Introduction aux Design Patterns (Suite)

Rezak AZIZ
rezak.aziz@lecnam.net

CNAM Paris

Les patterns de Création

Les *Patterns de création*

Définition

Les **Patterns de création** décrivent différentes manières de **gérer, contrôler et structurer l'instanciation d'objets**, afin de réduire le couplage et améliorer la flexibilité de conception.

Principaux Patterns de création

- **Singleton** — garantit qu'une classe ne possède qu'une seule instance.
- **Prototype** — crée de nouveaux objets par clonage d'un objet existant.
- **Factory Method** — délègue la création à une méthode spécialisée, redéfinie par les sous-classes.
- **Abstract Factory** — crée des familles d'objets liés sans connaître les classes concrètes.
- **Builder** — sépare la construction d'un objet complexe de sa représentation finale.

Section 1

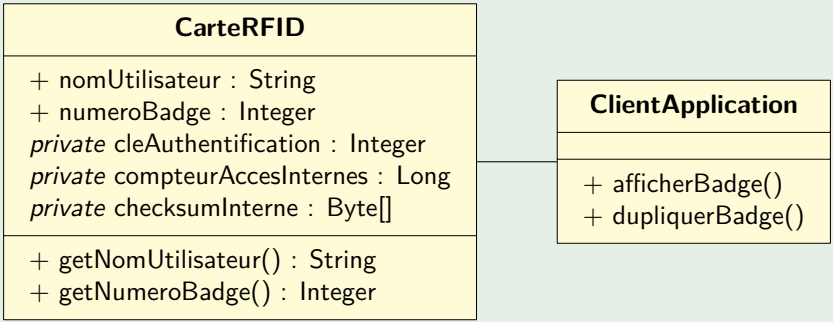
Prototype

Problème à résoudre

« Comment Copieriez vous un objet en programmation ? »

Problème à résoudre

« Comment Copieriez vous un objet en programmation ? »



« Comment implémenteriez vous le méthode dupliquerBadge() ? »

Copier un objet depuis l'extérieur pose problème ?

Approche naïve

1. Créer un nouvel objet de la même classe ;
2. Copier tous ses attributs un par un.

Question : où est le problème ?

- Certains attributs sont privés ;
- Il faut connaître la **classe concrète** ;
- Parfois on ne connaît que l'*interface* ;
- Hiérarchies complexes → difficile à reconstruire.

Intention générale

Intention

Prototype est un patron de conception qui crée de nouveaux objets à partir d'objets existants sans rendre le code dépendant de leur classe.

Idée clé

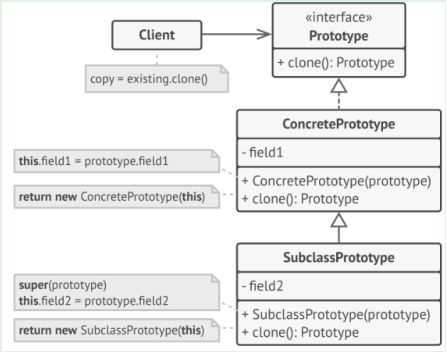
Définir une méthode **clone()** dans une interface commune et déléguer la duplication à l'objet.

Solution

Principe

Définir une méthode **clone()** dans une interface commune et déléguer la duplication à l'objet.

Structure



Rôles

- **Prototype** : interface déclarant `clone()` ;
- **ConcretePrototype** : implémente le clonage interne ;
- **Client** : clone des objets sans connaître leur classe.

Prototypes préconstruits

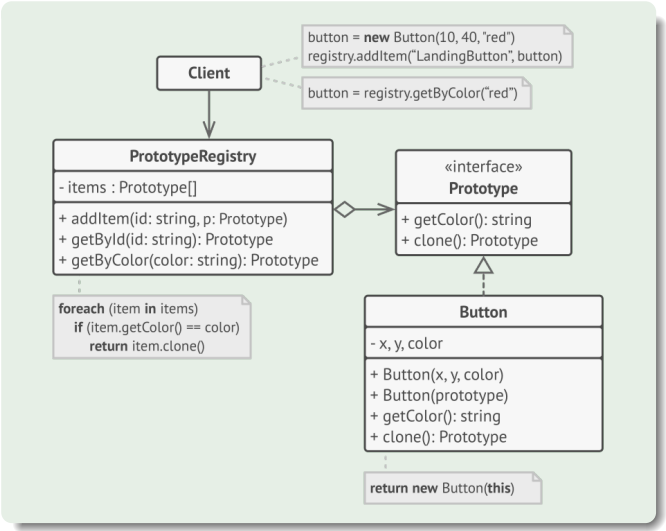
Principe

Stocker des prototypes préconstruits dans une table de hachage :

- clé → Prototype ;
- `clone()` pour obtenir une instance.

Utilité

- éviter la création répétée d'objets complexes ;
- fournir des modèles réutilisables ;
- substituer facilement une configuration.



Possibilités d'application

Quand utiliser le Prototype ?

- Le code ne doit pas dépendre des classes concrètes ;
- Les objets sont configurés via une interface externe ;
- Il existe de nombreuses configurations possibles ;
- Vous souhaitez éviter le sous-classage massif.

Cas concrets

- éléments graphiques ;
- objets lourds à initialiser ;
- documents modèles ;
- scènes 3D, textures, widgets préconfigurés.

Avantages et Inconvénients

Avantages

- Clonage sans connaître la classe concrète ;
- Suppression du code d'initialisation répétitif ;
- Création rapide d'objets complexes ;
- Alternative élégante au sous-classage.

Inconvénients

- Cloner des objets avec références circulaires est difficile ;
- Le clonage profond peut être coûteux ;
- Les constructeurs de copie doivent être bien maintenus.

Section 2

Factory Method

Problème à résoudre

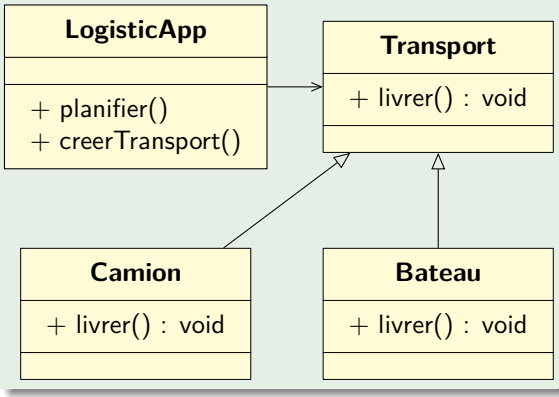
« Comment implémenter la méthode CréerTransport ? »

Approche naïve

- `new` dispersé dans tout le code ;
- Chaque ajout d'un nouveau transport impose de modifier LogisticApp ;
- Conditions multiples :
  - `if (type == camion)...`
  - `if (type == bateau)...`

Couplage fort + viol du principe ouvert/fermé (OCP).

Exemple



Intention générale

Intention

Fabrique est un patron de conception de création qui définit une interface pour créer des objets dans une classe mère, mais délègue le choix des types d'objets à créer aux sous-classes.

Idée clé

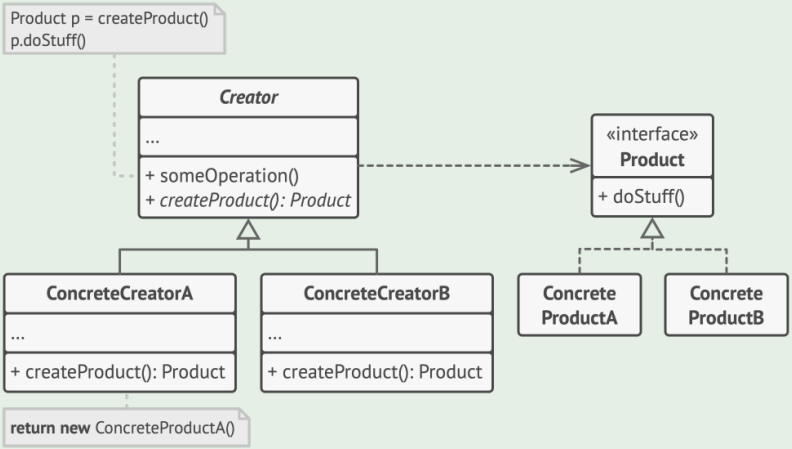
Remplacer : `new Camion()`

par : **`createTransport()`**

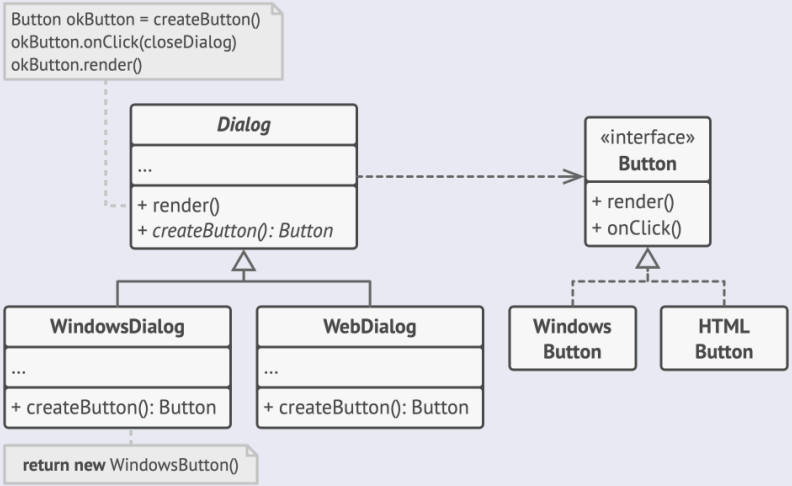
La sous-classe décide si elle renvoie un `Camion`, un `Bateau`, etc.

Solution

Structure



Exemple Création d'une application multi plateforme



Possibilités d'application

Quand utiliser la Factory Method ?

- Le code ne doit pas dépendre des classes concrètes ;
- Vous souhaitez ajouter facilement de nouveaux produits ;
- Le comportement varie selon la sous-classe ;
- Frameworks ou applications extensibles.

Cas concrets

- Widgets UI multiplateforme ;
- Connexions réseau ;
- Plugins et extensions ;
- Gestion de pools d'objets.

Avantages et Inconvénients

Avantages

- Découplage créateur / produits concrets ;
- Extensible sans modifier le code existant (OCP) ;
- Réduction des conditions dans le code ;
- Meilleure organisation du code.

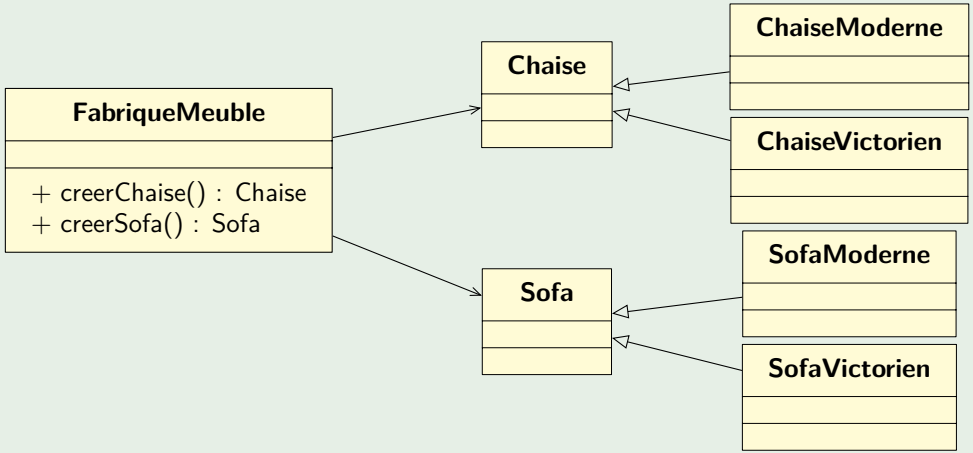
Inconvénients

- Plus de classes à créer ;
- Complexité initiale pour débutants ;
- Nécessite une interface produit commune.

Section 3

Fabrique Abstraite

Problème à résoudre



« Comment garantir que tous les meubles d'un style restent compatibles entre eux ? »

Combiner les variantes à la main pose problème

Approche naïve

- Créer les objets un par un avec `new` ;
- Mélanger les variantes selon des conditions :
  - `if (style == Moderne) new ChaiseModerne();`
  - `else if (style == Victorien) new ChaiseVictorienne();`
- Répéter ce schéma pour Sofa, Schéma, etc.

Où est le problème ?

- Risque de **mélanger les styles** par erreur ;
- Conditions dupliquées partout dans le code ;
- Difficile d'ajouter une nouvelle variante sans tout casser.

Intention générale

Intention

La **Fabrique Abstraite** est un patron de création qui permet de créer des **familles d'objets apparentés** sans préciser leur classe concrète.

Idée clé

- Le code client travaille uniquement avec des interfaces : `Chaise`, `Sofa`, `TableBasse`, etc.
- Une **fabrique concrète** garantit que tous ses produits appartiennent à la **même variante** (Moderne, Victorien, ArtDéco, ...).

Solution

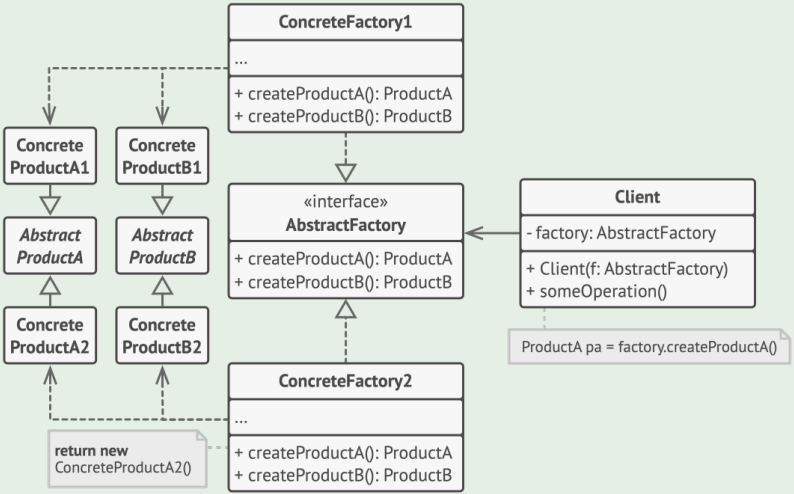
Principe

1. Définir des **produits abstraits** (`Chaise`, `Sofa`, `TableBasse`) ;
2. Définir une **Fabrique Abstraite** qui sait créer chaque type de produit ;
3. Implémenter une **Fabrique Concrète par variante** (`FabriqueModerne`, `FabriqueVictorienne`, …).

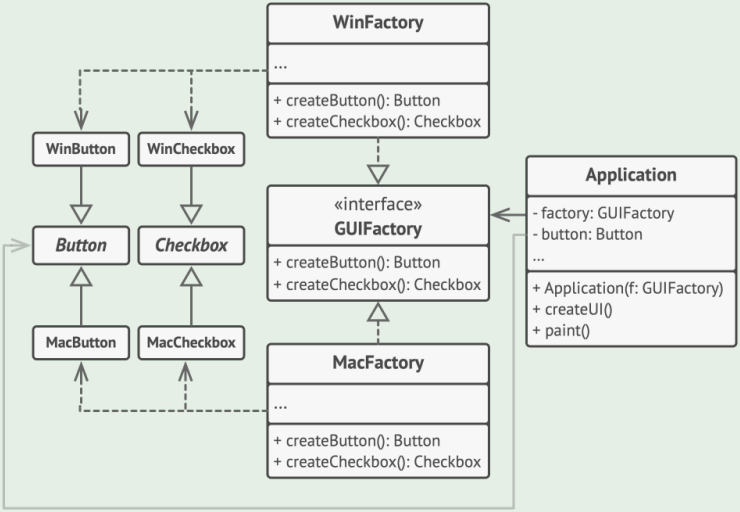
Effet

- Tous les objets créés appartiennent à la **même famille** ;
- Ils sont **compatibles** entre eux ;
- Le code client ne voit jamais les **classes concrètes**.

Structure du *Abstract Factory*



Exemple Création d'une application multi plateforme



Possibilités d'application

Quand utiliser la Fabrique Abstraite ?

- Vous avez des **familles d'objets liés** ;
- Vous voulez garantir leur **compatibilité** ;
- Vous voulez pouvoir changer de variante d'un coup (thème, skin, style graphique...) ;
- Le code ne doit dépendre d'aucune classe concrète.

Cas typiques

- Interfaces graphiques multiplateformes ;
- Thèmes / skins (clair, sombre, haute-contraste) ;
- Meubles modernes / victoriens / art déco ;
- Jeux vidéo : races, factions, environnements cohérents.

Avantages et inconvénients

Avantages

- Compatibilité garantie entre produits d'une même famille ;
- Découplage total du client et des produits concrets ;
- Changement de variante centralisé (une seule fabrique) ;
- Respect du principe ouvert/fermé pour les variantes.

Inconvénients

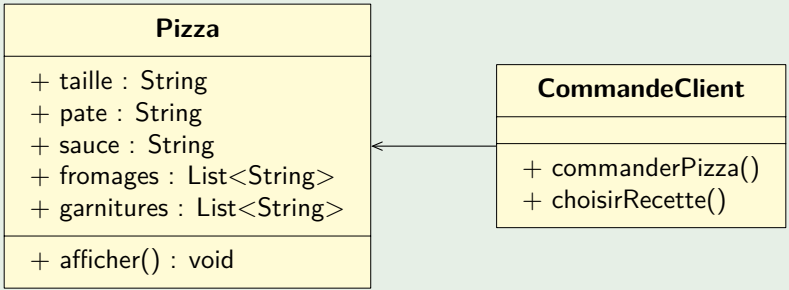
- Beaucoup de classes (produits + fabriques) ;
- Ajouter une nouvelle **famille de produits** implique de modifier toutes les fabriques ;
- Patron parfois jugé « lourd » pour de petits projets.

Section 4

Monteur (Builder)

Problème à résoudre

Exemple : Construire une Pizza



« Comment éviter un constructeur géant du type :

`Pizza(taille, pate, sauce, fromage1, fromage2, olive, champignons, ...)` ?

Constructeur classique : trop complexe

Approche naïve

```
new Pizza("Large", "Fine",
          "Tomate",
          ["Mozza", "Parmesan"],
          ["Olives", "Champignons",
           "Jambon"])
```

→ illisible, erreur facile, difficile à maintenir.

Approche naïve 2

```
Pizza p = new Pizza();
p.setSize("Large");
p.setPate("Fine");
p.setSauce("Tomate");
...
```

→ lisible, facile.

Problème

- Trop de paramètres ;
- Impossible de garantir la cohérence ;
- Combinaisons infinies de pizzas (Reine, 4 fromages, Veggie...).

Intention générale

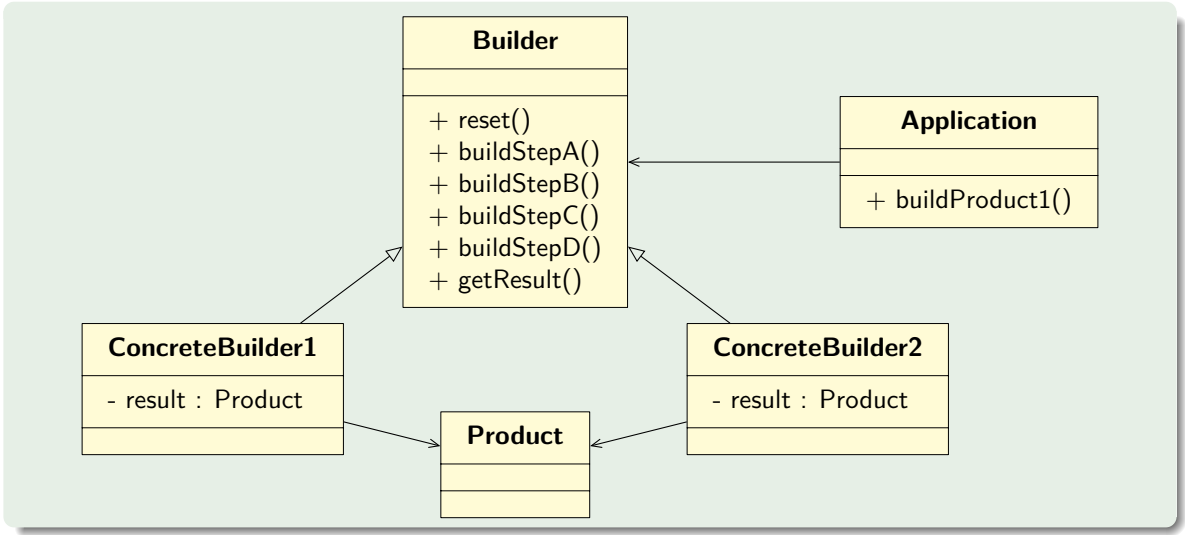
Intention

Monteur est un patron de conception de création qui permet de construire des objets complexes étape par étape. Il permet de produire différentes variations ou représentations d'un objet en utilisant le même code de construction.

Idée clé

Rendre le constructeur accessible uniquement par un builder et passer par le builder pour construire l'objet concret.

Structure du *Builder*



Possibilités d'application

Quand utiliser le Builder ?

- Objets complexes (lots d'options) ;
- Construction progressive étape par étape ;
- Plusieurs représentations possibles ;
- Éviter les constructeurs géants.

Cas concrets

- Construction d'objets immuables ;
- Génération de documents (manuel pour produit) ;
- Requêtes SQL complexes ;
- Scènes 3D, interfaces graphiques ;
- Pipelines DevOps.

Avantages et Inconvénients

Avantages

- Construction flexible et contrôlée ;
- Même séquence $\rightarrow$ produits différents ;
- Code lisible, organisé et cohérent ;
- Découplage construction / représentation.

Inconvénients

- Plus de classes ;
- Mise en place lourde pour petits objets ;
- Nécessite de définir clairement les étapes.