Харківський національний університет імені В.Н. Каразіна

Навчально-науковий інститут комп’ютерних наук та штучного інтелекту

Кафедра математичного моделювання та аналізу даних

Доповідь

на тему: «паттерн «будівельник»»

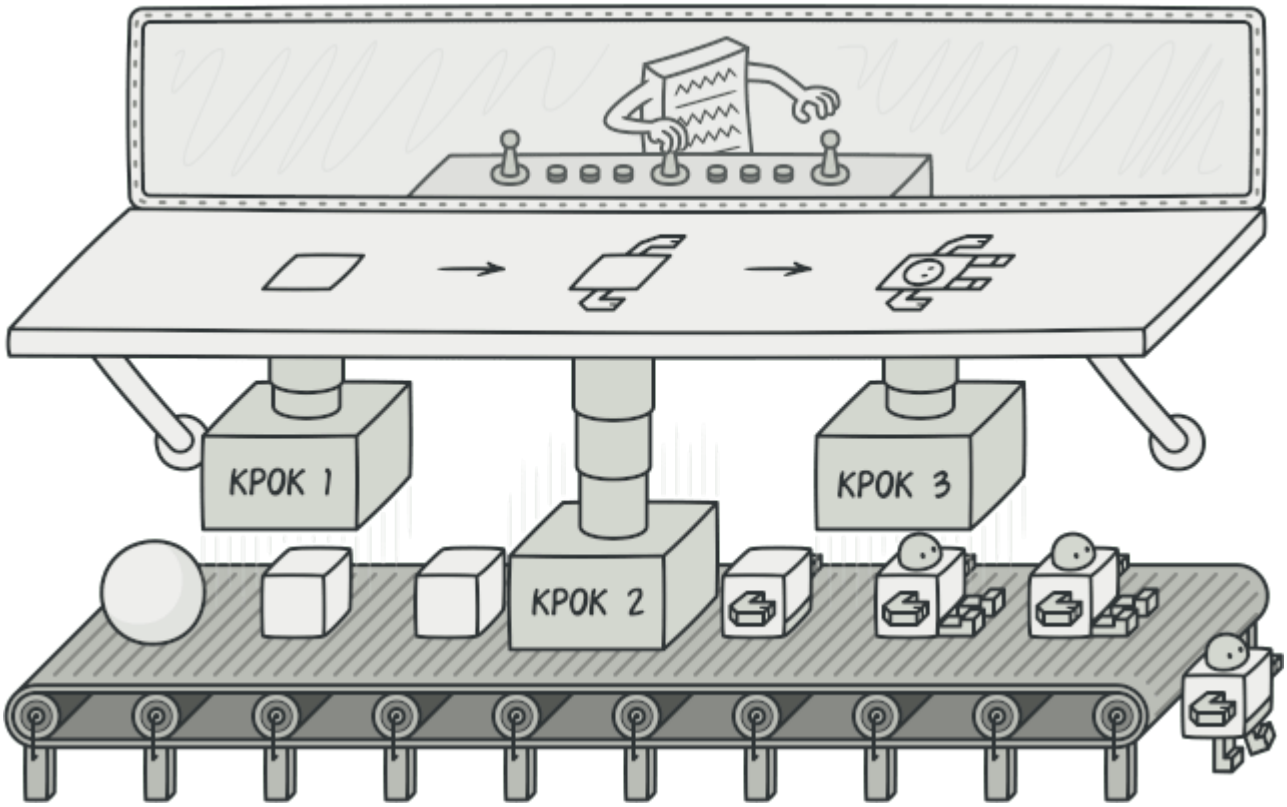
Виконав: студент 3 курсу групи 31
Спеціальності 124 «Комп’ютерні науки»
Грецький Д. В._____
Прийняв: викладач ЗВО
_____

Харків – 2025

Будівельник

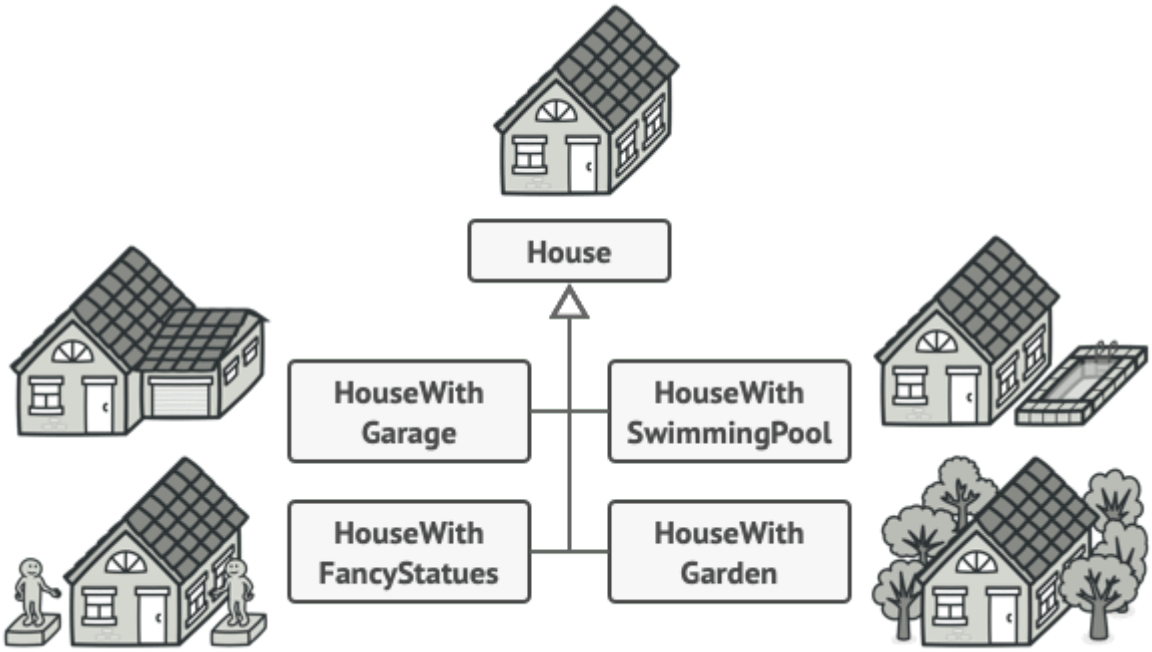
Суть патерна

Будівельник — це породжувальний патерн проектування, що 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.



Проблема

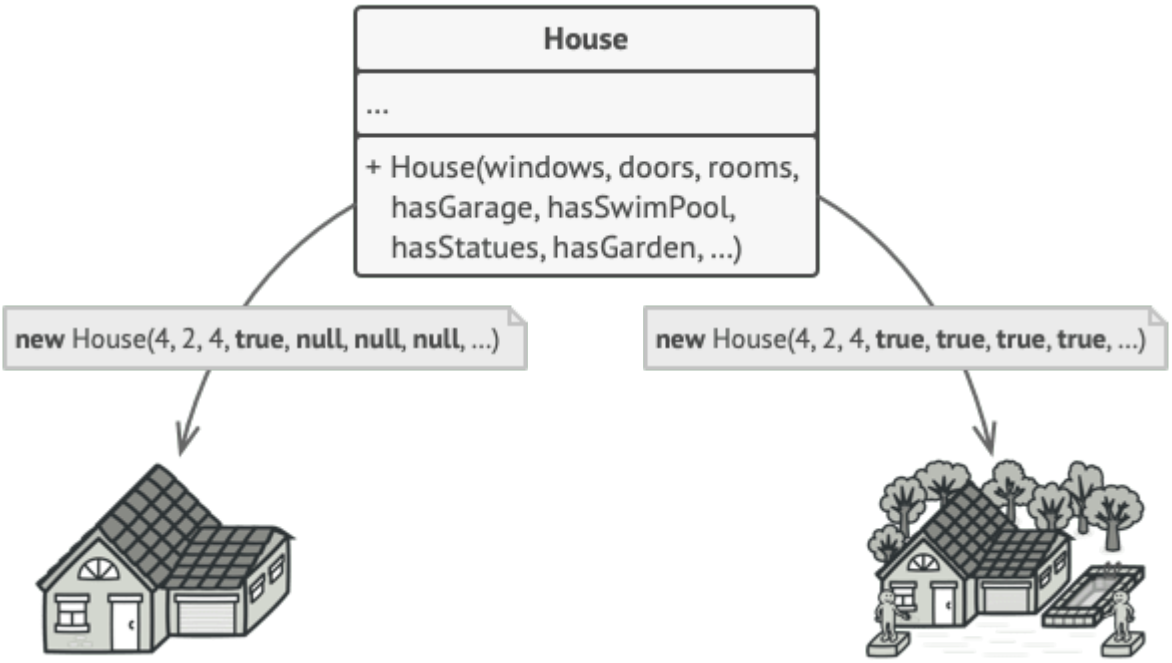
Уявіть складний об’єкт, що вимагає кропіткої покрокової ініціалізації безлічі полів і вкладених об’єктів. Код ініціалізації таких об’єктів зазвичай захований всередині монстроподібного конструктора з десятком параметрів. Або ще гірше — розпорошений по всьому клієнтському коду.



Наприклад, подумаймо про те, як створити об’єкт Будинок. Щоб побудувати стандартний будинок, потрібно: звести 4 стіни, встановити двері, вставити пару вікон та постелити дах. Але що робити, якщо ви хочете більший та світліший будинок, що має басейн, сад та інше добро?

Найпростіше рішення — розширити клас Будинок, створивши підкласи для всіх комбінацій параметрів будинку. Проблема такого підходу — величезна кількість класів, які вам доведеться створити. Кожен новий параметр, на кшталт кольору шпалер чи матеріалу покрівлі, змусить вас створювати все більше й більше класів для перерахування усіх можливих варіантів.

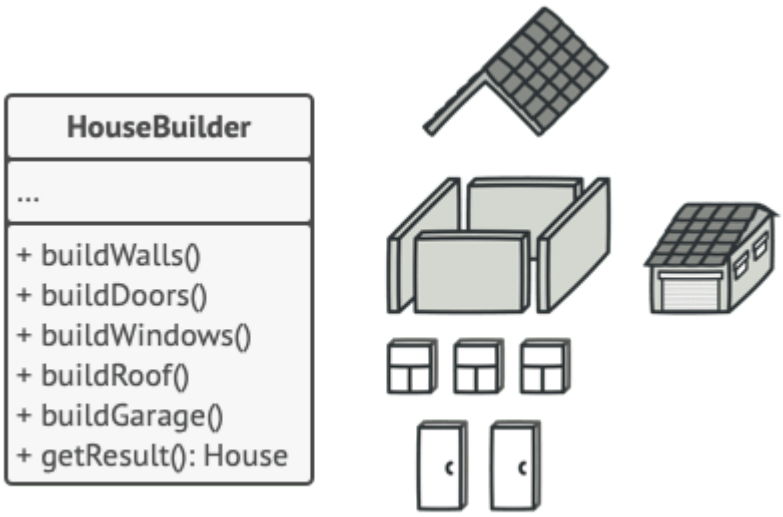
Аби не плодити підкласи, можна підійти до вирішення питання з іншого боку. Ви можете створити гігантський конструктор Будинку, що приймає безліч параметрів для контролю над створюваним продуктом. Так, це позбавить вас від підкласів, але призведе до появи іншої проблеми.



Більшість цих параметрів буде простоювати, а виклики конструктора будуть виглядати монстроподібно через **довгий список параметрів**. Наприклад, басейн є далеко не в кожному будинку, тому параметри, пов'язані з басейнами, даремно простоюватимуть у 99% випадків.

Рішення

Патерн Будівельник пропонує винести конструювання об'єкта за межі його власного класу, доручивши цю справу окремим об'єктам, які називаються *будівельниками*.



Патерн пропонує розбити процес конструювання об'єкта на окремі кроки (наприклад, побудуватиСтіни, встановитиДвері і т. д.) Щоб створити об'єкт, вам потрібно по черзі викликати методи будівельника. До того ж не потрібно викликати всі кроки, а лише ті, що необхідні для виробництва об'єкта певної конфігурації.

Зазвичай один і той самий крок будівництва може відрізнятися для різних варіацій виготовлених об’єктів. Наприклад, дерев’яний будинок потребує будівництва стін з дерева, а кам’яний — з каменю.

У цьому випадку ви можете створити кілька класів будівельників, які по-різному виконуватимуть ті ж самі кроки. Використовуючи цих будівельників в одному й тому самому будівельному процесі, ви зможете отримувати на виході різні об’єкти.

Наприклад, один будівельник робить стіни з дерева і скла, інший — з каменю і заліза, третій — із золота та діамантів. Викликавши одні й ті самі кроки будівництва, у першому випадку ви отримаєте звичайний житловий будинок, у другому — маленьку фортецю, а в третьому — розкішне житло. Зауважу, що код, який викликає кроки будівництва, повинен працювати з будівельниками через загальний інтерфейс, щоб їх можна було вільно замінювати один на інший.

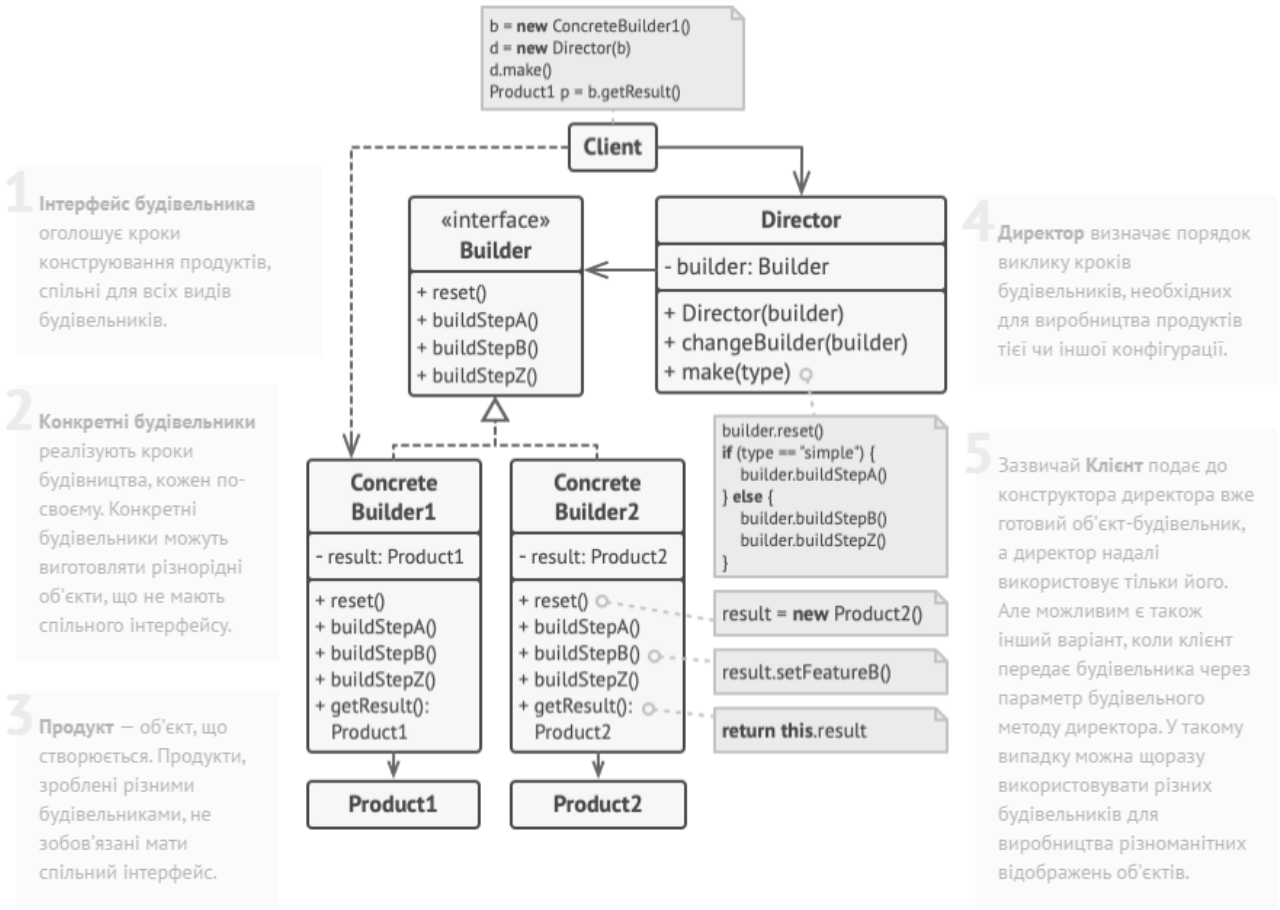
Директор

Ви можете піти далі та виділити виклики методів будівельника в окремий клас, що називається «Директором». У цьому випадку директор задаватиме порядок кроків будівництва, а будівельник — виконуватиме їх.

Окремий клас *директора* не є суворо обов’язковим. Ви можете викликати методи будівельника і безпосередньо з клієнтського коду. Тим не менш, директор корисний, якщо у вас є кілька способів конструювання продуктів, що відрізняються порядком і наявними кроками конструювання. У цьому випадку ви зможете об’єднати всю цю логіку в одному класі.

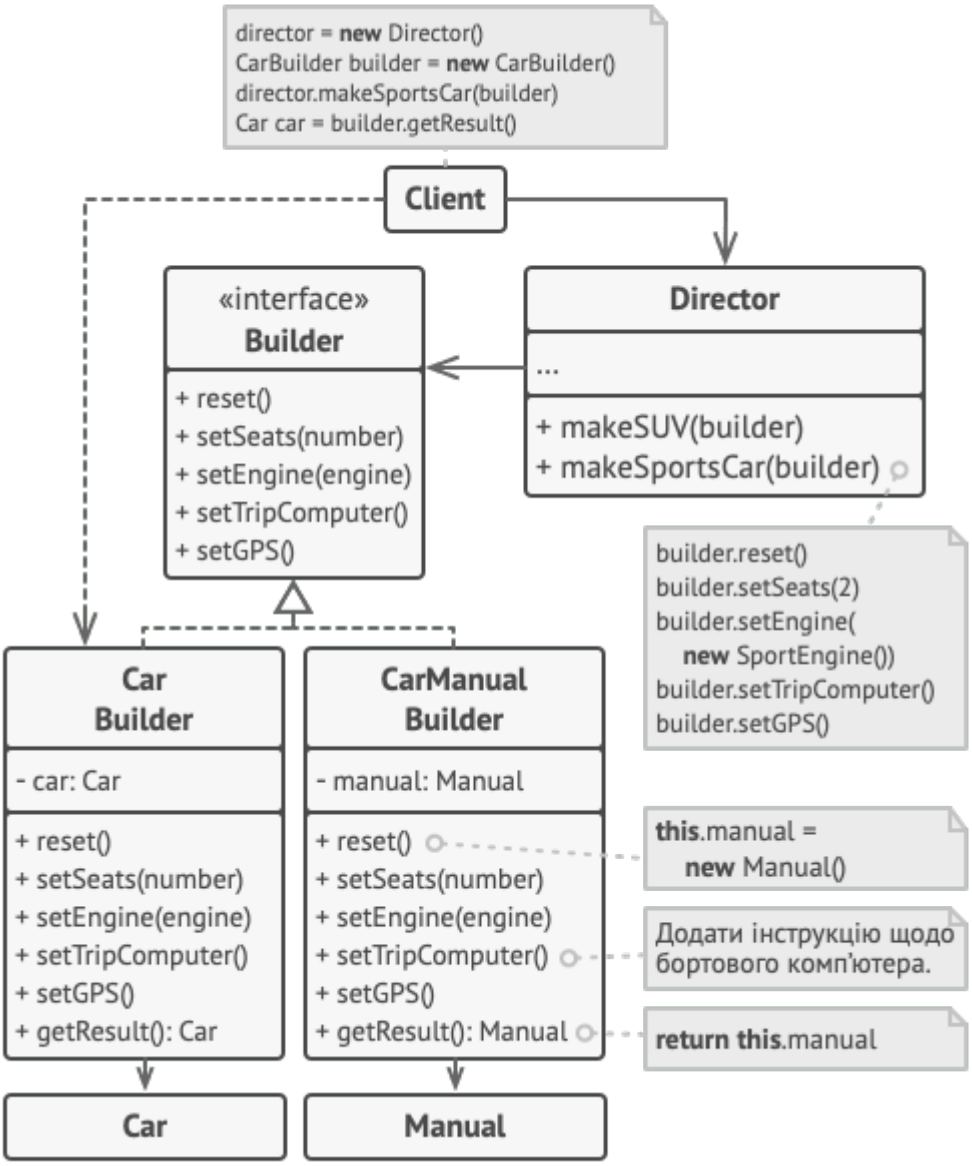
Така структура класів повністю приховає від клієнтського коду процес конструювання об’єктів. Клієнту залишиться лише прив’язати бажаного будівельника до директора, а потім отримати від будівельника готовий результат.

Структура



Псевдокод

У цьому прикладі **Будівельник** використовується для покрокового конструювання автомобілів та технічних посібників до них.



Автомобіль — це складний об'єкт, який можна налаштувати сотнею різних способів. Замість того, щоб налаштовувати автомобіль через конструктор, ми винесемо його збирання в окремий клас-будівельник, передбачивши методи для конфігурації всіх частин автомобіля.

Клієнт може збирати автомобілі, працюючи з будівельником безпосередньо. З іншого боку, він може доручити цю справу директору. Це об'єкт, який знає, які кроки будівельника потрібно викликати, щоб отримати кілька найпопулярніших конфігурацій автомобілів.

Проте, до кожного автомобіля ще потрібен посібник користувача, що відповідає його конфігурації. Для цього ми створимо ще один клас будівельника, який замість конструювання автомобіля друкуватиме сторінки посібника до тієї деталі, яку ми вбудовуємо в продукт. Тепер, пропустивши через одні й ті самі кроки обидва типи будівельників, ми отримаємо автомобіль та відповідний до нього посібник користувача.

Очевидно, що паперовий посібник і металевий автомобіль — це дві абсолютно різні речі. З цієї причини ми повинні отримувати результат безпосередньо від будівельників, а не від директора. Інакше нам довелося б жорстко прив’язати директора до конкретних класів автомобілів і посібників.

```
// Будівельник може створювати різні продукти, використовуючи
// один і той самий процес будівництва.
class Car is
    // Автомобілі можуть відрізнятися комплектацією: типом
    // двигуна, кількістю сидінь, можуть мати або не мати GPS і
    // систему навігації тощо. Крім того, автомобілі можуть бути
    // міськими, спортивними або позашляховиками.

class Manual is
    // Посібник користувача для даної конфігурації автомобіля.


// Інтерфейс будівельників оголошує всі можливі етапи та кроки
// конфігурації продукту.
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)

// Усі конкретні будівельники реалізують загальний інтерфейс по-
// своєму.
class CarBuilder implements Builder is
    private field car:Car
    method reset()
        // Помістити новий об'єкт Car в полі "car".
    method setSeats(...) is
        // Встановити вказану кількість сидінь.
    method setEngine(...) is
        // Встановити наданий двигун.
    method setTripComputer(...) is
        // Встановити надану систему навігації.
    method setGPS(...) is
        // Встановити або зняти GPS.
    method getResult(): Car is
        // Повернути поточний об'єкт автомобіля.

// На відміну від інших породжувальних патернів, де продукти
// мають бути частиною однієї ієрархії класів або слідувати
// загальному інтерфейсу, будівельники можуть створювати
// абсолютно різні продукти, які не мають спільного предка.
class CarManualBuilder implements Builder is
    private field manual:Manual
    method reset()
        // Помістити новий об'єкт Manual у полі "manual".
    method setSeats(...) is
        // Описати кількість місць в автівці.
```

```
    method setEngine(...) is
        // Додати до посібника опис двигуна.
    method setTripComputer(...) is
        // Додати до посібника опис системи навігації.
    method setGPS(...) is
        // Додати до посібника інструкцію для GPS.
    method getResult(): Manual is
        // Повернути поточний об'єкт посібника.


// Директор знає, в якій послідовності потрібно змушувати
// працювати будівельника, щоб отримати ту чи іншу версію
// продукту. Зауважте, що директор працює з будівельником через
// загальний інтерфейс, завдяки чому він не знає тип продукту,
// який виготовляє будівельник.
class Director is
    method constructSportsCar(builder: Builder) is
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)


// Директор отримує об'єкт конкретного будівельника від клієнта
// (програми). Програма сама знає, якого будівельника
// використати, аби отримати потрібний продукт.
class Application is
    method makeCar() is
        director = new Director()

        CarBuilder builder = new CarBuilder()
        director.constructSportsCar(builder)
        Car car = builder.getResult()

        CarManualBuilder builder = new CarManualBuilder()
        director.constructSportsCar(builder)

        // Готовий продукт повертає будівельник, оскільки
        // директор частіше за все не знає і не залежить від
        // конкретних класів будівельників та продуктів.
        Manual manual = builder.getResult()
```

Застосування

- **Коли ви хочете позбутися від «телескопічного конструктора».**

Припустімо, у вас є один конструктор з десятьма опціональними параметрами. Його незручно викликати, тому ви створили ще десять конструкторів з меншою кількістю параметрів. Все, що вони роблять, — це переадресовують виклик до базового конструктора, подаючи якісь типові значення в параметри, які відсутні в них самих.

```
class Pizza {
    Pizza(int size) { ... }
    Pizza(int size, boolean cheese) { ... }
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }
    // ...
```

Патерн Будівельник дозволяє збирати об’єкти покроково, викликаючи тільки ті кроки, які вам потрібні. Отже, більше не потрібно намагатися «запхати» до конструктора всі можливі опції продукту.

- **Коли ваш код повинен створювати різні уявлення якогось об’єкта. Наприклад, дерев’яні та залізобетонні будинки.**

Будівельник можна застосувати, якщо створення кількох відображень об’єкта складається з однакових етапів, які відрізняються деталями.

Інтерфейс будівельників визначить всі можливі етапи конструювання. Кожному відображенню відповідатиме власний клас-будівельник. Порядок етапів будівництва визначатиме клас-директор.

- **Коли вам потрібно збирати складні об’єкти, наприклад, дерева Компонувальника.**

Будівельник конструює об’єкти покроково, а не за один прохід. Більш того, кроки будівництва можна виконувати рекурсивно. А без цього не побудувати деревоподібну структуру на зразок **Компонувальника**.

Зауважте, що Будівельник не дозволяє стороннім об’єктам отримувати доступ до об’єкта, що конструюється, доки той не буде повністю готовий. Це захищає клієнтський код від отримання незавершених «битих» об’єктів.

Кроки реалізації

1. Переконайтеся в тому, що створення різних відображень об’єкта можна звести до загальних кроків.
2. Опишіть ці кроки в загальному інтерфейсі будівельників.
3. Для кожного з відображень об’єкта-продукту створіть по одному класу-будівельнику й реалізуйте їхні методи будівництва.

   Не забудьте про метод отримання результату. Зазвичай конкретні будівельники визначають власні методи отримання результату будівництва. Ви не можете описати ці методи в інтерфейсі будівельників, оскільки продукти не обов’язково повинні мати загальний базовий клас або інтерфейс. Але ви завжди можете додати метод отримання результату до загального інтерфейсу, якщо ваші будівельники виготовляють однорідні продукти, які мають спільного предка.

4. Подумайте про створення класу директора. Його методи створюватимуть різні конфігурації продуктів, викликаючи різні кроки одного і того самого будівельника.
5. Клієнтський код повинен буде створювати й об'єкти будівельників, й об'єкт директора. Перед початком будівництва клієнт повинен зв'язати певного будівельника з директором. Це можна зробити або через конструктор, або через сетер, або подавши будівельника безпосередньо до будівельного методу директора.
6. Результат будівництва можна повернути з директора, але тільки якщо метод повернення продукту вдалося розмістити в загальному інтерфейсі будівельників. Інакше ви жорстко прив'яжете директора до конкретних класів будівельників.

Переваги та недоліки

- Дозволяє створювати продукти покроково.
- Дозволяє використовувати один і той самий код для створення різноманітних продуктів.
- Ізолює складний код конструювання продукту від його головної бізнес-логіки.

- Ускладнює код програми за рахунок додаткових класів.
- Клієнт буде прив'язаний до конкретних класів будівельників, тому що в інтерфейсі будівельника може не бути методу отримання результату.

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).
- **Будівельник** концентрується на будівництві складних об'єктів крок за кроком. **Абстрактна фабрика** спеціалізується на створенні сімейств пов'язаних продуктів. *Будівельник* повертає продукт тільки після виконання всіх кроків, а *Абстрактна фабрика* повертає продукт одразу.
- **Будівельник** дозволяє покроково конструювати дерево **Компонувальника**.
- Патерн **Будівельник** може бути побудований у вигляді **Мосту**: *директор* гратиме роль абстракції, а *будівельники* — реалізації.
- **Абстрактна фабрика**, **Будівельник** та **Прототип** можуть реалізовуватися за допомогою **Одинака**.