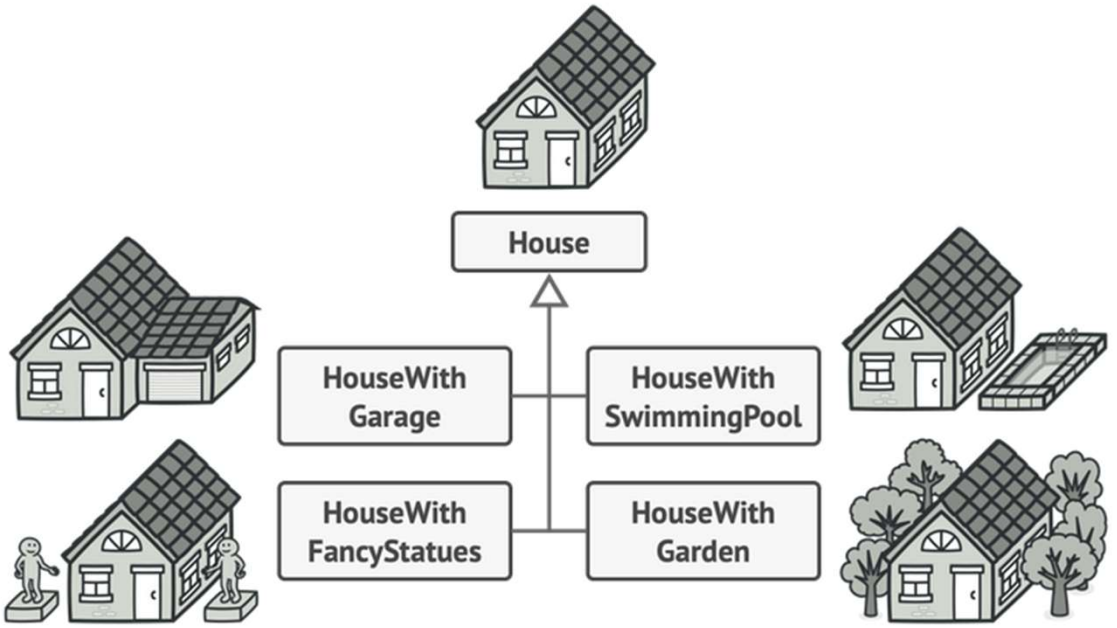
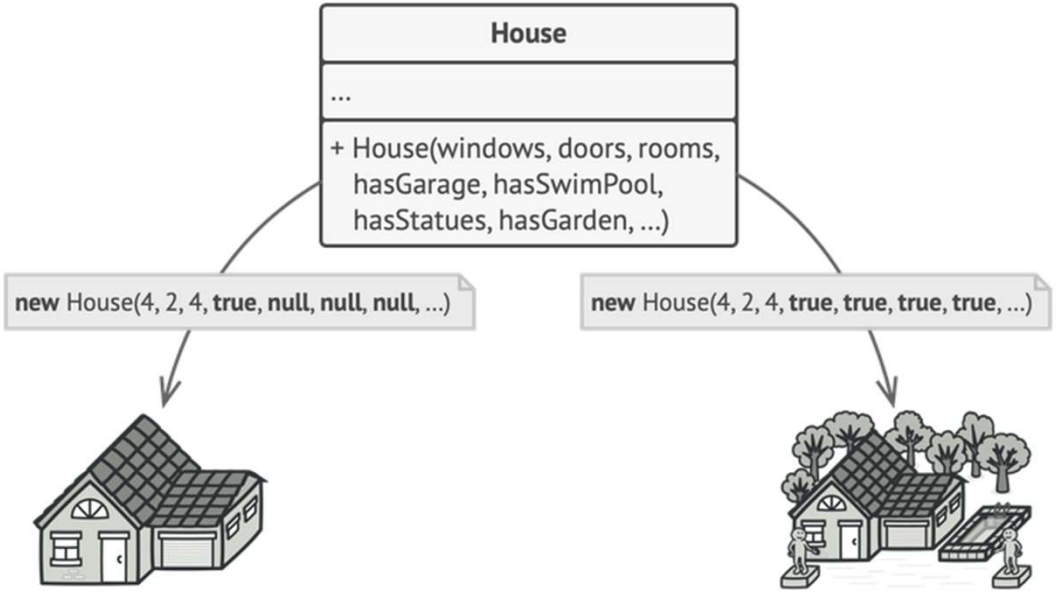
Builder

У чому проблема?



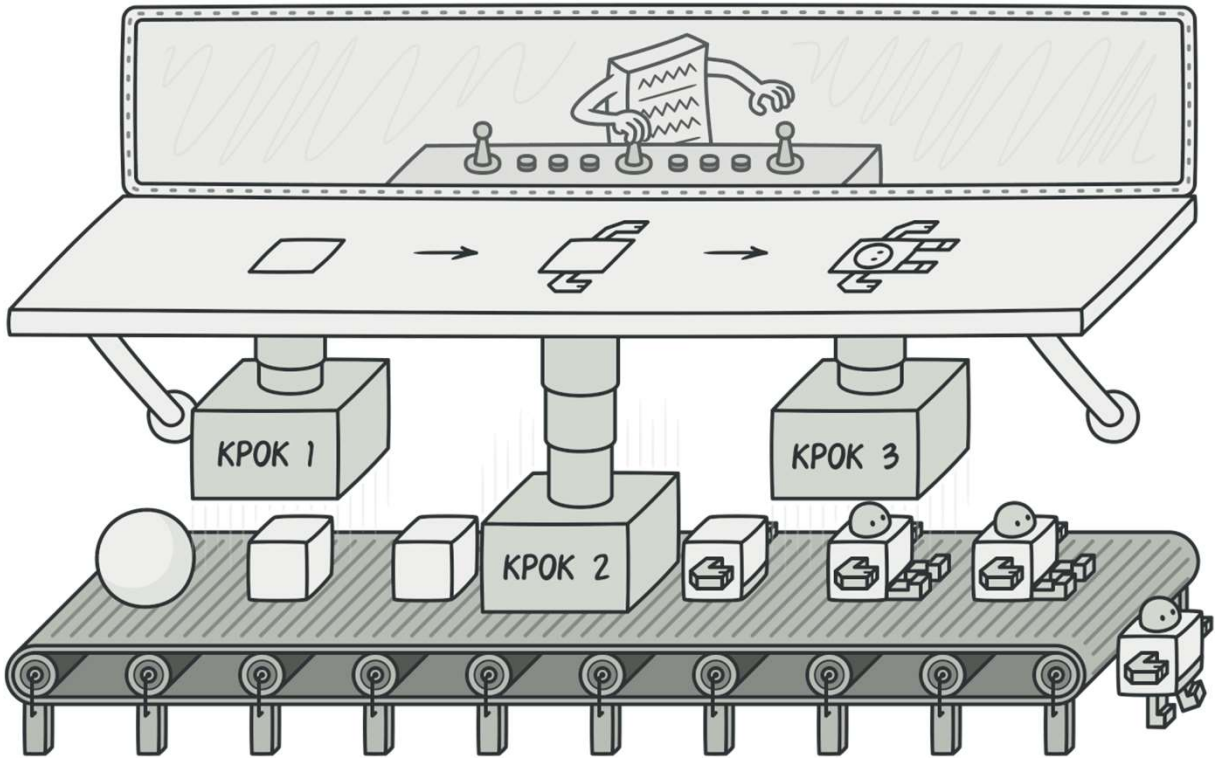
Створивши купу підкласів для всіх конфігурацій об'єктів, ви можете надміру ускладнити програму.

Уявіть складний об'єкт, що вимагає кропіткої покрокової ініціалізації безлічі полів та вкладених об'єктів. Код ініціалізації таких об'єктів зазвичай захований усередині монструозного конструктора з десятком параметрів. Або ще гірше - розпорошений по всьому клієнтському коду.

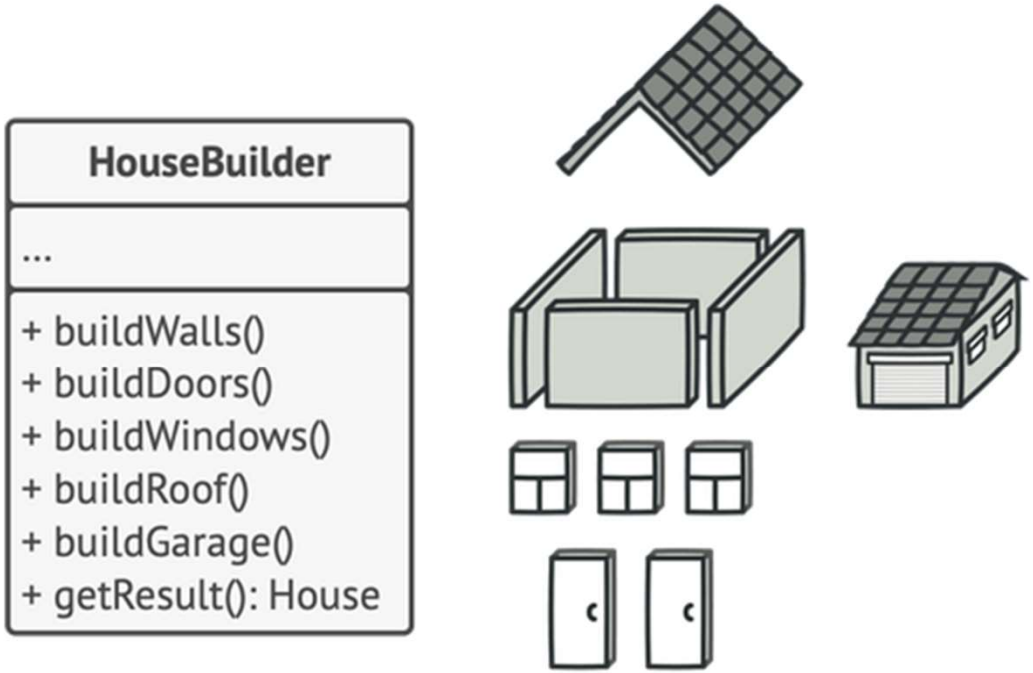


Конструктор з безліччю параметрів має свій недолік: не всі параметри потрібні протягом більшої частини часу.

Суть патерна



Уявімо, що ми маємо конвеєр для випуску автомобілів. Суть конвеєра полягає у поетапній побудові складного продукту, яким у цьому випадку є автомобіль. Конвеєр визначає загальну послідовність кроків (тобто алгоритм) конструювання. При цьому, специфіка кожного з кроків визначається, головним чином,моделлю автомобіля, що збирається. Такий розподіл загального алгоритму побудови та специфіки кожного з етапів дозволять компанії значно заощадити: на одному конвеєрі можуть випускатися автомобілі різних моделей з різними характеристиками.

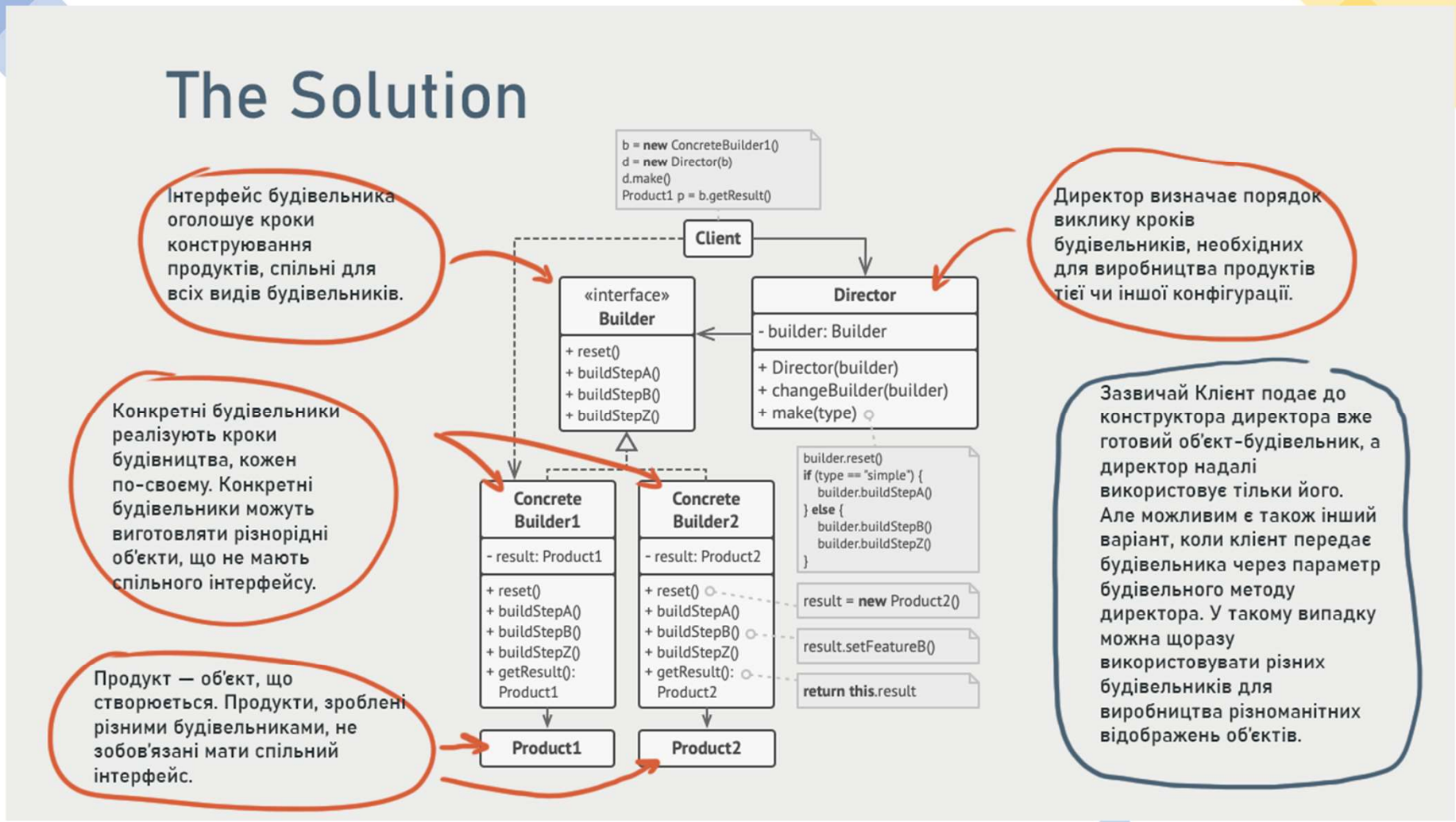


Будівельник дозволяє створювати складні об'єкти покроково. Проміжний результат захищений від стороннього втручання.

Директор знає, які кроки повинен виконати об'єкт-будівельник, щоб виготовити продукт.

- Builder (ТехнологіяМоделі) — будівельник
  - ▷ Забезпечує інтерфейс для поетапного конструювання складного об’єкта (проду
- ConcreteBuilder (ТехнологіяМініАвто та ін.) — конкретний будівельник
  - ▷ Реалізує етапи побудови складного об’єкта, визначені у базовому класі Builder.
  - ▷ Створює результат побудови (Product) і слідкує за покроковим конструюванням.
  - ▷ Визначає інтерфейс доступу до результату конструювання.
- Director (Конвеєр) — розпорядник
  - ▷ Визначає загальний алгоритм конструювання, використовуючи реалізації окремих кроків можливості класу Builder
- Product (МініАвто та ін.) — продукт
  - ▷ Складний об’єкт, що виходить у результаті конструювання.

The Solution



⚖ 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.

Використання анотації Lombok @Builder

Першим і єдиним кроком є додавання анотації до оголошення класу:

```
@Getter
@Builder
public class Widget {
    private final String name;
    private final int id;
}
```

Ломбок виконує всю роботу за нас. Тепер ми можемо створити *віджет* і протестувати його:

```
Widget testWidget = Widget.builder()
  .name("foo")
  .id(1)
  .build();

assertThat(testWidget.getName())
  .isEqualTo("foo");
assertThat(testWidget.getId())
  .isEqualTo(1);
```

Поєднання 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**.