Лекція 1. Вступ. Життєвий цикл розробки програмного забезпечення

Розробка ПЗ як інженерна діяльність: це складний творчо-технічний процес, який включає в себе планування, аналіз, проектування, реалізацію, тестування і підтримку програмного забезпечення.

Розробка ПЗ повинна мати чіткі стандарти, інструменти і методології, подібно до інших інженерних дисциплін.

Складнощі виконання проектів: часті затримки, перевищення бюджету, зміна вимог, низька якість ПЗ.

Причини: погане планування, нечіткі вимоги, недостатнє тестування, відсутність комунікації.

Потреба у формалізації практик: застосування методологій розробки допомагає знизити ризики, забезпечити якість та передбачуваність процесу.

Життєвий цикл розробки ПЗ (Software Development Life Cycle, SDLC): це структура процесів, які охоплюють повний цикл створення ПЗ – від ініціації до завершення та підтримки.

Етапи ЖЦ:

1) Підготовка (аналіз доцільності)
2) Збір вимог
3) Проектування
4) Реалізація (програмування)
5) Тестування
6) Розгортання
7) Впровадження
8) Підтримка.

Переваги класичного ЖЦ: чітка структура, зрозумілість, контроль.

Недоліки: негнучкість, складно адаптувати до змін, ризик виявлення помилок на пізніх етапах.

Моделі ЖЦ:

- Послідовна (Waterfall): етапи виконуються один за одним. Мінімум змін, максимальна плановість.

- Спіральна: ітераційна модель з фокусом на управління ризиками.

- Ітеративна: поступове вдосконалення системи через повторення циклів.

- Гнучкі методології (Agile): адаптивний підхід, короткі ітерації, постійний зворотний зв'язок (Scrum, Kanban тощо).

Лекція 2. Вимоги до програмного забезпечення

Вимоги – це описи того, що повинне робити ПЗ або які характеристики повинно мати.

Основні типи: функціональні і нефункціональні вимоги.

SWEBOK (Software Engineering Body of Knowledge) описує вимоги як фундаментальний етап інженерії ПЗ.

Аналіз і збирання вимог: інтерв'ю, опитування, спостереження, аналіз документів.

Інженерія вимог – це процес збору, документування, аналізу, перевірки та керування вимогами.

Керування вимогами: контроль змін, версійність, пріоритезація.

Якість вимог: вимоги мають бути повними, однозначними, перевіряльними, узгодженими.

Трасування вимог: встановлення зв’язку між вимогами та компонентами ПЗ.

Валідація – перевірка правильності (чи відповідає потребам).

Верифікація – перевірка відповідності вимогам.

Підходи до представлення: текстовий, візуальний (діаграми), об’єктно-орієнтований (використання UML, прецедентів, класів).

Лекція 3. Проектування програмного забезпечення

Архітектура ПЗ – високорівнева структура системи, що визначає компоненти та їх взаємодію.

Принципи проектування: модульність, абстракція, інкапсуляція, повторне використання, узгодженість.

Структурне проектування – функціональний підхід: використовує блок-схеми, DFD (Data Flow Diagram), ERD (Entity-Relationship Diagram).

Об'єктно-орієнтоване проектування – будується на класах і об'єктах.

Принципи SOLID:

S — Single Responsibility Principle (Принцип єдиної відповідальності)

Суть: клас повинен мати лише одну причину для зміни, тобто виконувати тільки одну функцію або відповідати за один аспект системи.

Пояснення: кожен клас має відповідати лише за один елемент логіки — наприклад, клас, який відповідає за зберігання даних, не повинен містити логіку відображення цих даних.

Приклад: клас UserRepository займається лише збереженням та отриманням користувачів, а не їх валідацією чи виводом.

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

Суть: програмні сутності (класи, модулі, функції) повинні бути відкритими для розширення, але закритими для змін.

Пояснення: потрібно писати код так, щоб для додавання нової поведінки не потрібно було змінювати вже існуючий код, а лише додавати новий.

Приклад: замість того, щоб змінювати клас при додаванні нового типу знижки, можна реалізувати інтерфейс IDiscount, а нові типи — як окремі класи.

L — Liskov Substitution Principle (Принцип підстановки Лісков)

Суть: об'єкти підкласів повинні заміняти об'єкти батьківських класів без порушення логіки програми.

Пояснення: підкласи не повинні змінювати очікувану поведінку батьківських класів. Інакше кажучи, будь-який екземпляр підкласу повинен «вести себе» так само, як базовий клас.

Приклад: якщо клас Bird має метод fly(), то підклас Penguin не повинен його успадковувати, бо пінгвіни не літають — краще створити окрему ієрархію.

I — Interface Segregation Principle (Принцип розділення інтерфейсу)

Суть: не слід змушувати клієнтів реалізовувати інтерфейси, які вони не використовують.

Пояснення: краще створювати кілька вузьких інтерфейсів, ніж один великий. Це полегшує підтримку і повторне використання.

Приклад: замість одного інтерфейсу IMachine з методами Print(), Scan(), Fax() краще мати окремі: IPrinter, IScanner, IFax.

D — Dependency Inversion Principle (Принцип інверсії залежностей)

Суть: високорівневі модулі не повинні залежати від низькорівневих модулів — обидва мають залежати від абстракцій.

Пояснення: залежності повинні бути на рівні інтерфейсів або абстрактних класів, а не конкретних реалізацій.

Приклад: замість того, щоб у класі OrderService створювати екземпляр MySQLDatabase, краще залежати від інтерфейсу IDatabase і передавати реалізацію через конструктор (інжекція залежностей).

Проектування розподілених систем: компоненти працюють на різних вузлах мережі, взаємодіють через API. Використання брокерів повідомлень, балансування навантаження.

Мікросервісна архітектура: система складається з окремих сервісів, що можуть масштабуватись і розгортатись незалежно.

Хмарні системи: розміщення в інфраструктурі хмари (AWS, Azure тощо), масштабування, висока доступність.

Вбудовані системи: інтегровані з апаратним забезпеченням, мають обмеження по ресурсах, потребують високої надійності.

Лекція 4. Конструювання

Конструювання – етап, коли програмний код реалізується згідно з проектом.

Принципи: читабельність, повторне використання, модульність, стандарти стилю.

Екстремальне програмування (XP):

- Парне програмування: два розробники на одному комп'ютері.
- Спільне володіння кодом: усі розробники можуть змінювати будь-який код.
- TDD (розробка через тестування): спочатку пишеться тест, потім код.
- Постійна інтеграція (CI): автоматичне збирання проєкту після кожного коміту.

Feature-Driven Development (FDD): розробка за фічами (властивостями), з короткими ітераціями.

Domain-Driven Design (DDD): моделювання домену (предметної області), створення багатої предметної моделі.

CI/CD: процес автоматизованого тестування, збирання і розгортання ПЗ.

Лекція 5. Забезпечення якості

Якість ПЗ – ступінь відповідності функціональним і нефункціональним вимогам.

Формальні специфікації: математичні моделі для опису вимог і перевірки правильності реалізації.

Верифікація – перевірка відповідності реалізації вимогам.
Валідація – перевірка, що ПЗ вирішує потрібну задачу.

Тестування – виявлення помилок. Основні методи:

- Статичні: аналіз коду без виконання (рев'ю, лінтери).
- Динамічні: з виконанням програми (модульне, інтеграційне, системне тестування).

Методи динамічного тестування:

- Чорна скринька: тестування без знання внутрішньої структури.
- Біла скринька: тестування логіки реалізації.
- Шляхове тестування: перевірка логічних гілок.
- Інтеграційне тестування: тестування взаємодії між модулями.

Класифікація помилок:

- Помилка (error): неправильна дія розробника.
- Дефект (bug): неправильне функціонування.

- Відмова (failure): несподіване завершення роботи.
- Інцидент (incident): будь-який незапланований результат.

План тестування: описує підхід, ресурси, критерії приймання, відповідальність, типи тестів.