ФУНДАМЕНТ АРХІТЕКТУРИ

БАЗА, ЯКОЇ ВАС НІКОЛИ НЕ ВЧИЛИ

СЕРГІЙ
НЕМЧИНСЬКИЙ

Фундамент архітектури

База, якої вас ніколи не вчили

Сергій Немчинський

Ця книга продається на https://leanpub.com/architecture-foundation

Ця версія була опублікована 2026-04-02

Зміст

Передмова: Чому ця книга існує і як її читати

Вітаю, мої дорогі!

Спочатку про те, хто я такий і чому взагалі маю право розповідати вам про архітектуру. Мене звати Сергій Немчинський. Я в айтішці з 1996 року, відколи закінчив університет. З того часу я встиг попрацювати в купі різних компаній — від дрібних контор до топових українських аутсорсерів та продуктових гігантів, серед яких ЛІГА, Luxoft, Ciklum, Netcracker, IntroPro.

Я пройшов майже всі можливі ролі. Був наймолодшим керівником відділу в ЛІГА:ЗАКОН (до речі, перша пристойна версія їхнього порталу на початку 2000-х писалася саме під моїм керівництвом). Був Project Manager-ом у Ciklum, Архітектором у Luxoft, Тімлідом майже у всіх цих компаніях. Коротше, я бачив процес розробки і з окопів програміста, і з крісла архітектора, і з позиції бізнесу.

Протягом усієї цієї кар'єри я постійно проводив технічні співбесіди та нерідко викладав у корпоративних навчальних центрах. І десь у 2015-2016 роках, коли до мене на співбесіди почали масово приходити випускники різноманітних IT-курсів, я з жахом побачив, що вони… майже нічого не знають.

Я запитував їх: "А що ж ви вивчали на ваших курсах?". Вони з гордістю відповідали: "Ми робили бульбашкове сортування і рахували числа Фібоначчі!". Я питав: "А Spring вивчали?". Вони дивилися круглими очима: "Ні, а що таке Spring?". І це мене просто дико бісило.

Ці курси намагалися за три місяці втиснути п'ятирічну університетську програму, викидаючи все те, що реально потрібно програмісту на роботі: фреймворки, патерни, Enterprise-підходи. Я хапався за голову і врешті-решт вирішив: "Все, повна фігня. Я буду навчати людей сам". Так у 2016 році з'явилася компанія FoxmindEd, яка успішно навчила вже багато тисяч спеціалістів.

Окрім навчання новачків, ми створюємо серйозні програми для мідлів та сеньйорів. Зокрема, я особисто ще з 2008 року викладаю теорію базової архітектури (GRASP та GoF Design Patterns), а згодом додав до неї і більш просунутий курс з Enterprise-патернів. Спостерігаючи за своїми студентами та за індустрією загалом, я зрозумів, що сучасна освіта для досвідчених розробників має три фундаментальні проблеми. Які мене дико дратують.

Біль №1: Знання у вакуумі

Якщо сучасний програміст хоче розібратися в архітектурі, перед ним відкривається безліч книжок — від класичної "Банди Чотирьох" до новинок, що виходять ледь не щотижня. Але всі вони страждають на хворобу "знань у вакуумі".

Якщо книжка про SOLID, вона жодним словом не згадає про GoF-патерни, ніби їх не існує. Якщо це книжка про GRASP, окрім GRASP там нічого немає. У нещасних студентів виникає суцільний когнітивний дисонанс: вони бачать одну абревіатуру, інтуїтивно відчувають, що вона пов'язана з іншою, але НІХТО не пояснює як саме. Знання літають у голові окремими шматками і ніяк не склеюються до купи.

Біль №2: Застаріле лайно мамонта

Це просто якесь жахіття. Всі курси та книжки по GoF-патернах, які я бачив, просто беруть і слово в слово переказують книгу 1994 року. Йоханий бабай! Минуло більше 30 років!

Вони використовують ті самі діаграми класів, що й у 94-му. Наприклад, патерн Factory Method в тому вигляді, як його малюють у класичних діаграмах, просто не існує в сучасній природі. Так код ніхто не пише. Ба більше, так писати *не можна*, бо це не вирішує жодної проблеми!

У результаті людина дивиться на UML-діаграму в книжці, дивиться на сучасний код у своєму проєкті, бачить, що вони не співпадають, і вирішує: "Ці ваші патерни — застаріле лайно мамонта". А це не так! Патерни еволюціонували. Тому в цій книзі я буду "єретиком" — я розповідатиму, як ці патерни використовуються *насправді* сьогодні.

Біль №3: Сліпа віра у ШІ

Багато хто вважає, що відколи є штучний інтелект, архітектуру можна не вчити: “Він же сам напише код, він такий самовпевнений!”. Ні.

Якщо штучний інтелект згенерує вам код, і на нього подивиться досвідчений архітектор, він одразу зрозуміє: цей код — суцільне лайно. ШІ поки не вміє системно мислити. І напевно ніколи не буде вміти. Єдине, що робить нас кращими за LLM — це наше критичне мислення. А воно неможливе без фундаменту і принципів.

Якщо ви не знаєте китайської, будь-який переклад від ШІ здаватиметься вам правильним. Так само і тут: якщо ви не розумієте архітектури, будь-яка відрижка штучного інтелекту здаватиметься вам нормальним кодом. Ви повинні мати принципи, щоб сказати ШІ: “Ні, так не роби. Роби ось так”. Але щоб це сказати, треба *знати як саме*.

Для кого ця книга

Ця книга створена для:

- **Middle та Senior розробників**, які відчувають, що їхній код перетворюється на суцільне лайно, і хочуть нарешті побачити “Велику картину”.
- **Амбітних Junior-ів**, які не хочуть роками писати лайнокод, страждати від цього і не розуміти, куди рухатись далі. Вони хочуть одразу вчити архітектуру.
- Всіх, хто прочитав книгу GoF (до речі, вона дуже нудна, нікому не раджу її читати), нахапався фрагментарних знань, і тепер прагне об’єднати все це в структуровану систему.

Як казали мені ще в університеті: *знання декількох принципів звільняє від запам’ятовування багатьох фактів*. Я хочу, щоб ви мали в голові єдину систему координат, за якою відрізняється хороший код від поганого.

Як ми будемо вчитися: Problem-First

Я інженер, а не науковець. Будь-яке рішення має сенс *лише тоді*, коли у нас є проблема, яку треба вирішити. Робити “бо це красиво” — це

шлях в нікуди. Тому мій підхід — **Problem-First**. Спочатку ми до болю розбираємо проблему, а лише потім застосовуємо патерн як рішення. І приклади будуть із сучасного життя, а не “сферичні коні” з 90-х.

Також я обіцяю **не мучити вас зайвим архітектурним брудом**. Багато підручників починаються з десятків сторінок зашифрованих UML-позначень, стрілочок і кружечків. Я дам вам лише те, що реально використовується.

Якось я здав американським замовникам (індійського походження) ідеальний архітектурний документ. Частина діаграм була у форматі Activity (схоже на шкільні блок-схеми). Вони повернули його зі словами: *“Будь ласка, намалюйте нам UML, ми взагалі не розуміємо, як виглядає ця діаграма”*. Це чудовий показник. Не треба займатися наукоподібним навчанням. Використовуйте прості речі, які зрозумілі всім. Це саме стосується, до речі, і різних тренажерів на кшталт LeetCode — на мій погляд, це сьогодні чистий карго-культ, який не робить ваш код кращим.

Структура: Куди ми йдемо

Ми маємо застекувати всі ваші знання:

1. Згадаємо базовий фундамент **ООП** (більшість пам’ятає його зовсім не так, як він є).
2. Розберемо правила розподілу відповідальності **GRASP** (це основа).
3. Поєднаємо все це з “технікою безпеки” — принципами **SOLID**.
4. Розглянемо сучасні інструменти мікро-рівня — **GoF патерни**.
5. Визначимо дальні сходинки — куди вам рухатися далі.

Подяка Early Access читачам

Окремо хочу подякувати всім, хто купив і читає цю книгу в процесі її написання. Це просто вогонь! Мене це дико надихає.

Я людина відповідальна, але я розумію: якби я вирішив писати одразу всю книгу від початку до кінця, я б її не закінчив ніколи. Будучи

директором компанії, я б завжди знаходив інші, “більш важливі” справи. Але якщо мені заплатили гроші — я вже закінчу її точно! Така вже я людина 😄.

Дуже прошу вас: **діліться фідбеком**. Саме ваші відгуки допомагають мені структурувати, приводити до ладу весь контекст книги і робити її максимально корисною та вивіреною для всіх читачів.

Люблю вас. Залишайтеся з нами, а ми поїхали розбиратися з фундаментом архітектури!

🔑 Висновки розділу

- **Архітектура потрібна всім:** Без неї ви не зможете ні контролювати свій код, ні ставити осмислені задачі штучному інтелекту.
- **Знання мають бути системними:** Не можна вчити патерни у відриві один від одного. ООП, GRASP, SOLID та GoF — це єдина взаємопов’язана система.
- **Problem-First:** Кожен патерн — це інструмент вирішення конкретного болю. Якщо болю немає, патерн не потрібен.
- **Геть академізм:** Патерни з 1994 року змінилися. Ми вивчатимемо їх так, як вони використовуються в реальному продакшені сьогодні, а не в інститутських методичках.

Частина I. Фундамент та Філософія

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Розділ 1. Еволюція болю: Від GOTO до ООП

Надворі 2026 рік. У нас є все: ШІ-асистенти, хмарні середовища розробки, фреймворки, які ледь не читають наші думки. Здавалося б, живи і радій, але чому тоді бізнес досі платить шалені гроші Software Architects? Чому на співбесідах сеньйорів досі ганяють по “древніх” патернах?

І чому проєкти, які починаються як “цукерочка”, через рік перетворюються на непідтримуване пекло, яке всі бояться чіпати?

Штучний інтелект генерує код блискавично. Але без розуміння архітектурних патернів ви навіть не зможете оцінити, чи він згенерував чисту архітектуру, чи нечитабельне місиво. Чат-бот видав двісті рядків коду — а ви навіть не знаєте, чи це адекватний дизайн, чи класичний антипатерн.

Відповідь ховається в одному слові: **Складність**.

Ми століттями боролися з хаосом. І щоб зрозуміти, *навіщо* вам потрібні правила (такі як GRASP чи SOLID), треба пройти цей шлях еволюції. Повірте, якщо ви не знаєте, як боляче б’ють граблі, на які наступали до вас, ви гарантовано наступите на них сьогодні.

Тож почнемо із шістдесятих років, коли програмісти були справжніми ковбоями, а пам’ять комп’ютера вимірювалася в кілобайтах.

Ера Хаосу: Spaghetti Code та GOTO (1960-ті)

Уявіть собі шістдесяті роки. Ми пишемо на Асемблері, Fortran або ранньому BASIC. У нас немає функцій у сучасному розумінні. Немає циклів while чи for. У нас є лише одна суперзброя: оператор GOTO.

Типовий код того часу виглядав приблизно так:

```
Рядок 10: Зроби щось.
Рядок 20: Якщо умова X — стрибни на рядок 500.
Рядок 500: Зроби щось і стрибни назад на рядок 30.
```

Це призводило до явища, яке отримало назву **Spaghetti Code**. Не в переносному сенсі, а буквально. Якщо ви спробуєте намалювати потік виконання на папері — це буде схоже на тарілку з макаронами. Зрозуміти логіку програми, просто читаючи код зверху вниз, було неможливо. Ви мусили тримати в голові всю карту пам’яті.

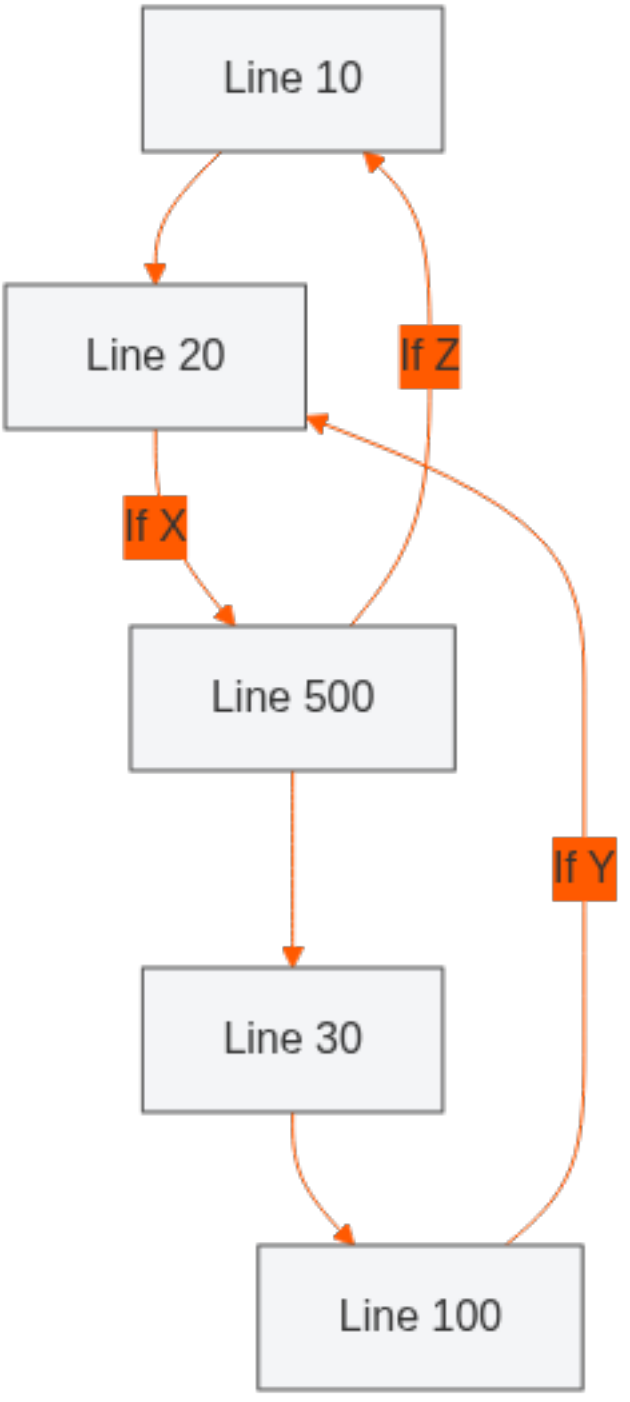

Рисунок 1. Структура потоку виконання при Spaghetti Code

Програми ставали непередбачуваними. І ось, у 1968 році, сталася революція. Великий і жахливий Едсгер Дейкстра (Edsger W. Dijkstra) пише свого знаменитого листа в ACM: *“Go To Statement Considered Harmful”*. Він сказав: “Хлопці, досить стрибати. Нам потрібна структура”.

Ера Структурного програмування та його “Спадщина” (1970-ті)

Так народилося Структурне програмування. Вчені Бом і Якопіні математично довели, що будь-яку програму можна написати, використовуючи всього три конструкції:

1. **Послідовність (Sequence):** Роби А, потім Б.
2. **Розгалуження (Selection):** if / else.
3. **Ітерація (Iteration):** while / for.

Усе. Більше ніяких стрибків GOTO. З’явилися процедури і функції. Це був прорив! Код став читабельним від початку до кінця.

Але тут є нюанс, про який мовчать у підручниках. Структурне програмування принесло нам не тільки добро, а й “архітектурні забобони”, які досі псують життя розробникам.

Одне з правил Дейкстри звучало так: **“One Entry, One Exit”** — один вхід, один вихід.
Для мов на кшталт Pascal чи C, де треба було вручну чистити пам’ять перед виходом із функції, це було критично важливо. Ти увійшов, виділив пам’ять, зробив роботу, очистив пам’ять, вийшов. В одній точці.

Але що ми бачимо сьогодні в Java, C# чи PHP? Розробники, сліпо слідуючи цьому правилу, пишуть “Arrow Code” або “Код-стрілу”, з величезною кількістю вкладеностей.

```java
// Антипатерн Arrow Code (Спадщина "Одного виходу")
public String processOrder(Order order) {
    String result = "Error";
    if (order != null) {
        if (order.isValid()) {
            if (order.hasPayment()) {
                // 100 рядків логіки
                result = "Success";
            }
        }
    }
    return result; // Той самий "єдиний вихід"
}
```

Це антипатерн у сучасному світі!

Сьогодні ми сповідуємо принцип **“Fail Fast”**. Якщо дані невалідні — робимо return одразу, на самому початку методу. У сучасному дизайні це називається **Guard Clauses** (загороджувальні конструкції). Це дозволяє тримати код плоским, чистим і зрозумілим.

```java
// Сучасний підхід: Guard Clauses (Загороджувальні конструкції)
public String processOrder(Order order) {
    if (order == null) return "Error";
    if (!order.isValid()) return "Error";
    if (!order.hasPayment()) return "Error";

    // 100 рядків логіки на першому рівні вкладеності
    return "Success";
}
```

Не треба тягнути виконання до кінця функції заради святого правила сімдесятих. Правило було для іншого контексту.

Ера Процедурного Пекла: Глобальні дані (1980-ті)

З потоком керування розібралися. Але виникла нова проблема. У процедурному програмуванні дані (змінні) і поведінка (функції) були розділені.

Уявіть собі комунальну квартиру.

- Змінні (дані) — це спільний коридор і кухня.
- Функції — це мешканці.

Функція А прийшла, взяла змінну User, змінила їй ім'я. Функція Б прийшла, побачила User, видалила його. Функція В намагається нарахувати User зарплату... і програма падає з помилкою *Null Reference*. Хто винен? Хто змінив дані? Відслідкувати це у великій системі було практично неможливо.

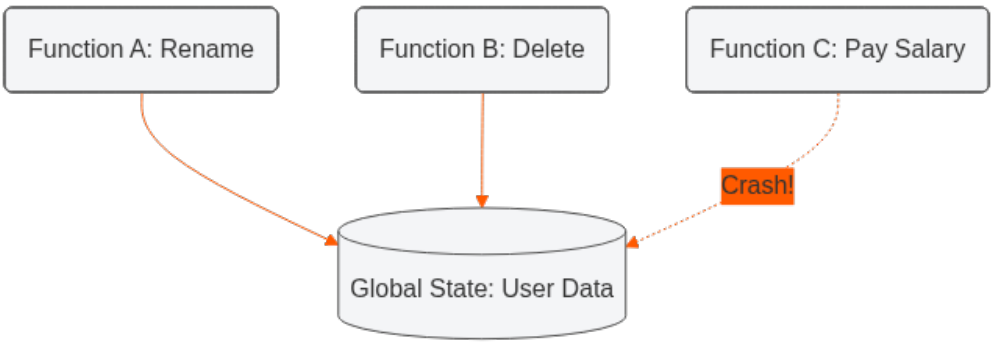

Рисунок 2. Проблема глобального стану (Global State Problem)

Дані були беззахисні. Будь-яка функція могла їх зіпсувати. Нам потрібен був сейф.

Народження ООП: Клітини і Повідомлення

Тут на сцену виходять скандинави Крістен Нюгорд і Оле-Йохан Дал зі своєю мовою Simula 67 (перша мова з об'єктами), і згодом Алан Кей (Alan Kay), батько мови Smalltalk.

Кей був біологом за освітою. Він подивився на клітину живого організму. Клітина має мембрану. Всередині йдуть складні хімічні процеси: мітохондрії, ядро, ДНК. Але ззовні клітина — це просто капсула. Клітини спілкуються одна з одною через обмін сигналами. Клітина А не лізе "руками" всередину Клітини Б. Вона просто надсилає повідомлення.

Так і народилася **Інкапсуляція**.
Ми вирішили: дані — це святе. Ми ховаємо їх всередину об'єкта. Ніхто ззовні не має права їх вільно модифікувати. Хочеш щось зробити? Виклич метод. Надішли об'єкту повідомлення.

Історичний парадокс: Ми перемогли глобальні дані через ООП та інкапсуляцію. Але натомість отримали іншу проблему — **Високу пов'язаність (Coupling)**. Коли C++ захопив світ, люди почали створювати тисячі класів, які залежали один від одного так міцно, що зміна одного файлу ламала половину проєкту.

І ось саме тут, коли класи вже існували, але порядку між ними ще не було, на сцену виходять Архітектори.

Від цегли до байтів: Історія патернів (1977-1994)

Рішення прийшло звідти, звідки не чекали. Від будівельників. Адже архітектори будівель вирішують ту саму задачу: як зі стандартних блоків побудувати щось надійне і довговічне.

У 1977 році Крістофер Александер, професор архітектури будівництва, пише книгу *"A Pattern Language"*. Він усвідомив: проблеми в архітектурі повторюються. Не треба щоразу вигадувати велосипед. Треба описати ці повторювані рішення:

- **Context** (Контекст): Де це відбувається?
- **Problem** (Проблема): Який конфлікт треба вирішити?
- **Solution** (Рішення): Як це зробити перевіреним способом.

Десять років по тому (1987) двоє геніальних програмістів, Кент Бек та Ворд Каннінгем, прочитали Александера і зрозуміли: "У коді ж те саме!". Вони описали кілька перших патернів програмування. Спочатку індустрія сприйняла це скептично. Але у 1994 році стався великий вибух.

Четверо вчених (Еріх Ґамма, Річард Хелм, Ральф Джонсон, Джон Вліссідес) випускають книгу **Design Patterns**. Вони увійшли в історію як **Банда Чотирьох (Gang of Four, GoF)**. Вони описали 23 класичні проблеми і дали їм назви: *Factory, Command, Observer...*

З'явилася спільна мова. Тепер розробник з Києва міг сказати колезі з Каліфорнії: *"Застосуй тут Синглтон"*, і їм не треба було малювати діаграми пів години.

Темний бік патернів

Здавалося б, щасливий кінець? Ми знайшли Святий Грааль? **Ні.**

Універсальна мова принесла нові «хвороби»:

- **Patternitis (Патерніт) та "Дитячі травми":** Спочатку це хвороба новачків. Прочитавши книгу GoF, розробник отримує "молоток", і все навколо починає здаватися йому цвяхом. Він пхає патерни туди, де достатньо одного if/else. Але найгірше починається потім: попекшись на цьому і створивши непідтримуваного монстра, такий розробник доходить висновку, що "патерни — це взагалі маячня". Дослужившись до позиції Тімліда, він через цю свою "дитячу травму" починає категорично забороняти підлеглим використовувати БУДЬ-ЯКІ патерни. Ця крайність дуже часто зустрічається в нашій індустрії.
- **Карго-культ:** Люди копіюють структуру патерну, бо "так круто", але не розуміють проблеми, яку він вирішує.

Джуніори ліпили складені Фабрики там, де треба було просто зробити new. Код ставав надскладним.

Протверезіння: GRASP і SOLID

У 1997 році Крейг Ларман випускає книгу *"Applying UML and Patterns"*. Він подивився на цей хаос і сказав: "Хлопці, ви побігли попереду батька в пекло. Ви вчите складні архітектурні форми (GoF), але ви досі не знаєте базових речей!"

Цікавий факт про книгу Лармана: Це дуже забавна книга сама по собі. Вона напхана абсолютно різнорідними темами, які майже ніяк не пов'язані між собою. У Лармана було багато класних ідей, і він просто "запхав" їх усі в один том, навіть не подбавши про зв'язність. Це часто викликає острах у розробників: людина хоче розібратися в GRASP, купує книгу про UML і патерни, а виявляється, що GRASP — це лише одна з глав, яка стоїть особняком від інших відірваних тем.

Але саме в цій книзі народився **GRASP** (General Responsibility Assignment Software Patterns). Це не про класи. Це про Відповідальність. *У кого є дані, той і робить роботу (Information Expert). Хто об'єкт використовує, той його і створює (Creator).* Це фундамент, без якого ваші Синглтони і Декоратори — лише купа сміття.

Пізніше, на початку 2000-х, з'являється Роберт Мартін (відомий як "Uncle Bob"). Треба відразу розвіяти популярний міф: **такої книги як "SOLID" у природі не існує**.

Насправді, ці п'ять принципів були лише частиною його великої монографії *"Agile Software Development, Principles, Patterns, and Practices"*. Більше того, сам акронім "SOLID" придумав не він, а Майкл Фезерс (Michael Feathers), який помітив, що якщо скласти перші літери п'яти головних принципів Мартіна, вийде красиве слово. Принципів у тій книзі набагато більше, але ми фокусуємось на базовій п'ятірці.

По суті, SOLID — це ваша техніка безпеки, яка гарантує, що зміна одного класу не підірве інший.

Велика Картина (The Big Picture)

Тепер у вас має скластися повний пазл еволюції інженерної думки:

Рисунок 3. Піраміда архітектурного знання

Бачите логіку? Наша піраміда спускається від загальних архітектурних ідей все ближче і ближче "до землі" — тобто до вашого реального коду та серверів:

- На самій вершині абстракції лежить ідеологія **ООП**. Це концепція капсул і повідомлень.
- Спускаючись на крок нижче, ми маємо **GRASP** — це приземлені правила розподілу відповідальності (у який клас покласти метод).
- Ще ближче до реальності знаходиться **SOLID** — "техніка безпеки", яка гарантує гнучкість зв'язків між вашими класами.
- Далі йде **GoF** — це інструменти для вирішення локальних, специфічних мікро-задач "на землі" (як створити об'єкт в конкретних умовах). Важливо розуміти: GoF — це **не** готові рішення для комплексних проблем системи. Це типові "сферичні коні", дуже локалізовані цеглинки.
- І вже після них, на самому фундаментальному рівні системної розробки, починається справжня, доросла **Макро-архітектура** (Enterprise Patterns, мікросервіси, HighLoad) — тобто те, як усі ці "дрібні" абстракції та патерни об'єднуються у величезну працюючу систему.

Більшість програмістів і книг роблять помилку: вони заучують патерни GoF, як вірші, не розуміючи ідеології над ними. Ми ж підемо правильним шляхом: спускатимемося від чистої ідеології крок за кроком, аж поки не опинимося в самій гущі реального Enterprise-коду. І коли ви зрозумієте цей базис, вам відкриється шлях до справжньої комплексної архітектури.

А як щодо Функціонального програмування?

Можливо, ви зараз думаєте: "Сергію, зараз хайп навколо функціонального програмування (ФП). Лямбди, стріми, незмінний стан (Immutability). Кажуть, що ООП застаріло. Чому ж ФП просто не вбило ООП і ці ваші патерни?".

Щоб одразу відбити атаки фанатів чистого ФП (яких у нашій індустрії вистачає), поясню технічні причини. Чисте ФП просто не підходить для реалій більшості сфер розробки (чи то Enterprise, e-commerce, мобільна розробка, фронтенд або геймдев) з трьох фундаментальних причин:

1. **Проблема стану (State).** Користувацький інтерфейс (UI) — це суцільний мінливий стан. Кнопка натиснута або ні, вікно активне або ні, користувач зробив свайп. У чистій ФП-парадигмі стану немає (Immutability). Намагатися малювати UI або обробляти введення від користувача на чистому ФП — це моделювати мінливий світ за допомогою статичних математичних формул. Це страшенно незручно і виглядає як мазохізм.
2. **Проблема послідовності (Sequence).** Класична бізнес-транзакція (відкрили транзакцію -> перевірили баланс юзера -> списали гроші -> оновили статус -> закрили транзакцію) — це імперативний, послідовний алгоритм команд. ФП за своєю природою мислить категоріями *обчислень*, а не *команд*. Тому моделювати послідовні транзакції в чистому ФП — це надзвичайно складна і неприродна задача.
3. **Побічні ефекти та Монади (Side Effects).** Логування, запис у БД, виклики сторонніх API — для чистого ФП це все "побічні ефекти". Чиста функція має лише прийняти дані і повернути результат без зовнішнього впливу. Щоб якось вирішити це обмеження (і щоб програма могла хоч щось записати в базу), фанати ФП придумали обхідний шлях — "Монади" (на кшталт IO Monad). Але писати реальну бізнес-логіку на Монадах — це гарантовано зламати мозок 90% вашої команди розробників.

Тому сучасний світ працює на **гібридній архітектурі**:

1. **Макро-рівень (Бізнес-процеси та UI) — це ООП.** Як тільки нам потрібна послідовна логіка, транзакції, збереження стану або малювання інтерфейсу — ми використовуємо ООП і патерни (контролери, сервіси). Побудувати великий Enterprise суто на функціях — це отримати непідтримуване спагеті з аргументів.
2. **Мікро-рівень (Трансформація даних) — це ФП.** Усередині методів, коли треба просто відфільтрувати колекцію чи знайти потрібний елемент, ми використовуємо map, filter, чисті функції. Тут ФП безумовно перемагає.

Більше того, ФП часто *спрощує* класичні патерни. Наприклад, патерн "Стратегія" в ООП вимагав інтерфейсу і низки класів. З появою лямбд він перетворився на просту передачу функції як параметра. Суть залишилася (ізолювати алгоритм від клієнта) — синтаксис спростився.

Тож ми не будемо воювати. Ми будемо прагматиками. Ми розберемося, як керувати складністю так, щоб код, який ви напишете через рік, все ще хотілося б підтримувати. Далі — більше.

Готові? Рухаємося до Архітектурних драйверів.

🔑 Висновки розділу

Що варто запам’ятати з цієї історії болю:

- **Еволюція — це вирішення попередньої проблеми.** Структурне програмування вбило Spaghetti Code, а ООП перемогло “комуналку” глобальних станів. Не знаючи цього болю, ви не зрозумієте логіку патернів.
- **Сліпе слідування правилам минулого — антипатерн.** Старе правило “одна точка виходу з функції” зараз породжує нечитабельний Arrow Code. Замінюйте його на сучасний Fail Fast (Guard Clauses).
- **Патерн — це інструмент, а не самоціль.** Якщо ви застосовуєте GoF-патерн (“Стратегію”, “Фабрику”) там, де було б достатньо звичайного if/else — ви захворіли на “Патерніт” та займаєтесь овер-інжинірингом.
- **Від абстракції до землі.** Архітектурне мислення будується згори донизу: спочатку розуміння ідеології ООП, потім правила розподілу відповідальності (GRASP), далі техніка безпеки зв’язків (SOLID), потім мікро-рішення (GoF), і лише на найнижчому (найширшому) рівні — Макро-архітектура (Enterprise, мікросервіси та HighLoad).
- **Жодної війни парадигм.** Ми живемо в епоху гібридної архітектури. ООП ідеально працює на макро-рівні (структура модулів і класів), а Функціональне програмування рятує нас від проблем стану (Mutable State) на мікро-рівні всередині функцій.

Розділ 2. За що вам платять гроші? Архітектурні драйвери та мистецтво Trade-offs

ЩО проти ЯК: Функціональні та Нефункціональні вимоги

Функціональні вимоги (FR) — “ЩО”

Нефункціональні вимоги (NFR) — “ЯК”

Сучасна шістка драйверів

1. Maintainability (Підтримуваність)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

2. Scalability (Масштабованість)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

3. Reliability (Надійність)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

4. Security (Безпека)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

5. Testability (Тестованість)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

6. Observability (Спостережуваність)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Мистецтво компромісу (Trade-offs)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Performance vs Maintainability (Продуктивність проти Підтримуваності)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Flexibility vs Time-to-Market (Гнучкість проти Швидкості запуску)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Теорема CAP

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Жорсткі обмеження (Constraints)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

🔑 Висновки розділу

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Розділ 3. Парадокс ООП: Ви думаєте, що знаєте, — але ні

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Що таке парадигма?

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Швидкий огляд «сусідів» ООП

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Функціональне програмування (ФП)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Аспектно-орієнтоване програмування (АОП)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Реактивне програмування

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Алан Кей та Біологічна метафора

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Messaging: Прохання замість Наказу

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Стан + Поведінка = Нероздільна Єдність

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Антипатерн: Анемічна Модель (Anemic Domain Model)

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Велика картина: де ми?

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

🔑 Висновки розділу

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Розділ 4. Інкапсуляція: Два обличчя “чорної скриньки”

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Велика ілюзія безпеки

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Дві трактовки: Від Сімули до Вікіпедії

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Інкапсуляція як архітектурний принцип

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Навіщо це треба: Гайка, Жопа та Race Conditions

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Рішення: Справжній ООП-об’єкт

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

🔑 Висновки розділу

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Частина II. GRASP: Мистецтво розподілу відповідальності

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Частина III. SOLID: Принципи гнучкості

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.

Частина IV. Патерни GoF (Банда Чотирьох) у сучасному світі

Цей вміст недоступний у демонстраційній книзі. Книгу можна придбати на Leanpub за адресою https://leanpub.com/architecture-foundation.