ФУНДАМЕНТ АРХІТЕКТУРИ

БАЗА, ЯКОЇ ВАС НІКОЛИ НЕ ВЧИЛИ

СЕРГІЙ
НЕМЧИНСЬКИЙ

Зміст книги

** Примітка: у цьому безкоштовному примірнику включено Передмову та Розділ 1.*

Передмова

Передмова: Чому ця книга існує і як її читати у фрагменті

Частина I. Фундамент та філософія

Розділ 1. Еволюція болю: Від GOTO до ООП у фрагменті

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

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

Розділ 4. Інкапсуляція: Два обличчя "чорної скриньки"

Розділ 5. Наслідування: коли світ б'є вас по потилиці

Розділ 6. Поліморфізм: вбивця if-else та справжнє серце ООП

Розділ 7. Абстракція: мистецтво ігнорувати деталі

Розділ 8. UML: Навіщо малювати, якщо можна кодити?

Розділ 9. Анатомія класу та битва стрілок: Class Diagram

Розділ 10. Життя об'єктів у рантаймі: Sequence Diagram

Частина II. GRASP: мистецтво розподілу відповідальності

Розділ 11. Вступ до GRASP та патерн Information Expert

Розділ 12. Creator (Творець): Хто дає життя об'єктам?

Розділ 13. Controller (Контролер): Перша лінія оборони

Розділ 14. Low Coupling та High Cohesion: Інь і Янь архітектури

Розділ 15. Polymorphism: Вбивця IF/ELSE

Розділ 16. Pure Fabrication: Чиста Вигадка

Розділ 17. Indirection: Посередник, що Розв'язує Руки

Розділ 18. Protected Variations: Як Локалізувати Вибух

Частина III. SOLID: принципи гнучкості

Розділ 19. Вступ до SOLID та Принцип SRP

Розділ 20. OCP: Відкриті для розширення, закриті для змін

Розділ 21. LSP: Принцип підстановки Лісков

Розділ 22. ISP: Інтерфейс належить клієнту

Розділ 23. DIP: Бізнес не повинен знати про MySQL

Частина IV. Патерни GoF у сучасному світі

Розділ 24. Вступ до GoF: Патерни як інструменти, а не релігія

Розділ 25. Singleton (Одинак): Чому глобальний стан вбиває ваші тести

Розділ 26. Factory Method та Abstract Factory

Розділ 27. Prototype та Object Pool

Розділ 28. Builder (Будівельник)

Розділ 29. Структурні патерни-посередники: Adapter, Proxy, Decorator, Facade

Розділ 30. Bridge (Міст): Порятунок від комбінаторного вибуху

Розділ 31. Composite та Flyweight

Розділ 32. Strategy: Король патернів

Розділ 33. Mediator та Observer

Розділ 34. Chain of Responsibility: Конвеєри обробки та Middleware

Розділ 35. Memento, Command та Interpreter

Розділ 36. Iterator: Ховаємо кишки колекцій

Розділ 37. State: Кінець епохи нескінченних switch

Розділ 38. Template Method: Скелет алгоритму та пастки наслідування

Розділ 39. Visitor: Подвійна диспетчеризація

Частина V. Епілог

Розділ 40. Епілог: Design Patterns у сучасному світі. Як ШІ змінює розробку

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. І я в айтішці з 1996 року, відколи закінчив університет. З того часу я багато де встиг попрацювати — 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 проєктним менеджером у Ciklum, архітектором у Luxoft і тімлідом майже в кожній із цих компаній. Тому процес розробки я бачив і з «окопів» програміста, і з крісла стратега, і з позиції бізнесу.

Впродовж усієї цієї кар'єри я 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 усвідомив, що вони… майже нічого не знають.

Коли я запитував: «А що ж ви вивчали на ваших курсах?», вони з гордістю відповідали: «Ми робили бульбашкове сортування й рахували числа Фібоначчі!». Але варто було запитати: «А Spring у вас був?», як новачки «сипались» і робили при цьому круглі очі: «Ні, а що таке Spring?». Мене це дико бісило.

А все тому, що ці курси за три місяці намагалися втиснути цілу п'ятирічну університетську програму, викидаючи при цьому все те, що реально потрібно програмісту на роботі: фреймворки, патерни, enterprise-підходи. Я хапався за голову і врешті-решт вирішив: «Все, досить цієї дичини. Я буду навчати 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» — саме цей курс і ліг в основу книги, яку ви зараз читаєте. Згодом я додав до програм просунутий курс з enterprise-патернів (про нього планую написати наступну книгу), а також курс «Ефективний тімлід». Спостерігаючи за своїми студентами та за індустрією загалом, я зрозумів,

що сучасна освіта для досвідчених розробників має три фундаментальні проблеми. Які мене люто дратують.

Біль №1: знання у вакуумі

Якщо сучасний програміст хоче розібратися в архітектурі, то запросто відкриє для себе аж цілу гору книжок: від класичної «Банди Чотирьох» до новинок, що виходять ледь не щотижня. Але всі вони «хворіють знаннями у вакуумі».

Проблема в тому, що ці книжки розповідають про архітектурні концепції ізольовано одна від одної. Якщо книжка про SOLID, вона жодним словом не згадає про GoF-патерни, ніби їх не існує. Якщо ж це книжка про GRASP, то, окрім GRASP, там теж не скажуть, як саме він перетинається з іншими підходами. У нещасних студентів виникає суцільний когнітивний дисонанс: вони бачать одну абревіатуру, інтуїтивно відчувають, що вона пов'язана з іншою, але НІХТО не пояснює їм як саме. Всі ці знання нагромаджуються в голові, як детальки від конструктора, що ніяк не з'єднуються докупи правильно.

Біль №2: застаріле лайно мамонта

Це просто якесь жахіття. Більшість із того, що сьогодні викладають про GoF-патерни, — це досі «законсервований» переказ підходів 30-річної давнини. Всі курси та навчальні матеріали, які я бачив, слово в слово повторюють ту саму книгу «Design Patterns» 1994 року. Ні, ну серйозно?! Минула третина століття!

Вони використовують ті самі діаграми класів, що й у 94-му. Наприклад, патерн Factory Method в тому вигляді, як його «малюють» в класичних діаграмах, вже давно «вимер» в реальному продакшені. Так код ніхто не пише. Ба більше, так писати не можна, бо це не вирішує жодної проблеми!

У результаті розробник дивиться на UML-діаграму[1] в книжці, а потім — на код у своєму проєкті, бачить, що вони не збігаються, й вирішує: «Ці ваші патерни — застаріле лайно мамонта». А це не так! Патерни еволюціонували. Тому в цій книзі я буду «єретиком» і розповідатиму, як сьогодні їх насправді використовують у розробці.

Біль №3: сліпа віра в ШІ

Багато хто вважає, що відколи є штучний інтелект, архітектуру можна не вчити: «Він же сам напише код, ще й так самовпевнено й переконливо!». Ні. ШІ зовсім

не скасовує потребу розуміти архітектуру — навпаки, він робить її ще більш важливою.

Якщо ви не знаєте китайської, будь-який переклад від ШІ буде виглядати правильним. Так само й тут: якщо ви не розумієте архітектури, будь-який «витвір» штучного інтелекту здаватиметься вам нормальним кодом. І якщо на цей ваш код погляне досвідчений архітектор, він одразу зрозуміє: перед ним — суцільне лайно.

А все тому, що ШІ поки що не вміє системно мислити. Й, напевно, ніколи не буде вміти. Тому єдине, що робить нас кращими за LLM[2] — це наше критичне мислення. А воно неможливе без фундаменту й принципів.

Ви повинні володіти принципами принаймні для того, щоб сказати ШІ: «Ні, так не роби. Роби ось так». Бо якщо ви самі не розумієте архітектурних підходів, то не зможете пояснити штучному інтелекту, чому ці його рішення, навіть такі переконливі, — насправді погана ідея.

Для кого ця книга

Ця книга створена для:

- Мідл- та сеньйор-розробників, які відчувають, що їхній код починає обростати хаосом, хочуть мислити архітектурно та нарешті навчитися бачити «загальну картину».
- Амбітних джунів, які не збираються роками писати лайнокод без розуміння, куди рухатись далі, а прагнуть одразу почати розбиратися в архітектурі.
- Та всіх тих, хто прочитав книгу GoF (до речі, вона дуже нудна, нікому не раджу її читати), «нахапався» фрагментарних знань і тепер намагається 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 у вашій голові склалася єдина система координат, яка відрізняє хороший код від поганого.

Як ми будемо вчитися: Problem-First

Я інженер, а не теоретик. Будь-яке рішення в розробці для мене має сенс тільки тоді, коли воно розв’язує конкретну проблему. Робити щось «бо це красиво» — шлях в нікуди. Тому мій підхід у цій книзі — це Problem-First: спочатку ми на атоми розбираємо проблему і тільки потім застосовуємо патерн як

рішення. І, звісно ж, приклади будуть з сучасного життя, а не «сферичні коні» з 90-х.

Також я обіцяю не мучити вас зайвим «архітектурним шумом» — тобто десятками сторінок зашифрованих UML-позначень, стрілочок і кружечків, з яких починається багато підручників. Я дам вам лише те, що реально використовується в сучасній розробці.

I> **Щодо мови прикладів:** більшість сніпетів коду в цій книзі написані мовою Java (або дуже наближеним до неї псевдокодом). Чому? По-перше, це мова, якій я присвятив понад 15 років свого професійного життя. По-друге, Java та C# — це своєрідна «латина» об'єктно-орієнтованого програмування. Їхній синтаксис легко читають розробники на TypeScript, PHP, Python, Swift та C++. Я хочу, щоб ви зосередилися на архітектурі, а не витрачали сили на парсинг незвичних конструкцій. Водночас там, де патерн зав'язаний на екосистему (наприклад, ізоляція запитів у PHP-FPM чи Event Loop у JavaScript), я буду наводити приклади відповідними мовами. Але пам'ятайте: фундаментальні принципи ООП від синтаксису не залежать!

I> **Важливо для читачів електронної версії (EPUB):** у книзі — багато блоків коду. Щоб читати було комфортніше й збереглося правильне форматування (відступи), я наполегливо рекомендую використовувати стандартні додатки, такі як Google Play Books, Apple Books, FBReader або Moon+ Reader. Уникайте додатків для паралельного читання та перекладу (наприклад, Smart Book), оскільки вони «розбивають» код на окремі слова й повністю ламають його структуру.

Структура: куди ми йдемо

Ми маємо застекувати всі ваші знання:

1. Згадаємо базовий фундамент ООП (більшість пам'ятає його зовсім не так, як він працює в реальному житті).
2. Розберемо GRASP — правила розподілу відповідальності (бо це основа проєктування систем).
3. Поєднаємо все це з «технікою безпеки» — принципами SOLID.
4. Розглянемо сучасні інструменти мікрорівня — GoF-патерни.
5. Визначимо наступні кроки — куди вам рухатися й що з усім цим робити далі.

Подяки

Ця книга навряд чи з’явилася б на світ, якби не мої Early Access читачі — ті, хто купив і читав її ще під час написання. Це просто вогонь! Ви мене дуже надихали.

Я людина відповідальна, але розумію: якби я вирішив писати «в шухляду» одразу всю книгу від початку до кінця, я б її не закінчив ніколи. Як директор компанії, я б завжди знаходив інші, «більш важливі» справи. Але той факт, що ви придбали книжку наперед і чекали на нові розділи, став для мене залізобетонним стимулом: «я точно її завершу!». Така вже я людина 😄.

Окрема подяка моїй дружині Наталі — без її віри та колосальної підтримки ця книга так і залишилася б «тією ідеєю, за яку я колись обов’язково візьмусь». Вона не тільки надихала мене, а й у прямому сенсі змушувала сідати за книгу, бо ж розділ сам себе не напише.

Величезне дякую моїй доньці Ані. Вона у нас професійна художниця, тож всі ці чудові ілюстрації, включно з обкладинкою, — її робота. Завдяки їй у книги з’явився власний візуальний стиль.

Величезна подяка моїй редакторці Тані Ворончук. Вона взяла на себе титанічну працю: вичитувати мій інженерний «потік свідомості», виловлювати описки й перетворювати емоційні пасажі на легкий, вивірений та структурований текст. Дякую за залізне терпіння й за те, що допомогла зберегти той самий "вулично-інженерний" дух книги!

Особлива подяка Євгенію Піддубному — найактивнішому рев'юеру цієї книги, який вніс найбільше пропозицій та правок.

Також дякую всім тим, хто ділився своїм фідбеком. Саме ваші відгуки допомогли мені структурувати й відшліфовувати книгу так, щоб вона була максимально корисною та вивіреною для всіх читачів.

Люблю вас. Ну що ж, погнали 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 ізольовано один від одного. ООП, GRASP, SOLID та GoF — це єдина взаємопов’язана система.

- Problem-First: кожен патерн — це інструмент для 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.

Розділ 1. Еволюція болю: від GOTO до ООП

Надворі 2026-й. 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 софтвер-архітекторам, а сеньйорів все ще ганяють по «древніх» патернах на співбесідах? І чому проєкти, які починаються як «цукерочка», через рік стають непідтримуваним пеклом, яке всі бояться чіпати?

Штучний інтелект генерує код блискавично. Але без розуміння патернів ви навіть не зможете оцінити, що саме він вам видав: чисту архітектуру й адекватний дизайн чи нечитабельне місиво на двісті рядочків.

А все тому, що за цими фреймворками, хмарами й розумними технологіями все ще стоїть одна й та сама проблема. Складність.

Ми десятиліттями боролися з хаосом у розробці. Й для того, щоб ви нарешті зрозуміли, навіщо потрібні правила (такі як General Responsibility Assignment Software Patterns, GRASP (шаблони розподілу обов'язків) чи SOLID (п'ять принципів проєктування гнучкого коду)), доведеться пройти весь цей шлях від самого початку — через історію підходів, проблем та рішень. Повірте, якщо ви не знаєте, як боляче б’ють граблі, на які наступали до вас, ви гарантовано наступите на них сьогодні.

Тож «відмотаємо» до самих витоків, коли програмісти ще були справжніми ковбоями, а пам’ять комп’ютера вимірювалася в кілобайтах.

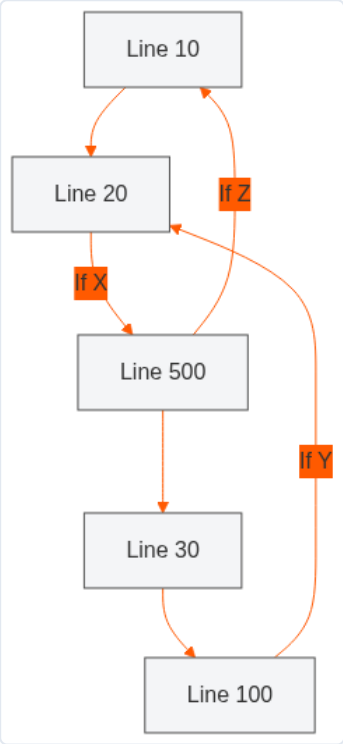
Ера хаосу: Spaghetti Code та GOTO (1960-ті)

Ось ми в шістдесятих. Пишемо на асемблері, Fortran або ранньому BASIC. У нас немає функцій в сучасному розумінні. Немає циклів while чи for. У нас є лише одна «ковбойська» суперзброя: оператор GOTO.

А з ним 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 породило явище під назвою Spaghetti Code (спагеті-код). Якщо намалювати потік виконання програми стрілочками на папері, ці лінії переходів виглядатимуть як велика тарілка макаронів, де важко знайти початок і кінець. Тому зрозуміти логіку, просто читаючи код зверху вниз, було неможливо — доводилось тримати в голові всю карту цих заплутаних зв'язків.



Структура потоку виконання Spaghetti Code

Такий код неможливо було масштабувати. Мозок не справлявся з хаосом, і розробка великих систем перетворювалася на замкнене коло, де кожен виправлений баг створював десяток нових.

У 1968 році стався злам підходу. Великий і жахливий[3] Едсгер Дейкстра (Edsger W. Dijkstra) пише свого знаменитого листа в ACM[4]: «Go To Statement Considered Harmful». Фактично він заявив: «Хлопці, досить стрибати. Нам потрібна структура».

Це й дало початок структурному програмуванню, а розробники розклали хаотичні «макарони» в зручні «порційні контейнери».

Ера структурного програмування та його «спадщина» (1970-ті)

Вчені Коррадо Бем (Corrado Böhm) та Джузеппе Якопіні (Giuseppe Jacopini) математично довели, що будь-яку програму можна написати, використовуючи всього три конструкції:

1. **Sequence (послідовність)**: роби А, потім Б.
2. **Selection (розгалуження)**: if / else.
3. **Iteration (ітерацію)**: while / for.

Усе. Більше ніяких стрибків GOTO. Об’єднання цих трьох конструкцій із процедурами та функціями, які дали можливість «запакувати» логіку в окремі блоки, стало справжнім проривом. Код став читабельним від початку й до кінця.

I> **Історична довідка:** цікаво, що сам інструмент — subroutines (підпрограми), або окремі блоки коду для перевикористання, — з'явився задовго до формування структурної парадигми. Складність була в тому, щоб повернути виконання коду саме туди, звідки була викликана підпрограма, яка вже відпрацювала. Щоб розв’язати проблему, британський інженер Девід Вілер (David Wheeler) ще в 1949 році на комп'ютері EDSAC вигадав трюк, відомий як Wheeler Jump, або стрибок Вілера: програма перед викликом сама записувала адресу повернення безпосередньо в останню інструкцію підпрограми. За це Вілер отримав перший у світі ступінь Ph.D. з комп'ютерних наук, захистивши дисертацію про автоматичні обчислення на EDSAC. Пізніше, у 1960 році, під час проєктування мови Algol 60, процедури отримали сучасний вигляд: локальні змінні на стеку та підтримку рекурсії. Проте розробники все ще продовжували писати код як заманеться, хаотично використовуючи GOTO разом із процедурами. І тільки в 1970-х структурне програмування закріпило їх як обов'язковий стандарт організації коду, остаточно витіснивши GOTO.

Але тут є нюанс, про який мовчать у підручниках. Структурне програмування не тільки полегшило розробку, але й залишило нам у спадок «архітектурні забобони», які досі псують життя розробникам.

Одне з правил Дейкстри звучало так: «One Entry, One Exit» — один вхід, один вихід.

Для сучасного розробника вимога One Entry (одного входу) звучить як абсурд — бо як взагалі можна викликати функцію не з початку? Але в 60-х та 70-х роках на асемблері чи Fortran це було буденною справою. Програміст міг спокійно взяти `GOTO` і стрибнути прямо в середину тіла функції, оминаючи ініціалізацію

локальних змінних. Код перетворювався на нечитабельний жах, а програма падала через кашу в регістрах пам'яті. Тому Дейкстра наполягав: увійти у функцію можна тільки через «парадні двері» — її початок.

З One Exit (одним виходом) історія інша. Для мов на кшталт Pascal чи C, де треба було руками чистити пам’ять перед виходом із функції, це мало критичне значення. 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 тягнуть це правило в сучасні мови буквально й, сліпо йому слідуючи, загортають ключовий функціонал в Arrow Code, або код-стрілу, з величезною кількістю вкладеностей.

```
// 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 в сучасній розробці!

Такий підхід ховає головну бізнес-логіку на самому «дні» вкладеностей — там, де її важко побачити й легко зламати.

Сьогодні ми сповідуємо принцип Fail Fast. Якщо дані невалідні — робимо return одразу, на самому початку методу. В сучасному дизайні це називається Guard Clauses (загороджувальні конструкції) та зберігає код плоским, чистим і зрозумілим.

```
// 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. Те правило було для іншого контексту.

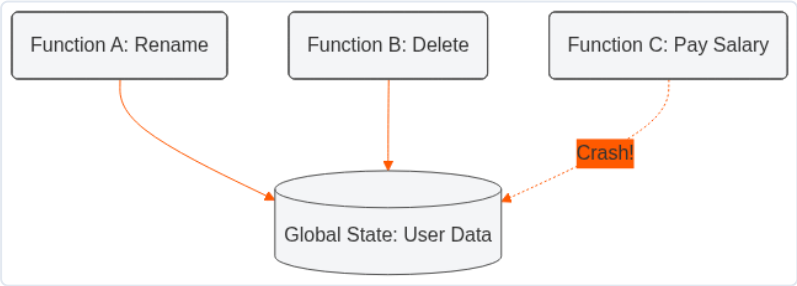
Хоча структурне програмування й розв'язало проблему з хаотичним потоком — тепер код рухався зверху вниз через if / else, цикли та функції — сама природа програм залишалася процедурною. Функції та дані продовжували існувати окремо. Тому чим більшими ставали проєкти, тим більше функцій ділили між собою спільні змінні та глобальний стан. Хаос нікуди не зник — він «переїхав» від керування потоком до керування даними.

Ера процедурного пекла: глобальні дані (1980-ті)

Перенесемося на десятиліття вперед, у вісімдесяті. Тогочасна архітектура нагадувала комунальну квартиру, в якій:

- **Змінні (дані)** — це речі у спільному коридорі та кухні.
- **Функції** — це мешканці.

Функція А прийшла, взяла змінну User, змінила їй ім'я. Функція Б прийшла і вирішила, що User більше не потрібен та видалила його. Функція В намагається нарахувати User зарплату... і програма падає з помилкою Null Reference. Хто винен? Хто змінив дані? Відстежити було практично неможливо. Коли системи виростали до сотень функцій та модулів і великих команд розробників, ситуація виходила з-під контролю. Будь-яка зміна в одному модулі відгукувалася несподіваними багами в іншому.



Global State Problem (проблема глобального стану)

В проєкті не існувало безпечних меж відповідальності. Потрібен був спосіб зачинити двері — дати кожному мешканцю окрему кімнату з замком та заборонити чіпати чужі речі.

Народження ООП: клітини й повідомлення

У цей час на горизонті індустрії з'являються скандинави Крістен Нюгорд (Kristen Nygaard) і Оле-Йохан Дал (Ole-Johan Dahl) зі своєю мовою Simula 67 (перша мова з об'єктами). А згодом — й Алан Кей (Alan Kay), «батько» мови Smalltalk.

Як біолог за освітою, Кей надихався живими клітинами. Кожна з них була для нього окремим мікросвітом. З мембраною, мітохондріями, ядром, ДНК та складними хімічними процесами всередині. Проте ззовні цей мікросвіт — просто капсула, що «спілкується» з іншими клітинами через обмін сигналами. Клітина А не має прямого доступу до внутрішньої «кухні» клітини Б. Вона лише надсилає «повідомлення».

З цієї ідеї і народилася інкапсуляція.

Відтепер дані — це святе. Ми ховаємо їх всередину об'єкта. Ніхто ззовні не має права модифікувати їх так, як йому заманеться. Хочеш щось зробити? Виклич метод. Надішли об'єкту «повідомлення».

I> **Історичний парадокс:** ми приборкали глобальні дані через ООП та інкапсуляцію. Але натомість отримали іншу проблему — Coupling, або жорстку залежність. Коли C++ «захопив світ», усі кинулися створювати тисячі класів, які були настільки сильно пов'язані між собою, що зміна одного файлу ламала половину проєкту.

І чим більшою ставала система, тим більше таких взаємодій накопичувалося. Ніхто вже не міг безкарно залізти в приватні дані об'єкта `Employee`, але щоб просто надрукувати платіжку, клас `SalaryCalculator` мусив жорстко залежати від конкретного драйвера матричного принтера `EpsonFX80` та файлової системи. Варто було оновити модель принтера — і розробнику доводилося переписувати логіку розрахунку податків, а C++ компілятор через Header Hell (пекло взаємних інклудів) йшов перезбирати весь проєкт на пів години.

Залишалося питання: якщо класи вже існують, то за якими правилами вони мають взаємодіяти між собою? Рішення надійшло звідти, звідки не чекали.

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

У 1977 році Крістофер Александер (Christopher Alexander), професор архітектури будівництва, пише книгу «A Pattern Language». Він зрозумів, що проблеми в архітектурі повторюються. Тож не треба щоразу вигадувати велосипед, якщо можна описати повторювані рішення:

- **Context (контекст)**: де це відбувається?
- **Problem (проблема)**: яку типову ситуацію треба вирішити?
- **Solution (рішення)**: як це зробити перевіреним способом.

А коли десять років по тому (1987) двоє геніальних програмістів, Кент Бек (Kent Beck) та Ворд Каннінгем (Ward Cunningham), прочитали Александера — їх осяяло: «В коді ж те саме!». Архітектори будівель, по суті, розв'язують ту ж задачу: як зі стандартних блоків побудувати щось надійне й довговічне. Саме Бек та Каннінгем й описали декілька перших патернів програмування. Спочатку індустрія сприйняла це скептично. Проте в 1994 році стався переломний момент.

Четверо вчених — Еріх Ґамма (Erich Gamma), Річард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) та Джон Вліссідес (John Vlissides) — випускають книгу Design Patterns та входять в історію як Банда Чотирьох (Gang of Four, GoF). У ній вони сформували 23 класичні проблеми проєктування й дали їм назви: Factory, Command, Observer…

З'явилася спільна мова. Тепер розробник з Києва міг сказати колезі з Каліфорнії: «Застосуй тут синглтон» — без малювання діаграм і зайвих пояснень на пів години.

Темний бік патернів

Здавалося б, щасливий кінець. Ми знайшли Святий Грааль. Проте ні, не цього разу.

Універсальна мова принесла з собою 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: обпікшись на цьому й створивши непідтримуваного монстра, такий розробник вирішує, що «патерни — це взагалі маячня». Допрацювавшись до позиції тімліда, через цю свою «дитячу травму» він починає категорично забороняти підлеглим використовувати БУДЬ-ЯКІ патерни. Ця крайність дуже часто трапляється в нашій індустрії.
- **Синдром карго-культу**: багато хто копіює структуру патерну, бо «так круто», але не розуміє проблеми, яку він вирішує.

Джуніори ліпили складні фабрики там, де вистачило б звичайного new. Код ставав надскладним.

Протверезіння: GRASP і SOLID

У 1997 році Крейг Ларман (Craig Larman) видає книгу «Applying UML and Patterns». Він бачив, до чого котиться хаос навколо GoF-патернів, і спробував повернути галузь до здорового глузду. Ідея Лармана зводилася до простого: «Хлопці, ви побігли поперед батька в пекло. Ви вчите складні архітектурні форми (GoF), але досі не знаєте базових речей!»

I> **Цікавий факт про книгу Лармана:** по суті, ця книга — ще той «атракціон». У ній намішані абсолютно різнорідні теми, які майже ніяк не пов'язані між собою. У Лармана було багато класних ідей, і він просто втиснув їх усі в один том, навіть не подбавши про зв'язність. Це часто викликає острах у розробників: хочеш розібратися в GRASP, купуєш книгу про UML і патерни, а виявляється, що GRASP — це лише один із розділів, відірваний від решти ізольованих тем.

Але саме ця книга й дала світові концепцію GRASP, яка ставить у центр розробки не класи, а відповідальність в системі. У кого є дані, той і робить роботу — це патерн Information Expert (Інформаційний Експерт). Хто об'єкт використовує, той його і створює — це патерн Creator (Творець). Це фундамент, без якого ваші синглтони й декоратори — лише купа сміття.

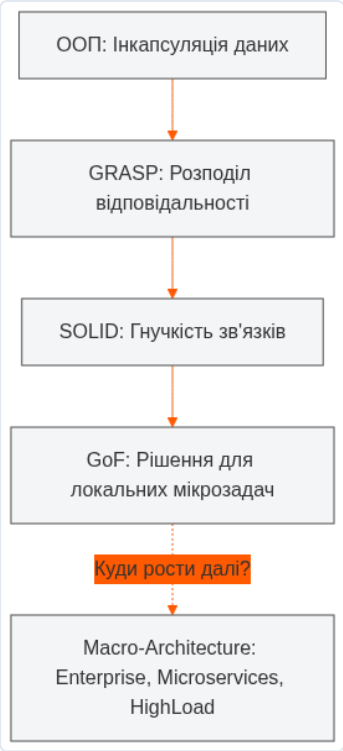
Пізніше, на початку 2000-х, з'являється Роберт Мартін (Robert C. Martin), відомий як Uncle Bob. Він шукав свої відповіді на архітектурні болі індустрії, і знайшов їх у правилах гнучкого проєктування. Тому сьогодні його ім'я намертво зрослося з абревіатурою SOLID. Тут відразу треба розвіяти популярний міф: окремої книги «SOLID» ніколи не існувало.

Більше того, сам акронім SOLID вигадав не Роберт Мартін, а Майкл Фезерс (Michael Feathers), який помітив, що якщо скласти перші літери п'яти головних принципів Мартіна з його монографії «Agile Software Development, Principles, Patterns, and Practices», вийде красиве слово. Принципів у ній набагато більше, але ми фокусуємось на базовій п'ятірці.

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

The Big Picture (Велика картина)

Тепер у вас має скластися цілісний пазл еволюції інженерної думки:



Піраміда архітектурного знання

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

- **На самій вершині абстракції** лежить ідеологія ООП і три її принципи: інкапсуляція, поліморфізм та наслідування.
- **Кроком нижче** ми маємо GRASP — це приземлені правила розподілу відповідальності (у який клас покласти метод).
- **Ще ближче до реального коду** знаходиться SOLID — «техніка безпеки», яка гарантує гнучкість зв’язків між вашими класами.
- **Далі йде GoF** — шаблони для вирішення локальних, специфічних мікрозадач (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 макроархітектура (ентерпрайз-патерни, мікросервіси, хайлоад) — тобто те, як усі ці «дрібні» абстракції та патерни об’єднуються у величезну живу систему.

Більшість програмістів роблять одну й ту ж помилку: вони завчають патерни GoF, як вірші, не розуміючи їхньої ідеології. І більшість книжок тільки підсилюють такий ефект. Ми ж підемо правильним шляхом: спускатимемося від чистої ідеології крок за кроком, аж поки не опинимося в самій гущі реального

ентерпрайз-коду. І коли ви зрозумієте цей базис, вам відкриється шлях до справжньої комплексної архітектури.

А як щодо функціонального програмування?

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

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

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

Тому сучасна розробка працює на гібридній архітектурі:

1. Макрорівень (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 ентерпрайз суто на функціях — це отримати непідтримуване спагеті з аргументів.
2. Мікрорівень (трансформація даних) — це ФП. Усередині методів, коли потрібно лише 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 Strategy в ООП вимагав інтерфейсу й 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 за «правильну» парадигму. Ми будемо прагматиками. Ми розберемося, як керувати складністю так, щоб код, який ви напишете, через рік усе ще хотілося підтримувати.

Сучасність: як виглядають патерни зараз?

Раніше, у дев'яностих, патерни GoF жили безпосередньо у вашому коді. Ви щоразу писали масивні класи ітераторів, обсерверів і багатоповерхових фабрик вручну.

Сьогодні ситуація докорінно змінилася: більшість фундаментальних патернів «переїхали під капот» сучасних фреймворків та мов програмування:

- **Iterator**: вам більше не треба створювати клас ітератора власноруч. Ви пишете foreach в C# або for..of у JavaScript. Патерн вшитий безпосередньо в синтаксис мови.
- **Observer**: в екосистемах React, Vue чи Angular це основа реалізації реактивності (Hooks, Signals). У бекенд-фреймворках (наприклад, Laravel або Spring) Observer «чудово почувається» в системах Events і Listeners. Де-факто, ви використовуєте цей патерн щодня, навіть не замислюючись про його GoF-коріння.
- **Singleton та Prototype**: у Spring Framework ви просто ставите анотацію `@Scope("singleton")` над біном, і Inversion of Control, IoC-контейнер (контейнер інверсії керування) сам керує життєвим циклом єдиного об'єкта системи.
- **Decorator**: у Python і TypeScript цей складний структурний патерн став звичайною синтаксичною конструкцією `@decorator`.

I> **Важливо:** може здатися, що патерни більше не потрібні, адже розробники фреймворків уже все написали за вас. Але насправді вам треба знати патерни не

для того, щоб щоразу збирати їх з нуля (хоча іноді для складної бізнес-логіки це необхідно). Ви мусите їх знати, щоб бачити архітектурну суть за синтаксичним цукром сучасних інструментів. Без розуміння логіки патерну ви ніколи повноцінно не збагнете, чому ваш фреймворк поводиться саме так.

Готові зануритись у цю глибинну суть? Тоді перегортайте сторінку — 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 (Strategy, Factory) там, де достатньо звичайного if/else — ви «захворіли на патерніт» та займаєтесь оверінжинірингом.
- **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 мікрорішення (GoF) і лише на найнижчому (найширшому) рівні — макроархітектура (Enterprise, мікросервіси та HighLoad).
- **Жодної війни парадигм**. Ми живемо в епоху гібридної архітектури. ООП ідеально працює на макрорівні (структура модулів і класів), а функціональне програмування рятує нас від проблем Mutable State (змінного стану) на мікрорівні всередині функцій.
- **Знаходьте архітектурну суть**. Багато класичних патернів (Iterator, Singleton, Observer) зараз «вшиті» безпосередньо в синтаксис мов або заховані «під капотом» фреймворків (Hooks, анотації `@Scope`, Dependency Injection, DI-контейнери (контейнери впровадження залежностей)). Ми вивчаємо патерни не для того, щоб писати їх з нуля, а щоб розуміти справжню логіку роботи наших сучасних інструментів.

---

1. **UML** (Unified Modeling Language) — уніфікована мова моделювання; графічна мова опису, візуалізації та проєктування програмних систем. ↩

2. **LLM** (Large Language Model) — велика мовна модель; архітектура штучного інтелекту, що спеціалізується на обробці та генерації тексту й коду. ↩

3. Дейкстру називали «великим і жахливим» через його безкомпромісний характер та гострий язик. Лауреат премії Тюрінга, творець структурного програмування та автор легендарного алгоритму пошуку найкоротшого шляху, він нещадно критикував індустрію та популярні мови програмування. Його вислів про те, що «навчання мови BASIC систематично калічить розум студентів...», став культовим. ↩

4. ACM (Association for Computing Machinery) — Асоціація обчислювальної техніки, найстаріша та найбільша у світі міжнародна наукова й освітня організація в галузі комп'ютерних наук. ↩