помилок і пропозиції щодо вдосконалення

системи. Усі елементи у переліку зазвичай впорядковуються за пріоритетністю, що визначається з урахуванням бізнес-цінності, потреб користувачів та складності реалізації.

Відповідальність за формування, підтримку та актуалізацію *Product Backlog* покладається на власника продукту, який визначає пріоритети елементів, уточнює вимоги та забезпечує їх готовність до реалізації під час наступних спринтів.

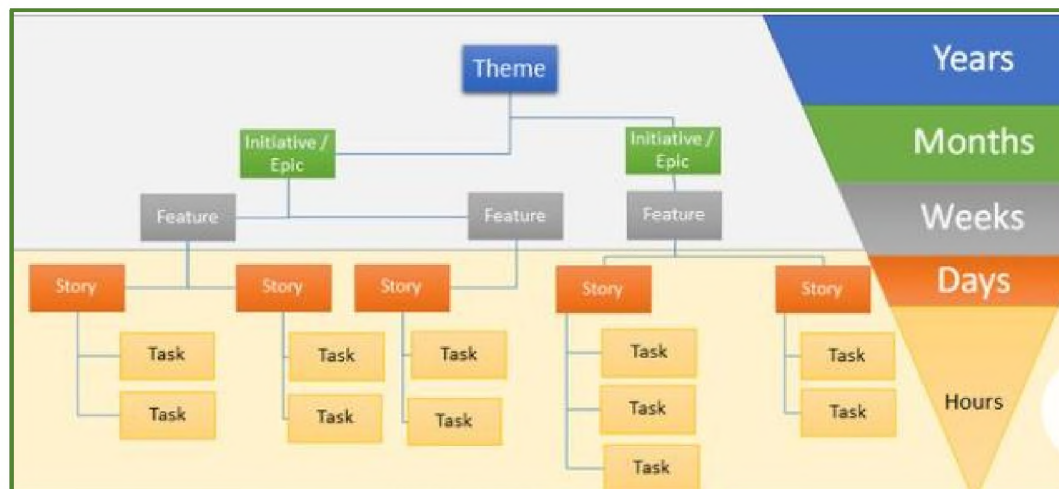


Рис. 5.2. Структура Product Backlog

На рисунку 5.2. представлені типи елементів та їх зв'язки, які використовуються для структурування беклогу продукту:

- ✓ *Theme* - відображає глобальну стратегічну ціль продукту, напрям розвитку або велику бізнес-ініціативу, яка триває роками і може включати в себе багато епіків.

- ✓ *Initiative / Epic* - великі блоки роботи, що представляють собою значний функціональний модуль або бізнес-напрямок і їх реалізація зазвичай триває кілька місяців. Епік (*Epic*) декомпозується на функції (*Features*).

- ✓ *Feature* - конкретна функціональність, що приносить цінність користувачу або бізнесу і її реалізація зазвичай займає кілька тижнів. Функція декомпозується на історії користувача (*User Stories*).

✓ *Story (User Story)* - невелика одиниця вимоги з конкретною цінністю користувачу, тривалістю виконання від 1 до кількох днів. Історії користувача (*User Stories*) декомпозуються на завдання (*Tasks*).

✓ *Task* - це конкретний технічний крок, який виконує розробник, дизайнер або тестувальник за години або максимум за один день. *Tasks* не обов'язково несуть цінність користувачу напямую, оскільки це внутрішні роботи.

Перелік робіт спринту (*Sprint Backlog*) - це вибрана частина *Product Backlog*, яку команда зобов'язується реалізувати протягом одного конкретного спринту. Він містить відібрані історії користувача (або інші елементи беклогу), їх деталізовані задачі, план роботи команди на спринт.

Перелік робіт конкретної команди (*Team Backlog*) - це впорядкований список роботи однієї конкретної команди у масштабованих *Agile*-фреймворках, таких як *SAFe*. Він містить усі елементи, за які відповідає саме ця команда, включно з *User Stories*, технічними завданнями, багами (*bugs*), покращеннями, технічним боргом та *research*-задачами. На відміну від *Product Backlog*, який охоплює весь продукт, *Team Backlog* відображає роботу тільки однієї команди, відповідно до її зони відповідальності та компетенцій.

У *Backlog* досить часто потрапляють занадто великі *User stories* або взагалі *Epics*.

Декомпозиція *User Stories* - це розбиття великих запитів користувачів на менші, зрозумілі й реалізовані шматки роботи, які команда може зробити за один спринт.

Схема на рисунку 5.3 демонструє, як велика потреба (*User Need*) поступово перетворюється на конкретні задачі для спринтів, оскільки потреба користувача є широким, стратегічним поняттям, що описує загальну мотивацію користувача.

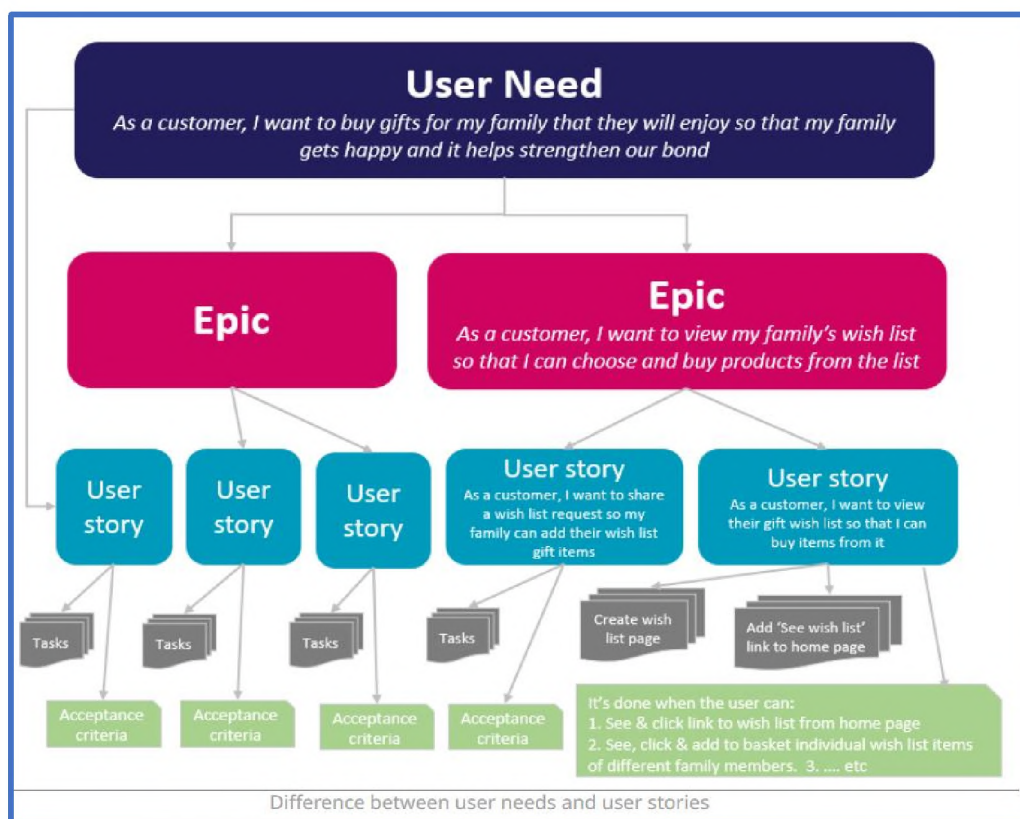


Рис. 5.3. Схема декомпозиції потреб користувача

Декомпозиція - це процес розбиття *Epic* → на *User Stories*, а *User Story* → на менші *User Stories* або *Tasks*. Головна ціль - отримати такі користувацькі історії, які можна реалізувати за один спринт і які все ще несуть видиму цінність користувачу.

Виділяють декілька причин, навіщо проводити декомпозицію:

1. керований обсяг роботи (занадто велика історія користувача породжує ризики, зриви спринтів);
2. точніше оцінювання (маленькі історії простіше оцінювати (сторі-поінти, час);
3. прозорість прогресу (видно, що саме зроблено й що ще в процесі);
4. краще планування спринтів (легше заповнити спринт адекватним набором історій);

5. раннє отримання цінності (вже за один-два спринти можна дати користувачу частину корисного функціоналу).

Стори-поїнти (Story Points) - це умовна одиниця вимірювання, що використовується для оцінювання складності, обсягу роботи та невизначеності реалізації історій користувача. На відміну від оцінювання у годинах або днях, *Story Points* не відображають точний час виконання завдання, а характеризують відносну складність роботи, яку необхідно виконати для реалізації певної історії користувача. При оцінюванні враховуються кілька факторів: обсяг роботи, технічна складність, рівень невизначеності та потенційні ризики.

Але як зрозуміти, що історія користувача занадто велика і постає потреба її розбити? Тоді, коли виникають «дзвіночки»:

- ✓ її оцінка дуже велика (наприклад, 13–21 *Story Points*);
- ✓ її неможливо завершити за один спринт;
- ✓ вона містить кілька різних цілей користувача в одній фразі;
- ✓ під час обговорення всі говорять: «це щось на 2–3 спринти».

Основні підходи до декомпозиції історій користувачів базуються на важливому принципі: мають відбуватись вертикальні зрізи, а не просто «розрізати по технологіям» (*UI, backend, DB*).

Розбиття за кроками сценарію (Workflow steps). Є загальний процес користувача і він ділиться на кроки. Кожен крок - окрема історія з окремою цінністю.

Приклад

Epic: «Як студент, я хочу записуватися на курси, щоб проходити навчання».

Можливі *User Stories*:

«Як студент, я хочу переглядати список доступних курсів...»

«Як студент, я хочу фільтрувати/шукати курси...»

«Як студент, я хочу переглядати деталі курсу...»

«Як студент, я хочу записуватися на курс...»

«Як студент, я хочу бачити список курсів, на які я записаний...»

Розбиття за ролями (хто користується). Кожна роль має свій піднабір історій.

Приклад

Epic: «Система управління курсами».

Можливі *User Stories*:

User Stories для студента: студент дивиться курси, проходить тести

User Stories для викладача: викладач створює курси, завдання

User Stories для адміністратора: адміністратор модерує, керує доступами

Розбиття за варіантами поведінки.

Приклад

Story (занадто широка): «Як користувач, я хочу керувати своїм профілем (змінювати ім'я, email, пароль, аватар, мову, налаштування сповіщень), щоб налаштувати свій досвід.»

Можливі *User Stories*:

«Як користувач, я хочу змінювати основну інформацію профілю (ім'я, прізвище)...»

«Як користувач, я хочу змінювати email та підтверджувати його...»

«Як користувач, я хочу змінювати пароль...»

«Як користувач, я хочу завантажувати або змінювати аватар...»

«Як користувач, я хочу налаштовувати, які сповіщення отримувати...»

Розбиття за CRUD (створення (*Create*)/ перегляд (*Review*)/ оновлення (*Update*)/ видалення (*Delete*)) при умові, що кожна дія несе цінність. Не завжди треба розбивати на всі 4, але інколи це логічно.

Приклад

Story: «Як викладач, я хочу керувати завданнями курсу...».

Можливі *User Stories*:

User Stories для створення завдання

User Stories для перегляду списку завдань

User Stories для редагування завдання

User Stories для видалення завдання

Розбиття за рівнем деталізації. Спочатку більш загальна історія, потім - уточнення.

Приклад

«Як студент, я хочу бачити прогрес по курсу (загальний %)...»

«Як студент, я хочу бачити прогрес по кожному модулю...»

«Як студент, я хочу бачити, які саме лекції залишилися непройденими...»

Отже, спочатку проводиться декомпозиція так, щоб кожна *User Story* несла помітну цінність користувачу, а потім уже вона розбивається всередині команди на *dev/test/design Tasks*.

Приклад декомпозиції. *Epic*: «Як викладач, я хочу створювати й керувати онлайн-тестами для своїх курсів, щоб оцінювати знання студентів».

Крок 1. Розбиваємо *Epic* на *User Stories*:

«Як викладач, я хочу створювати тест із питаннями, щоб перевіряти знання студентів».

«Як викладач, я хочу додавати різні типи питань (одна відповідь, кілька відповідей, відкрита), щоб мати гнучкі тести».

«Як викладач, я хочу призначати тест конкретній групі студентів, щоб розподілити задачу в часі».

«Як студент, я хочу проходити тест онлайн, щоб отримувати миттєвий зворотний зв'язок щодо своїх відповідей».

«Як студент, я хочу бачити результат після завершення тесту, щоб знати, чи склав я тест відповідно до вимог курсу».

«Як викладач, я хочу бачити результати тестів по всіх студентах, щоб сформувати рейтинг оцінок».

Крок 2. Декомпонуємо одну з *User Stories* (наприклад, №1): «Як викладач, я хочу створювати тест із питаннями, щоб перевіряти знання студентів» на:

«Як викладач, я хочу створити порожній тест, щоб мати стартовий шаблон для подальшого наповнення тестовими питаннями».

«Як викладач, я хочу додавати питання до тесту, щоб гнучко оновлювати та розширювати тестовий банк питань».

«Як викладач, я хочу редагувати питання в тесті, щоб покращувати їх якість та забезпечувати коректність і актуальність оцінювання».

«Як викладач, я хочу видаляти питання в тесті, щоб забезпечувати коректність тесту та виключати помилкові або застарілі питання».

Крок 3. Кожну з цих маленьких *User Stories* команда потім розіб'є на задачі, наприклад, для «створити порожній тест»: *Task 1* (дизайн форми створення тесту), *Task 2* (API для створення тесту), *Task 3* (збереження в БД), *Task 4* (валідація даних) та *Task 5* (unit-тести / UI-тести).

Практичні поради успішної декомпозиції *User Stories*:

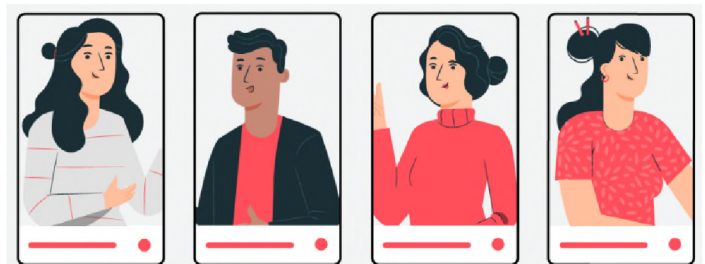
1. історія після декомпозиції має бути *INVEST*;

2. якщо історія не входить у спринт - майже напевно її треба розбивати на менші;
3. не слід проводити розбиття історії до абсурду: *User Story* несе цінність для користувача, а *Task* - це робота для команди;
4. потрібно пам'ятати, що у будь-якому випадку одна *User Story* завжди належить тільки одному *Epic*. Коли ж виконують всі *Tasks* конкретної *User Story*, її закривають і так відбувається з усіма блоками аж до *Done* усіх *Epics* - тоді можна сказати, що *System* = *Backlog* побудована.

5.3. Як писати ефективні *User Stories*

User Stories, як вже попередньо говорилося, це короткі описи потреб користувача, сформульовані з точки зору цінності, яку він отримує. Вони мають бути зрозумілими, перевірюваними й корисними для команди. Тому, перш ніж писати історії користувачів, важливо провести відповідне дослідження користувачів і визначити персону. Оскільки саме на цих персонажах користувачів базуватимуться історії користувачів.

Персона користувача - це узагальнений, але реалістичний образ типового представника цільової аудиторії. Вона описує цілі, мотивації, поведінку, потреби



та проблеми реальних користувачів. Персони допомагають команді краще розуміти, для кого саме вони створюють продукт.

Коли мова заходить про техніку написання історії користувача, є три основні вимоги, про які слід пам'ятати, перш ніж почати. Вони відомі як 3 Cs (перші літери англійських назв).



Рис.5.4. Ілюстрація методики The 3C's

Методика *The 3C's* описує три основні елементи ефективної *User Story* у гнучкій розробці. Вона допомагає командам створювати прості, зрозумілі, обговорювані й перевірювані вимоги.

Card (картка) - формулювання *User Story* (короткий текстовий опис потреби користувача у стислому форматі: Як [користувач], я хочу [дія], щоб [цінність]). Вона має бути короткою, зрозумілою та зосередженою на цінності для користувача. *Card* відповідає на запитання: «Що потрібно користувачу?»

Conversation (розмова) - уточнення вимог. Тут *User Story* постає як привід для діалогу між Замовником, аналітиками, розробниками, дизайнерами та тестувальниками. Під час розмови команда з'ясовує деталі поведінки системи, бізнес-правила, варіанти використання, альтернативні сценарії, обмеження. Розмова допомагає уникнути непорозумінь і створює спільне бачення функціональності. *Conversation* відповідає на запитання: «Як саме це має працювати?»

Confirmation (підтвердження) - критерії прийняття (набір умов, за яких *User Story* вважається виконаною). Вони забезпечують чіткість, можливість тестування, однакове розуміння готовності, впевненість Замовника, що вимога реалізована правильно. Критерії прийняття часто описують у форматі *Given* (початкові умови)/ *When* (дія користувача)/ *Then* (очікуваний результат). *Confirmation* відповідає на запитання: «Як знаємо, що все зроблено правильно?»

The 3C's означає, що *User Story* - це не просто текст, а процес: картка дає короткий опис потреби → розмова уточнює деталі → підтвердження визначає критерії, які дозволяють перевірити правильність реалізації. Саме така структура робить історії користувача ефективними, зрозумілими та корисними для всієї команди.

Для того, щоб перевірити якість *User Story* і переконатися, що вона буде зрозумілою, реалізованою та корисною для команди використовується метод *INVEST*. Це своєрідний чекліст, за яким оцінюють, наскільки добре сформульована історія користувача.

Метод *INVEST* - емпіричне правило, яке допомагає створювати найкращі історії користувачів та є мнемосхемою для таких найкращих практик:

- | | | |
|---|-------------|--|
| I | independent | ✓ <i>Independent (незалежність)</i> - історія повинна бути самостійною і не залежати від завершення інших історій, щоб уникнути проблем з розстановкою пріоритетів та плануванням; |
| N | negotiable | ✓ <i>Negotiable (обговорюваність)</i> - історії користувачів обговорювані, відображають зміст, а не деталі, не містять конкретних кроків реалізації; |
| V | valuable | ✓ <i>Valuable (цінність)</i> - історії користувачів мають бути цінними як для команди, так і для самих користувачів. Історія |

користувача має дати команді чітке уявлення про цілі, яких має досягти функція під час її розробки, а для користувача це означає продукт, який постійно відповідає його потребам;

- ✓ *Estimable (оцінювання)* - має бути можливість оцінити обсяг роботи за складністю та ресурсовитратами, необхідний для функції, запропонованої в історії користувача;

- ✓ *Small (маленький)* - історії користувачів мають бути короткими та такими, що запам'ятовуються. Лише два рядки про те, чого користувач хоче досягти за допомогою конкретної функції продукту. Якщо це загальний опис програми або певної категорії функцій, тоді необхідно створювати епік та розбивати його на окремі історії. Ідеальна історія користувача повинна вписуватися в одну ітерацію. Також не потрібно згадувати будь-які особливості дизайну інтерфейсу користувача в історії користувача;

- ✓ *Testable (тестованість)* - як і більшість речей у процесі розробки, історії користувачів мають легко перевірятися за допомогою чітко прорахованих результатів для ключових показників ефективності. Зазвичай тут вступають у дію критерії прийняття.

Найкраще, що можна порадити, коли справа доходить до регулярного написання прийнятних історій користувачів - використовувати шаблон. Це чудовий спосіб переконатися, що будь-хто у команді, хто пише історії користувачів, завжди розповідають про те, Хто, Що та Чому. Це деталі, які потрібні, щоб переконатися, що було зібрано всю необхідну інформацію в максимально стислий спосіб.

Існує багато корисних цифрових шаблонів, які можна роздрукувати або використати онлайн. Добре і те, що шаблони також можуть мати багато інших полів, які можуть бути корисними, наприклад точки *scrum* і область для написання критеріїв прийняття.

У сучасних командах розробки дуже популярним підходом є використання готових цифрових шаблонів *User Stories*, які можна застосовувати в *Jira*, *Trello*, *Azure DevOps*, канбан-дошках або просто роздрукувати й використовувати на фізичній стіні спринту.

Такі шаблони допомагають стандартизувати роботу й зробити кожну *User Story* структурованою, повною та готовою до розробки. На рисунку 5.5 наведено зразок історії користувача в *Azure DevOps*. Розглянемо більш детально анатомію *User Story* в *Azure DevOps* (переважно її структура зберігається і в інших інструментах).

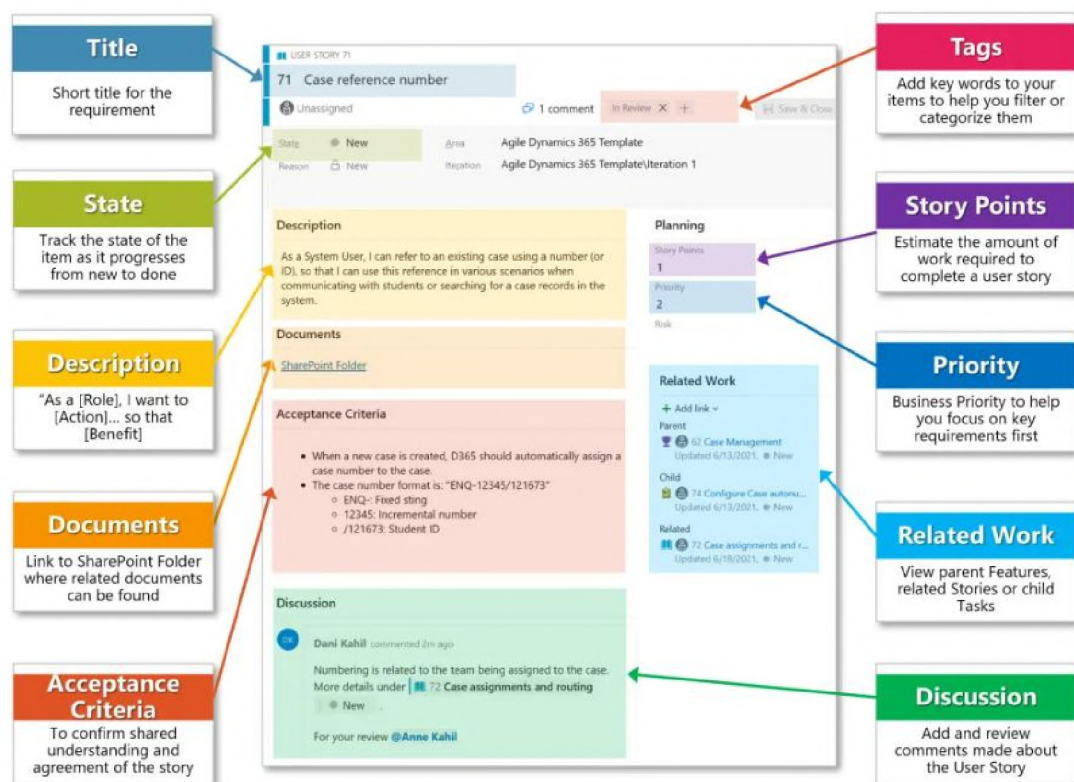


Рис. 5.5. Анатомія User Story в Azure DevOps

Title (заголовок) - коротка назва вимоги, стислий опис сутності *User Story*, щоб команда легко її знаходила в структурі *Backlog* продукту (можна включати число в заголовок, якщо є логічна послідовність історій в епіку).

State (стан) - показує поточний статус *User Story* (дає змогу бачити, у якій фазі робочого процесу перебуває вимога). Використовують наступні стани: *New*, *In Analysis*, *In Dev*, *In Test*, *In UAT* та *Done*.

Description (опис) - містить основний текст *User Story*, оформлений у форматі: «Як [роль], я хочу [дія]... так що [користь]».

Documents (документи) - посилання на пов'язані файли, технічні специфікації, макети, діаграми (допомагає команді знайти додаткову інформацію, пов'язану з історією).

Acceptance Criteria (критерії прийняття) - перелік умов, за яких *User Story* вважається виконаною (забезпечує спільне розуміння того, що саме потрібно реалізувати).

Tags (теги) - дозволяють додавати ключові слова для фільтрації, пошуку й категоризації історій, наприклад: *UI*, *backend*, *bug*, *high priority*.

Story Points (оцінка складності) - показує, скільки зусиль потрібно команді для реалізації *User Story* (використовується у *Scrum*-плануванні).

Priority (пріоритет) - визначає бізнес-важливість вимоги, команда бачить, що потрібно робити в першу чергу (*high / medium / low* або інші схеми).

Related Work (пов'язані задачі) - блок, який показує пов'язані *User Stories*, технічні задачі, підзадачі, епіки (допомагає бачити контекст та залежності між задачами).

Discussion (обговорення) - коментарі команди, уточнення, питання, рішення (обговорення ведеться безпосередньо в *User Story*, щоб не втратити контекст).

Найчастіше бізнес-аналітик буде тісно співпрацювати з власником продукту, щоб допомогти описати та проаналізувати вимоги. Бізнес-аналітик створить початкову історію користувача, а потім співпрацюватиме з командою

розробників, щоб завершити додаткові деталі, поки історія користувача не буде «готовою» до призначення в команду *Dev* для розробки.

Розглянемо детальніше кожен стан *User Story*, що використовується для управління вимогами в *Azure DevOps Boards*.



Коли нова вимога визначена, необхідно створити її як історію користувача. Для нових вимог, рекомендовано створити її з мінімальною кількістю інформації для початку. Нова вимога повинна бути розглянута і схвалена

власником продукту, перш ніж вона може бути проаналізована. Тому достатньо створити вимогу та звернути увагу власника продукту, щоб він міг її переглянути. Зазвичай для нових вимог заповнюють такі поля: *Title*, *State* (в даному випадку стан *NEW*) та *Priority*. Як тільки власник продукту вирішить, що вимога може прогресувати, тоді можна перейти до наступного етапу.

Саме тут зазвичай витрачається більше часу на розуміння деталей вимоги. Як правило, в цей час відбуваються наступні заходи:

✓ *Conversations & Workshops* (обговорення

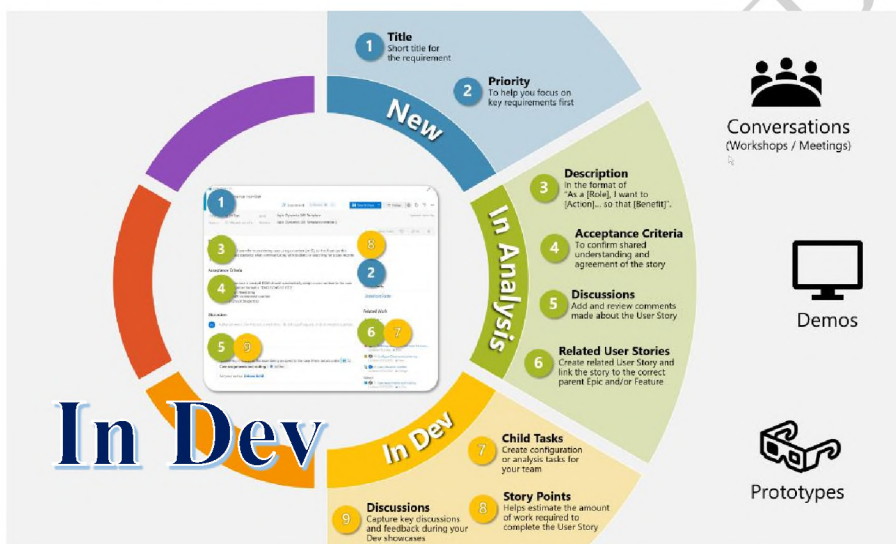
та проведення семінарів) з метою переходу до конкретних процесів і правил та документування їх у критеріях прийняття.



✓ *Demo*: під час семінарів можна продемонструвати функції «з коробки», доступні в *Dynamics 365* або *Power Platform* для задоволення вимог, що допоможе надати деталі, необхідні для роботи.

✓ *Prototyping* (прототипування): якщо вимога не може бути задоволена функціями «з коробки», можна приділити більше часу, щоб подумати про інші рішення для задоволення вимоги (наприклад, прототип), а потім прототип може бути використаний бізнес-аналітиком для зображення як може виглядати рішення.

На цьому етапі, як правило, фіксують наступну інформацію в історії користувача: *State (In Analysis)*, *Description*, *Acceptance Criteria*, *Discussion*, *Related Work*, *Tags*.



Коли отримується спільне розуміння того, що таке вимога, настає час планувати роботу розробки. Зазвичай історію оцінюють в *Story Points*, щоб зрозуміти її складність. Мета

оцінки полягає в тому, щоб виявити будь-яке потенційне непорозуміння між розробниками і переконатися, що вони всі більш-менш погоджуються щодо складності вимоги. Оцінка історії також допомагає з плануванням потенціалу під час планування спринту. Тут історія користувача доповнюється *Child Tasks* (завдання аналізу, дизайну або розробки створюються таким чином, щоб команда могла донести історію), *Story Points*, *Discussion*.



In TEST. Саме тут відбувається внутрішнє тестування (як правило, *QA* буде створювати і виконувати тестові випадки для кожної історії користувача). Мета полягає в тому, щоб визначити будь-які проблеми так, щоб *Dev Team* може швидко виправити, перш ніж розгортати його в *UAT*. Тести вручну зазвичай створюються для перевірки відповідності кожної історії користувача критеріям прийняття, визначеним відповідно до вимог користувача.

Подальші обговорення відбуваються між членами команди щодо прогресу тестування або з'ясування будь-якого непорозуміння щодо вимоги.

IN UAT. Коли модульне тестування завершено, *User Story* переходить до середовища приймального тестування (*User Acceptance Testing, UAT*), де його тестує спеціальна команда. Якщо історія користувача потребує значної переробки, стан можна оновити назад до *IN Dev*.

Done. Після *UAT* власник продукту підписує історію користувача і вона може бути розгорнута у виробництві для задоволення вигод кінцевих користувачів з бізнес-цінності вимоги. Тут можна закрити історію користувача як зроблено.

У проєктах з використанням *Azure DevOps* часто використовують теги, щоб доповнити стан і допомогти керувати вимогами.

Загальні (*General*) теги, незалежні від стану:

✓ *In Review*: вказує на те, що історія повинна бути розглянута/обговорена на наступному семінарі або зустрічі з бізнес-аналітиком.

✓ *PO Review*: вказує на те, що історія тепер може бути розглянута власником продукту, щоб отримати схвалення для прогресу від *New* до *In Analysis* та від *In Analysis* до *In Dev*.

✓ *Demo*: вказує, що зроблений аналіз, дизайн та прототипування, і вже можна демонструвати забезпечення історії за допомогою функцій *Dynamics 365*.

✓ *Blocked*: вказує на те, що не можна прогресувати в історії через зовнішні фактори (наприклад, історія описує занадто складну вимогу і може знадобитися додаткове обговорення для її прогресу).

5.4. Критерії прийняття та завершеності *User Stories*

Критерії прийняття (*Acceptance Criteria*) - це умови, які має виконати *User Story*, щоб власник продукту міг її прийняти (вони застосовуються на рівні *User Story*), а саме :

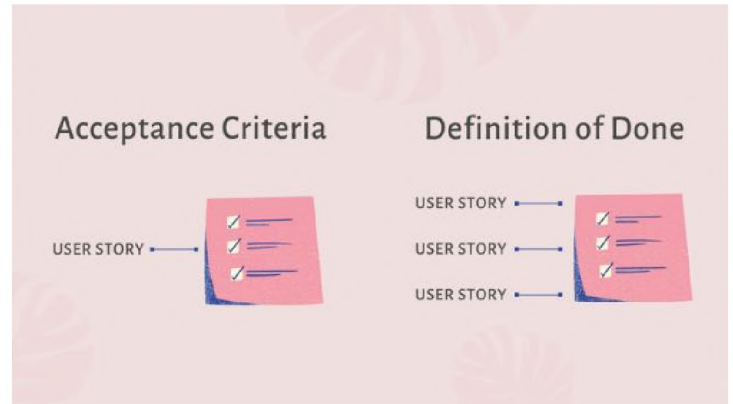
✓ дати опис очікуваної поведінки системи з точки зору користувача;
✓ чітко визначити, що саме має бути реалізовано;
✓ забезпечити спільне розуміння між бізнесом, аналітиками, розробниками й тестувальниками;

✓ надати можливість створити тест-кейси;

✓ гарантувати, що команда реалізує саме те, що очікує Замовник.

Кожна *User Story* має свій унікальний набір *Acceptance Criteria*.

Критерії завершеності (Definition of Done) - це загальний стандарт якості, який визначає, що робота (*історія*, задача або інколи цілий інкремент) завершена повністю і правильно. Це не опис функціональності, а умови, що гарантують якість виконання. Критерії завершеності застосовуються на рівні всієї команди, всіх історій користувача, всіх задач. Це - спільне правило для всіх вимог, яке:



- ✓ забезпечує єдині стандарти виконання для всієї команди;
- ✓ гарантує, що історія не тільки працює, але й готова до релізу;
- ✓ зменшує кількість переробок;
- ✓ підтримує якість продукту.

Отже, *Acceptance Criteria* прописуються окремо для кожної *User Story*, а *Definition of Done* - загальні вимоги для всіх *User Stories* проєкту. Головна різниця між *Acceptance Criteria* та *Definition of Done* наведена в таблиці 5.2.

Таблиця 5.2 - Різниця між Acceptance Criteria та Definition of Done

<i>Критерії прийняття (Acceptance Criteria)</i>	<i>Критерії завершеності (Definition of Done)</i>
Стосуються що повинна робити функція	Стосуються як повинна бути виконана робота
Унікальні для кожної <i>User Story</i>	Однакові для всіх <i>User Stories</i>
Описують поведінку, функціональність	Описують якість, процес, завершені етапи
Дають тестувальнику основу для перевірки функції	Дають впевненість, що задача повністю готова
Створює власник продукту/ бізнес-аналітик	Створює команда / <i>Scrum Team</i>

Definition of Done - це домовленість про те, як команда буде працювати в процесі - чіткі критерії, за якими домовляються працювати та створювати якісний продукт на кожній ітерації.

Приклад

Прописати критерії завершеності (*DoD*) для трьох *Work Items*:
Зробити *Scrum*-презентацію
Зробити *DevOps*-презентацію
Зробити *PMP*-презентацію

Критерії завершеності, загальні для всіх трьох задач, можуть бути такими:

- сторінки презентації пронумеровані;
- тексти вчитані редактором на предмет помилок;
- вказана інформація про авторські права;
- на кожній сторінці є логотип компанії;
- щонайменше 30% кожної з презентацій є візуальні образи;
- презентація пройшла «4 eyes principle», тобто її переглянули і погодили дві людини.

Acceptance Criteria - точні деталі, унікальні для кожної *User Story*, це набір умов, які підтверджують, що вона реалізована. Критерії прийняття завжди можна перевірити по чіткому параметру «так/ні». Неможливо виконати якийсь критерій наполовину: він або виконаний, або ні. Дякуючи *Acceptance Criteria* команда розробки знає, як повинен виглядати готовий результат певної вимоги. В цілому *Acceptance Criteria* повинні відповідати наступним характеристикам:

- ✓ *бінарні*: можуть мати тільки два унікальні результати – «успіх» або «відмова» (не може бути терміну «частковий», тому що критерій прийняття завжди повинен давати «зелене» або «червоне» світло для подальшого руху);
- ✓ *однозначні*: вони інтерпретуються лише одним способом (неправильно прописати: «Форма пофарбована в яскравий колір» або «Система повинна бути *user friendly*», адже такі критерії можна розуміти по-різному);

- ✓ *мають підтвердження*: повинні бути написані таким чином, щоб клієнт зміг швидко їх перевірити;
- ✓ *повні*: список критеріїв повинен включати всі функціональні вимоги (все, що необхідно зробити по кожній вимозі, описують в критеріях прийняття).

Приклад

Вимога - дозволити користувачу завантажувати файл з рисунком.

Критерії прийняття:

- доступні визначені формати файлів, наприклад, *jpeg, png, gif*;
- після завантаження з'являється повідомлення, що файл успішно завантажено.

Приклад

Вимога - дозволити користувачу змінювати пароль.

Критерії прийняття:

- новий пароль містить не менше 8 символів;
- новий пароль не може дублювати старий;
- пароль не може починатися та закінчуватись пробілом.

Перераховані атрибути повинні бути виконані для конкретних вимог, вони не описують весь процес.

Отже, основна мета критеріїв прийняття - окреслити умови, які мають бути виконані, щоб історія користувача була прийнята. Вони допомагають переконатися, що всі, від користувачів до розробників, розуміють, що повинен робити готовий продукт.

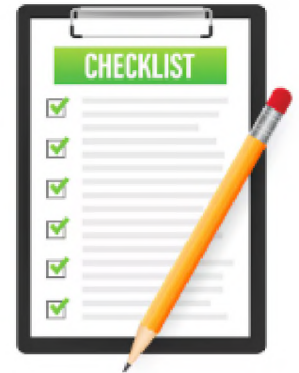
Формати критеріїв
прийняття

Є кілька способів написати критерії прийняття. Одним із популярних методів є формат *Given-When-Then*:



- ✓ *Given* - умови початку
- ✓ *When* - дія користувача
- ✓ *Then* - очікуваний результат

Ще один простий метод
- контрольний список
(*Checklist*). Тут ви



перераховуєте всі умови, які мають бути виконані в пунктах. Цей підхід зрозумілий і простий у використанні.

Загалом існує *два типи критеріїв прийняття*:

- ✓ *на основі правил*: цей тип є простим, керується політикою або бізнес-правилами та найкраще підходить для інформації, що керується даними;

- ✓ *на основі сценарію*: цей тип є описовим і описує різні випадки використання та найкраще підходить для результатів, керованих поведінкою.

На основі правил виконується більше перевірок такого плану: чи щось відображається, чи правильно, чи завершено, чи порядок правильний, чи можуть користувачі виконувати деякі функції тощо.

Приклад

Вимога - дозволити користувачу змінювати пароль.

Критерії прийняття:

- пароль має містити щонайменше 8 символів;
- пароль повинен містити хоча б одну велику літеру;
- пароль повинен містити хоча б одну цифру;
- пароль не повинен збігатися з логіном;
- пароль не повинен містити пробілів.

Критерії прийняття на основі сценарію описують очікувану поведінку системи під час виконання певних дій користувача, а також результати, які повинні бути отримані після виконання цих дій.

Приклад

Вимога - email має бути валідним і унікальним.

Критерії прийняття:

Given користувач вводить email у полі

When він натискає «Зареєструватися»

Then система перевіряє формат email і показує помилку, якщо формат невірний.

Given користувач вводить email, який вже існує в системі

When він подає форму

Then система показує повідомлення: «Такий email уже зареєстрований»

Шаблон *Given-When-Then* не створюється одноосібно власником продукту або бізнес-аналітиком. Цей формат прописується на командній зустрічі, яка називається «Троє друзів».

Командна зустріч
«Троє друзів»

Agile-практика «Троє друзів» допомагає донести голос команди до клієнта та дійти до загального розуміння вимог.

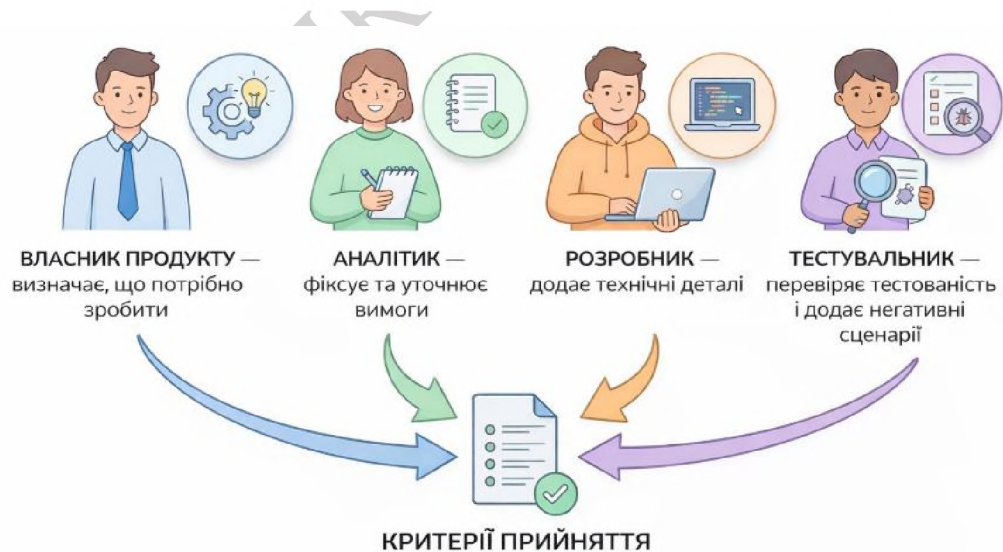


Рис.5.6. Ілюстрація командної зустрічі з метою формулювання критеріїв прийняття

На зустрічі збираються представники трьох ролей:

- ✓ той, хто описує вимоги (бізнес-аналітик / власник продукту) та слідкує, щоб у всіх в команді було однакове розуміння та очікування від користувацьких історій;
- ✓ хто буде розробляти (розробники обговорюють своє розуміння вимог і того, що необхідно для створення інкременту);
- ✓ хто буде тестувати те, що описано у вимогах (тестувальники сліdkують, щоби всі критерії приймання відповідали тестовим випадкам).

Зустріч «Троє друзів» зазвичай проводиться на етапі уточнення історій користувача перед початком їх реалізації, наприклад під час підготовки елементів *Product Backlog* до виконання у наступному спринті. Під час цієї зустрічі учасники детально аналізують історію користувача, визначають її межі, уточнюють бізнес-правила та погоджують критерії прийняття. Тестувальник зазвичай пропонує можливі перевірки або тестові сценарії, розробник оцінює технічні аспекти реалізації, а представник бізнесу уточнює очікувану цінність і поведінку системи з точки зору користувача.

Результатом такої зустрічі є уточнена історія користувача, сформульовані критерії прийняття та узгоджене розуміння вимоги між усіма учасниками команди. Це дозволяє зменшити кількість непорозумінь під час розроблення, підвищити якість реалізованої функціональності та забезпечити більш ефективну співпрацю між бізнесом, розробниками та тестувальниками.

Дану техніку можна використовувати і поза *Agile*-процесом. Практика обговорень у форматі «трьох ролей» буде корисна в будь-якому проєкті, де є розробники та тестувальники, які далі будуть впроваджувати та перевіряти створені вимоги.

5.5. User Story Mapping: від ідеї до релізу

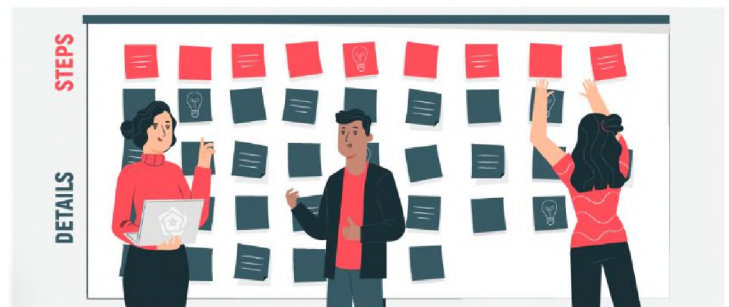
Якщо важко оцінити етапи проєкту, в беклогу з'являються прогалини, зривається запланований реліз, а команда не може домовитися, хто найбільше вкладає в продукт. Оскільки, у найпростішій формі, беклог продукту - це довгий список робіт, які потрібно зробити, щоб зрештою надати певну цінність користувачам або Замовникам. Такі беклоги стають дедалі довшими, і вони неминуче стають непосильними. Незалежно від того, чи команда витягує роботу з беклогу по одній, чи групуює її в спринти, визначення пріоритетів роботи в ньому має свої труднощі.

Створення карти історій користувача (*User Story Mapping*) – це можливість відійти від банального беклогу та створити більш цілісний, клієнтоорієнтований огляд роботи. Карта історій користувача - це візуалізація шляху, який клієнт проходить з продуктом, і включає дії та завдання, які він зазвичай виконує. Завдяки цій техніці в команди складається наочне розуміння цілісної картини проєкту, поділеної на спринти: від ідеї до реалізації. Візуальне оформлення допомагає побачити взаємозв'язок між окремими етапами, правильно розставити пріоритети й не пропустити важливі задачі.

User Story Mapping

Технологія створення карт історій користувача - чудовий спосіб зібрати всіх зацікавлених у проєкті осіб, поставити задачі, а також візуально відобразити як виглядає подорож користувача продуктом.

Створення карт (картування) історій користувачів – це часто використовуваний метод, який допомагає



організувати, структурувати та визначити пріоритети беклогу продукту за двома вимірами. Він візуально зображує весь беклог вздовж шляху користувача,

одночасно показуючи пріоритет історій користувачів, пов'язаних з кожним кроком процесу. Картування історій користувачів – це чудовий комунікаційний засіб для забезпечення однакової координації всіх учасників та дозволяє легко визначати релізи. Саме тут ви можете чітко побачити кроки від *MVP* до повноцінного рішення.

Структура карти історій користувача

В основі карти історій користувача лежать базові поняття: *Audience*, *Activities*, *Backbone*, *Tasks*, *Releases*.

✓ *Audience* - аудиторія є ключовим фактором для карти історій користувача та її відправною точкою.

Аудиторія описує персони користувачів, тому слід виділити деякий час для розуміння та визначення кому розробляється продукт, для яких рішень та які проблеми вони мають вирішити. Вибрану персону пишуть на стікері (липкому папірці) спеціального кольору, який буде використовуватись лише для персон. Персона розміщується у верхньому лівому куті карти (якщо є декілька, то може знадобитися кілька карт історій для цих персон).

✓ *Activities* - це цілі або основні види діяльності користувача-персони, які він прагне виконати під час взаємодії з системою. Це завдання високого рівня, які розташовані праворуч від персонажа; це завдання, які персона виконує за один раз (наприклад, написання електронного листа, створення системного облікового запису). Це найважливіші справи, які починаються та завершуються без перерви, вони існують самі по собі та досягають мікромети.

✓ *Backbone* - це основна структура карти історій користувача, яка формується з ключових активностей користувача, що відображають загальну логіку взаємодії з системою. На *User Story Map Backbone* (основа) розташовується у верхній частині карти та визначає послідовність основних етапів користувацького шляху (*User Journey*). Ці етапи об'єднують детальніші користувацькі історії, які описують конкретні функції або можливості системи.

✓ *Tasks* - це конкретні дії або кроки, які повинен виконати користувач для реалізації певної користувацької історії. На карті історій користувача завдання є найбільш деталізованим рівнем декомпозиції функціональності. Вони уточнюють, які саме операції виконує користувач у межах конкретної користувацької історії. Завдання є історіями користувачів і пов'язані з беклогом продукту.

✓ *Releases* – найбільша перевага карт історій користувача полягає в плануванні релізів - груп користувацьких історій, реалізація яких планується в межах певної версії програмного продукту. Виділення релізів на карті дозволяє визначити мінімально необхідний набір функціональності, який забезпечує цінність для користувача та може бути представлений у першій або наступних версіях системи.

На рисунку 5.7 показано структуру карти користувацьких історій з її двома вимірами. Зліва направо - це часовий вимір (не завжди обов'язковий), що показує послідовність взаємодій користувачів із продуктом. Другий вимір стосується користувацьких історій. Вони розташовані зверху вниз за зменшенням важливості або пріоритету.

Два верхні рядки карти користувацьких історій утворюють основу карти - *Backbone*. У першому рядку знаходяться дії (*Activities*), тобто основні цілі користувачів (*Audience*) під час взаємодії з продуктом (наприклад, для платформи електронної комерції такими діями можуть бути «Пошук певного продукту», «Купівля продукту» або «Додавання продукту до продажу». Як показує цей приклад, часовий вимір не обов'язково є дійсним для дій (додавання продукту до списку не залежить від пошуку в часі).

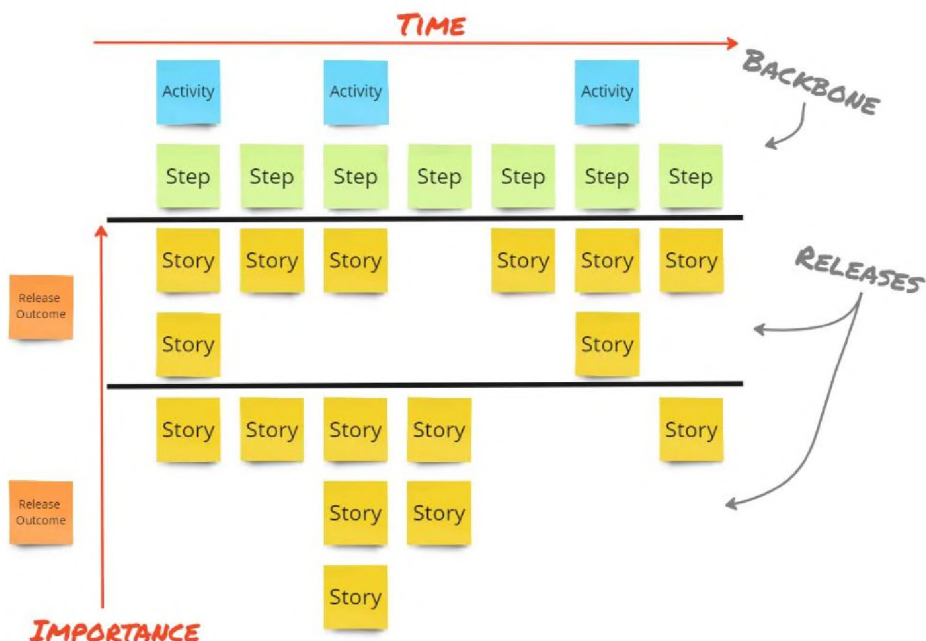


Рис.5.7. Схематичний вигляд карти історій користувача

Другий ряд *Backbone* містить фактичні кроки в межах шляху користувача, які пов'язані з активністю, розташованою над ними. Ці кроки розміщуються у певній послідовності, і до цього ряду застосовується часовий вимір (у нашому прикладі електронної комерції кроками, пов'язаними з активністю «Пошук конкретного товару», можуть бути: «Введення пошукового запиту в рядок пошуку», «Перегляд результатів» та «Застосування фільтрів»).

Нижче *Backbone* розташовуються користувацькі історії. Під кожним кроком шляху користувача подано всі історії, необхідні для повного задоволення потреб користувача на відповідному етапі. Вони впорядковуються зверху вниз у порядку зменшення важливості або пріоритету (наприклад, користувацькими історіями для кроку «Введення пошукового запиту в рядок пошуку» можуть бути: «Надання рекомендацій автодоповнення», «Відображення нещодавно введених пошукових запитів» або «Рекомендація популярних товарів»).

Користувацькі історії розподіляються між різними релізами шляхом введення горизонтальних розділювачів. Кожен реліз має визначений результат. Цей результат зазначається ліворуч від відповідного релізу і також містить інформацію про те, як буде вимірюватися успіх.

Проведення User Story Mapping

Щоб зустріч вийшла продуктивною, бізнес-аналітик заздалегідь має передбачити кілька речей: достатню кількість часу, список учасників та місце проведення.

Запрошення на зустріч. На першу сесію необхідно зібрати стейкхолдерів, які дивляться на продукт із різних позицій:

- ✓ представників тих користувачів, хто складав *User Journey* або людей з досвідом користувача, фахівців підтримки, які допомагають клієнтам проходити всі етапи;

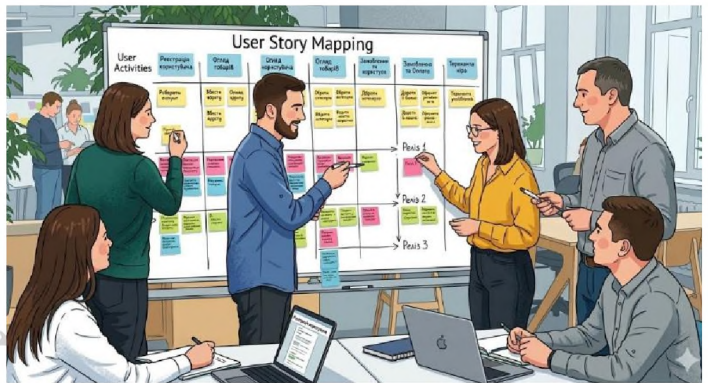
- ✓ якщо в компанії є *relationship* менеджери, *User Researchers*, обов'язково запросити їх, тому що вони знають, що користувачі вже вміють робити, а де поки відчують складності;

- ✓ представників команд розробки, які зможуть донести результати сесії решті команди;

- ✓ представників бізнесу, власників продукту, зацікавлених в отриманні вигоди з продукту та які приймають рішення про релізи;

- ✓ спеціалістів *UI* та *UX*;

- ✓ архітекторів, щоб підказали стейкхолдерам черговість змін.



Побудова скелет-основи (Backbone) User Journey

- ✓ верхня горизонтальна лінія «скелету» *User Journey* - кроки користувача, записані в тому порядку, в якому він їх виконує;
- ✓ вертикальні лінії («ребра») - деталі-дії для кожного кроку, побудовані за пріоритетністю зверху донизу.

Організація робочого простору. Для сесії знадобиться онлайн або офлайн дошка та різнокольорові стікери для позначення різних задач або активностей (наприклад, шлях користувача - зелені стікери, те, що буде в *User Story* - жовтим, а можливі складнощі - червоним, всі побажання бізнесу про пріоритети або релізи - невеликі яскраві наклейки розміром з чверть стікера).

Коли стейкхолдери зайдуть у кімнату, там повинно бути підготовлено все необхідне для процесу створення *User Story Map*:

- ✓ *зразок-легенда*: розміщені на окремому місці приклади стікерів з написом на кожному, що він означає, щоб було зрозуміло, який колір і навіщо використовувати;
- ✓ *паркувальне місце*: простір для питань і запису потенційно можливих складнощів;
- ✓ *територія самої карти*: в офлайн це може бути спеціальна дошка, стіна або навіть вікно;
- ✓ *місце для crazy ideas праворуч від картки*: для запису ідей, які виникли у процесі обговорення і, можливо, їх вдасться запровадити у майбутньому;

Попередню підготовку проведено: прописано покроковий шлях користувача, зібрано інформацію про клієнта, складено портрети персон. Простір для проведення організовано належним чином. Залишилося покликати стейкхолдерів і розпочати сесію.

Візуально карта історій на всіх трьох етапах виглядає як горизонтальна лінія «хребта» й вертикальні «ребра», але цілі для кожної стадії побудови будуть різними.

Створення деталей. User Journey готують заздалегідь і розміщують кроки в порядку виконання: на рисунку 5.8 це горизонтальна лінія із червоними квадратами. Зелені стікери – дії: створюються безпосередньо на зустрічі разом із учасниками. Тривалість такої сесії – близько 4 годин.

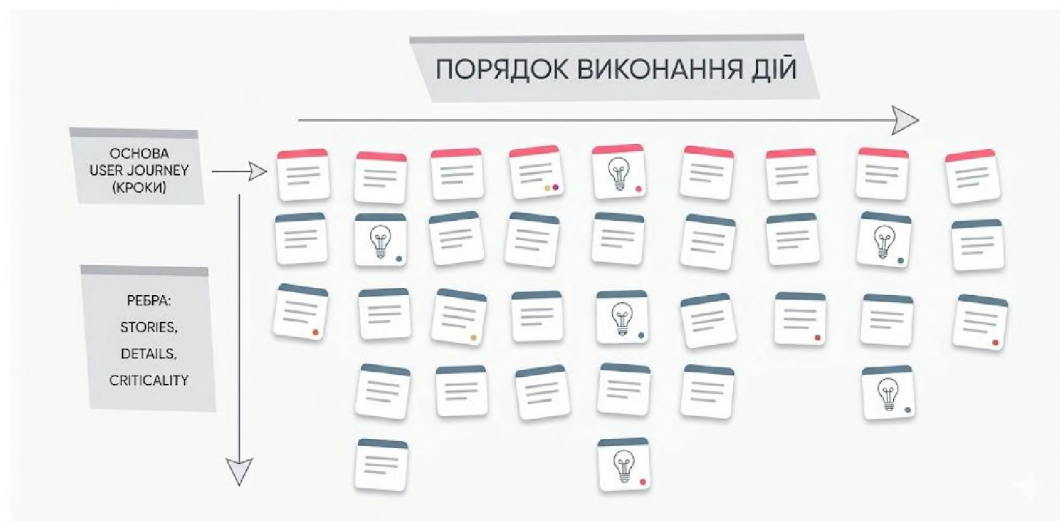


Рис.5.8. Створення карти історій користувача

Уточнення підготовленої карти. Після того, як прописані всі дії, озвучені складнощі, зафіксовані ідеї, саме час оцінити та пріоритезувати рішення. Для цього знову збирається група, проте трохи змінивши коло учасників. Правильно розставити пріоритети допоможуть представники трьох напрямків: бізнесу, користувачів, розробки. Під час обговорення окремим кольором виділяється те, що увійде до першого релізу, і лініями відокремлюються релізи один від одного (див.Рис.5.9). За часом другий етап займає близько 2 годин, його мета - виділити більш-менш важливі дії для бізнесу і провести «червону лінію»: що увійде до першого, другого релізу або ітерації.



Рис.5.9. Схема розбиття User Stories на релізи

Опрацювання технічної складової. На цьому етапі команда розробників прописує функціональні деталі, які знадобляться користувачеві при кожній дії: *frontend*, *backend* та інші. Дії системи створюють для кожної з персон-акторів. У кожного актора є своя ціль і своя цінність у користуванні продуктом. В той же час у команді розробників також є поділ на акторів за функціоналом: *frontend*, *backend*, тестування, дизайн. Над реалізацією кожної з функцій зазвичай працюють одразу кілька фахівців з різним обсягом роботи.

Після проходження трьох етапів отримується готова картина для всіх команд. Тепер можна починати втілювати те, що запланували, звертаючись до *user story map* протягом усього проєкту (додавати нові деталі, оновлювати пріоритети, відзначати виконані задачі).

Отже, *User Story Map* допомагає гнучко керувати проєктами та працювати з вимогами Замовника, представляючи їх наочно. Побудова карт історій користувача дозволяє правильно ставити і планувати задачі, враховуючи складнощі та зміни, які виникають вже безпосередньо в процесі розробки.

Для побудови карти віддалено можна використовувати і *діджитал-інструменти*, такі як *CardBoard*, *StoriesOnBoard*, *Easy Agile User Story Map for Jira*, *Agile User Story Map for Jira* тощо.

Діджитал-інструменти User Story Mapping

CardBoard (*Cardboardit.com*) - це онлайн-дошка (*digital whiteboard*), орієнтована на *User Story Mapping*, планування продуктів і спільну візуалізацію ідей.

Інструмент задуманий як простий і гнучкий спосіб співпраці команд над беклогом, функціоналом, картами користувачьких історій, роадмапами, майнд-мапами тощо (див. Рис. 5.10). *CardBoard* створено з ідеєю, що інструмент має бути легким, зрозумілим і спеціалізованим задля планування продукту.

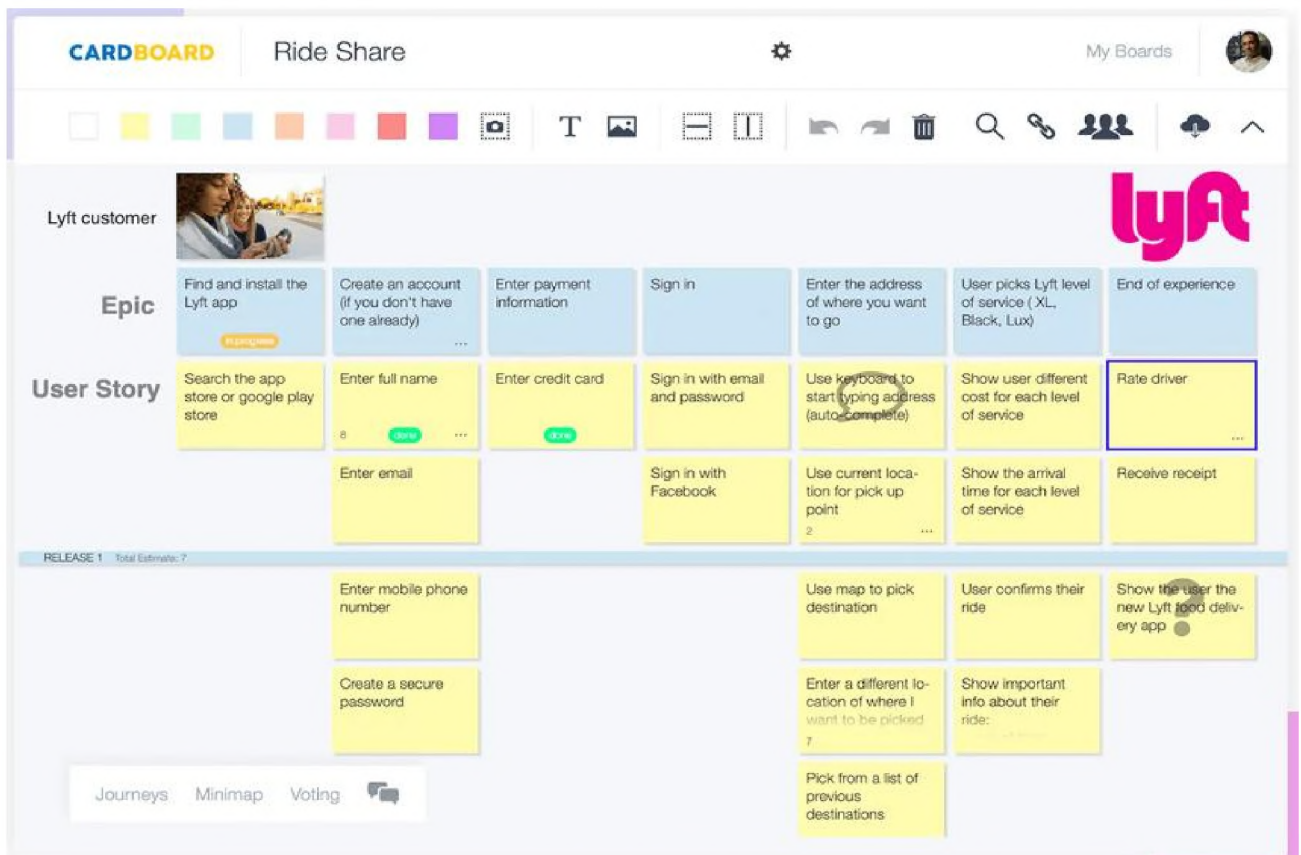


Рис.5.10. Зразок User Story Map у CardBoard

CardBoard дає низку корисних функцій, які часто потрібні командам продуктового чи *Scrum*-розробки (див. Табл. 5.3).

Таблиця 5.3 - Корисні функції CardBoard

Можливість	Корисність
Дошки, на які можна додавати «картки».	Візуальне планування, організація <i>Backlog/ Feature-set</i> /етапів, просте перетягування елементів
Картки із «зворотною» стороною.	На картці можна написати <i>User Story</i> , додати опис, критерії, статус, коментарі, вкладення, тобто все, що потрібно, щоб не забути деталі.
Шаблони: <i>User Story Map</i> , <i>Mind Map</i> , <i>Product Roadmap</i> тощо	Швидкий старт: не треба «з нуля» будувати структуру, є готові формати для різних задач (розробка, UX, стратегія).
Лінки між дошками / зв'язки / залежності	Можна створити «зв'язок» між різними частинами проєкту, показати, як задачі або функції взаємопов'язані.
Співпраця в реальному часі + гостьові посилання	Команда (або навіть зовнішні зацікавлені сторони) може працювати одночасно, обговорювати, додавати коментарі, робити зміни і це все онлайн.
Інтеграції з популярними інструментами.	Можна легко перенести <i>Backlog</i> / задачі між системами, а це зручно для тих, хто вже використовує ці інструменти, і хоче візуалізувати <i>Roadmap</i> або <i>Story Map</i> перед реалізацією.
Гнучкість: від <i>simple</i> до складних моделей.	<i>CardBoard</i> підходить і для «мозкового штурму», і для планування продукту, для упорядкування <i>Backlog</i> та планування релізу.

StoriesOnBoard (*Storiesonboard.com*) - це онлайн-інструмент для управління продуктом, побудований навколо концепції *User Story Mapping*. Він дає змогу структурувати *Backlog*, організувати *User Stories*, відобразити *User Journey*, спланувати релізи та відслідковувати прогрес і це все візуально, на карті (див.Рис.5.11). Платформа підходить для *Agile / Scrum / Kanban*-команд, для невеликих стартапів і великих організацій.

StoriesOnBoard пропонує набір функцій, які спрощують роботу над продуктом (див.Табл.5.4).

Agile User Story Map for Jira та *Easy Agile User Story Map for Jira* - це плагіни (додатки) для *Jira*, які дозволяють створювати й використовувати *User Story Mapping* безпосередньо всередині *Jira*, без сторонніх сервісів (див. Рис.5.12 та 5.13). Це два різні плагіни для *Jira*, хоча обидва виконують схожу роль - створення *User Story Map*.

Таблиця 5.4 - Корисні функції *StoriesOnBoard*

Можливість	Корисність
Візуальний беклог у вигляді двовимірної карти	Дає загальну картину продукту, <i>User Journeys</i> , допомагає виявити пропуски та спланувати логіку функціональності.
<i>Backlog / Release Planning</i> - створення релізів, планування, розподіл <i>Story Cards</i> по релізах	Легко бачити, що буде у першому <i>MVP</i> , у наступних релізах; легко планувати черговість реалізації.
Пріоритезація	Допомагає вирішити, які історії мають першочергове значення, з огляду на цінність та ресурси.
Оцінки	Планування завантаження команди, оцінка складності, розподіл роботи.
Співпраця / командна робота онлайн	Усі зацікавлені сторони мають спільне бачення і працюють на одній «дошці».
Інтеграції з іншими інструментами.	Легко переносити задачі, синхронізувати <i>Backlog</i> із task-менеджерами, зручно при впровадженні в існуючий робочий процес.
Динамічні <i>Roadmap / Timeline View</i>	Дає високорівневий огляд розвитку продукту, релізів, термінів, що корисно для планування та презентацій перед стейкхолдерами.
Збір фідбеку / ідей / інсайтів.	Допомагає систематизувати побажання користувачів і функції, які можна потім перевести у <i>Backlog</i> .

Easy Agile User Story Maps for Jira (дуже популярний плагін серед власників продукту, бізнес-аналітиків, UX-фахівців, команди дослідників продукту) володіє наступними особливостями:

- ✓ найбільш візуальний, зручний та привабливий для користувачів інструмент *User Story Mapping* у *Jira*;
- ✓ має панель *Backlog* справа - можна перетягувати історії на мапу;
- ✓ підтримує автоматичний розподіл по спринтах, релізах, версіях;
- ✓ має потужну інтеграцію з *Epics* - їх видно як горизонтальні «activities»;
- ✓ дуже легкий у використанні (підходить навіть людям без досвіду *Jira*);

✓ має функції для *User Journeys*, кількох *Swimlanes*, *Customer Goals*, *Story Grouping*.

Easy Agile Story Maps - це візуальна *Story Map* для продуктової роботи, проектування досвіду користувача та дослідження.

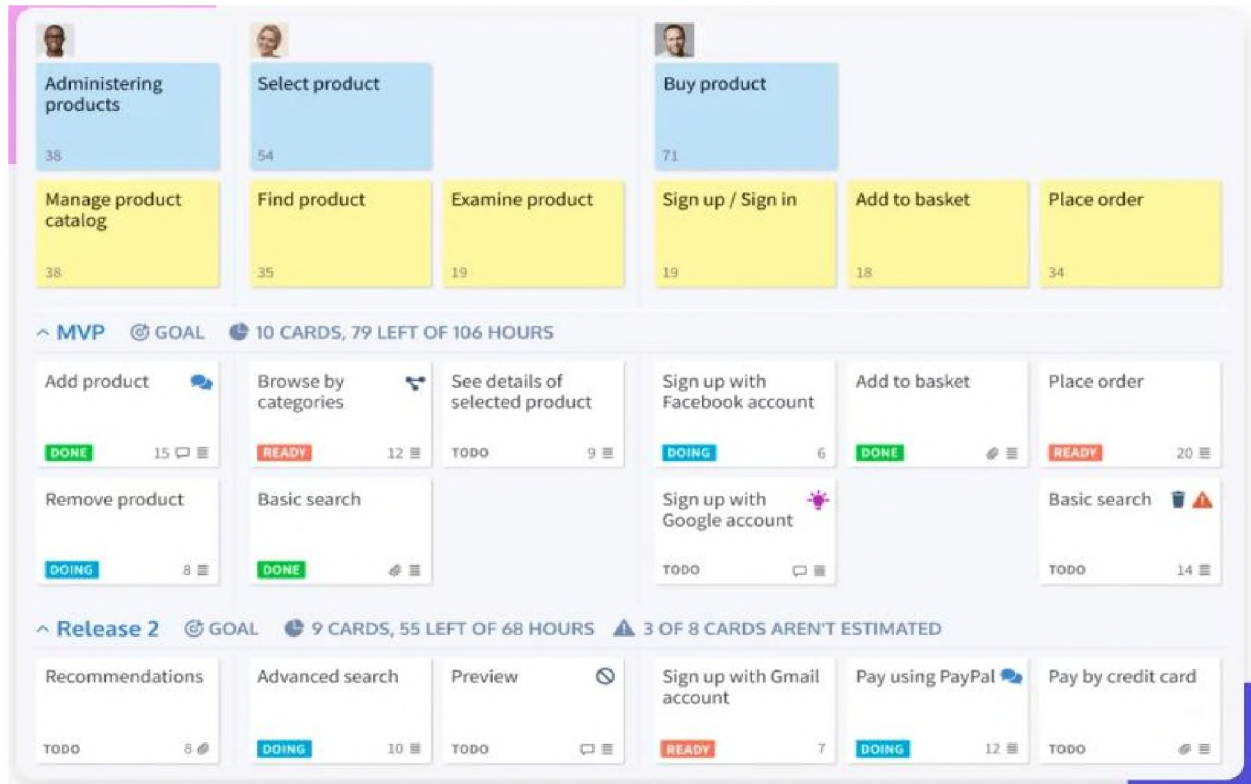


Рис.5.11. Зразок User Story Map у StoriesOnBoard

Agile User Story Map for Jira (популярний серед *Scrum Master*, *Dev Team*, які хочуть бачити *Backlog* у структурах *Jira*) володіє такими властивостями:

- ✓ функціональніший, має більш «джирний» і технічний інтерфейс;
- ✓ має оцінювання та фільтри;
- ✓ нагадує *Kanban*-дошку, але у форматі *Story Map*;
- ✓ сильний акцент на аналітиці та роботі з *Jira attributes* (статуси, оцінювання, пріоритети);
- ✓ немає яскравого *User Journey*, але добре працює з *Epic* → *Story*.

Agile User Story Map for Jira - це функціональна *Story Map* для технічних команд, які живуть у *Jira* та спринтах.

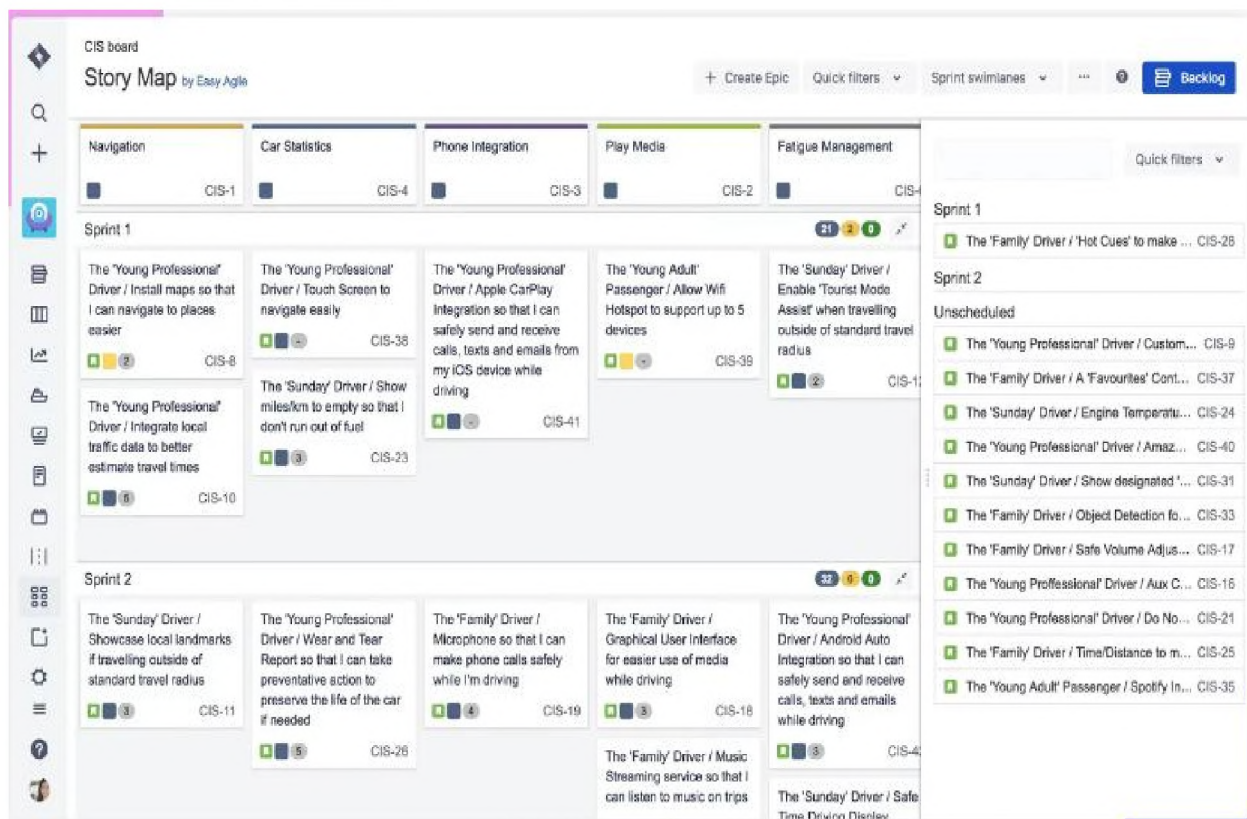


Рис. 5.12. Зразок *User Story Map* у *Easy Agile User Story Map*

Таблиця 5.5 - Ключові відмінності двох плагінів *Easy Agile User Story Maps* та *Easy Agile User Story Map for Jira*

Порівняння	<i>Easy Agile Story Maps</i>	<i>Agile User Story Map for Jira</i>
Інтерфейс	дуже візуальний, сучасний, інтуїтивний	технічний, схожий на <i>Jira Boards</i>
Фокус	<i>User Journey, Discovery, UX, Story Mapping</i>	<i>Delivery, Sprints, States, Story Points</i>
<i>Backlog panel</i>	є і дуже зручна	менш виражена
Планування релізів	просте та наочне	більш технічне
<i>Story Points + Epic Link</i>	є, але не основний акцент	основний акцент
Використання	<i>PO, BA, UX</i> -команди	<i>Dev</i> -команди, <i>Scrum Masters</i>
Для презентацій	ідеально	менш підходить
Ціна	дорожчий	дешевший
Горизонтальна структура	<i>Goals → Activities → Stories</i>	<i>Epics → Stories</i> (менше рівнів)
Додаткові фішки	<i>Customer Personas, User Goals</i>	технічні атрибути <i>Jira</i>

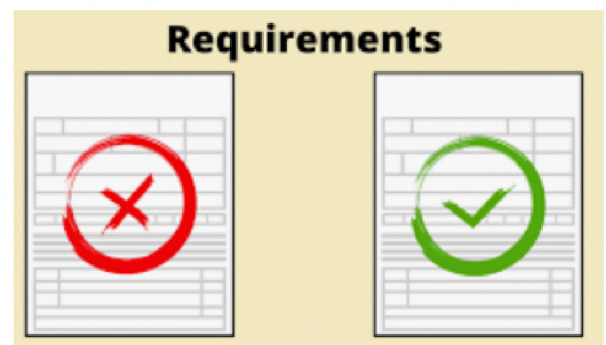


Рис. 5.13. Зразок User Story Map у Agile User Story Map

5.6. Правила написання історій користувачів та критеріїв прийняття

Сідаючи за написання *User Story* та *Acceptance Criteria*, варто постійно стежити за дотриманням норм граматики та орфографії. Граматичні помилки роблять формулювання неоднозначними та важкими для розуміння. Це особливо відчутно, коли вимоги пишуть нерідною мовою (зокрема, англійською). Якщо не дотримуватися їх, команда розробників може неправильно розтлумачити зміст.

Орфографічні помилки зменшують зрозумілість тексту з цієї ж причини. Деякі слова звучать однаково, але відрізняються значеннями залежно від написання, як-от в англійській: «red» і «read» або «ordinance» і «ordnance». Але є ще і правила, яких особливо важливо дотримуватися під час створення вимог.



Нижче наведемо 10 загальних правил, що в майбутньому дадуть змогу усунути значну частину проблем з формулюванням вимог і налагодити порозуміння між бізнес-аналітиком і командою розробників. Усі приклади наведено англійською мовою, оскільки більшість середовищ послугуються саме нею під час розробки програмних продуктів.



Правило 1. Дотримання визначеної структури. Побудова вимог за певним шаблоном допомагає забезпечити однорідність викладу інформації. Добре структуровані вимоги читач може зрозуміти і засвоїти без зайвого когнітивного навантаження.

User Story: дотримання загальноприйнятої структури.

Acceptance Criteria: вибір одного з двох форматів написання (правила або сценарій).



Правило 2. Уникання частки «not». Коли вимога містить «not», це означає «в жодному разі», а отже немає можливості перевірити виконання такої умови (якщо переписати вимогу без вживання «not», вона стане зрозумілішою і, щонайважливіше, піддаватиметься перевірці).

User Story

✗ неприйнятно: «*As a user, I want not to enter my credentials every time, so that I can access my account*» (Як користувач, я хочу не вводити свої облікові дані кожен раз, так що я можу отримати доступ до мого облікового запису).

✓ прийнятно: «*As a user, I want my credentials automatically filled out, so that I can access my account without re-entering them*» (Як користувач, я хочу, щоб

мої облікові дані автоматично заповнювалися, щоб я міг отримати доступ до свого облікового запису без повторного введення їх).

Acceptance Criteria

- ✗ неприйнятно: *The system shall not fail* (Система не підведе).
- ✓ прийнятно: *The system shall have an Availability of greater than or equal to 95%* (Система повинна мати доступність більше або дорівнює 95%).

Виняток: використовувати «*not*» у вимогах можна, якщо мається на увазі логічне заперечення, наприклад, «*The LogIn form shall not be red in color*» (форма *LogIn* не повинна бути червоного кольору). У більшості випадків це стосуватиметься *non-functional* вимог.



Правило 3. Формулювання речення з використанням активного стану дієслів. В *User Story* це стосується здебільшого частини «*I want*». Активний стан - це коли підмет у реченні виконавець дії, а не її об'єкт (якщо відповідальну за виконання дії сутність не вказати явно, буде не зрозуміло, хто чи що має виконати дію, тож перевірити виконання вимоги буде складніше).

User Story

- ✗ неприйнятно: «*As an online shopper, I want filters to be applied, so that I can find what I want*» (Як інтернет-покупець, я хочу, щоб були застосовані фільтри, щоб я міг знайти те, що я хочу).
- ✓ прийнятно: «*As a user, I want to apply search filters, so that I can find an item*» (Як користувач, я хочу застосувати фільтри пошуку, так що я можу знайти елемент).

Acceptance Criteria

- ✗ неприйнятно: *The Identity of the Customer shall be confirmed* (Особа Замовника має бути підтверджена, оскільки незрозуміло хто чи що є відповідальним за підтвердження особи Замовника чи виконує його).

✓ прийнятно: *The Accounting_System shall confirm the Customer_Identity* (*Accounting_System повинена підтвердити Customer_Identity*). Зауваження: глосарій повинен містити визначення термінів «*Accounting_System*» і «*Customer_Identity*».



Правило 4. Уникнення вживання займенників, особливо - невизначених. Займенники - слова на зразок «*it*», «*this*», «*that*», «*he*», «*she*», «*they*», and «*them*» («воно», «цей», «той», «він», «вона», «вони», «їх»). Цього треба уникати, щоб не виникло двояке тлумачення, тому краще вживати іменники замість займенників. Це особливо важливо, коли вимоги зберігаються в інструментах управління вимогами у вигляді окремих тверджень, що необов'язково організовані за порядком. Невизначені займенники, слова на зразок «*all*», «*another*», «*any*», «*anybody*», «*anything*», «*everybody*», etc. («усі», «інший», «будь-який», «будь-хто», «будь-що» тощо). Вони позначають неназваних людей або явища, а тому можуть бути інтерпретовані неоднозначно. Це призводить до того, що вимогу неможливо буде перевірити.

User Story

✗ неприйнятно: «*As a site member, I want to describe it, so that others can learn about me*» (Як учасник сайту, я хочу описати його, щоб інші могли дізнатися про мене).

✓ прийнятно: «*As a site member, I want to add a profile description, so that others can learn about me*» (Як учасник сайту, я хочу додати опис профілю, щоб інші могли дізнатися про мене).

Acceptance Criteria

✗ неприйнятно: *The controller shall send the Driver his itinerary for the day. It shall be delivered at least 8 hours prior to his Shift* (Контролер відправляє Водієві свій маршрут на день. Він повинен бути доставлений принаймні за 8

годин до його зміни). По-перше, критерій розбито на два речення (див. наступне правило 5); по-друге, в обох реченнях використано займенник «it».

✓ прийнятно: *The Controller shall send the Driver_Itinerary for the day to the Driver at least 8 hours prior to the Driver_Shift* (Контролер надсилає *Driver_Itinerary* протягом дня Водієві принаймні за 8 годин до *Driver_Shift*.)
Зауваження: терміни необхідно пояснити в глосарії, щоб однозначно визначити взаємозв'язок між *Controller*, *Driver_Itinerary* та *Driver_Shift*.



Правило 5. Уникнення сполучників. Сполучники (слова і словосполучки), що поєднують прості речення в складне, наприклад, «and», «or», «but», «as well as» («і», «або», «але», «замість того щоб») тощо. Їх вживання у вимогах зазвичай є ознакою того, що речення можна розбити на кілька окремих вимог. Виняток: «and», «or», «not» можна вживати на позначення логічних умов і кваліфікаторів (див. правило 6).

User Story

✗ неприйнятно: «*As a UI designer, I want to create and view an issue, so that I know what to test*» (Як дизайнер інтерфейсу, я хочу створити та переглянути проблему, щоб я знав, що перевірити).

✓ прийнятно: «*As a UI designer, I want to create an issue, so that I know what to test*» або «*As a UI designer, I want to view an issue, so that I know what to track and test*» (Як дизайнер інтерфейсу, я хочу створити проблему, щоб я знав, що тестувати / Як дизайнер інтерфейсу, я хочу переглянути проблему, щоб я знав, що відстежувати та тестувати).

Acceptance Criteria

✗ неприйнятно: *The User shall either be trusted or not trusted*. Тут причин кілька. Суть у тому, щоб класифікувати користувачів за однією з двох ознак, але пункт записаний у пасивному стані (див. правило 3) і допускає двояке

тлумачення: вимогу можна задовольнити, навіть якщо система всіх користувачів підряд вважатиме довіреними.

✓ прийнятно: *The Security_System shall categorize each User as one of the following: Trusted, Not_Trusted.*



Правило 6. Використання чітко визначеної конвенції запису логічних висловів. Кожна вимога має формулюватися як закінчена думка, окремим реченням. Саме тому слід уникати використання «and» в ролі сполучника для поєднання двох окремих думок. Однак вживати «and», «or», «not» в ролі логічних операторів цілком доцільно, коли йдеться про умови, яких стосується дієслово. В *User Story* також можливе використання «and», «or», «not» у мотиваційній частині «so that».

User Story

✗ неприйнятно: «*As a user, I want to receive notifications via emails and phone text message, so that I stay updated*» (Як користувач, я хочу отримувати повідомлення по електронній пошті і текстове повідомлення по телефону, так що я залишаюся в курсі). Сполучник «and» вжито як об'єднання двох вимог в одну. Під час тестування, якщо одна частина вимоги буде не виконана, це призведе до браку цілої вимоги, навіть якщо інша частина буде робочою.

✓ прийнятно: «*As a user, I want to receive notifications via emails, so that I stay updated*» / «*As a user, I want to receive notifications via phone text message, so that I stay updated*».

Acceptance Criteria

✗ неприйнятно: *The Engine_Management_System shall disengage the Speed_Control_Subsystem when the Cruise_Control is engaged and the Driver applies the Accelerator.* Сполучник «and» вжито в ситуації, яку можна інтерпретувати як пов'язування двох окремих думок.

✓ прийнятно: *The Engine_Management_System shall disengage the Speed_Control_Subsystem, when [the Cruise_Control is engaged AND the Driver applies the Accelerator].*



Правило 7. Уникнення вживання неточних термінів. Це правило порушують найбільше. Слід уникати слів, що дають неточну кількісну оцінку, таких як «some», «allowable», «several», «many», «a lot of», «a few» тощо. А також неточних прикметників, зокрема «ancillary», «relevant», «routine», «common», «generic», «significant», «typical» тощо. Прислівники використовують, щоб певним чином охарактеризувати дію, і зазвичай з ними проблем найбільше. Тому слід уникати слів на зразок «usually», «approximately», «sufficiently», «typically» тощо. Через вживання неточних прикметників і прислівників вимоги втрачають однозначність, можуть не піддаватися перевірці й загалом неповно показують сподівання зацікавлених сторін.

User Story

✗ неприйнятно: «*As an editor, I want to autocorrect some mistakes, so that I have a proper doc*» (Як редактор, я хочу автоматично виправити деякі помилки, щоб у мене був правильний документ).

✓ прийнятно: «*As a doc editor, I want to autocorrect detected spelling mistakes in an active document, so that I have no spelling mistakes in the doc*» (Як doc-редактор, я хочу автоматично виправляти виявлені орфографічні помилки в активному документі, щоб у мене не було орфографічних помилок у документі).

Acceptance Criteria

✗ неприйнятно: *The Flight_Information_System shall display the Tracking_Information for relevant aircraft.* Незрозуміло, які літаки вважати «відповідними».

✓ прийнятно: *The Flight_Information_System shall display the Tracking_Information of each Aircraft located less than or equal to 20 kilometers from the Airfield.* Тепер зрозуміло, для яких літаків треба показувати інформацію. Зауважте, що глосарій має містити визначення термінів «Aircraft», «Tracking_Information» та «Airfield».



Правило 8. Наведення точних цільових показників, які можна виміряти і які були б доцільними на тому рівні, на якому сформульовано вимоги. Хоч це правило і схоже на попереднє, але різниця в тому, що воно стосується конкретно показників. Деякі слова виражають невимірне кількісне означення, такі як «fast», «maximum», «minimum», «optimum», «user-friendly» тощо. Вони допускають неоднозначне тлумачення, а отже, їх треба замінити на точні значення, що піддаються вимірюванню.

User Story

✗ неприйнятно: «*As a passenger, I want to get my taxi promptly after I press confirm the pick-up point, so that I don't have to wait long*» (Як пасажир, я хочу, щоб отримати моє таксі швидко після того, як я натискаю підтвердити точку посадки, так що мені не доведеться чекати довго).

✓ прийнятно: «*As a passenger, I want to get my taxi in less than 120 seconds after I press confirm the pick-up point, so that I don't have to wait long*».

Acceptance Criteria

✗ неприйнятно: *The system shall use minimum power.* Незрозуміло, яке живлення вважати «мінімальним».

✓ прийнятно: *The system shall draw less than or equal to 50W of main power.*



Правило 9. Уникнення недосяжних абсолютів. Абсолют - це дещо недосяжне, як-от 100% availability (100% доступність системи).

Постає питання: як перевірити показник? Чи можливо буде довести, що рівень доступності системи саме 100%? Також треба уникати слів «*all*», «*always*» та «*never*», оскільки перевірка таких абсолютних вимог потребуватиме нескінченної кількості тестів. Увага: якщо показники вписали в *User Story*, з великою ймовірністю вони мають бути винесені в *Acceptance Criteria*.

User Story

✘ неприйнятно: «*As a Traveler, I want to know my precise location updated in real-time, so that I don't get lost*» (Як мандрівник, я хочу знати, що моє точне місце розташування оновлюється в режимі реального часу, щоб я не заблукав) «*Real-time*» може трактуватися по-різному, як приклад, це абсолют, який має на увазі відсутність затримки, що неможливо досягти і який не піддається перевірці.

✓ прийнятно: «*As a Traveler, I want to know my precise location every second, so that I don't get lost*» (Як мандрівник, я хочу знати своє точне місце розташування кожну секунду, щоб я не заблукав).

Acceptance Criteria

✘ неприйнятно: *The system shall have 100% availability*. 100% - це абсолют, якого неможливо досягти і який не піддається перевірці.

✓ прийнятно: *The system shall have an Availability of greater than or equal to 98%*.



Правило 10. Уникнення давати готові рішення, якщо немає передумов для використання саме вашого варіанта. Необхідно зосередитися на проблемі («що»), а не на її рішенні («як»). Будь-яка система потребує відповідний рівень вимог, де формулюються проблеми, але не їхні рішення. *User Stories* і *Acceptance Criteria* повинні містити високорівневі характеристики вирішення проблеми в загальному вигляді. Першою ознакою того, що у вимозі вписані рішення - вживання технічної термінології та жаргону, якими користуються розробники.

User Story

✗ неприйнятно: «*As a user, I want to join DebtTable1 with DebtTable2, so that I can make a payment list*» (Як користувач, я хочу приєднатися до *DebtTable1* з *DebtTable2*, так що я можу зробити список платежів).

✓ прийнятно: «*As a user, I want to see money that I owe, so that I can make a payment list*» (Як користувач, я хочу бачити гроші, які я винен, так що я можу зробити список платежів).

Acceptance Criteria

✗ неприйнятно: *By pressing a button on the traffic-light pillar, the Pedestrian signals his presence, and the light turns red for the traffic to stop.* Тут є деталі, пов'язані з розв'язком проблеми.

✓ прийнятно: *The Traffic_Control_System shall allow the Pedestrian to signal intent to cross the road.* Таке формулювання забезпечує певну свободу в розв'язанні проблеми, ймовірно, замість кнопки створять засіб автоматичного виявлення пішоходів. Зауваження: глосарій має містити визначення терміна «*Pedestrian*».



Питання для самоперевірки

1. Що таке *User Story* і для чого вона використовується в *Agile*? Яка стандартна структура *User Story*?
2. Чому *User Story* має бути короткою та зрозумілою?
3. Для чого в *User Story* вказують роль користувача?
4. Що означає фраза «*so that*» у *User Story*?
5. Чому *User Story* описується з точки зору кінцевого користувача?
6. Чим *User Story* відрізняється від *Task*?

7. Які характеристики має якісна *User Story* згідно з *INVEST*? Поясніть аббревіатуру *INVEST*.
8. Хто зазвичай пише *User Stories*?
9. Що таке *persona* і як вона впливає на написання *User Stories*?
10. Що таке *Epic* в *Agile*?
11. Чим *Epic* відрізняється від *User Story*?
12. Навіщо потрібні *Epics* у *Product Backlog*?
13. Хто вирішує, коли *Epic* потрібно розбити на *User Stories*?
14. Як *Epics* допомагають структурувати продукт?
15. Що таке *Epic Goal*? Як *Epic* пов'язаний з *Roadmap* продукту?
16. Що таке критерії прийняття (*Acceptance Criteria*)? Навіщо вони потрібні?
17. Чим критерії прийняття відрізняються від *Definition of Done*?
18. Чому критерії прийняття мають бути тестованими?
19. Що таке формат *Given/When/Then*?
20. Хто створює *Acceptance Criteria*?
21. Чому погано, якщо *User Story* не має критеріїв прийняття?
22. Чим ризикує команда, якщо *Acceptance Criteria* надто загальні?
23. Які переваги *rule-based* критеріїв прийняття?
24. Чи можуть критерії прийняття змінюватися під час спринту?
25. Що таке *Definition of Done*?
26. Чому *Definition of Done* допомагає забезпечити якість продукту?
27. Що таке *User Story Mapping*? Його використання у розробці продукту?
28. Що зазвичай розміщено на горизонтальній осі *User Story Mapping*?
29. Що розміщено на вертикальній осі *User Story Mapping*?
30. Як *User Story Mapping* допомагає визначити *MVP*?
31. У чому перевага *User Story Mapping* порівняно з довгим списком *Product Backlog*?

Тема 6

ПРОТОТИПИ ЯК ЗАСІБ ЗМЕНШЕННЯ РИЗИКІВ

Створення прототипів програмного забезпечення робить вимоги більш реальними, наближує варіанти використання до «життя» та закриває пробіли у розумінні вимог. Прототипи надають користувачеві експериментальну модель та перший «зріз» нової системи, стимулюючи їх мислення та обговорення вимог.

Обговорення прототипів на ранніх стадіях процесу розробки допомагає зацікавленим в проєкті особам прийти до спільного розуміння вимог до системи, що зменшує ризик незадоволеності замовників.

Навіть при застосуванні методів розробки вимог, які вже були розглянуті раніше, окремі їх частини можуть бути невизначеними чи незрозумілими як для клієнтів, так і для розробників, або і тих і інших.

Якщо не виправити ці проблеми, різниця в очікуваннях між баченням продукту користувачами та розумінням розробників гарантована. Важко точно уявити поведінку нового програмного забезпечення при читанні текстової документації чи вивченні аналітичних моделей. Користувачі більш готові експериментувати з прототипом (це цікаво!!!), ніж читати специфікацію вимог до програмного забезпечення (це нудно).

Прототип має багато значень, тому очікування учасників процесу прототипування можуть досить розрізнитись. Прототип літака дійсно літає - адже це перший варіант справжнього літака. З іншої сторони, прототип програмного забезпечення - це тільки частина чи взірєць реальної системи – і він може в принципі не робити нічого корисного.

Прототипи програмного забезпечення можуть бути діючими моделями або його ескізами, наочною демонстрацією чи прикладами реальної функціональності, імітацією чи наслідуванням.

6.1. Що таке прототип програмного продукту і для чого він потрібний

Прототип (Prototype) - це спрощена, часто неповна версія майбутнього продукту, що створюється для перевірки ідей, вимог, взаємодії користувача з системою та можливих технічних рішень до початку повної розробки.

Прототип програмного забезпечення - це симуляція того, як реальний проєкт буде виглядати, працювати та сприйматися, але при цьому він не обов'язково має точну логіку, притаманну етапу розробки програмного забезпечення. Учасники команди розробників використовують його для проєктування зворотного зв'язку та тестування з користувачами. Прототипи бувають різних рівнів складності. Прототипування програмного забезпечення - це процес, який починається зі створення ідеї, її ескізування (скетчингу) та перетворення на клікабельний прототип, що імітує реальну програму.

Прототип може забезпечити симуляцію всього програмного забезпечення або мобільного додатка відповідно до вимог користувача. Власне, прототипування в розробці програмного забезпечення схоже на масштабовану макетну модель, яку використовують в архітектурному проєктуванні. У такому разі створення складного прототипу є необхідним, оскільки він дозволяє представити комплексний проєкт у доступному та зрозумілому вигляді. Це дає Замовникам можливість побачити точний план розробки програмного забезпечення і, ознайомившись із ним, надати зворотний зв'язок Виконавцю.

Перевага прототипу програмного забезпечення полягає в тому, що він є дешевшим і простішим для внесення змін, оскільки коригування можна виконувати під час процесу розроблення програмного забезпечення, а не після його завершення. Тому під час створення програмного продукту, зазвичай наперед, віддається перевага розробленню початкового прототипу, який можна використати для отримання зворотного зв'язку від Замовника проєкту.

Коли говориться про створення програмного забезпечення для Замовника з самого початку, процес розпочинається з етапу початкового планування. Команда розроблення програмного забезпечення формує ідею застосунку, яка ретельно продумується, щоб подальша реалізація була максимально простою та ефективною. Для забезпечення чіткого й послідовного підходу до роботи члени команди дотримуються процесу життєвого циклу розроблення програмного забезпечення (*SDLC*). Це дає змогу організувати діяльність команди, узгодити дії між її учасниками та систематизувати всі етапи створення програмного продукту.

Прототипи програмного забезпечення допомагають розробникам візуалізувати кінцевий продукт ще до того, як компанія-розробник почне інвестувати значні ресурси у його створення. Прототип не є точною копією майбутньої системи, а радше швидким макетом, який надає загальне уявлення про функціонування та зовнішній вигляд програмного продукту для команди розробників.

Ознайомившись із прототипом, команда може оцінити доцільність реалізації проєкту та визначити, чи варто інвестувати ресурси в його подальше розроблення. Крім того, оскільки прототипи формують більш чітке розуміння майбутньої системи, це сприяє створенню кінцевого продукту, який буде максимально зручним і зрозумілим для користувачів.

Прототипи дозволяють вирішувати три основні задачі:

- ✓ *пояснення та завершення процесу формування вимог* (використовуваний в якості інструменту формулювання вимог прототип являє собою попередню версію частини системи, розуміння якої викликає труднощі. Оцінка прототипу користувачами вказує на помилки в формулюванні вимог, котрі можна виправити без великих витрат до створення реального продукту);

- ✓ *дослідження альтернативних рішень* (прототип, як інструмент конструювання, дозволяє зацікавленим в проєкті особам досліджувати

різноманітні варіанти реалізації взаємодії користувачів, оптимізувати зручність роботи та оцінювати можливі технічні прийоми. Прототипи дозволяють на робочих зразках показати, наскільки здійсненні вимоги);

✓ *створення кінцевого продукту* (використовуваний в якості інструменту розробки прототип - це не що інше, як функціональна реалізація первинних елементів системи, котру можна перетворити в готовий продукт, здійснюючи послідовний ланцюг невеликих циклів розробки).

На рисунку 6.1. наведено узагальнену модель прототипування в процесі розроблення програмного забезпечення, яка демонструє послідовність етапів створення, оцінювання та вдосконалення прототипу перед реалізацією остаточної системи.



Рис.6.1. Узагальнена модель прототипування

Як видно з рисунка, процес починається з аналізу вимог (*Requirements Analysis*), на якому здійснюється збір та уточнення потреб Замовника, визначаються функціональні та нефункціональні вимоги до майбутнього

програмного продукту. На основі отриманої інформації переходять до етапу проєктування (*Design*), де формується початкова концепція системи, визначається її структура та основні характеристики інтерфейсу. Далі відбувається створення прототипу (*Prototyping*), який відображає ключові функції або елементи інтерфейсу майбутньої системи. Створений прототип передається на оцінювання користувачем або Замовником (*Customer Evaluation*). На цьому етапі зацікавлені сторони аналізують прототип, перевіряють його відповідність своїм очікуванням і надають зауваження або пропозиції щодо вдосконалення. Отримані відгуки враховуються під час етапу перегляду та вдосконалення (*Review and Refine*), у межах якого команда Виконавця вносить необхідні зміни до прототипу. Після цього процес знову повертається до етапу проєктування та прототипування, утворюючи ітераційний цикл, який повторюється доти, доки прототип не відповідатиме вимогам і очікуванням замовника. Після отримання схвалення Замовника (*Customer Approval*) розпочинається створення повноцінної системи. На етапі розроблення (*Develop*) реалізується функціональність програмного продукту, після чого проводиться тестування (*Test*) для перевірки коректності роботи системи та виявлення можливих помилок. Завершальним етапом є випуск або впровадження програмного продукту (*Release*).

Таким чином, представлена схема демонструє, що прототипування виступає важливим інструментом уточнення вимог і взаємодії із Замовником, дозволяючи зменшити ризики непорозуміння між сторонами, своєчасно виявляти недоліки концепції системи та підвищувати якість кінцевого програмного продукту.

6.2. Типи прототипів програмного забезпечення

Горизонтальний прототип *Behavioral Prototype*

Коли говорять про прототип програмного забезпечення, зазвичай мають на увазі горизонтальний прототип (*Horizontal Prototype*) пропонованого інтерфейсу користувача. Його також називають поведінковим прототипом (*Behavioral Prototype*) або моделю.

Горизонтальним прототип називається тому, що в ньому не реалізуються всі шари архітектури та нюанси системи, але втілюються деякі особливості інтерфейсу користувача. Він дозволяє користувачам досліджувати поведінку пропонованої системи в тих чи інших випадках для уточнення вимог, а також виявити чи зможуть вони за допомогою системи, основаної на прототипі, виконувати свою роботу.

Горизонтальний прототип *створює видимість функціональності, не забезпечуючи її дійсного втілення*. Він показує зовнішній вигляд екранів користувацького інтерфейсу та дозволяє здійснювати часткову навігацію між ними, але не містить ніякої або майже ніякої дійсної функціональності.

Горизонтальний прототип *демонструє функціональні можливості, котрі будуть доступні користувачу, зовнішній вид користувацького інтерфейсу* (колір, планування, графіку, елементи керування) та *структуру доступу до інформації* (структуру навігації).

Горизонтальний прототип не виконує корисної роботи, хоча іноді і здається, що повинен. Здебільшого, імітації достатньо, щоб користувачі могли вирішити чи немає певних упущень, неправильних або непотрібних функцій. Деякі прототипи висловлюють концепцію реалізації конкретних варіантів використання, як її бачать розробники. Оцінюючи прототип, користувачі можуть вказати на альтернативні можливості їх реалізації, пропущені кроки взаємодії чи додаткові умови виключень.

Горизонтальний прототип використовують, щоб:

- ✓ показати загальну концепцію системи (Які модулі будуть? Які розділи бачить користувач? Як виглядає структура меню, панелей, вкладок?);
- ✓ перевірити логіку навігації та сценарії верхнього рівня (Чи інтуїтивно користувач знаходить потрібний розділ? Чи немає «зайвих» екранів або заплутаних переходів?);
- ✓ узгодити бачення з бізнесом і стейкхолдерами (Чи все, що вони очікують, є в системі на верхньому рівні? Чи немає пропусків у функціональних областях?);
- ✓ оцінити обсяг системи (Скільки розділів / екранів / сторінок? Яка можлива складність реалізації?).

Вертикальний прототип *Structural Prototype*

Вертикальний прототип (*Vertical Prototype*) також званий структурним прототипом (*Structural Prototype*) чи перевіркою концепції, втілює зріз функціональності додатку від інтерфейсу користувача до сервісних функцій. Вертикальний прототип діє як справжня система, оскільки зачіпає всі рівні її реалізації.

Термін «вертикальний» означає, що прототип охоплює всі архітектурні шари системи:

- ✓ рівень представлення (*Presentation layer* (UI/UX)): екрани, форми, кнопки, навігація);
- ✓ прикладний рівень та рівень бізнес-логіки (*Application / Business logic layer*): правила обробки даних, перевірки, сценарії поведінки);
- ✓ рівень даних (*Data layer*): робота з БД (реальна або спрощена), API, інтеграції з іншими сервісами).

На відміну від горизонтального прототипу, який демонструє багато функцій поверхнево, вертикальний - одну або декілька функцій, але глибоко.

Вертикальний прототип створюють з метою перевірити технічну здійсненність ключового сценарію, зменшити архітектурні ризики, оцінити продуктивність і складність реалізації, перевірити логіку бізнес-процесів, а також отримати ранній зворотний зв'язок від стейкхолдерів.

Розробляти вертикальний прототип слід щоразу, коли є сумніви в здійсненності та стабільності пропонованого підходу до архітектури системи чи коли є ціль оптимізувати алгоритми, оцінити пропоновану схему бази даних або перевірити критично важливі тимчасові вимоги. Щоб отримати значні результати, вертикальні прототипи створюють, використовуючи засоби розробки, аналогічні середовищу розробки.

Одноразовий
прототип
Throwaway prototype

У сучасній розробці програмного забезпечення інформаційних систем і цифрових сервісів ключовою проблемою залишається невизначеність вимог. Замовники часто описують ідею на рівні бачення, користувачі не завжди усвідомлюють власні потреби, а команда розробки змушена інтерпретувати ці очікування, спираючись на власний досвід і припущення. Саме тому помилки виникають на самому початку розроблення програмного забезпечення (неправильна логіка системи, незручний інтерфейс, зайві або відсутні функції), а згодом переростають у високовартісне їх усунення.

Одноразовий (відкідний) прототип (Throwaway Prototype) - це тимчасова модель системи або її окремої частини, яка створюється з єдиною метою - краще зрозуміти вимоги. Його не планують використовувати у фінальній версії продукту, не оптимізують і не підтримують у довгостроковій перспективі. Важливо підкреслити, що слово «одноразовий» не означає «неякісний», а навпаки, такий прототип має бути достатньо зрозумілим і наочним, щоб користувач міг взаємодіяти з ним і висловити аргументовану думку.

Команда Виконавця робіт з розробки програмного забезпечення швидко створює одноразовий прототип, щоб продемонструвати, як початкові вимоги системи можуть виглядати у візуальній формі. В такому прототипі зворотний зв'язок від Замовника або користувачів відіграє ключову роль, оскільки саме він визначає необхідність внесення змін до вимог. Після отримання відгуків користувачів прототип переглядається або створюється заново, щоб краще відповідати потребам зацікавлених сторін. Зазвичай такі прототипи використовуються лише для уточнення вимог і після цього відкидаються, не стаючи частиною майбутньої системи.



Рис.6.2. Схема життєвого циклу з одноразовим прототипом

Як видно зі схеми, прототип не є частиною подальших етапів. Він виконує роль містка між ідеєю та чіткими вимогами.

Найчастіше одноразовий прототип *фокусується не на технічній реалізації, а на зовнішній поведінці системи*. Це дозволяє користувачу оцінити, чи відповідає система його очікуванням. Зазвичай моделюються екрани інтерфейсу, навігація між екранами, базова логіка сценаріїв користувача та повідомлення системи.

Нижче подано умовний графік, який ілюструє, чому одноразові прототипи вигідні.

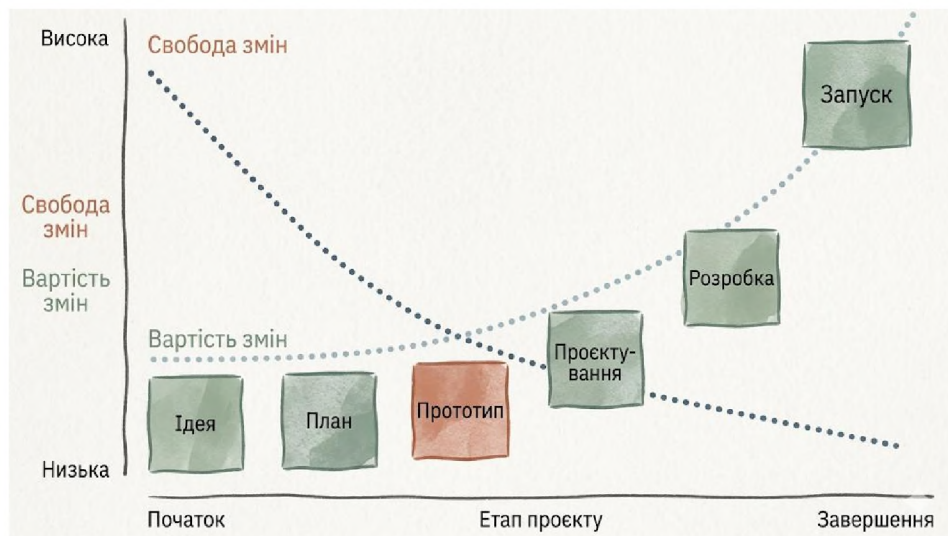


Рис.6.3. Графік впливу одноразового прототипу на вартість змін

Дана діаграма ілюструє загальновизнаний принцип у проєктному менеджменті та розробці продуктів, відомий як крива вартості змін (*Cost of Change Curve*). Вона показує, що з плином часу та проходженням різних фаз проєкту вартість внесення змін зростає експоненціально, тоді як свобода змін зменшується. Свобода змін (*Freedom of change*) на початку проєкту це етапи *Idea*, *Plan*, *Prototype*, коли вимоги ще не затверджені, а роботи мінімальні, вносити зміни легко та недорого. Вартість змін (*Cost of change*) на пізніших етапах це етапи *Design*, *Develop*, *Launch*, коли рішення вже реалізовані в архітектурі, коді або фізичній структурі, виправлення помилок або додавання нових функцій вимагає переробки значної кількості роботи, що робить цей процес дуже дорогим. Ця діаграма підкреслює важливість ретельного планування та виявлення потенційних проблем якомога раніше в життєвому циклі проєкту, оскільки це значно економить час і ресурси.

Для створення одноразових прототипів часто використовують найпростіші засоби. Це може бути папір і маркер, дошка, або ж цифрові інструменти для створення *wireframe*-ів. Ключовим є не інструмент, а швидкість

і наочність. Важливо свідомо уникати надмірної деталізації, щоб команда не витратила час на те, що все одно буде викинуто.

Однією з найпоширеніших помилок у створенні одноразового прототипу є ситуація, коли Замовник починає сприймати одноразовий прототип як майже готову систему. У такому разі виникає спокуса «трішки доробити» прототип і перетворити його на фінальний продукт, що суперечить самій ідеї підходу. Інший ризик - створення занадто складного прототипу, який починає жити власним життям і відволікає команду від головної мети: розуміння вимог.

Порівняння одноразового прототипу, MVP та PoC

У практиці проектування та розробки цифрових продуктів одноразовий прототип часто плутають з такими поняттями, як мінімально життєздатний продукт (*Minimum Viable Product, MVP*) та підтвердження концепції (*Proof of Concept, PoC*). Хоча всі три підходи спрямовані на зменшення ризиків, вони застосовуються на різних етапах і мають різну мету.

Одноразовий прототип орієнтований насамперед на *уточнення вимог і перевірку розуміння потреб користувача*. Він *не призначений для реального використання і не несе бізнес-цінності* сам по собі. Його цінність полягає у знанні, яке отримує команда після взаємодії з прототипом.



Мінімально життєздатний продукт (*MVP*), на відміну від одноразового прототипу, є *працюючим продуктом*, який запускається на ринок. Він містить мінімальний набір функцій, достатній для того, щоб користувач міг отримати цінність, а команда - зібрати реальні метрики та зворотний зв'язок. Код *MVP* зазвичай еволюціонує у фінальну систему.

Підтвердження концепції (*PoC*) *фокусується не на користувачеві, а на технічній здійсненності* ідеї, тобто метою є довести, що певне технологічне рішення або підхід працює в принципі. *PoC* може взагалі не мати інтерфейсу і не демонструватися кінцевим користувачам.

Таким чином, одноразовий прототип доцільно застосовувати до *MVP* і *PoC*, коли команда ще формує бачення продукту та намагається уникнути концептуальних помилок.

Еволюційний прототип *Evolutionary Prototype*

Еволюційний прототип (Evolutionary Prototype) - це підхід до проектування та розробки інформаційних систем, коли прототип не відкидається після демонстрації, а поступово розвивається в повноцінний продукт. На відміну від одноразового прототипу, еволюційний прототип є «живим артефактом», тобто він *змінюється, ускладнюється, оптимізується та зрештою стає основою фінальної системи*. У сучасних ІТ-проектах, де вимоги часто є неповними або змінюються під впливом бізнесу, користувачів та ринку, еволюційний прототип дозволяє зменшити невизначеність і ризики, зберігаючи водночас швидкість розробки.

Еволюційний прототип створюється не як допоміжний або тимчасовий артефакт, а як *початкова версія майбутньої системи*, що з самого початку має практичну цінність.

На ранньому етапі еволюційний прототип зазвичай охоплює обмежений набір функцій, які відображають ключову бізнес-цінність системи. Він може мати спрощений дизайн, мінімальні нефункціональні характеристики та неповну бізнес-логіку, проте вже дозволяє користувачам виконувати реальні дії.

Еволюційний прототип є ітераційним процесом поступового вдосконалення початкової версії прототипу, який складається з таких етапів. Спочатку формується початкове проєктне рішення системи на основі вимог

(*Design*), на основі якого створюється первинний прототип (*Build prototype*), що демонструє основні функції або інтерфейс майбутньої системи. Після цього проводиться тестування прототипу користувачами або Замовником (*User tests prototype*) для оцінювання його зручності та відповідності очікуванням та надання користувачами зворотного зв'язку (*Feedback provided*) щодо функціональності, інтерфейсу та можливих покращень для подальшого доопрацювання та вдосконалення прототипу (*Refine prototype*) з урахуванням отриманих зауважень.



Рис.6.4. Ілюстрація ітераційного процесу еволюційного прототипування

Після цього процес може повторюватися декілька разів, утворюючи цикл вдосконалення. У результаті прототип поступово еволюціонує, набуваючи нових функцій і характеристик. На відміну від відкидного прототипу, еволюційний прототип не відкидається, а поступово перетворюється на кінцевий програмний продукт (*Final Product*).

У процесі використання такого прототипу команда отримує безпосередній зворотний зв'язок. Цей зворотний зв'язок не лише уточнює вимоги, а й змінює

бачення продукту загалом. Таким чином, система еволюціонує разом із розумінням потреб користувачів та бізнесу.

Важливою рисою еволюційного прототипу є те, що кожна ітерація додає цінність, а не просто виправляє помилки попередньої. Прототип поступово набуває стабільності, масштабованості та відповідності стандартам якості, перетворюючись із експериментального рішення на промисловий продукт.

Еволюційний прототип органічно вписується в ітеративні та інкрементальні моделі життєвого циклу програмного забезпечення. На відміну від каскадного підходу, де кожен етап має чіткі межі, тут межі між аналізом, проєктуванням, розробкою та тестуванням є умовними. Фактично еволюційний прототип поєднує ці етапи в єдиний безперервний процес. Аналіз вимог відбувається паралельно з розробкою, а тестування - разом із використанням системи кінцевими користувачами. Це дозволяє швидко виявляти невідповідності між очікуваннями та реальною поведінкою системи.

У такій моделі життєвого циклу запуск продукту може відбуватися кілька разів: спочатку у вигляді внутрішньої версії, потім - пілотного рішення, і зрештою - повноцінного релізу. Кожен запуск є частиною еволюції системи, а не фінальною точкою проєкту.

На відміну від одноразового прототипу, який використовується переважно для дослідження вимог або інтерфейсу і після цього відкидається, еволюційний прототип, навпаки, розробляється з урахуванням того, що код, архітектура та рішення залишаться у фінальній системі.

Еволюційний прототип створює постійний канал комунікації з користувачами. Замість довгих специфікацій вони отримують робочий продукт, який можна оцінити, протестувати та критикувати.

Еволюційний прототип особливо ефективний у середовищах з високою невизначеністю вимог. Водночас він вимагає дисципліни команди, постійного рефакторингу та контролю технічного боргу. За правильного застосування цей підхід дозволяє створювати системи, які краще відповідають реальним потребам бізнесу та користувачів.

Порівняння еволюційного прототипу, MVP та Agile-інкременту

Концепції еволюційного прототипу, *MVP* та *Agile*-інкременту часто плутають, оскільки вони мають спільну мету - швидкий вихід на ринок. Проте вони мають фундаментальні відмінності в роботі з кодом і продуктом в подальшому.

На відміну від «одноразового» прототипу, про який ми говорили раніше, еволюційний - це фундамент майбутньої будівлі. Все починається з розробки найбільш зрозумілої частини системи. Коли впевнились, що вона працює, код не відкидається, а на нього «нарощуються» нові функції, тобто код «живе» від початку до кінця. Ризик створення: якщо архітектура на початку була закладена неправильно, з кожним новим рівнем вона ставатиме все слабшою, що може призвести до краху системи.

MVP - це найпростіша версія продукту, яка дозволяє команді зібрати максимум перевіреної інформації про клієнтів з мінімальними зусиллями. *MVP* може бути еволюційним прототипом, а може бути просто лендингом (односторінковим сайтом) з кнопкою «Купити», якої ще не існує. *MVP* створюється для перевірки гіпотези - чи потрібен цей продукт людям взагалі? Ризик створення:



витратити забагато часу на «ідеальну» розробку функцій, які насправді не потрібні ринку.

В методології гнучкої розробки *Agile* інкремент - це повністю готова, протестована і працездатна частина продукту, яка додається до попередніх частин після кожного спринту. Наприклад, у першому місяці зробили «Логін», у другому - «Профіль», у третьому - «Платежі». Кожен інкремент має бути такої якості, щоб його можна було одразу випускати в реліз. Це постійне постачання цінності користувачеві маленькими, але завершеними порціями. Ризик створення: можна так захопитися деталями окремих частин і втратити бачення цілісного продукту.

Як бачимо, різниця між еволюційним прототипом та інкрементом тонка, але фундаментальна. Вона полягає в меті та стані готовності коду.

Логіка еволюційного прототипу: «Ми будуємо, щоб зрозуміти, що ми будуємо». Це процес дослідження, оскільки система будується поступово, оскільки на самому початку не відомо точно, як вона має працювати (спочатку пишемо код, показуємо користувачу, отримуємо відгук і переробляємо цей самий код далі).

Логіка *Agile*-інкременту: «Ми знаємо, що будуємо, і здаємо це частинами, готовими до використання». Це процес постачання, оскільки система будується частинами, щоб давати цінність якомога раніше. Кожен інкремент - це повністю протестований, задокументований та стабільний код, який має бути придатним до релізу (якщо зробили функцію «Реєстрація», то вона має працювати на 100% надійно).

Паперові та
електронні
прототипи

У сучасному циклі розробки програмного забезпечення етап проектування інтерфейсів займає центральне місце, виступаючи містком між абстрактною ідеєю та функціональним кодом. Прототипування - це не просто

створення макета, а специфічна форма комунікації всередині команди та з кінцевим користувачем. Розглянемо два фундаментальні підходи, які зазвичай послідовно змінюють один одного: паперове (*Low-fidelity*) та електронне прототипування (*High-fidelity*).

Паперовий
прототип –
сила низької
деталізації

Паперове прототипування є першим кроком у візуалізації майбутньої системи. Паперові прототипи допомагають встановити, чи дійсно користувачі та розробники однаково розуміють вимоги. Вони дозволяють спробувати, практично не ризикуючи, зробити крок при розробці коду продукту. Схожий метод, названий *методом розкадровки (Storyboard)*, показує пропонувані інтерфейс користувача, без залучення користувачів до роботи з ним.

Цей метод передбачає використання звичайного паперу, олівців, стікерів та ножиць для створення ескізів екранів. Основна перевага цього підходу полягає в його швидкості та доступності. Коли дизайнер малює інтерфейс від руки, він не відволікається на вибір ідеальних кольорів, шрифтів чи відступів. Замість цього увага фокусується на інформаційній архітектурі та користувацьких сценаріях.



Процес тестування паперового прототипу нагадує театральну виставу. Один учасник команди бере на себе роль «комп'ютера», маніпулюючи шматочками паперу (меню, спливаючими вікнами), тоді як реальний користувач «натискає» на намальовані кнопки пальцем. Це дозволяє виявити логічні провали в навігації ще до того, як буде створено перший рядок цифрового проєкту. Низька якість виконання тут працює на користь: користувачі сміливіше критикують

грубі ескізи, ніж досконалі цифрові макети, оскільки розуміють, що дизайнер не витратив на цей малюнок багато часу.

Якими б ідеальними не були інструменти прототипування, нариси екранів на папері робити все рівно швидше. Паперове прототипування пришвидшує кожний цикл, а це - ключовий фактор успіху при розробці вимог. Це чудовий метод уточнення вимог для проєктування деталізованих інтерфейсів користувачів, створення еволюційних прототипів чи традиційного проєктування та конструювання. Він також допомагає команді краще керувати очікуваннями клієнтів.

Електронний
прототип – міст
до реального
продукту

Коли основна логіка підтверджена на папері, розробка переходить до електронного етапу. Електронні прототипи варіюються від статичних каркасів (*Wireframes*) до повноцінних інтерактивних моделей, створених у спеціалізованих програмах зразку *Figma* чи *Adobe XD*. На цьому етапі до гри вступає візуальна ієрархія. Дизайнер визначає точні розміри елементів, тестує колірні схеми та перевіряє доступність інтерфейсу для будь-якого типу користувача.

Електронний прототип дозволяє імітувати реальний досвід взаємодії з пристроєм. Тут схематичні квадрати паперового прототипу замінюються на повноцінні блоки із зображеннями і орієнтовним контентом. Переходи між екранами, анімація кнопок, прокрутка списків - усе це дає розробникам чітку специфікацію, а Замовнику - реалістичне уявлення про фінальний продукт. Важливо розуміти, що сучасні інструменти електронного прототипування дозволяють створювати «високоточні» моделі, які зовні майже не відрізняються від готового додатка, хоча під ними все ще немає реальної бази даних чи складної логіки обробки запитів.

Для створення електронних прототипів використовуються онлайн-сервіси або спеціалізоване програмне забезпечення. Макет включає повну ієрархію сторінок, завдяки цьому можна реалізувати детальний прототип програмного продукту.

Застосування таких макетів дозволяє оцінити обсяг роботи і визначитися із структурою. Тому вони в першу чергу потрібні Виконавцю.

Порівнюючи ці два методи, важливо бачити в них не конкурентів, а взаємодоповнюючі інструменти. Паперове прототипування виграє на етапі мозкового штурму, коли ідеї змінюються щохвилини. Воно забезпечує максимальну інклюзивність, адже для малювання на папері не потрібно знати складне програмне забезпечення - у процесі може брати участь і бізнес-аналітик, і клієнт. Проте воно має суттєві обмеження у передачі динаміки. На папері важко адекватно протестувати складні жести або адаптивність інтерфейсу під різні розміри екранів.

Електронні прототипи, навпаки, вимагають більше часу та спеціальних навичок, але вони є незамінними для віддаленого тестування та презентацій Замовнику. Електронна модель дозволяє зібрати реальну метрику: скільки секунд користувач шукав кнопку або на якому етапі він припинив виконання сценарію. Окрім того, електронні макети є живим документом для розробників, з якого вони можуть експортувати іконки, кольори та параметри шрифтів, що значно прискорює етап програмування.

Ефективний процес створення цифрового продукту зазвичай передбачає перехід від низької до високої точності. Починаючи з паперу, команда валідує ідеї та виправляє фундаментальні помилки логіки. Переходячи до електронних інструментів, вона шліфує деталі та готує ґрунт для написання коду. Розуміння того, коли варто відкласти стилус і взяти олівець (і навпаки), є ознакою зрілості

дизайнера та запорукою створення ергономічного, доступного та успішного програмного забезпечення.

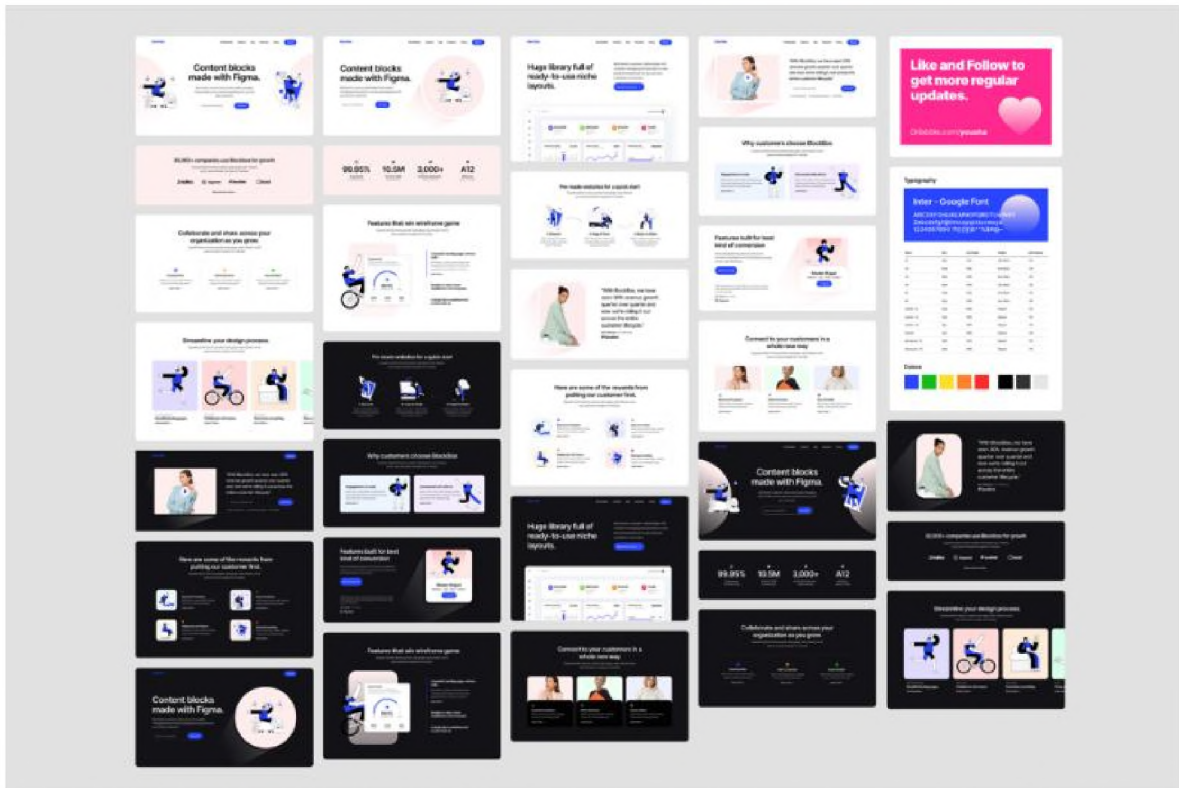


Рис.6.5. Приклад High-fidelity прототипу сайту, створеного у Figma

6.3. Ризики прототипування

Не дивлячись на те, що прототипування зменшує ризик невдачі проєкту з розробки програмного забезпечення, воно володіє власними ризиками. Найбільший з них полягає у тому, що будь-яка із зацікавлених в проєкті сторін, побачивши діючий прототип, вирішить, що кінцевий продукт майже готовий. «ОГО, здається мені, що ви вже все зробили! – з ентузіазмом говорить той, кого ви попросили оцінити прототип. – Виглядає прекрасно. Можливо, ви швиденько доробите його і віддасте мені?» Якщо коротко, то: НІ! Одноразовий прототип ні в якому випадку не призначений для роботи, як би він не був схожий на готовий продукт. Це всього лише модель, зразок, експеримент. Не слід дозволяти страхам, пов'язаними із передчасним випуском продукту, відвернути від створення

прототипу. Слід пояснити зацікавленим сторонам, що його неможливо випустити як кінцевий продукт. Один із способів контролювати цей ризик - використовувати паперові, а не електронні прототипи. Ні одному з тих, хто оцінює паперовий прототип, і в голову не прийде думка, що продукт вже майже готовий. Ще один спосіб – обрати інструмент прототипування, який відрізняється від тих, які застосовують для розробки кінцевого продукту. Це допоможе протистояти тим, хто просить «швидко завершити» та випустити прототип. Залишивши зовнішній вигляд прототипу невідлагодженим і незавершеним ви також зменшите цей ризик.

Слід остерігатись також, коли користувачів починає турбувати запитання «Як?»: як інтерфейс користувача буде виглядати і діяти. Працюючи із прототипами, зовнішньо схожими на кінцевий продукт, користувачі легко забувають, що на стадії уточнення вимог вони повинні загалом думати, що ж вони хочуть бачити в системі. Тому доцільно обмежити прототип лише тими демонстраціями, функціями та можливостями навігації, котрі допоможуть забрати невизначеності у вимогах.

Третій ризик полягає в тому, що користувачі почнуть робити висновки про продуктивність кінцевого продукту по продуктивності прототипу. Якщо користувачі побачать, що прототип миттєво реагує на модельований запит до бази даних, використовуючи жорстко закодовані результати запиту, вони можуть очікувати такої ж високої продуктивності від продукту, який включає величезну розподілену базу даних. Слід подумати про реалізації в прототипі часових затримок, щоб модель очікуваної поведінки кінцевого продукту виглядала більш реалістичною (а прототип – менш готовим для реалізації).

Наостанок, необхідно остерігатись дій по прототипуванню, які вимагають таких зусиль, що команда розробників виб'ється із графіку і буде вимушена випустити прототип у якості готового продукту чи спішно і безсистемно

добробляти продукт. Відноситись до прототипу слід як до експерименту. Перевіряти, що вимоги достатньо визначені, і що ключові аспекти взаємодії людини і комп'ютера, а також всі архітектурні питання вирішені так, що можна переходити до проєктування та конструювання. Можливостей прототипу повинно бути рівно стільки, скільки необхідно для перевірки гіпотез, відповідей на запитання та уточнення розуміння вимог.

6.4. Фактори успіху прототипування

Успішне застосування прототипування в процесі розроблення програмного забезпечення залежить від низки організаційних, технічних та комунікаційних чинників. Вони забезпечують ефективне використання прототипів для уточнення вимог, зменшення ризиків та підвищення якості майбутнього програмного продукту.

Одним із ключових факторів є чітке визначення мети створення прототипу. Перед початком роботи команда повинна розуміти, для чого саме створюється прототип: для уточнення вимог, перевірки концепції системи, оцінювання інтерфейсу користувача або тестування певних технічних рішень. Чітке формулювання мети дозволяє уникнути зайвих витрат часу та ресурсів.

Важливим фактором також є активна участь користувачів і Замовника у процесі оцінювання прототипу. Регулярний зворотний зв'язок від зацікавлених сторін допомагає своєчасно виявляти неточності у вимогах, коригувати функціональні можливості системи та забезпечувати відповідність продукту реальним потребам користувачів.

Не менш важливою є ітераційність процесу прототипування. Створення прототипу зазвичай передбачає кілька циклів розроблення, оцінювання та

вдосконалення. Такий підхід дозволяє поступово уточнювати вимоги, перевіряти різні варіанти реалізації та вдосконалювати концепцію системи.

Ще одним важливим фактором є простота та швидкість створення прототипу. Прототип не повинен бути надто складним або детально реалізованим, оскільки його основною метою є демонстрація ключових ідей і функцій системи. Використання спеціалізованих інструментів прототипування дозволяє швидко створювати макети інтерфейсів і демонструвати основні сценарії взаємодії користувача з системою.

Крім того, важливу роль відіграє ефективна комунікація між Виконавцем та Замовником. Чітке обговорення результатів оцінювання прототипу, документування зауважень і пропозицій сприяють узгодженню вимог та прийняттю обґрунтованих рішень щодо подальшого розвитку системи.

Отже, щоб зробити прототип ефективною частиною процесу зіставлення вимог слід виконувати наступні рекомендації:

- ✓ включати задачі прототипування в план проєкту (скласти графік розподілу витрат часу і засобів на розробку, оцінку та модифікацію прототипів);
- ✓ сформулювати ціль кожного прототипу до початку його створення;
- ✓ планувати створення декількох прототипів (в більшості випадків не вдасться створити те, що потрібно, з першої спроби, в чому, загалом, і полягає зміст прототипування);
- ✓ одноразові прототипи створити так швидко і дешево, як можливо (вкласти мінімум зусиль в підготовку прототипів, котрі допоможуть відповісти на запитання чи вирішувати незрозумілості у вимогах, не доводити до бездоганності одноразові прототипи);
- ✓ не вбудовувати в одноразовий прототип детальну перевірку вхідних даних, методи безпечного програмування, процедур обчислень помилок чи документування коду;

- ✓ не прототипувати елементи, котрі вже зрозумілі, крім випадків дослідження альтернатив дизайну;
- ✓ не використовувати правдоподібні дані в екранних формах та повідомленнях прототипу (користувачі будуть відволікатись і не зможуть сприйняти прототип як модель, котра описує зовнішній вигляд та поведінку реальної системи);
- ✓ не очікувати, що прототип повністю замінить специфікацію вимог до програмного забезпечення (більша частина прихованої функціональності лише мається на увазі в прототипі і повинна бути задокументована в специфікацію вимог до програмного забезпечення, щоб її вдалось реалізувати повноцінно, точно та прозоро).

Видима частина додатку - це так би мовити вершина айсбергу. Зображення екранів не дають деталей визначення полів даних, критеріїв перевірки, взаємозв'язків між полями (таких, як елементи керування користувацьким інтерфейсом, які з'являються лише, якщо користувач здійснює певні дії з іншими елементами керування), обробки виключень, бізнес-правил тощо.

6.5. Інструменти створення електронних прототипів

Електронний прототип дозволяє перетворити абстрактну вимогу або концепцію на форму, з якою можна взаємодіяти, обговорювати її, тестувати та вдосконалювати ще до етапу розробки.

Інструменти створення електронних прототипів - це не лише програмне забезпечення, це середовище, у якому поєднуються аналітика, дизайн, логіка бізнес-процесів та користувацький досвід. Саме тому вибір і правильне використання таких інструментів безпосередньо впливає на якість продукту, швидкість прийняття рішень та рівень ризиків у проєкті.

Електронний прототип - це цифрове представлення майбутнього продукту, яке імітує структуру, навігацію, логіку взаємодії та, у деяких випадках, візуальний стиль системи, який масштабується, легко редагується, дозволяє створювати інтерактивні сценарії, може використовуватися для юзабіліті-тестування та стає спільною мовою для дизайнерів, аналітиків, розробників і Замовників.



Цифрові інструменти дають змогу працювати ітеративно, зберігати історію змін, швидко перевіряти гіпотези та демонструвати альтернативні рішення без значних витрат.

Low-fidelity прототипи - це прості, схематичні моделі майбутньої програмної системи або інтерфейсу користувача, які створюються на ранніх етапах розроблення для швидкої перевірки ідей, структури системи та основних сценаріїв взаємодії користувача з програмним продуктом. Такі прототипи не мають детальної візуальної реалізації і зазвичай відображають лише базову логіку розташування елементів інтерфейсу та послідовність дій користувача.

Low-fidelity прототипи можуть створюватися у вигляді ескізів, схем, паперових макетів або простих каркасних моделей інтерфейсу (*Wireframes*). У практиці бізнес-аналізу та *UX*-дизайну *Low-fidelity* прототипи часто створюються навіть під час зустрічі як спільний результат обговорення.

Для створення каркасних моделей інтерфейсу широко використовуються цифрові інструменти. Одним із найпоширеніших інструментів є *Figma*, який

дозволяє створювати прості макети інтерфейсів, організовувати спільну роботу команди та швидко демонструвати структуру майбутнього застосунку.

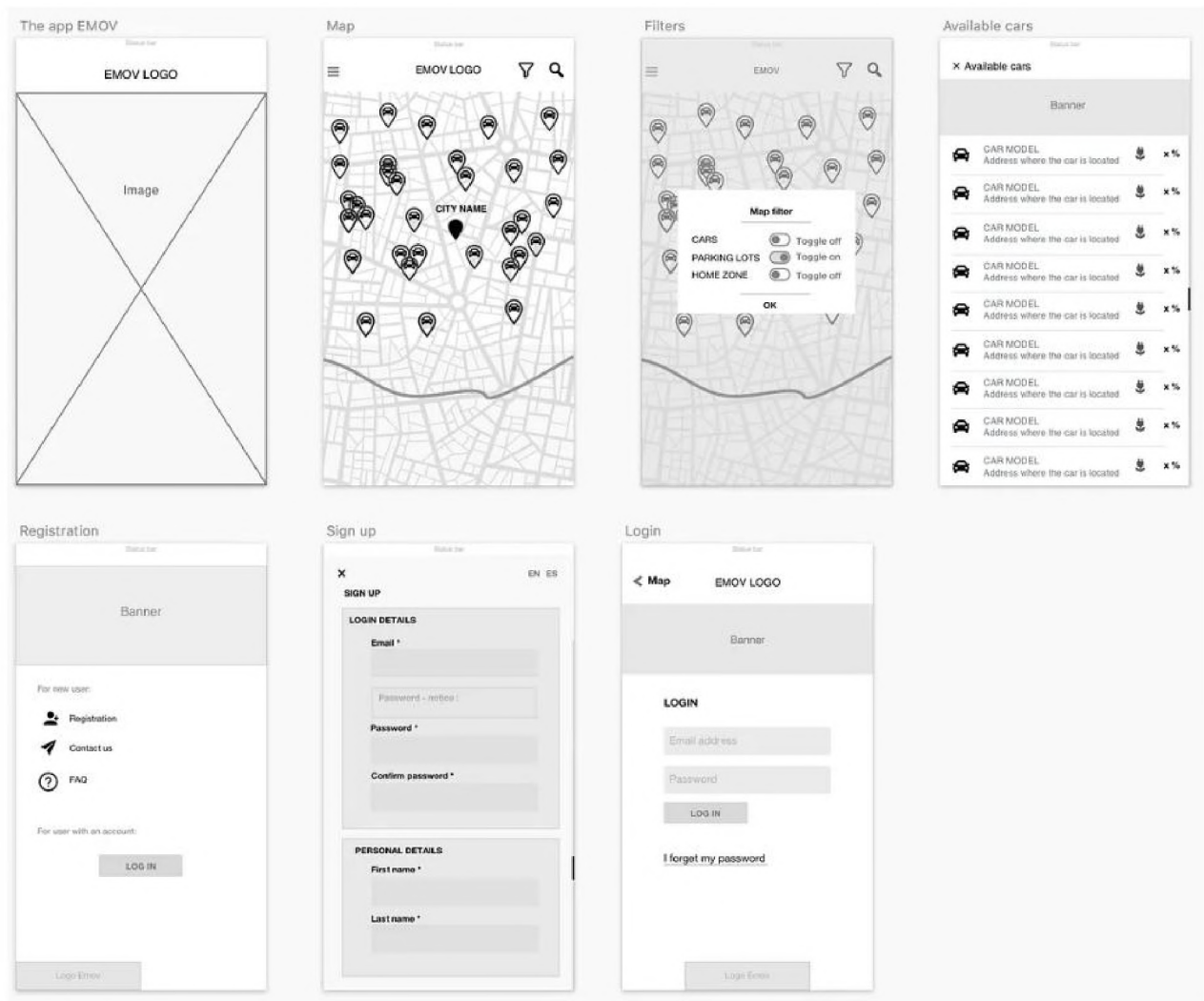


Рис. 6.6. Приклад Wireframes для простого у використанні додатку, щоб у режимі реального часу бачити доступні в околі автомобілі та швидко бронювати автомобіль

Ще одним популярним інструментом є *Balsamiq*, що спеціально призначений для створення *Low-fidelity* макетів. Його інтерфейс імітує вигляд намальованих від руки ескізів, що допомагає зосередитися на структурі інтерфейсу, а не на деталях дизайну.

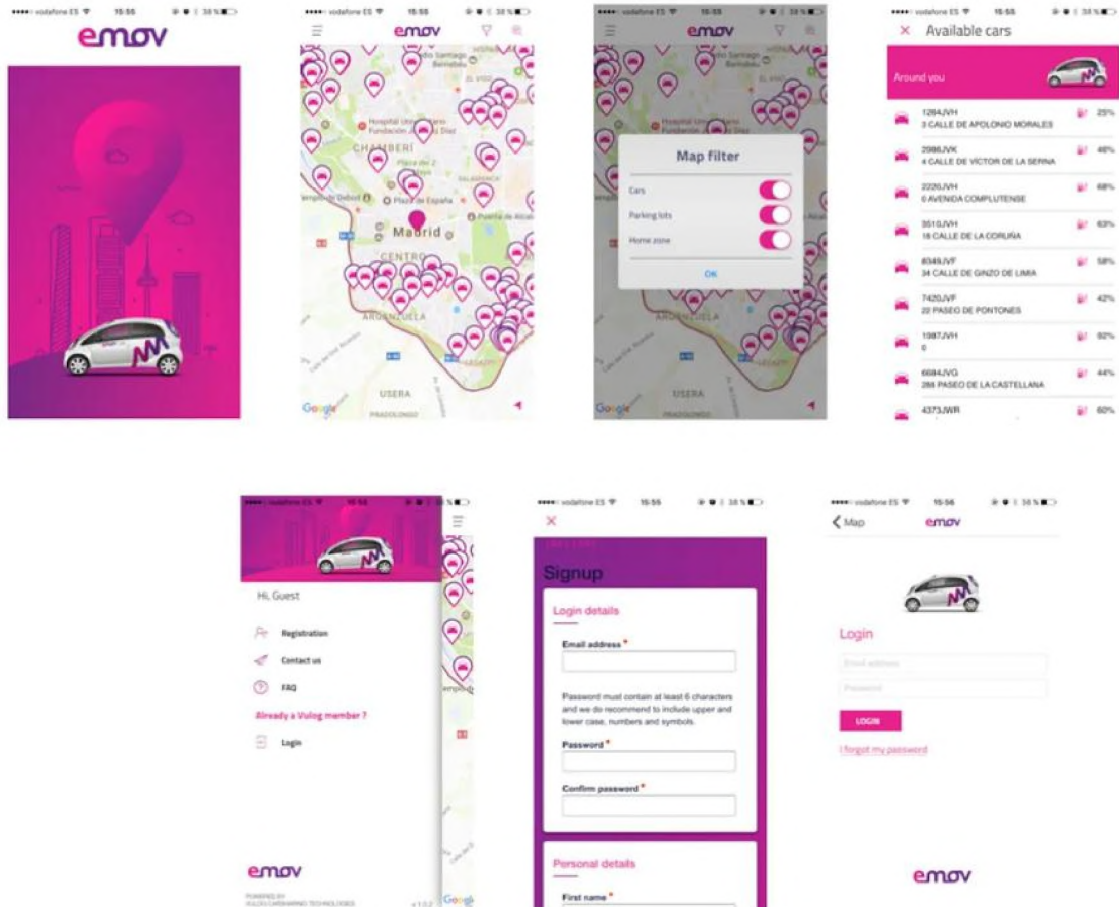
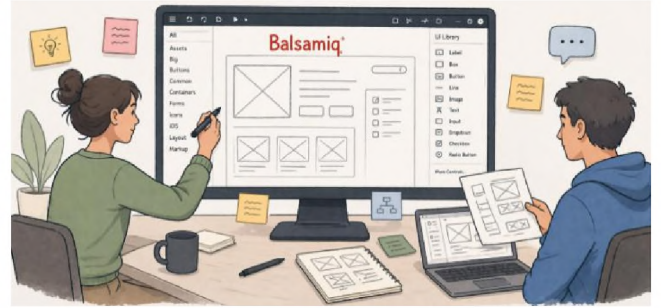


Рис. 6.7. Скріни простого у використанні додатку, щоб у режимі реального часу бачити доступні в околі автомобілі та швидко бронювати автомобіль

Для створення простих схем і каркасних моделей також використовують *Miro* - онлайн-дошку для командної роботи, яка дозволяє спільно створювати ескізи, схеми та прототипи. Інструмент *Lucidchart* також застосовується для

моделювання структури інтерфейсу та візуалізації взаємозв'язків між екранами системи.

Крім спеціалізованих засобів, для створення простих прототипів іноді використовують універсальні інструменти, такі як *Microsoft PowerPoint* або *Microsoft Visio*, які дозволяють швидко створювати схеми екранів і демонструвати логіку переходів між ними.

Сучасні інструменти зі штучним інтелектом також активно використовуються для створення *Low-fidelity* прототипів, оскільки вони дозволяють швидко генерувати структуру інтерфейсу, базові екрани застосунку та сценарії взаємодії користувача із системою. Одним із популярних інструментів є *Uizard*. Платформа використовує штучний інтелект для автоматичного створення каркасних моделей інтерфейсу на основі текстових описів або намальованих ескізів. *AI* може розпізнавати намальовані від руки макети та перетворювати їх у цифрові прототипи. Ще одним інструментом є *Whimsical*, який дозволяє створювати прості каркасні моделі інтерфейсів, схеми та карти користувацьких сценаріїв, в якому *AI*-функції допомагають швидко генерувати структуру інтерфейсу та пропонувати варіанти організації елементів.

Для створення простих прототипів і схем *Miro* також підтримує *AI*-можливості для генерування ідей, створення структури інтерфейсів і моделювання користувацьких сценаріїв. Крім того, і у середовищі *Figma* доступні *AI*-плагіни, які дозволяють швидко створювати каркасні макети інтерфейсу, генерувати структуру екранів і пропонувати базові компоненти інтерфейсу.

High-fidelity прототипи - це деталізовані моделі майбутньої програмної системи, які максимально наближені до вигляду та поведінки кінцевого програмного продукту. На відміну від *Low-fidelity* прототипів, вони відображають не лише структуру інтерфейсу, але й реалістичний дизайн,

кольорову гаму, типографіку, інтерактивні елементи та логіку взаємодії користувача із системою.

Для створення *High-fidelity* прототипів використовують спеціалізовані інструменти для дизайну інтерфейсів та прототипування. Одним із найбільш поширених інструментів є *Figma*, який дозволяє створювати детальні макети інтерфейсів, налаштовувати інтерактивні переходи між екранами та організовувати спільну роботу команди.

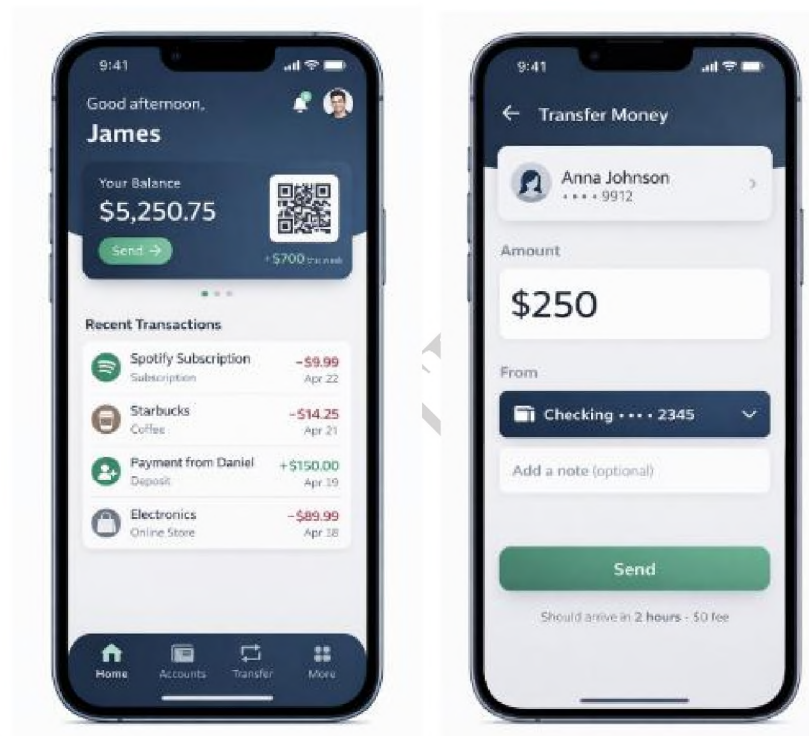


Рис. 6.8. *High-fidelity* прототипи головного екрану та екрану переказу коштів мобільного банкінгу

Ще одним популярним інструментом є *Adobe XD*, що використовується для розроблення високодеталізованих прототипів, тестування сценаріїв взаємодії користувача з інтерфейсом і створення інтерактивних демонстрацій програмного продукту. Для створення складних дизайнів і прототипів інтерфейсів також застосовують *Sketch*, який широко використовується у сфері *UX/UI*-дизайну для розроблення інтерфейсів веб- і мобільних застосунків. Крім того, інструмент

InVision дозволяє перетворювати статичні макети інтерфейсів на інтерактивні прототипи та проводити тестування користувацького досвіду.

Насьогодні існує багато інструментів зі штучним інтелектом, які допомагають створювати *High-fidelity* прототипи значно швидше. Такі інструменти можуть автоматично генерувати макети інтерфейсів, пропонувати дизайн екранів, створювати компоненти або навіть перетворювати текстовий опис у готовий прототип. Одним із найвідоміших інструментів є *Figma*, який має *AI*-плагіни та функції генерації дизайну та вже згадуваний *Uizard*. Інструмент *Galileo AI* також призначений для генерації дизайнів інтерфейсів. Він здатний створювати високодеталізовані *UI*-макети на основі текстового запиту, що значно пришвидшує процес створення прототипів.

Прототип стає особливо цінним, коли з ним можна взаємодіяти. Інтерактивні інструменти дозволяють створювати переходи між екранами, умовну логіку, анімацію, а також симуляцію реальних сценаріїв користувача.

Такі прототипи часто використовують у юзабіліті-тестах, де важливо спостерігати поведінку, а не оцінки «подобається/не подобається». Серед поширених рішень - *InVision*, *Axure*, *ProtoPie*, *Marvel*.

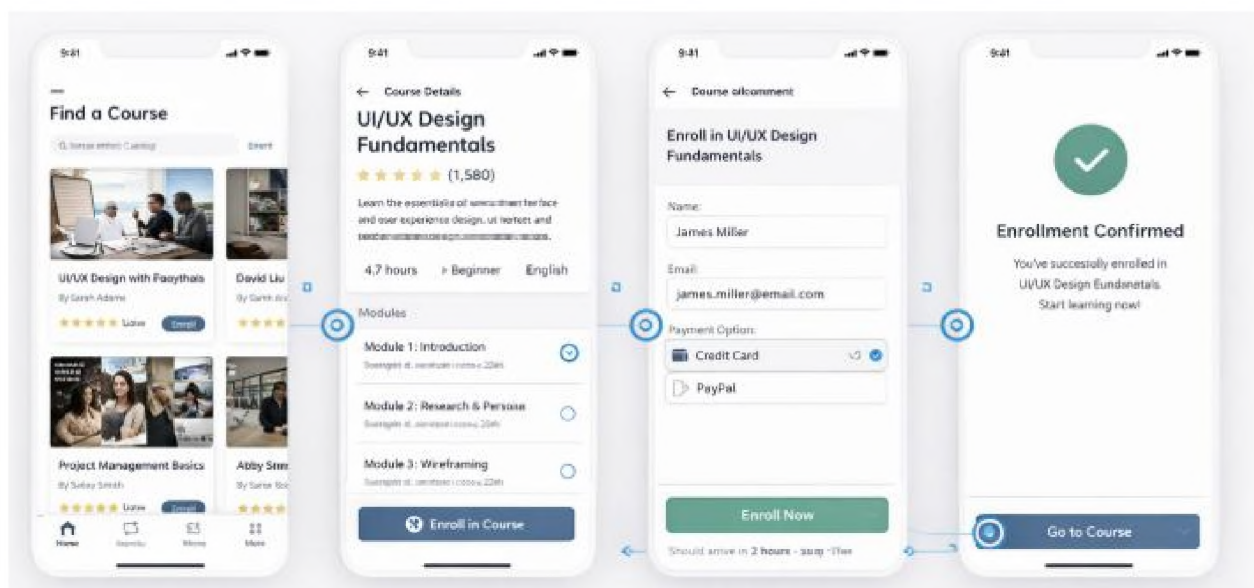


Рис. 6.9. Приклад інтерактивного UX-потoku для платформи онлайн-курсів

Сучасні *AI*-інструменти дозволяють створювати екрани за текстовим описом, варіанти дизайну, ілюстрації для прототипів, фони та UI-елементи. Вони не замінюють дизайнера чи аналітика, але радикально прискорюють процес ітерацій.

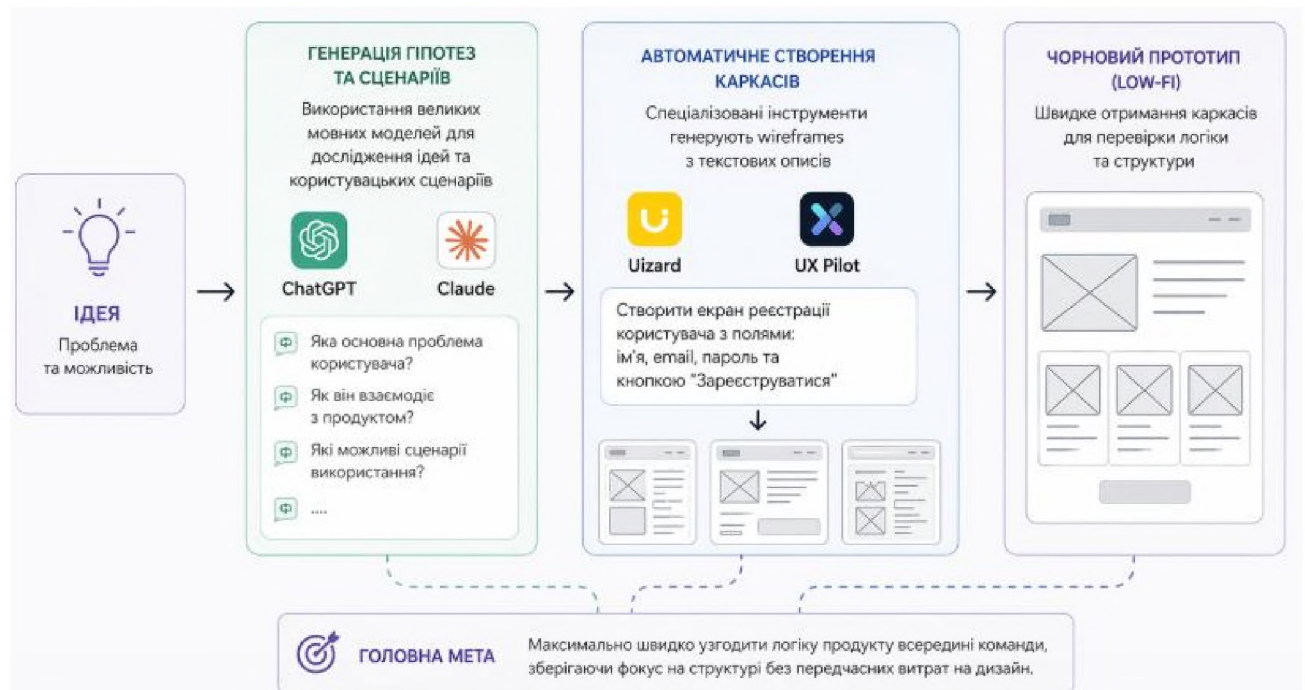


Рис.6.10. *AI*-інструменти на початкових фазах UX/UI дизайну

Рисунок 6.10 ілюструє роль штучного інтелекту на початкових фазах UX/UI дизайну, де процес переходить від ідеї до чорнового прототипу. Він демонструє використання великих мовних моделей (*ChatGPT* або *Claude*) для генерації гіпотез і сценаріїв, а також спеціалізованих інструментів (*Uizard*, *UX Pilot*) для автоматичного створення каркасів з текстових описів. Головна мета такого підходу - максимально швидко узгодити логіку продукту всередині команди, зберігаючи фокус на структурі без передчасних витрат на дизайн.

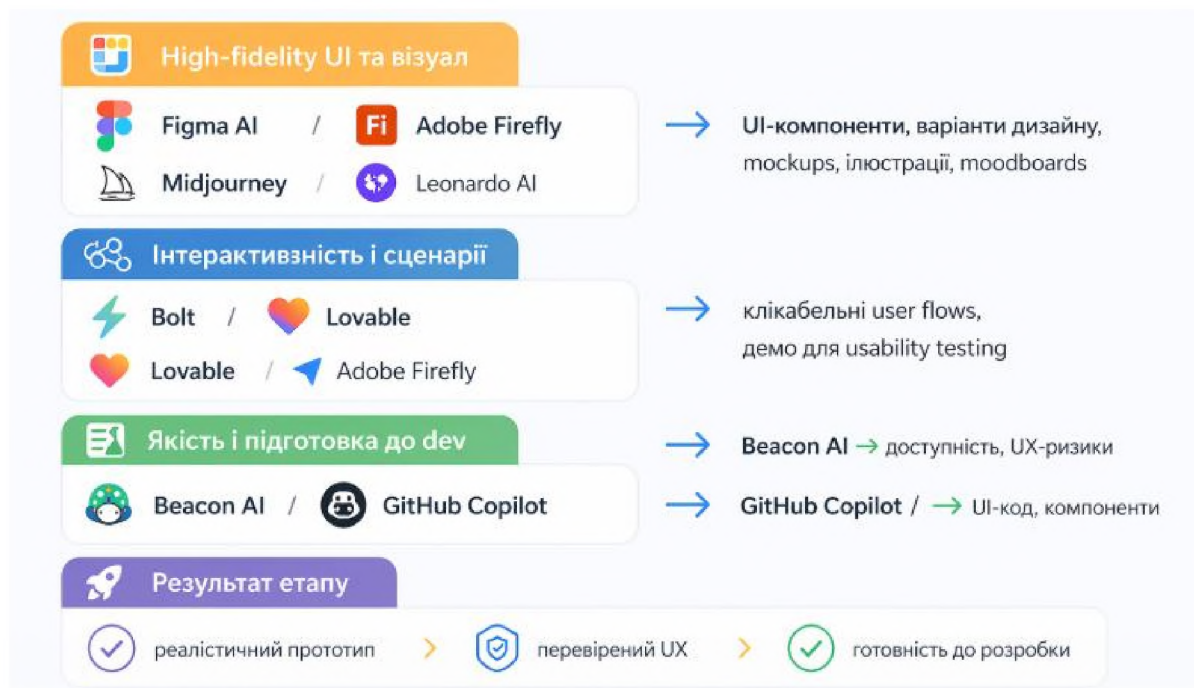


Рис.6.11. AI-інструменти для дизайну, тестування та реалізації

Рисунок 6.11 ілюструє етап переходу від концепції до реалістичного прототипу та підготовки проєкту до реалізації за допомогою спеціалізованих *AI*-інструментів. Процес охоплює створення високодеталізованого візуального дизайну (*Figma AI*, *Midjourney*), налаштування інтерактивних сценаріїв для тестування (*Bolt*, *Lovable*) та фінальну перевірку якості коду й доступності (*Beacon AI*, *GitHub Copilot*). Ключовим результатом цього етапу є перевірений *UX* та повна готовність макетів до передачі в розробку з мінімальними технічними ризиками.

Застосування *AI*-інструментів у процесі прототипування дозволяє значно підвищити ефективність проєктування інтерфейсів та скоротити час підготовки прототипів, що є важливим для сучасних процесів розроблення програмного забезпечення.

Питання для самоперевірки



1. Що таке прототип у контексті розробки продукту?
2. Яка основна мета створення прототипу?
3. Чим прототип відрізняється від фінального продукту?
4. На яких етапах життєвого циклу продукту використовують прототипи?
5. Які проблеми дозволяє виявити прототипування на ранніх етапах?
6. Чому прототипування вважається дешевшим способом перевірки ідей?
7. Яку роль прототип відіграє у прийнятті рішень?
8. Як прототип допомагає узгодити бачення між стейкхолдерами?
9. У яких випадках прототип є обов'язковим?
10. Чи може продукт існувати без прототипу? Які наслідки?
11. Що таке ризик у проєкті розробки продукту?
12. Які типи ризиків можна зменшити за допомогою прототипів?
13. Як прототипування знижує бізнес-ризик?
14. Як прототип допомагає зменшити технічні ризики?
15. Як прототипування зменшує ризики, пов'язані з користувачами?
16. Яку роль відіграє прототип у перевірці гіпотез?
17. Як прототип допомагає уникнути зайвих витрат?
18. Чому ранній фідбек користувачів знижує ризики?
19. Як прототипування впливає на ймовірність провалу продукту?
20. Які ризики не може повністю усунути прототипування?
21. Які основні типи прототипів існують?
22. Що таке *Low-fidelity* прототип?
23. У яких випадках доцільно використовувати паперовий прототип?
24. Що таке *High-fidelity* прототип?
25. Чим відрізняються горизонтальні та вертикальні прототипи?

26. Що таке еволюційний прототип?
27. Що таке одноразовий (відкидний) прототип?
28. У яких випадках обирають інтерактивний прототип?
29. Як вибір типу прототипу залежить від ризиків?
30. Чи можна поєднувати різні типи прототипів в одному проєкті?
31. Які ризики пов'язані з надмірною деталізацією прототипу?
32. Чому стейкхолдери можуть сприймати прототип як готовий продукт?
33. Які ризики виникають при неправильному виборі типу прототипу?
34. Як прототипування може затягнути строки проєкту?
35. Чи може прототип ввести команду в оману?
36. Які комунікаційні ризики пов'язані з прототипуванням?
37. Як уникнути хибних очікувань користувачів?
38. Які ризики виникають при відсутності чіткої мети прототипу?
39. Як мінімізувати ризики, пов'язані з прототипуванням?
40. Коли прототипування стає зайвим?
41. Які інструменти використовують для створення *Low-fidelity* прототипів?
42. Які інструменти використовують для створення *High-fidelity* прототипів?
43. Які цифрові інструменти підходять для інтерактивних прототипів?
44. За якими критеріями обирають інструмент прототипування?
45. Як інструменти прототипування впливають на швидкість розробки?
46. Чим відрізняються інструменти для дизайнерів і для аналітиків?
47. Чи завжди потрібні спеціалізовані інструменти для прототипування?
48. Які ризики пов'язані з використанням складних інструментів?
49. Як інструменти прототипування допомагають у тестуванні ідей?
50. Як обрати інструмент прототипування залежно від типу ризику?

Тема 7

ТРАСОВАНІСТЬ ТА ВАЛІДАЦІЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

7.1. Встановлення зв'язків між вимогами

У сучасній розробці програмного забезпечення вимоги перестали бути просто текстовими описами очікуваної поведінки системи. Вони утворюють складну, багатовимірну структуру взаємозалежностей, де кожна зміна в одному елементі може мати каскадний ефект для десятків інших артефактів: бізнес-цілей, архітектурних рішень, сценаріїв користувача, тестів, нефункціональних обмежень та навіть контрактних зобов'язань.

На перших етапах розвитку програмної інженерії вимоги здебільшого фіксувалися у вигляді монолітного документа, де аналіз обмежувався перевіркою повноти та формальної узгодженості. Проте зі зростанням складності систем, появою гнучких методологій, *DevOps*-практик і безперервної доставки програмного продукту стало очевидно, що ключовою проблемою є не просто формулювання вимог, а встановлення, підтримка та аналіз зв'язків між ними.

Встановлення зв'язків і формалізація їх контролю між вимогами до програмного забезпечення може принести такі вигоди:

- ✓ впевненість розробників програмного забезпечення в досягненні поставлених цілей, що приводить до чіткого розуміння того, як саме досягають зазначених цілей;
- ✓ можливість оцінити вплив змін до компонент програмного забезпечення на додаткову тривалість та вартість процесу розроблення програмного забезпечення, а також на покращення його функціональних можливостей;

- ✓ можливість оцінити внесок підрядників, тобто, оцінити обсяг та вартість тієї частини роботи, яку виконують за проектом інші організації;
- ✓ можливість контролювати процес реалізації програмного проекту і оцінювати обсяг виконаної роботи;
- ✓ можливість зіставляти витрати і можливу вигоду від них (наприклад, визначати економічну доцільність чи ефективність реалізації деяких компонент програмного забезпечення).

Вимоги до програмного забезпечення є відображенням очікувань зацікавлених сторін, але водночас вони є інженерними артефактами, що повинні бути точними, перевірними та взаємно узгодженими.

Кожна вимога існує не ізольовано, а в контексті інших вимог, обмежень і рішень. Нагадаємо, функціональні вимоги описують, що саме має робити система, але їх неможливо коректно інтерпретувати без нефункціональних вимог, які визначають як це має відбуватися - з якою продуктивністю, надійністю, безпекою чи масштабованістю. Бізнес-вимоги, у свою чергу, задають сенс існування системи, тоді як користувацькі вимоги формують міст між бізнесом і реалізацією.



Таким чином, вимоги формують багаторівневу структуру, де зв'язки мають як вертикальний характер (від бізнес-цілей до коду), так і горизонтальний (між вимогами одного рівня).

Зв'язок між вимогами - це формалізоване або неформалізоване відношення, яке відображає залежність, вплив або походження однієї вимоги від іншої. У практиці аналізу вимог ці зв'язки часто називають *зв'язками простежуваності (Traceability Links)*.

Найважливіше усвідомити, що зв'язки не є абстрактною теорією, вони виконують цілком практичну функцію - дозволяють відстежити життєвий шлях вимоги від початкової ідеї до реалізації, тестування та експлуатації.

Вертикальні та
горизонтальні
зв'язки

Вертикальні зв'язки відображають логіку декомпозиції, а саме: бізнес-вимога породжує одну або кілька користувацьких вимог, які, у свою чергу, деталізуються у функціональних вимогах, що реалізуються у компонентах системи та перевіряються тестами.

Горизонтальні зв'язки виникають між вимогами одного рівня (наприклад, дві функціональні вимоги можуть бути логічно пов'язані, оскільки реалізують різні аспекти одного бізнес-процесу або нефункціональна вимога до безпеки може накладати обмеження на цілу групу функціональних вимог).

Зв'язки між вимогами до програмного забезпечення, зазвичай, мають тип «багато-до-багатьох» (наприклад, одну вимогу нижнього рівня можна пов'язати з декількома вимогами вищого рівня та навпаки). У найпростішому випадку зв'язки можуть встановлюватись між різними вимогами до програмного забезпечення, але одного рівня.

Методи аналізу
зв'язків

Для підтримки процесу розроблення вимог до програмного забезпечення застосовують різні методи

аналізу зв'язків між вимогами, які допомагають забезпечити їхню узгодженість, повноту та контроль змін. До основних належать аналіз впливу, що дозволяє оцінити наслідки внесення змін до окремої вимоги для інших елементів системи; аналіз наслідків, спрямований на виявлення можливих ризиків і результатів реалізації або невиконання певної вимоги; а також аналіз покриття, який дає змогу перевірити, наскільки всі бізнес-цілі, користувацькі потреби та функціональні сценарії відображені у системних вимогах і реалізації програмного продукту.

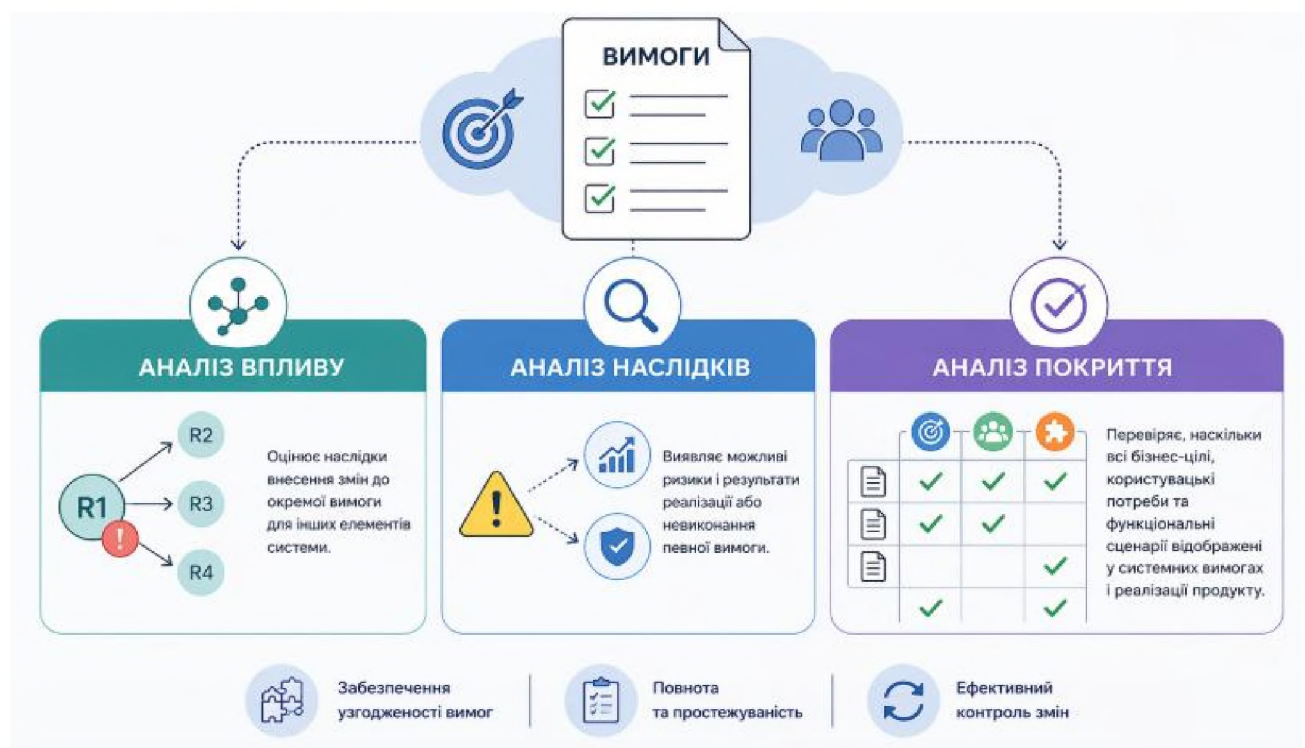


Рис.7.1. Методи аналізу зв'язків між вимогами

Аналіз впливу (Impact Analysis) дає можливість оцінити, на які інші компоненти програмної системи (компоненти нижнього рівня) вплине внесення зміни в цю конкретну компоненту (компоненту верхнього рівня). Аналіз впливу вимог відповідає на питання: «Що станеться, якщо ми це змінимо?». Ця структурована оцінка відстежує, як будь-яка запропонована або фактична

модифікація вимог вплине на все інше у роботі: обсяг, дизайн, інтерфейси, терміни, бюджети, щодо відповідності та очікування зацікавлених сторін.

Такі стандарти, як ISO/IEC/IEEE 29148, позиціонують аналіз впливу як ключовий елемент управління вимогами та контролю змін. Він базується на одному критичному інгредієнті: відстежуваності артефактів проєкту. Коли є надійна відстежуваність від початку до кінця, тобто вимоги пов'язані з дизайном, дизайн пов'язаний з кодом, код пов'язаний з тестами, тести пов'язані з ризиками та нормативними актами, можна миттєво побачити радіус впливу будь-якої зміни. У добре пов'язаній моделі вимог аналіз впливу стає керованим процесом. Аналітик може простежити ланцюжок від зміненої бізнес-вимоги до конкретних модулів коду та тестових сценаріїв. У відсутності таких зв'язків аналіз впливу перетворюється на експертне вгадування.

Аналіз наслідків працює в протилежному до аналізу зв'язків напрямку. Аналізуються компоненти нижнього рівня, наприклад, зміст вимоги, частина Специфікації вимог або результат тесту, а потім за допомогою встановлення зв'язків перевіряється його відповідність одній з компонент децю вищого рівня. Аналіз наслідків починається в точці, де виникає зміна (наприклад, нова вимога або зміна існуючої), і рухається вглиб проєкту. Він відповідає не на питання «Чому це зробили?», а на питання «Що зламається, якщо це зачепимо?». Цей рух відбувається за такою логікою:

- ✓ Точка входу: зміна у вимозі А.
- ✓ Горизонтальний вплив: які інші вимоги того ж рівня залежать від вимоги А? (наприклад, зміна системи логіну впливає на систему профілю).
- ✓ Вертикальний вплив (вниз): які елементи дизайну, модулі коду та бази даних реалізують цю вимогу?
- ✓ Кінцевий вплив: які тести, інструкції та сертифікації стануть невалідними?

Щоб зрозуміти, як працює цей «зворотний рух», аналітики розглядають три типи наслідків:

- ✓ *наслідки залежностей*: дивимось вниз по ланцюжку трасованості (якщо відбувається зміна типу даних у полі «Ціна», аналіз показує, що потрібно змінити код калькулятора, логіку податкових звітів та інтерфейс кошика);
- ✓ *наслідки виконання*: оцінка того, скільки роботи потрібно переробити (відбувається підрахунок кількості зв'язків: «Зміна однієї вимоги потягне за собою зміну 4 сторінок електронного прототипу та 12 тест-кейсів»);
- ✓ *ризикові наслідки*: аналіз виявляє критичні вузли (якщо вимога має 50 зв'язків, наслідок її зміни може бути катастрофічним для стабільності всього програмного забезпечення).

Аналіз покриття використовується для аналізу вхідних зв'язків, щоб перевірити чи всі вимоги вищого рівня пов'язані зі Специфікаціями вимог на нижчих рівнях і відповідними тестами.

Покриття тестами (*Test Coverage*) - найпоширеніший вид аналізу покриття, який відповідає на питання: «Який відсоток вимог фактично перевірили?» і проводиться з метою виявити «сліпі зони» - вимоги, для яких не написано жодного тест-кейсу (оскільки вимога може бути реалізована в коді, але через відсутність тесту вона може працювати некоректно або зламатися при першому ж оновленні). Покриття дизайном та архітектурою відбувається раніше, коли перевіряється, чи кожна функціональна вимога врахована в архітектурній схемі або в електронному прототипі з метою переконатися, що логіка продукту цілісна (якщо вимога не покрита дизайном, розробник буде змушений приймати рішення на свій розсуд, що часто веде до помилок).

Для аналізу використовується матриця трасованості (пізніше до неї повернемося), але фокус зміщується на пошук порожніх значень. Перевірка покриття проводиться в двох напрямках:

✓ вертикальне покриття: перевірка ланцюжка для кожної вимоги: *Вимога* → *Дизайн* → *Код* → *Тест* (якщо в рядку є прогалина - це ризик незавершеності);

✓ горизонтальне покриття: перевірка в межах одного рівня (наприклад, чи всі сценарії використання покривають бізнес-цілі Замовника).

Також метод аналізу покриття вимогами застосовується для вимірювання процесу виконання роботи, тобто наскільки системні інженери просунулись в процесі задоволення вимог користувачів. У процесі розроблення програмного забезпечення кожна системна вимога зв'язується з тими вимогами користувача, для задоволення яких вона й призначена. На будь-якому етапі розроблення системних вимог до програмного забезпечення частку їх готовності можна порівняти з часткою покриття користувацьких вимог системними. Це дуже зручно для управлінського персоналу, оскільки дає змогу контролювати прогрес розроблення програмного забезпечення навіть на початкових його стадіях.

7.2. Відображення елементарних зв'язків між вимогами

У процесі розроблення програмного забезпечення вимоги ніколи не існують ізольовано. Кожна вимога формується в контексті інших, впливає на них або, навпаки, зазнає їхнього впливу. Саме тому аналіз вимог не може обмежуватися простим переліком функцій чи характеристик системи. Він потребує розуміння зв'язків, які поєднують вимоги між собою в єдину логічну структуру.

Елементарні зв'язки між вимогами є базовим інструментом такого аналізу. Вони дозволяють формально зафіксувати відношення між окремими вимогами, пояснити їхню взаємозалежність і передбачити наслідки змін. Без явного відображення цих зв'язків система вимог втрачає цілісність, а управління проєктом ускладнюється через приховані залежності та конфлікти.

Під елементарними зв'язками між вимогами слід розуміти найпростіші, атомарні типи відношень, які не підлягають подальшій декомпозиції. Вони відображають фундаментальні логічні залежності, що виникають між вимогами різних рівнів - від бізнес-цілей до конкретних функціональних або нефункціональних характеристик системи. Такі зв'язки не описують поведінку системи безпосередньо, але визначають, яким чином окремі вимоги формують узгоджену модель майбутнього програмного продукту. Через них стає можливим встановлення трасованості, аналіз повноти вимог і прогнозування наслідків змін на будь-якому етапі життєвого циклу розробки програмного забезпечення.

Основні типи зв'язків

Для ефективного управління проектом недостатньо просто мати список вимог - потрібно розуміти, як вони взаємодіють між собою. Основні типи зв'язків допомагають підтримувати цілісність системи, проводити аналіз наслідків та забезпечувати повне покриття вимог.



Рис.7.2. Основні типи зв'язків між вимогами

Зв'язки ієрархії та декомпозиції (Parent-Child). Це найпоширеніший тип зв'язку, який виникає, коли високорівнева вимога розбивається на детальніші складові (наприклад, залежність «*Parent-Child*» наступна: високорівнева бізнес-вимога «Система повинна приймати оплату» породжує кілька функціональних вимог «Підтримка *Visa/Mastercard*», «Інтеграція з *PayPal*», «Генерація квитанції»). Декомпозиція допомагає відстежити, чи всі «дочірні» вимоги в сукупності повністю реалізують «батьківську» ціль.

Зв'язки залежності (Dependency) вказують на технічний або логічний взаємозв'язок між різними функціями:

- ✓ *Requires* (потребує) означає, що вимога А не може бути реалізована або не має сенсу без вимоги Б (наприклад, функція «Оформлення замовлення» потребує наявності функції «Кошик»).

- ✓ *Precedes* (передуює) визначає послідовність впровадження (вимога А повинна бути реалізована раніше за вимогу Б через архітектурні чи логічні обмеження).

- ✓ *Conflicts* (конфліктує) вказує на те, що дві вимоги не можуть бути реалізовані одночасно або суперечать одна одній (наприклад, вимога до надвисокої безпеки може конфліктувати з вимогою до миттєвого входу без пароля).

Зв'язки трасованості (Traceability) з'єднують вимоги з іншими артефактами проєкту протягом усього життєвого циклу:

- ✓ *Satisfies* задовольняє зв'язок між проєктним рішенням (дизайном) або кодом та вимогою, підтверджує, що конкретний елемент архітектури реалізує конкретну вимогу.

- ✓ *Verifies* верифікує зв'язок між вимогою та тест-кейсом (основа для аналізу покриття вимог).

✓ *Refines* уточнює, що одна вимога є більш детальною версією іншої, не обов'язково будучи її «дитиною», а просто додаючи контекст (наприклад, вимога до продуктивності уточнює функціональну вимогу).

Зв'язки впливу (Impact) використовуються для оцінки того, як зміна в одному модулі може зачепити інші частини системи:

✓ *Impacted by* (зазнає впливу) вказує на те, що зміна вимоги А призведе до необхідності перегляду вимоги Б.

✓ *Relates to* (пов'язаний з) вказує на те, що вимоги мають спільний контекст, але не мають прямої жорсткої залежності.

Розуміння типів зв'язків дозволяє автоматизувати аналіз наслідків. Якщо відомо, що зв'язок між вимогами має тип «*Requires*», тоді стає зрозуміло, що видалення однієї функції автоматично робить іншу непридатною. У випадку зв'язку «*Refine*» наслідки будуть менш критичними та стосуватимуться лише документації чи уточнення деталей.

Для аналізу зв'язків використовуються різні моделі: матриці простежуваності, графи залежностей, контекстні діаграми. Вибір моделі залежить від цілей аналізу та складності системи. Хоча в *UML* діаграми вимог нема, для формулювання вимог там інколи пристосовують діаграму прецедентів. Графові моделі особливо ефективні для виявлення непрямих залежностей і циклів.

У мові моделювання систем *SysML* (*Systems Modeling Language*) зв'язки між вимогами є строго типізованими. Це дозволяє створювати цифрові моделі, де будь-яка зміна автоматично підсвічує наслідки для всієї системи. Модель вимог на рисунку 7.3 є прикладом діаграми вимог у стилі *SysML*. Модель демонструє не просто перелік вимог, а структуровану мережу зв'язків між ними, що дозволяє простежити логіку походження, деталізації та обмежень. У діаграмі використано

кілька типів зв'язків: *satisfy* (відповідність зовнішнім стандартам або нормативам), *refine* (уточнення та деталізація вимог) та ієрархічні зв'язки між загальними та конкретними вимогами.

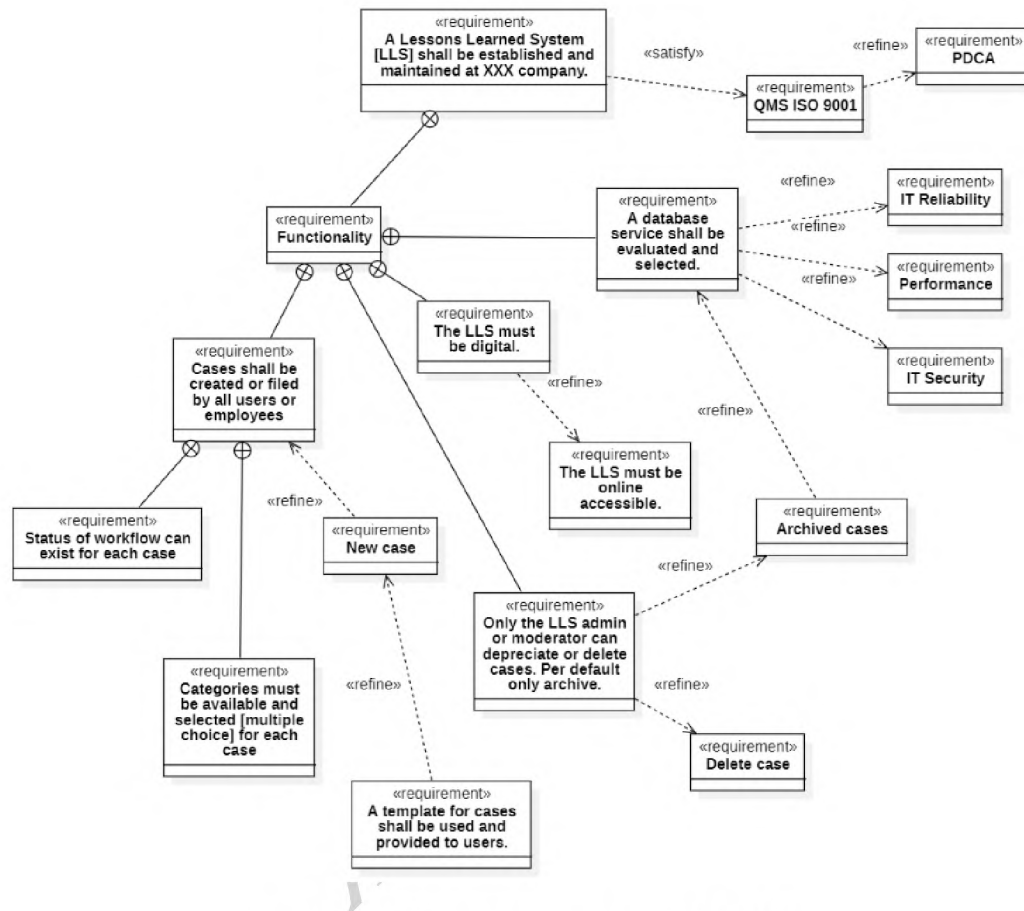


Рис.7.3. Модель вимог у стилі SysML

Використання *SysML* для представлення зв'язків вимог дозволяє перетворити вимоги з простого тексту на інтерактивну мережу, де зміна однієї вимоги через зв'язки миттєво покаже, які блоки архітектури треба перепроєктувати і які тести провести заново.

7.3. Матриця покриття та матриця відстеження вимог

У процесі розроблення програмного забезпечення вимоги постійно еволюціонують: змінюються бізнес-цілі, уточнюються очікування користувачів,

з'являються технічні обмеження. У такому середовищі ключовою проблемою стає не формулювання вимог як таких, а контроль того, чи всі вони реалізовані, перевірені та не загублені на жодному етапі життєвого циклу програмного забезпечення

Матриця відстеження вимог

Матриця відстеження вимог (Requirements Traceability Matrix, RTM) - це не просто таблиця або допоміжний артефакт проєктної документації. У процесі розроблення програмного забезпечення вона виконує роль логічного «каркаса», який з'єднує всі етапи життєвого циклу системи: від бізнес-очікувань Замовника до конкретних реалізацій у коді та результатів тестування. Її основне призначення полягає у забезпеченні прозорості та керованості вимог, а також у можливості обґрунтовано відповідати на запитання: чому ця функція існує, яку вимогу вона реалізує і як перевірено її коректність.

У практиці розроблення складних інформаційних систем вимоги рідко залишаються незмінними (еволюціонують, уточнюються, іноді суперечать одна одній або втрачають актуальність) і ось тут матриця відстеження стає інструментом контролю змін, оскільки дозволяє побачити, які елементи системи залежать від конкретної вимоги і які наслідки матиме її модифікація або вилучення.

Ключовою ідеєю матриці відстеження є встановлення зв'язків між вимогами та артефактами проєкту. Ці зв'язки не є статичними, вони утворюють динамічну модель відповідностей. У класичному розумінні виділяють пряме та зворотне відстеження, але на практиці вони існують одночасно і доповнюють одне одного.

Пряме відстеження дозволяє простежити шлях від вимоги до її реалізації. Бізнес-вимога трансформується у користувацьку або системну, далі у проєктне рішення, програмний модуль і, зрештою, у тестовий сценарій і якщо на будь-

якому з цих рівнів ланцюг обривається, це сигналізує про ризик: або вимога не реалізована повністю, або її реалізація не перевірена.

Зворотне відстеження працює у протилежному напрямку. Воно відповідає на запитання, чи кожен елемент реалізації має під собою обґрунтовану вимогу. Це особливо важливо для запобігання так званому «золотому напильнику», коли в систему додаються функції, що не мають цінності для Замовника і ускладнюють підтримку продукту.

З формального погляду матриця відстеження зазвичай має вигляд таблиці, у якій рядки представляють вимоги, а стовпці - пов'язані з ними артефакти: проектні рішення, компоненти коду, модулі, тест-кейси або сценарії приймального тестування. Однак її справжня цінність полягає не у формі, а у змісті зв'язків, які вона фіксує.

Кожен запис у матриці є своєрідним твердженням того, що цей елемент реалізує або перевіряє конкретну вимогу. Сукупність таких тверджень створює модель відповідності між очікуваннями та результатами. У складних проєктах матриця може набувати багатовимірного характеру.

Project Name: TestKarts.com Project Manager: Manager TestKarts.com Project Description: Testing and Development project for the fresher and experience learner.					Requirements Traceability Matrix									
Requirements					Testing									
Req. ID	Requirements Description	Requirements Source	Requirement Type	WBS Deliverables	Test Case ID	Test Description	TEST	UAT	QA	PROD	PRE PROD	NON PROD	Defective	Defe
REQ001	User registration functionality	Business Analyst	Functional	WBS-001	TC001	Verify user can successfully register an account	Pass	Pass	Pass	Pass	Fail		No	
REQ001	User registration functionality	Business Analyst	Functional	WBS-001	TC002	Verify error message is displayed for invalid inputs	N/A	Fail	Pass	Fail	N/A		Yes	DEFOO
REQ001	User registration functionality	Business Analyst	Functional	WBS-001	TC003	Verify user receives a confirmation email	Pass	N/A	Fail	N/A	Pass		Yes	DEFOO
REQ002	Login functionality	Product Owner	Functional	WBS-002	TC004	Verify user can log in with valid credentials	Fail	Pass	N/A	N/A	N/A		NO	
REQ002	Login functionality	Product Owner	Functional	WBS-002	TC005	Verify error message is displayed for incorrect login	Fail	Fail	N/A	Pass	Fail		Yes	DEFOO
REQ002	Login functionality	Product Owner	Functional	WBS-002	TC006	Verify "Forgot Password" link redirects correctly	Fail	Fail	N/A	Pass	Pass		No	
REQ003	Create new post	User Stories	Functional	WBS-003	TC007	Verify user can create a new post				N/A	N/A		No	
REQ003	Create new post	User Stories	Functional	WBS-003	TC008	Verify error message is displayed for empty content	Pass	Pass	Pass	Fail	Fail		Yes	DEFOO

Рис.7.4. Зразок матриці відстеження вимог

Матриця відіграє важливу роль у комунікації між стейкхолдерами. Для аналітиків ж вона є засобом контролю повноти вимог, для розробників - орієнтиром у реалізації, для тестувальників - основою побудови тестового покриття, а для менеджерів - джерелом об'єктивної інформації про стан проєкту.

Матриця покриття вимог

Матриця покриття вимог (Requirements Test Coverage Matrix) - це структурований інструмент аналітичного контролю, який дозволяє встановити явний і перевірюваний зв'язок між вимогами та артефактами розроблення: дизайном, кодом, тестами, релізами або бізнес-цілями. Вона виконує роль «карти відповідності», що демонструє, які вимоги чим реалізовані і чим перевірені. В основі матриці покриття лежить проста, але принципово важлива ідея: жодна вимога не повинна існувати ізольовано. Кожна вимога має походження (бізнес або користувач), реалізацію (функціональність, модуль, компонент) і підтвердження коректності (тест, сценарій перевірки, критерій приймання).

Матриця формує двовимірний простір, де по одній осі розміщуються вимоги, а по іншій - об'єкти, з якими ці вимоги мають бути пов'язані. У перетині фіксується факт зв'язку: наприклад, що конкретний тест перевіряє конкретну вимогу, або що певний модуль реалізує визначену бізнес-потребу.

Requirement ID	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5
REQ-001	✓		✓		
REQ-002		✓		✓	
REQ-003	✓		✓		✓
REQ-004		✓		✓	
REQ-005			✓		✓
REQ-006	✓		✓		
REQ-007		✓			✓

Таким чином, матриця перетворює абстрактний перелік вимог на керовану систему, яку можна аналізувати, перевіряти та масштабувати.

З точки зору якості програмного забезпечення матриця покриття виконує функцію «детектора прогалин». Якщо вимога не має пов'язаного тесту, вона формально не перевірена. Якщо тест не пов'язаний із жодною вимогою, виникає підозра надлишкової або зайвої функціональності. Якщо функціональний модуль

не має посилання на вимогу, це сигнал про неконтрольоване розширення обсягу системи.

Завдяки матриці покриття команда може аргументовано відповісти на запитання чи реалізовано все, що було погоджено, чи перевірено все, що реалізовано і чи не робиться щось зайве?

Однією з найсильніших сторін матриці покриття є підтримка аналізу впливу змін. Коли змінюється або видаляється вимога, матриця одразу показує, які компоненти системи та які тести потрапляють у зону ризику. Це дає змогу приймати рішення не інтуїтивно, а на основі структурованих даних. Без матриці зміни часто призводять до каскадних помилок (щось виправили в коді, але забули оновити тест; змінили бізнес-логіку, але не скоригували документацію). Матриця покриття мінімізує такі ситуації, виступаючи механізмом «раннього попередження».

Незважаючи на назву, матриця покриття не завжди існує у вигляді класичної таблиці. На початкових етапах або в невеликих проєктах вона може бути реалізована у звичайній електронній таблиці. У складних системах матриця інтегрується в спеціалізовані інструменти управління вимогами, де зв'язки підтримуються автоматично.

Таблиця 7.1 - Різниця між матрицею покриття та матрицею відстеження

<i>Критерій</i>	<i>Матриця покриття</i>	<i>Матриця відстеження</i>
Основна мета	Перевірка повноти реалізації і тестування вимог	Простеження життєвого циклу вимог
Основний акцент	Покриття тестами і функціональністю	Зв'язок між артефактами проєкту
Використання	Тестування ПЗ	Управління вимогами
Рівень деталізації	Зазвичай менший	Більш повний

7.4. Валідація вимог до програмного забезпечення

У межах визначення процесу аналізу вимог до програмного забезпечення валідацію доцільно розглядати не як формальну перевірку тексту Специфікації вимог, а як процес підтвердження цінності та доцільності вимог для бізнесу і стейкхолдерів. Такий підхід узгоджується з положеннями *A Guide to the Business Analysis Body of Knowledge (BABOK)*, який є одним із ключових міжнародних стандартів у сфері бізнес-аналізу і описаний в межах області знань аналізу вимог та визначення проєктування (*Requirements Analysis and Design Definition*) через задачу перевірки вимог (*Validate Requirements*) на відповідність бізнес-потребі, створення очікуваної цінності, узгоджуваності з цілями організації та були готовими до реалізації. Методично це означає, що валідація не зводиться лише до логічної перевірки формулювань (це більше верифікація), а передбачає оцінювання доцільності та значущості кожної вимоги. *BABOK* чітко розрізняє:

- ✓ *Verify Requirements* - перевірка якості формулювання (повнота, однозначність, узгодженість, тестованість);
- ✓ *Validate Requirements* - підтвердження, що вимоги дійсно ведуть до досягнення бізнес-цілей.

Таке розмежування допомагає усвідомити стратегічний характер валідації та визначити необхідні інструменти валідації.

Інструменти валідації вимог до програмного забезпечення - це систематизований комплекс методів, процедур, моделей і технічних засобів, спрямованих на підтвердження правильності, повноти, несуперечливості, здійсненності та перевірюваності сформульованих вимог з метою забезпечення створення програмного продукту, що відповідає очікуванням стейкхолдерів і встановленим стандартам (див. Табл 7.2).

Таблиця 7.2. Інструменти валідації вимог

Інструменти валідації функціональних вимог	Інструменти валідації нефункціональних вимог
Перегляд вимог (<i>Review of Requirements</i>) Інспекція Специфікації (<i>SRS inspection</i>) Прототипування (<i>Low-fidelity, High-fidelity</i>) <i>Use Case validation, User Story validation</i> Перевірка критеріїв прийнятності (<i>Acceptance Criteria check</i>) <i>Traceability Matrix</i> Моделювання UML (<i>Use Case, Sequence, Activity</i>) та BPMN-моделі Сесії з розгляду питань зацікавленими сторонами (<i>Stakeholder review sessions</i>)	Тестування продуктивності (Тестування продуктивності) Тестування навантаження / стресу (<i>Load / Stress testing</i>) Тестування безпеки (<i>Security testing</i>) Тестування зручності використання (<i>Usability testing</i>) Тестування доступності (<i>Accessibility testing</i>) Тестування надійності (<i>Reliability testing</i>) Тестування масштабованості (<i>Scalability testing</i>)

Особливо ефективною є імітація колективної сесії валідації, де частина учасників виступає в ролі Замовника, а частина - бізнес-аналітиків. Це дозволяє продемонструвати, що валідація - це процес узгодження цінностей, а не лише перевірка тексту.

Воркшоп із вимог

Поняття *воркшоп із вимог (Requirements Workshop)* належить насамперед до теми елісітації (виявлення) вимог, тобто до процесу отримання вимог від зацікавлених сторін, оскільки це фасилітована групова сесія, під час якої аналітик працює зі стейкхолдерами для уточнення бізнес-цілей, формування функціональних і нефункціональних вимог, виявлення конфліктів з метою досягнення спільного розуміння.

Хоча воркшоп формально належить до елісітації, його часто використовують під час аналізу і узгодження вимог, оскільки під час воркшопу не лише збирають інформацію, а й структурують вимоги, пріоритизують їх, усувають суперечності та формують початкову модель системи. Тобто воркшоп - це водночас інструмент колективного аналізу і якщо він проводиться після підготовки чернетки вимог (наприклад, для *review SRS* або *Backlog*), тоді він

відноситься до процесу валідації та перевірки вимог і у такому випадку його мета - перевірити повноту, логічну узгодженість та відповідність бізнес-цілям.

Зображення на рисунку 7.5 ілюструє *Requirements Workshop* як інструмент комплексної валідації вимог через шість ключових питань: *What, Why, Who, How, Where, When*.

Якщо розглядати його саме з позиції валідації вимог, то кожен елемент виступає окремим виміром перевірки їхньої коректності та повноти.



Рис. 7.5. Модель 6W-валідації вимог у форматі *Requirements Workshop*

WHAT (Що саме має бути реалізовано?): зміст вимог. З точки зору валідації це означає уточнення формулювань, усунення двозначностей, перевірку повноти функціональних та нефункціональних вимог та наявність критеріїв приймання. Тут команда відповідає на питання: Чи правильно описали систему? Якщо вимогу неможливо однозначно інтерпретувати або протестувати - вона не пройшла валідацію.

WHY (Навіщо ця вимога?): зв'язок із бізнес-цілями. У валідації це перевірка релевантності чи підтримує вимога стратегічну мету, чи створює вона

цінність і чи не є вона зайвою або надмірною. Це захист від так званих «*gold plating*» - функцій, які не мають обґрунтованої потреби.

WHO (Хто зацікавлений?): стейкхолдери. У процесі валідації важливо перевірити чи враховані всі групи користувачів, чи немає конфлікту інтересів, чи узгоджені очікування між сторонами. Якщо вимога задовольняє лише одну сторону і суперечить іншій - вона потребує перегляду.

HOW (Як це буде реалізовано або використано?): механізм функціонування. З позиції валідації це означає перевірку реалізованості, технічну здійсненність, відповідність архітектурним обмеженням, наявність альтернативних сценаріїв (у т.ч. виняткових). Це вже межа між валідацією і техніко-економічною експертизою вимог.

WHERE (У якому середовищі працює система?): контекст використання. Валідація тут передбачає перевірку відповідності вимог операційному середовищу, врахування нормативних чи галузевих обмежень, сумісність із зовнішніми системами. Неврахований контекст часто стає причиною пізніх змін.

WHEN (Коли та за яких умов?): часовий вимір вимог. З точки зору валідації це перевірка пріоритетів, фаз реалізації, часових обмежень та залежностей між вимогами. Також сюди належать вимоги до продуктивності.

Зображення на рисунку можна інтерпретувати як модель *6W*-валідації вимог, яка демонструє, що валідація вимог - це багатовимірний процес, який виходить за межі простої перевірки формулювань, а сам воркшоп слугує майданчиком, де всі ці аспекти аналізуються одночасно через структуровані запитання, а кожне питання є фільтром якості, що дозволяє забезпечити повноту, досягти узгодженості, підтвердити бізнес-доцільність та мінімізувати ризики реалізації.

Питання для самоперевірки



1. Що означає встановлення зв'язків між вимогами в контексті життєвого циклу програмного забезпечення?
2. Які типи зв'язків можуть існувати між вимогами різних рівнів?
3. Чим відрізняється ієрархічний зв'язок від залежності?
4. Як виявити приховані залежності між вимогами?
5. Які ризики виникають при відсутності явних зв'язків між вимогами?
6. Як встановлення зв'язків впливає на аналіз впливу змін?
7. Чи можуть нефункціональні вимоги бути пов'язані з кількома функціональними?
8. Як визначити конфлікт між вимогами?
9. Які методи використовуються для моделювання зв'язків (діаграми, графи, таблиці)?
10. Як підтримувати актуальність зв'язків у процесі змін?
11. Чи можлива автоматизація встановлення зв'язків?
12. Як оцінити повноту встановлених зв'язків?
13. Що таке елементарний зв'язок між вимогами?
14. Які базові типи елементарних зв'язків існують?
15. Як документувати елементарні зв'язки у специфікації вимог?
16. Які позначення використовуються для фіксації залежностей?
17. Як виявити дублювання вимог через аналіз зв'язків?
18. Чи можуть існувати циклічні залежності між вимогами?
19. Як обробляти конфліктні зв'язки?
20. Яку роль відіграють унікальні ідентифікатори вимог?
21. Як відображати зв'язки у табличній формі?
22. Як використати матрицю залежностей для аналізу складності?

23. Які помилки найчастіше виникають при встановленні зв'язків?
24. Як забезпечити узгодженість між зв'язками та ієрархією вимог?
25. Як перевірити коректність встановлених елементарних зв'язків?
26. Що таке матриця трасованості вимог?
27. У чому різниця між матрицею покриття та матрицею відстеження?
28. Які артефакти можуть бути пов'язані через *RTM*?
29. Як будується двостороння трасованість?
30. Як виявити «осиротілі» вимоги за допомогою матриці?
31. Як матриця покриття допомагає контролювати повноту тестування?
32. Які показники можна розрахувати на основі *RTM*?
33. Як інтегрувати матрицю трасованості в *Agile*-проекті?
34. Які інструменти підтримують автоматичну трасованість?
35. Як оновлювати матрицю при зміні вимог?
36. Які ризики виникають при відсутності *RTM*?
37. Як матриця допомагає в аудиті або сертифікації?
38. Чи потрібно відображати нефункціональні вимоги в *RTM*?
39. Чим відрізняється валідація від верифікації вимог?
40. Які критерії якості перевіряються під час валідації?
41. Які методи валідації є найбільш ефективними?
42. Як перевірити тестованість вимоги?
43. Як виявити неоднозначність формулювань?
44. Яку роль відіграють стейкхолдери у валідації?
45. Як документуються результати валідації?
46. Яка мета проведення *Requirements Workshop*?
47. Які етапи підготовки до воркшопу?
48. Які техніки фасилітації застосовуються під час сесії?
49. Як узгодити конфліктні вимоги під час воркшопу?
50. Як результати воркшопу інтегруються у процес трасованості та валідації?

Тема 8

ПОГОДЖЕННЯ ТА УПРАВЛІННЯ ВИМОГАМИ

8.1. Погодження вимог до програмного забезпечення

У системній та програмній інженерії вимога розглядається як формалізований опис потреби або обмеження, що має бути реалізовано системою. Вимога відображає очікування стейкхолдерів, бізнес-цілі організації та умови функціонування майбутнього продукту.

З позиції сучасної інженерії вимог, зокрема підходів, описаних у *BABOK Guide*, вимоги поділяються на бізнес-вимоги, вимоги стейкхолдерів, рішення (*Solution Requirements*) та перехідні вимоги. Така структура дозволяє простежити логічну еволюцію від стратегічного бачення до конкретної функціональності.

Вимога має низку характеристик: повнота, несуперечливість, однозначність, перевірюваність, досяжність, трасованість. Порушення хоча б однієї з них створює потенційну зону ризику.

Більшості розробників знайоме відчуття розчарування, яке виникає, якщо їх просять реалізувати надто неясні та неповні вимоги, коли вони не можуть отримати необхідну інформацію. Тоді вони вимушені орієнтуватись на власні здогадки, котрі не завжди вірні, що в кінцевому випадку приводить до затрати великих зусиль, щоб виправити помилки у вимогах, робота над якими вже завершена. Дослідження показали, що в 100 разів дорожче виправляти помилки у вимогах, якщо на них вказує клієнт, ніж в процесі розробки цих вимог. Інші дослідження свідчать, що на виправлення помилки, виявленої на стадії роботи над вимогами, витрачається в середньому 30 хвилин, тоді як на виправлення помилки, виявленої під час тестування системи, необхідно від 5 до 17 годин. Зрозуміло, що будь-які зусилля, затрачені на виявлення помилок в специфікації до вимог, зекономлять реальні гроші та час.

Якщо планувати тестування та розробку варіантів тестування на ранній стадії проекту, тоді можна виявити більшість помилок одразу після їх появи, при цьому будуть попереджені додаткові збитки та знизяться затрати на тестування та обслуговування. На рисунку 8.1 зображена V-подібна модель розробки програмного забезпечення, коли дії, пов'язані із тестуванням і розробкою проекту виконуються паралельно. Ця модель показує, що приймальне дослідження базується на вимогах користувачів, тестування системи - на функціональних вимогах, а тестування цілісності – на архітектурі системи.



Рис.8.1. V-подібна модель розробки програмного забезпечення із плануванням тестування на ранній стадії проєкту

Погодження вимог являється четвертим етапом розробки вимог, поряд із збором інформації, аналізом та створенням Специфікації вимог до програмного забезпечення. Погодження вимог дозволяє впевнитись в тому, що:

- ✓ в Специфікації вимог до програмного забезпечення відповідним чином описані передбачувані можливості та характеристики системи, котрі задовольняють потреби різних зацікавлених в проєкті осіб;

- ✓ вимоги до програмного забезпечення точно відображають системні вимоги, бізнес-правила, вимоги користувачів;
- ✓ вимоги повні та високоякісні;
- ✓ всі вимоги погоджені одна з одною;
- ✓ вимоги забезпечують якісну основу для дизайну та збірки програмного забезпечення.

Перевірка підтверджує, що в документі представлені бажані характеристики вимог (повні, коректні, здійсненні, необхідні, пріоритетні, ясні та ті, що піддаються перевірці) і відмінно описані специфікації до вимог. Звісно, що погодити можна тільки задокументовані вимоги.



Погодження – це не лише окремих етап процесу, що виконується після збору та документування вимог. Деякі перевірочні дії, наприклад перегляд все більш збільшуваної Специфікації вимог, виконуються після кожної процедури збору інформації, її аналізу та документування. Інші дії, такі, як офіційна перевірка Специфікації вимог, забезпечують досягнення того рівня якості, котрий передувє фінальній версії Специфікації вимог.

Учасники проєкту іноді з небажанням витрачають час на перевірку та тестування Специфікації вимог до програмного забезпечення. Інтуїтивно здається, що якщо виділити час на покращення якості вимог, дата випуску продукту затримається на такий же строк. В дійсності ж ці зусилля можуть скоротити графік постачання, за рахунок зменшення потрібних виправлень та прискорення інтеграції та тестування системи. *Чим кращі вимоги, тим вища*

якість продукту, і тим більше задоволений клієнт, що в свою чергу знизить витрати на обслуговування, покращення та клієнтську підтримку продукту.

Погодження вимог - це процес формального й неформального досягнення згоди щодо змісту, пріоритетів та критеріїв прийняття вимог. Його можна розглядати у трьох вимірах:

- ✓ когнітивному – як формування спільного розуміння;
- ✓ організаційному – як узгодження ролей та відповідальностей;
- ✓ процедурному – як затвердження документації.

У межах традиційних моделей погодження завершується підписанням *SRS*. У гнучких підходах погодження інтегроване в ітераційний цикл.

Будь-яке погодження передбачає взаємодію із зацікавленими сторонами. Стейкхолдери можуть мати різні рівні впливу, інтересів і компетенцій, тому часто у великих організаціях часто виникають приховані конфлікти: стратегічні цілі керівництва можуть суперечити оперативним потребам користувачів. Непогоджені очікування створюють латентні ризики, які проявляються на етапі тестування або впровадження. Аналітик у цьому процесі виконує функцію медіатора; його завдання - перетворити абстрактні очікування на формалізовані, перевірювані та досяжні вимоги.

Досягнення консенсусу між зацікавленими сторонами - це складний комунікаційний процес. На практиці для цього застосовуються фасилітаційні сесії, воркшопи, прототипування, моделювання сценаріїв. Особливе значення має візуалізація, оскільки графічні моделі (*UML*-діаграми, *BPMN*, *User Journey Maps*) знижують ризик різночитань. Погодження без візуалізації часто базується на припущеннях, що формує так звану ілюзію спільного розуміння.

8.2. Перегляд вимог до програмного забезпечення

Дослідження продукту програмного забезпечення на предмет виявлення проблем будь-яким іншим лицем, крім його автора, називається *експертною оцінкою (Peer Review)*. Перегляд задокументованих вимог - це ефективний прийом виявлення неясних вимог або тих, які не піддаються перевірці, котрі були визначені недостатньо ясно для розробки, а також інші проблеми.

Різноманітні типи експертних оцінок називаються по-різному. Неофіційні перегляди корисні при знайомстві людей з продуктом та зборі відгуків про нього (формування зворотного зв'язку). Однак вони не є систематичними, повними та погодженими. Є декілька видів неофіційних переглядів вимог:

- ✓ *Перевірка «за столом»* - коли автор просить свого одного колегу дослідити свій продукт.
- ✓ *Колективна перевірка* – коли автор запрошує декілька колег для паралельної перевірки продукту.
- ✓ *Критичний аналіз* – коли автор описує створений продукт та просить його прокоментувати.

На відміну від природних неформальних переглядів, офіційна перевірка являє собою суворо регламентований процес. Після його завершення формується звіт, в котрому вказується матеріал, рецензенти та думка команди рецензентів про придатність продукту. Головний результат – це сукупність всіх знайдених дефектів та піднятих запитань. Найбільш добре себе зарекомендувала форма офіційної оцінки під назвою *експертиза*.

Експертиза

Експертиза була розроблена Майклом Феганом з *IBM* в середині 70-х рр., а інші доповнили та модифікували його



методи. Експертиза була признана кращим прийомом в області розробки програмного забезпечення. Цей метод придатний для перевірки будь-якого продукту, в тому числі документації вимог та дизайну, вихідного коду, документації тестування та планів проекту.

Експертиза – це чітко визначений багатоетапний процес, в якому приймає участь невелика команда кваліфікованих спеціалістів, котрі ретельно перевіряють продукт на наявність дефектів та вивчають можливості покращення. Експертиза забезпечує рівень якості продукту, що передують завершальному; вона є потужним прийомом підвищення якості продукту.



Учасники експертизи

Учасники експертизи повинні презентувати чотири точки зору на продукт.

1. *Автор продукту і, можливо, колеги автора.* Це може бути аналітик, який склав документацію вимог.

2. *Автор будь-якого попереднього продукту чи Специфікації для елемента, котрий варто перевіряти.* В цій ролі може виступити системний інженер чи архітектор, котрий здатний підтвердити, що в Специфікації вимог до програмного забезпечення система описана досить детально. При відсутності Специфікації більш високого рівня в експертизі повинні прийняти участь представники клієнта для того, щоб підтвердити, що вимоги в Специфікації вимог описані правильно та повно.

3. *Люди, які будуть виконувати роботу, основу на перевірочному елементі.* Для експертизи Специфікації вимог до програмного забезпечення можна залучити розробника, тестувальника, менеджера проекту, а також автора

користувацької документації. Вони будуть виявляти наявність проблем різних типів (вимогу, котра не піддається перевірці, скоріш за все зможе виявити тестувальник, а розробник зможе визначити вимоги, котрі технічно не виконувані).

4. *Люди, які відповідають за роботу продуктів, що взаємодіють із елементом, що перевіряється.* Вони виявлять проблеми з вимогами до зовнішнього інтерфейсу, а також зможуть виявити вимоги, зміни яких у даній Специфікації впливають на інші системи.

Оптимальна команда має налічувати шість-сім членів. Якщо менше - це означає, що деякі точки зору не будуть представлені при кожній перевірці. Великі команди можуть легко загрузнути в сторонніх дискусіях, спробах вирішення проблем та дебатів про те, що вважається помилкою. При цьому знижується темп перевірки та зростає вартість виявлення кожного дефекту.

Всі учасники експертизи, включаючи автора, шукають недоліки та можливості покращення. Окремі учасники при цьому грають різноманітні ролі.

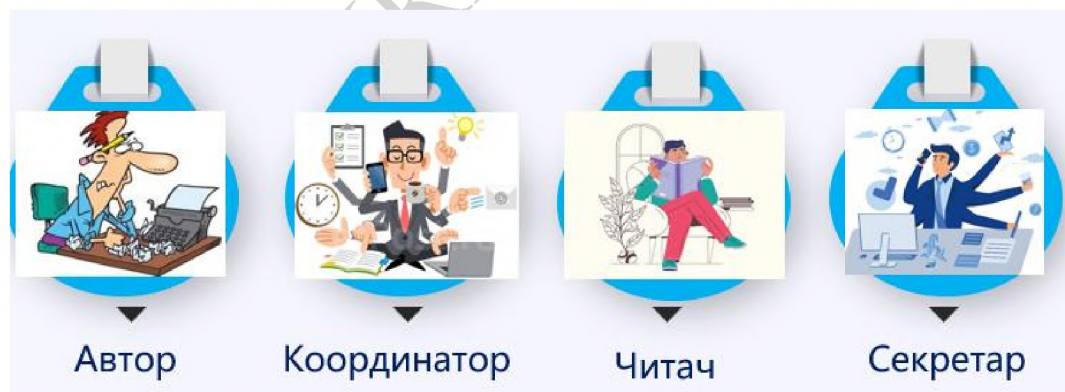


Рис. 8.2. Ролі учасників експертизи

Автор. Автор створює чи підтримує продукт, що перевіряється. В ролі автора Специфікації вимог до програмного забезпечення, як правило, виступає аналітик, котрий здійснює відбір вимог клієнта та створює Специфікацію.

Оскільки автор не приймає активну участь, він має можливість вислухати інших учасників інспектування, відповісти (але не вступати в дебати) на запитання та подумати. Досить часто автору вдається виявити помилки, які не помічені іншими учасниками.

Координатор. Координатор планує експертизу сумісно з автором, погоджує особливості та організовує наради, розподіляє матеріал, що перевіряється, між учасниками за декілька днів до наради, відповідає за своєчасний початок нарад, заохочення вкладу кожного учасника та управління дискусією, котра повинна бути направлена на виявлення дефектів, а не на рішення проблем. Крім того, в його обов'язки входить інформувати менеджерів чи відповідальну особу за збір результатів різноманітних експертиз, про результати. Координатор також працює разом з автором над пропозиціями зміни, щоб всі дефекти та проблеми, підняті під час інспектування, були розглянуті необхідним чином.

Читач. На нараді він перефразовує положення Специфікації - по одній вимозі за раз, щоб інші учасники вказували на потенційні дефекти та проблеми. Висловлюючи вимоги своїми словами, читач трактує вимоги з різницею від інтерпретації інших членів команди. Це хороший спосіб виявлення незрозумілостей, можливих недоліків чи пропозицій (важливо, щоб роль читача виконував не автор).

Секретар. Секретар використовує стандартні форми для документування виникаючих запитань і дефектів, виявлених під час нарад. Секретар повинен вголос прочитати свої записи, щоб впевнитись в їх точності, а інші учасники експертизи повинні допомогти йому зафіксувати зміст кожної проблеми таким чином, щоб автор зміг чітко вияснити природу та місцезнаходження кожної проблеми.

Вхідні критерії експертизи

Документ про вимоги готовий до експертизи, якщо він задовольняє напередвизначеним умовам. Набір вхідних критеріїв чітко вимальовує учасникам те, як варто підготуватися до рецензування, а також утримують команду від витрачання часу на проблеми, котрі варто вирішувати до початку експертизи. Координатор використовує вхідні критерії в якості контрольного списку до того, як прийняти рішення про початок рецензування. Ось декілька вхідних критеріїв для складання документації по вимогам:

- ✓ документ повинен відповідати шаблону;
- ✓ в документі виконана перевірка правопису;
- ✓ автор переглянув документ на наявність помилок в макеті;
- ✓ для експертизи підготовлені всі необхідні для перевірки документи, попередні документи чи документи, на котрі є посилання;
- ✓ всі невирішені запитання помічаються значком «TBD» (*to be determined*) (бути вирішеним);
- ✓ координатор виявляє не більше трьох суттєвих дефектів після 10-хвилинної перевірки обраного фрагменту документу.

Експертиза – це багатостадійний процес, як показано на рисунку 8.3.



Рис. 8.3. Етапи експертизи

Планування. Експертизу сумісно планують автор та координатор, визначаючи склад учасників, матеріали, котрі експерти повинні отримати до початку першої наради, і кількість нарад, необхідних для охоплення всього матеріалу. Кількість виявлених дефектів залежить від темпу роботи (чим повільніше вивчається Специфікація вимог до програмного забезпечення, тим більше дефектів вдається виявити). Досить поширено альтернативне трактування цієї зв'язки: «Темпи перевірки знижуються, якщо ви виявляєте багато помилок». Оскільки ні одна команда не може витратити безмежну кількість часу на перевірку вимог, слід обрати оптимальний темп, приймаючи до уваги ризик пропуску важливих дефектів. Частіше за все – від двох до чотирьох сторінок в годину, хоча оптимальна швидкість, при якій досягається максимальний ефект нижче приблизно в два рази.

Оглядова зустріч. Під час оглядової зустрічі автор описує вихідні дані матеріалу, призначеного для перевірки, припущення, котрі він робив, і його особисті цілі (якщо всі, хто перевіряє, добре знайомі з матеріалом, що перевіряється, тоді цю стадію можна пропустити).

Приготування. До наради кожний із експертів досліджує продукт, щоб визначити можливі дефекти та виникаючі запитання; для цього вони використовують контрольні списки типових дефектів (до 75% дефектів, що виявляються при експертизі, виявляються саме на етапі приготування, тому цей етап пропускати неможна). Для того, щоб допомогти експертам виявити типові помилки в інспектованих продуктах, розробляють контрольні списки дефектів для кожного типу документу про вимоги (за допомогою таких контрольних списків привертається увага експертів до часто виникаючих проблем з вимогами). Такі списки є досить великими, тому слід скорочувати їх відповідно до потреб компанії, і змінювати у відповідності з тим, які проблеми у вимогах повторюються частіше за все. Можна попросити деяких експертів при

приготуванні використовувати різні піднабори контрольних списків (наприклад, один експерт перевіряє, чи вірні перехресні посилання внутрішньої документації, другий – чи можуть вимоги слугувати основою для дизайну, а третій - оцінює перевірюваність вимог). Проведені дослідження показали, що такий розподіл обов'язків, коли експертів забезпечуються чіткими вказівками чи сценаріями, котрі допомагають їм при пошуку помилок, більш ефективний, ніж коли експерти працюють за однаковими контрольними списками.

Нарада. На нараді читач знайомить членів команди з кожною вимогою специфікації (по одній за раз), перефразовуючи їх іншими словами. По мірі обговорення дефектів та інших проблем, секретар заносить їх у форму, котра для автора вимог стає керівництвом до дій. Ціль наради: виявити в документі якомога більше помилок. Нарада не повинна тривати більше двох годин, оскільки роботу стомлених людей не можна назвати продуктивною (якщо ж потрібно більше часу, варто запланувати ще одну нараду). Наостанок наради команда приймає рішення: чи варто прийняти документ без змін, з незначними правками чи вказати на необхідність значної переробки. Рішення «необхідні значні зміни» свідчить про наявність серйозних проблем в процесі розробки вимог. Можливо варто вивчити вже пророблену роботу і спробувати знайти вихід до того, як приступати до наступного етапу роботи над Специфікацією вимог.

Іноді експерти повідомляють тільки про поверхневі та незначні проблеми. На додачу, вони легко відволікаються на дискусії про те, чи дійсно обговорюване запитання є дефектом, на суперечки про межі проекту та на обговорення можливих рішень проблем. Це все, звісно, корисно, однак ці дії відволікають їх від безпосередньої цілі наради – виявлення серйозних дефектів та можливостей покращення.

Деякі дослідники притримуються думки, що наради надто трудомісткі, проте на них можна виявити ті дефекти, котрі не вдається виявити експертам при індивідуальній роботі.

Переробка. Практично на кожному етапі роботи, пов'язаним із контролем якості, виявляються певні дефекти, тому після наради автору варто відвести визначений час на переробку вимог, оскільки виправлення в майбутньому дефектів вимог обійдеться дорожче, а тому саме зараз слід вирішити неясності, усунути двозначності та закласти основу успішного проєкту при розробці (немає змісту проводити перевірку, якщо після неї не будуть виправлені виявлені помилки).

Заключний етап. На цьому етапі експертизи координатор чи спеціально призначений член команди працює з автором, щоб впевнитись, що всі підняті проблеми вирішені та всі помилки виправлені відповідним чином. Заключний етап завершує процес експертизи, а координатор визначає, чи задовольняє експертиза вихідні критерії.

Проблеми при перегляді вимог

Експертні оцінки можна віднести як до технічних прийомів, так і до роботи з людьми. Прохання до колег висловити їх думку про те, що не так з вашою роботою, - свідоме, а не інтуїтивне дія. Нижче описані типові проблеми, з якими спеціалісти компанії зіштовхнуться при перевірці документації вимог і пропозиції по їх вирішенні.

Великий об'єм документації. Перспективу ретельної перевірки Специфікації вимог до програмного забезпечення, яка складається із декількох сотень сторінок, можна сміливо назвати страхітливою. Навіть в Специфікації середнього розміру, як правило, всі експерти ретельно перевіряють першу частину, декілька найбільш стійких вивчать середину, але мало імовірно, щоб хтось дійшов до кінця. Тому, щоб уникнути перевантажень, слід:

- ✓ перевіряти документ по мірі його розробки, не чекати, коли він буде завершений;
- ✓ визначити області підвищеного ризику, котрі вимагають ретельної експертизи (для менш важливих фрагментів вистачить і неофіційного перегляду);
- ✓ попросити експертів почати з різноманітних частин документу (це дозволить окинути кожен сторінку свіжим поглядом);
- ✓ використовувати декілька невеликих команд для перевірки різних фрагментів документу (однак при цьому є ризик не помітити непогоджованості);
- ✓ дослідити репрезентативну вибірку фрагментів, щоб оцінити, чи варто проводити експертизу всієї Специфікації (вивчивши кількість та типи виявлених помилок вибірки, визначите, чи варто проводити повну перевірку).

Велика команда експертів. Більшість учасників проєкту та клієнти хочуть займатись вимогами, тому список бажаючих може виявитись досить довгим. Однак чим більша команда експертів, тим важче скласти графік нарад, тим більше на них зайвих розмов і тим важче прийти до погодження по кожному питанню. Слід дотримуватись наступних порад при роботі з потенційно великими командами експертів:

- ✓ переконатися, що кожний учасник розуміє, що його задача – виявляти дефекти, а не підвищувати кваліфікацію чи відстоювати позицію;
- ✓ чітко уявити собі, чия точка зору (клієнта, розробника чи тестувальника) представляє кожний учасник і, можливо, тактовно відхилити участь осіб, котрі готові відстоювати свою точку зору, котра вже представлена (якщо в команді декілька людей відносяться до однієї групи, то вони можуть об'єднати свою інформацію і делегувати для участі в нараді когось одного);
- ✓ зібрати декілька невеликих груп для паралельної перевірки Специфікації вимог та поєднати їх контрольні списки дефектів в один, видаливши всі повтори. Дослідження показали, що експертиза документу про

вимоги, що виконується декількома командами, виявляє більше помилок, ніж одна велика команда і, як правило, результати паралельних перевірок скоріш доповнюють, ніж повторюють один одного.

Географічний розподіл рецензентів. Все більше компаній розробляють продукти, залучаючи команди, роз'єднані географічно (перевірки проводити важче). В такому випадку відеоконференції можуть стати ефективним рішенням, але на них не можна зрозуміти мову тіла та міміку інших учасників, як при особистих зустрічах. Перегляд електронного файлу, розміщеного в загальній мережевій папці, - метод, альтернативний традиційним нарадам експертів. В цьому випадку рецензенти використовують функції текстового редактора, щоб внести свої коментарі в документ, що перевіряється (кожний коментар помічається ініціалами рецензента, таким чином будь-хто може ознайомитись із думкою інших рецензентів).

8.3. Управління змінами вимог

Розробку вимог до програмного забезпечення можна розділити на дві стадії: розробку вимог та управління вимогами. Стадія розробки вимог включає збір інформації, аналіз, уточнення та погодження вимог (як правило, вона завершується створенням пакету документів: документу про бачення та межі продукту, документації до варіантів використання, Специфікації вимог до програмного забезпечення, словника даних та відповідних моделей аналізу). Цей пакет визначає базову версію вимог для погодження між розробниками та клієнтами.

Погодження вимог є зв'язною ланкою між розробкою вимог та управлінням ними (у членів команди розробників повинен бути доступ до поточних вимог протягом всього проєкту). Управління вимогами надає спосіб уникнути помилок, відстежуючи зміни вимог і сприяючи комунікації із

зацікавленими сторонами від початку проекту і протягом усього його життєвого циклу.

Управління вимогами (Requirements Management) є одним із ключових процесів інженерії програмного забезпечення, що забезпечує систематичне виявлення, документування, аналіз, узгодження, відстеження та контроль змін вимог протягом усього життєвого циклу програмного продукту. Як показано на рисунку 8.4, управління вимогами це: управління змінами базової версії вимог, підтримка планів проекту актуальними у відповідності зі змінними вимогами, управління версіями окремих вимог та документації вимог, контроль стану вимог в базовій версії та управління логічними зв'язками між окремими вимогами та іншими матеріалами для роботи з проектом.

Ефективне управління вимогами спрямоване на те, щоб програмний продукт відповідав потребам Замовників, користувачів і інших зацікавлених сторін, а також щоб вимоги залишалися актуальними, узгодженими та контрольованими в умовах змін.

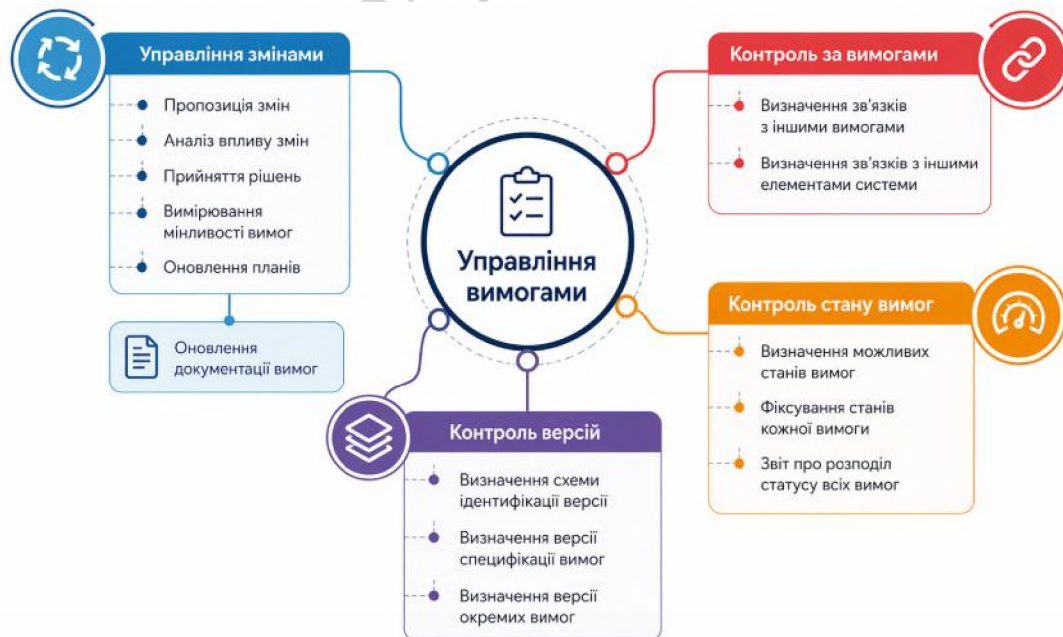


Рис.8.4. Основні складові управління вимогами

Насьогодні, вимоги у проєктах розроблення програмного забезпечення можуть змінюватися досить часто через зміну бізнес-потреб, технологічних умов або ринкових факторів. Саме тому управління вимогами розглядається як безперервний процес, який охоплює весь життєвий цикл системи - від початкового формування ідеї до супроводу та еволюції програмного продукту. У міжнародних стандартах, зокрема в ISO/IEC/IEEE 29148, управління вимогами визначається як сукупність процесів, що забезпечують контроль і підтримку вимог протягом їх життєвого циклу.

Управління вимогами охоплює роботу з різними типами вимог, серед яких функціональні, нефункціональні, бізнес-вимоги, вимоги користувачів та системні вимоги. Кожен із цих типів має власну роль у формуванні цілісного бачення майбутньої системи. Крім того, управління вимогами передбачає підтримку зв'язків між різними рівнями вимог, а також між вимогами та іншими артефактами проєкту, такими як архітектурні рішення, проєктна документація, тестові сценарії та результати перевірки системи.

Основні процеси управління вимогами

Процес управління вимогами зазвичай включає декілька взаємопов'язаних дій. Однією з ключових є *ідентифікація вимог*, що передбачає їх унікальне позначення, структурування та класифікацію. Це дозволяє ефективно організувати великий обсяг інформації та забезпечити можливість подальшого відстеження.

Наступним важливим аспектом є *підтримка трасованості вимог*, що означає можливість простежити походження кожної вимоги, її зв'язки з іншими вимогами, а також із відповідними компонентами системи, тестами та результатами перевірки. Завдяки цьому стає можливим оцінювати вплив змін на систему та контролювати повноту реалізації вимог.

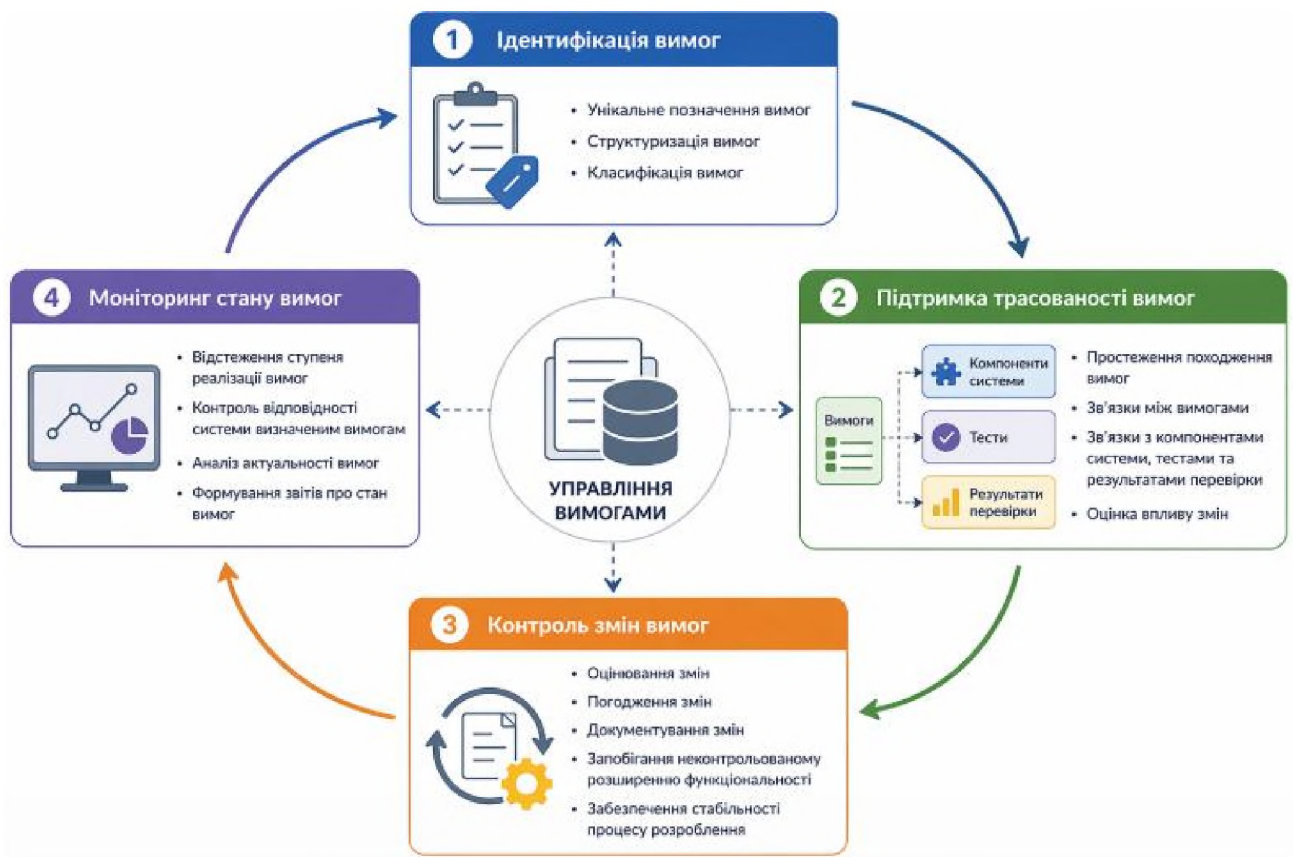


Рис. 8.5. Складові процесу управління вимогами

Ще одним важливим елементом управління є *контроль змін вимог*. У процесі розроблення програмного забезпечення вимоги часто уточнюються або змінюються, тому необхідно впроваджувати формалізовані процедури для оцінювання, погодження та документування змін. Це дозволяє уникнути неконтрольованого розширення функціональності системи та забезпечити стабільність процесу розроблення.

Крім того, управління вимогами передбачає *регулярний моніторинг* їх стану. Це включає відстеження ступеня реалізації вимог, контроль відповідності системи визначеним вимогам та аналіз їх актуальності.

Управління вимогами може реалізовуватися в межах різних моделей організації процесу розроблення програмного забезпечення. Вибір конкретної моделі залежить від масштабу проєкту, рівня невизначеності вимог, організаційної структури команди та обраної методології розроблення.

Каскадна модель (Waterfall). У межах цієї моделі управління вимогами здійснюється переважно на початкових етапах проєкту. Вимоги детально визначаються та документуються у Специфікації вимог до програмного забезпечення, після чого переходять до етапів проєктування та реалізації. Зміни вимог у цій моделі допускаються, але зазвичай супроводжуються значними витратами часу та ресурсів.

Ітеративна модель. Ітеративний підхід передбачає поступове уточнення та деталізацію вимог протягом кількох циклів розроблення, де на кожній ітерації частина вимог реалізується, перевіряється та уточнюється на основі отриманих результатів. Такий підхід дозволяє гнучко реагувати на зміни потреб користувачів.

Інкрементна модель. В інкрементній моделі система створюється у вигляді послідовності функціональних частин (інкрементів), де кожен інкремент реалізує певний набір вимог, що дозволяє поступово розширювати функціональність системи. Управління вимогами в цьому випадку передбачає визначення пріоритетів і планування реалізації вимог у різних інкрементах.

Agile-модель управління вимогами. У гнучких методологіях вимоги зазвичай формулюються у вигляді користувацьких історій і постійно уточнюються під час ітерацій. Управління вимогами відбувається через ведення беклогу продукту, регулярне уточнення вимог та їх пріоритизацію. Такий підхід дозволяє швидко реагувати на зміну бізнес-потреб і забезпечує тісну взаємодію з Замовником.

Спіральна модель. Ця модель поєднує елементи ітеративного підходу та управління ризиками. У межах кожного витка спіралі здійснюється аналіз вимог, оцінювання ризиків і уточнення функціональності системи. Такий підхід часто використовується у складних або інноваційних проєктах.

Метрики управління вимогами

Метрики управління вимогами використовуються для оцінювання ефективності процесу роботи з вимогами, контролю якості Специфікації та моніторингу стану реалізації вимог. Використання метрик дозволяє приймати обґрунтовані управлінські рішення та своєчасно виявляти проблеми в процесі розроблення.

✓ *Кількість вимог* характеризує загальну кількість вимог у системі та може використовуватися для оцінювання масштабу проєкту. Вона також дозволяє відстежувати зміну обсягу вимог протягом життєвого циклу системи.

✓ *Стабільність вимог* показує частоту їх змін у процесі розроблення (висока частота змін може свідчити про недостатню якість початкового аналізу вимог або про динамічність бізнес-середовища).

✓ *Покриття вимог тестами* відображає частку вимог, для яких розроблено відповідні тестові сценарії (високий рівень покриття тестами свідчить про належний рівень перевірки реалізації вимог).

✓ *Трасованість вимог* визначає, яка частка вимог має встановлені зв'язки з іншими артефактами проєкту, такими як елементи архітектури, модулі програмного коду або тестові сценарії.

✓ *Щільність дефектів вимог* показує кількість помилок або дефектів, виявлених у вимогах, відносно їх загальної кількості (допомагає оцінити якість Специфікації вимог та ефективність процесу їх перевірки).

✓ *Час обробки змін вимог* відображає середній час, необхідний для аналізу, погодження та впровадження змін до вимог (дозволяє оцінити ефективність процесу управління змінами).

8.4. Управління ризиками

Програмні інженери - вічні оптимісти, вони часто очікують, що наступний проєкт пройде гладко, не дивлячись на велику кількість проблем в минулих проєктах. Всупереч різним упевненостям, менеджери проєктів з розробки програмного забезпечення *повинні виявляти ризик та управляти ним*, починаючи із факторів, пов'язаних із вимогами.

Ризик - це умова, котра може спричинити якісь втрати чи іншим способом поставити під загрозу успіх проєкту (умова, яка ще не породила проблему). Це потенційні проблеми, які можуть проявити вплив на вартість, строки, технічний успіх, якість продукту та ефективність роботи команди.

Управління ризиком (краща практика індустрії програмного забезпечення) – це процес виявлення, оцінки та управління ризиком до того, як він принесе збиток проєкту (якщо щось недобре вже відбулось з проєктом, то це - проблема, а не ризик).



Оскільки неможливо впевнено передбачити майбутнє, управління ризиком – це спосіб мінімізації імовірності потенційних проблем чи збитку від них. Управління ризиком означає роботу над потенційною небезпекою до того, як вона перейде в кризову фазу, щоб покращити імовірність успіху проєкту та зменшити фінансові чи інші наслідки ризику, котрого не вдасться уникнути.

Оскільки вимоги такі важливі в проєктах по створенню програмного забезпечення, завбачливий менеджер вже на ранніх стадіях проєкту виявить ризик, пов'язаний із ними, і буде достатньо агресивно контролювати його. Розповсюджені фактори ризику, пов'язані із вимогами, включають невірне розуміння вимог, недостатнє залучення користувачів, неточності чи зміни в

масштабах та цілях проєкту, та постійно змінні самі вимоги. Аналітики (менеджери проєктів) можуть керувати ризиком, який пов'язаний із вимогами, тільки у співробітництві із клієнтами та їх представниками. Сумісне документування факторів ризику, пов'язаних із вимогами і планування дій по зменшенню їх наслідків зміцнює партнерство Замовника та Виконавця.

Складові
управління
ризиком

Управління ризиком (*Risk Management*) передбачає застосування інструментальних засобів та процедур для обмеження факторів ризику в проєкті прийнятними рамками. Управління ризиком надає стандартний підхід до виявлення та документування факторів ризику, оцінці їх потенційного збитку та пропонує стратегії їх пом'якшення. Управління ризиком включає в себе дії, показані на рисунку 8.6.

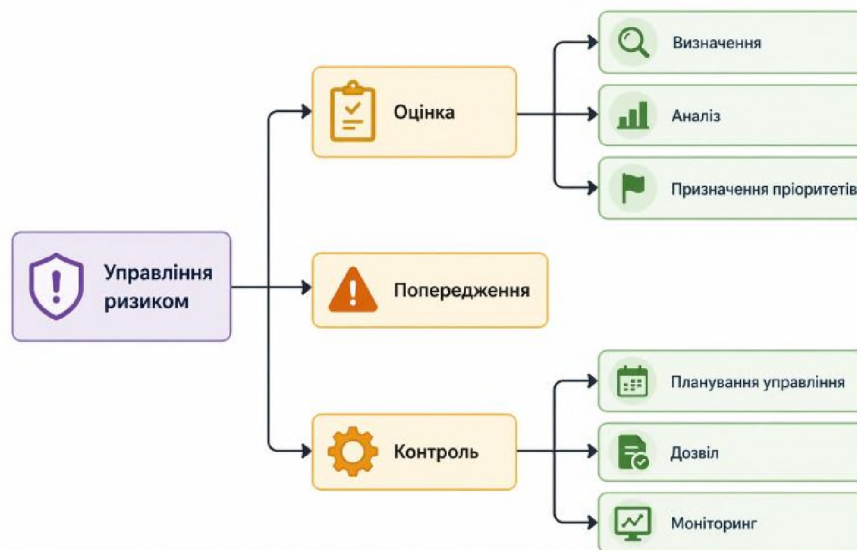


Рис. 8.6. Елементи управління ризиком

Оцінка ризику (Risk Assessment) – це процес дослідження проєкту для виявлення областей потенційного ризику. Полегшити задачу виявлення факторів ризику можна, складаючи списки факторів ризику, типових для розробки програмного забезпечення, у тому числі пов'язаних із вимогами. При *аналізі*

*ризик*у досліджуються потенційні наслідки конкретних факторів ризику для проєкту, а *визначення пріоритетів* факторів ризику допоможе сконцентруватись на найбільш небезпечних, оцінюючи потенційну схильність кожного з них (схильність – це функція, яка включає як імовірність зазнати збиток через ризик, так і масштаби цього збитку).

Попередження ризику (Risk Avoidance) – це один із способів вирішення питання: не робити того, що ризиковано (можна уникнути ризику, не починаючи певні проєкти, спираючись на випробування, а не на передові технології чи виключаючи функції, котрі будуть особливо важко правильно впровадити). Проте розробка програмного забезпечення сама по собі ризикована дія, так що уникнути ризику - означає втратити можливість.

Більшу частину часу приходиться виконувати *контроль ризику (Risk Control)*, що дозволяє управляти виявленими факторами ризику із високим пріоритетом. *Планування управління ризиком* полягає у створенні плану дій для кожного окремого фактору, включаючи методи пом'якшення, плани на випадок непередбачуваних обставин, відповідальних осіб та строки виконання. Ціль дій із пом'якшення впливу ризику – або не дозволити ризику стати проблемою, або зменшити його негативний вплив. Відстеження свого просування до вирішення кожного ризику відбувається за допомогою *моніторингу ризику*. Слід відстежувати, наскільки добре працюють дії із пом'якшення ризику, шукати нові фактори ризику, які виникли недавно, забирати ризики, загроза яких минула, та періодично оновлювати пріоритети у списку факторів ризику.

Фактори ризику групуються за приналежністю до основних етапів конструювання вимог: виявленню, аналізу, документуванню, перевірці та управлінню. Нижче пропонуються методи, які зменшать імовірність або вплив ризику.

Виявлення вимог

Бачення та межі проєкту. Розходження меж стає більш імовірним, якщо у зацікавлених у проєкті осіб немає ясної, єдиної думки про те, що продукт повинен із себе представляти і робити.

Починаючи проєкт, слід скласти документ про бачення та межі, котрий містить бізнес-вимоги, і використовувати його при виробленні рішень про прийняття чи зміну вимог.

Час, витрачений на розробку вимог. Жорсткі строки проєкту часто змушують менеджерів та клієнтів нехтувати складанням вимог (вони рахують, що якщо розробники не почнуть працювати над кодом негайно, то не завершать вчасно). Хоча складність проєктів широко варіюється в залежності від розміру та класу додатку (наприклад, інформаційні системи, системне програмне забезпечення, комерційне чи військове), але за самими грубими підрахунками варто витратити 10-15% ресурсів проєкту на дії з розробки вимог.

Повнота та коректність Специфікації вимог. Щоб вимоги точно визначали потреби клієнта, слід застосовувати метод варіантів використання, щоб виявити вимоги, концентруючись на користувацьких задачах; складати конкретні сценарії використання, писати варіанти тестування на основі вимог та просити клієнтів розробляти свої критерії прийняття, створити прототипи, щоб вимоги були зрозумілі користувачам і щоб отримувати від них конкретні відгуки. Також слід залучити представників клієнтів до експертизи Специфікації вимог та моделей аналізу.

Вимоги для суперсучасних продуктів. Легко помилитись в оцінці реакції ринку на продукти, перші в своєму роді, тому слід приділити велику увагу дослідженню ринку, створити прототипи та використовувати фокус-групи користувачів, щоб отримувати ранню та зворотну реакцію відносно пропонованого продукту.

Визначення нефункціональних вимог. Оскільки основна увага, звісно, направлена на функціональність продукту, легко випустити з виду нефункціональні вимоги. Тому слід запитувати клієнтів про якісні характеристики, такі як продуктивність, легкість та простота використання, цілісність та надійність; документувати ці нефункціональні вимоги та критерії їх прийняття як можна точніше в Специфікації вимог.

Єдність думок клієнтів відносно вимог до продукту. Якщо різні клієнти продукту не досягли згоди відносно того, що саме повинно бути розроблено, хтось з них залишиться незадоволений результатом. В такому випадку слід визначити основних клієнтів та використати прибічників продукту для отримання адекватного представництва та залучення клієнтів (але слід переконатись, що покладаєтесь на думку правильно обраних людей, котрі мають повноваження для прийняття рішень).

Використання існуючого продукту в якості базової версії. Досить часто розробка вимог рахується не важливою в проєкті наступного покоління чи при переробці попередніх проєктів і тоді розробники часто використовують існуючий продукт в якості джерела вимог, зі списком змін та доповнень, а це примушує їх збирати вимоги через інженерний аналіз поточного проєкту. Проте це не ефективний і не повний спосіб виявлення вимог, тому і не дивно, якщо у новій системі проявляються деякі недоліки вихідної системи (слід документувати вимоги, вилучені таким способом, і просити клієнтів перевірити їх, щоб впевнитись, що вони вірні і до сих пір актуальні).

Аналіз вимог

Визначення пріоритетів вимог. Слід переконатися, що кожна функціональна вимога, функція чи варіант використання отримали пріоритет і приписані до конкретної версії чи ітерації системи, а також оцінити пріоритет кожної нової вимоги по відношенню до об'єму роботи,

що залишилась, щоб мати можливість приймати розумні рішення про компроміси.

Технічно складні функції. Слід оцінити здійсненність кожної вимоги, щоб визначити ті з них, реалізація яких може зайняти більше часу, ніж очікувалось. Тому необхідно трасувати статус проєкту, щоб слідкувати за вимогами, реалізація котрих відстає від графіку.

Незнайомі технології, методи, мови, інструменти чи апаратне забезпечення. Не потрібно недооцінювати криву навчання у роботі із новими методами, необхідними для задоволення деяких вимог (необхідно виявити ці ризиковані вимоги на початку проєкту та виділити достатньо часу на фальстарт, навчання, експерименти та прототипування).

Специфікація вимог

Розуміння вимог. Різниця в інтерпретації вимог Виконавця та Замовника призводить до розходження в очікуваннях. Формальна експертиза документації вимог розробниками, тестувальниками і Замовниками зменшує цей ризик (підготовані, досвідчені аналітики вимог можуть задати Замовникам вірні запитання та скласти високоякісні специфікації). Моделі та прототипи, які представляють вимоги із різних точок зору, також дозволяють виявити неясні, невизначені вимоги.

Поспіх, який змушує пропускати помітки «TBD». Корисно виділяти області Специфікації вимог, котрі потребують доробки спеціальними помітками, наприклад «TBD», але ризиковано починати конструювання, доки вони не зняті.

Неоднозначна термінологія. Необхідно створити словник для визначення бізнес чи технічних термінів, котрі можуть бути трактовані різними читачами по-різному, зокрема, визначити терміни, які мають як загальне, так і технічне чи вузькоспеціалізоване значення. Також слід створити словник даних, де визначаються елементи та структури даних.

Включення дизайну у вимоги. Опис дизайну в Специфікації вимог накладає непотрібні обмеження на можливості, доступні розробникам, чим стримують створення оптимальних конструкцій. Слід переконатися, що вимоги описують, що необхідно зробити для рішення бізнес-проблеми, а не те, як це повинно бути зроблено.

Погодження вимог

Непогодженні вимоги. Перспектива погодження довгої Специфікації вимог лякає, як і думка про написання варіантів тестування в самому початку процесу розробки (якщо підтвердити коректність та якість вимог до початку конструювання, то можна уникнути значної та дорогої переробки на більш пізніх стадіях проєкту). Необхідно зобов'язати представників клієнтів приймати участь в перевірці вимог, тому що тільки вони можуть судити, чи задовольняють сформульовані вимоги їх потреби. Також слід проводити покрокові, неформальні експертизи, щоб виявляти проблеми якомога раніше і дешевше.

Якість перевірки. Якщо рецензенти не знають, як проводити експертизу документації вимог та які дії ефективні при перевірці, вони можуть упустити серйозні дефекти. Тому необхідно навчити всіх членів команди, котрі будуть приймати участь в експертизі (можна запросити людину із досвідом перевірки для спостереження і, можливо, координації та тренування учасників).

Управління вимогами

Зміна вимог. Можна зменшити розходження меж проєкту, використовуючи документ про його бачення та межі в якості контрольних точок для погодження змін. Поєднання виявлення вимог із існуючою участю користувачів може зменшити розходження вимог приблизно вполовину. Методи контролю якості, які визначають помилки у вимогах на ранніх стадіях, зменшують кількість наступних модифікацій (щоб зменшити вплив зміни вимог, слід відкласти реалізацію тих з них, котрі, скоріш

за все, зміняться, доки вони не будуть визначені), а також проєктувати систему, закладаючи можливість легкої модернізації.

Нереалізовані вимоги. Матриця трасованості вимог допомагає уникнути пропуску будь-яких вимог під час проєктування, конструювання та тестування.

Збільшення об'єму проєкту. Якщо вимоги першочергово погано визначені, подальше їх виявлення може збільшити об'єм проєкту. Реалізація нечітко визначених областей продукту потребує більше зусиль, ніж очікувалось. Ресурси проєкту, які розподілені у відповідності з початковими неповними вимогами, можуть виявитись недостатніми для задоволення повного спектру потреб користувача. Для пом'якшення цього ризику слід спланувати життєвий цикл, який складається з поетапних версій (реалізація базової функціональності в першому випуску та подальше нарощення здатності системи).

Питання для самоперевірки



1. У чому полягає сутність погодження вимог у процесі інженерії вимог?
2. Які відмінності між валідацією та верифікацією вимог?
3. Як визначається поняття «ризик» у контексті ІТ-проєктів?
4. Чому непогоджені вимоги є джерелом ризиків?
5. Які основні характеристики якісної вимоги?
6. Які типи вимог потребують найбільш складного погодження і чому?
7. У чому полягає взаємозв'язок між невизначеністю та ризиком?
8. Яку роль відіграє документування вимог у зниженні ризиків?
9. Чим відрізняється стратегічний ризик від операційного в контексті вимог?
10. Які міжнародні стандарти регламентують управління ризиками?
11. Хто є ключовими стейкхолдерами процесу погодження вимог?
12. Як ідентифікувати приховані інтереси зацікавлених сторін?
13. Чому конфлікти інтересів є неминучими в процесі погодження?

14. Які комунікаційні бар'єри можуть створювати ризики?
15. Яку роль виконує бізнес-аналітик у досягненні консенсусу?
16. Як фасилітація сприяє зменшенню ризиків непорозуміння?
17. Чим небезпечна «ілюзія спільного розуміння»?
18. Як культурні особливості команди впливають на процес погодження?
19. Яким чином неформальні домовленості можуть створювати ризики?
20. Як побудувати ефективну комунікаційну стратегію в проєкті?
21. Які етапи включає процес погодження вимог?
22. Які методи візуалізації сприяють узгодженню вимог?
23. Чому прототипування зменшує ризик помилкової інтерпретації?
24. Яким чином критерії приймання впливають на якість погодження?
25. Як забезпечити однозначність формулювання вимог?
26. У чому полягає роль трасованості у зниженні ризиків?
27. Як зміни вимог впливають на ризиковий профіль проєкту?
28. Чому контроль версій вимог є критично важливим?
29. Які інструменти підтримують управління вимогами?
30. Як інтегрувати погодження вимог у гнучкі методології?
31. Які етапи процесу управління ризиками?
32. У чому різниця між якісним і кількісним аналізом ризиків?
33. Як будується матриця «ймовірність — вплив»?
34. Що таке реєстр ризиків і яку інформацію він містить?
35. Які стратегії реагування на ризики існують?
36. Коли доцільно приймати ризик, а коли — уникати його?
37. Як оцінити вплив ризику на бюджет і строки?
38. Які ризики найчастіше виникають на етапі аналізу вимог?
39. Як впровадження *DevOps* впливає на управління ризиками?
40. Яким чином автоматизоване тестування зменшує ризики?
41. Як оцінити ефективність процесу погодження вимог?

42. Чи може надмірна формалізація підвищити ризики? Обґрунтуйте.
43. Які когнітивні упередження впливають на оцінювання ризиків?
44. Як виміряти рівень невизначеності вимоги?
45. У чому полягає економічна доцільність раннього управління ризиками?
46. Як зміна бізнес-стратегії впливає на систему погоджених вимог?
47. Які наслідки ігнорування регуляторних ризиків?
48. Які метрики можна використовувати для моніторингу ризиків?
49. Які принципи організаційної культури сприяють ефективному погодженню вимог?
50. Що таке управління вимогами і яку роль воно відіграє у процесі розроблення програмного забезпечення?
51. Які основні завдання та цілі процесу управління вимогами в програмному проєкті?
52. Які основні принципи лежать в основі ефективного управління вимогами?
53. Що таке трасованість вимог і чому вона є важливою для забезпечення якості програмного продукту?
54. Які основні етапи або процеси включає управління вимогами протягом життєвого циклу програмного забезпечення?
55. Які методи та підходи використовуються для контролю змін вимог у процесі розроблення програмного забезпечення?
56. Які моделі управління вимогами застосовуються у сучасних методологіях розроблення програмного забезпечення?
57. Які метрики можуть використовуватися для оцінювання ефективності процесу управління вимогами?
58. Яку роль відіграють зацікавлені сторони у процесі управління вимогами?

Основні типи вимог

Бізнес-вимоги (*Business Requirements*) - визначають стратегічні цілі та задачі бізнесу, які повинна підтримувати система.

Вимоги користувача (*User Requirements*) - високорівневий опис потреб, цілей і очікувань користувачів щодо системи, сформульований зрозумілою для них мовою без технічних деталей.

Вимоги системного рівня (*System Requirements*) - деталізований опис функцій, поведінки та обмежень системи як цілого, що слугує основою для розробки та тестування.

Операційні вимоги (*Operational Requirements*) - визначають умови функціонування системи: середовище, режими роботи, процедури експлуатації та обслуговування.

Архітектурні вимоги (*Architectural Requirements*) - встановлюють принципи побудови системи, структуру компонентів, їх взаємодію та технологічні обмеження.

Структурні вимоги (*Structural Requirements*) - описують внутрішню організацію системи, включаючи модулі, підсистеми та їх взаємозв'язки.

Функціональні вимоги (*Functional Requirements*) - чітко визначають, які функції, операції та сценарії використання повинна підтримувати система.

Нефункціональні вимоги (*Non-functional Requirements*) - описують якісні характеристики системи: продуктивність, безпека, надійність, масштабованість тощо.

Якісні характеристики

Вимоги до продуктивності (*Performance Requirements*) - визначають швидкодію системи, час відгуку, обсяг оброблюваних даних і допустиме навантаження.

Вимоги дизайну (*Design Requirements*) - описують принципи візуального оформлення, стилістику, структуру інтерфейсу та відповідність стандартам дизайну.

Інтерфейс користувача (*User Interface, UI*) - сукупність елементів (екрани, кнопки, меню), через які користувач взаємодіє із системою.

Досвід взаємодії (*User Experience, UX*) - загальне сприйняття користувачем системи, включаючи зручність, ефективність, емоції та задоволення від використання.

Моделі та артефакти

Життєвий цикл вимог (*Requirements Life Cycle, RLC*) - сукупність етапів роботи з вимогами: збір, аналіз, документування, перевірка, зміни та супровід.

Життєвий цикл програмного забезпечення (*Software Development Lifecycle, SDLC*) - повний процес створення ПЗ: від планування і аналізу до розробки, тестування, впровадження та підтримки.

Діаграма варіантів використання (*Use Case Diagram*) - візуальна модель, що показує акторів системи та їх взаємодію з функціями системи.

Діаграма активності (*Activity Diagram*) - графічна модель, що відображає послідовність дій, потоків керування та паралельних процесів у системі або бізнес-процесі. Використовується для опису алгоритмів, сценаріїв роботи та логіки виконання операцій.

Діаграма станів (*State Diagram*) - описує можливі стани об'єкта або системи та переходи між ними залежно від подій. Дозволяє моделювати поведінку системи в різних ситуаціях і життєвих циклах.

Діаграма послідовності (*Sequence Diagram*) - відображає взаємодію між об'єктами або компонентами у часовій послідовності. Показує, як повідомлення передаються між учасниками для виконання певного сценарію.

Нотація моделювання бізнес-процесів (*Business Process Model and Notation, BPMN*) - стандарт графічного моделювання бізнес-процесів, який дозволяє описувати потоки робіт, події, задачі та ролі учасників у зрозумілому як для бізнесу, так і для технічних спеціалістів вигляді.

Матриця якості вимог (*Requirements Quality Matrix, RQM*) - інструмент оцінювання вимог за критеріями якості (повнота, однозначність, тестованість тощо).

Матриця трасування вимог (*Requirements Traceability Matrix, RTM*) - таблиця, що відображає зв'язки між вимогами, тестами, реалізацією та бізнес-цілями.

Ролі учасників проєкту

Власник продукту (*Product Owner, PO*) - представник бізнесу, який визначає пріоритети, формує бачення продукту та керує беклогом.

Менеджер продукту (*Project Manager, PM*) - фахівець, який відповідає за планування, організацію та контроль виконання проєкту. Координує команду, управляє ресурсами, термінами, ризиками та комунікацією зі стейкхолдерами, забезпечуючи досягнення цілей проєкту в межах бюджету і часу.

Бізнес-аналітик (*Business Analyst, BA*) - фахівець, що збирає, аналізує, уточнює та документує вимоги, забезпечуючи їх зрозумілість для команди.

Розробник (*Developer, Dev*) - інженер, який реалізує вимоги у вигляді програмного коду.

Тестувальник (*Tester / QA Engineer*) - спеціаліст, що перевіряє відповідність системи вимогам і виявляє дефекти.

Agile-артефакти

Історія користувача (*User Story*) - короткий опис функціональності у форматі «Як [роль], я хочу [мета], щоб [цінність]».

Спринт (*Sprint*) - короткий ітераційний період (зазвичай 1–4 тижні), протягом якого створюється інкремент продукту.

Список робіт для продукту (*Product Backlog*) - пріоритезований перелік усіх вимог, задач і ідей щодо продукту.

Епік (*Epic*) - велика вимога або функціональність, що розбивається на менші історії користувача.

Функціональність (*Feature*) - завершена можливість системи, яка приносить цінність користувачу.

Мінімально життєздатний продукт (*Minimum Viable Product, MVP*) - початкова версія продукту з мінімальним набором функцій для перевірки гіпотез і отримання зворотного зв'язку.

Інкремент продукту (*Product Increment*) - частина готового функціоналу, створена за ітерацію та придатна до використання.

Якісні вимоги до ПЗ

Вимоги продуктивності (*Performance Requirements*) - визначають ефективність роботи системи під навантаженням.

Вимога надійності (*Reliability Requirement*) - характеризує стабільність і безвідмовність роботи системи протягом часу.

Вимога безпеки (*Security Requirement*) - визначає механізми захисту даних, доступу та протидії загрозам.

Вимога зручності використання (*Usability Requirement*) - описує простоту навчання, зрозумілість і комфорт використання.

Вимога масштабованості (*Scalability Requirement*) - визначає здатність системи ефективно працювати при зростанні навантаження.

Вимога супровідності (*Maintainability Requirement*) - визначає легкість внесення змін, виправлення помилок та розвитку системи.

Вимога портативності (*Portability Requirement*) - визначає можливість перенесення системи між різними платформами.

Вимога сумісності (*Compatibility Requirement*) - забезпечує взаємодію з іншими системами або компонентами.

Вимога доступності (*Availability Requirement*) - визначає рівень готовності системи до роботи.

Валідація, тестування та якість

Тестованість вимоги (*Requirement Testability*) - властивість вимоги, що дозволяє перевірити її виконання за допомогою чітких критеріїв.

Відстежуваність вимоги (*Requirement Traceability*) - здатність простежити вимогу від джерела до реалізації і тестування.

Тестовий сценарій (*Test Case*) - узагальнений опис перевірки певної функції або поведінки системи.

Валідація вимог (*Validate Requirements*) - процес підтвердження, що вимоги відображають реальні потреби користувачів і бізнесу та забезпечують створення потрібного продукту.

Верифікація вимог (*Verify Requirements*) - процес перевірки правильності, повноти, узгодженості та формальної якості вимог.

Документація та інструменти

Специфікація вимог до ПЗ (*Software Requirements Specification, SRS*) - формальний документ, що містить повний, структурований і погоджений опис усіх вимог до системи.

Документ бачення (*Vision Document*) - документ високого рівня, що описує цілі створення системи, її основну ідею, цінність для користувачів і бізнесу, ключові можливості та очікувані результати.

Документ меж (системи / проєкту) (*Scope Document*) - документ, що визначає межі системи або проєкту: що входить у розробку, а що виключається.

Прототип (*Prototype*) - попередня модель системи (низької або високої точності), що використовується для перевірки ідей і уточнення вимог.

Пріоритет вимоги (*Requirement Priority*) - характеристика вимоги, яка визначає ступінь її важливості та черговість реалізації відносно інших вимог. Пріоритизація допомагає приймати рішення щодо розподілу ресурсів, планування релізів і формування беклогу, враховуючи бізнес-цінність, ризики, залежності та обмеження.

Карта історій користувача (*User Story Map*) - візуальний інструмент організації історій користувача за сценаріями та пріоритетами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. «Що таке аналіз вимог? Процес і методи» [Електронний ресурс] - Режим доступу: <https://visuresolutions.com/uk/alm-guide/>
2. «Стадії циклу розробки ПЗ» [Електронний ресурс] - Режим доступу: <https://qalight.ua/baza-znaniy/stadiyi-tsiklu-rozrobki-pz/>
3. Ляшенко Л.А., Хіцко Я.В. Методичні рекомендації до виконання курсового проектування з дисципліни «Компоненти програмної інженерії»; навчальний посібник. – Київ: КПІ ім.Ігоря Сікорського, 2020. - 63с.
4. «Специфікація вимог до програмного забезпечення (SRS): Поради та шаблон» [Електронний ресурс] - Режим доступу: <https://visuresolutions.com/uk/>
5. Грицюк Ю.І. Аналіз вимог до програмного забезпечення. Львів: Львівська політехніка. 2018. - 456с.
6. «Що включає в себе аналіз вимог?» [Електронний ресурс] - Режим доступу: <https://testmatick.com/uk/shho-vklyuchaye-v-sebe-analiz-vymog/>
7. «Аналіз та збирання вимог» [Електронний ресурс] - Режим доступу: <https://studfile.net/preview/10015384/page:2/>
8. Грицюк Ю.І., Лешкевич І.Ф. Особливості визначення вимог до програмного забезпечення та проблеми їх аналізу. Науковий Вісник НЛТУ України. Вип. 27(4), 2017. - с.148-158.
9. «Визначення вимог до програмних систем» [Електронний ресурс] - Режим доступу: https://elib.lntu.edu.ua/sites/default/files/elib_upload
10. Михайлуца О. М., Скрипник І. А. Аналіз вимог до програмного забезпечення : навчально-методичний посібник для здобувачів ступеня вищої освіти бакалавра спеціальності 121 «Інженерія програмного забезпечення»



освітньо-професійної програми «Програмне забезпечення систем». Запоріжжя: ЗНУ, 2022. - 175 с.

11. «Як виглядає специфікація вимог і як її тестувати» [Електронний ресурс] – Режим доступу: <https://training.qatestlab.com/blog/technical-articles/what-the-requirements-specification-looks-like-and-how-to-test-it/>

12. «Аналіз програмних вимог із прикладом» [Електронний ресурс] – Режим доступу: <https://www.guru99.com/uk/learn-software-requirements-analysis-with-case-study.html>

13. Грицюк Ю.І., Немова Е.А. Особливості формулювання вимог до програмного забезпечення. Науковий Вісник НЛТУ України. Вип. 28(7), 2018. - с.135-148.

14. «Документування вимог: не що, а для чого і чому» [Електронний ресурс] - Режим доступу: <https://dou.ua/forums/topic/39737/>

15. «Верифікація та валідація вимог у розробці програмного забезпечення» [Електронний ресурс] - Режим доступу: <https://visuresolutions.com/uk/alm-guide>

16. «Критерії якості вимог до ІТ-продукту: що це, які існують і навіщо потрібні?» [Електронний ресурс] - Режим доступу: <https://iampm.club/ua/blog/kriteriyi-yakosti-vimog-do-it-produktu-shho-cze-yaki-isnuyut-i-navishho-potribni/>

17. «Інженерія вимог» [Електронний ресурс] - Режим доступу: <https://www.scribd.com/document/705631102>

18. «Інженерія вимог» [Електронний ресурс] - Режим доступу: <https://studfile.net/preview/10015384/page:3/>

19. «Тестування вимог - перший крок в роботі над проектом» [Електронний ресурс] – Режим доступу: <https://dou.ua/lenta/articles/test-requirements/>

20. «Виявлення вимог: як дізнатися, що потрібно замовнику» [Електронний ресурс] - Режим доступу: <https://iampm.club/ua/blog/viyavlennya-vimog-yak-diznatisya-shho-potribno-zamovniku/>
21. «Ефективний спосіб виявити вимоги в ІТ проектах» [Електронний ресурс] - Режим доступу: <https://blog.natec.tech/uk/yakij-efektivnij-sposib-viyaviti-vimogi-v-it-proektah/>
22. «Функціональні та нефункціональні вимоги (з прикладами)» [Електронний ресурс] - Режим доступу: <https://visuresolutions.com/uk/alm-guide>
23. «Ключова різниця між функціональними та нефункціональними вимогами» [Електронний ресурс] – Режим доступу: <https://www.guru99.com/uk/functional-vs-non-functional-requirements.html>
24. «Нефункціональні вимоги; приклади, типи, підходи» [Електронний ресурс] - Режим доступу: <https://training.qatestlab.com/blog/technical-articles/non-functional-requirements-examples-types-approaches/>
25. «Розбираємо ази архітектури й не функціональні вимоги» [Електронний ресурс] - Режим доступу: <https://dou.ua/forums/topic/49766/>
26. «Поняття та види вимог до програмного продукту» [Електронний ресурс] - Режим доступу: <https://intro.beetroot.academy/ua-qa-lessons/itit-qa-lesson-2>
27. «Матриця «Будинок якості»: як перетворити потреби клієнта на продукт» [Електронний ресурс] - Режим доступу: <https://speka.ua/business/matricya-budinok-yakosti-yak-peretvoriti-potrebi-klijenta-na-produkt-pj2eg4>
28. «Метод розгортання функції якості QFD» [Електронний ресурс] – Режим доступу: <https://studfile.net/preview/9360330/>
29. Коваль Г.І., Мороз Г.Б. Моделювання вимог до якості програмних систем оброблення даних. Методи та засоби програмної інженерії. [Електронний ресурс] - Режим доступу:

<https://nasplib.isofts.kiev.ua/server/api/core/bitstreams/fale1111-4336-4d41-b17e-4922115e94d5/content>

30. «Процес моделювання предметної області. Гайд для бізнес-аналітиків» [Електронний ресурс] – Режим доступу: <https://dou.ua/forums/topic/42366>

31. Говорущенко Т.О., Поморова О.В. Моделювання процесу оцінювання достатності інформації щодо якості у специфікаціях вимог до програмного забезпечення. Вісник Хмельницького національного університету. Технічні науки. 2017. № 6. – с.70-80

32. «Моделювання процесів та вимог до інноваційного продукту» [Електронний ресурс] – Режим доступу: <https://www.scribd.com/document/995704292>

33. «Фаза моделювання у ERP проектах» [Електронний ресурс] – Режим доступу: <https://tqm.com.ua/ua/likbez/ua-articles/faza-modeluvannya-v-erp-proekтах>

34. Кузь М.В., Незамай Б.С., Ровінський В.А., Подубинська Н.Д. Прогнозування якості програмних засобів на основі аналізу якості вимог. Методи та прилади контролю якості. Вип.1(50). 2023. – С.101-112

35. Комлева Н.О. Метод трансформації формалізованих вимог у розширені UML-моделі на основі граматичного підходу. Вісник Херсонського національного технічного університету. Том 2 №3(94). 2025. – с. 235-242

36. «Уніфікована мова моделювання (Unified Modeling Language – UML)» [Електронний ресурс] – Режим доступу: <https://www.maxzosim.com/unifikovana-mova-modeluvannia/>

37. Комлева Н.О. Розширення базису UML для підтримки недосконалих та змінюваних вимог у проектуванні інтелектуальних систем. Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки. Том 36 (75) №4. 2025. – с.170-176

38. Нестеренко О.В. Метод пріоритезації вимог в прорамній інженерії. Проблеми програмування. 2024. №2-3. – с.132-139

39. M. Sufian, Z. Khan, S. Rehman, W. Haider Butt, A systematic literature review: Software requirements prioritization techniques, in: Proceedings of the 2018 International Conference on Frontiers of Information Technology (FIT 2018), 2018, pp. 35–40.
40. H. Ahuja, Sujata, G. N. Purohit, Understanding requirement prioritization techniques, in: Proceeding of the IEEE International Conference on Computing, Communication and Automation (ICCCA 2016), 2016, pp. 257-262.
41. R. Berntsson Svensson, R. Torkar, Not all requirements prioritization criteria are equal at all times: A quantitative analysis, in: Journal of Systems and Software, 2024, 209, art. no. 111909.
42. S. Ahmad, R. Rizawanti, T. Woodings, I. Ermahani A. Jalil, MCBRank Method to Improve Software Requirements Prioritization, in: International Journal of Advanced Computer Science and Applications (IJACSA), 2022, 13(7), pp. 215-222.
43. H. Arshad, S. Shaheen, J.A. Khan et al., A novel hybrid requirement's prioritization approach based on critical software project factors, in: Cogn Tech Work, 2023, 25, pp. 305–324.
44. M. Sadiq, T. Hassan, S. Nazneen, AHP_GORE_PSR: Applying analytic hierarchy process in goal oriented requirements elicitation method for the prioritization of software requirements, in: Proceedings of the 2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT), 2017, pp. 1-5.
45. M. Kassab, N. Kilicay-Ergin, Applying analytical hierarchy process to system quality requirements prioritization, in: Innovations in Systems and Software Engineering, 2015, 11 (4), pp. 303-312.
46. O.V. Chebanyuk, O.V.Palahin, K. K. Markov. Domain engineering approach of software requirements analysis. Проблеми програмування, 2020. № 2-3, с. 164-172.
47. Ю. В. Рогущина. Засоби та методи аналізу неструктурованих даних. Проблеми програмування, 2019, № 1. С. 57-77.

48. O. Nesterenko, I. Netesin, V. Polischuk, Y Selin. Graph-based decision making for varying complexity multicriteria problems, in: Computer Science Journal of Moldova. 2022, 30(3), pp. 391- 412
49. «Пріоритизація функцій програмного продукту: як обрати найважливіше» [Електронний ресурс] – Режим доступу: <https://wezom.com.ua/ua/blog/prioritizatsiya-funktsiy-programnogo-produktu-yak-obrati-nayvazhlivishe>
50. «Як ставити задачам пріоритети, коли важливо все», [Електронний ресурс] – Режим доступу: <https://worksection.com/ua/blog/things-to-prioritize-work.html>
51. «Управління вимогами: Пріоритезація вимог», [Електронний ресурс] – Режим доступу: <https://ua.linkedin.com/pulse/requirements-management-suggest-prioritization-ireb-gmbh-el1u7f?tl=uk>
52. «Від теорії до практики: бізнес-аналітик та продакт-овнер на одному проєкті», [Електронний ресурс] – Режим доступу: <https://e5.ua/uk/blogpost-2/vid-teoriyi-do-praktyky-biznes-analytyk-ta-prodakt-ovner-na-odnomu-proyekti/>
53. «Управління вимогами та їх пріоритезація в продакт менеджменті», [Електронний ресурс] – Режим доступу: <https://www.orbita.school/post/upravlinnya-vimogami-ta-yih-prioritezaciya-v-prodakt-menedzhmenti>
54. «Техніки бізнес-аналізу – Пріоритезація», [Електронний ресурс] – Режим доступу: <https://www.ba.in.ua/2023/12/08/tehniky-biznes-analizu-priorytezacziya/>
55. «Kano Analysis», [Електронний ресурс] – Режим доступу: <https://www.maxzosim.com/kano/>
56. «MoSCoW method», [Електронний ресурс] – Режим доступу: <https://www.maxzosim.com/moscow/>

57. «Цінність vs Складність», [Електронний ресурс] – Режим доступу: <https://www.maxzosim.com/value-vs-complexity/>
58. «Модель визначення пріоритетів – RICE Scoring Model», [Електронний ресурс] – Режим доступу: <https://www.maxzosim.com/rice-scoring-model/>
59. «Що таке User Story і як її писати», [Електронний ресурс] – Режим доступу: <https://training.qatestlab.com/blog/technical-articles/user-story/>
60. «User Story – що це, для чого і чи можна обійтися без них», [Електронний ресурс] – Режим доступу: <https://brainrain.com.ua/uk/user-story/>
61. «User Story та Acceptance Criteria: пишемо чіткі та зрозумілі вимоги», [Електронний ресурс] – Режим доступу: <https://dou.ua/lenta/articles/clear-user-stories/>
62. «User Stories», [Електронний ресурс] – Режим доступу: <https://www.mountangoatsoftware.com/agile/user-stories>
63. «User Story d IT-проектах: Як писати вимоги з точки зору користувача», [Електронний ресурс] – Режим доступу: <https://flexi-project.com/uk/user-story>
64. «Що таке acceptance criteria та які бувають помилки при критеріях прийомки», [Електронний ресурс] – Режим доступу: <https://iampm.club/ua/blog/shho-take-acceptance-criteria-ta-yaki-buvayut-pomilki-pri-kriteriyah-prijomki/>
65. «Everything you need to know about acceptance criteria», [Електронний ресурс] – Режим доступу: <https://webbylab.com/uk/news/acceptance-criteria/>
66. «What is acceptance criteria? Examples and best practices», <https://www.atlassian.com/work-management/project-management/acceptance-criteria>

67. «Уточнення беклогу продукту: 14 основних принципів», [Електронний ресурс] – Режим доступу: <https://brainrain.com.ua/uk/product-backlog-refinement/>
68. «Беклог спринту», [Електронний ресурс] – Режим доступу: <https://brainrain.com.ua/uk>
69. «Story Mapping That Works: How Objectives Drive Better Requirements», [Електронний ресурс] – Режим доступу: <https://www.iiba.org/iiba-business-analysis-member-articles/story-mapping-that-works-how-objectives-drive-better-requirements>
70. «Story Mapping: малювання загальної картини User Stories вашого продукту», [Електронний ресурс] – Режим доступу: <https://www.scrum.ua/post/story-mapping-malyuvannya-zagalnoyi-kartini-user-stories-vashogo-produktu>
71. «User Story Mapping: від ідеї до релізу», [Електронний ресурс] – Режим доступу: <https://iampm.club.ua/blog/user-story-mapping-vid-ideyi-do-relizu/>
72. «Прототипування в створенні продукту», [Електронний ресурс] – Режим доступу: <https://ux.pub/qualitygeekllc/prototipuvannia-v-stvorienni-produktu-4bfg>
73. «5 ситуацій, коли вам потрібний прототип продукту», [Електронний ресурс] – Режим доступу: <https://founder.ua/publication/5-situations-when-you-need-a-product-prototype>
74. «Прототипування як важливий етап розробки мобільних застосунків», [Електронний ресурс] – Режим доступу: <https://freelancehunt.com/blog/prototipuvannia-iak-vazhlivii-etap-rozrobki-mobilnikh-zastosunkiv/>
75. «Модель прототипу в програмній інженерії», [Електронний ресурс] – Режим доступу: <https://www.guru99.com/uk/software-engineering-prototyping-model.html>

76. «11 інструментів для створення мокапів та прототипів ваших сайтів та веб-додатків», [Електронний ресурс] – Режим доступу: <https://hostenko.com/uk/wpcafe>
77. «Найкращі інструменти для швидкого створення прототипів для вашого наступного проекту», [Електронний ресурс] – Режим доступу: <https://senyorapid.com/uk/the-best-rapid-prototype-tooling/4/>
78. «Прототипування. Огляд основних програм», [Електронний ресурс] – Режим доступу: <https://medmarketing.ua/uk/blog/prototipirovanie-obzor-osnovnyh-programm/>
79. Боднар О.А., Тімчинський О.О., Тихевич А.С., Паславський В.Г., Тесленко П.О. Управління вимогами складного ІТ-проєкту. Таврійський науковий вісник. Серія Технічні науки. Вип.1(5). 2025. – с. 46-55
80. A. Alhumam, S. Ahmed. Software requirement engineering over the federated environment in distributed software development process. Journal of King Saud University - Computer and Information Sciences, Volume 36, Issue 9. 2024.
81. Hernández, R., Moros, B. & Nicolás, J. Requirements management in DevOps environments: a multivocal mapping study. Requirements Eng 28. 2023. – pp.317-346
82. Umar M.A., Lano K. Advances in automated support for requirements engineering: a systematic literature review. Requirements Eng 29. 2024. – pp.177-207
83. Тесленко П., Паславський В., Скопін Р. Інтелектуальна система управління вимогами ІТ-проєкту. Матеріали VI МНПК «Сучасні тенденції розвитку інформаційної економіки в Україні», 2024. – с.460-462.
84. Dalpiaz F., Ferrari A., Franch X., Palomares C. Natural Language Processing for Requirements Engineering: The Best Is Yet to Come. IEEE Software. Vol. 35, No. 5. 2018. – pp.115-119.

85. Yang M., Ban A. Automated UML Class Diagram Generation from Textual Requirements Using NLP Techniques. JOIV International Journal on Informatics Visualization. Vol. 8, No. 3-2. 2024. –pp. 1905-1915.
86. Ražinskas M., Miliūnas B., Jurgelaitis M., Čeponienė L., Bisikirskienė L. Transforming Sketches of UML Use Case Diagrams to Models. IEEE Access. Vol.12. 2024. – pp.185826-185837.
87. «Побудова матриці відповідності вимог», [Електронний ресурс] – Режим доступу: <https://training.qatestlab.com/blog/technical-articles/construction-matrix-compliance-requirements/>
88. «Все про матрицю відстеження вимог: Як її створити та використовувати», [Електронний ресурс] – Режим доступу: <https://cloudfresh.com/ua/cloud-blog/vse-pro-matrytsyu-vidstezhennya-vymoh/>
89. «Що таке матриця відстеження вимог (RTM)?», [Електронний ресурс] – Режим доступу: <https://visuresolutions.com/uk/alm-guide/>
90. «Як створити план управління змінами», [Електронний ресурс] – Режим доступу: <https://stfalcon.com/uk/blog/post/how-to-create-a-change-management-plan>
91. Ліщина Н., Бойко Л., Гульчук Ю. (2024). Оцінка ризиків та їх вплив на життєвий цикл розробки програмного забезпечення. Herald of Khmelnytskyi National University. Technical Sciences, 343(6(1). 2024. - с.141-145
92. Ліщина Н. М., Ліщина В.О., Матвіїв Ю.Я., Ящук А.А. Аналіз методів визначення ризиків та управління ними при розробці програмного забезпечення. Комп'ютерно-інтегровані технології: освіта, наука, виробництво. No 41. 2020.– с.65-70.
93. Прохорова В. Методи оцінки проєктних ризиків ІТ-компаній. Адаптивне управління: теорія і практика. Серія Економіка. Т.16. No.32. 2023. – с.4-13.

94. Трофименко О. Г., Логінова Н.І., Тесленко П.О., Савельєва О.С., Поляков В.М. Класифікація ризиків у проектах із розробки програмного забезпечення. Вісник Херсонського національного технічного університету. No 3. 2023. – с.119-128
95. «Що таке BPMN-діаграма і навіщо вона потрібна з прикладами», [Електронний ресурс] – Режим доступу: <https://iampm.club/ua/blog/shho-take-bpmn-diagrama-i-navishho-vona-potribna-z-prikladami-2/>
96. «7 правил, щоб створювати зрозумілі BPMN-моделі», [Електронний ресурс] – Режим доступу: <https://laba.ua/blog/4828-7-pravil-shchob-stvoryuvati-zrozumili-bpmn-modeli>
97. «UML – панацея, отрута чи альтернатива BPMN?», [Електронний ресурс] – Режим доступу: <https://e5.ua/uk/blogpost-2/uml-panatseya-otruta-chi-alternativa-bpmn/>
98. «Різниця між UML та BPMN: коротке порівняння», [Електронний ресурс] – Режим доступу: <https://under.liberty.cx.ua/psikhologiya/riznicya-mizh-uml-ta-bpmn-kоротке-porivnyannya.html>
99. Paanichella S., Ruiz M. Requirements-Collector: Automating Requirements Specification from Elicitation Sessions and User Feedback. IEEE 28th International Requirements Engineering Conference (RE). 2020. - pp.404-407
100. Djeddi C., Zarour N., Charrel P.-J. A Novel Method of Requirements Analysis for Big Data Projects. 4th International Conference on Pattern Analysis and Intelligent Systems (PAIS). 2022. - pp.1-7
101. Dietsch D., Langenfeld V., Westphal B. Formal Requirements in an Informal World. IEEE Workshop on Formal Requirements (FORMREQ). 2020. - pp.14-20
102. Meincke W. Requirements in the loop: A computer-aided analysis of consistency, completeness, and correctness of requirements. IEEE 28th International Requirements Engineering Conference (RE). 2020. - pp.396-399

103. Roy M., Deb N., Cortesi A., Chaki R., Chaki N. CARO: A Conflict-Aware Requirement Ordering Tool for DevOps. IEEE 29th International Requirements Engineering Conference (RE). 2021. - pp. 442-443
104. Kaur J., Kaur N. Identifying Key Factors for Effective User Requirement Elicitation in Agile and Traditional Software Development: An Empirical Study. International Conference on Innovations in Intelligent Systems: Advancements in Computing, Communication, and Cybersecurity (ISAC3). 2025. - pp.1-8
105. Panis M. An Analysis of Requirements-Related Problems that Occurred in an organization Using a Mature Requirements Engineering Process. IEEE 28th International Requirements Engineering Conference (RE). 2020. - pp. 291-299
106. Batool R., Naseer A. Functional and Non-functional Requirements Classification: A Comparative Evaluation of Pre-trained LLMs and ML Techniques. International Conference on Communication Technologies (ComTech). 2025. - pp.1-6
107. Zhenkai Y., Xin W., Hao G., Yanhong D. An Analysis Method of System Safety Requirements Based on Usage Scenario. Global Reliability and Prognostics and Health Management Conference (PHM-Hangzhou). 2023. - pp.1-8