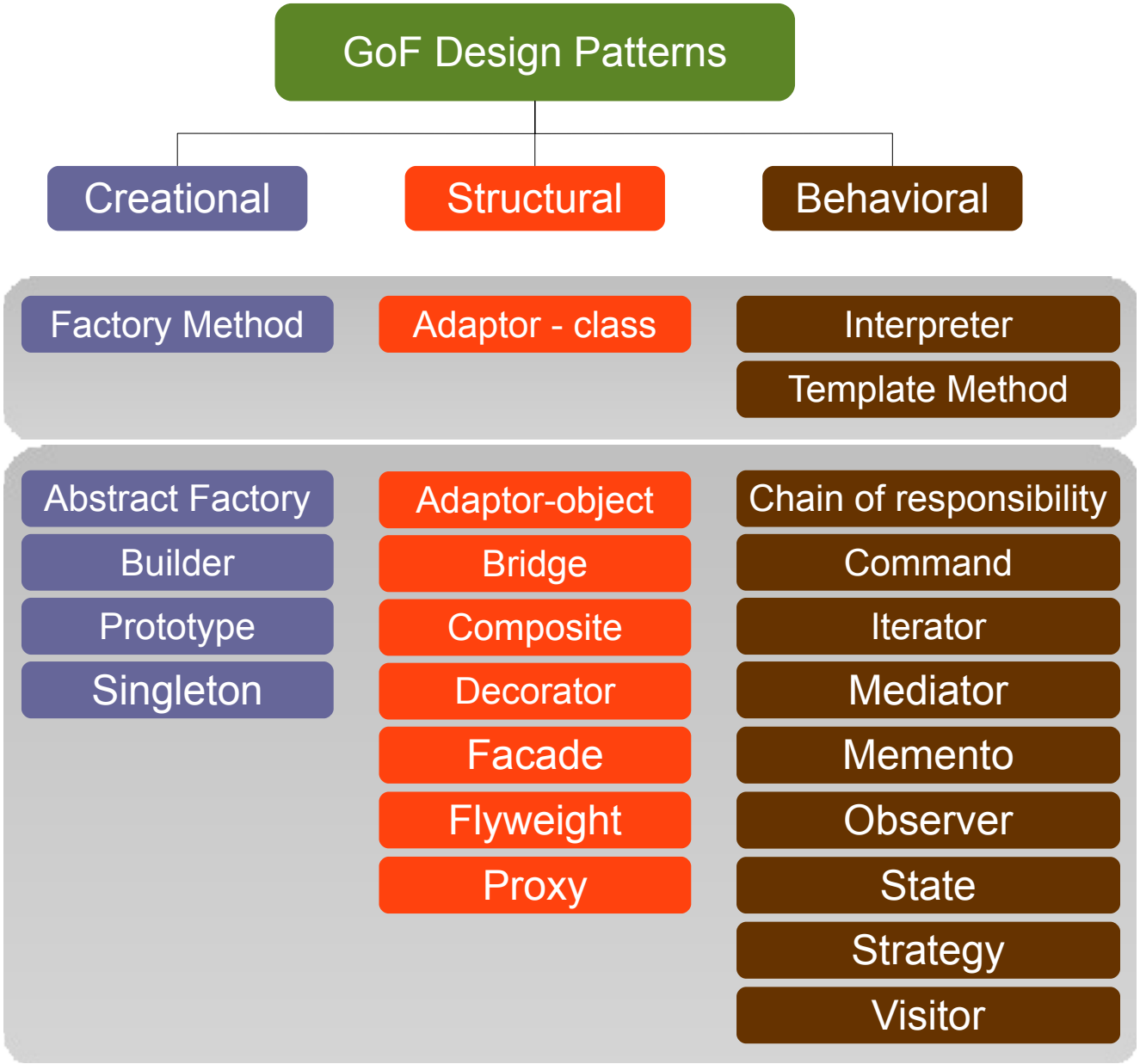
Classification des patrons de conception



Outline

Factory Method [2] - Méthode de Fabrique

- Intention :
  - Fournir une **interface** pour créer des objets dans une classe mais en permettant aux sous-classes de **modifier** le type d'objets qui seront créés
- Exemple : vous construisez une application pour la gestion logistique
  - Pour le premier temps, l'application se focalise seulement sur le transport par camions

2. Also know as : Virtual Constructor

Factory Method - Exemple - Solution

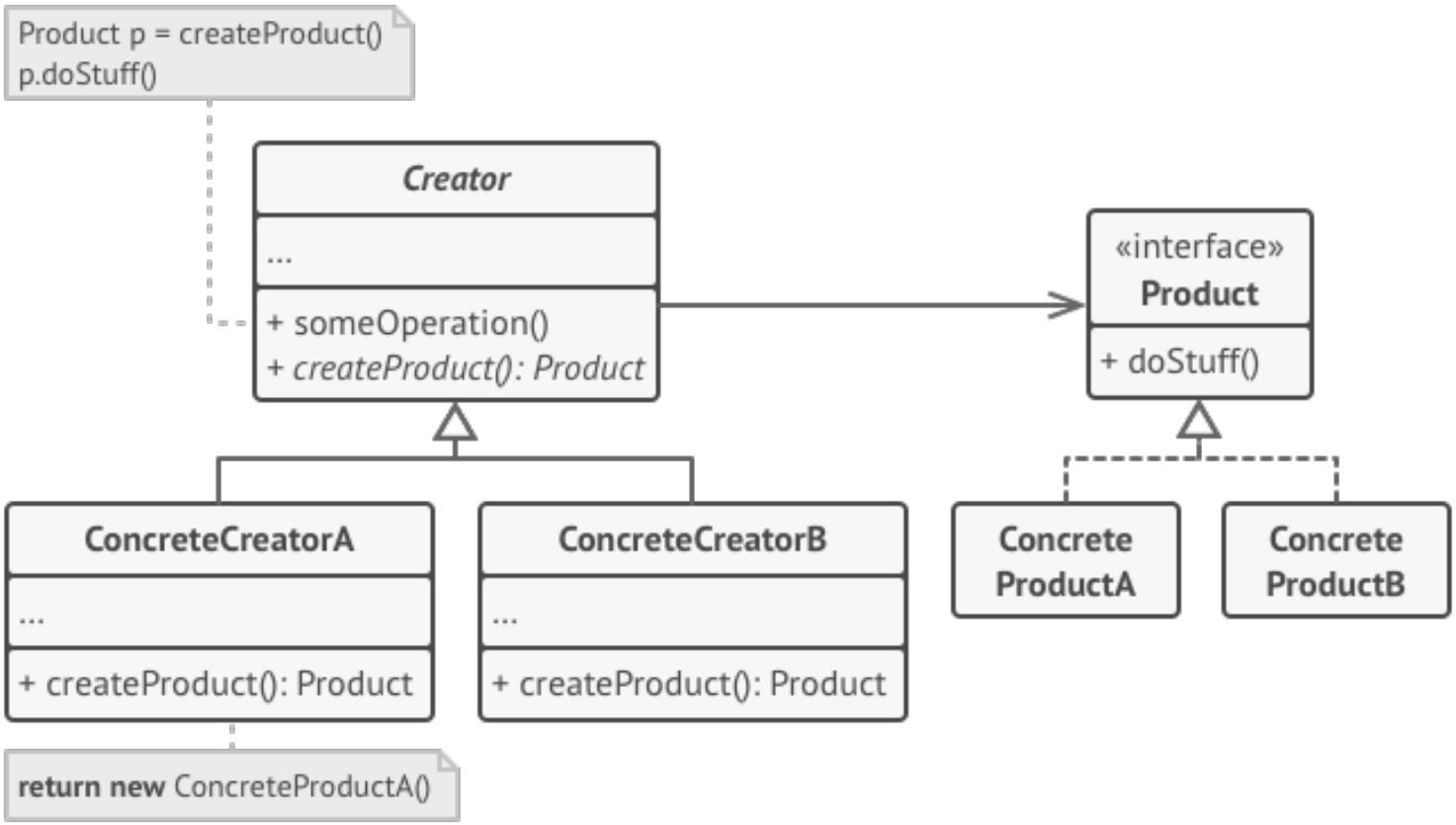
Vous voulez :

- Un mécanisme pour créer des objets de transport sans connaître au préalable le type exact (i.e., `Ship` ou `Truck` ou au future `Avion` etc.)
- L'application ne connaît pas le détail des concrets classes (i.e., `Ship`, `Truck`), et la différence entre ces classes. L'application les considère comme une abstraite `Transport`
  - Dans l'avenir, si on a besoin un autre type `Avion` $\Rightarrow$ pas de problème avec l'application, comme elle le considère également comme un `Transport`

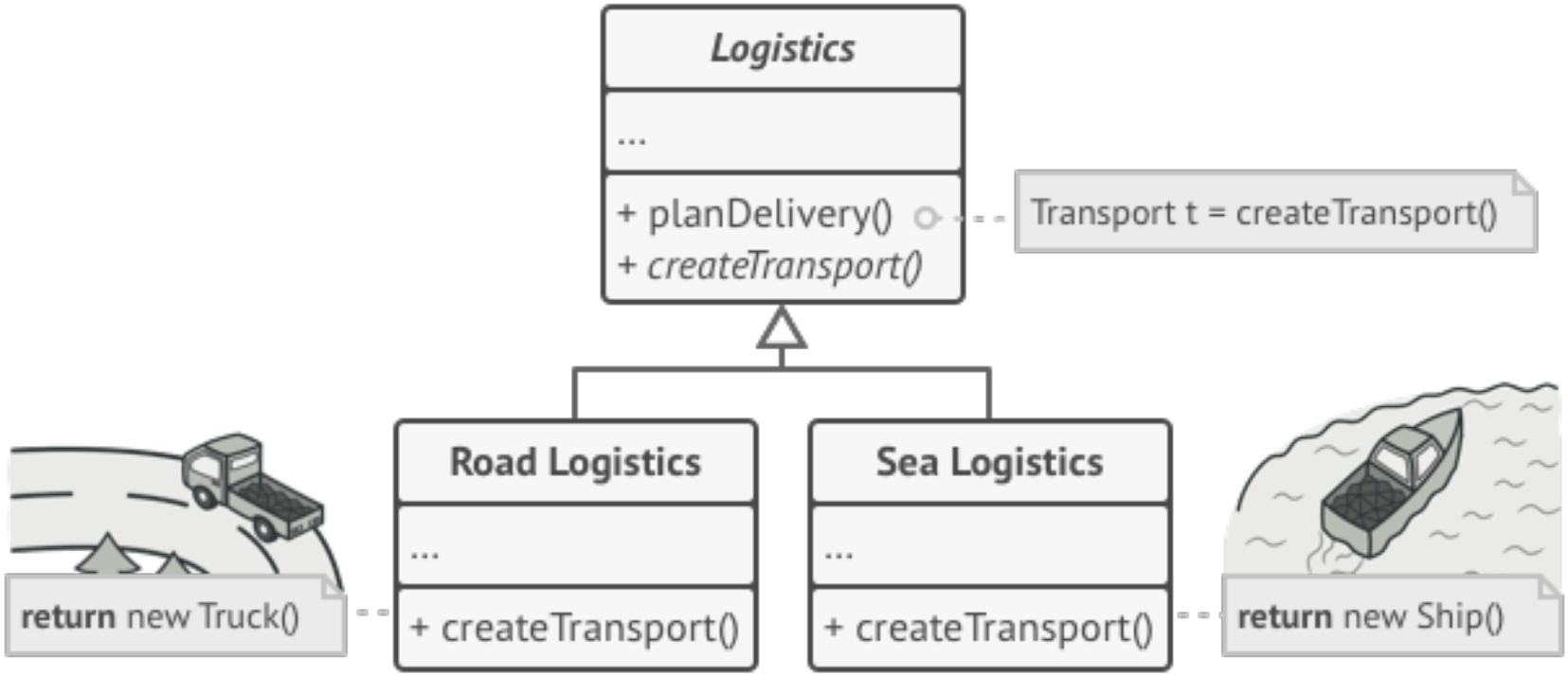
Factory Method - Structure

Vous utilisez Factory Method pour le faire :

- Vous créez une méthode de fabrique (*factory method*) dont le rôle est pour créer des objets concrets selon le besoin de l'utilisateur.
- Objets créés par cette méthode sont appelés *produits* (*products*)

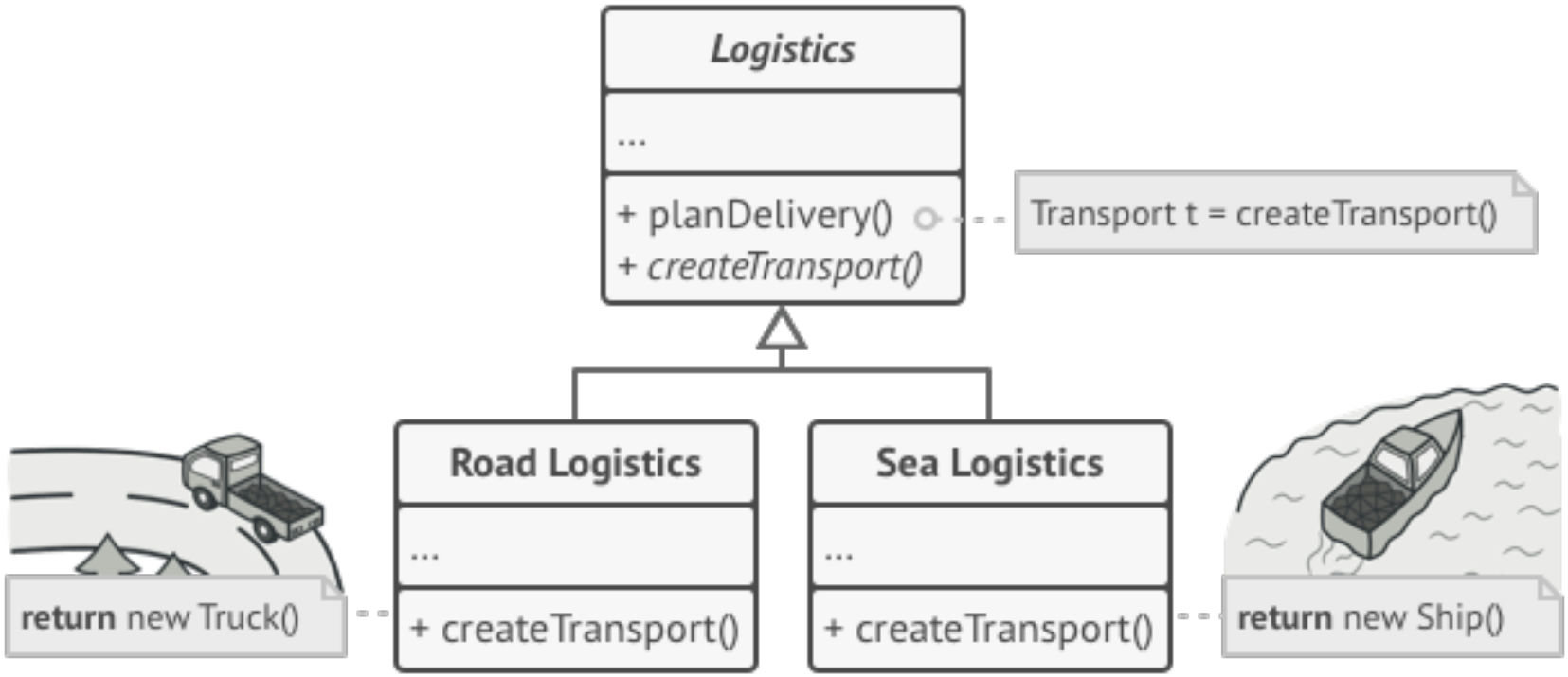


Factory Method - Exemple - Solution



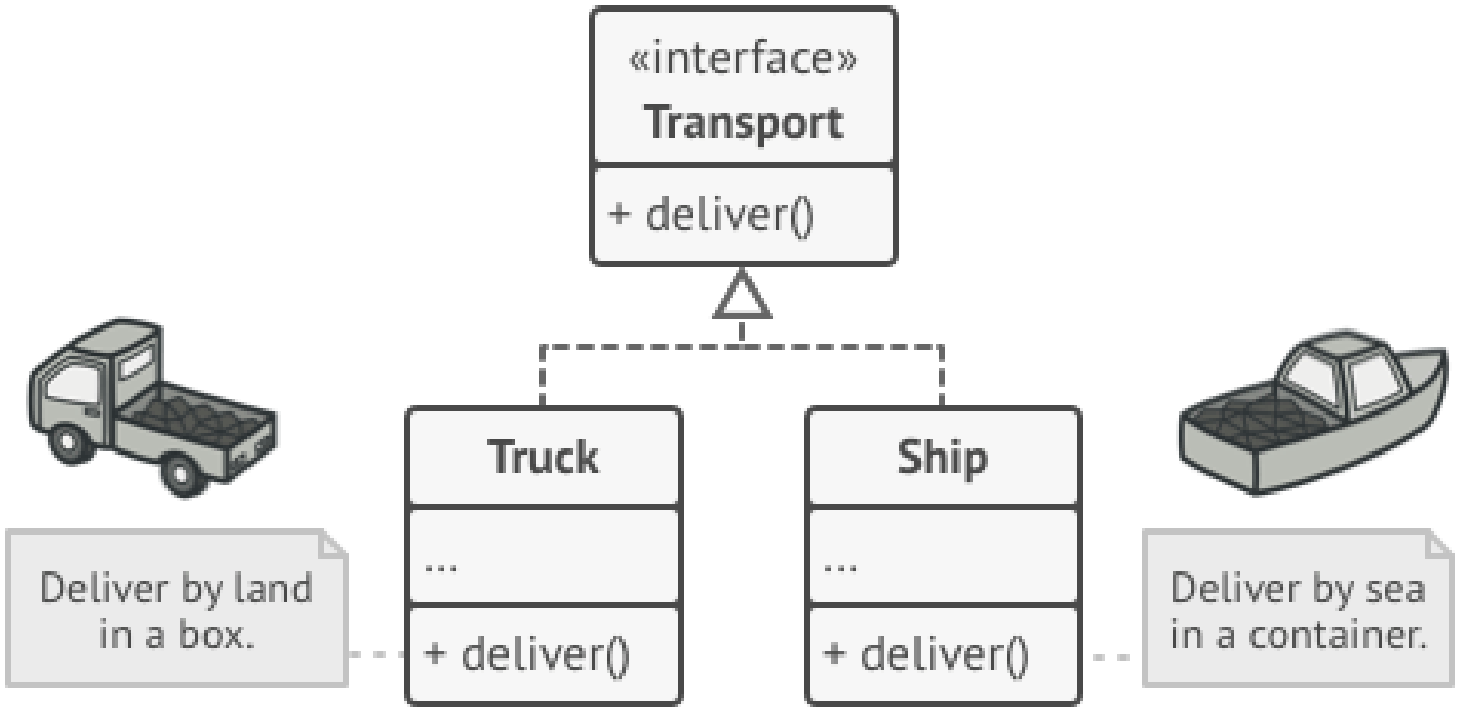
- Le créateur de base (`Logistics`) connaît seulement le produit abstrait `Transport` qui est déclaré comme une *interface* et fournit une certaine méthode commune (i.e., `deliver`)
- Le type de retour de la méthode de fabrique de base doit être déclaré comme `Transport`

Factory Method - Exemple - Solution



- Le type exact de `Transport` est créé par la méthode de fabrique `createTransport` des créateurs concrets (`RoadLogistics`, `SeaLogistic`)

Factory Method - Exemple - Solution



- Tous les produits concrets va implémenter la méthode commune `deliver` de façon différente

Factory Method - Applicabilité

2. Vous utilisez la méthode de fabrique lorsque vous voulez permettre aux utilisateurs qui utilise votre bibliothèque ou votre framework de pouvoir facilement étendre les composants internes
   - L'héritage est probablement le moyen le plus simple pour étendre le comportement d'une bibliothèque ou d'un framework.
   - Mais comment le framework peut reconnaître que votre sous-classe devrait être utilisée à la place d'un composant standard ?

Solution

- Créer une sous classe de `UIFramework` : `UIWithRoundButtons`
- Override la méthode `createButton` pour créer un objet de `RoundButton` au lieu de `Button`
- Utiliser `UIWithRoundButtons` au lieu de `UIFramework`

Factory Method - Applicabilité

3. Vous utilisez la méthode de fabrique lorsque vous souhaitez enregistrer des ressources du système en réutilisant des objets existants au lieu de les reconstruire à chaque fois
   - Les objets sont créés par des méthodes *constructor* (i.e., utiliser le keyword `new`) mais il conduit à la création de nouveaux objets chaque fois d'appel
   - Il faut donc avoir un autre méthode qui permet de créer de nouveaux objets ou de réutiliser des objets existants $\Rightarrow$ méthode de fabrique

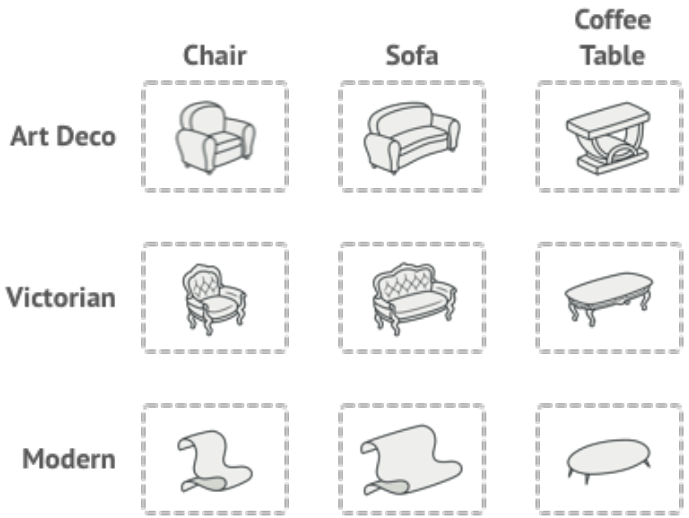
Outline

Abstract Factory - Fabrique abstrait

Abstract Factory est un modèle de conception créative qui vous permet de produire des familles d'objets associés sans spécifier

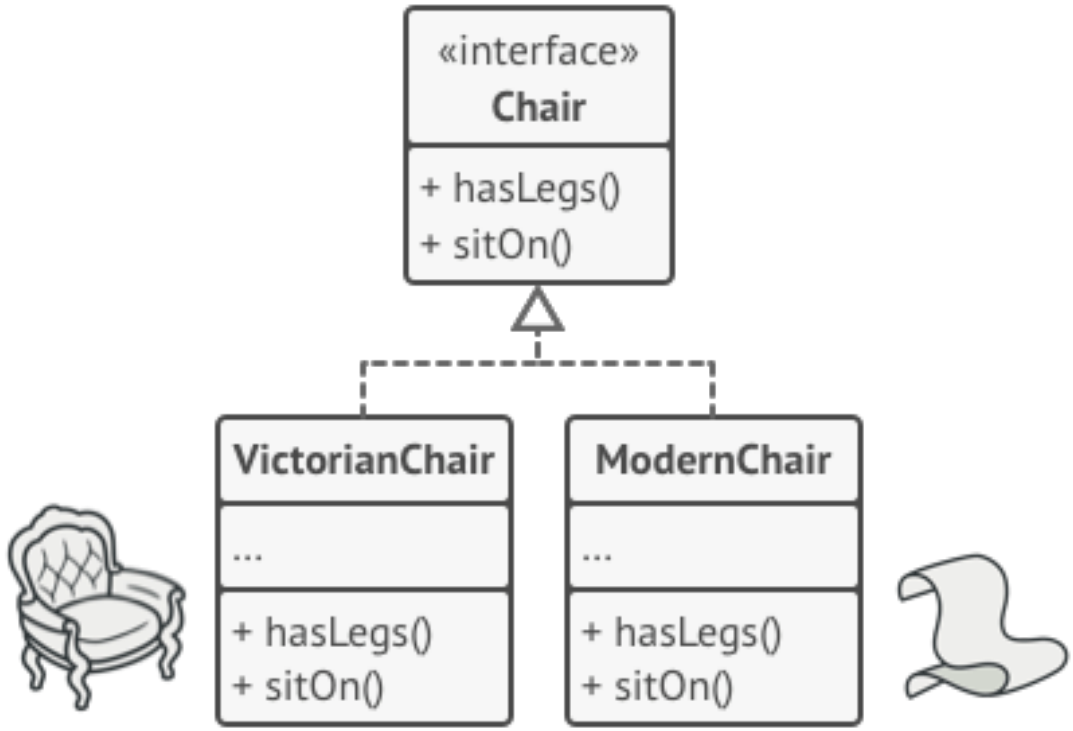
Etude de cas : Un simulateur de magasin de meubles. Votre code se compose de classes qui représentent :

- Une famille d'objets associés : `Chair` + `Sofa` + `CoffeeTable`
- Plusieurs variantes de cette famille. Par exemple :
  - Modern
  - Victorian
  - ArtDeco



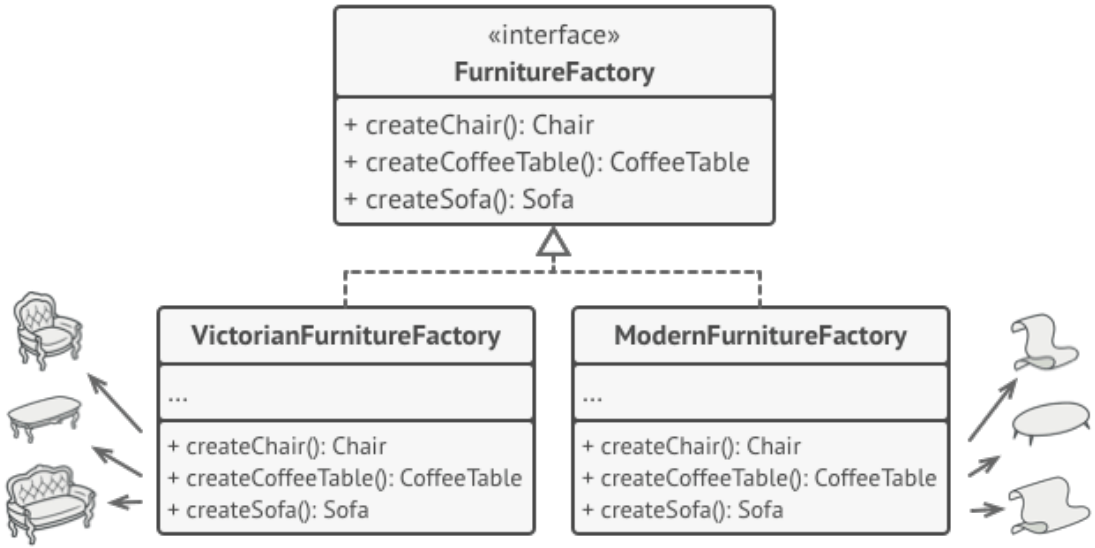
Abstract Factory - Solution

- Pour chaque produit distingué, il faut créer une interface : `Chair`, `Sofa`, `CoffeeTable`
- Chaque variante du produit va suivre son interface : toutes les variantes de `Chair` va implémenter `Chair` etc.



Abstract Factory - Solution

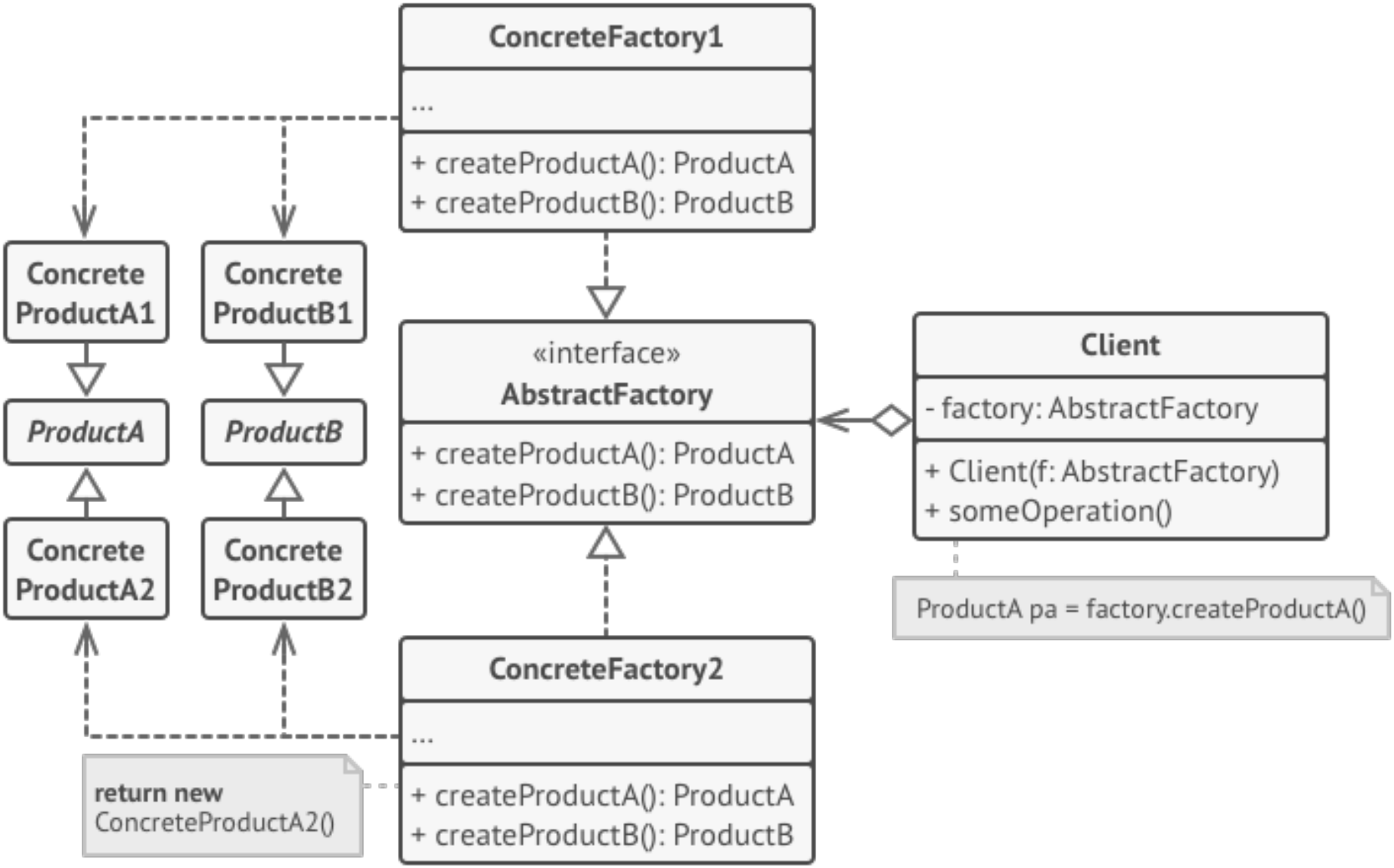
- Pour pouvoir facilement créer une famille des produits, il faut avoir un fabrique abstrait qui fournit toutes les méthodes nécessaires `FurnitureFactory` : `createChair`, `createSofa`, `createTable`
- Le type de retour de ces méthodes devrait être abstrait représenté par des interfaces
- Chaque concrete fabrique permet de créer des concret variantes des produits : `VictorianFurnitureFactory`, `ModernFurniture` etc.



Abstract Factory - Solution

- Le client code veut produire un `chair`. Le client n'a pas connu ni la class de fabrique ni le type de `chair` qu'il reçoit.
- Quelque soit un modèle `modern` ou `victorian`, le client doit traiter tous les `chair` de la même manière, en utilisant l'interface abstrait `Chair`
- Avec cette approche, la seule chose que le client sait du produit `chair` est qu'elle implémente la méthode `sitOn` d'une manière ou d'une autre
- Quelque soit la variante de `chair` retournée, elle correspondra toujours au type de `sofa` ou `table` créés par la même classe de fabrique

Abstract Factory - Structure



Abstract Factory - Structure

1. **Produits Abstraits** déclarent des interfaces pour un ensemble de distingués mais reliés produits qui permettent de créer une famille de produits : `ProductA`, `ProductB`
2. **Produits concrets** sont diverses implémentations de produits abstraits, regroupés par variantes. Chaque produit abstrait doit être mis en œuvre dans toutes les variantes données : `ConcreteProductA1`, `ConcreteProductB1` etc.
3. **Fabrique Abstrait** est déclaré comme une interface fournissant un ensemble de méthodes pour créer chacun des produits abstraits : `AbstractFactory`
4. **Fabriques Concrets** implémentent des méthode de création du fabrique abstrait. Chaque fabrique concret correspond à une variante spécifique des produits et permet de créer une famille de ce type variante des produits : `ConcreteFactory1`, `ConcreteFactory2` etc.

Abstract Factory - Etude de cas

Retourner à l'application multi-plateforme UI :

Problématique

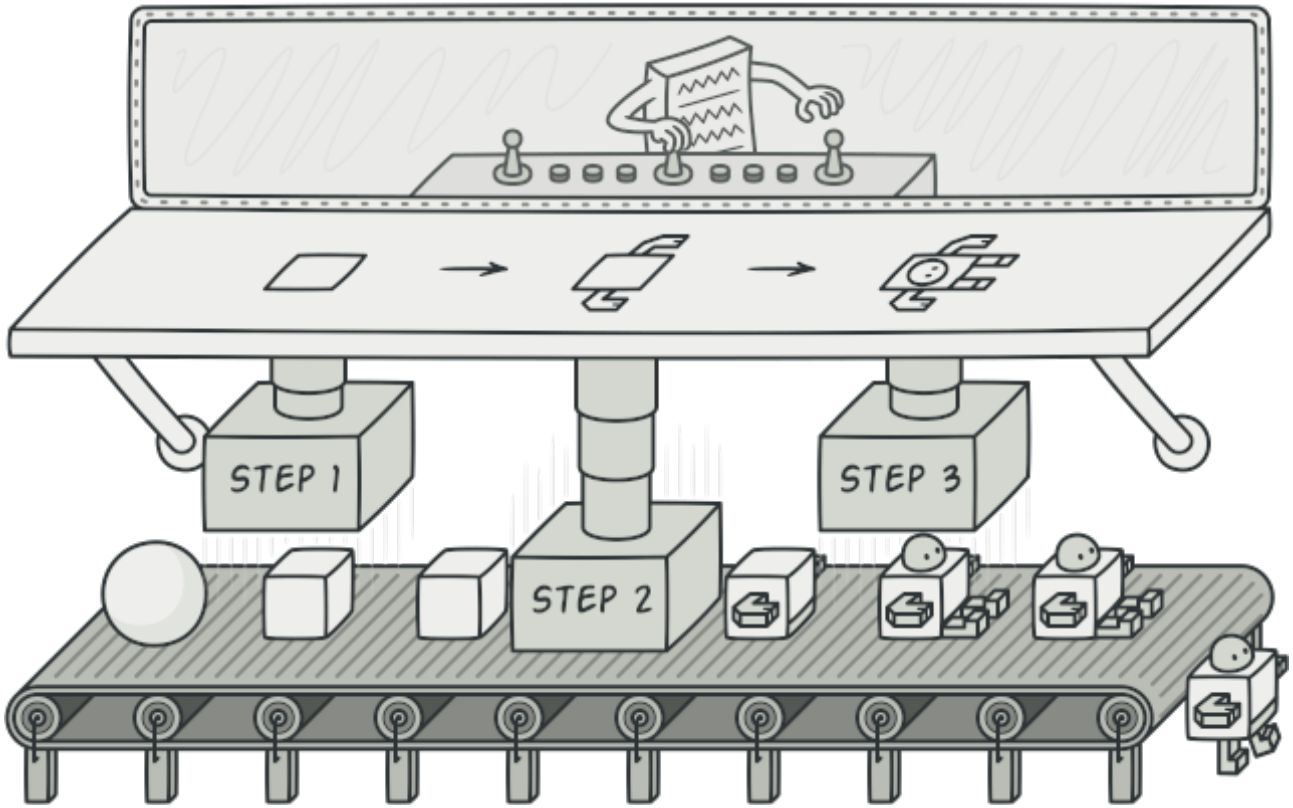
- Vous voulez créer une famille de UI éléments : bouton, case à cocher, éditeur de texte, etc.
- Il existe plusieurs variantes de ces UI éléments qui correspondent à différents plateformes : Windows, MacOS etc.
- Vous allez refactoring la conception dans laquelle nous avons utilisé Factory Method, le remplacez par Abstract Factory

Outline

Builder Pattern

Intention

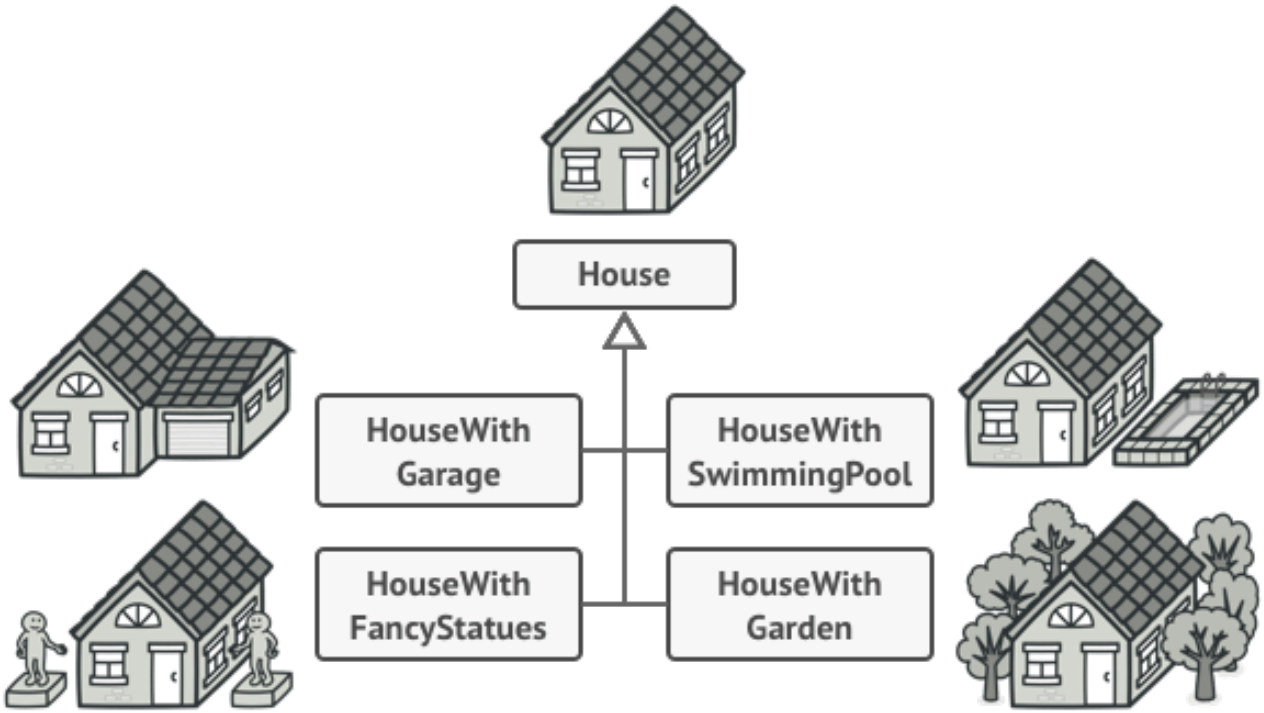
Builder est un modèle de conception qui permet de construire des objets complexes étape par étape. Ce modèle vous permet de produire différents types et représentations d'un objet en utilisant le même code de construction.



Builder - Exemple : Problème

Vous voulez construire une maison :

- Pour une petite et simple maison, il vous faut construire quatre murs et un plancher, installer une porte, installer une paire de fenêtres et construire un toit.
- Qu'est-ce qui se passe si vous voulez une maison plus grande et plus lumineuse avec un garage, une piscine, un jardin etc. ?



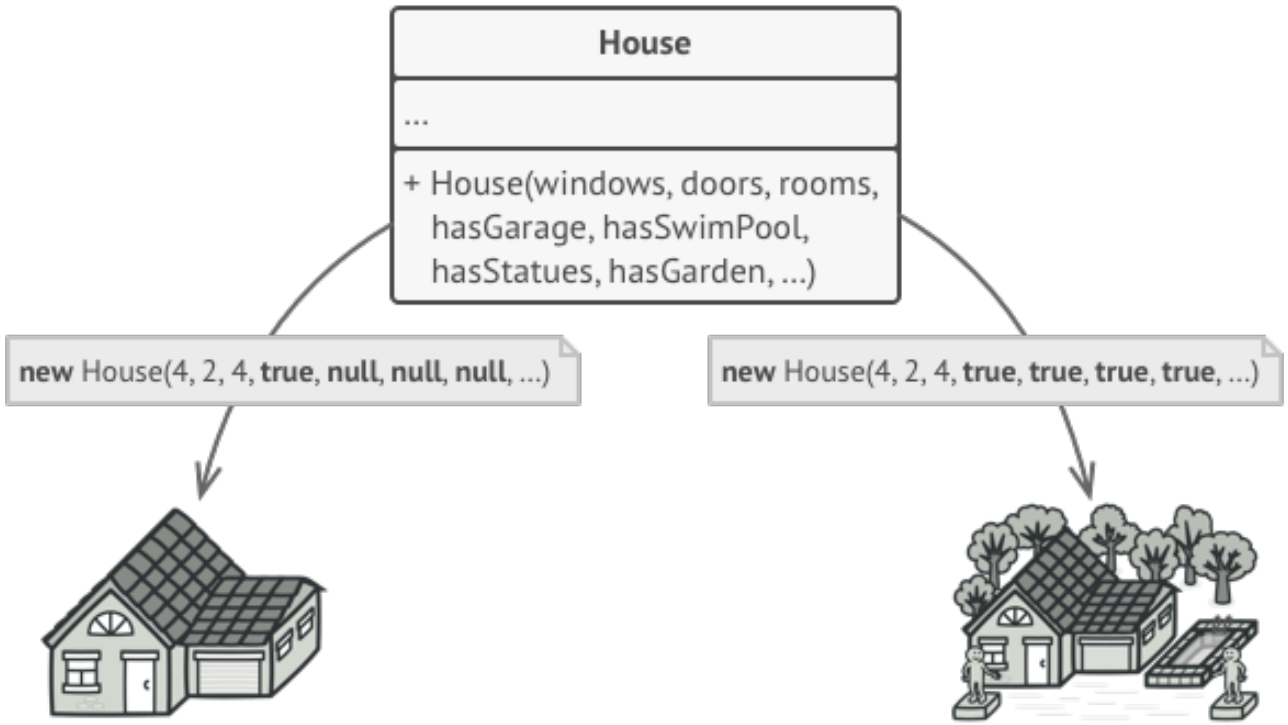
Builder - Example : Problème

Une solution simple peut être :

- Etendre la classe de base `House` en créant un ensemble de sous-classes pour couvrir toutes les combinaisons possibles des paramètres.
- On pourrait avoir besoin énormément sous-classes et élargir l'arbre de hiérarchie quand on a besoin d'un nouveau paramètre pour la maison

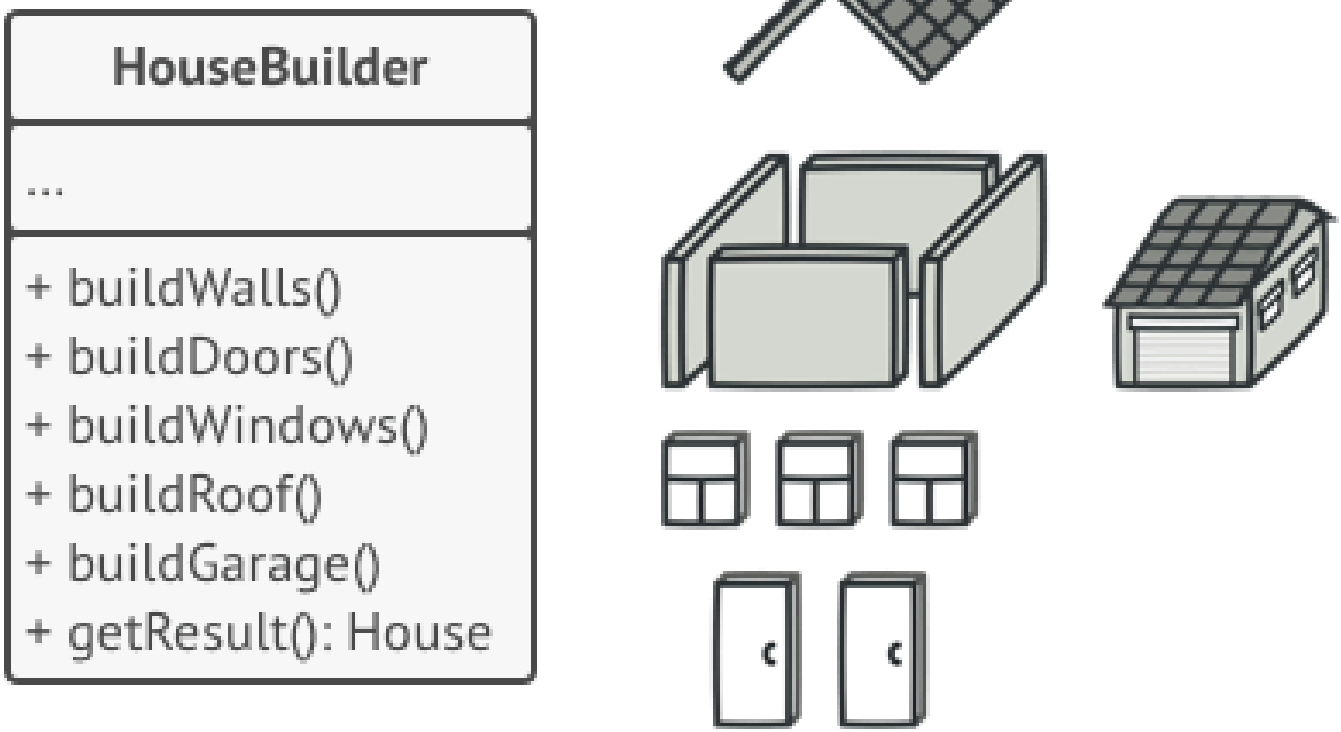
Builder - Example : Problème

- On tombe cependant dans un autre problème : beaucoup de constructeurs, la plupart de paramètres d'un constructeur peuvent être rarement utils (ex., 9/10 des cas on n'a pas besoin d'une maison avec piscine, donc ce paramètre est inutil)
- Au pire de cas, ces paramètres sont dispersés partout dans le code client - ceux qu'il faut éviter



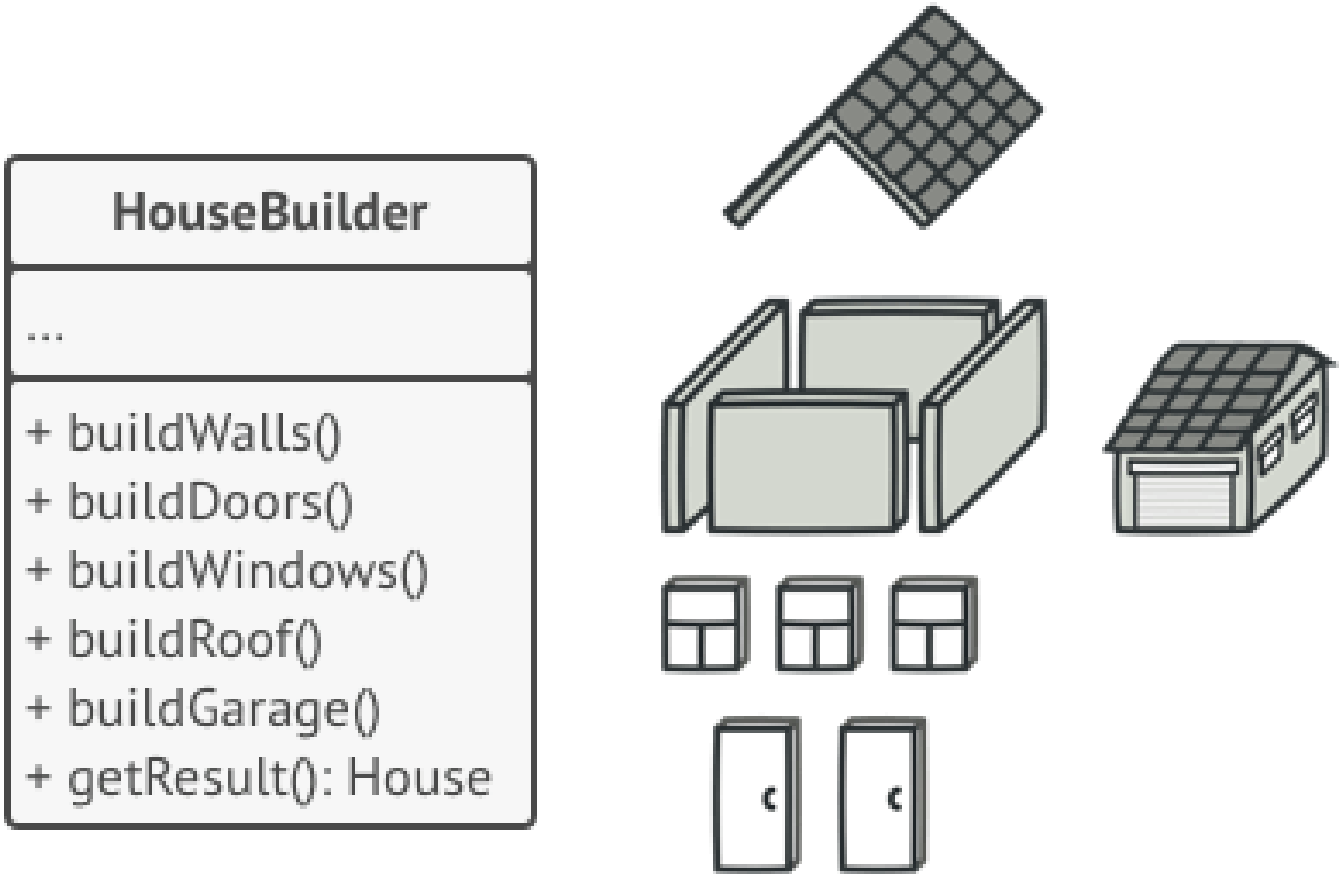
Builder Pattern - Solution

- Le modèle de conception Builder va séparer le code de construction des objets de la classe de lui-même.
- La construction des objets est mise dans les classes spécifiques qui s'appellent *builders*



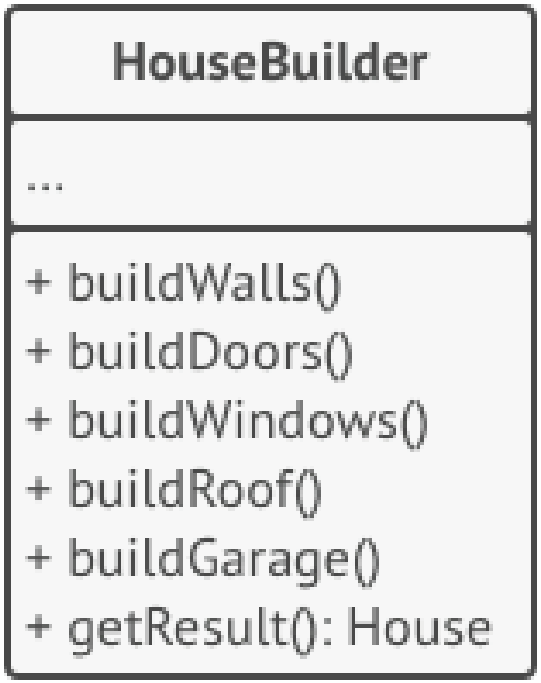
Builder Pattern - Solution

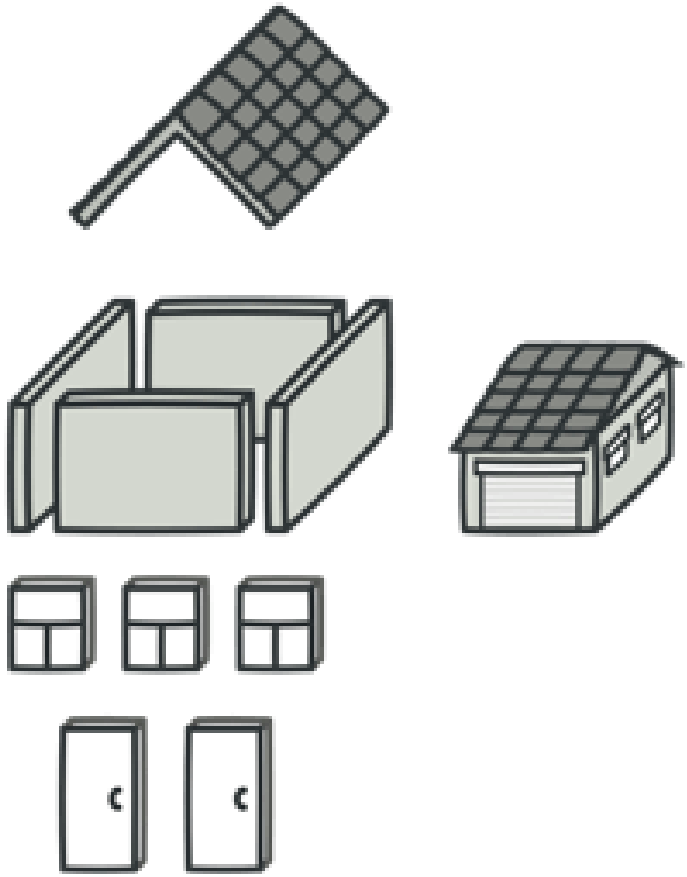
- Une maison va être construite dans un ensemble des étapes : `buildWalls`, `buildDoor` etc.
- Pour créer la maison, on va exécuter une série de ces étapes sur l'objet de la classe `HouseBuilder`



Builder Pattern - Solution

- Ce qui d'important est que l'on n'a pas besoin de suivre toutes les étapes mais peut choisir certains d'entre elles
- On est donc libre de configurer la maison qu'on a besoin de construire

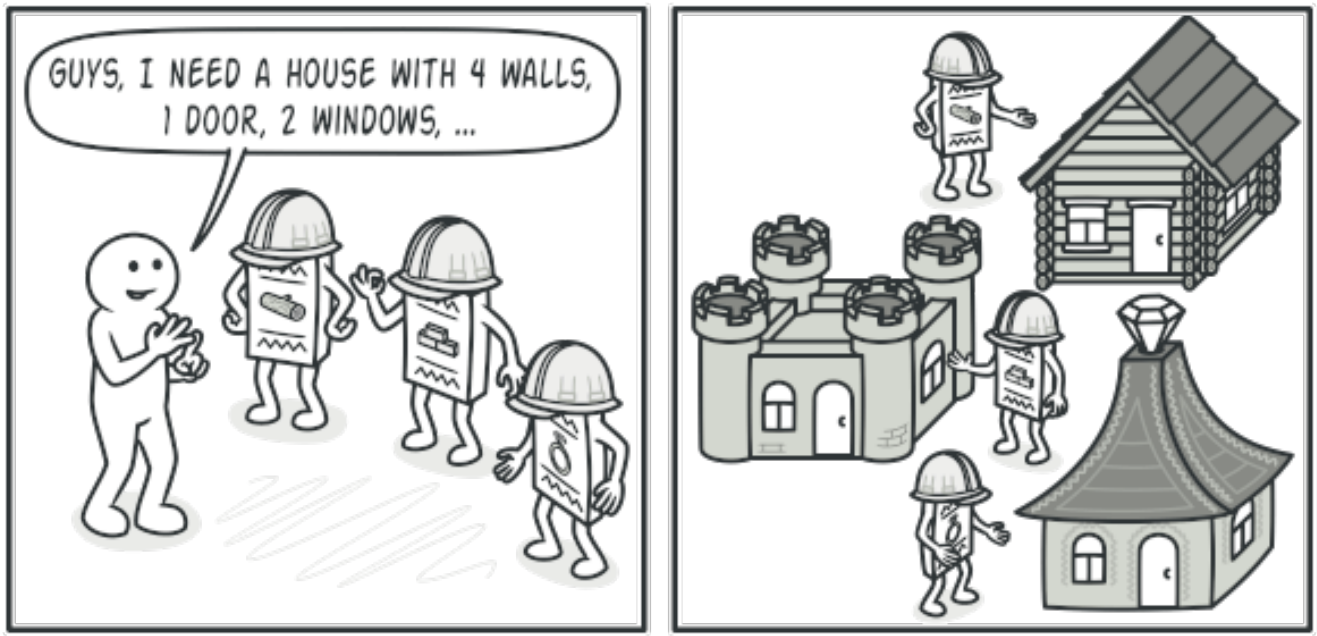




Builder Pattern - Solution

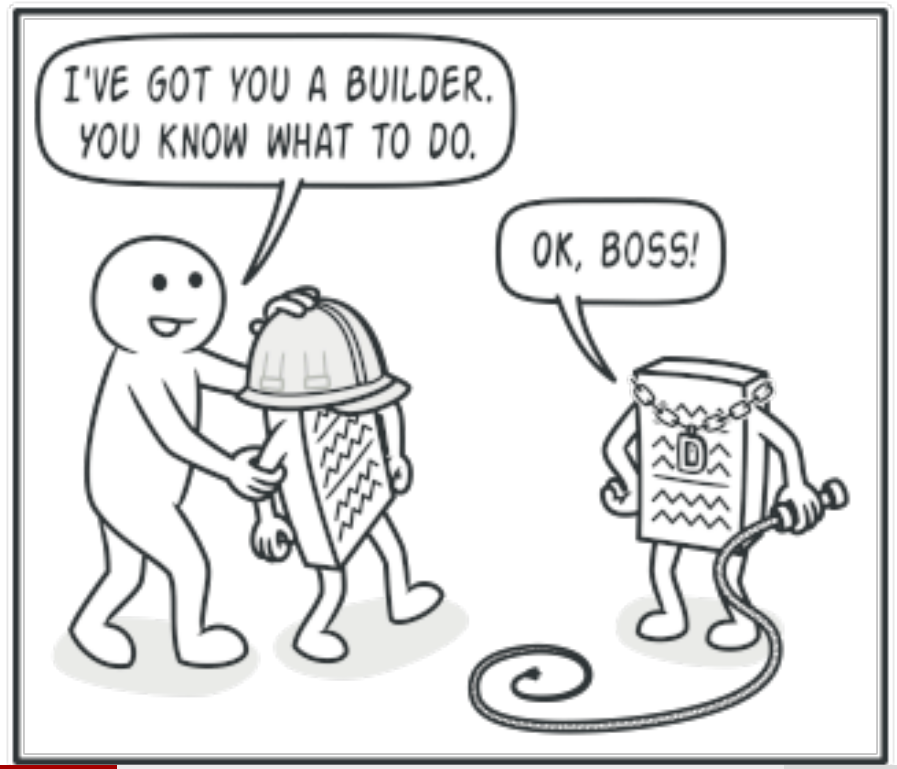
Qu'est-ce qui se passe si on a besoin de varier la représentation de certaines caractéristiques ?

- On peut créer plusieurs classes *builder* différentes qui implémentent le même ensemble d'étapes de construction mais d'une manière différente et les utiliser dans le processus de construction pour produire différents représentation d'objets

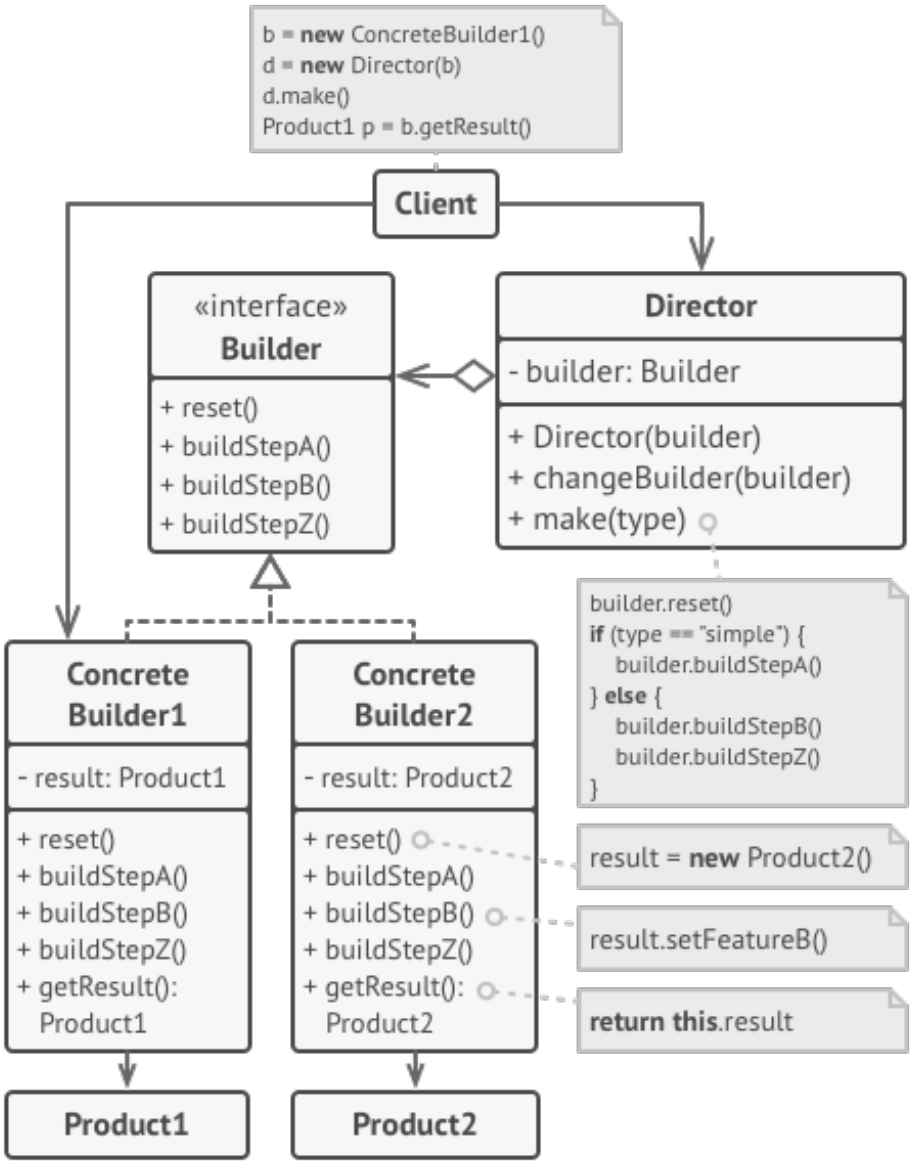


Builder Pattern - Solution

- On a besoin également d'une classe to définir un ordre dans lequel le process de construction est exécuté (étape par étape).
- La classe `Director` ne connaît que la classe abstraite `Builder` dans laquelle il y a des étapes pour construire le produit
- Les concretes `builders` fournissent les implémentations pour ces étapes



Builder Pattern - Structure



Builder Pattern - Structure

1. L'interface `Builder` déclare un ensemble des étapes pour construire le produit. Ces étapes sont communes pour toutes les concretes classes *builder* qui implémentent cette interface
2. Les concretes classes *builder* fournissent différentes implémentations des étapes de construction. Ces classes peuvent également créer des produits qui ne suivent pas l'interface commune : `ConcreteBuilder1`, `ConcreteBuilder2` etc.
3. Les classes `Product1`, `Product2` etc. sont des différents objets construits par les `builder`
4. La classe `Director` définit un ordre dans lequel les étapes de construction sont exécutées $\Rightarrow$ il est donc facile pour réutiliser et varier la configuration de produit
5. Le code client devrait associer un `builder` concret avec le `director`. Ensuite, le directeur utilise cet objet de constructeur pour toute autre construction.

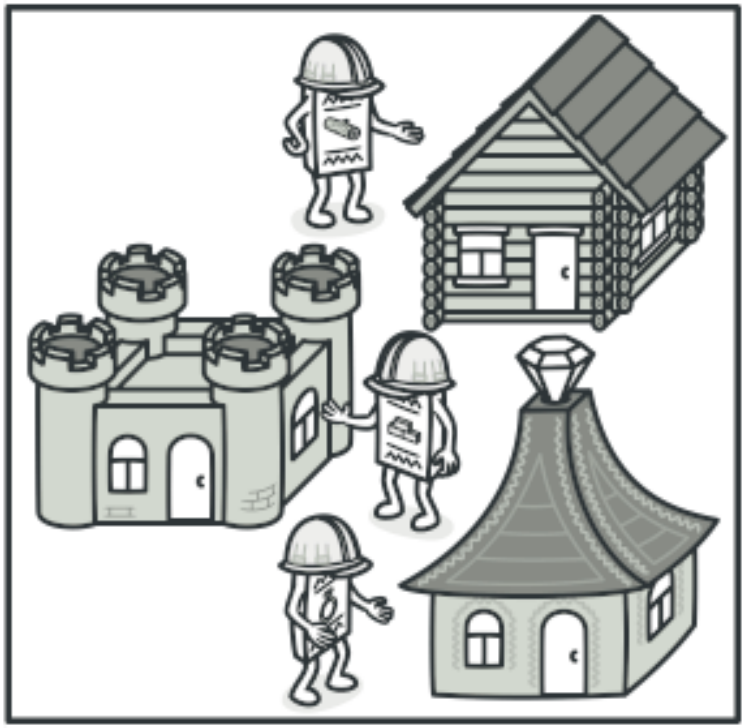
Builder Pattern - Applicabilité

1. Le modèle Builder vous permet d'éviter des constructeurs télescopiques

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

Builder Pattern - Applicabilité

2. Vous utilisez le modèle Builder lorsque vous souhaitez que votre code puisse créer différentes représentations de certains produits (par exemple, des maisons en *pierre* et en *bois*



Builder Pattern - Avantages et Inconvénients

Avantages

- On peut construire complexes objets étape par étape, différer des étapes de construction ou exécuter des étapes de manière récursive
- On peut réutiliser le même code de construction lors de la création de diverses représentation de produits
- On respecte également le principe SRP : on isole un code de construction complexe de la logique du produit

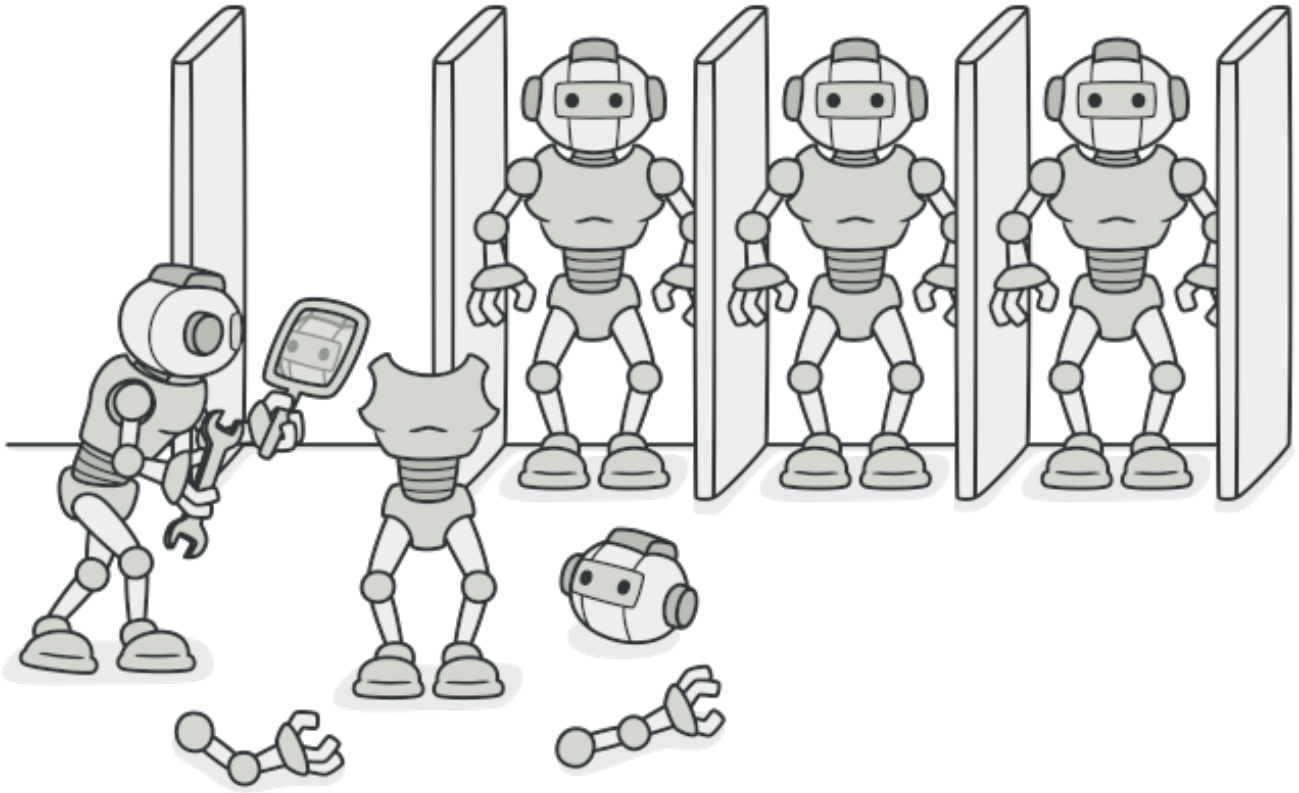
Inconvénients

Par contre, la complexité globale du code augmente comme le modèle nécessite la création de plusieurs nouvelles classes

Outline

Intention

Prototype est un modèle de conception qui permet de copier des objets existants sans rendre votre code dépendant de leurs classes



Prototype Pattern - Problème

On a un objet et on veut créer une exacte copie de cet objet.

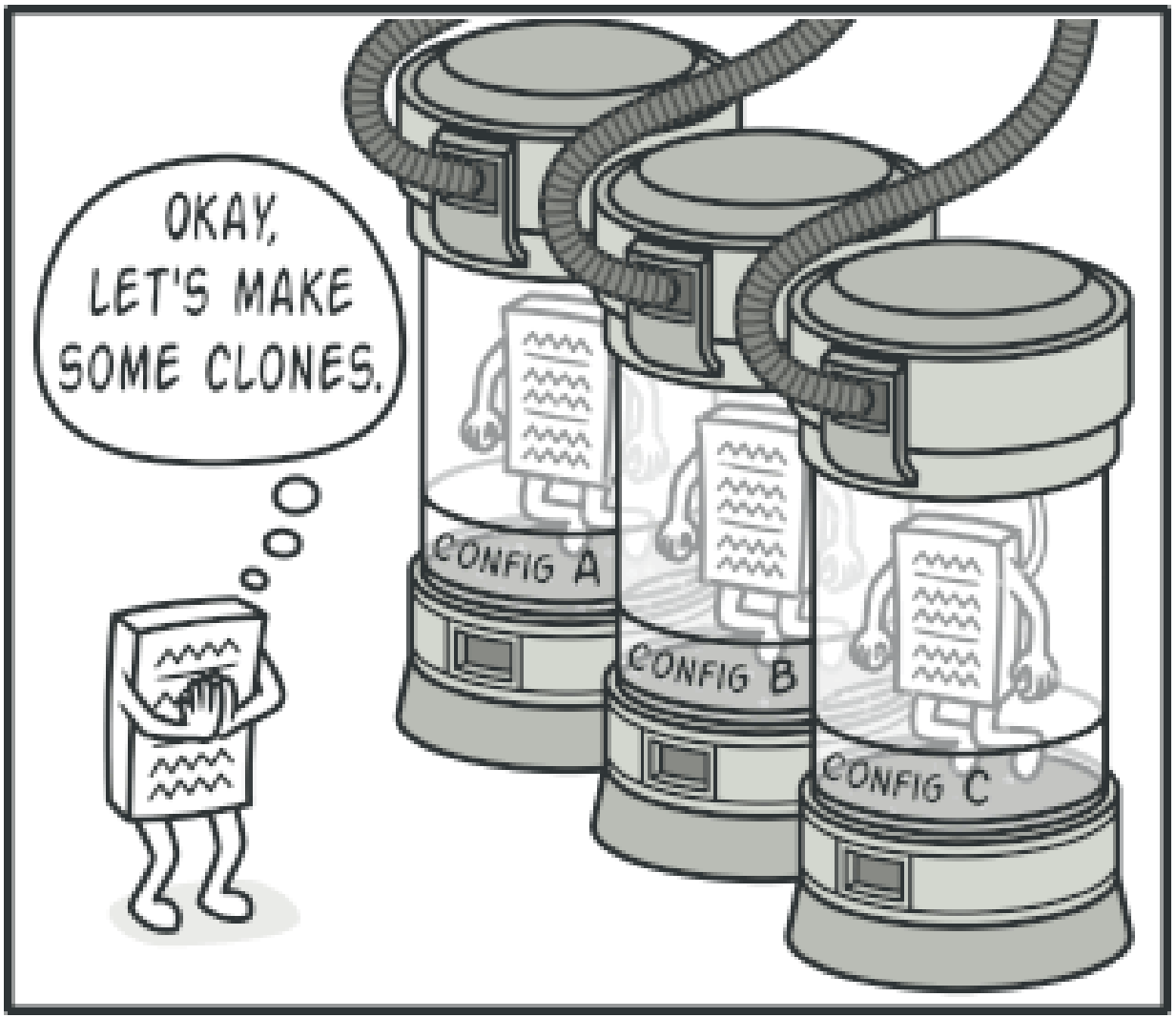
- On peut créer donc un nouveau objet d'une même classe et puis, on doit traverser tous les attributes de l'objet original et copier leur valeurs à le nouveau.
- Problème 1 : Certains attributes sont privés et pas visibles pour l'extérieur de l'objet $\Rightarrow$ on ne peut pas copier leur valeurs
- Problème 2 : Vous devez connaître la classe que vous vouloir créer un doublon, votre code devrait donc dépend de cette classe $\Rightarrow$ Il faut éviter cette dépendance
- Problème 3 : Très souvent vous connaissez juste l'interface que suit l'objet mais pas la classe concrete.
  - *Par exemple : a paramètre dans une méthode accepte n'importe quel objet qui implémente une certaine interface.*

Prototype Pattern - Solution

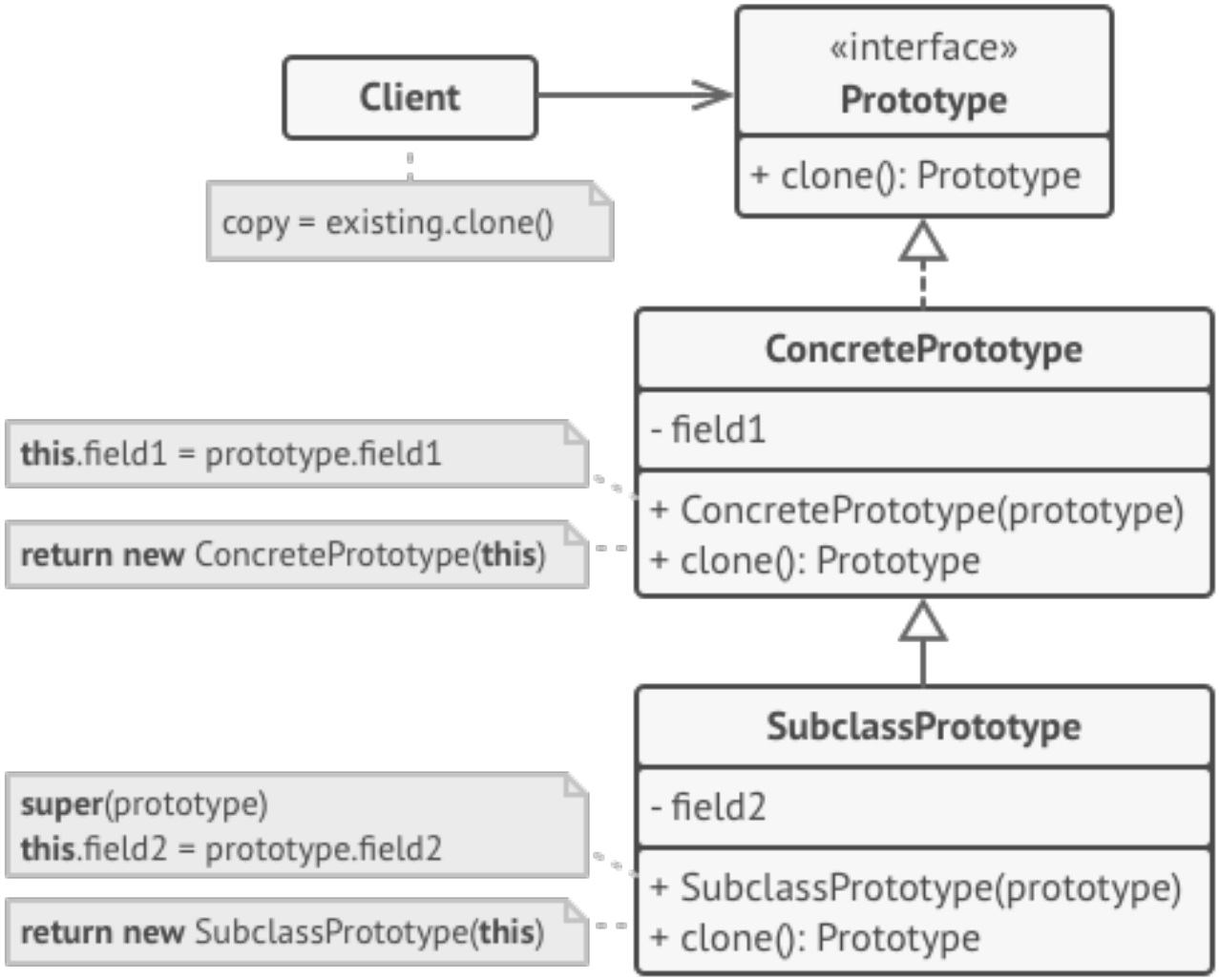
- Le modèle de conception Prototype délègue le processus de colonage aux objets qui sont clonés.
- Le modèle déclare une interface commune pour tous les objets qui prennent en charge le clonage
- Cette interface vous permet de cloner un objet sans coupler votre code à la classe de cet objet
- Une telle interface ne contient qu'une seule méthode de clonage
- Lorsque vos objets ont des dizaines de champs et des centaines de configurations possibles, leur clonage peut servir d'une alternative manière que la création des sous-classes

Prototype Pattern - Solution

Un objet qui prend en charge le clonage est appelé *prototype*.



Prototype Pattern - Structure



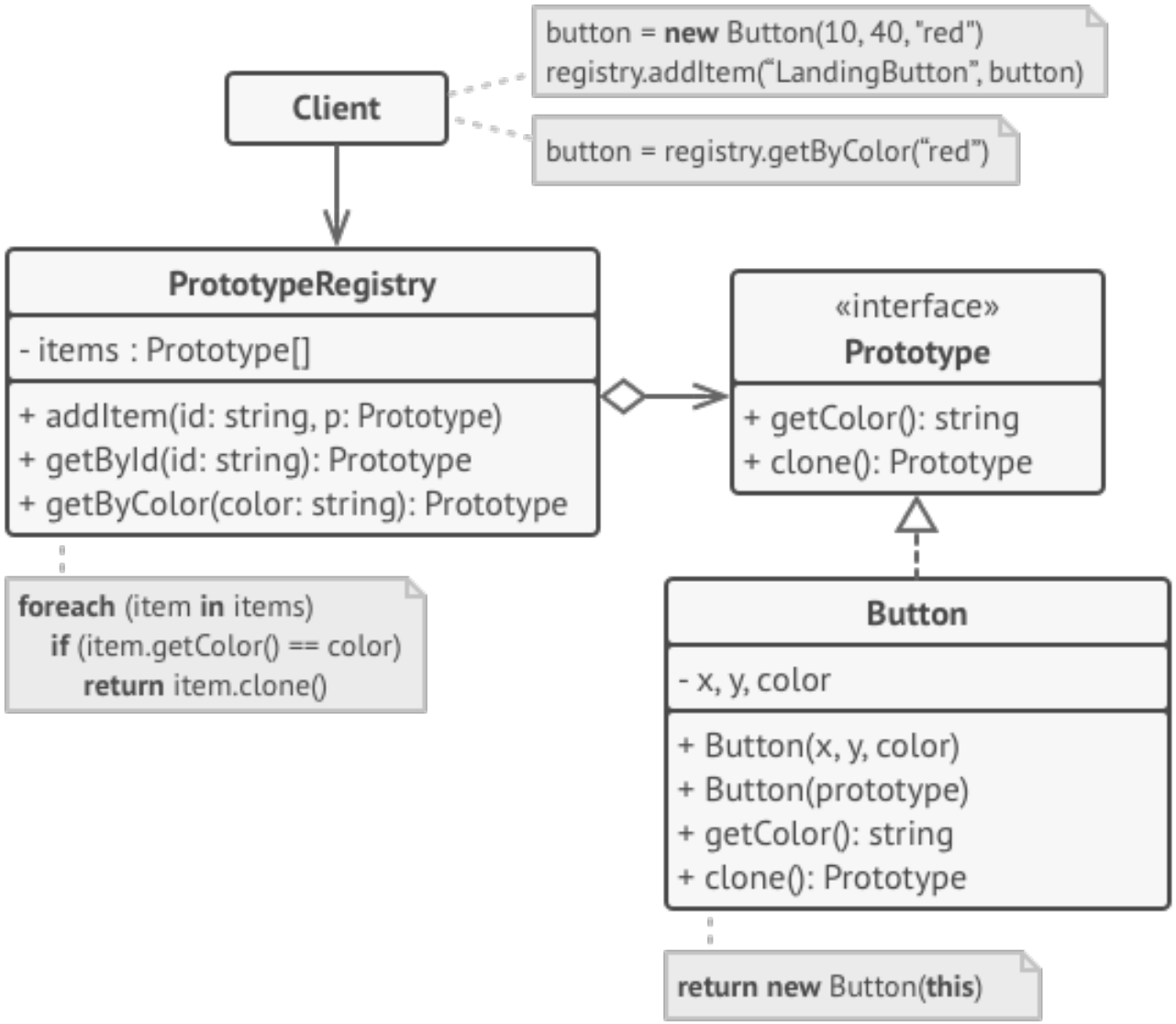
Prototype Pattern - Structure

1. L'interface `Prototype` déclare les méthodes de clonage. Dans la plupart des cas, on a seulement une seule méthode `clone`
2. La classe `ConcretePrototype` implémente la méthode de clonage. En plus de copier les données de l'objet d'origine dans la version de clonage, cette méthode peut également gérer certains cas concernant le clonage d'objets liés, démêler les dépendances récursives etc.
3. Le code client peut produire une copie de n'importe quel objet qui implémente l'interface `prototype`

Prototype Pattern - Structure

- On peut implémenter le registre de prototypes pour permettre d'accéder facilement aux prototypes fréquemment utilisés.
- Le registre de prototypes enregistre un ensemble d'objets prédéfinis prêts à être copiés
- Une implémentation simple du registre de prototype est une sorte de paires `nom` $\rightarrow$ `prototype`.

Prototype Pattern - Structure



Prototype Pattern - Applicabilité

1. On va utiliser le modèle de conception Prototype lorsque notre code ne devrait pas dépendre des classes concretes des objets que l'on veut faire les copies
   - Cela se produit souvent lorsque le code fonctionne avec des objets transmis à partir d'un code du système extérieur (c.a.d., une partie tiers) via une interface.
   - Les classes concretes de ces objets sont donc inconnues et on ne peut pas en dépendre même si on le souhait.
   - Le modèle Prototype fournit au code client une interface générale pour travailler avec tous les objets qui prennent en charge le clonage
   - Cette interface rend le code client indépendant des classes concrètes d'objets qu'il clone

Prototype Pattern - Applicabilité

2. On utilise le modèle de conception Prototype lorsque on souhaite réduire le nombre de sous-classes qui ne diffèrent que par la façon dont elles initialisent leurs objets respectifs
   - Le modèle Prototype permet d'utiliser un ensemble d'objets prédéfinis, configurés de diverses manières, comme prototypes
   - Au lieu d'instancier une sous-classe qui correspond à une configuration spécifique, le client peut simplement rechercher un prototype approprié et le cloner

Prototype Pattern - Avantages et Inconvénients

Avantages

- On peut cloner des objets sans les coupler à leurs classes concrètes
- On peut éviter le répétition du code d'initialisation en faveur du clonage de prototype prédéfinis
- On peut produire des objets complexes plus facilement
- On obtient une alternative manière que l'héritage lorsque l'on fait face à différentes configurations pour des objets complexes

Inconvénients

Le clonage d'objets complexes ayant des références circulaires peut être très délicat

Prototype Pattern - Etude de cas

Imaginez que vous voulez créer une application pour manipuler avec des formes (ex., circle, eclipse, rectangle, triangle etc.)

- Une `Shape` possède quelques champs comme : `x`, `y`, `color` et une méthode abstraite `clone`
- Des concrètes versions de `Shape` : `Circle`, `Square`, `Rectangle` qui fournissent l'implémentation différente pour la méthode `clone`
- Le code client peut interagir avec des prototypes via le prototype abstrait `Shape` et créer des copies