МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет
імені Івана Пулюя

Кафедра програмної інженерії

Конспект лекцій

з дисципліни

«Основи програмної інженерії (SE201)»

СЕН ID: **6822**

для здобувачів першого (бакалаврського) рівня вищої освіти

ОПП «Інженерія програмного забезпечення»
всіх форм навчання

Тернопіль — 2025

УДК 004.4(075.8)
ББК 32.973.202я73
Б83

Укладач:
В.М. Бревус, PhD
Г.Б. Бревус

Рецензент:
В.В. Яцишин, к.т.н., доцент
завідувач кафедри систем штучного інтелекту та аналізу даних ТНТУ ім. І. Пулюя

Розглянуто й затверджено на засіданні кафедри програмної інженерії.
Протокол № 1 від 01.09.2025.

Розглянуто й затверджено на засіданні методичної комісії факультету комп'ютерно-інформаційних систем та програмної інженерії Тернопільського національного технічного університету імені Івана Пулюя.
Протокол № 1 від 01.09.2025.

Б83 Конспект лекцій з дисципліни «Основи програмної інженерії (SE201)» для здобувачів першого (бакалаврського) рівня вищої освіти ОПП «Інженерія програмного забезпечення» всіх форм навчання / Укладач: Бревус В.М., Бревус Г.Б. – Тернопіль: ТНТУ імені Івана Пулюя, 2025 – 127 с.

Відповідальний за випуск: М.Р. Петрик, докт. фіз.-мат. наук, професор

АНОТАЦІЯ

У конспекті лекцій систематизовано фундаментальні поняття, принципи та методи програмної інженерії. Розглянуто життєвий цикл програмного забезпечення (SDLC), моделі процесів розробки (Waterfall, Agile, Scrum, Kanban), інженерію вимог, архітектуру та проєктування ПЗ. Висвітлено питання конструювання, забезпечення якості, тестування, управління конфігурацією та супроводу програмних систем. Окрему увагу приділено управлінню проєктами, професійній етиці та стандартам (SWEBOK, ISO/IEC), а також сучасним тенденціям, таким як DevOps, хмарні технології та використання штучного інтелекту в розробці. Матеріал орієнтований на здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 121 «Інженерія програмного забезпечення» та спрямований на формування професійних компетентностей у сфері промислової розробки програмних продуктів.

Ключові слова: програмна інженерія, SDLC, Agile, вимоги, архітектура ПЗ, тестування, якість ПЗ, управління проєктами, DevOps, SWEBOK.

ABSTRACT

The lecture notes systematize fundamental concepts, principles, and methods of software engineering. The software development life cycle (SDLC), development process models (Waterfall, Agile, Scrum, Kanban), requirements engineering, software architecture and design are considered. Issues of construction, quality assurance, testing, configuration management, and software maintenance are highlighted. Special attention is paid to project management, professional ethics and standards (SWEBOK, ISO/IEC), as well as modern trends such as DevOps, cloud technologies, and the use of artificial intelligence in development. The material is aimed at applicants for the first (bachelor's) level of higher education in the specialty 121 "Software Engineering" and is directed at forming professional competencies in the field of industrial software product development.

Keywords: software engineering, SDLC, Agile, requirements, software architecture, testing, software quality, project management, DevOps, SWEBOK.

Зміст

Вступ

Конспект лекцій з дисципліни «Основи програмної інженерії» (Software Engineering Fundamentals) розроблено для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності F2 «Інженерія програмного забезпечення».

Метою курсу є формування у майбутніх фахівців систематизованих знань про програмну інженерію як інженерну дисципліну, що займається всіма аспектами виробництва програмного забезпечення: від початкових стадій створення специфікації системи до підтримки системи після здачі в експлуатацію.

Сучасна розробка програмного забезпечення вимагає від фахівців не лише навичок написання коду, але й глибокого розуміння процесів життєвого циклу, вміння працювати в команді, управляти вимогами, проєктувати архітектуру, забезпечувати якість продукту та дотримуватися професійних стандартів.

У цьому курсі розглядаються фундаментальні поняття та принципи програмної інженерії, що базуються на міжнародних стандартах (SWEBOK, ISO/IEC/IEEE) та кращих світових практиках.

Основні теми, що висвітлюються у конспекті:

- **Вступ до програмної інженерії:** визначення, історія, відмінність від комп'ютерних наук, професійна етика.
- **Життєвий цикл програмного забезпечення (SDLC):** моделі процесів розробки (Waterfall, V-Model, Spiral) та гнучкі методології (Agile, Scrum, Kanban).
- **Інженерія вимог:** процеси виявлення, аналізу, специфікації та валідації вимог до ПЗ.
- **Проєктування та архітектура:** архітектурні стилі, патерни проєктування, принципи SOLID, моделювання за допомогою UML.

- **Конструювання ПЗ:** інструментальні засоби, стандарти кодування, рефакторинг, контроль версій.
- **Забезпечення якості та тестування:** рівні та типи тестування, верифікація та валідація.
- **Управління проєктами:** планування, оцінка ризиків, командна робота.
- **Сучасні тенденції:** DevOps, хмарні технології, використання штучного інтелекту в розробці.

Матеріал структуровано відповідно до робочої програми дисципліни та охоплює ключові області знань SWEBOK (Software Engineering Body of Knowledge). Кожна лекція містить теоретичні відомості, приклади з реальної практики та контрольні запитання для самоперевірки.

Вивчення дисципліни дозволить студентам зрозуміти різницю між аматорським програмуванням та професійною інженерією, а також підготує фундамент для подальшого поглибленого вивчення спеціалізованих дисциплін та успішної кар'єри в IT-індустрії.

Лекція 1

Вступ до програмної інженерії

Визначення програмної інженерії, стандарти, історія, сучасні тенденції.

Організаційні питання

Про курс

Ресурси:

Структура курсу: 9 лекцій, лабораторні завдання, підсумковий проект.

Результати навчання: Ознайомлення здобувачів з загальними принципами життєвого циклу програмного забезпечення, методологіями, ключовими процесами та артефактами.

Предмет, мета та завдання курсу

Предмет: Теоретичні та практичні аспекти інженерії програмного забезпечення, методи, засоби та процеси розробки ПЗ.

Мета: Формування у здобувачів систематизованих знань про програмну інженерію як інженерну дисципліну, ознайомлення з життєвим циклом ПЗ, методологіями розробки та управлінням якістю.

Завдання:

- Зрозуміти відмінність між написанням коду та створенням програмного продукту.
- Вивчити основні етапи життєвого циклу ПЗ (SDLC).
- Ознайомитися з ролями в команді розробки.
- Опанувати базові принципи роботи з вимогами, архітектурою та тестуванням.

Навчальні результати

Після завершення лекції студент зможе:

1. Надати визначення програмної інженерії згідно зі стандартом IEEE.
2. Розрізнити різниці між Computer Science та Software Engineering.
3. Пояснити роль стандартів (SWEBOK, ISO/IEC/IEEE 12207, ISO/IEC 25010) у розробці ПЗ.
4. Описати історію розвитку програмної інженерії та криз ПЗ.
5. Класифікувати методології розробки та вибрати відповідну для сценарію.
6. Обґрунтувати вплив сучасних тенденцій (хмарні технології, DevOps, AI) на розробку ПЗ.

Що таке програмна інженерія?

Визначення

Програмна інженерія (Software Engineering) — це застосування систематичного, дисциплінованого, вимірюваного підходу до розробки, експлуатації та супроводу програмного забезпечення (IEEE Standard Glossary of Software Engineering Terminology) [1], [2], [3].

Це інженерна дисципліна, яка займається *всіма* аспектами виробництва ПЗ: від початкових стадій створення специфікації системи до підтримки системи після здачі в експлуатацію.

Програмна інженерія vs Комп'ютерні науки

Основні відмінності:

Комп'ютерні науки (CS) Фокусуються на теорії та основах обчислень (алгоритми, структури даних, теорія складності, AI). Це наука про те, як працюють комп'ютери та програми. Часто орієнтована на теоретичні дослідження.

Програмна інженерія (SE) Фокусується на практичних проблемах створення якісного ПЗ (вимоги, архітектура, тестування, процеси розробки). Це дисципліна про те, як створювати якісні програмні продукти в умовах обмежених ресурсів (час, бюджет).

> *Вчені будують, щоб вчитися; інженери вчаться, щоб будувати.*
> (Fred Brooks [4])

Життєвий цикл розробки програмного забезпечення (SDLC)

Визначення та мета

Життєвий цикл розробки програмного забезпечення (Software Development Life Cycle, SDLC) — це систематичний процес розробки високоякісного програмного забезпечення у передбачувані терміни та в рамках бюджету.

Мета SDLC — створення програмного забезпечення, яке відповідає бізнес-вимогам клієнта.

SDLC визначає фази процесу розробки, кожна з яких має свої процеси та результати (deliverables). Це цикл планування, проектування та розробки, який може бути реалізований як ітеративний підхід. Дотримання SDLC мінімізує ризики та витрати на розробку якісного ПЗ.

Історія виникнення

SDLC почав формуватися в середині 1960-х років, коли розробка ПЗ почала вимагати більш детального підходу через зростаючу складність систем. Великі

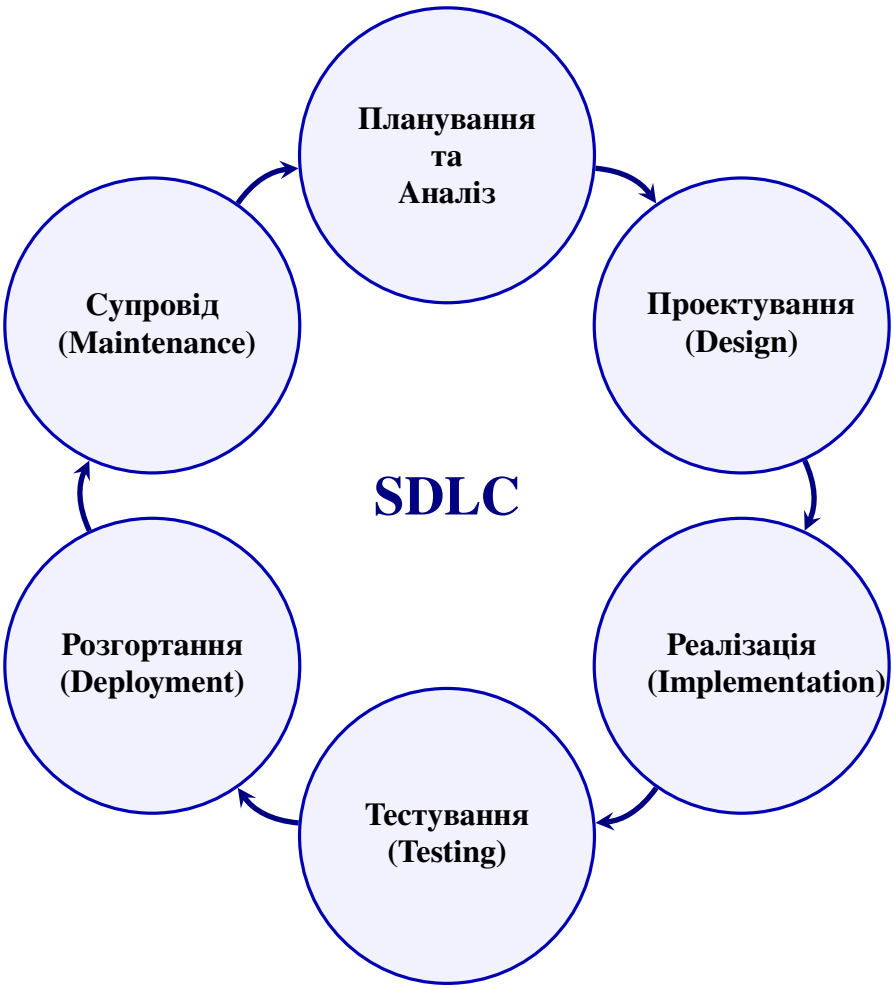

Рис. 1.1: Циклічна модель життєвого циклу розробки ПЗ

корпорації потребували управління складними бізнес-системами, що вимагали значних обчислювальних ресурсів.

Спочатку використовувався так званий **«каскадний метод» (Waterfall)**, де розробка слідувала лінійній схемі через дискретні етапи. Згодом SDLC був адаптований до більш **ітеративних методів** для кращого реагування на потреби клієнтів та змінні вимоги.

Ключові переваги SDLC

Використання SDLC надає бізнесу та командам розробки ряд переваг:

- **Систематичний процес:** Замість хаотичного (ad hoc) підходу, команди мають чіткий план, що підвищує ефективність та знижує ризики.
- **Чіткі фази:** Кожен етап добре визначений, тому учасники знають, над

чим і коли працювати.

- **Покращена комунікація:** Чіткі фази полегшують спілкування між замовником, стейкхолдерами та командою розробки. Стейкхолдери розуміють своє місце в процесі.
- **Чіткість завершення завдань:** Міжфункціональні команди знають, коли їхні завдання виконані і можна переходити до наступного етапу.
- **Можливість ітерацій:** Процес дозволяє повертатися назад для врахування нових вимог.
- **Раннє вирішення проблем:** Проблеми виявляються та вирішуються на етапі проектування, а не під час кодування, що значно дешевше.
- **Чіткі ролі:** Кожен член команди має визначену роль, що зменшує конфлікти та дублювання обов'язків.

Стандартизація та професійні знання

Програмна інженерія як зріла дисципліна спирається на систематизоване тіло знань та міжнародні стандарти.

SWEBOK (Software Engineering Body of Knowledge)

SWEBOK Guide v4 (ISO/IEC TR 19759) [5] — це документ, розроблений IEEE Computer Society, який консолідує загальноприйняті знання в галузі.

- Він структурує дисципліну на **18 розділів (Chapters)**:
  - **Chapter 1: Software Requirements** — інженерія вимог
  - **Chapter 2: Software Architecture** — архітектура ПЗ
  - **Chapter 3: Software Design** — проектування
  - **Chapter 4: Software Construction** — конструювання
  - **Chapter 5: Software Testing** — тестування
  - **Chapter 6: Software Engineering Operations** — операції

- **Chapter 7: Software Maintenance** — супровід
- **Chapter 8: Software Configuration Management** — управління конфігурацією
- **Chapter 9: Software Engineering Management** — управління проектами
- **Chapter 10: Software Engineering Process** — процеси розробки
- **Chapter 11: Software Engineering Models and Methods** — моделі та методи
- **Chapter 12: Software Quality** — якість ПЗ
- **Chapter 13: Software Security** — безпека
- **Chapter 14: Software Engineering Professional Practice** — професійна практика
- **Chapter 15: Software Engineering Economics** — економіка ПЗ
- **Chapter 16: Computing Foundations** — основи обчислень
- **Chapter 17: Mathematical Foundations** — математичні основи
- **Chapter 18: Engineering Foundations** — інженерні основи

- Використовується як основа для акредитації університетських програм та професійної сертифікації.

Ключові стандарти

ISO/IEC/IEEE 12207 (Software Life Cycle Processes) Фундаментальний стандарт, що описує повний набір процесів життєвого циклу ПЗ:

- Замовлення (Acquisition)
- Постачання (Supply)
- Розробка (Development)
- Експлуатація (Operation)
- Супровід (Maintenance)

ISO/IEC 25010 (System and Software Quality Models) Визначає модель якості продукту, що включає 8 характеристик:

- Функціональна придатність
- Ефективність продуктивності
- Сумісність
- Зручність використання
- Надійність
- Безпека інформації
- Супроводжуваність
- Переносимість

IEEE Code of Ethics Кодекс професійної етики інженера-програміста [6], що наголошує на відповідальності перед суспільством.

Історичний огляд та криза програмного забезпечення

Криза програмного забезпечення (Software Crisis)

Термін виник наприкінці 1960-х років (конференція NATO, 1968).

Основні проблеми:

- Проєкти перевищували бюджет та терміни виконання.
- ПЗ було ненадійним, складним у супроводі та модифікації.
- Невідповідність вимогам замовника.
- Складність управління великими командами розробників.

Це призвело до розуміння необхідності інженерного підходу до розробки ПЗ, а не просто “написання коду”.

Еволюція методологій розробки ПЗ

1970-ті роки: Структурне програмування Боротьба зі “спагеті-кодом”, введення функцій та модулів. Акцент на лінійній структурі програм.

1980–1990-ті роки: Об’єктно-орієнтоване програмування Введення класів, інкапсуляції, спадкування, поліморфізму для управління складністю. Модельні підходи (UML).

2000-ні роки: Agile та гнучкі методології Реакція на громіздкі “важковагові” процеси (Waterfall). Маніфест Agile (2001) [7]: людиноцентричний підхід, короткі ітерації, адаптивність до змін.

2010-ні рр. і досі: Гібридні та сучасні методології Комбінація Agile, DevOps, хмарних технологій, автоматизація тестування та розгортання.

Сучасні тенденції в програмній інженерії

Хмарні технології (Cloud Computing)

Перехід від локальних серверів до хмарних платформ (AWS, Azure, Google Cloud) змінив архітектуру та способи розгортання ПЗ.

Вирішені проблеми:

- **Масштабованість:** Автоматичне збільшення ресурсів під час пікових навантажень (Auto-scaling).
- **Капітальні витрати:** Заміна великих початкових інвестицій в “залізо” на операційні витрати (Pay-as-you-go модель).

Відкриті виклики:

- **Vendor Lock-in:** Складність міграції між хмарними провайдерами через специфічні API та сервіси.
- **Конфіденційність даних:** Питання суверенітету даних та відповідності регуляціям (GDPR, CCPA).
- **Безпека:** Управління доступом та захист від хмарних атак.

DevOps (Development + Operations)

Культура та набір практик, що об'єднують розробку (Dev) та експлуатацію (Ops) для скорочення циклу розробки та забезпечення високої якості.

Ключові компоненти:

- **CI/CD (Continuous Integration / Continuous Delivery):** Автоматизація збірки, тестування та розгортання.
- **Infrastructure as Code (IaC):** Опис інфраструктури через код (Terraform, Ansible).
- **Контейнеризація:** Упакування додатків з залежностями (Docker, Kubernetes).
- **Моніторинг та логування:** Постійне спостереження за системою в продакшені.

Вирішені проблеми:

- **"It works on my machine":** Використання контейнеризації забезпечує однакове середовище на всіх етапах.
- **Повільні релізи:** Автоматизація дозволяє робити деплоймент кілька разів на день, а не раз на місяць.
- **Розділення обов'язків:** Об'єднання розробників та операціоністів в одну команду.

Відкриті виклики:

- **Складність інструментарію:** Необхідність знання десятків інструментів (Kubernetes, Jenkins, Terraform, Prometheus).
- **Безпека в швидкому темпі:** Інтеграція безпеки в швидкий процес розробки (DevSecOps) часто відстає від швидкості змін.

Штучний інтелект в розробці (AI in Software Engineering)

Використання AI-асистентів (GitHub Copilot [8], ChatGPT, Claude) для генерації коду, написання тестів, пошуку помилок та рефакторингу змінює роль розробника.

Вирішені проблеми:

- **Рутинний код (Boilerplate):** Генерація типових конструкцій, класів, тестів, що економить час.
- **Пошук інформації:** Швидке отримання пояснень щодо бібліотек, помилок без довгого пошуку.
- **Поточне документування:** AI може генерувати коментарі та документацію.

Відкриті виклики та ризики:

- **Галюцинації:** Генерація синтаксично правильного, але логічно хибного або вразливого коду.
- **Юридичні питання:** Проблеми авторського права на згенерований код та ліцензійна чистота.
- **Безпека та конфіденційність:** Передання коду стороннім сервісам; витік комерційної інформації.
- **Залежність від AI:** Можливість атрофії навичок розробників, надмірна довіра до AI.

Приклади інженерної досконалості

Ядро Linux (The Linux Kernel)

Один з найбільших та найскладніших проєктів з відкритим кодом у світі.

Масштаб та характеристики:

- Понад 30 мільйонів рядків коду.

- Тисячи контриб'юторів з різних організацій та країн.
- Підтримка тисяч апаратних платформ (від вбудованих пристроїв до суперкомп'ютерів).

Інженерні практики:

- **Суворий процес Code Review:** Кожна зміна повинна пройти рецензування досвідченими розробниками.
- **Розподілена система контролю версій (Git):** Git [9] був створений саме для управління Linux Kernel.
- **Модульна архітектура:** Розділення на чітко визначені модулі для незалежної розробки та тестування.
- **Зворотна сумісність:** Забезпечення сумісності з попередніми версіями та старим апаратним забезпеченням.
- **Автоматизовані тести та CI/CD:** Постійна перевірка коду перед інтеграцією.

Ключовий урок: Навіть складні системи з великою кількістю контриб'юторів можуть бути успішними, якщо дотримуватися чітких процесів та архітектурних принципів.

Програмне забезпечення марсоходів NASA (Curiosity/Perseverance)

Приклад критично важливих систем (Safety-Critical Systems), де ціна помилки — провал місії вартістю мільярди доларів.

Особливості та виклики:

- Марсохід розташований за мільйони км, комунікація займає десятки хвилин.
- Неможливо оновити або виправити код після запуску (на відміну від наземного ПЗ).
- Вбудовані системи з обмеженою пам'яттю та енергією.

- Різні умови виживання: радіація, екстремальні температури, пил.

Інженерні практики:

- **JPL Coding Standards:** Суворі стандарти кодування для мінімізації ризиків (подібні принципи описані в [10]).
  - Обов'язкова типізація.
  - Забарона динамічного виділення пам'яті (malloc заборонено).
  - Максимальне обмеження функцій та циклічної складності.
- **Екстремальне тестування:** Тестування на різних сценаріях, включаючи збої та відмови.
- **Формальна верифікація:** Математичне доведення коректності критичних алгоритмів.
- **Надмірність та дублювання:** Апаратне дублювання основних систем та дублювання програмного забезпечення.
- **Висока покритість тестами:** Понад 90% коду покрито тестами перед розгортанням.

Результати: Марсоходи Curiosity та Perseverance працюють набагато довше очікуваного терміну (більше 10 років для Curiosity замість планових 2 років), що демонструє якість розробки.

Ключовий урок: В критично важливих системах якість коду та надійність не є ненужною розкішшю — це матерія життя або смерті (або в цьому випадку успіху чи провалу дорогої місії).

Висновки та рекомендації

- Програмна інженерія — це більше, ніж просто програмування. Це управління складністю, якістю та процесами.
- Галузь постійно розвивається, реагуючи на нові виклики: від кризи ПЗ 1960-х років до ери AI в 2020-х.

- Розуміння основ (SDLC, вимоги, архітектура, тестування) є фундаментом для успішної кар'єри в IT.
- Сучасні тенденції (хмарні технології, DevOps, AI) змінюють роль розробника від "написання коду" до "проектування рішень та верифікації коректності".
- Вчимося зі прикладів успішних проектів (Linux, NASA) та розуміємо інженерні принципи, що лежать в їхній основі.

Контрольні запитання

1. Дайте визначення програмної інженерії згідно зі стандартом IEEE.
2. У чому полягає головна відмінність між Computer Science та Software Engineering?
3. Що таке SWEBOK і скільки областей знань він охоплює?
4. Які основні процеси життєвого циклу описує стандарт ISO/IEC/IEEE 12207?
5. Перелічіть основні характеристики якості ПЗ згідно з ISO/IEC 25010.
6. Чому етика є важливою складовою професії інженера-програміста?
7. Що таке "криза програмного забезпечення" (Software Crisis) і коли виник цей термін?
8. Які основні етапи еволюції методологій розробки (від 1970-х до сьогодення)?
9. Як структурне програмування вирішувало проблеми ранньої розробки ПЗ?
10. У чому суть Agile Manifesto та гнучких методологій?
11. Як хмарні технології змінили підхід до капітальних витрат (CapEx vs OpEx)?

12. Що таке масштабованість (Auto-scaling) у контексті хмарних обчислень?
13. Поясніть поняття Vendor Lock-in.
14. Що таке DevOps і які дві сфери він об'єднує?
15. Яку проблему вирішує практика CI/CD?
16. Як контейнеризація допомагає вирішити проблему "It works on my machine"?
17. Як штучний інтелект змінює роль розробника ПЗ?
18. Що таке "галюцинації" AI і чому вони небезпечні при генерації коду?
19. Які специфічні інженерні практики використовуються в проєкті ядра Linux?
20. Чому в критично важливих системах (наприклад, NASA) існують суворі обмеження стандартів кодування?

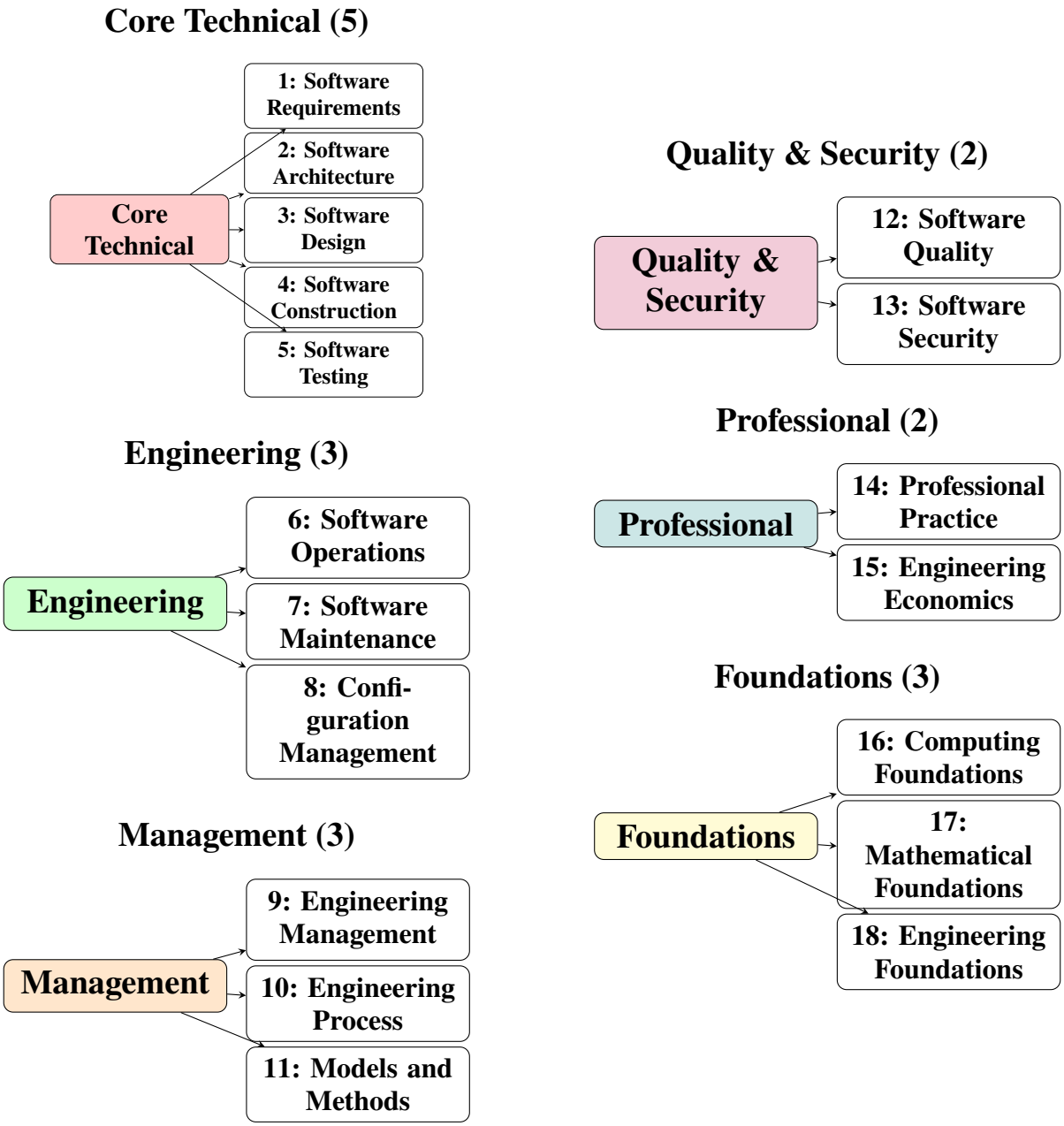

Рис. 1.2: SWEBOK v4: 18 розділів у 6 категоріях (2-колонковий макет)

Лекція 2

Процеси та моделі життєвого циклу ПЗ

SDLC моделі: Waterfall, V-Model, спіральна модель та Agile підходи (Scrum, Kanban).

Предмет, мета та завдання лекції

Предмет: Моделі та процеси життєвого циклу програмного забезпечення (SDLC — Software Development Life Cycle), методології та практики управління розробкою.

Мета: Сформувати розуміння різноманітності підходів до організації розробки ПЗ, навчитися аналізувати переваги та недоліки різних моделей та обирати оптимальну для конкретного проєкту, зважаючи на контекст, обмеження та ризики.

Завдання:

- Визначити поняття SDLC та його основні фази з точки зору SWEBOK.
- Розглянути традиційні (передбачувальні) моделі: Waterfall та V-Model.
- Вивчити еволюційні та ітеративні підходи: спіральну модель та DevOps.
- Опанувати цінності та принципи гнучкої розробки (Agile).
- Детально розібрати популярні фреймворки: Scrum та Kanban.
- Розуміти місце та роль управління вимогами, конфігурацією та змінами.

Навчальні результати

Після завершення лекції студент зможе:

1. Пояснити основні фази SDLC та їхні цілі.
2. Порівняти переваги та недоліки традиційних моделей (Waterfall та V-Model).
3. Описати спіральну модель та її застосування для управління ризиками.
4. Інтерпретувати цінності Agile Manifesto та їхній вплив на розробку.
5. Пояснити структуру, артефакти та події Scrum.
6. Описати принципи та практики Kanban.
7. Вибрати оптимальну методологію для конкретного контексту проєкту.

Життєвий цикл програмного забезпечення (SDLC)

Визначення та основні фази

Життєвий цикл програмного забезпечення (Software Development Life Cycle, SDLC) — це структура, що описує етапи, через які проходить програмний продукт від моменту виникнення ідеї до виведення з експлуатації.

За SWEBOK v4, SDLC охоплює весь спектр діяльності від планування, опису вимог, проєктування, реалізації, верифікації та валідації до розгортання та супроводу. Вибір відповідної моделі SDLC впливає на відповідність продукту вимогам, якість, витрати та терміни проєкту.

Незалежно від обраної моделі, розробка зазвичай включає наступні основні активності та практики:

1. **Аналіз вимог (Requirements Analysis):** Що ми будуємо? Збір та документування потреб замовника та стейкхолдерів.
2. **Проєктування (Design):** Як ми будемо це будувати? Визначення архітектури системи, інтерфейсів, структур даних та взаємодії компонентів.

3. **Реалізація (Implementation/Coding):** Безпосереднє написання програмного коду в обраній мові програмування.

4. **Тестування (Testing):** Перевірка відповідності вимогам, пошук дефектів та забезпечення якості продукту.

5. **Розгортання (Deployment):** Встановлення програмного забезпечення у робоче середовище та передача користувачам.

6. **Супровід (Maintenance):** Виправлення помилок, оптимізація, адаптація до змін та додавання нових функцій після релізу.

Традиційні моделі розробки

Каскадна модель (Waterfall)

Класична модель, запропонована Вінстоном Ройсом у 1970 році. Передбачає суворо послідовне виконання етапів: наступний етап починається тільки після повного завершення та затвердження результатів попереднього.

Рис. 2.1: Каскадна модель (Waterfall)

Переваги Waterfall

- **Простота та зрозумілість процесу:** Лінійна послідовність етапів легко розуміється всіма учасниками проєкту.
- **Чітка документація на кожному етапі:** Кожен етап завершується вичерпною документацією (специфікація, дизайн, тестові плани).
- **Легкість управління термінами та бюджетом:** За умови стабільних вимог можна достатньо точно спланувати терміни та витрати.
- **Відповідність регуляціям:** Для критично важливих систем часто вимагається докладна документація на кожному етапі.

Недоліки Waterfall

- **Неможливість повернутися на попередній етап без значних витрат:** Виявлена помилка у вимогах на етапі тестування потребує переробки всіх наступних етапів.
- **Робочий продукт з'являється лише в самому кінці проєкту:** Замовник бачить результат лише після завершення розробки, що ускладнює зворотний зв'язок.
- **Високий ризик:** Помилка у вимогах коштує дуже дорого для виправлення, так як вона виявляється занадто пізно.
- **Жорсткість щодо змін:** Нові вимоги, які виникають під час розробки, важко інтегрувати без перепроектування.

Застосування

Waterfall найкраще підходить для:

- Проєктів з чіткими, незмінними вимогами та низькою невизначеністю.
- Систем з низьким рівнем технічних ризиків.
- Вбудованих систем, медичного ПЗ, авіації та критично важливих систем, де документація та верифікація є критичними.

V-Model (Verification and Validation)

Розширення моделі Waterfall, де кожному етапу розробки відповідає певний етап тестування. Графічно зображується у вигляді літери V. Модель демонструє гарантії якості на кожному рівні.

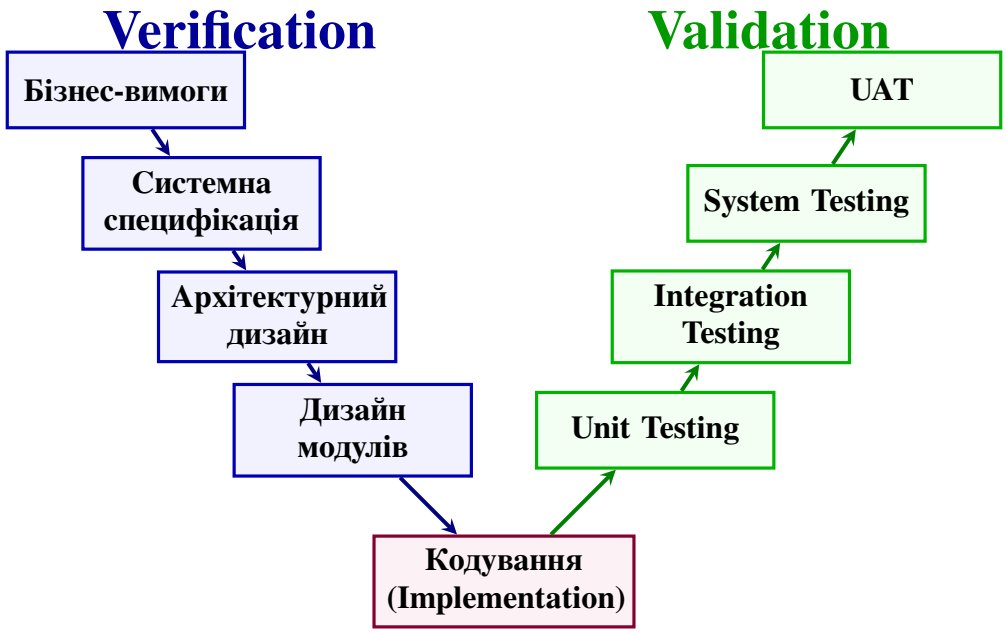

Рис. 2.2: V-Model: Verification (ліва сторона) і Validation (права сторона)

Особливості V-Model

Ліва сторона (Verification): Декомпозиція вимог та створення специфікацій (від бізнес-вимог до дизайну модулів). Фокус на «ми будуємо продукт правильно?».

Права сторона (Validation): Інтеграція та перевірка (від модульних тестів до приймального тестування). Фокус на «ми будуємо правильний продукт?».

Переваги та недоліки

Переваги:

- Сильний акцент на якість та тестування на кожному рівні.
- Чітка відповідність між специфікацією та тестами.

- Ранне планування тестування.

Недоліки:

- Залишається негнучкою стосовно змін вимог.
- Робочий продукт все ще з'являється лише в кінці.
- Більш складна у управлінні ніж проста Waterfall.

Еволюційні та ітеративні моделі

Ітеративна та інкрементальна розробка

Ітеративна розробка: Система розробляється шляхом повторюваних циклів (ітерацій). В кожній ітерації продукт аналізується, покращується та доопрацьовується. Це процес "уточнення" рішення (як малювання ескізу, а потім деталізація).

Інкрементальна розробка: Система будується частинами (інкрементами). Кожен інкремент додає нову функціональність до системи. Це процес "складання" рішення (як будівництво будинку по цеглині).

Сучасні Agile-підходи зазвичай є ітеративно-інкрементальними, поєднуючи обидві стратегії для максимальної гнучкості та цінності для клієнта.

Спіральна модель (Spiral Model)

Запропонована Баррі Бомом [11]. Головний фокус — **управління ризиками**. Процес рухається по спіралі, проходячи 4 сектори в кожному витку. Моделю було подалі розвинено та деталізовано у [12]:

Чотири сектори спіралі

1. **Визначення цілей та альтернатив:** Чіткість цілей поточної ітерації та розглядання альтернативних підходів.

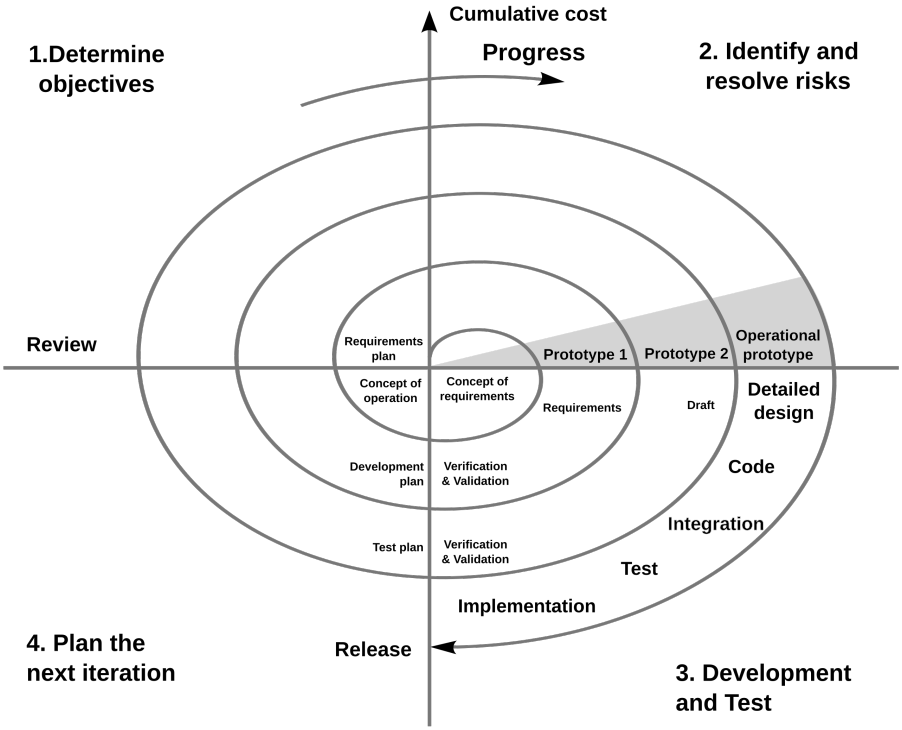

Рис. 2.3: Спіральна модель Боема: 4 сектори в кожному витку

2. **Оцінка та вирішення ризиків:** Ідентифікація потенційних ризиків та розробка стратегій їхнього пом'якшення. Часто виконується через створення прототипів.

3. **Розробка та тестування продукту:** Безпосередня розробка та перевірка вибраного підходу.

4. **Планування наступної ітерації:** Оцінка прогресу та підготовка наступного витка спіралі.

Застосування

Спіральна модель застосовується для:

- Великих, дорогих, складних проєктів з високим рівнем невизначеності та ризиків.
- Системи критичного призначення, де управління ризиками є пріоритетом.
- Проєктів, де необхідна гнучкість в поєднанні з систематичним управлінням.

Гнучкі методології (Agile)

Agile Manifesto (2001)

Реакція на бюрократичність та неповороткість традиційних методів. У лютому 2001 р. група розробників підписала Agile Manifesto, що визначив 4 ключові цінності:

1. **Люди та взаємодія** важливіші за процеси та інструменти.

   Акцент на якості комунікації та взаємодії в команді, а не просто дотримання процедур.

2. **Працюючий продукт** важливіший за вичерпну документацію.

   Пріоритет на постійну поставку робочого кода замість багатотомної документації.

3. **Співпраця з замовником** важливіша за узгодження умов контракту.

   Активна участь замовника в розробці замість простого укладання контракту та відсторонення.

4. **Готовність до змін** важливіша за дотримання плану.

   Гнучкість та адаптивність до нових вимог і умов замість сліпої відповідності первинному плану.

Принципи Agile

Крім 4 цінностей, Agile Manifesto також визначив 12 принципів, серед яких:

- Задоволення замовника через ранню й постійну поставку цінного ПЗ.
- Прийняття змінених вимог навіть на пізніх етапах розробки.
- Частої поставки робочого ПЗ (щотижнів або щомісяців, як правило).
- Щоденна співпраця між розробниками та замовниками.
- Побудова проєктів навколо мотивованих індивідуумів, наділення їх довір'ям.

- Тісна комунікація в команді (бажано в одному офісі).
- Робочий продукт як основна міра прогресу.
- Стійке розвитку темпу розробки.
- Постійна увага до технічної досконалості та якому дизайну.

Scrum

Найпопулярніший Agile-фреймворк для управління складними проєктами. Базується на емпіризмі (прозорість, інспекція, адаптація).

Ролі в Scrum

Product Owner (Власник продукту): Відповідає за максимізацію цінності продукту та управління Product Backlog. Часто це бізнес-аналітик або представник замовника. Основні обов'язки:

- Визначення та пріоритизація вимог.
- Комунікація з замовником та стейкхолдерами.
- Прийняття рішень щодо функціональності та часування релізів.

Scrum Master: Слуга-лідер, що допомагає команді зрозуміти Scrum, усуває перешкоди та фасилітує події. На відміну від традиційного Project Manager, SM не дає наказів, а фасилітує. Основні обов'язки:

- Навчання Scrum процесу.
- Видалення impediments (перешкод).
- Фасилітація подій (Planning, Daily Standup, Review, Retrospective).

Developers (Команда розробки): Крос-функціональна команда професіоналів (програмісти, QA, дизайнери), що створює готовий інкремент продукту. Розмір команди зазвичай 3–9 осіб.

Артефакти Scrum

Product Backlog: Впорядкований список всього, що може знадобитися в продукті. Постійно еволюціонує в залежності від вимог замовника та умов ринку. PO відповідає за його управління.

Sprint Backlog: План на поточний спринт, обраний на Sprint Planning. Включає вибрані користувацькі історії та технічні завдання. Видимий для всієї команди.

Increment (Product Increment): Готова, потенційно релізна частина продукту на кінці спринту. Повинна відповідати "Definition of Done" (чіткі критерії завершення).

Події (Events) у Scrum

Sprint: Контейнер для всіх подій та розробки. Фіксований період часу (1–4 тижні, традиційно 2 тижні). Кожен спринт закінчується готовим інкрементом.

Sprint Planning: Планування роботи на спринт. Команда вибирає Storia з Product Backlog, що зможе виконати за спринт. Тривалість: 2–4 години для 2-тижневого спринту.

Daily Scrum: Щоденна 15-хвилинна зустріч для синхронізації (максимум 15 хвилин, незалежно від розміру команди). Кожен розробник розповідає:

- Що зроблено вчора?
- Що планується зробити сьогодні?
- Які перешкоди?

Sprint Review (Demo): Демонстрація результатів замовникам та стейкхолдерам. Команда показує готові функції та отримує зворотний зв'язок. Тривалість: 1–2 години.

Sprint Retrospective: Аналіз процесу роботи команди та планування покращень. Обговорюються питання:

- Що пройшло добре?

- Що можна покращити?
- Які конкретні дії ми виконаємо в наступному спринті?

Тривалість: 45 хвилин – 1 година.

Цикл Scrum

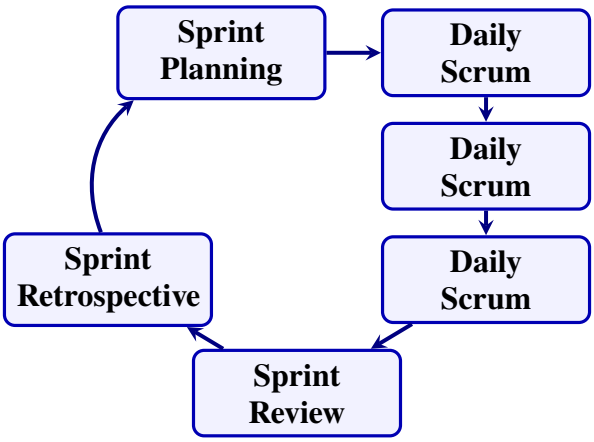

Рис. 2.4: Цикл подій у Scrum

Kanban

Метод управління потоком роботи (Flow), що походить від Toyota (Lean Manufacturing). На відміну від Scrum, Kanban не вимагає фіксованих спринтів.

Ключові принципи Kanban

1. **Візуалізуй роботу (Visualize)**: Використання Kanban-дошки для представлення всіх завдань та їхніх статусів. Типові колонки: *To Do*, *In Progress*, *Done*.

2. **Обмежуй кількість незавершеної роботи (Limit WIP)**: Встановлення максимального числа завдань на кожний етап. Це запобігає перевантаженню розробників та допомагає виявити “пляшкові горлечки”.

3. **Керуй потоком (Manage Flow)**: Фокус на швидкому проходженні задач від початку до кінця (Time-to-Completion). Аналіз середнього часу виконання завдання (Lead Time, Cycle Time).

4. **Робімо політику явною (Make Policies Explicit)**: Чіткі правила щодо переходу між етапами, критеріїв готовності.

5. **Реалізуй механізми зворотного зв'язку (Implement Feedback Loops)**: Регулярні засідання для аналізу потоку та покращення.

Kanban дошка

Типова Kanban дошка містить колонки:

To Do	In Progress	Review/QA	Done
Task 1	Task 5	Task 3	Task 2
Task 4	Task 6		Task 7
Task 8			

Табл. 2.1: Приклад Kanban дошки

WIP Limit для кожної колонки:

- *To Do*: Необмежено (усі потенційні завдання).
- *In Progress*: 3 завдання максимум.
- *Review/QA*: 2 завдання максимум.
- *Done*: Необмежено.

Scrum vs Kanban

Відмінність від Scrum

- **Немає фіксованих спринтів:** Робота безперервна, завдання йдуть через систему в їхньому власному темпі.
- **Немає суворих ролей:** На відміну від Scrum, немає обов'язкових ролей PO та SM.

Аспект	Scrum	Kanban
Ітерації	Фіксовані спринти (1–4 тижні)	Безперервний потік
Планування	Sprint Planning на початку спринту	Безперервне планування
Релізи	На кінець кожного спринту	Коли готово (постійні релізи)
Ролі	PO, SM, Team (суворі)	Більш гнучкі, можуть змінюватись
Змінюваність	Зміни в середині спринту небажані	Зміни допускаються будь-коли
Метрики	Velocity (завдань за спринт)	Lead Time, Cycle Time
Придатність	Проєкти з чіткими вимогами	Потокові роботи, підтримка ПЗ

Табл. 2.2: Порівняння Scrum та Kanban

- **Гнучкість змін:** Нові завдання можна додати в будь-який момент, якщо є вільна ємність (WIP дозволяє).
- **Метрики потоку:** Замість velocity (завдань за спринт) Kanban використовує Lead Time та Cycle Time.

Гібридні підходи

Scrumban

Комбінація Scrum та Kanban, що поєднує:

- **Фіксовані спринти** від Scrum для отримання ритму та планування.
- **WIP Limits та Kanban дошку** для управління потоком роботи.

  Застосовується коли:

- Команда потребує деякої структури (спринти), але також потребує гнучкості (Kanban).
- Потрібно обробляти як планові, так і екстрені завдання.

Управління вимогами та конфігурацією

Управління вимогами (Requirements Management)

Згідно з SWEBOK v4, управління вимогами є критичною практикою, яка включає:

Збір вимог (Elicitation): Взаємодія з зацікавленими сторонами (stakeholders) для розуміння їх потреб та очікувань.

Аналіз вимог (Analysis): Вивчення, переробка та уточнення зібраних вимог для забезпечення їх повноти, послідовності та здійсненності.

Специфікація: Документування вимог у формі, яка дозволяє розробникам розуміти та реалізовувати функціональність та якості.

Валідація вимог: Переконання, що задокументовані вимоги відповідають справжнім потребам користувачів та стейкхолдерів.

Управління змінами вимог: Контроль та відстеження змін вимог протягом проєкту.

Значення: Проблеми з вимогами є однією з основних причин невдач проєктів. Своєчасна та правильна обробка вимог значно зменшує ризики та витрати.

Управління конфігурацією (Configuration Management)

Управління конфігурацією включає системні процеси для контролю та отримання інформації про стан артефактів проєкту:

- **Планування конфігурації:** Визначення артефактів, які потребують контролю (вихідний код, документація, конфігураційні файли).
- **Контроль змін:** Запити на зміни (Change Requests) проходять процес огляду та затвердження перед впровадженням.
- **Версіонування:** Ведення історії змін, можливість повернення до попередніх версій.
- **Аудит статусу:** Перевірка та забезпечення узгодженості конфігурації з плануванням.

Інструменти: Git, SVN, Mercurial (версіонування); Jenkins, GitLab CI/CD (автоматизація збірки та розгортання).

Практики поліпшення процесів

Неперервне вдосконалення (Continuous Improvement)

SWEBOK v4 наголошує на важливості неперервного вдосконалення та рефлексії:

- **Post-Mortem аналіз:** Огляд проєкту після його завершення для виявлення уроків та рекомендацій.
- **Метрики та вимірювання:** Збір даних про процес та продукт для обґрунтованого прийняття рішень (velocity, defect rate, lead time тощо).
- **Адаптація практик:** На основі зібраних даних коригування процесів та практик для поліпшення якості та продуктивності.

Висновки

- Не існує "срібної кулі" — ідеальної методології для всіх проєктів. Вибір залежить від контексту, вимог, команди та ризиків.
- **Waterfall** підходить для проєктів з низькою невизначеністю, стабільними вимогами та високою ціною помилки.
- **V-Model** подібний до Waterfall, але з більшим акцентом на якість та тестування на кожному рівні.
- **Спіральна модель** фокусується на управлінні ризиками та найкраще підходить для складних, дорогих проєктів.
- **Agile** (Scrum/Kanban) є стандартом де-факто для більшості сучасних веб- та мобільних розробок, де вимоги швидко змінюються, а швидкість виходу на ринок (Time-to-Market) є критичною.
- **Scrum** забезпечує структурований підхід з фіксованими спринтами, ролями та подіями.
- **Kanban** забезпечує більш гнучкий, потоко-орієнтований підхід без фіксованих спринтів.

- Розуміння SDLC моделей дозволяє інженерам та менеджерам обирати оптимальний підхід для конкретної ситуації.

Контрольні запитання

1. Що таке SDLC? Розшифруйте абревіатуру та дайте визначення.
2. Назвіть 6 основних фаз життєвого циклу ПЗ та коротко опишіть кожну.
3. У чому головна особливість каскадної моделі (Waterfall)?
4. Назвіть три основні недоліки моделі Waterfall.
5. Для яких типів проєктів найкраще підходить Waterfall? Наведіть приклади.
6. У чому різниця між верифікацією та валідацією у контексті V-Model?
7. Поясніть різницю між ітеративним та інкрементальним підходами (можна на прикладі створення картини або іншого продукту).
8. Яка головна особливість спіральної моделі Баррі Боема і які 4 сектори вона включає?
9. Назвіть 4 цінності Agile Manifesto.
10. Чому в Agile “співпраця з замовником” важливіша за “узгодження умов контракту”?
11. Хто такий Product Owner у Scrum і за що він відповідає?
12. Яка роль Scrum Master у команді? Чим він відрізняється від традиційного Project Manager?
13. Що таке Sprint і яка його рекомендована тривалість?
14. Яка мета події Daily Scrum і скільки вона тривала?
15. Чим відрізняється Sprint Review від Sprint Retrospective?
16. Що таке WIP (Work In Progress) у Kanban і навіщо його обмежувати?

17. Назвіть основну відмінність між Scrum та Kanban у підході до планування роботи та ітерацій.

18. Що таке “технічний борг” і як він може накопичуватися при неправильному виборі процесу?

19. Як вибір методології впливає на вартість внесення змін у проєкт на пізніх етапах?

20. Чи можна комбінувати елементи різних методологій (наприклад, Scrumban)? Наведіть приклад ситуації, де це доцільно.

21. Що таке управління вимогами та які основні практики воно включає?

22. Назвіть етапи управління вимогами та поясніть кожен з них.

23. Чому валідація вимог є важливою та яких помилок вона допомагає уникнути?

24. Що таке управління конфігурацією та які артефакти воно контролює?

25. Поясніть різницю між версіонуванням та контролем змін.

26. Які роль continuous improvement в поліпшенні якості проєктів?

27. Що таке post-mortem аналіз і як він допомагає командам вчитися?

28. Назвіть основні метрики для вимірювання якості та ефективності SDLC процесу.

29. Як SWEBOK v4 переглядає та розширює традиційне розуміння життєвого циклу ПЗ?

Лекція 3

Робота з вимогами

Інженерія вимог: збір, аналіз та документування вимог до програмного забезпечення.

Предмет, мета та завдання лекції

Предмет: Інженерія вимог (Requirements Engineering), методи збору, аналізу та документування вимог до програмного забезпечення.

Мета: Навчитися виявляти потреби замовника, розрізняти типи вимог та формулювати їх у вигляді чітких специфікацій або історій користувачів.

Завдання:

- Зрозуміти важливість якісних вимог для успіху проєкту.
- Вивчити класифікацію вимог (функціональні vs нефункціональні).
- Розглянути процес інженерії вимог.
- Ознайомитися з техніками збору вимог.
- Навчитися писати User Stories та складати SRS.

Навчальні результати

Після завершення лекції студент зможе:

1. Пояснити визначення вимоги та її роль у розробці ПЗ.

2. Розрізняти функціональні та нефункціональні вимоги.
3. Назвати рівні вимог та описати кожний з них.
4. Описати чотири основні етапи процесу інженерії вимог.
5. Застосовувати різні техніки збору вимог залежно від контексту.
6. Писати якісні User Stories з критеріями прийняття.
7. Розуміти структуру та вимоги до SRS документу.
8. Проводити валідацію вимог.

Значення вимог

Визначення вимог та інженерії вимог

Вимога (Requirement) — це умова або здатність, якій повинна відповідати система, щоб задовольнити угоду, стандарт, специфікацію або інші офіційно встановлені документи.

Інженерія вимог (Requirements Engineering) — це системний процес виявлення, аналізу, документування та перевірки вимог до програмного забезпечення [13].

Відомий фахівець у галузі програмної інженерії Fred Brooks писав

> Найважча частина створення програмної системи — це вирішити, що саме створювати.

Ця цитата підкреслює критичну важливість етапу роботи з вимогами для успіху проєкту.

Ціна помилки

Однією з найважливіших причин невдач програмних проєктів є помилки у вимогах. Дослідження показують:

- **Виявлення на етапі аналізу:** Вартість виправлення — відносно низька (базова одиниця).

- **Виявлення на етапі тестування:** Вартість виправлення — 5–10 разів дорожче.
- **Виявлення на етапі експлуатації:** Вартість виправлення — 100 разів дорожче.

Це означає, що рання інвестиція в якісну роботу з вимогами значно зменшує загальну вартість проєкту.

Класифікація вимог

Рівні вимог (Pyramid of Requirements)

Вимоги організовуються у ієрархію, де кожний рівень деталізує попередній:

Бізнес-вимоги (Business Requirements)

Визначають **найвищорівневі цілі** організації та причини розпочинання проєкту.

- Відповідають на питання *Навіщо нам цей проєкт?*
- Зазвичай формулюються у термінах бізнес-цілей та переваг.
- *Приклад:* “Збільшити обсяги продажів на 20

Вимоги користувача (User Requirements)

Описують **задачі та цілі** окремих користувачів або груп користувачів.

- Відповідають на питання *Що користувачі мають робити із системою?*
- Часто формулюються на основі User Stories.
- *Приклад:* “Як покупець, я хочу переглядати каталог товарів та фільтрувати їх за ціною”

Системні вимоги (System Requirements)

Надають **детальний опис** функцій, обмежень та поведінки системи, яку розробники мають реалізувати.

- Найбільш конкретні та технічні.
- Поділяються на функціональні та нефункціональні вимоги.
- *Приклад:* "Система повинна обробляти до 1000 одночасних користувачів"

Типи системних вимог

Функціональні вимоги (Functional Requirements)

Описують **що** система повинна робити — її поведінку, функції та операції.

Характеристики:

- Описують певні дії та реакції системи.
- Повинні бути вимірювані та тестовані.
- Часто формулюються як "система повинна . . . "

Приклади:

- "Система повинна надсилати лист підтвердження електронною поштою після успішної реєстрації користувача"
- "Користувач повинен мати можливість фільтрувати товари за ціною, категорією та рейтингом"
- "Система повинна обчислювати вартість доставки на основі ваги та відстані"

Нефункціональні вимоги (Non-functional Requirements)

Описують **як** система повинна працювати — атрибути якості та властивості (часто називаються "-ilities" або "Quality Attributes").

Основні категорії:

Продуктивність (Performance): Швидкість роботи, час відгуку, пропускна здатність. *Приклад:* “Час завантаження домашної сторінки не повинен перевищувати 2 секунди”

Надійність (Reliability): Час безвідмовної роботи (uptime), частота відмов, можливість відновлення. *Приклад:* “Система повинна працювати з часом безвідмовної роботи не менше 99.9%”

Безпека (Security): Захист даних, контроль доступу, шифрування, аудит. *Приклад:* “Усі передачі даних повинні бути захищені TLS 1.2 або вищою версією”

Зручність (Usability): Легкість навчання та використання, доступність. *Приклад:* “Новий користувач повинен мати змогу виконати першу покупку протягом 5 хвилин”

Масштабованість (Scalability): Можливість системи обробляти зростаючі обсяги даних та користувачів. *Приклад:* “Система повинна підтримувати зростання кількості користувачів на 100% без переробки архітектури”

Обслуговуваність (Maintainability): Легкість внесення змін, розширення та виправлення помилок. *Приклад:* “Код повинен відповідати стандартам Python PEP 8 та мати тестове покриття не менше 80%”

Портативність (Portability): Можливість запуску на різних платформах. *Приклад:* “Застосунок повинен працювати на iOS та Android”

Процес інженерії вимог

Процес роботи з вимогами є ітеративним і складається з чотирьох основних етапів:

1. Виявлення вимог (Elicitation)

Спілкування та взаємодія зі стейкхолдерами (зацікавленими сторонами) для розуміння їхніх потреб, проблем та очікувань.

Ключові питання:

- Хто такі стейкхолдери?
- Які проблеми вони вирішують зараз?
- Які цілі вони хочуть досягти?
- Які обмеження та ризики вони бачать?

2. Аналіз вимог (Analysis)

Глибоке вивчення зібраних вимог для вирішення конфліктів, уточнення неясних моментів та пріоритизації.

Діяльності:

- Вирішення конфліктів між вимогами різних стейкхолдерів.
- Пріоритизація вимог (наприклад RICE Scoring):

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

- Вивчення здійсненності та залежностей вимог.
- Оцінка впливу на архітектуру та розклад.

3. Специфікація вимог (Specification)

Документування вимог у формальному та структурованому вигляді, який розумітимуть всі учасники проєкту.

Формати:

- Software Requirements Specification (SRS) документ за стандартом IEEE 830.
- User Stories з критеріями прийняття.
- Зображувальні моделі (діаграми, макети).

4. Валідація вимог (Validation)

Перевірка того, що задокументовані вимоги:

- Відповідають справжнім потребам замовника та користувачів.
- Є повними та несуперечливими.
- Можуть бути реалізовані та протестовані.
- Узгоджені всіма зацікавленими сторонами.

Техніки збору вимог

Для ефективного збору вимог розроблено низку перевірених техник [14].

Інтерв'ю (Interviews)

Пряме спілкування з користувачами та стейкхолдерами.

Типи:

- **Структуровані:** Заздалегідь підготовлені питання.
- **Напівструктуровані:** Загальна тема, але гнучкість у питаннях.
- **Вільні:** Розмова без конкретної структури.

Переваги: Глибоке розуміння, отримання контексту, можливість уточнення.

Недоліки: Часозатратно, залежить від навичок інтерв'юера.

Анкетування (Questionnaires)

Збір даних від великої кількості людей через письмові питання.

Переваги: Охоплює багато людей, економічне, анонімне.

Недоліки: Низька якість відповідей, не можна уточнити.

Мозковий штурм (Brainstorming)

Генерація ідей групою людей без критики та оцінювання.

Правила:

- Жодна ідея не критикується.
- Вільна асоціація та поєднання ідей.
- Кількість переважає якість (на початку).

Спостереження (Observation/Job Shadowing)

Аналіз того, як користувачі насправді працюють та які проблеми виникають під час роботи.

Переваги: Виявляються неявні процеси та проблеми, які люди не завжди артикулюють.

Прототипування (Prototyping)

Створення макетів інтерфейсу або мінімально функціонального прототипу для отримання швидкого зворотного зв'язку.

Переваги: Користувачам легше обговорювати те, що вони бачать. Раннє виявлення помилок у концепції.

Документування вимог

SRS (Software Requirements Specification)

Класичний документ, що містить повний опис системи. Зазвичай складається за стандартом IEEE 830 або ISO/IEC/IEEE 29148:2018 [15], [16].

Типова структура SRS

1. **Вступ** — Мета документу, область застосування, визначення термінів.
2. **Загальний опис** — Контекст продукту, функції продукту, користувачі та обмеження.

3. **Специфічні вимоги —**

    - Функціональні вимоги (детальні, з прикладами та сценаріями).
    - Вимоги до користувацького інтерфейсу.
    - Вимоги до апаратного забезпечення.
    - Вимоги до програмного забезпечення.
    - Вимоги до комунікацій.
    - Нефункціональні вимоги (продуктивність, безпека, надійність тощо).

4. **Додатки** — Діаграми, таблиці, інший допоміжний матеріал.

Характеристики якісної вимоги

Коректність (Correctness): Вимога точно описує те, що система повинна робити.

Недвозначність (Unambiguity): Вимога має лише одне тлумачення. *Приклад неоднозначної вимоги:* “Система повинна працювати швидко” *Коректна формулювання:* “Час відгуку на запит користувача не повинен перевищувати 2 секунди”.

Повнота (Completeness): Вимога достатня для розробки та тестування, без посилання на зовнішні джерела.

Несуперечливість (Consistency): Вимога не суперечить іншим вимогам.

Можливість перевірки (Testability/Verifiability): Можна визначити тест, що перевіряє виконання вимоги.

Можливість трасування (Traceability): Можна простежити вимогу до бізнес-цілей та реалізації.

Рангованість (Ranked): Вимога має визначений пріоритет.

User Stories

Підхід Agile до документування вимог. Короткий опис функціональності з точки зору користувача.

Формат User Story

Стандартний формат:

Як <роль>, я хочу <дія/функція>, щоб <цінність/причина>.

Приклади:

- “Як покупець, я хочу додати товар у кошик, щоб купити його пізніше”.
- “Як адміністратор, я хочу переглядати логи системи, щоб діагностувати проблеми”.
- “Як новий користувач, я хочу здійснити реєстрацію через Google, щоб не запам’ятовувати ще один пароль”.

Переваги:

- Фокус на цінності для користувача, а не на технічних деталях.
- Коротко, зрозуміло, легко обговорювати.
- Прості для оцінки вартості та планування.

Критерії прийняття (Acceptance Criteria)

Конкретні умови, які мають бути виконані, щоб User Story вважалася завершеною.

Формат Gherkin:

```
Given (Дано) <контекст>
When (Коли) <дія>
Then (Тоді) <очікуваний результат>
```

Приклад для Story “Додати товар у кошик”:

```
Given (Дано) користувач перебуває на сторінці товару
When (Коли) користувач натискає кнопку "Додати у кошик"
Then (Тоді) товар додається до кошика
And (І) на сторінці з’являється повідомлення про успіх
And (І) лічильник товарів у кошику збільшується на 1
```

Модель INVEST для User Stories

Критерії якісної User Story (запропоновані Bill Wake):

Independent: Історія повинна бути незалежною від інших, наскільки це можливо.

Negotiable: Деталі історії можуть обговорюватися та змінюватися.

Valuable: Історія повинна мати цінність для користувача або бізнесу.

Estimable: Команда повинна мати змогу оцінити вартість роботи.

Small: Історія повинна бути достатньо малою, щоб бути завершеною в одному спринті.

Testable: Повинна бути можливість написати тести для перевірки.

Міжнародні стандарти та практики

ISO/IEC/IEEE 29148:2018

Сучасний міжнародний стандарт “Systems and software engineering — Life cycle processes — Requirements engineering” [5].

Основні положення:

- Визначає процеси для розвитку та управління вимогами.
- Визначає структуру для документів вимог (SyRS — System Requirements Specification, SRS — Software Requirements Specification).
- Вимагає трасування від бізнес-цілей до реалізації та тестування.

ISO/IEC 25010 — Модель якості ПЗ

Стандарт “Systems and software Quality Requirements and Evaluation (SQuaRE)” визначає категорії нефункціональних вимог (надійність, ефективність, зручність супроводу тощо).

SWEBOK

Область знань “Software Requirements” у SWEBOK описує фундаментальні поняття та практики для роботи з вимогами.

Приклади з Open Source проєктів

Kubernetes Enhancement Proposals (KEPs)

Проєкт Kubernetes використовує структуровану форму для документування планів щодо нових функцій.

Структура KEP:

- Summary — Коротке резюме.
- Motivation — Чому це потрібно?
- Proposal — Що ми пропонуємо?
- Non-Goals — Що ми НЕ робимо?
- Design — Детальний дизайн.
- Test Plan — Як ми це тестуватимемо?
- Risks and Mitigations — Потенційні ризики та способи їхнього пом’якшення.

Чому це якісно: Чітка структура, обов’язковий розгляд альтернатив та ризиків, залучення спільноти.

GitLab — публічна розробка та Epics/Issues

GitLab розвивається повністю публічно. Вимоги оформлюються як:

- **Epics** — великі функції або ініціативи.
- **Issues** — конкретні завдання з детальним описом.

Кожна Issue містить:

- “Problem to solve” — що вирішуємо?
- “Intended users” — для кого це?
- “Success criteria” — як ми виміримо успіх?
- Пряме посилання на Merge Request (код, що реалізує вимогу).

Чому це якісно: Фокус на цінності для користувача, прозорість обговорення, прямий зв’язок між вимогою та реалізацією.

Висновки

- **Інженерія вимог — це фундамент проєкту.** Погані вимоги неможливо компенсувати гарним кодом.
- **Розрізняйте функціональні та нефункціональні вимоги.** Обидва типи критично важливі для успіху системи.
- **Використовуйте правильні техніки збору вимог** залежно від контексту проєкту та типу стейкхолдерів.
- **У сучасному світі відбувається перехід** від важких документів (SRS) до гнучких форматів (User Stories), але суть процесу (зрозуміти потребу) залишається незмінною.
- **Якість вимог** можна забезпечити дотриманням критеріїв коректності, недвозначності, повноти, тестовності.
- **Валідація вимог** — це не одноразовий процес, а постійна комунікація зі стейкхолдерами.

Контрольні запитання

1. Що таке вимога в контексті програмної інженерії?
2. Чому виправлення помилки у вимогах на етапі тестування коштує дорожче, ніж на етапі аналізу?

3. Назвіть три рівні вимог (піраміда вимог).
4. У чому різниця між функціональними та нефункціональними вимогами? Наведіть по 2 приклади.
5. Що таке стейкхолдер? Наведіть приклади стейкхолдерів для інтернет-магазину.
6. Перелічіть 4 основні етапи процесу інженерії вимог.
7. Яка техніка збору вимог є найбільш ефективною для вирішення конфліктів між стейкхолдерами?
8. У чому перевага прототипування при зборі вимог?
9. Що таке SRS? Розшифруйте абревіатуру.
10. Назвіть 5 характеристик якісної вимоги.
11. Що означає “недвозначність” вимоги? Наведіть приклад двозначної вимоги.
12. Що таке User Story? Назвіть стандартний шаблон її написання.
13. Для чого використовуються критерії прийняття (Acceptance Criteria)?
14. Що таке модель INVEST для User Stories?
15. Як нефункціональні вимоги впливають на архітектуру системи?
16. Що таке “Scope Creep” (розповзання меж проєкту) і як якісні вимоги допомагають цьому запобігти?
17. Чим відрізняється інтерв’ю від анкетування? Коли краще використовувати кожен метод?
18. Що таке Traceability (трасування) вимог і навіщо воно потрібне?
19. Як перевірити (валідувати) вимогу, якщо код ще не написаний?
20. Наведіть приклад нефункціональної вимоги до безпеки банківського додатку.

Лекція 4

Основи архітектури та дизайну ПЗ

Архітектура програмного забезпечення, принципи об'єктно-орієнтованого проєктування (SOLID) та шаблони проєктування (Design Patterns).

Предмет, мета та завдання лекції

Предмет: Архітектура програмного забезпечення, принципи об'єктно-орієнтованого проєктування (SOLID) та шаблони проєктування (Design Patterns).

Мета: Навчитися проєктувати гнучкі, масштабовані та придатні до супроводу програмні системи, розуміти різницю між архітектурою та дизайном.

Завдання:

- Зрозуміти відмінність між архітектурою та детальним дизайном.
- Розглянути основні архітектурні стилі (Моноліт, Мікросервіси, Багатошарова).
- Вивчити принципи SOLID.
- Ознайомитися з класифікацією патернів проєктування (GoF).
- Оглянути стандарти документування архітектури.

Навчальні результати

Після завершення лекції студент зможе:

1. Розрізняти архітектуру та детальний дизайн ПЗ.
2. Описати основні архітектурні стилі та їхні характеристики.
3. Пояснити кожен принцип SOLID та навести приклади їхнього порушення.
4. Класифікувати патерни проєктування за трьома категоріями (Твірні, Структурні, Поведінкові).
5. Обрати відповідний архітектурний стиль для конкретної задачі.
6. Розуміти роль архітектурних стандартів у розробці ПЗ.
7. Застосовувати SOLID принципи та патерни при проєктуванні систем.

Архітектура vs Дизайн

Хоча ці терміни часто використовуються як синоніми, вони мають різний масштаб та впливають на різні аспекти розробки:

Архітектура ПЗ (Software Architecture)

Визначення: Це високорівнева структура системи. Вона визначає компоненти системи, їх взаємодію, вибір технологій та глобальні обмеження. Архітектура — це “стратегія” системи.

Приклади архітектурних рішень:

- Вибір між мікросервісами та монолітом.
- Вибір бази даних та типу сховища даних.
- Вибір протоколів комунікації (REST, gRPC, WebSocket).
- Рішення про розподіл системи на модулі та їхні межі.

Характеристики:

- Вплив на всю систему.
- Важко змінювати після рішення.
- Визначається на ранніх етапах проєкту.
- Стосується бізнес-вимог і загальних стратегій.

Дизайн ПЗ (Software Design)

Визначення: Це деталізація окремих модулів, класів та функцій. Дизайн — це "тактика" реалізації окремих компонентів.

Приклади дизайнерських рішень:

- Як реалізувати конкретний клас та його методи.
- Який патерн використати для створення об'єкта (Singleton, Factory).
- Як організувати структуру класів та їхні взаємозв'язки.
- Як розподілити відповідальність між методами.

Характеристики:

- Вплив на окремі компоненти або модулі.
- Відносно легко змінюватись під час розробки.
- Деталізується під час розробки.
- Стосується технічних аспектів реалізації.

Висловлювання Martin Fowler

Відомий архітектор та лідер думок у галузі ПЗ Martin Fowler висловився так [17]:

Архітектура — це ті рішення, які важко змінити пізніше.

Ця цитата підкреслює критичну важливість якісних архітектурних рішень на ранніх етапах розробки.

Архітектурні стилі (Architectural Patterns)

Архітектурні стилі — це встановлені способи організації загальної структури програмної системи. Кожен стиль має свої переваги та недоліки, та підходить для різних типів проєктів.

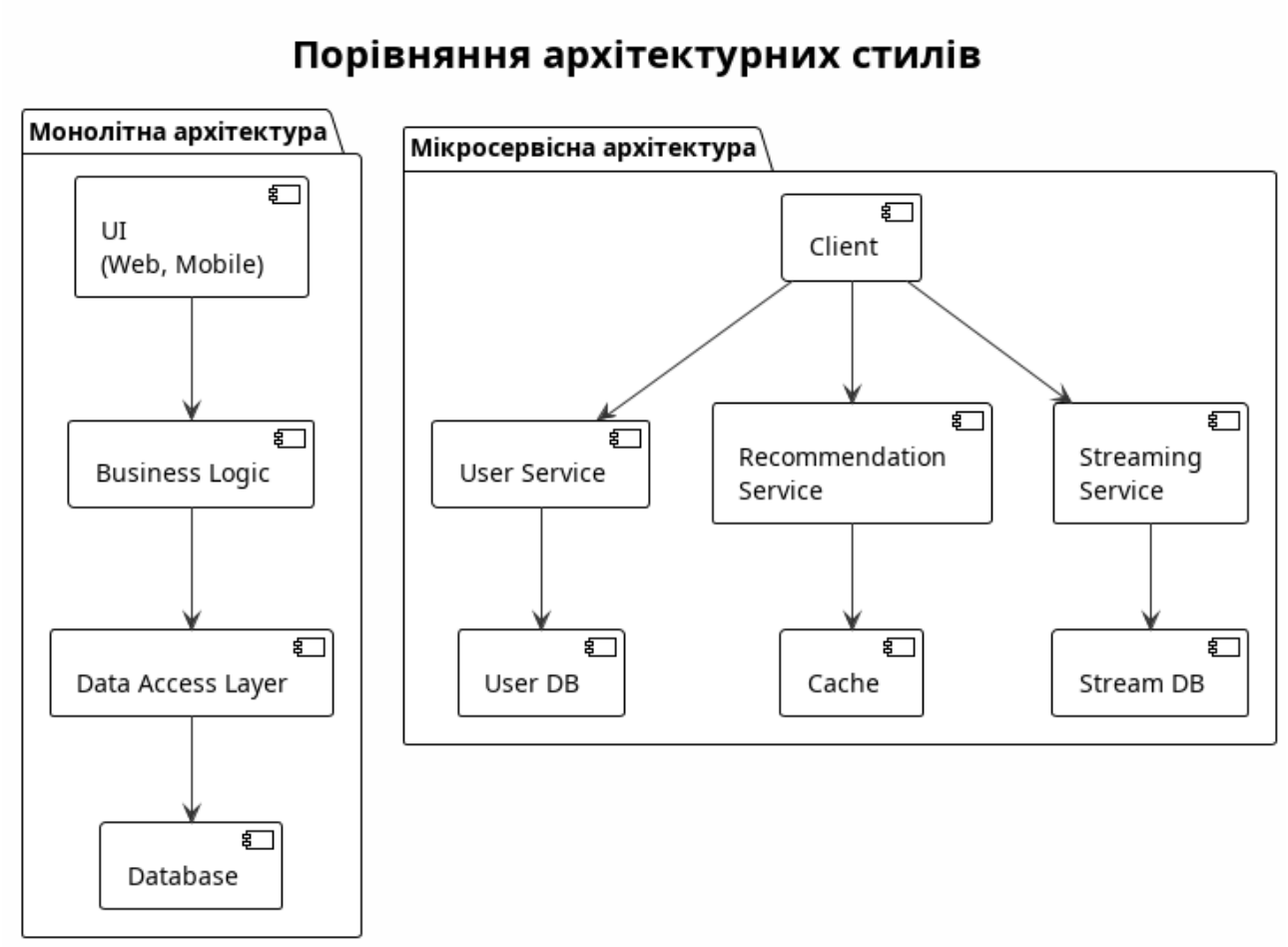

Рис. 4.1: Порівняння архітектурних стилів: Монолітна та мікросервісна архітектура

1. Монолітна архітектура (Monolithic)

Вся система є єдиним модулем, що розгортається (deploy) разом.

Характеристики

- Всі компоненти скомпільовані в один виконуваний файл або розгорнуті як один сервіс.
- Спільна база даних для всіх функцій.

- Синхронна комунікація між компонентами.

Переваги

- **Простота розробки:** Легше розробляти та тестувати на початкових етапах.
- **Простота розгортання:** Розгортається як один файл.
- **Середовище виконання:** Менше накладних витрат на мережеву комунікацію.
- **Простота налагодження:** Всі логи та трасування в одному місці.

Недоліки

- **Масштабованість:** Неможливо масштабувати окремі компоненти незалежно.
- **Надійність:** Одна помилка може "покласти" всю систему.
- **Гнучкість технологій:** Всі компоненти мають бути написані однією мовою.
- **Складність оновлення:** Навіть невеликі зміни вимагають переконфігурації та перевикористання всієї системи.

2. Багатошарова архітектура (Layered / N-Tier)

Система розділена на логічні шари, кожен з яких має свою відповідальність та інтерфейс.

Структура багатошарової архітектури

Presentation Layer (Шар представлення): Користувацький інтерфейс, взаємодія з користувачем. Це може бути веб-інтерфейс, мобільний додаток або десктопний додаток.

Business Logic Layer (Шар бізнес-логіки): Обробка даних, правила ведення бізнесу, валідація та трансформація даних.

Data Access Layer (Шар доступу до даних): Робота з базою даних, кеш, зовнішні сервіси. Абстрагує способи зберігання даних.

Database Layer (Шар бази даних): Фізичне сховище даних.

Переваги

- **Розділення відповідальності:** Кожен шар має чіткі обов'язки.
- **Легкість заміни:** Можна замінити окремий шар без змін інших.
- **Тестованість:** Легше писати unit тести для окремих шарів.
- **Розуміння:** Структура легко зрозуміла новим розробникам.

Недоліки

- **Продуктивність:** Перетворення даних між шарами може бути дорогою операцією.
- **Монолітність:** Часто залишається монолітною архітектурою.
- **Масштабованість:** Складно масштабувати окремі шари незалежно.

3. Мікросервісна архітектура (Microservices)

Система складається з набору невеликих, незалежних сервісів, що спілкуються через мережу (REST, gRPC, повідомлення).

Характеристики

- Кожен сервіс відповідає за одну бізнес-функцію.
- Кожен сервіс розгортається незалежно.
- Кожен сервіс має власну базу даних.
- Асинхронна комунікація між сервісами.

Переваги

- **Гнучкість:** Можна розробляти різні сервіси різними мовами.
- **Масштабованість:** Кожен сервіс масштабується незалежно.
- **Надійність:** Падіння одного сервісу не впливає на інші.
- **Швидкість розробки:** Малі команди можуть працювати незалежно.

Недоліки

- **Складність інфраструктури:** Потребує розвинутої DevOps культури та інструментів.
- **Розподілені транзакції:** Складно забезпечити узгодженість (consistency) даних.
- **Складність налагодження:** Помилка може бути у будь-якому сервісі.
- **Накладні витрати мережі:** Комунікація між сервісами повільніша, ніж локальні виклики.

4. Клієнт-серверна архітектура (Client-Server)

Розподіл навантаження та функцій між постачальником ресурсу (сервер) та замовником (клієнт).

Характеристики

- **Клієнт:** Ініціює запити та отримує відповіді.
- **Сервер:** Обробляє запити та надає ресурси.
- **Комунікація:** Зазвичай синхронна через мережу.

Приклади: Веб-додатки (браузер — сервер), мобільні додатки, класичні клієнт-серверні програми.

Принципи проєктування SOLID

SOLID — це мнемонічна абревіатура для п'яти основних принципів об'єктно-орієнтованого програмування, розроблених Robert Martin (Uncle Bob) [18]. Дотримання цих принципів допомагає створювати код, який легко читати, тестувати та розширювати.

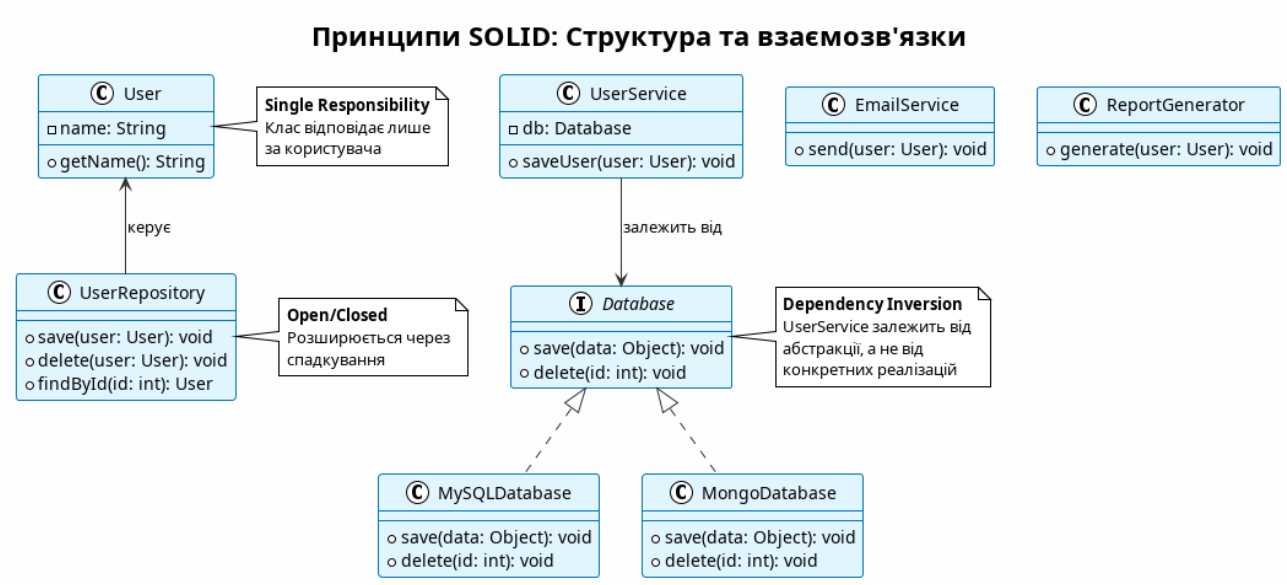

Рис. 4.2: Принципи SOLID: Структура та взаємозв'язки (Single Responsibility, Open/Closed, Dependency Inversion)

S — Single Responsibility Principle (Принцип єдиної відповідальності)

Формулювання: Клас повинен мати лише одну причину для змін (робити щось одне, і робити це добре).

Сенс: Кожен клас повинен відповідати за одну функцію або аспект системи. Якщо клас відповідає за декілька не пов'язаних обов'язків, він важче змінювати та тестувати.

Приклад порушення:

```
class User:
    def __init__(self, name):
        self.name = name
```

```
    def save_to_database(self):
        # Збереження у БД
        pass

    def send_email(self):
        # Надсилання листа
        pass

    def generate_report(self):
        # Генерація звіту
        pass
```

Клас *User* відповідає за багато речей: управління користувачем, роботу з БД, надсилання листів, генерацію звітів. Це порушує SRP.

Коректна реалізація:

```
class User:
    def __init__(self, name):
        self.name = name

class UserRepository:
    def save(self, user):
        # Збереження у БД
        pass

class EmailService:
    def send(self, user):
        # Надсилання листа
        pass

class ReportGenerator:
    def generate(self, user):
        # Генерація звіту
        pass
```

O — Open/Closed Principle (Принцип відкритості/закритості)

Формулювання: Програмні сутності (класи, модулі, функції) повинні бути **відкриті для розширення**, але **закриті для модифікації**.

Сенс: Ви повинні мати можливість додавати нову функціональність, не змінюючи існуючий код. Зазвичай це досягається через спадкування або композицію.

Приклад порушення:

```
class PaymentProcessor:
    def process(self, payment):
        if payment.type == "credit_card":
            # Обробка кредитної карти
        elif payment.type == "paypal":
            # Обробка PayPal
        elif payment.type == "bitcoin":
            # Обробка Bitcoin
```

Щоб додати новий спосіб оплати, потрібно змінювати існуючий клас.

Коректна реалізація:

```
class PaymentProcessor:
    def process(self, payment):
        payment.process()

class CreditCardPayment:
    def process(self):
        # Обробка кредитної карти
        pass

class PayPalPayment:
    def process(self):
        # Обробка PayPal
        pass
```

Тепер можна додавати нові способи оплати без змін *PaymentProcessor*.

L — Liskov Substitution Principle (Принцип підстановки Лісков)

Формулювання: Об'єкти підкласів повинні замінювати об'єкти батьківських класів без порушення роботи програми.

Сенс: Якщо клас S є підклас класу T, то об'єкти типу T можна замінити об'єктами типу S без будь-яких побічних ефектів.

Приклад порушення:

```
class Bird:
    def fly(self):
        pass

class Penguin(Bird):
    def fly(self):
        raise Exception("Пінгвіни не літають!")
```

Penguin є підклас *Bird*, але не може літати. Це порушує принцип, оскільки код, який очікує *Bird*, зламається при *Penguin*.

Коректна реалізація:

```
class Bird:
    pass

class FlyingBird(Bird):
    def fly(self):
        pass

class Penguin(Bird):
    def swim(self):
        pass
```

I — Interface Segregation Principle (Принцип розділення інтерфейсу)

Формулювання: Клієнти не повинні залежати від інтерфейсів, які вони не використовують. Краще багато вузькоспеціалізованих інтерфейсів, ніж один

універсальний.

Сенс: Уникайте створення “жирних” інтерфейсів, які примушують класи реалізовувати непотрібні методи.

Приклад порушення:

```
class Worker:
    def work(self):
        pass

    def eat(self):
        pass

class Robot(Worker):
    def work(self):
        pass

    def eat(self):
        raise Exception("Роботи не їдять!")
```

Коректна реалізація:

```
class Workable:
    def work(self):
        pass

class Eatable:
    def eat(self):
        pass

class Human(Workable, Eatable):
    pass

class Robot(Workable):
    pass
```

D — Dependency Inversion Principle (Принцип інверсії залежностей)

Формулювання: Модулі вищого рівня не повинні залежати від модулів нижчого рівня. Обидва повинні залежати від абстракцій.

Сенс: Залежності мають бути спрямовані на абстракції (інтерфейси), а не на конкретні реалізації.

Приклад порушення:

```
class MySQLDatabase:
    def save(self, data):
        pass

class UserService:
    def __init__(self):
        self.db = MySQLDatabase()

    def save_user(self, user):
        self.db.save(user)
```

UserService залежить від конкретної реалізації *MySQLDatabase*.

Коректна реалізація:

```
class Database:
    def save(self, data):
        pass

class MySQLDatabase(Database):
    def save(self, data):
        pass

class UserService:
    def __init__(self, db: Database):
        self.db = db

    def save_user(self, user):
```

```
        self.db.save(user)
```

Тепер *UserService* залежить від абстракції *Database*.

Патерни проєктування (Design Patterns)

Патерн — це типове рішення поширеної проблеми при проєктуванні. Класична книга “Gang of Four: Design Patterns” (Gamma, Helm, Johnson, Vlissides) [19] описує 23 класичні патерни, розділені на 3 категорії.

Класифікація патернів

1. Твірні патерни (Creational Patterns)

Відповідають за механізми створення об’єктів. Вони дозволяють створювати об’єкти гнучко, незалежно від того, як вони мають бути скомпоновані та представлені.

Основні патерни:

Singleton (Одинак): Гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до цього екземпляра. *Приклад:* Logger, DatabaseConnection, ConfigManager.

Factory Method (Фабричний метод): Визначає інтерфейс для створення об’єкта, але дозволяє підкласам вирішувати, екземпляр якого класу створити. *Приклад:* Створення різних типів платежів (CreditCard, PayPal, Bitcoin).

Abstract Factory (Абстрактна фабрика): Надає інтерфейс для створення сімейств пов’язаних або залежних об’єктів без указання їхніх конкретних класів.

Builder (Будівельник): Дозволяє створювати складні об’єкти покроково, розділяючи побудову та представлення. *Приклад:* Конструювання SQL запитів, побудова конфігурацій.

Prototype (Прототип): Дозволяє копіювати об’єкти без залежності від їхніх класів.

2. Структурні патерни (Structural Patterns)

Визначають, як класи та об'єкти компонуються в більші структури, зберігаючи гнучкість та ефективність цих структур.

Основні патерни:

Adapter (Адаптер): Дозволяє об'єктам з несумісними інтерфейсами працювати разом, перетворюючи інтерфейс одного класу на інтерфейс іншого. *Приклад:* Адаптація старого коду до нового інтерфейсу.

Bridge (Міст): Розділяє абстракцію від реалізації, щоб вони могли змінюватися незалежно.

Composite (Композит): Складає об'єкти в деревоподібні структури для представлення ієрархії «частина-ціле». Дозволяє клієнтам однаково обробляти окремі об'єкти та комбінації об'єктів.

Decorator (Декоратор): Додає нову поведінку об'єкту динамічно, обгортаючи його в об'єкті, який надає цю поведінку. *Приклад:* Додання функціональності до HTTP запитів (логування, кеш).

Facade (Фасад): Надає простий, уніфікований інтерфейс до складної системи класів, бібліотеки або підсистеми. *Приклад:* Фасад для складної бібліотеки для роботи з PDF.

Flyweight (Легковаговик): Використовує спільний об'єкт для ефективного підтримання великої кількості дрібних об'єктів.

Proxy (Замісник): Надає замісник іншого об'єкту для контролю доступу до нього. *Приклад:* Lazy loading, кешування, контроль доступу.

3. Поведінкові патерни (Behavioral Patterns)

Відповідають за взаємодію та розподіл обов'язків між об'єктами. Вони описують, як об'єкти взаємодіють та розподіляють відповідальність.

Основні патерни:

Chain of Responsibility (Ланцюг відповідальності): Дозволяє передавати запит вдовж ланцюга обробників, де кожен обробник вирішує, чи він буде його обробляти.

Command (Команда): Перетворює запит на об'єкт, який дозволяє параметризувати клієнтів з різними запитами, ставити запити в чергу, логувати запити та підтримувати скасування операцій.

Interpreter (Інтерпретатор): Визначає представлення для граматики мови та інтерпретатор для обробки речень цієї мови.

Iterator (Ітератор): Забезпечує спосіб послідовного доступу до елементів об'єкта-контейнера без розкриття його базового представлення.

Mediator (Посередник): Визначає об'єкт, який інкапсулює спосіб взаємодії набору об'єктів.

Memento (Знімок): Без порушення інкапсуляції, захоплює та екстеріоризує внутрішній стан об'єкта, дозволяючи пізніше відновити об'єкт до цього стану.

Observer (Спостерігач): Визначає залежність один-до-багатьох між об'єктами, так що коли один об'єкт змінює стан, всі його залежні об'єкти автоматично сповіщаються та оновлюються. *Приклад:* Система подій (наприклад, натискання кнопки), підписка на оновлення.

State (Стан): Дозволяє об'єкту змінювати свою поведінку залежно від свого внутрішнього стану. При цьому здається, що змінився клас об'єкта.

Strategy (Стратегія): Визначає сімейство алгоритмів, інкапсулює кожен з них та робить їх взаємозамінними. *Приклад:* Різні алгоритми сортування, різні способи обчислення ціни доставки.

Template Method (Шаблонний метод): Визначає основу алгоритму в методі базового класу, залишаючи деталі для підкласів.

Visitor (Відвідувач): Представляє операцію, яка буде виконана над елементами об'єктної структури. Дозволяє визначити нову операцію без зміни класів елементів, над якими вона функціонує.

Ключові міжнародні стандарти

ISO/IEC/IEEE 42010:2011

“Systems and software engineering — Architecture description” [20]

Цей міжнародний стандарт регулює документування архітектури систем та програмного забезпечення.

Основні концепції:

- **Architecture View (Архітектурне подання):** Представлення архітектури з точки зору конкретних зацікавлених сторін та їхніх проблем. *Приклад:* Подання для менеджерів, для розробників, для операторів.
- **Architecture Viewpoint (Точка зору):** Абстракція, яка визначає типову структуру для певного класу View.
- **Stakeholder (Зацікавлена сторона):** Особа, команда або організація, мають інтерес до системи.
- **Concern (Проблема):** Набір вимог, цілей та зацікавлень, які зацікавлена сторона хоче розглядати.

UML (Unified Modeling Language)

Стандарт: ISO/IEC 19505

Графічна мова для візуалізації, специфікації та документування систем.

Основні діаграми для архітектури:

Class Diagram (Діаграма класів): Показує статичну структуру системи, включаючи класи, їхні атрибути, методи та взаємозв’язки. *Використання:* Розуміння структури системи, комунікація між розробниками.

Sequence Diagram (Діаграма послідовності): Показує взаємодію компонентів системи в часі, як вони обмінюються повідомленнями. *Використання:* Розуміння динаміки системи, виявлення проблем з синхронізацією.

Component Diagram (Діаграма компонентів): Показує архітектуру системи на рівні компонентів та їхніх залежностей.

Deployment Diagram (Діаграма розгортання): Показує, як компоненти розміщуються на фізичних вузлах (сервери, комп'ютери).

Приклади архітектурних рішень

Netflix (Microservices Architecture)

Контекст: Netflix є одним із найбільших сервісів потокового передавання відео з мільйонами користувачів по всьому світу.

Архітектурне рішення: Netflix використовує мікросервісну архітектуру, розділяючи систему на тисячі невеликих, незалежних сервісів.

Приклади сервісів:

- User Service — управління обліковими записами користувачів.
- Recommendation Service — рекомендація фільмів та серіалів.
- Streaming Service — потокова передача вмісту.
- Payment Service — обробка платежів.
- Analytics Service — аналіз поведінки користувачів.

Переваги цього підходу:

- **Надійність:** Якщо один сервіс (наприклад, рекомендацій) падає, основна функція (перегляд відео) продовжує працювати.
- **Швидкість доставки:** Різні команди можуть розробляти та розгортати сервіси незалежно.
- **Масштабованість:** Популярні сервіси (наприклад, Streaming) можна масштабувати незалежно від інших.
- **Гнучкість:** Різні сервіси можуть використовувати різні технології.

Linux Kernel (Monolithic with Modules)

Контекст: Linux Kernel — це основа операційної системи Linux, яка повинна бути дуже ефективною та надійною.

Архітектурне рішення: Linux використовує монолітну архітектуру, але з важливою модифікацією — підтримкою динамічних модулів.

Характеристики:

- Всі компоненти скомпільовані як один образ ядра.
- Динамічні модулі можуть бути завантажені/вивантажені без перезавантаження ядра.
- Драйвери та розширення можуть бути додані через модулі.

Переваги цього підходу:

- **Продуктивність:** Монолітна архітектура мінімізує накладні витрати.
- **Гнучкість:** Модулі дозволяють розширювати функціональність без перекомпіляції всього ядра.
- **Надійність:** Монолітна архітектура дозволяє оптимізувати критичні компоненти.

Висновки

- **Архітектура — це фундамент системи.** Помилки в архітектурі найважче виправляти. Важливо робити правильні архітектурні рішення на ранніх етапах.
- **Архітектура vs Дизайн:** Розрізняйте масштаб вашого рішення. Архітектурні рішення — це стратегічні, дизайнерські — це тактичні.
- **Не існує "найкращої" архітектури.** Вибір залежить від вимог проєкту, команди та контексту. Моноліт може бути прекрасним для стартапу, тоді як мікросервіси можуть бути необхідні для масштабного проєкту.

- **Принципи SOLID допомагають писати код, який легко читати, тестувати та розширювати.** Вони — не правила, а рекомендації. Деякі проєкти можуть не потребувати повного дотримання всіх принципів.
- **Патерни проєктування — це спільна мова розробників.** Знання патернів прискорює комунікацію та розв'язання типових проблем. Однак, не зловживайте патернами. Простіше часто краще.
- **Документування архітектури критично важливо.** Стандарти як ISO/IEC/IEEE 42010 та UML допомагають ефективно спілкуватися про архітектуру системи.

Контрольні запитання

1. У чому головна відмінність між архітектурою та дизайном ПЗ?
2. Назвіть переваги та недоліки монолітної архітектури.
3. Опишіть трирівневу (3-Tier) архітектуру. Які шари до неї входять?
4. Що таке мікросервіси? Яку головну проблему вони розв'язують?
5. Розшифруйте абревіатуру SOLID.
6. Поясніть принцип єдиної відповідальності (Single Responsibility Principle).
7. Що означає принцип відкритості/закритості (Open/Closed Principle)?
8. Наведіть приклад порушення принципу підстановки Лісков (Liskov Substitution Principle).
9. На які три групи поділяються патерни проєктування за класифікацією GoF?
10. Для чого використовується патерн Singleton? Які його недоліки?
11. У чому суть патерну Factory Method?
12. Яку проблему розв'язує патерн Adapter?

13. Опишіть принцип роботи патерну Observer. Де він часто використовується?

14. Що таке “технічний борг” (Technical Debt) і як він пов’язаний з поганою архітектурою?

15. Який стандарт регулює опис архітектури систем?

16. Для чого використовуються діаграми класів (Class Diagrams) в UML?

17. Що таке діаграма послідовності (Sequence Diagram)?

18. Чому Netflix використовує мікросервісну архітектуру?

19. Чи може монолітна архітектура бути ефективною для сучасних проєктів? Коли?

20. Як принцип інверсії залежностей (Dependency Inversion) допомагає зменшити зв’язність коду (Coupling)?

Лекція 5

Конструювання та реалізація ПЗ

Процес написання коду (Software Construction), стандарти кодування, рефакторинг та інструментальні засоби розробника.

Предмет, мета та завдання лекції

Предмет: Процес написання коду (Software Construction), стандарти кодування, рефакторинг та інструментальні засоби розробника.

Мета: Зрозуміти принципи написання чистого, зрозумілого та ефективного коду, навчитися управляти технічним боргом та використовувати сучасні інструменти розробки.

Завдання:

- Визначити місце конструювання в життєвому циклі ПЗ.
- Ознайомитися з поняттям якості коду та стандартами кодування (Code Style).
- Вивчити основи рефакторингу та розпізнавання «запахів коду» (Code Smells).
- Зрозуміти природу технічного боргу.
- Оглянути основні інструменти розробника (IDE, лінтери, системи контролю версій).

- Ознайомитися з можливостями та ризиками використання AI-інструментів у розробці.

Навчальні результати

Після завершення лекції студент зможе:

1. Пояснити роль конструювання в життєвому циклі розробки ПЗ.
2. Описати ключові характеристики якісного коду.
3. Пояснити важливість стандартів кодування та навести приклади.
4. Дати визначення рефакторингу та ідентифікувати «запахи коду».
5. Пояснити концепцію технічного боргу та його наслідки.
6. Назвати та описати основні інструменти, що використовують при конструюванні ПЗ.
7. Оцінювати переваги та недоліки використання AI-асистентів при написанні коду.

Конструювання ПЗ (Software Construction)

Згідно з SWEBOK v4 [5], **конструювання ПЗ** — це детальне створення робочого програмного забезпечення за допомогою комбінації кодування, верифікації, модульного тестування, інтеграційного тестування та налагодження.

Це етап, де «проєкти» перетворюються на реальний продукт. Область знань «Software Construction» у SWEBOK v4 структурована так:

Основи конструювання (Software Construction Fundamentals)

- **Мінімізація складності (Minimizing Complexity):** Основний драйвер конструювання. Код має бути простим для розуміння та підтримки.
- **Передбачення змін (Anticipating Change):** ПЗ постійно змінюється, тому код має бути гнучким.

- **Конструювання для верифікації (Constructing for Verification):** Написання коду таким чином, щоб його було легко тестувати (наприклад, Unit Testing, Code Review).
- **Повторне використання (Reuse):** Використання наявних бібліотек та фреймворків.
- **Стандарти конструювання (Standards in Construction):** Дотримання зовнішніх (індустріальних) та внутрішніх (корпоративних) стандартів.

Управління конструюванням (Managing Construction)

Включає планування, вимірювання прогресу та вибір моделей життєвого циклу (наприклад, Agile, Waterfall).

Практичні міркування (Practical Considerations)

Охоплює дизайн конструювання, мови програмування, кодування, тестування під час конструювання, якість конструювання та інтеграцію.

Технології конструювання (Software Construction Technologies)

Використання API, обробка помилок та винятків, паралелізм, middleware, розподілені системи та інструменти розробки.

Якість коду та стандарти

> «Будь-який дурень може написати код, зрозумілий комп'ютеру. Хороші програмісти пишуть код, зрозумілий людям.» — Martin Fowler [21].

Характеристики якісного коду

- **Читабельність (Readability):** Наскільки легко інший розробник може зрозуміти логіку.

- **Підтримуваність (Maintainability):** Наскільки легко вносити зміни.
- **Ефективність:** Оптимальне використання ресурсів.

Стандарти кодування (Code Style)

Набір правил, що регулюють оформлення коду (відступи, іменування змінних, розставлення дужок).

Приклади: PEP 8 (Python), Google Java Style Guide, Airbnb JavaScript Style Guide.

Інструменти:

- **Linters (Лінтери):** Аналізують код на наявність помилок та відхилень від стандарту (ESLint, Pylint).
- **Formatters (Форматери):** Автоматично виправляють форматування (Prettier, Black).

Рефакторинг (Refactoring)

Рефакторинг [22] — це процес зміни внутрішньої структури програми без зміни її зовнішньої поведінки з метою полегшення розуміння та здешевлення модифікації.

Коли проводити рефакторинг?

- **Правило трьох:** Якщо ви копіюєте код втретє — рефакторіть.
- Перед додаванням нової функції (щоб спростити її впровадження).
- Під час виправлення багів (щоб код став зрозумілішим).
- Під час Code Review.

«Запахи коду» (Code Smells)

Ознаки того, що код потребує рефакторингу:

1. **Дубльований код (Duplicated Code):** Один і той самий фрагмент трапляється в кількох місцях.
2. **Довгий метод (Long Method):** Функція робить занадто багато речей.
3. **Великий клас (Large Class):** Клас бере на себе забагато відповідальності (порушення SRP).
4. **Магічні числа (Magic Numbers):** Використання чисел без пояснення в коді (краще замінити на іменовані константи).

Техніки рефакторингу

- **Extract Method (Виділення методу):** Перенесення фрагмента коду в окрему функцію.
- **Rename Variable (Перейменування змінної):** Заміна незрозумілої назви на описову.
- **Move Method (Переміщення методу):** Перенесення методу в клас, де його використовують найчастіше.

Технічний борг (Technical Debt)

Метафора, що описує накопичені наслідки «швидких та брудних» рішень у коді [18]. Як і фінансовий борг, технічний борг вимагає сплати «відсотків» у вигляді додаткового часу на розробку нових функцій через складність наявного коду.

- **Свідомий борг:** «Ми випускаємо реліз зараз, щоб випередити конкурентів, але виправимо це наступного тижня».
- **Несвідомий борг:** Написання поганого коду через брак знань.

Інструменти розробника

1. **IDE (Integrated Development Environment):** Середовище, що об'єднує редактор, компілятор/інтерпретатор, налагоджувач (VS Code, IntelliJ

IDEA, Visual Studio).

2. **Version Control Systems (VCS):** Системи для відстеження змін у коді (Git). Дозволяють працювати в команді, повертатися до попередніх версій.

3. **Debuggers (Налагоджувачі):** Інструменти для покрокового виконання коду та пошуку помилок.

4. **Profilers (Профайлери):** Інструменти для аналізу продуктивності (використання пам'яті, CPU).

Сучасні підходи: AI-інструменти в конструюванні

Сучасне конструювання програмного забезпечення зазнає значних змін завдяки впровадженню інструментів на основі штучного інтелекту (AI-coding assistants). Ці інструменти функціонують як віртуальні «парні програмісти», що дозволяє підвищити продуктивність розробників.

Функціональні можливості

- **Генерація коду (Code Generation):** Створення фрагментів коду на основі опису природною мовою (наприклад, з коментарів).

- **Інтелектуальне доповнення (Intelligent Autocomplete):** Пропонування логічних продовжень коду, враховуючи контекст проєкту.

- **Пояснення та документування:** Автоматичне створення описів для складних алгоритмів та генерація документації.

- **Генерація тестів:** Створення модульних тестів для перевірки коректності коду.

- **Виявлення помилок:** Пошук потенційних вразливостей та помилок на етапі написання коду.

Популярні інструменти

- **GitHub Copilot:** Використовує модель OpenAI Codex, інтегрується в більшість популярних редакторів коду.
- **Tabnine:** Використовує моделі глибокого навчання, пропонує можливість локального запуску для збереження приватності.
- **Amazon Q Developer (раніше CodeWhisperer):** Орієнтований на хмарну розробку та інтеграцію з AWS.

Особливості використання

Використання AI-інструментів вимагає від інженера нових навичок та відповідальності:

- **Верифікація:** Код, згенерований AI, може містити помилки або «галюцинації», тому потребує обов'язкової перевірки людиною.
- **Безпека даних:** Необхідно бути обережним при роботі з конфіденційним кодом, щоб уникнути його передачі на сервери навчання моделей.
- **Юридичні аспекти:** Слід враховувати ліцензійні обмеження щодо коду, на якому навчався AI.

Ключові міжнародні стандарти

- **SWEBOK v4 (Guide to the Software Engineering Body of Knowledge):** Визначає «Software Construction» як одну з ключових областей знань (KA), що охоплює основи, управління, практичні аспекти та технології створення ПЗ.
- **ISO/IEC 25010:** Стандарт якості ПЗ, який визначає такі характеристики як Maintainability (Зручність супроводу), що є критичним для етапу конструювання.
- **Стандарти безпечного кодування (Secure Coding Standards):**

- **CERT Coding Standards:** Правила для C, C++, Java для запобігання вразливостям.
- **OWASP Secure Coding Practices:** Рекомендації для веб-розробки.

Приклади підходів до якості коду

Linux Kernel Coding Style

Лінус Торвальдс встановив жорсткі правила для коду ядра Linux (наприклад, відступи у 8 символів, обмеження довжини рядка). Це дозволяє тисячам розробників писати код, який виглядає як єдине ціле, і забезпечує високу надійність ядра.

> «Якщо вам потрібно більше 3 рівнів вкладеності, ви щось робите неправильно».

Google Engineering Practices (Code Review)

У Google кожен рядок коду повинен пройти перевірку (Code Review) іншим інженером перед тим, як потрапити в репозиторій [21]. Це не тільки знаходить помилки, але й поширює знання в команді та підтримує єдиний стиль.

Висновки

- Конструювання — це не просто набір тексту програми, а створення структури, яку зможуть підтримувати інші.
- Чистий код (Clean Code) економить гроші компанії в довгостроковій перспективі.
- Рефакторинг має бути постійною частиною процесу розробки, а не окремою подією “раз на рік”.
- Технічний борг — це інструмент, яким треба вміти управляти, а не просто уникати.

Контрольні запитання

1. Що таке конструювання програмного забезпечення (Software Construction)?
2. Чому читабельність коду важливіша за швидкість його написання?
3. Що таке Code Style і навіщо він потрібен? Наведіть приклад.
4. У чому різниця між лінтером та форматером?
5. Дайте визначення рефакторингу.
6. Що таке «Code Smells»? Наведіть 3 приклади.
7. Поясніть техніку рефакторингу “Extract Method”.
8. Що таке «магічні числа» в коді і чому їх слід уникати?
9. Що таке технічний борг? Які є його види?
10. Які наслідки накопичення великого технічного боргу?
11. Назвіть основні функції сучасних IDE.
12. Для чого використовують систему контролю версій (наприклад, Git) під час конструювання?
13. Що таке Code Review і яка його користь?
14. Який принцип лежить в основі правила «Boy Scout Rule» у програмуванні?
15. Що таке «спагетті-код»?
16. Як коментарі в коді можуть стати «запахом коду»?
17. Назвіть стандарт безпечного кодування.
18. Чому важливо дотримуватися єдиного стилю кодування в команді?
19. Що таке «мертвий код» (Dead Code)?

20. Як принцип KISS (Keep It Simple, Stupid) застосовують при конструюванні ПЗ?

21. Які основні функції виконують AI-асистенти при написанні коду?

22. Які ризики пов’язані з використанням AI-інструментів у розробці ПЗ?

Лекція 6

Тестування програмного забезпечення

Процеси забезпечення якості (QA), контролю якості (QC) та тестування програмного забезпечення.

Предмет, мета та завдання лекції

Предмет: Процеси забезпечення якості (QA), контролю якості (QC) та тестування програмного забезпечення.

Мета: Оволодіти основними поняттями тестування, зрозуміти різницю між верифікацією та валідацією, вивчити рівні та методи тестування.

Завдання:

- Розрізняти QA, QC та тестування.
- Зрозуміти концепцію «Верифікація vs Валідація».
- Вивчити піраміду тестування (Unit -> Integration -> System -> Acceptance).
- Розглянути методи «Білої скриньки» та «Чорної скриньки».
- Ознайомитися з підходом TDD (Test Driven Development).

Навчальні результати

Після завершення лекції студент зможе:

1. Пояснювати різницю між QA, QC та тестуванням.
2. Розрізняти процеси верифікації та валідації.
3. Описувати рівні піраміди тестування та їх призначення.
4. Вибирати відповідні методи тестування (біла/чорна скринька) для конкретних задач.
5. Розуміти принципи TDD та циклу Red-Green-Refactor.
6. Орієнтуватися в основних інструментах та стандартах тестування.

Основні поняття

QA vs QC vs Testing

- **Quality Assurance (QA):** Забезпечення якості. Це набір превентивних процесів, спрямованих на запобігання дефектам (налагодження процесів розробки).
- **Quality Control (QC):** Контроль якості. Це набір дій для виявлення дефектів у готовому продукті (перевірка відповідності вимогам).
- **Testing (Тестування):** Це процес виконання програми з метою виявлення помилок. Це частина QC.

Верифікація та Валідація (V&V)

- **Верифікація (Verification):** «Чи робимо ми продукт правильно?» Перевірка того, чи відповідає система специфікаціям та стандартам.
- **Валідація (Validation):** «Чи робимо ми правильний продукт?» Перевірка того, чи задовольняє система реальні потреби користувача.

  «Тестування може довести наявність помилок, але ніколи не може довести їх відсутність.» — Edsger W. Dijkstra.

Рівні тестування

Класична модель рівнів тестування часто зображується у вигляді піраміди:

1. **Модульне тестування (Unit Testing):**
    - Перевірка окремих найменших частин коду (функцій, методів, класів) ізольовано від інших.
    - Виконується розробниками.
    - Інструменти: JUnit, NUnit, PyTest.
2. **Інтеграційне тестування (Integration Testing):**
    - Перевірка взаємодії між кількома модулями або компонентами.
    - Мета: виявити помилки в інтерфейсах та взаємодії.
3. **Системне тестування (System Testing):**
    - Перевірка повної, інтегрованої системи на відповідність вимогам.
    - Включає функціональне та нефункціональне тестування.
4. **Приймальне тестування (Acceptance Testing / UAT):**
    - Перевірка системи замовником або кінцевим користувачем перед релізом.
    - Мета: вирішити, чи готова система до експлуатації.

Методи тестування

За доступом до коду

- **Чорна скринька (Black Box):** Тестувальник не бачить внутрішнього коду. Перевірка базується лише на вхідних даних та очікуваному результаті (поведінка системи).
- **Біла скринька (White Box):** Тестувальник знає внутрішню структуру коду. Перевірка логічних шляхів, умов, циклів.

- **Сіра скринька (Grey Box):** Комбінація методів (наприклад, тестування API з доступом до бази даних).

За метою

- **Функціональне:** Перевірка функцій системи (що вона робить).
- **Нефункціональне:** Перевірка атрибутів якості (продуктивність, навантаження, безпека, зручність).
- **Регресійне (Regression):** Повторне тестування після внесення змін, щоб переконатися, що старий функціонал не зламався.
- **Димове (Smoke):** Швидка перевірка критичного функціоналу («чи вмикається система?»).

Автоматизація та TDD

Ручне vs Автоматизоване

- **Ручне:** Людина виконує тест-кейси. Ефективно для дослідницького тестування (Exploratory Testing) та UX.
- **Автоматизоване:** Скрипти виконують тести. Ефективно для регресійного тестування та навантажувального тестування.

TDD (Test Driven Development)

Розробка через тестування — це підхід, де тести пишуться *перед* кодом. **Цикл Red-Green-Refactor:**

1. **Red:** Написати тест, який падає (бо функціоналу ще немає).
2. **Green:** Написати мінімальний код, щоб тест пройшов.
3. **Refactor:** Покращити код, зберігаючи тест «зеленим».

Ключові міжнародні стандарти

- **ISO/IEC/IEEE 29119** — Серія міжнародних стандартів для тестування програмного забезпечення. Вона визначає словник, процеси, документацію та техніки тестування.
- **ISTQB (International Software Testing Qualifications Board)** — Хоча це не стандарт ISO, глосарій та сертифікація ISTQB є де-факто стандартом у індустрії тестування. Визначає термінологію та найкращі практики.

Приклади та інструменти

1. **Netflix Chaos Monkey (Chaos Engineering):** Унікальний підхід до тестування надійності. Інструмент спеціально вимикає сервери у продакшн-середовищі, щоб перевірити, чи зможе система автоматично відновитися. Це «тестування на стійкість» в екстремальних умовах.
2. **Selenium WebDriver:** Найпопулярніший інструмент для автоматизації веб-браузерів. Дозволяє писати скрипти на різних мовах (Java, Python, C#), які імітують дії користувача (кліки, введення тексту) для перевірки веб-додатків.
3. **Linter (Лінтер):** Інструмент статичного аналізу коду, який перевіряє вихідний текст програми на наявність синтаксичних помилок, багів, стилістичних порушень та підозрілих конструкцій без її запуску. Він діє як автоматизований контролер якості.

   Використання лінтерів дозволяє виявляти та виправляти помилки на найбільш ранніх етапах. Крім того, лінтери забезпечують дотримання єдиного стилю кодування (Code Style) у команді.

Висновки

- Тестування — це невід'ємна частина SDLC, яка має починатися якомога раніше.

- Важливо розуміти різницю між перевіркою відповідності специфікації (Верифікація) та потребам користувача (Валідація).
- Автоматизація є ключем до швидких релізів (CI/CD), але не замінює ручне тестування повністю.

Контрольні запитання

1. У чому різниця між QA (Quality Assurance) та QC (Quality Control)?
2. Поясніть різницю між верифікацією та валідацією.
3. Назвіть 4 рівні піраміди тестування.
4. Що таке модульне тестування (Unit Testing) і хто його зазвичай виконує?
5. У чому суть методу «Чорної скриньки»?
6. Що таке регресійне тестування і коли воно проводиться?
7. Опишіть цикл TDD (Red-Green-Refactor).
8. Що таке «Димове тестування» (Smoke Testing)?
9. Назвіть перевагу автоматизованого тестування над ручним.
10. Коли краще використовувати ручне тестування?
11. Що таке баг-репорт (Bug Report) і які його обов'язкові поля?
12. Що таке покриття коду (Code Coverage)?
13. Яка мета навантажувального тестування (Load Testing)?
14. Що таке позитивне та негативне тестування?
15. Поясніть термін «Mock-об'єкт» у контексті модульного тестування.
16. Що таке дослідницьке тестування (Exploratory Testing)?
17. Який стандарт регулює процеси тестування ПЗ?

18. Що робить інструмент Selenium?

19. У чому ідея Chaos Engineering (на прикладі Netflix)?

20. Чому неможливо протестувати програму на 100% (вичерпне тестування)?

Лекція 7

Управління проєктами в програмній інженерії

Основи управління проєктами (Project Management) в контексті розробки програмного забезпечення, методи оцінки задач та управління ризиками.

Предмет, мета та завдання лекції

Предмет: Основи управління проєктами (Project Management) в контексті розробки програмного забезпечення, методи оцінки задач та управління ризиками.

Мета: Зрозуміти роль менеджера проєкту, навчитися оцінювати трудомісткість розробки, ідентифікувати ризики та ефективно організовувати командну роботу.

Завдання:

- Ознайомитися з "Залізним трикутником" управління проєктами.
- Вивчити методи оцінки задач (Story Points, Planning Poker).
- Розглянути процес управління ризиками.
- Оглянути популярні інструменти (Jira, Trello).
- Зрозуміти важливість комунікації та динаміки команди.

Основи управління проєктами

Управління проєктом (Project Management) — це застосування знань, навичок, інструментів і технік до проєктної діяльності для задоволення вимог проєкту [23].

Проєктний трикутник (Iron Triangle)

Баланс між трьома обмеженнями:

1. **Scope (Обсяг):** Що саме ми робимо? (Функціонал).
2. **Time (Час):** Скільки часу ми маємо? (Дедлайни).
3. **Cost (Вартість):** Який у нас бюджет? (Ресурси, люди).

   Швидко, якісно, дешево — оберіть будь-які два.

Зміна одного параметру неминуче впливає на інші. Наприклад, якщо замовник хоче додати нові функції (Scope) без зміни дедлайну (Time), доведеться збільшити бюджет (Cost) або пожертвувати якістю.

Проєкт vs Продукт

Важливо розрізняти ці два поняття, оскільки підходи до управління ними суттєво відрізняються.

Визначення

- **Проєкт (Project):** Тимчасова діяльність, спрямована на створення унікального продукту, послуги або результату. Має чіткий початок і кінець.
  - *Фокус:* Виконання плану, дотримання термінів та бюджету (Output).
- **Продукт (Product):** Товар або послуга, що задовольняє потреби користувачів. Має життєвий цикл (від ідеї до виведення з експлуатації), який може включати багато проєктів.

- *Фокус:* Цінність для користувача, прибутковість, розвиток (Outcome).

Управління проєктом vs Управління продуктом

- **Project Management** — Як зробити це правильно? (Процес). Відповідає за те, щоб задача була виконана вчасно і в межах бюджету.
- **Product Management** — Що саме ми робимо і навіщо? (Бачення). Відповідає за успіх продукту на ринку.

Ролі

- **Project Manager (PM):** Планує спринти, слідкує за дедлайнами, усуває перешкоди для команди, звітує перед стейкхолдерами про прогрес.
- **Product Manager / Product Owner (PO):** Визначає пріоритети (Backlog), спілкується з користувачами, формує стратегію розвитку (Roadmap), приймає роботу команди.

Оцінка трудомісткості (Estimation)

Оцінка в розробці ПЗ є складною через невизначеність [24].

Одиниці виміру

- **Людино-години (Man-hours):** Абсолютний час. Часто неточний, бо залежить від кваліфікації розробника.
- **Story Points (SP):** Відносна одиниця складності. Враховує обсяг роботи, складність та ризики. Дозволяє порівнювати задачі між собою (ця задача вдвічі складніша за ту).

Техніки оцінки

1. **Експертна оцінка:** Досвідчений розробник дає прогноз на основі минулого досвіду.

2. **Planning Poker (Покер планування):**
    - Команда збирається разом.
    - Кожен учасник має карти з числами (часто послідовність Фібоначчі: 1, 2, 3, 5, 8, 13. . .).
    - Озвучується задача. Всі одночасно показують карту з оцінкою.
    - Якщо оцінки різняться, проводиться обговорення, поки не буде досягнуто консенсусу.
    - *Перевага:* Усуває когнітивне викривлення (вплив авторитету лідера).

Управління ризиками (Risk Management)

Ризик — це невизначена подія, яка в разі настання має позитивний або негативний вплив на цілі проєкту [25].

Процес управління ризиками

1. **Ідентифікація:** Що може піти не так? (Втрата ключового розробника, зміна вимог, технічні проблеми).
2. **Аналіз:** Оцінка ймовірності (Probability) та впливу (Impact).
3. **Планування реагування:**
    - *Avoid (Уникнення):* Змінити план, щоб виключити ризик.
    - *Mitigate (Зменшення):* Знизити ймовірність або вплив (наприклад, робити бекапи).
    - *Transfer (Передача):* Перекласти відповідальність (страхування, аутсорсинг).
    - *Accept (Прийняття):* Нічого не робити, але мати план "Б".

Інструменти управління задачами

Для відстеження прогресу використовуються спеціалізовані системи (Issue Trackers):

1. **Jira:** Найпотужніший інструмент для Agile-команд. Підтримує Scrum, Kanban, складні звіти. Стандарт в ентерпрайзі.
2. **Trello:** Простий інструмент на основі Kanban-дошок. Ідеальний для невеликих команд та стартапів.
3. **GitHub Projects:** Інтегрований в GitHub інструмент управління, що дозволяє пов'язувати задачі (Issues) безпосередньо з кодом (Pull Requests).

Командна робота та комунікація

Успіх проєкту залежить не тільки від коду, але й від людей.

- **Soft Skills:** Вміння спілкуватися, вирішувати конфлікти, давати зворотний зв'язок.
- **Bus Factor:** Кількість учасників команди, яких може "збити автобус", перш ніж проєкт зупиниться. Якщо Bus Factor = 1, це критичний ризик (знання зосереджені в одній голові).
- **Daily Stand-up:** Коротка зустріч (15 хв) щоранку, де кожен відповідає на 3 питання:
    1. Що я зробив вчора?
    2. Що я буду робити сьогодні?
    3. Які є перешкоди (Blockers)?

Ключові міжнародні стандарти

- **PMBOK (Project Management Body of Knowledge):** Фундаментальний стандарт від PMI (Project Management Institute). Описує процеси управління проєктами, області знань (інтеграція, зміст, розклад, вартість, якість, ресурси, комунікації, ризики, закупівлі, стейкхолдери).
- **ISO 21500:** "Guidance on project management" [26]. Міжнародний стандарт, що надає загальні рекомендації щодо управління проєктами.

Приклади організаційних моделей

1. **Spotify Model (Squads, Tribes, Chapters, Guilds):** Spotify популяризував модель масштабування Agile, де основною одиницею є **Squad** (невелика крос-функціональна команда, схожа на Scrum-команду). Squad-и об'єднуються в **Tribes** (племена) за бізнес-напрямком. Для обміну знаннями існують **Chapters** (люди однієї спеціалізації в межах Tribe) та **Guilds** (спільноти за інтересами по всій компанії).

2. **Amazon "Two-Pizza Teams":** Правило Джеффа Безоса: команда має бути настільки маленкою, щоб її можна було нагодувати двома піцами (6-8 людей). Це зменшує комунікаційні витрати та прискорює прийняття рішень.

Висновки

- Управління проєктом — це мистецтво балансування між обсягом, часом та вартістю.
- Оцінка в Story Points є більш точною для довгострокового планування, ніж оцінка в годинах.
- Ризиками потрібно управляти проактивно, а не реагувати на проблеми, коли вони вже сталися.
- Ефективна комунікація в команді важливіша за інструменти.

Контрольні запитання

1. Що таке "Залізний трикутник" (Iron Triangle) управління проєктами?
2. Як зміна Scope (обсягу) впливає на Time (час) та Cost (вартість)?
3. Що таке Story Point і чим він відрізняється від людино-години?
4. Опишіть процес Planning Poker. Яку проблему він вирішує?
5. Що таке ризик у проєкті?

6. Назвіть 4 стратегії реагування на ризики.
7. Що таке Bus Factor і чому він має бути більше 1?
8. Які три питання обговорюються на Daily Stand-up?
9. Для чого використовуються діаграми Ганта (Gantt Charts)?
10. У чому різниця між Jira та Trello?
11. Що таке MVP (Minimum Viable Product) в контексті управління проєктом?
12. Що таке Velocity (швидкість) команди в Scrum?
13. Як розрахувати Velocity?
14. Що таке "Залізний трикутник" (Iron Triangle) управління проєктами?
15. Що таке MVP (Minimum Viable Product) в контексті управління проєктом?
16. Що таке "Burn-down Chart"?
17. Поясніть правило "Two-Pizza Teams".
18. Що таке Stakeholder Management (управління стейкхолдерами)?
19. Яка роль Project Manager (PM) на відміну від Product Owner (PO)?
20. Що таке "Scope Creep" і як менеджер проєкту має з ним боротися?
21. Що таке ретроспектива (Retrospective) і навіщо вона потрібна?
22. Як стандарт PMBOK класифікує області знань управління проєктами?

Лекція 8

Професійні стандарти та етика

Професійні стандарти, етичні норми, правові аспекти (ліцензування) та соціальна відповідальність у галузі програмної інженерії.

Предмет, мета та завдання лекції

Предмет: Професійні стандарти, етичні норми, правові аспекти (ліцензування) та соціальна відповідальність у галузі програмної інженерії.

Мета: Сформувати розуміння того, що програмна інженерія — це не лише написання коду, а й відповідальність перед суспільством, клієнтами та професією.

Завдання:

- Ознайомитися з Кодексом етики інженера-програміста (ACM/IEEE).
- Розглянути види ліцензій на програмне забезпечення (Open Source vs Proprietary).
- Зрозуміти поняття інтелектуальної власності.
- Обговорити соціальну відповідальність та наслідки неякісного ПЗ.
- Оглянути стандарти конфіденційності даних (GDPR).

Професіоналізм в інженерії

Програмна інженерія, як і медицина чи юриспруденція, є професією, що вимагає дотримання певних стандартів поведінки. Професіонал відрізняється від аматора не лише рівнем навичок, а й відповідальністю за результати своєї роботи.

Кодекс етики ACM/IEEE

Найавторитетнішим документом у цій сфері є «Software Engineering Code of Ethics and Professional Practice», розроблений спільно ACM (Association for Computing Machinery) та IEEE Computer Society [27]. Він містить 8 принципів:

1. **Public (Суспільство):** Інженери повинні діяти в інтересах суспільства.
2. **Client and Employer (Клієнт і Роботодавець):** Діяти в інтересах клієнта та роботодавця, якщо це не суперечить інтересам суспільства.
3. **Product (Продукт):** Забезпечувати найвищу можливу якість продукту.
4. **Judgment (Судження):** Зберігати цілісність і незалежність своїх професійних суджень.
5. **Management (Менеджмент):** Менеджери повинні підтримувати етичні підходи до розробки.
6. **Profession (Професія):** Підвищувати репутацію професії.
7. **Colleagues (Колеги):** Бути чесними та підтримувати колег.
8. **Self (Саморозвиток):** Постійно навчатися та вдосконалювати свої навички.

Інтелектуальна власність та ліцензування

Програмне забезпечення є об’єктом інтелектуальної власності.

Типи захисту

- **Авторське право (Copyright):** Захищає конкретний код (текст програми). Виникає автоматично при створенні.
- **Патент (Patent):** Захищає ідею або алгоритм (якщо це дозволено законодавством країни).
- **Комерційна таємниця (Trade Secret):** Захист інформації шляхом її нерозголошення (наприклад, алгоритм пошуку Google).

Ліцензії на ПЗ

Ліцензія — це юридичний документ, що визначає права користувача щодо ПЗ.

1. **Proprietary (Власницьке ПЗ):** Код закритий. Користувач купує право на використання, але не володіє програмою (Microsoft Windows, Adobe Photoshop).
2. **Open Source (Відкрите ПЗ):** Код доступний для перегляду та модифікації [28].
    - **Permissive (Дозвільні):** Дозволяють використовувати код у закритих проєктах (MIT, Apache 2.0, BSD).
    - **Copyleft (Вірусні):** Вимагають, щоб похідні роботи також були відкритими під тією ж ліцензією (GPL v3).

Соціальна відповідальність та безпека

Програмне забезпечення керує літаками, медичним обладнанням, банківськими системами. Помилки в коді можуть призвести до фінансових втрат, травм або навіть загибелі людей.

Safety-Critical Systems

Системи, відмова яких може призвести до катастрофічних наслідків.

- *Приклад:* Therac-25 (апарат променевої терапії). Через помилку в ПЗ (Race Condition) пацієнти отримували смертельні дози радіації. Це класичний приклад того, чому тестування та етика важливі.

Algorithmic Bias (Алгоритмічна упередженість)

Штучний інтелект може успадковувати упередження з даних, на яких він навчався.

- *Приклад:* Система розпізнавання облич, яка погано працює з певними расами, або алгоритм найму, що дискримінує жінок.

Конфіденційність даних (Data Privacy)

У цифрову епоху дані — це «нова нафта». Захист персональних даних є критично важливим.

- **GDPR (General Data Protection Regulation):** Регламент ЄС. Основні права користувачів:
  - Право на забуття (видалення даних).
  - Право на доступ до даних.
  - Privacy by Design (приватність на етапі проєктування).
- **CCPA (California Consumer Privacy Act):** Аналог GDPR у США (Каліфорнія).

Етика та регулювання штучного інтелекту

З розвитком технологій штучного інтелекту (AI) та машинного навчання (ML), професійні стандарти та законодавство адаптуються до нових викликів.

SWEBOK v4: Штучний інтелект та Етика

Четверта версія SWEBOK (Guide to the Software Engineering Body of Knowledge) визнає AI не як «магію», а як інженерну дисципліну [5].

- **AI Engineering:** Розробка систем з елементами AI вимагає специфічних підходів до управління даними (Data Quality), тестування ймовірнісних систем та моніторингу моделей.
- **Професійна практика:** Інженери несуть відповідальність за етичні наслідки алгоритмів, зокрема за упередженість (bias), прозорість рішень та вплив на права людини.

Законодавство ЄС: EU AI Act

Перший у світі комплексний закон, що регулює використання ШІ, базується на ризик-орієнтованому підході [29]:

1. **Неприйнятний ризик (Заборонено):** Системи соціального рейтингу, біометрична ідентифікація в реальному часі у громадських місцях (з винятками), маніпулятивні системи.
2. **Високий ризик (High Risk):** Медицина, транспорт, рекрутинг, правосуддя. Вимагають суворої оцінки відповідності, якості даних та людського нагляду.
3. **Обмежений ризик:** Чат-боти, діпфейки. Вимагають прозорості (маркування контенту).

Законодавство США: Executive Order on AI

Указ Президента США про безпечний та надійний ШІ (2023) фокусується на:

- **Безпека:** Розробники потужних моделей зобов'язані ділитися результатами тестів на безпеку з урядом.
- **Стандарти NIST:** Розробка стандартів для тестування AI на вразливості.
- **Маркування:** Впровадження стандартів для водяних знаків на згенерованому контенті.

Ключові міжнародні стандарти

- **ISO/IEC 27001** — Стандарт управління інформаційною безпекою (ISMS) [30]. Визначає вимоги до захисту інформаційних активів.
- **ISO/IEC 12207** — Процеси життєвого циклу ПЗ (згадувався раніше, але є базовим стандартом професії).
- **IEEE 730** — Стандарт забезпечення якості програмного забезпечення (Software Quality Assurance).

Приклади етичних дилем

1. **Volkswagen «Dieselgate»:** Інженери Volkswagen написали код, який визначав, коли автомобіль проходить тест на викиди, і вмикав режим очищення. В реальних умовах викиди перевищували норму в 40 разів.
    - *Етична проблема:* Інженери свідомо створили обманне ПЗ на вимогу менеджменту, порушивши принцип «Public» (шкода екології та здоров'ю).
2. **Cambridge Analytica:** Використання персональних даних мільйонів користувачів Facebook без їхньої чіткої згоди для політичної маніпуляції.
    - *Етична проблема:* Порушення конфіденційності та маніпуляція суспільною думкою.

Висновки

- Бути інженером-програмістом — це більше, ніж писати код. Це означає нести відповідальність за наслідки своєї роботи.
- Знання ліцензій є обов'язковим для уникнення юридичних проблем, особливо при використанні Open Source бібліотек.
- Етичні принципи допомагають приймати складні рішення, коли інтереси бізнесу конфліктують з інтересами суспільства.

Контрольні запитання

1. Чому програмну інженерію називають професією?
2. Назвіть 8 принципів Кодексу етики ACM/IEEE.
3. Який принцип Кодексу етики є найважливішим (пріоритетним)?
4. Що таке інтелектуальна власність?
5. У чому різниця між авторським правом (Copyright) та патентом?
6. Що таке ліцензія на програмне забезпечення?
7. Поясніть різницю між пропрієтарним та вільним (Open Source) ПЗ.
8. Що таке ліцензія MIT? Які права вона надає?
9. Що таке Copyleft ліцензія (наприклад, GPL)?
10. Чи можна використовувати код під ліцензією GPL у закритому комерційному продукті?
11. Що сталося з апаратом Therac-25 і чому це важливо для інженерів?
12. Що таке «алгоритмічна упередженість» (Algorithmic Bias)?
13. Що таке GDPR? Кого він захищає?
14. Що означає принцип «Privacy by Design»?
15. Опишіть суть скандалу Volkswagen «Dieselgate» з точки зору етики розробника.
16. Що таке NDA (Non-Disclosure Agreement)?
17. Який стандарт ISO регулює інформаційну безпеку?
18. Що таке «Whistleblowing» (викривання) і чи є це етичним вчинком?
19. Як ліцензія Creative Commons відрізняється від ліцензій на ПЗ?
20. Чому важливо вказувати авторство при використанні чужого коду (Attribution)?

Лекція 9

Кар'єра в програмній інженерії та майбутнє галузі

Кар'єрний розвиток інженера-програміста, необхідні навички (Hard & Soft Skills), концепція безперервного навчання та майбутні тренди галузі.

Предмет, мета та завдання лекції

Предмет: Кар'єрний розвиток інженера-програміста, необхідні навички (Hard & Soft Skills), концепція безперервного навчання та майбутні тренди галузі.

Мета: Сформувати бачення професійного шляху, зрозуміти вимоги ринку праці та підготуватися до викликів майбутнього, пов'язаних зі штучним інтелектом.

Завдання:

- Розглянути рівні кваліфікації (Junior, Middle, Senior).
- Зрозуміти різницю між Hard та Soft Skills.
- Ознайомитися з концепцією Life Long Learning.
- Проаналізувати вплив AI (LLM) на розробку ПЗ.
- Оглянути тренди Low-code/No-code.

Кар'єрний шлях (Career Path)

Розвиток розробника зазвичай проходить через кілька стандартних етапів:

1. **Junior (Початківець):**
    - Виконує прості задачі під наглядом ментора.
    - Фокус на вивченні технологій та інструментів.
    - Потребує чітких інструкцій.
2. **Middle (Спеціаліст):**
    - Самостійно виконує більшість задач.
    - Розуміє бізнес-логіку проєкту.
    - Може менторити джуніорів.
3. **Senior (Експерт):**
    - Приймає архітектурні рішення.
    - Розв'язує складні проблеми, які не мають очевидного розв'язку.
    - Відповідає за якість коду команди.
4. **Staff Engineer (Провідний інженер):**
    - Роль рівня «Senior+», яка передбачає технічний вплив на рівні кількох команд або всієї організації.
    - Займається складними архітектурними питаннями та технічною стратегією.
    - На відміну від менеджера, не керує людьми безпосередньо (Individual Contributor), але є технічним лідером і ментором.
5. **Lead / Architect / Manager:**
    - *Tech Lead:* Технічний лідер команди.
    - *Architect:* Проєктує глобальну структуру системи.
    - *Engineering Manager:* Керує людьми та процесами.

Навички: Hard vs Soft

Успіх у кар'єрі залежить від балансу технічних та «м'яких» навичок.[21]

Hard Skills (Тверді навички)

Це професійні технічні знання, які можна виміряти.

- Мови програмування (Java, Python, C++).
- Фреймворки (React, Spring, .NET).
- Бази даних (SQL, NoSQL).
- Алгоритми та структури даних.

Soft Skills (М'які навички)

Це особистісні якості, що впливають на взаємодію з людьми.

- **Комунікація:** Вміння чітко висловлювати думки (усно та письмово).
- **Робота в команді:** Вміння слухати, домовлятися, розв'язувати конфлікти.
- **Тайм-менеджмент:** Вміння планувати свій час та дотримуватися дедлайнів.
- **Адаптивність:** Готовність до змін.

  «Вас наймають за Hard Skills, а звільняють (або підвищують) за Soft Skills.»

Навчання протягом життя (Life Long Learning)

IT-галузь змінюється надзвичайно швидко.[31] Те, що було актуальним 5 років тому, сьогодні може бути застарілим.

- **T-shaped спеціаліст:** Глибокі знання в одній галузі (вертикальна риса) та широкі знання в суміжних галузях (горизонтальна риса).

- **Джерела знань:** Офіційна документація, онлайн-курси (Coursera, Udemy), конференції, Open Source проєкти, технічні блоги.

Майбутнє галузі: Вплив AI

Штучний інтелект, зокрема великі мовні моделі (LLM) на кшталт GPT-4, Copilot, Claude, кардинально змінюють процес розробки.

Як AI допомагає розробникам?

1. **Генерація коду:** Написання шаблонного коду (Boilerplate), тестів, документації.
2. **Пояснення коду:** Швидке розуміння чужого або заплутаного коду.
3. **Пошук помилок:** AI може діяти як «другий пілот» (Pair Programmer), знаходячи баги.

Чи замінить AI програмістів?

Найближчим часом — ні. AI замінить *кодерів* (тих, хто просто пише код за інструкцією), але не *інженерів* (тих, хто розв'язує проблеми).

- **Нова навичка:** Prompt Engineering (вміння формулювати запити до AI).

Low-code та No-code платформи

Тенденція до спрощення розробки для не-програмістів.

- **No-code:** Дозволяє створювати прості застосунки (лендінги, форми) візуально, без написання жодного рядка коду (Wix, Bubble).
- **Low-code:** Прискорює розробку для професіоналів, автоматизуючи рутину, але дозволяє додавати кастомний код (OutSystems, Mendix).

Ключові міжнародні стандарти

- **SFIA (Skills Framework for the Information Age):** Модель опису навичок та компетенцій для фахівців у сфері IT. Використовується компаніями для оцінки персоналу та планування кар'єри.
- **e-CF (European e-Competence Framework):** Європейський стандарт компетенцій для IT-фахівців (EN 16234).

Приклади кар'єрних треків

1. **Individual Contributor (IC):** Шлях експерта, який хоче писати код, а не керувати людьми.
    - *Junior → Middle → Senior → Staff Engineer → Principal Engineer.*
2. **Management Track:** Шлях керівника, який фокусується на людях та бізнесі.
    - *Senior → Team Lead → Engineering Manager → CTO (Chief Technology Officer).*

Побудова професійного бренду

Для успішного старту та розвитку кар'єри важливо не лише мати навички, а й вміти їх продемонструвати.

Професійний профіль

1. **GitHub:** Ваше портфоліо. Роботодавці дивляться на:
    - Якість коду в пет-проєктах.
    - Наявність документації (README.md).
    - Активність (історія комітів).
2. **LinkedIn:** Ваша візитна картка. Дозволяє рекрутерам знайти вас. Важливо заповнити розділи про досвід, навички та освіту.

Пошук роботи

- **Аналіз ринку:** Регулярний перегляд вакансій (DOU, Djinni, LinkedIn) допомагає розуміти актуальні вимоги до стеку технологій.
- **Резюме (CV):** Має бути релевантним до позиції. Для Junior-розробників важливо вказати навчальні проєкти, курси та посилання на GitHub.

Психологічні аспекти роботи

Робота в IT пов'язана з високим когнітивним навантаженням та стресом.[25]

Синдром самозванця (Impostor Syndrome)

Психологічне явище, коли людина не здатна приписати свої досягнення власним якостям, здібностям і зусиллям. Часто виникає через те, що в IT обсяг знань є безмежним.

- *Як боротися:* Визнати, що знати все неможливо; фіксувати свої досягнення; просити конструктивний зворотний зв'язок.

Вигорання (Burnout)

Стан емоційного, фізичного і розумового виснаження, викликаний надмірним і тривалим стресом.

- *Профілактика:* Work-life balance (баланс між роботою та життям), хобі, фізична активність, регулярні відпустки.

Висновки

- Професія інженера-програміста вимагає постійного навчання. Зупинка в розвитку означає деградацію.
- Soft Skills є критично важливими для кар'єрного зростання.

- Штучний інтелект — це інструмент, який посилює продуктивність інженера, а не ворог.
- Майбутнє за тими, хто вміє поєднувати глибокі технічні знання з розумінням бізнесу та вмінням використовувати AI.

Контрольні запитання

1. Назвіть основні рівні кваліфікації розробника (грейди).
2. Чим Senior розробник відрізняється від Middle, окрім кількості років досвіду?
3. Що таке Hard Skills? Наведіть 3 приклади.
4. Що таке Soft Skills? Наведіть 3 приклади.
5. Чому комунікація вважається найважливішою «м'якою» навичкою?
6. Поясніть концепцію «T-shaped спеціаліст».
7. Що таке Life Long Learning і чому це важливо в IT?
8. Як штучний інтелект (AI) змінює роботу програміста?
9. Що таке GitHub Copilot?
10. Чи замінить AI професію програміста в найближчому майбутньому? Обґрунтуйте.
11. Що таке Low-code платформа?
12. У чому різниця між Low-code та No-code?
13. Хто такий Tech Lead?
14. Що таке «синдром самозванця» (Imposter Syndrome) і як з ним боротися?
15. Яка роль CTO (Chief Technology Officer) в компанії?
16. Що таке SFIA?

17. Чому англійська мова вважається обов'язковою для IT-фахівця?
18. Що таке «вигорання» (Burnout) і як його уникнути?
19. Які переваги та недоліки віддаленої роботи (Remote Work)?
20. Назвіть одну технологію, яка, на вашу думку, буде домінувати в найближчі 5 років.

Словник ключової термінології

Acceptance Criteria (Критерії прийняття) Конкретні умови, які мають бути виконані, щоб User Story вважалася завершеною, часто у форматі Gherkin (Given-When-Then).

Acceptance Testing (Приймальне тестування) Рівень тестування, що проводиться замовником або кінцевим користувачем для вирішення, чи готова система до експлуатації.

Agile Гнучка методологія розробки, що базується на ітеративному підході, співпраці з клієнтом та адаптивності до змін.

Agile Manifesto Декларація 4 ключових цінностей гнучкої розробки: люди та взаємодія, працюючий продукт, співпраця з клієнтом та готовність до змін.

Algorithmic Bias (Алгоритмічна упередженість) Систематичні та повторювані помилки в комп'ютерній системі, що створюють несправедливі результати, наприклад, привілеювання однієї групи користувачів над іншою.

Behavioral Patterns (Поведінкові патерни) Патерни, що відповідають за взаємодію та розподіл обов'язків між об'єктами (наприклад, Observer, Strategy).

Black Box Testing (Тестування чорної скриньки) Метод тестування, при якому тестувальник не має доступу до внутрішнього коду і перевіряє систему на основі вхідних даних та очікуваних результатів.

CI/CD Continuous Integration / Continuous Delivery — автоматизація збірки, тестування та розгортання ПЗ.

Clean Code (Чистий код) Код, що написаний таким чином, щоб бути легким для розуміння, підтримання та модифікації.

Code Readability (Читабельність коду) Міра того, наскільки легко інший розробник може розуміти логіку та призначення коду.

Code Review (Перевірка коду) Процес, при якому інший розробник аналізує написаний код перед його включенням у репозиторій.

Code Smell (Запах коду) Поверхневий показник можливої глибшої проблеми в коді, що вказує на необхідність рефакторингу.

Communication Skills (Навички комунікації) Вміння чітко висловлювати думки усно та письмово, слухати та розуміти позицію інших.

Computer Science Наука, що фокусується на теорії та основах обчислень (алгоритми, структури даних, теорія складності, AI).

Copyleft License (Копілефт ліцензія) Ліцензія, яка вимагає, щоб усі похідні роботи розповсюджувалися на тих самих умовах (наприклад, GPL).

Copyright (Авторське право) Форма захисту інтелектуальної власності, що надає авторам виключне право на використання їхніх творів (коду).

Creational Patterns (Твірні патерни) Патерни, що відповідають за механізми створення об'єктів (наприклад, Singleton, Factory Method).

Design Patterns (Патерни проєктування) Типові розв'язання поширених проблем при проєктуванні програмного забезпечення.

DevOps Культура та набір практик, що об'єднують розробку (Dev) та експлуатацію (Ops) для скорочення циклу розробки.

Elicitation (Виявлення вимог) Перший етап процесу інженерії вимог — спілкування зі стейкхолдерами для розуміння їхніх потреб, проблем та очікувань.

Estimation (Оцінка трудомісткості) Процес прогнозування часу та зусиль, необхідних для виконання певної задачі або проєкту.

Functional Requirements (Функціональні вимоги) Вимоги, що описують, ЩО система повинна робити — її поведінку, функції та операції.

GDPR General Data Protection Regulation — регламент ЄС щодо захисту персональних даних усіх осіб у межах Європейського Союзу.

GitHub Copilot AI-асистент для написання коду, розроблений OpenAI та GitHub, що використовує модель Codex для генерації коду.

GitHub Profile (GitHub профіль) Портфоліо розробника на платформі GitHub, що демонструє якість коду, активність та досвід через публічні репозиторії та контрибуції.

Integration Testing (Інтеграційне тестування) Рівень тестування, що перевіряє взаємодію між кількома модулями або компонентами системи.

Intellectual Property (Інтелектуальна власність) Результати інтелектуальної, творчої діяльності, права на які захищаються законом (авторське право, патенти).

Iron Triangle (Залізний трикутник) Концептуальна модель, що ілюструє баланс між трьома основними обмеженнями проєкту: обсягом (Scope), часом (Time) та вартістю (Cost).

Junior Developer (Джуніор-розробник) Початківець у розробці, що виконує простих завдань під наглядом ментора, фокусується на вивленні технологій та потребує чітких інструкцій.

Kanban Метод управління потоком роботи, що походить від Toyota, базується на візуалізації та обмеженні незавершеної роботи.

KISS Principle (Принцип KISS) Keep It Simple, Stupid — принцип, що рекомендує тримати код простим та зрозумілим.

Layered Architecture (Багатошарова архітектура) Архітектурний стиль, де система розділена на логічні шари (наприклад, представлення, бізнес-логіка, доступ до даних).

Lifelong Learning (Навчання протягом життя) Концепція постійного самовдосконалення та оновлення знань протягом всієї професійної кар'єри.

Linter (Лінтер) Інструмент, що аналізує код на наявність помилок, проблем безпеки та відхилень від стандарту кодування.

Mentoring (Менторинг) Процес, при якому досвідчений фахівець надає рекомендації, знання та підтримку менш досвідченому колегі.

Microservices Architecture (Мікросервісна архітектура) Архітектурний стиль, де система складається з набору невеликих, незалежних сервісів, що спілкуються через мережу.

Middle Developer (Мідл-розробник) Спеціаліст з досвідом, що самостійно виконує більшість завдань, розуміє бізнес-логіку проєкту та може менторити джуніорів.

Monolithic Architecture (Монолітна архітектура) Архітектурний стиль, де вся система є єдиним модулем, що розгортається разом.

MVP (Minimum Viable Product) Версія продукту з мінімальним набором функцій, достатнім для задоволення перших користувачів та збору зворотного зв'язку.

Non-functional Requirements (Нефункціональні вимоги) Вимоги, що описують, ЯК система повинна працювати — атрибути якості та властивості (performance, reliability, security, usability, scalability, maintainability, portability).

Open Source Software (Відкрите ПЗ) Програмне забезпечення з відкритим вихідним кодом, який доступний для перегляду, вивчення та зміни.

Pair Programming (Програмування вдвох) Техніка розробки, при якій два програмісти працюють разом за одним комп'ютером: один пише код (Driver), інший переглядає (Navigator).

Permissive License (Дозвільна ліцензія) Ліцензія на вільне ПЗ з мінімальними обмеженнями щодо використання та розповсюдження (наприклад, MIT, Apache).

Planning Poker (Покер планування) Техніка групової оцінки задач в Agile, що використовує картки з числами для досягнення консенсусу.

Privacy by Design Підхід до проєктування систем, де захист приватних даних враховується на всіх етапах розробки, а не додається пізніше.

Product Backlog Впорядкований список всього, що може знадобитися в продукті, за управління яким відповідає Product Owner.

Product Owner Роль у Scrum, що відповідає за максимізацію цінності продукту та управління Product Backlog.

Project (Проєкт) Тимчасова діяльність, спрямована на створення унікального продукту, послуги або результату, що має чіткий початок і кінець.

Project Management (Управління проєктами) Застосування знань, навичок, інструментів і технік до проєктної діяльності для задоволення вимог проєкту.

Quality Assurance (QA) Набір превентивних процесів, спрямованих на запобігання дефектам через налагодження процесів розробки.

Quality Control (QC) Набір дій для виявлення дефектів у готовому продукті через перевірку відповідності вимогам.

Refactoring (Рефакторинг) Процес зміни внутрішньої структури програми без зміни її зовнішньої поведінки з метою поліпшення читабельності та підтримуваності.

Regression Testing (Регресійне тестування) Повторне тестування після внесення змін для перевірки того, що існуючий функціонал не зламався.

Remote Work (Віддалена робота) Форма роботи, при якій співробітник виконує свої обов'язки з дому або іншого місця поза офісом.

Requirement (Вимога) Умова або здатність, якій повинна відповідати система, щоб задовольнити угоду, стандарт, специфікацію або інші офіційно встановлені документи.

Requirements Engineering (Інженерія вимог) Системний процес виявлення, аналізу, документування та перевірки вимог до програмного забезпечення.

Safety-Critical Systems Системи, відмова яких може призвести до людських жертв, значних фінансових втрат або екологічної катастрофи.

Scrum Найпопулярніший Agile-фреймворк для управління складними проєктами, що використовує спринти, ролі та чіткі артефакти.

Scrum Master Роль у Scrum — слуга-лідер, що допомагає команді зрозуміти Scrum, усуває перешкоди та фасилітує події.

SDLC Життєвий цикл розробки програмного забезпечення — систематичний процес розробки високоякісного ПЗ у передбачувані терміни та в рамках бюджету.

Senior Developer (Сеніор-розробник) Експерт з глибокими знаннями, що приймає архітектурні рішення, розв'язує складні проблеми та відповідає за якість коду команди.

Soft Skills (М'які навички) Нетехнічні навички, такі як комунікація, вирішення конфліктів, командна робота та емоційний інтелект.

Software Architecture (Архітектура ПЗ) Високорівнева структура системи, що визначає її компоненти, їхню взаємодію, технології та глобальні обмеження.

Software Construction (Конструювання ПЗ) Детальне створення робочого програмного забезпечення за допомогою комбінації кодування, верифікації, модульного тестування, інтеграційного тестування та налагодження.

Software Crisis Термін, що описує проблеми розробки ПЗ (перевищення бюджету, ненадійність), які виникли наприкінці 1960-х років.

Software Design (Дизайн ПЗ) Процес деталізації окремих модулів, класів та функцій для реалізації архітектурних рішень.

Software Engineering Застосування систематичного, дисциплінованого, вимірюваного підходу до розробки, експлуатації та супроводу програмного забезпечення.

SOLID Мнемонічна абревіатура для п'яти принципів об'єктно-орієнтованого проєктування, що сприяють створенню гнучкого та підтримуваного коду.

Specification (Специфікація вимог) Третій етап процесу інженерії вимог — документування вимог у формальному та структурованому вигляді.

Sprint Фіксований період часу (1–4 тижні) у Scrum, що містить планування, розробку та огляд роботи.

Stakeholder (Стейкхолдер) Будь-яка особа або організація, яка має інтерес у розробці системи (замовник, користувач, розробник, менеджер).

Story Points (SP) Відносна одиниця виміру складності задачі в Agile, що враховує обсяг роботи, складність та ризики.

Structural Patterns (Структурні патерни) Патерни, що визначають, як класи та об'єкти компонуються в більші структури (наприклад, Adapter, Decorator).

SWEBOK Software Engineering Body of Knowledge — документ, що консолідує загальноприйняті знання в галузі програмної інженерії.

System Testing (Системне тестування) Рівень тестування, що перевіряє повну, інтегровану систему на відповідність функціональним та нефункціональним вимогам.

Technical Debt (Технічний борг) Метафора, що описує накопичені наслідки швидких та брудних рішень у коді, які потребують подальшої сплати часу.

Test-Driven Development (TDD) Підхід до розробки, де тести пишуться перед кодом за циклом Red-Green-Refactor.

Trade Secret (Комерційна таємниця) Інформація, яка не є загальновідомою і розголошення якої може завдати шкоди інтересам особи, якій вона належить.

Unit Testing (Модульне тестування) Рівень тестування, що перевіряє окремі найменші частини коду (функції, методи, класи) ізольовано від інших.

User Story (Історія користувача) Короткий опис функціональності з точки зору користувача, у форматі "Як роль, я хочу дію, щоб отримати цінність".

V-Model Модель розробки, що розширює Waterfall, де кожному етапу розробки відповідає певний етап тестування для гарантії якості.

Velocity (Швидкість команди) Метрика в Scrum, що показує, скільки роботи (в Story Points) команда може виконати за один спринт.

Waterfall Каскадна модель розробки, де процес слідує лінійній схемі через дискретні етапи.

White Box Testing (Тестування білої скриньки) Метод тестування, при якому тестувальник має доступ до внутрішнього коду і перевіряє логічні шляхи, умови та цикли.

Аналіз вимог Процес збору та документування потреб замовника та стейкхолдерів для розуміння того, що потрібно будувати.

Проєктування Етап життєвого циклу, де визначається архітектура системи, інтерфейси, структури даних та взаємодія компонентів.

Реалізація Етап безпосереднього написання програмного коду в обраній мові програмування.

Спіральна модель Модель розробки, запропонована Баррі Бомом, що фокусується на управлінні ризиками через ітеративні цикли.

Тестування Процес перевірки відповідності програмного забезпечення вимогам, пошуку дефектів та забезпечення якості.

Список джерел

[1] I. Sommerville, *Software Engineering*, 10th. Pearson, 2016, Дата звернення: 16.08.2025. https://software-engineering-book.com/.

[2] Є. Левус та Н. Мельник, *Вступ до інженерії програмного забезпечення*. Львів: Видавництво Львівської політехніки, 2017, с. 280.

[3] ISO/IEC/IEEE, *Systems and software engineering — Vocabulary*, Standard, ISO/IEC/IEEE 24765:2017, International Organization for Standardization, 2017.

[4] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Anniversary. Addison-Wesley, 1995.

[5] H. Washizaki, *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*, Version 4.0. IEEE Computer Society, 2024, Дата звернення: 16.08.2025. https://www.computer.org/education/bodies-of-knowledge/software-engineering.

[6] IEEE. «IEEE Code of Ethics.» Дата звернення: 16.08.2025. https://www.ieee.org/about/corporate-governance/constitution-bylaws/code-of-ethics.html.

[7] K. Beck, M. Beedle, A. van Bennekum та ін. «Manifesto for Agile Software Development.» Дата звернення: 16.08.2025. http://agilemanifesto.org/.

[8] Microsoft. «GitHub Copilot Fundamentals Part 1 of 2.» Дата звернення: 16.08.2025. https://learn.microsoft.com/en-us/training/paths/copilot/.

[9] S. Chacon та B. Straub. «Pro Git.» Дата звернення: 16.08.2025. https://git-scm.com/book/uk/v2.

[10] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd. Microsoft Press, 2004.

[11] B. W. Boehm, «A Spiral Model of Software Development and Enhancement,» *IEEE Computer*, т. 21, № 5, с. 61—72, 1988.

[12] B. W. Boehm, «Spiral Development: Experience, Principles, and Refinements,» Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, тех. звіт. CMU/SEI-2000-SR-008, лип. 2000, Дата звернення: 02.12.2025. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5180.

[13] K. E. Wiegers та J. Beatty, *Software Requirements*, 3rd. Microsoft Press, 2013, Розширена версія з практичними прикладами та техніками документування вимог у сучасних проектах.

[14] D. C. Gause та G. M. Weinberg, *Exploring Requirements: Quality Before Design*. Dorset House, 1989, Класична книга про методи взаємодії зі стейкхолдерами та виявлення справжніх потреб.

[15] K. E. Wiegers, *Software Requirements*, 2nd. Microsoft Press, 2003, Фундаментальна книга про методи збору, аналізу та специфікації вимог.

[16] S. Robertson та J. Robertson, *Mastering the Requirements Process: Getting Requirements Right*, 2nd. Addison-Wesley, 2006, Практичний посібник з методологій збору та аналізу вимог, включаючи сценарії та use cases.

[17] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002, Класична книга про архітектурні патерни для корпоративних систем.

[18] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017, Книга, що детально розглядає принципи SOLID та архітектуру ПЗ.

[19] E. Gamma, R. Helm, R. Johnson та J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, Оригінальна книга "Банди чотирьох" (GoF) про патерни проєктування.

[20] ISO/IEC/IEEE, *Systems and software engineering — Architecture description*, Standard, ISO/IEC/IEEE 42010:2011, International Organization for Standardization, 2011.

[21] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008, Фундаментальна книга про якість коду та професійні практики.

[22] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd. Addison-Wesley, 2018.

[23] PMI, *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*, 6th. Project Management Institute, 2021.

[24] S. McConnell, *Software Estimation: Demystifying the Black Art*. Microsoft Press, 2006.

[25] N. N. Taleb, *The Black Swan: The Impact of the Highly Improbable*. Random House, 2007, Ризик-менеджмент в програмній інженерії.

[26] ISO, *Project, programme and portfolio management — Guidance on project management*, Standard, ISO 21502:2020, International Organization for Standardization, 2020.

[27] ACM. «ACM Code of Ethics and Professional Conduct.» Дата звернення: 16.08.2025. https://www.acm.org/code-of-ethics.

[28] E. S. Raymond, *The Cathedral and the Bazaar*. O'Reilly Media, 2001.

[29] DataCamp. «EU AI Act Fundamentals.» Дата звернення: 16.08.2025. https://app.datacamp.com/learn/skill-tracks/eu-ai-act-fundamentals.

[30] ISO/IEC, *Information Technology Security Techniques Information Security Management Systems*. 2022.

[31] A. Hunt та D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.