Лекція 10

Об'єктно-орієнтоване проєктування

Від ERD до UML Class Diagram

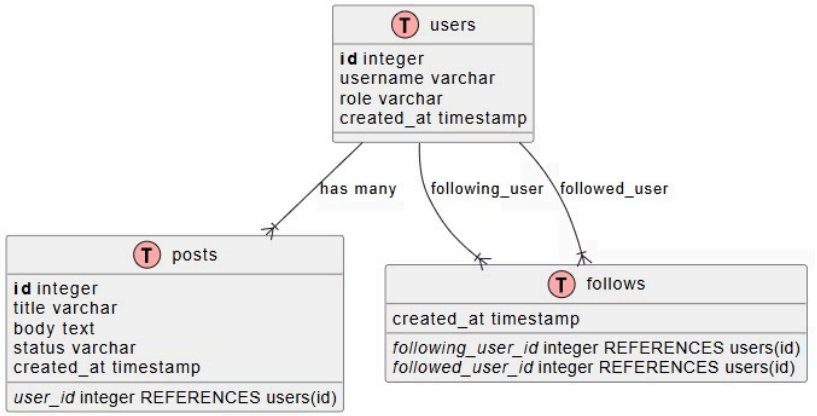
У попередніх темах ми детально розібрали проєктування баз даних за допомогою ER-діаграм. Тепер поговоримо про класи та об'єкти.

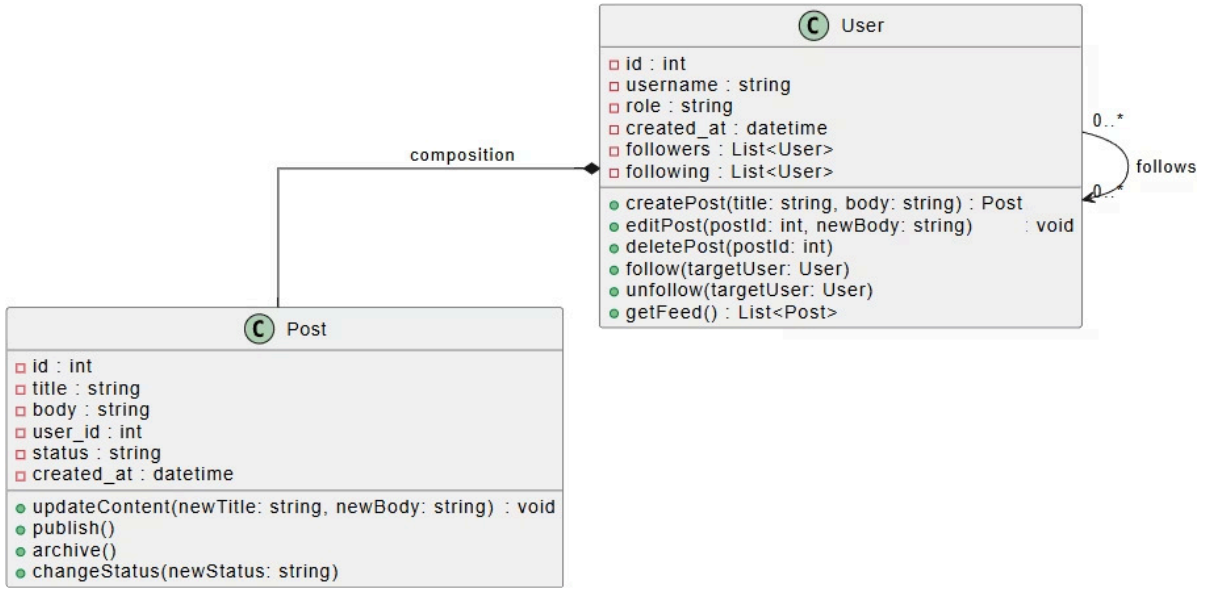
Реляційна модель (SQL)

Орієнтована на ефективне зберігання рядків.

Об'єктна модель (ООР)

Орієнтована на поведінку, інкапсуляцію та ієрархію.





Правила трансляції: Від ERD до UML

Перетворення ER-діаграми бази даних у діаграму класів відбувається за чітким алгоритмом. Саме цей процес автоматизують ORM-фреймворки (Hibernate у Java, Entity Framework у .NET), але архітектор повинен розуміти його "під капотом".

Правило 1: Таблиця → Клас

Кожна сутність (таблиця) з ERD стає класом у коді. Назви таблиць часто пишуться у множині (snake_case), а класи — завжди в однині (PascalCase).

- users_table → class User
- order_items → class OrderItem

Правило 2: Колонка → Атрибут

Кожна колонка бази даних стає змінною (полем) всередині класу. Типи даних конвертуються зі специфічних для СКБД типів у типи мови програмування.

- VARCHAR(255) → String
- DATETIME → LocalDateTime
- DECIMAL(10,2) → BigDecimal

Правило 3: Foreign Key → Асоціація

Найголовніша концептуальна зміна. В ООП ми уникаємо використання "голих" ID для зв'язку об'єктів. Замість поля customerId маємо поле customer типу Customer.

Чому так роблять? Тому що в об'єктно-орієнтованому коді природніше написати `order.getCustomer().getName()`, аніж вручну шукати клієнта за його ID у базі даних (імітуючи JOIN у коді).

Анатомія класу в UML

Прямокутник, що позначає клас в UML, є набагато інформативнішим, ніж символ сутності в ER-діаграмі. Він завжди розділений на три горизонтальні секції, кожна з яких несе критично важливу інформацію для розробника.

Ім'я класу

Верхня секція. Пишеться жирним шрифтом, вирівнюється по центру. Якщо клас абстрактний, ім'я пишеться курсивом. Тут також можуть вказуватися стереотипи.

Атрибути / Стан

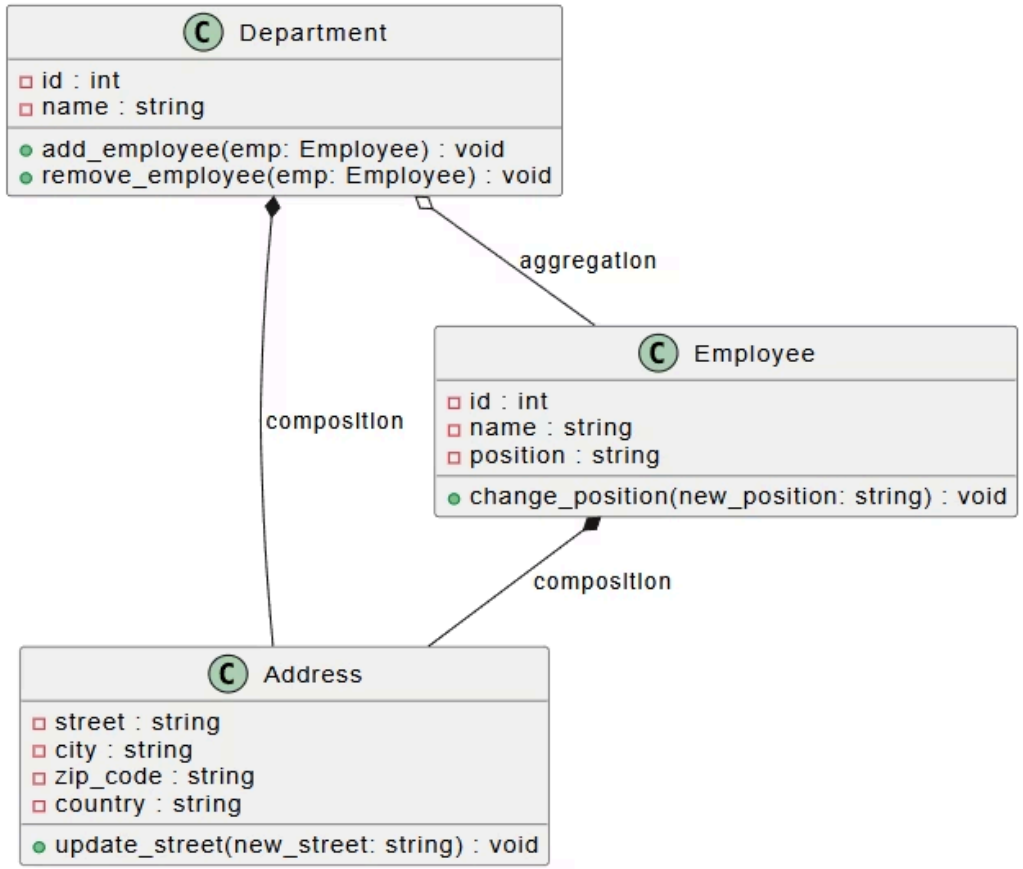
Середня секція. Змінні, що зберігають дані об'єкта. Формат: [видимість] ім'я : тип [= значення].

Приклад: `- balance : double = 0.0`

Операції / Поведінка

Нижня секція. Методи (функції) класу. Це принципова відмінність від баз даних! Таблиці пасивні, класи активні.

Формат: [видимість] метод(аргумент : тип) : тип_повернення



Видимість та Інкапсуляція

В ER-моделі всі дані у таблиці зазвичай "публічні". В об'єктно-орієнтованому програмуванні фундаментальним принципом є **Інкапсуляція** — приховування внутрішнього стану об'єкта та надання доступу до нього лише через контрольовані методи.

Private (-)

Приватний доступ. Атрибут або метод доступний лише всередині цього ж класу. Золотий стандарт для всіх полів з даними.

Public (+)

Публічний доступ. Доступно будь-якому іншому класу в системі. Зазвичай використовується для методів, що формують API класу.

Protected (#)

Захищений доступ. Доступно всередині класу та у всіх класах, що від нього успадковані (класах-нащадках).

Package (~)

Доступно для інших класів у межах того ж самого пакета (папки). Використовується для внутрішньої організації модулів.

Типовий патерн проєктування: Робити всі атрибути приватними (наприклад, - salary : int), а читати чи змінювати їх дозволяти лише через публічні методи (наприклад, + getSalary() : int). Це захищає об'єкт від некоректних змін ззовні.

Навігація та Кратність

На діаграмі класів лінії зв'язку (Асоціації) часто мають стрілки на кінцях. Ці стрілки показують **Навігацію (Navigability)** — напрямок, у якому можна передавати повідомлення або який об'єкт зберігає посилання на інший.

Типи навігації

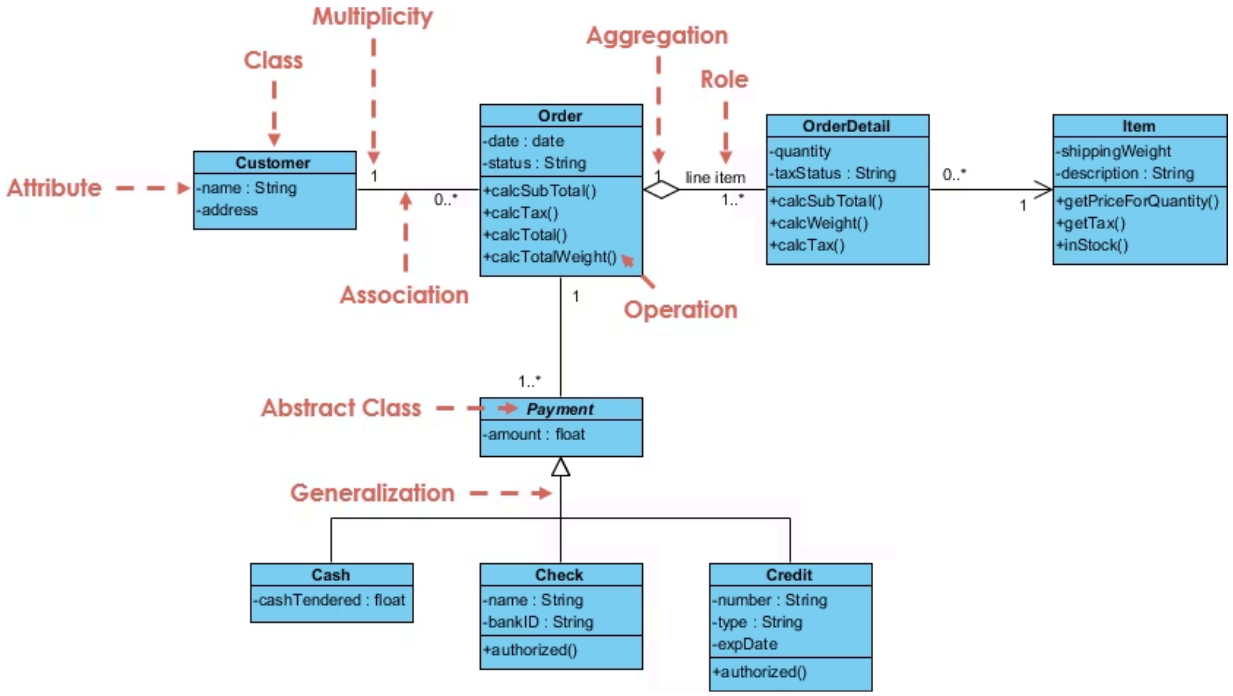
Направлена стрілка (Order → Customer): Об'єкт Order "знає" про Customer. У коді клас Order має поле типу Customer. Зворотного зв'язку немає.

Двонаправлений зв'язок: Взаємна обізнаність. Order знає про Customer, а Customer зберігає колекцію своїх Order. У коді складніше підтримувати.

Кратність (Multiplicity)

Визначає, скільки екземплярів одного класу може бути пов'язано з екземпляром іншого:

1 — Рівно один обов'язковий
0..1 — Нуль або один (опціональний)
***** або **0..*** — Будь-яка кількість
1..* — Мінімум один



Види відносин між класами

Тип лінії між класами в UML диктує розробнику дві надважливі речі: життєвий цикл об'єктів та реалізацію в коді. Всі відношення діляться на дві великі групи: "IS-A" (Є різновидом) через Спадкування та "HAS-A" (Має/Містить) через Асоціацію.

—

Асоціація (Association)

Базовий, найслабший тип зв'язку. Об'єкти взаємодіють, але є рівноправними і незалежними. Життєвий цикл незалежний. Приклад: Студент і Викладач.

◇

Агрегація (Aggregation)

Логіка "Частина — Ціле", але слабкий зв'язок. Символ: порожній ромб (◇). Життєвий цикл незалежний. Приклад: Факультет та Професор — факультет може розформуватися, професор залишається.

◆

Композиція (Composition)

Найжорсткіша форма "Частина — Ціле". Символ: чорний ромб (◆). Життєвий цикл залежний — знищення "Цілого" вимагає знищення "Частин". Приклад: Замовлення та Рядок замовлення.

▷

Спадкування (Inheritance)

Відношення "IS-A". Символ: порожня трикутна стрілка (△). Нащадок автоматично отримує всі не-приватні поля та методи батьківського класу. Приклад: Admin є User.

DESIGN PATTERNS

Чому ми не вигадуємо колесо?

У програмуванні є парадокс: задачі, які ви вирішуєте (створення об'єкта, підписка на події, обхід списку), вже були вирішені тисячі разів до вас. Коли Junior стикається з проблемою, він часто пише унікальне, "геніальне" і заплутане рішення. Це називається "Винаходити велосипед".

> Коли Senior стикається з тією ж проблемою, він каже: "О, це ж задача для патерну Стратегія".

Патерн проєктування (Design Pattern) — це перевірене часом, універсальне рішення поширеної архітектурної проблеми. Це не готовий код (бібліотека), який можна скопіювати. Це креслення або шаблон, який ви адаптуєте під свій проєкт.

Головна мета патернів: Зробити код стійким до змін.

Пам'ятайте: код пишеться один раз, а читається і змінюється — сотні разів.

Архітектурні вороги: Coupling та Cohesion

Перш ніж лікувати систему патернами, треба зрозуміти, чим вона хворіє. Усі "хвороби" архітектури зводяться до порушення балансу між двома метриками.

Висока зв'язність (High Coupling) — ПОГАНО

Класи "знають занадто багато" один про одного.

Симптоми: Ви змінюєте один рядок у класі Order, а ламається клас User, Payment і навіть Printer.

Ефект: Spaghetti Code. Система стає крихкою (Fragile). Зміни вносити страшно.

Рішення: Патерни допомагають розірвати прямі зв'язки через інтерфейси або події.

Низьке зчеплення (Low Cohesion) — ПОГАНО

Один клас робить все підряд: і в базу пише, і PDF генерує, і email відправляє.

Симптоми: God Object (Божественний об'єкт). Клас на 3000 рядків коду.

Ефект: Такий клас неможливо протестувати і неможливо перевикористати.

Рішення: Патерни допомагають розбити монстра на маленькі, спеціалізовані класи.

Low

Coupling

Класи незалежні (як деталі LEGO)

High

Cohesion

Клас робить одну справу, але робить її добре (SRP)

Типові "запахи" коду

Ось конкретні проблеми, які сигналізують: "Тобі потрібен патерн!"

Проблема: "Забагато new"

У вас по всьому коду розкидано `new Database()`, `new Service()`. Якщо конструктор Database зміниться, вам доведеться правити 50 файлів.

Ліки: Factory або Builder. Вони централізують створення об'єктів.

Проблема: "Величезний if-else або switch"

Коли з'явиться новий формат експорту, ви полізете змінювати старий, протестований клас (порушення Open/Closed Principle).

Ліки: Strategy або State. Замінюємо умови на поліморфізм.

Проблема: "Зміни в одному модулі ламають інший"

Коли змінюється ціна товару, треба оновити вітрину, надіслати пуш і перерахувати кошик. Клас Product починає викликати всі ці сервіси напряму.

Ліки: Observer. Товар просто повідомляє "Я змінився!", а інші самі реагують.

Проблема: "Копіпаста алгоритмів"

У двох класах метод `save()` на 90% однаковий, відрізняється лише один крок.

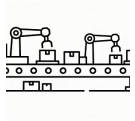
Ліки: Template Method. Виносимо спільне в батьківський клас, а відмінності — в абстрактні методи.

Група 1: Породжуючі патерни

Суть: Абстрагування процесу створення об'єктів (Instantiation). Головна проблема: Оператор `new` — це зло в великих системах. Коли ви пишете `new SQLDatabase()`, ви жорстко прив'язуєте свій код до конкретного класу. Породжуючі патерни ховають `new` всередину спеціальних методів.

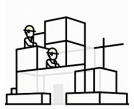
Singleton (Одинак)

Гарантує, що клас має лише один екземпляр, і надає глобальну точку доступу до нього. Кейс: Логер, конфігурація, пул з'єднань з БД.



Factory Method

Визначає інтерфейс для створення об'єкта, але дозволяє підкласам вирішувати, який саме клас інстанціювати. Кейс: Кросплатформні UI-елементи.



Builder (Будівельник)

Дозволяє створювати складні об'єкти покроково. Рятує від конструкторів з 10 параметрами. Кейс: Створення SQL-запиту або HTTP-request.

Prototype (Прототип)

Створює нові об'єкти шляхом клонування вже існуючих. Корисно, коли створення об'єкта з нуля дороге.

Група 2: Структурні патерни

Суть: Композиція класів та об'єктів. Головна проблема: "Як зліпити з різних деталей єдине ціле, якщо вони не підходять одна одній?" або "Як розширити функціонал класу, не використовуючи спадкування?". Структурні патерни використовують інтерфейси та делегування.

Adapter (Адаптер)

Дозволяє об'єктам з несумісними інтерфейсами працювати разом. Кейс: Система очікує XML, а нова бібліотека приймає JSON.

Proxy (Замісник)

Замінник іншого об'єкта для контролю доступу до нього. Кейс: Ліниве завантаження картинок, кешування, перевірка прав.

Facade (Фасад)

Надає простий інтерфейс до складної системи класів. Кейс: Кнопка "Запустити двигун" приховує складну логіку.

Composite (Компонувальник)

Дозволяє згрупувати об'єкти в деревоподібну структуру. Кейс: Файлова система (папка містить файли та інші папки).

Група 3: Поведінкові патерни

Суть: Ефективна комунікація та розподіл відповідальності. Головна проблема: "Spaghetti Code" у потоці виконання. Коли один клас занадто сильно контролює інший, або коли логіка розгалужена на сотні if-else. Ці патерни фокусуються на тому, як об'єкти "спілкуються".

Observer (Спостерігач)

Визначає залежність "один-до-багатьох", щоб при зміні стану одного об'єкта всі залежні дізнавалися автоматично. Кейс: Підписка на події (YouTube, React).

Strategy (Стратегія)

Визначає сімейство алгоритмів, інкапсулює їх і робить взаємозамінними. Кейс: Сортування масиву, розрахунок ціни доставки.

Command (Команда)

Перетворює запит на об'єкт. Дозволяє ставити запити в чергу, скасовувати їх (Ctrl+Z) або логувати.

State (Стан)

Дозволяє об'єкту змінювати свою поведінку при зміні внутрішнього стану. Об'єкт ніби змінює свій клас.

Template Method

Задає скелет алгоритму в базовому класі, а деталі реалізації залишає підкласам.

Iterator (Ітератор)

Дає спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої структури.

Сучасний погляд: Що актуально в 2026?

Деякі патерни GoF "померли" або трансформувалися. Розуміння їхньої еволюції допомагає приймати правильні архітектурні рішення сьогодні.

Singleton → DI-контейнери

Зараз вважається "Anti-pattern" в чистому вигляді, бо заважає юніт-тестуванню. Його роль взяли на себе DI-контейнери (Spring Bean за замовчуванням є сінглтоном, але керується контейнером).

Iterator → Вбудований у мови

Вбудований у всі мови (foreach, stream). Писати свій ітератор вручну майже не доводиться. Патерн живе, але невидимо.

Prototype → Механізми копіювання

Рідко використовується вручну, але є основою механізмів копіювання в JS/Python. Працює "під капотом".

Builder, Strategy, Observer → Живіші всіх

Це основа сучасного реактивного програмування та мікросервісів. Використовуються повсюдно в React, Angular, Spring, Kafka.

Singleton: Король конфігурацій

Хоча Singleton часто критикують, він незамінний там, де потрібна Єдина Точка Істини (Single Source of Truth). Реальний кейс: ConfigurationManager. У вашому додатку є файл .env або application.properties, де лежать паролі до бази даних, API-ключі та налаштування портів.

Чому Singleton?

- Нераціонально зчитувати файл з диска при кожному зверненні
- Небезпечно мати дві різні версії налаштувань у пам'яті одночасно
- Потрібна глобальна точка доступу з будь-якого місця коду

Ключові елементи реалізації

1. Приватний конструктор: Ніхто ззовні не може зробити "new"
2. Статичний метод getInstance(): Глобальна точка доступу
3. Лінива ініціалізація: Об'єкт створюється тільки при першому зверненні

Де ви це бачите щодня

Loggers: Logger.getInstance(). Писати в один файл логів з різних потоків треба синхронізовано.

Database Connection Pool: Пул з'єднань має бути спільним для всього додатку.

Spring Beans: За замовчуванням усі сервіси в Spring Framework — це сінглтони (Scope = Singleton).

Factory Method: Вбивця жорстких зв'язків

Цей патерн рятує, коли ви не знаєте заздалегідь, з якими об'єктами доведеться працювати. Реальний кейс: Кросплатформний інтерфейс. Ви пишете додаток, який працює і в Web, і на Mobile.

1

Проблема

У Web кнопка — це HTML button. На Mobile — це нативний віджет UIButton. Писати if (platform == WEB) по всьому коду — жахливо.

2

Рішення

Створюємо Фабрику. Інтерфейс Button з методом render(). Класи WebButton та MobileButton реалізують цей інтерфейс.

3

Результат

Бізнес-логіка не знає, яка це кнопка. Вона просто каже "Створи" і отримує правильну реалізацію.

> Де ви це бачите щодня: **JDBC Drivers** (DriverManager.getConnection() — це фабрика), **Парсери** (ParserFactory.create('json') повертає JSON-парсер).

Strategy: Вбивця if-else

Це улюблений патерн для рефакторингу. Якщо ви бачите довгий ланцюжок if-else або switch, який вибирає алгоритм — це крик про допомогу. Реальний кейс: Розрахунок вартості доставки в інтернет-магазині.

Поганий код (Anti-pattern)

```
if (type == "NOVA_POSHTA") {
  return weight * 50;
} else if (type == "UBER") {
  return distance * 10;
} else if (type == "SELF") {
  return 0;
}
// Якщо додасться "Укрпошта",
// ми зламаємо цей клас
```

Гарний код (Strategy)

1. Інтерфейс DeliveryStrategy з методом calculate(Order)
2. Класи NovaPoshtaStrategy, UberStrategy, SelfPickupStrategy
3. Контекст Order має поле strategy

Можемо додавати нові служби доставки, не змінюючи код класу Order. Це дотримання Open/Closed Principle.

3

Типи доставки

Нова Пошта, Uber, Самовивіз

0

Змін у Order

При додаванні нової стратегії

Де ви це бачите щодня: Автентифікація (Login via Google, Facebook, Email), Сортування (Collections.sort(list, comparator)), Payment Gateways (PayPal / Stripe / ApplePay).

Observer: Реактивна система

Цей патерн є фундаментом сучасного UI та асинхронних систем. Він дозволяє об'єктам "підписуватися" на події. Реальний кейс: Система сповіщень про замовлення.

Коли статус замовлення змінюється на "Відправлено", треба: надіслати SMS клієнту, надіслати Email менеджеру, оновити лог на складі. Якщо ми запхаємо це все в метод Order.setStatus(), ми порушимо принцип єдиної відповідальності (SRP).

Клас Order не має знати про SMS. Він просто повідомляє всіх підписників про зміну стану.

Frontend (React/Angular)

Кнопка має onClick. Ваша функція "підписується" на клік.

Node.js

EventEmitter (server.on('request', ...)). Асинхронна обробка подій.

Message Brokers

Kafka/RabbitMQ — це архітектурна реалізація Observer у масштабі мікросервісів (Pub/Sub).

ФІНАЛ

Підсумок курсу: Від ідеї до архітектури

Ми пройшли інтенсивний шлях інженерного проєктування. Головна мета цього курсу полягала не в тому, щоб навчити вас малювати "красиві картинки" в UML, а в тому, щоб навчити мисленню системного архітектора.

Бізнес-процеси

Activity Diagram — формалізація хаосу реального життя, розділення зон відповідальності через Swimlanes.

Патерни проєктування

GoF Patterns — готові архітектурні рішення, лікування Coupling/Cohesion через Strategy, Factory, Observer.

Логіка станів

State Machine — боротьба з boolean flags hell, проєктування життєвого циклу складних сутностей.

Фундамент даних

ERD & Class Diagram — нормалізація до 3NF, трансляція SQL в OOP, вирішення Impedance Mismatch.

Архітектура C4

Context → Containers → Components. Обґрунтування технологічного стеку інженерними вимогами.

Динаміка системи

Sequence Diagram — різниця між синхронними та асинхронними викликами, життєвий цикл об'єктів.

Roadmap: Що вчити далі?

Цей курс дав вам фундамент System Design. Але IT-світ глибший. Щоб стати Senior Architect або Tech Lead, вам потрібно заглибитися в теми, які залишилися "за кадром".

1

High Load & Distributed Systems

CAP-теорема, Sharding баз даних, Реплікація (Master-Slave), Consistent Hashing. Інструменти: Redis, Kafka, Cassandra, Kubernetes.

2

Cloud Native & IaC

Автоматичне розгортання інфраструктури кодом (Terraform, Ansible). Serverless архітектура (AWS Lambda). Blue-Green Deployment, Canary Release.

3

API Design & Integration

gRPC для швидкого зв'язку між мікросервісами, GraphQL для гнучких фронтендів, Webhooks. OpenAPI (Swagger) для контрактів.

4

Security Architecture

OAuth 2.0 / OIDC (як працює "Log in with Google"), OWASP Top 10 (захист від злому), Cryptography basics.

5

Observability

Centralized Logging (ELK Stack), Distributed Tracing (Jaeger), Metrics & Alerting (Prometheus/Grafana).

> "Всі моделі неправильні, але деякі з них корисні." — Джордж Бокс