1. Intention Métier du Builder

Le Pattern Builder est un **Pattern de Création** dont l'objectif est de :

> **"Séparer la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction puisse créer différentes représentations."**

Il résout le problème des constructeurs lourds et illisibles (ou **Constructeurs Télescopiques**) en permettant la construction d'un objet étape par étape, sans que le code client n'ait à connaître les détails internes de l'assemblage.

2. Diagramme de Classes Générique (GoF)

Le Pattern Builder s'articule autour de quatre acteurs principaux (voir Diagramme UML sur vos slides):

- **Client**
- **Builder** (<<interface>>>)
- **ConcreteBuilder** (un ou plusieurs)
- **Director**
- **Product** (un ou plusieurs)

Principe de Flux : Le **Client** crée un **ConcreteBuilder** et l'injecte dans le **Director**. Le **Director** ordonne ensuite les étapes de construction. Le **Client** récupère finalement l'objet assemblé via la méthode getResult() du **ConcreteBuilder**.

3. Description des Classes Participantes

Rôle	Description
Product	L'objet complexe en cours de construction et le résultat final (ex : une House).
Builder	Interface ou classe abstraite qui **définit l'ensemble des étapes** de construction (buildWalls(), buildDoors(), getResult(), etc.).
ConcreteBuilder	Implémente l'interface Builder. Il sait comment assembler les pièces et détient l'objet Product en interne. Chaque

	ConcreteBuilder peut aboutir à une représentation différente (ex : WoodenHouseBuilder).
Director	**Contrôle l'ordre** d'exécution des étapes. Il connaît les recettes (ex : constructSimpleHouse()). Il rend la construction interchangeable car il travaille avec l'interface Builder

4. Exemple dans un Contexte Métier (La Construction de Maisons)

Problème initial : Sans Builder, la construction d'une maison avec des options variables (garage, piscine, jardin, statues) conduit soit à une multitude de sous-classes (HouseWithGarage, HouseWithGarden), soit à un constructeur unique trop long et illisible qui nécessite de passer beaucoup de valeurs nulles ou par défaut

Solution avec Builder :

1. L'objet House (le **Product**) contient toutes ses parties.
2. L'interface HouseBuilder définit les méthodes : buildWalls(), buildRoof(), buildGarage(), getResult().
3. Le Director propose des méthodes comme constructSimpleHouse() (qui appelle juste buildWalls() et buildRoof()) ou constructLuxuryHouse() (qui appelle toutes les méthodes, y compris buildSwimmingPool()).
4. Le Client peut facilement créer n'importe quelle configuration en utilisant le Director et un ConcreteBuilder (ex: StoneHouseBuilder).

5. Explication de la SOLIDité

Le Pattern Builder est une excellente illustration des principes

SOLID, qui rendent le code plus robuste et extensible:

- **SRP (Single Responsibility Principle)** : Chaque rôle a une seule responsabilité. Le **Director** est responsable de l'ordre, le **ConcreteBuilder** est responsable de l'assemblage, et le **Product** est responsable de ses données.
- **OCP (Open/Closed Principle)** : Le code est **ouvert à l'extension** mais **fermé à la modification**. On peut ajouter un nouveau type de construction (ConcreteBuilder3) sans avoir à modifier le Director ou les autres Builders.
- **DIP (Dependency Inversion Principle)** : Le **Director** dépend de l'**abstraction** (Builder interface) et non des classes de construction concrètes (ConcreteBuilder1). Ceci assure le découplage.