

Занурення в **ПАТЕРНИ ПРОЕКТУВАННЯ**



Олександр Швець

Занурення в **ПАТЕРНИ** **ПРОЕКТУВАННЯ**

v2021-2.33

ДЕМО-ВЕРСІЯ

Придбати повну книгу:

<https://refactoring.guru/uk/design-patterns/book>

Замість копірайту

Привіт! Мене звуть Олександр Швець, я автор книги Занурення в Патерни, а також онлайн-курсу Занурення в Рефакторинг.



Ця книга призначена для вашого особистого користування. Будь ласка, не передавайте її третім особам, за винятком членів своєї сім'ї. Якщо ви хочете поділитися книгою з друзями чи колегами — придбайте і подаруйте їм легальну копію книги. Також ви можете придбати корпоративну ліцензію для всієї вашої команди або організації.

Всі гроші, отримані з продажу моїх книг і курсів, ідуть на розвиток Refactoring.Guru. Це один з небагатьох ресурсів програмістської тематики, доступних українською мовою. Кожна придбана копія продовжує життя проекту й наближає момент виходу нового курсу чи книги.

© Олександр Швець, Refactoring.Guru, 2021

✉ support@refactoring.guru

🖼 Ілюстрації: Дмитро Жарт

🇺🇦 Переклад українською: Віталій Гальцев, Олександр Швець

✎ Редактор: Ельвіра Мамонтова

*Присвячую цю книгу своїй дружині Марії,
без якої я б не довів діло до кінця ще
років тридцять.*

Зміст

Зміст.....	4
Як читати цю книгу.....	6
ВСТУП ДО ООП.....	7
Згадуємо ООП.....	8
Наріжні камені ООП.....	12
Відносини між об'єктами.....	19
ОСНОВИ ПАТЕРНІВ.....	25
Що таке патерн?.....	26
Навіщо знати патерни?.....	30
ПРИНЦИПИ ПРОЕКТУВАННЯ.....	31
Якості хорошої архітектури.....	32
Базові принципи проектування.....	36
§ Інкапсулюйте те, що змінюється.....	37
§ Програмуйте на рівні інтерфейсу.....	41
§ Віддавайте перевагу композиції перед спадкуван- ням.....	46
Принципи SOLID.....	50
§ S: Принцип єдиного обов'язку.....	51
§ O: Принцип відкритості/закритості.....	53
§ L: Принцип підстановки Лісков.....	56
§ I: Принцип поділу інтерфейсу.....	62
§ D: Принцип інверсії залежностей.....	65

КАТАЛОГ ПАТЕРНІВ.....	68
Породжувальні патерни	69
§ Фабричний метод / <i>Factory Method</i>	71
§ Абстрактна фабрика / <i>Abstract Factory</i>	87
§ Будівельник / <i>Builder</i>	101
§ Прототип / <i>Prototype</i>	118
§ Одинак / <i>Singleton</i>	132
Структурні патерни.....	141
§ Адаптер / <i>Adapter</i>	144
§ Міст / <i>Bridge</i>	157
§ Компонувальник / <i>Composite</i>	171
§ Декоратор / <i>Decorator</i>	184
§ Фасад / <i>Facade</i>	202
§ Легковаговик / <i>Flyweight</i>	212
§ Замісник / <i>Proxu</i>	226
Поведінкові патерни	238
§ Ланцюжок обов'язків / <i>Chain of Responsibility</i>	242
§ Команда / <i>Command</i>	260
§ Ітератор / <i>Iterator</i>	280
§ Посередник / <i>Mediator</i>	295
§ Знімок / <i>Memento</i>	310
§ Спостерігач / <i>Observer</i>	325
§ Стан / <i>State</i>	340
§ Стратегія / <i>Strategy</i>	356
§ Шаблонний метод / <i>Template Method</i>	369
§ Відвідувач / <i>Visitor</i>	381
Заключення	395

Як читати цю книгу?

Ця книга складається з опису 22-х класичних патернів проектування, вперше відкритих «Бандою Чотирьох» (“Gang of Four” або просто GoF) у 1994 році.

Кожен розділ книги присвячений тільки одному патерну. Саме тому книгу можна читати як послідовно, від краю до краю, так і в довільному порядку, вибираючи тільки ті патерни, які вас цікавлять на даний момент.

Більшість патернів пов’язані між собою, тому ви зможете з легкістю стрибати по пов’язаних темах, використовуючи величезну кількість гіперпосилань, якими всіяні всі розділи книги. В кінці кожного розділу наведені відносини поточного патерна з іншими. Якщо ви бачите там назву патерна, до якого ще не дійшли, продовжуйте читати далі, цей пункт буде повторено в іншому розділі.

Патерни проектування універсальні. Тому всі приклади коду у цій книзі наведено на псевдокодi, без прив’язки до конкретної мови програмування.

Перед вивченням патернів ви можете освіжити пам’ять, пройшовшись **основними термінами об’єктного програмування.** Паралельно я розповім про UML-діаграми, яких у цій книзі приведено в достаток. Якщо ви все це вже знаєте, сміливо приступайте до **вивчення патернів.**

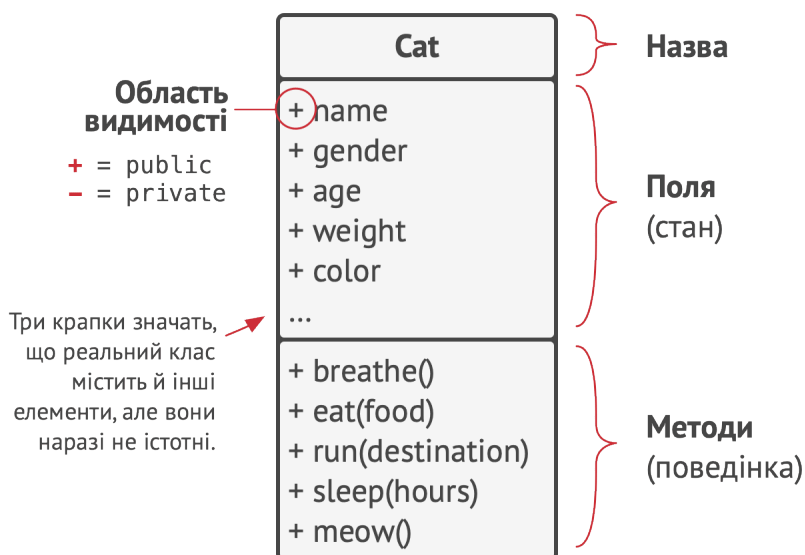
ВСТУП ДО ООП

Згадуємо ООП

Об'єктно-орієнтоване програмування — це методологія програмування, в якій усі важливі речі представлені **об'єктами**, кожен з яких є екземпляром того чи іншого **класу**, а класи утворюють **ієрархію** успадкування.

Об'єкти, класи

Ви любите кошенят? Сподіваюсь, що любите, тому я спробую пояснити усі ці речі на прикладах з котами.



Це UML-діаграма класу. У книзі буде багато таких діаграм.

Отже, у вас є кіт Пухнастик. Він є *об'єктом класу* **Кіт**. Усі коти мають однаковий набір властивостей: ім'я, стать, вік, вагу, колір, улюблену їжу та інше. Це — *поля класу*.

Крім того, всі коти поведуться схожим чином: бігають, дихають, сплять, їдять і муркочуть. Все це — *методи класу*. Узагальнено, поля і методи іноді називають *членами класу*.

Значення полів певного об'єкта зазвичай називають його *станом*, а сукупність методів — *поведінкою*.



Pushystyk: Cat

```
name  = "Pushystyk"
sex    = "male"
age    = 3
weight = 5.5
color  = gray
```



Murka: Cat

```
name  = "Murka"
sex    = "female"
age    = 1
weight = 3.5
color  = white
```

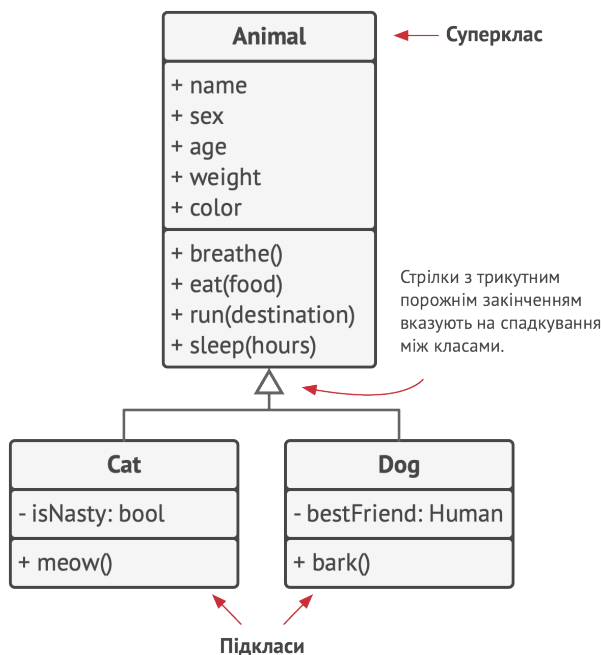
Об'єкти — це екземпляри класів.

Мурка, кішка вашої подруги, теж є екземпляром класу **Кіт**. Вона має такі самі властивості та поведінку, що й Пухнастик,

а відрізняється від нього лише значеннями цих властивостей — вона іншої статі, має інший колір, вагу тощо. Отже, **клас** — це своєрідне «креслення», на підставі якого будуються **об'єкти** — екземпляри цього класу.

Ієрархії класів

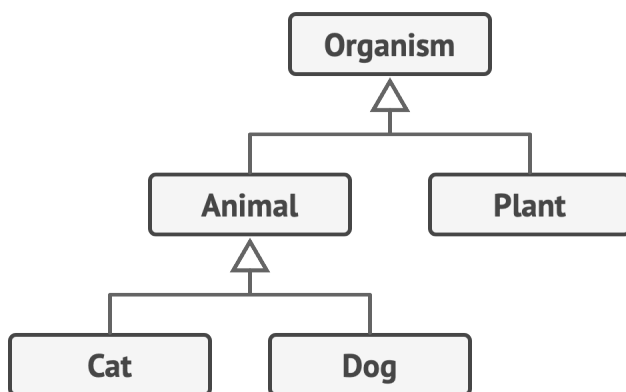
Ідемо далі. У вашого сусіда є собака Жучка. Як відомо, і собаки, і коти мають багато спільного: ім'я, стать, вік, колір є не тільки в котів, але й у собак. Крім того, бігати, дихати, спати та їсти можуть не тільки коти. Виходить так, що ці властивості та поведінка притаманні усьому класу **Тварини**.



UML-діаграма ієрархії класів. Усі класи на цій діаграмі є частиною ієрархії **Тварин**.

Такий батьківський клас прийнято називати **суперкласом**, а його нащадків — **підкласами**. Підкласи успадковують властивості й поведінку свого батька, тому в них міститься лише те, чого немає у суперкласі. Наприклад, тільки коти можуть муркотіти, а собаки — гавкати.

Ми можемо піти далі та виділити ще більш загальний клас живих **Організмів**, який буде батьківським і для **Тварин**, і для **Риб**. Таку «піраміду» класів зазвичай називають **ієрархією**. Клас **Котів** успадкує все, як з **Тварин**, так і з **Організмів**.



Класи на UML-діаграмі можна спрощувати, якщо важливо показати відносини між ними.

Варто згадати, що підкласи можуть перевизначати поведінку методів, які їм дісталися від суперкласів. При цьому вони можуть, як повністю замінити поведінку методу, так і просто додати щось до результату виконання батьківського методу.

19 сторінок

з повної книги пропущені в демо-версії

ПРИНЦИПИ ПРОЕКТУВАННЯ

Якості хорошої архітектури

Перш ніж перейти до вивчення конкретних патернів, поговорімо про сам процес проектування, про те, до чого треба прагнути і чого потрібно уникати.

Повторне використання коду

Не секрет, що вартість і час розробки — це найбільш важливі метрики при розробці будь-яких програмних продуктів. Чим менші обидва ці показники, тим більш конкурентним продукт буде на ринку і тим більше прибутку отримає розробник.

Повторне використання програмної архітектури та коду — це один з найбільш поширених способів зниження вартості розробки. Логіка проста: замість того, щоб розробляти щось повторно, чому б не використати минулі напрацювання у новому проекті?

Ідея виглядає чудово на папері, але, на жаль, не весь код можна пристосувати до роботи в нових умовах. Занадто тісні зв'язки між компонентами, залежність коду від конкретних класів, а не абстрактних інтерфейсів, вшиті в код операції, які неможливо розширити, — все це зменшує гнучкість вашої архітектури та перешкоджає її повторному використанню.

На допомогу приходять патерни проектування, які ціною ускладнення коду програми підвищують гнучкість її частин, що полегшує подальше повторне використання коду.

Наведу цитату Еріха Гамми¹, одного з першовідкривачів патернів, про повторне використання коду та ролі патернів у ньому.

“

Існує три рівні повторного використання коду. На самому нижньому рівні знаходяться класи: корисні бібліотеки класів, контейнери, а також «команди» класів типу контейнерів/ітераторів.

Фреймворки стоять на найвищому рівні. В них важливою є тільки архітектура. Вони визначають ключові абстракції для вирішення деяких бізнес-завдань, представлені у вигляді класів і відносин між ними. Візьміть JUnit, це дуже маленький фреймворк. Він містить усього декілька пов'язаних між собою класів: `Test`, `TestCase` та `TestSuite`. Зазвичай фреймворк має набагато більший обсяг, ніж один клас. Ви вклинюєтесь у фреймворк, розширяючи декотрі його класи. Все працює за так званим голлівудським принципом: «не телефонуйте нам, ми самі вам зателефонуємо». Фреймворк дозволяє вам задати якусь свою поведінку, а потім, коли приходить черга щось робити, сам викликає її. Те ж саме відбувається і в JUnit. Він звертається до вашого класу, коли

1. Erich Gamma on Flexibility and Reuse: <https://refactoring.guru/gamma-interview>

потрібно виконати тест, але все інше відбувається всередині фреймворка.

Є ще середній рівень. Це те, де я бачу патерни. Патерни проектування менші за об'ємом та більш абстрактні, ніж фреймворки. Вони, насправді, є просто описом того, як парочка класів відноситься і взаємодіє один з одним. Рівень повторного використання підвищується, коли ви рухаєтеся в напрямку від конкретних класів до патернів, а потім до фреймворків.

Ще одною привабливою рисою цього середнього рівня є те, що патерни — це менш ризикований спосіб повторного використання, ніж фреймворки. Розробка фреймворку — це вкрай ризикована й дорога інвестиція. У той же час патерни дозволяють повторно використовувати ідеї та концепції у відриві від конкретного коду.

”



Розширюваність

Зміни часто називають головним ворогом програміста.

- Ви придумали ідеальну архітектуру інтернет-магазину, але через місяць довелося додати інтерфейс для замовлень телефоном.
- Ви випустили відеогру під Windows, але потім знадобилася підтримка macOS.
- Ви зробили інтерфейсний фреймворк з квадратними кнопками, але клієнти почали просити круглі.

У кожного програміста кільканадцять подібних історій. Є кілька причин, чому так відбувається.

По-перше, всі ми починаємо розуміти проблему краще в процесі її вирішення. Нерідко до кінця роботи над першою версією програми ми вже готові повністю її переписати, оскільки стали краще розуміти деякі аспекти, які не були настільки нам зрозумілими спочатку. Зробивши другу версію, ви починаєте розуміти проблему ще краще, вносите ще зміни і так далі — процес не зупиняється ніколи, адже не тільки ваше розуміння, але ще й та сама проблема може змінитися з часом.

По-друге, зміни можуть прийти ззовні. У вас є ідеальний клієнт, який з першого разу сформулював те, що йому потрібно, а ви все це зробили. Чудово! Аж ось виходить нова версія операційної системи, в якій ваша програма перестає працювати. Бідкаючись, ви лізете в код, щоб внести деякі зміни.

Проте, на це все можна дивитися оптимістично: якщо хтось просить вас щось змінити в програмі, отже, вона комусь все ж таки ще потрібна.

Ось чому вже навіть трохи досвідчений програміст проектує архітектуру й пише код з урахуванням майбутніх змін.

Базові принципи проектування

Що таке хороший дизайн? За якими критеріями його оцінювати, і яких правил дотримуватися при розробці? Як забезпечити достатній рівень гнучкості, зв'язаності, керованості, стабільності та зрозумілості коду?

Все це правильні запитання, але для кожної програми відповідь буде трохи відрізнятися. Давайте розглянемо універсальні принципи проектування, які допоможуть вам формулювати відповіді на ці запитання самостійно.

До речі, більшість патернів, наведених у цій книзі, базується саме на перерахованих нижче принципах.

Інкапсулюйте те, що змінюється

Визначте аспекти програми, класу або методу, які змінюються найчастіше, і відокремте їх від того, що залишається постійним.

Цей принцип має на меті зменшити наслідки, викликані змінами. Уявіть, що ваша програма — це корабель, а зміни — то підступні міни на його шляху. Натикаючись на міну, корабель заповнюється водою та тоне.

Знаючи це, ви можете розділити трюм корабля на незалежні секції, проходи між якими наглухо зачиняти. Тепер після зіткнення з міною корабель залишиться на плаву. Вода затопить лише одну секцію, залишивши решту без змін.

Ізолюючи мінливі частини програми в окремих модулях, класах або методах, ви зменшуєте кількість коду, якого торкнуться наступні зміни. Отже, вам потрібно буде витратити менше зусиль на те, щоб привести програму до робочого стану, налагодити та протестувати код, що змінився. Де менше роботи, там менша вартість розробки. А там, де менша вартість, там і перевага перед конкурентами.

Приклад інкапсуляції на рівні методу

Припустімо, що ви розробляєте інтернет-магазин. Десь всередині вашого коду знаходиться метод `getOrderTotal`, що

розраховує фінальну суму замовлення з урахуванням розміру податку.

Ми можемо припустити, що код обчислення податків, імовірно, буде часто змінюватися. По-перше, логіка нарахування податку залежить від країни, штату й навіть міста, в якому знаходиться покупець. До того ж, розмір податку не сталий і може змінюватися з часом.

Через ці зміни вам доведеться постійно торкатися методу `getOrderTotal`, який, насправді, не особливо цікавиться *деталлями* обчислення податків.

```
1  method getOrderTotal(order) is
2      total = 0
3      foreach item in order.lineItems
4          total += item.price * item.quantity
5
6      if (order.country == "US")
7          total += total * 0.07 // US sales tax
8      else if (order.country == "EU"):
9          total += total * 0.20 // European VAT
10
11     return total
```

ДО: правила обчислення податків змішані з основним кодом методу.

Ви можете перенести логіку обчислення податків в окремий метод, приховавши деталі від оригінального методу.

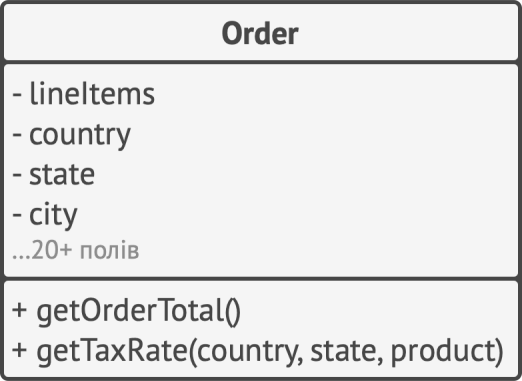
```
1  method getOrderTotal(order) is
2      total = 0
3      foreach item in order.lineItems
4          total += item.price * item.quantity
5
6      total += total * getTaxAmount(order.country)
7
8      return total
9
10 method getTaxAmount(country) is
11     if (country == "US")
12         return 0.07 // US sales tax
13     else if (country == "EU")
14         return 0.20 // European VAT
15     else
16         return 0
```

ПІСЛЯ: розмір податку можна отримати, викликавши один метод.

Тепер зміни податків будуть ізольовані в рамках одного методу. Більш того, якщо логіка обчислення податків ще більш ускладниться, вам буде легше отримати цей метод до власного класу.

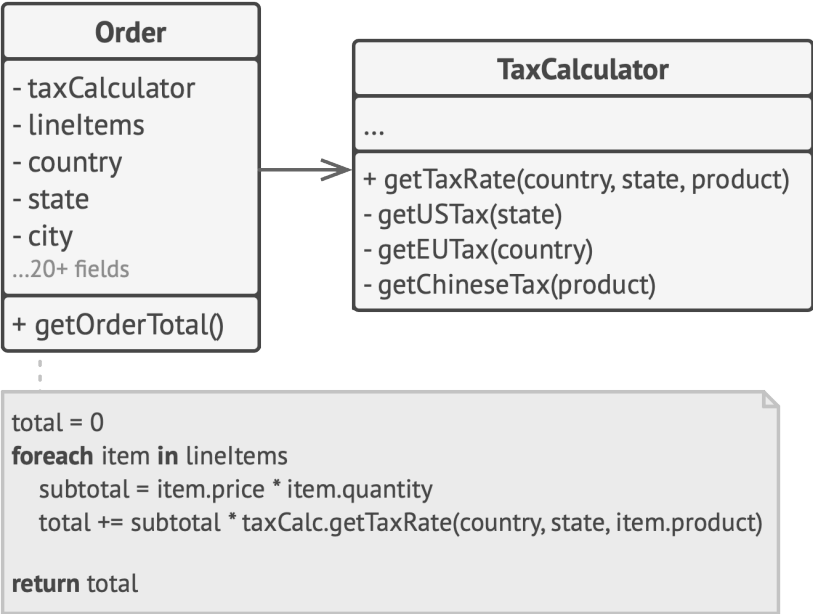
Приклад інкапсуляції на рівні класу

Видобути логіку податків до власного класу? Якщо логіка податків стала занадто складною, то чому б і ні?



ДО: обчислення податків у класі замовлень.

Об'єкти замовлень делегуватимуть обчислення податків окремому об'єкту-калькулятору податків.



ПІСЛЯ: обчислення податків приховано в класі замовлень.

27 сторінок

з повної книги пропущені в демо-версії

КАТАЛОГ ПАТЕРНІВ

Породжувальні патерни

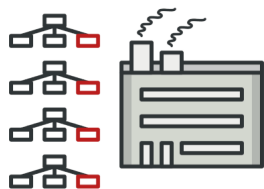
Ці патерни відповідають за зручне та безпечне створення нових об'єктів або навіть цілих сімейств об'єктів.



Фабричний метод

Factory Method

Визначає загальний інтерфейс для створення об'єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об'єктів.



Абстрактна фабрика

Abstract Factory

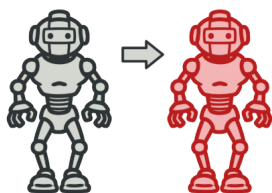
Дає змогу створювати сімейства пов'язаних об'єктів, не прив'язуючись до конкретних класів створюваних об'єктів.



Будівельник

Builder

Дає змогу створювати складні об'єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об'єктів.



Прототип

Prototype

Дає змогу копіювати об'єкти, не вдаючись у подробиці їхньої реалізації.



Одинак

Singleton

Гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього.



ФАБРИЧНИЙ МЕТОД

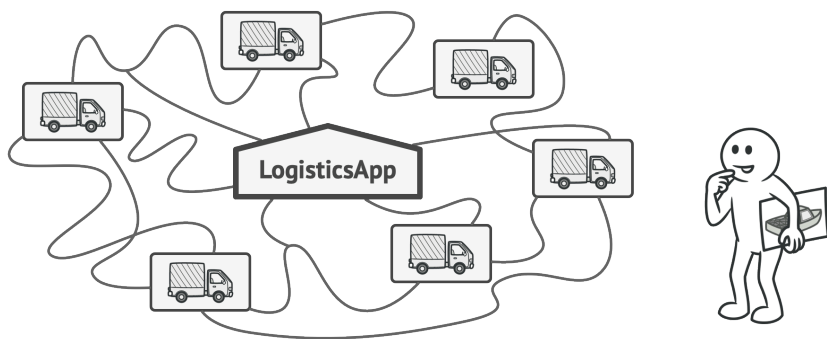
Також відомий як: Віртуальний конструктор, Factory Method

Фабричний метод — це породжувальний патерн проектування, який визначає загальний інтерфейс для створення об'єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об'єктів.

☹ Проблема

Уявіть, що ви створюєте програму керування вантажними перевезеннями. Спочатку ви плануєте перевезення товарів тільки вантажними автомобілями. Тому весь ваш код працює з об'єктами класу `Вантажівка`.

Згодом ваша програма стає настільки відомою, що морські перевізники шикуються в чергу і благають додати до програми підтримку морської логістики.



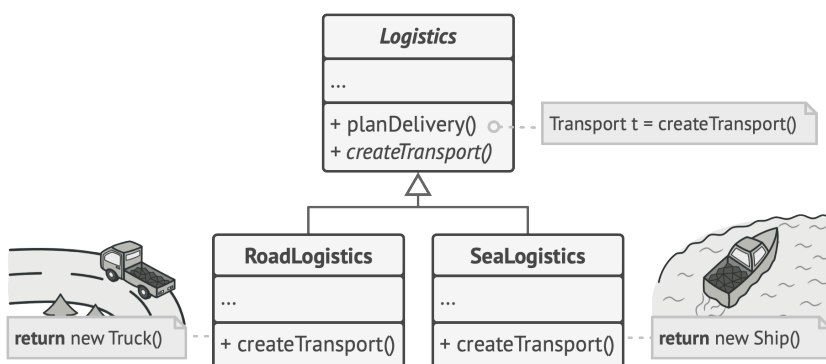
Додати новий клас не так просто, якщо весь код вже залежить від конкретних класів.

Чудові новини, чи не так?! Але як щодо коду? Велика частина існуючого коду жорстко прив'язана до класів `Вантажівка`. Щоб додати до програми класи морських `Суден`, знадобиться перелопачувати весь код. Якщо ж ви вирішите додати до програми ще один вид транспорту, тоді всю цю роботу доведеться повторити.

У підсумку ви отримаєте жадливий код, переповнений умовними операторами, що виконують ту чи іншу дію в залежності від вибраного класу транспорту.

😊 Рішення

Патерн Фабричний метод пропонує відмовитись від безпосереднього створення об'єктів за допомогою оператора `new`, замінивши його викликом особливого *фабричного* методу. Не лякайтеся, об'єкти все одно будуть створюватися за допомогою `new`, але робити це буде фабричний метод.

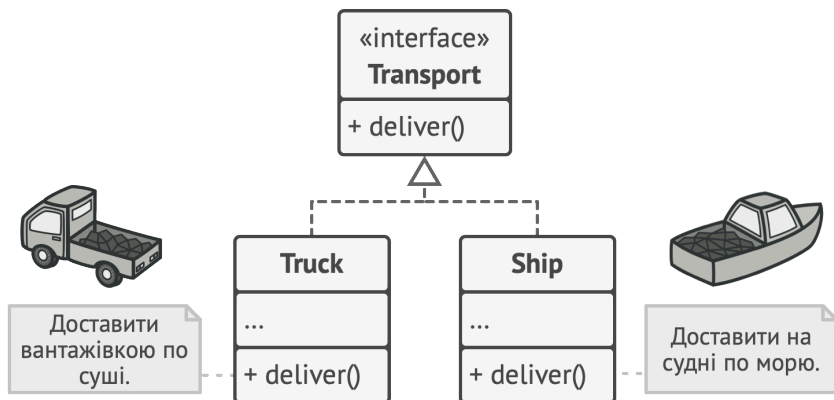


Підкласи можуть змінювати клас створюваних об'єктів.

На перший погляд це може здатись безглуздом — ми просто перемістили виклик конструктора з одного кінця програми в інший. Проте тепер ви зможете перевизначити фабричний метод у підкласі, щоб змінити тип створюваного продукту.

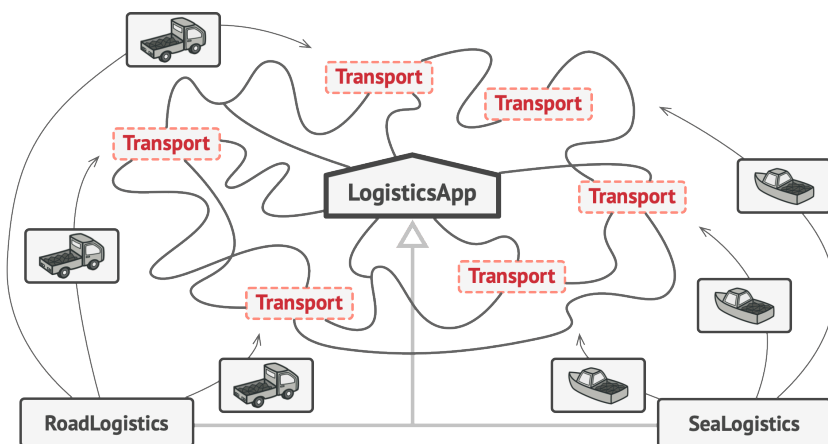
Щоб ця система запрацювала, всі об'єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть виго-

товляти об'єкти різних класів, що відповідають одному і тому самому інтерфейсу.



Всі об'єкти-продукти повинні мати спільний інтерфейс.

Наприклад, класи **Вантажівка** і **Судно** реалізують інтерфейс **Транспорт** з методом **доставити**. Кожен з цих класів реалізує метод по-своєму: вантажівки перевозять вантажі сушею, а судна — морем. Фабричний метод класу **ДорожньоїЛогістики** поверне об'єкт-вантажівку, а класу **МорськоїЛогістики** — об'єкт-судно.

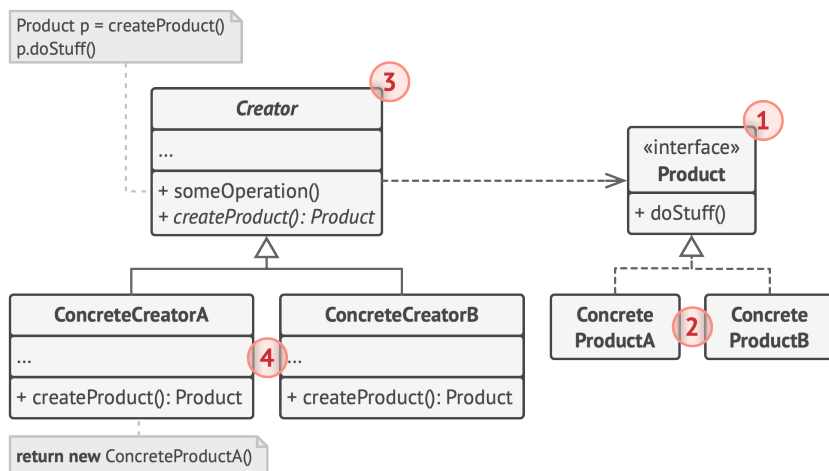


Допоки всі продукти реалізують спільний інтерфейс, їхні об'єкти можна змінювати один на інший у клієнтському коді.

Клієнт фабричного методу не відчує різниці між цими об'єктами, адже він трактуватиме їх як якийсь абстрактний **Транспорт**.

Для нього буде важливим, щоб об'єкт мав метод **доставити**, а не те, як конкретно він працює.

Структура



1. **Продукт** визначає загальний інтерфейс об'єктів, які може створювати творець та його підкласи.
2. **Конкретні продукти** містять код різних продуктів. Продукти відрізнятимуться реалізацією, але інтерфейс у них буде спільним.
3. **Творець** оголошує фабричний метод, який має повертати нові об'єкти продуктів. Важливо, щоб тип результату цього методу співпадав із загальним інтерфейсом продуктів.

Зазвичай, фабричний метод оголошують абстрактним, щоб змусити всі підкласи реалізувати його по-своєму. Однак він може також повертати продукт за замовчуванням.

Незважаючи на назву, важливо розуміти, що створення продуктів **не є** єдиною і головною функцією творця. Зазвичай він містить ще й інший корисний код для роботи з продуктом. Аналогія: у великій софтверній компанії може бути центр підготовки програмістів, але все ж таки основним завданням компанії залишається написання коду, а не навчання програмістів.

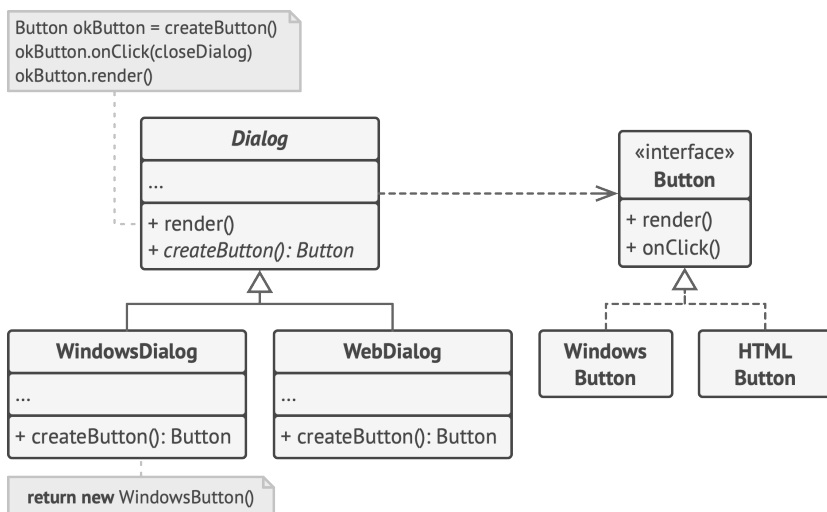
4. **Конкретні творці** по-своєму реалізують фабричний метод, виробляючи ті чи інші конкретні продукти.

Фабричний метод не зобов'язаний створювати нові об'єкти увесь час. Його можна переписати так, аби повертати з якогось сховища або кешу вже існуючі об'єкти.

Псевдокод

У цьому прикладі **Фабричний метод** допомагає створювати крос-платформові елементи інтерфейсу, не прив'язуючи основний код програми до конкретних класів кожного елементу.

Фабричний метод оголошений у класі діалогів. Його підкласи належать до різних операційних систем. Завдяки фабричному методу, вам не потрібно переписувати логіку діалогів під кожну систему. Підкласи можуть успадкувати майже увесь код базового діалогу, змінюючи типи кнопок та інших елементів, з яких базовий код будує вікна графічного користувацького інтерфейсу.



Приклад крос-платформового діалогу.

Базовий клас діалогів працює з кнопками через їхній загальний програмний інтерфейс. Незалежно від того, яку варіацію кнопок повернув фабричний метод, діалог залишиться робочим. Базовий клас не залежить від конкретних класів кнопок, залишаючи підкласам прийняття рішення про тип кнопок, які необхідно створити.

Такий підхід можна застосувати і для створення інших елементів інтерфейсу. Хоча кожен новий тип елементів наблизить вас до **Абстрактної фабрики**.

```

1  // Патерн Фабричний метод має сенс лише тоді, коли в програмі є
2  // ієрархія класів продуктів.
3  interface Button is
4      method render()
  
```

```


5     method onClick(f)
6
7     class WindowsButton implements Button is
8         method render(a, b) is
9             // Відобразити кнопку в стилі Windows.
10        method onClick(f) is
11            // Навісити на кнопку обробник подій Windows.
12
13    class HTMLButton implements Button is
14        method render(a, b) is
15            // Повернути HTML-код кнопки.
16        method onClick(f) is
17            // Навісити на кнопку обробник події браузера.
18
19
20    // Базовий клас фабрики. Зауважте, що "фабрика" – це всього лише
21    // додаткова роль для цього класу. Скоріше за все, він вже має
22    // якусь бізнес-логіку, яка потребує створення продуктів.
23    class Dialog is
24        method render() is
25            // Щоб використати фабричний метод, ви маєте
26            // пересвідчитися, що ця бізнес-логіка не залежить від
27            // конкретних класів продуктів. Button – це загальний
28            // інтерфейс кнопок, тому все гаразд.
29            Button okButton = createButton()
30            okButton.onClick(closeDialog)
31            okButton.render()
32
33            // Ми виносимо весь код створення продуктів до особливого
34            // методу, який називають "фабричним".
35            abstract method createButton():Button
36


```

```
37
38  // Конкретні фабрики перевизначають фабричний метод і повертають
39  // з нього власні продукти.
40  class WindowsDialog extends Dialog is
41      method createButton():Button is
42          return new WindowsButton()
43
44  class WebDialog extends Dialog is
45      method createButton():Button is
46          return new HTMLButton()
47
48
49  class Application is
50      field dialog: Dialog
51
52      // Програма створює певну фабрику в залежності від
53      // конфігурації або оточення.
54      method initialize() is
55          config = readApplicationConfigFile()
56
57          if (config.OS == "Windows") then
58              dialog = new WindowsDialog()
59          else if (config.OS == "Web") then
60              dialog = new WebDialog()
61          else
62              throw new Exception("Error! Unknown operating system.")
63
64      // Якщо весь інший клієнтський код працює з фабриками та
65      // продуктами тільки через загальний інтерфейс, то для нього
66      // байдуже, якого типу фабрику було створено на початку.
67      method main() is
68          this.initialize()
```


69 `dialog.render()`


Застосування

 **Коли типи і залежності об'єктів, з якими повинен працювати ваш код, невідомі заздалегідь.**

 Фабричний метод відокремлює код виробництва продуктів від решти коду, який використовує ці продукти.

Завдяки цьому код виробництва можна розширювати, не зачіпаючи основний код. Щоб додати підтримку нового продукту, вам потрібно створити новий підклас та визначити в ньому фабричний метод, повертаючи звідти екземпляр нового продукту.

 **Коли ви хочете надати користувачам можливість розширювати частини вашого фреймворку чи бібліотеки.**

 Користувачі можуть розширювати класи вашого фреймворку через успадкування. Але як же зробити так, аби фреймворк створював об'єкти цих класів, а не стандартних?

Рішення полягає у тому, щоб надати користувачам можливість розширювати не лише бажані компоненти, але й класи, які їх створюють. Тому ці класи повинні мати конкретні створюючі методи, які можна буде перевизначити.

Наприклад, ви використовуєте готовий UI-фреймворк для свого додатку. Але — от халепа — вам необхідно мати круглі кнопки, а не стандартні прямокутні. Ви створюєте клас `RoundButton`. Але як сказати головному класу фреймворку `UIFramework`, щоб він почав тепер створювати круглі кнопки замість стандартних прямокутних?

Для цього з базового класу фреймворку ви створюєте підклас `UIWithRoundButtons`, перевизначаєте в ньому метод створення кнопки (а-ля, `createButton`) і вписуєте туди створення свого класу кнопок. Потім використовуєте `UIWithRoundButtons` замість стандартного `UIFramework`.



Коли ви хочете зекономити системні ресурси, повторно використовуючи вже створені об'єкти, замість породження нових.



Така проблема зазвичай виникає під час роботи з «важкими», вимогливими до ресурсів об'єктами, такими, як підключення до бази даних, файлової системи й подібними.

Уявіть, скільки дій вам потрібно зробити, аби повторно використовувати вже існуючі об'єкти:

1. Спочатку слід створити загальне сховище, щоб зберігати в ньому всі створювані об'єкти.
2. При запиті нового об'єкта потрібно буде подивитись у сховище та перевірити, чи є там невикористаний об'єкт.

3. Потім повернути його клієнтському коду.
4. Але якщо ж вільних об'єктів немає, створити новий, не забувши додати його до сховища.

Увесь цей код потрібно десь розмістити, щоб не засмічувати клієнтський код.

Найзручнішим місцем був би конструктор об'єкта, адже всі ці перевірки потрібні тільки під час створення об'єктів, але, на жаль, конструктор завжди створює **нові** об'єкти, тому він не може повернути існуючий екземпляр.

Отже, має бути інший метод, який би віддавав як існуючі, так і нові об'єкти. Ним і стане фабричний метод.



Кроки реалізації

1. Приведіть усі створювані продукти до загального інтерфейсу.
2. Створіть порожній фабричний метод у класі, який виробляє продукти. В якості типу, що повертається, вкажіть загальний інтерфейс продукту.
3. Пройдіться по коду класу й знайдіть усі ділянки, що створюють продукти. По черзі замініть ці ділянки викликами фабричного методу, переносючи в нього код створення різних продуктів.

Можливо, доведеться додати до фабричного методу декілька параметрів, що контролюють, який з продуктів потрібно створити.

Імовірно за все, фабричний метод виглядатиме гнітюче на цьому етапі. В ньому житиме великий умовний оператор, який вибирає клас створюваного продукту. Але не хвилюйтеся, ми ось-ось все це виправимо.

4. Для кожного типу продуктів заведіть підклас і перевизначте в ньому фабричний метод. З суперкласу перемістіть туди код створення відповідного продукту.
5. Якщо створюваних продуктів занадто багато для існуючих підкласів творця, ви можете подумати про введення параметрів до фабричного методу, аби повертати різні продукти в межах одного підкласу.

Наприклад, у вас є клас `Пошта` з підкласами `АвіаПошта` і `НаземнаПошта`, а також класи продуктів `Літак`, `Вантажівка` й `Потяг`. `Авіа` відповідає `Літакам`, але для `НаземноїПошти` є відразу два продукти. Ви могли б створити новий підклас пошти й для потягів, але проблему можна вирішити по-іншому. Клієнтський код може передавати до фабричного методу `НаземноїПошти` аргумент, що контролює, який з продуктів буде створено.

6. Якщо після цих всіх переміщень фабричний метод став порожнім, можете зробити його абстрактним. Якщо ж у

ньому щось залишилося — не страшно, це буде його типовою реалізацією (за замовчуванням).

Переваги та недоліки

- ✓ Позбавляє клас від прив'язки до конкретних класів продуктів.
- ✓ Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
- ✓ Спрощує додавання нових продуктів до програми.
- ✓ Реалізує *принцип відкритості/закритості*.
- ✗ Може призвести до створення великих **паралельних ієрархій класів**, адже для кожного класу продукту потрібно створити власний підклас творця.

Відносини з іншими патернами

- Багато архітектур починаються із застосування **Фабричного методу** (простішого та більш розширюваного за допомогою підкласів) та еволюціонують у бік **Абстрактної фабрики**, **Прототипу** або **Будівельника** (гнучкіших, але й складніших).
- Класи **Абстрактної фабрики** найчастіше реалізуються за допомогою **Фабричного методу**, хоча вони можуть бути побудовані і на основі **Прототипу**.

- **Фабричний метод** можна використовувати разом з **Ітератором**, щоб підкласи колекцій могли створювати необхідні їм ітератори.
- **Прототип** не спирається на спадкування, але йому потрібна складна операція ініціалізації. **Фабричний метод**, навпаки, побудований на спадкуванні, але не вимагає складної ініціалізації.
- **Фабричний метод** можна розглядати як окремий випадок **Шаблонного методу**. Крім того, *Фабричний метод* нерідко буває частиною великого класу з *Шаблонними методами*.

309 сторінок

з повної книги пропущені в демо-версії