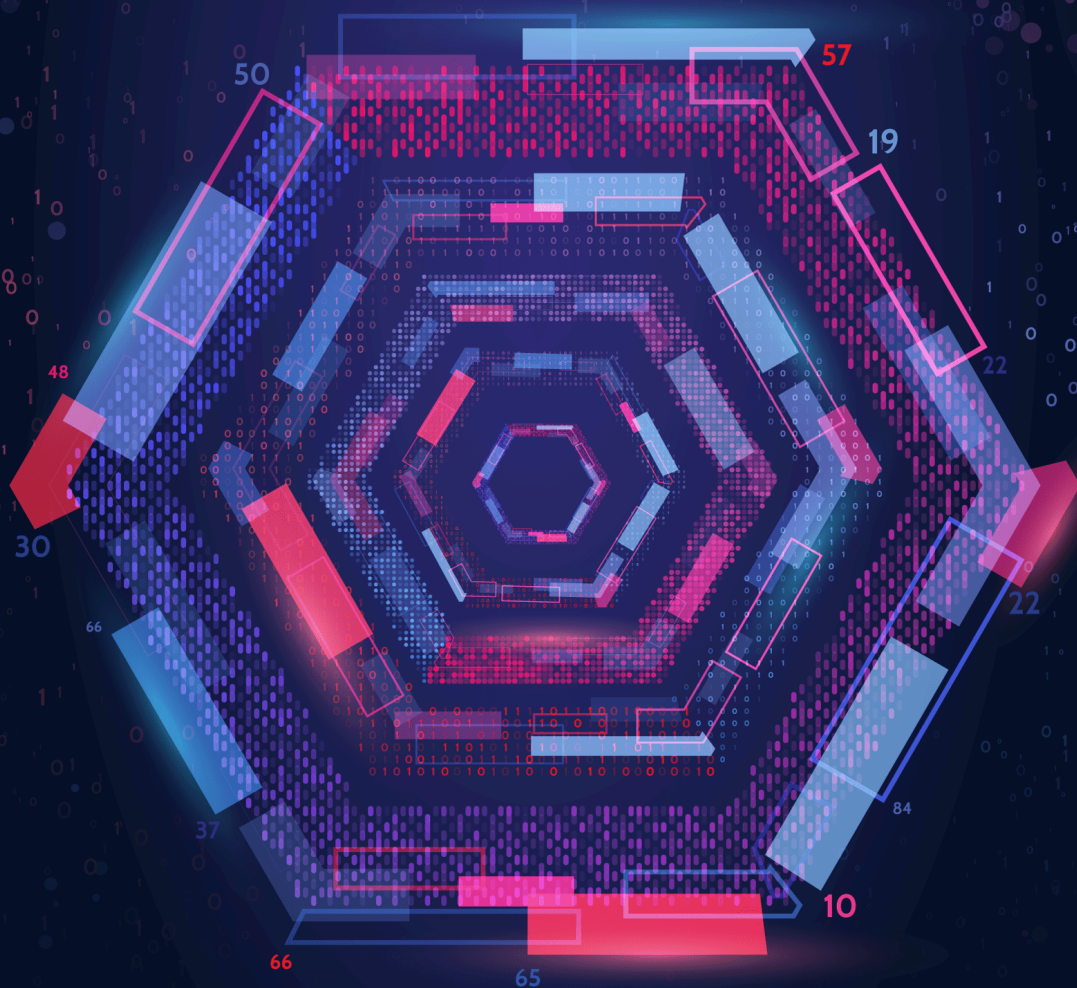


Plongée au cœur des

PATRONS DE CONCEPTION



Alexander Shvets

Plongée au cœur des

PATRONS DE CONCEPTION

v2021-1.2

VERSION DÉMO

Achetez le livre complet :

<https://refactoring.guru/fr/design-patterns/book>

Quelques mots à propos du copyright

Salut ! Je m'appelle Alexander Shvets.
Je suis l'auteur du livre **Plongée au cœur des patrons de conception** et du cours en ligne **Dive into refactoring**.



Ce livre est réservé pour votre utilisation personnelle. Ne le prêtez qu'aux membres de votre famille svp. Si vous voulez partager le livre avec un ami ou un collègue, achetez-leur une copie. Vous pouvez également acheter une licence de site pour toute votre équipe ou pour votre entreprise.

Tous les bénéfices de la vente de mes livres et cours sont utilisés pour le développement de **Refactoring.Guru**. Chaque copie vendue aide énormément le projet et nous rapproche de la sortie d'un nouveau livre.

© Alexander Shvets, Refactoring.Guru, 2021

✉ support@refactoring.guru

🖼 Illustrations : Dmitry Zhart

🇫🇷 Traduction : David Simon

✎ Révision : Luc Perret

*Je dédie ce livre à ma femme, Maria. Sans elle,
son écriture m'aurait sans doute pris 30 ans
de plus.*

Table des matières

| | |
|--|-----------|
| Table des matières | 4 |
| Comment lire ce livre | 6 |
| INTRODUCTION À LA POO | 7 |
| Les bases de la POO | 8 |
| Les piliers de la POO | 14 |
| Relations entre les objets | 22 |
| INTRODUCTION AUX PATRONS DE CONCEPTION | 28 |
| Qu'est-ce qu'un patron de conception? | 29 |
| Pourquoi devrais-je apprendre les patrons? | 34 |
| PRINCIPES DE CONCEPTION LOGICIELLE | 35 |
| Caractéristiques d'une bonne conception | 36 |
| Principes de conception | 41 |
| § Encapsuler ce qui varie | 42 |
| § Programmez avec les interfaces, et non pas avec les implémentations | 47 |
| § Préférer la composition à l'héritage | 53 |
| Principes SOLID | 57 |
| § Principe de responsabilité unique | 58 |
| § Principe ouvert/fermé | 61 |
| § Principe de substitution de Liskov | 65 |
| § Principe de ségrégation des interfaces | 72 |
| § Principe d'inversion des dépendances | 75 |

| | |
|---|------------|
| CATALOGUE DES PATRONS DE CONCEPTION..... | 79 |
| Patrons de création | 80 |
| § Fabrique / <i>Factory Method</i> | 82 |
| § Fabrique abstraite / <i>Abstract Factory</i> | 99 |
| § Monteur / <i>Builder</i> | 116 |
| § Prototype / <i>Prototype</i> | 137 |
| § Singleton / <i>Singleton</i> | 153 |
| Patrons structurels | 162 |
| § Adaptateur / <i>Adapter</i> | 165 |
| § Pont / <i>Bridge</i> | 179 |
| § Composite / <i>Composite</i> | 196 |
| § Décorateur / <i>Decorator</i> | 211 |
| § Façade / <i>Facade</i> | 231 |
| § Poids mouche / <i>Flyweight</i> | 242 |
| § Procuration / <i>Proxy</i> | 258 |
| Patrons comportementaux | 272 |
| § Chaîne de responsabilité / <i>Chain of Responsibility</i> | 276 |
| § Commande / <i>Command</i> | 294 |
| § Itérateur / <i>Iterator</i> | 316 |
| § Médiateur / <i>Mediator</i> | 332 |
| § Memento / <i>Memento</i> | 348 |
| § Observateur / <i>Observer</i> | 365 |
| § État / <i>State</i> | 382 |
| § Stratégie / <i>Strategy</i> | 399 |
| § Patron de méthode / <i>Template Method</i> | 414 |
| § Visiteur / <i>Visitor</i> | 428 |
| Conclusion | 444 |

Comment lire ce livre

Ce livre contient 22 patrons de conception classiques décrits par le « Gang of Four » (le gang des quatre, ou GoF) en 1994.

Chaque chapitre explore un patron différent. Vous pouvez donc tout lire depuis le début ou simplement choisir les patrons qui vous intéressent.

De nombreux patrons sont connectés, vous pouvez donc facilement sauter de l'un à l'autre en utilisant les ancres. À la fin de chaque chapitre, une liste de liens présentant les relations entre ce patron et d'autres vous est proposée. Si vous apercevez le nom d'un patron que vous ne connaissez pas encore, continuez à lire : vous l'aborderez tôt ou tard dans l'un des chapitres suivants.

Les patrons de conception sont universels. Les échantillons de code sont écrits en pseudo-code, ce qui permet de ne pas se limiter à un seul langage de programmation.

Avant d'étudier les patrons de conception, vous pouvez vous rafraîchir la mémoire en lisant les **termes clés de la programmation orientée objet**. Ce chapitre couvre également les bases des diagrammes UML, ce qui vous sera fortement utile, car le livre vous en proposera un certain nombre. Bien sûr, si vous connaissez déjà tout cela, vous pouvez vous diriger tout de suite vers les **patrons de conception**.

INTRODUCTION

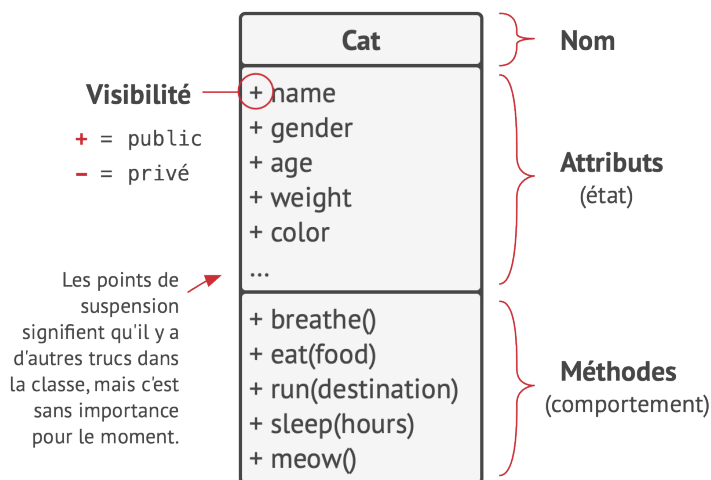
À LA POO

Les bases de la POO

La **Programmation Orientée Objet** est un paradigme qui se base sur la représentation des données et de leur comportement dans des briques logicielles appelées **objets**, qui sont fabriqués à partir d'un ensemble de « plans » définis par un développeur, que l'on appelle des **classes**.

Objets, classes

Aimez-vous les chats ? J'espère que oui, parce que je vais tenter de vous expliquer le concept de la POO avec différents exemples de chats.



Ceci est un diagramme de classes UML. Vous allez voir de nombreux diagrammes de ce type dans le livre. La norme est de laisser les noms de la classe et de ses membres en anglais, comme vous le feriez dans du code. Cependant, les commentaires et remarques seront parfois écrits en français.

Supposons que vous possédez un chat nommé Félix. Félix est un objet, une instance de la classe `Chat`. Chaque chat possède un nombre standard d'attributs : nom, sexe, âge, poids, couleur, nourriture préférée, etc. Ce sont les *attributs* (ou champs) de la classe.

Je ferai parfois référence au nom des classes en français, même si elles apparaissent en anglais dans le code et les diagrammes (comme c'est le cas pour la classe `Chat`). Je veux que vous puissiez lire ce livre comme si vous aviez une conversation entre amis et vous éviter de buter sur des mots extra-terrestres lorsque je mentionne le nom d'une classe.

Tous les chats se comportent également de la même manière : ils respirent, mangent, courent, dorment et miaulent. Ce sont les *méthodes* de la classe. Les attributs et méthodes d'une classe sont des *membres* de leur classe.

Les données stockées dans les attributs d'un objet sont appelées *l'état* et les méthodes que l'objet définit représentent son *comportement*.

**Félix: Cat**

```

name   = "Félix"
sex     = "Mâle"
age     = 3
weight  = 7
color   = brun
texture = tigré
  
```

**Minette: Cat**

```

name   = "Minette"
sex     = "Femelle"
age     = 2
weight  = 5
color   = gris
texture = uni
  
```

Les objets sont des instances de classes.

Minette, l'amie de votre chat, est également une instance de la classe `Chat`. Elle possède les mêmes attributs que Félix. Mais le contenu de ses attributs diffère de ceux de Félix : son sexe est femelle, sa couleur est différente et elle est plus légère.

Une *classe* est donc un peu comme un plan qui définit la structure pour les *objets*. Les objets sont les instances concrètes de cette classe.

Hiérarchies de classes

Tout va pour le mieux lorsque l'on ne s'occupe que d'une seule classe, mais bien entendu, un vrai programme en contient bien

plus. Certaines de ces classes peuvent être organisées dans une **hiérarchie de classes**. Voyons ce que cela signifie.

Disons que votre voisin possède un chien qui s'appelle Médor. Il se trouve que les chiens et les chats ont beaucoup de points communs : le nom, le sexe, l'âge et la couleur sont des attributs qui peuvent s'appliquer aux chiens et aux chats. Les chiens respirent, dorment et mangent tout comme les chats. Il semblerait donc que l'on peut définir une classe de base `Animal` qui recense les attributs et comportements communs.

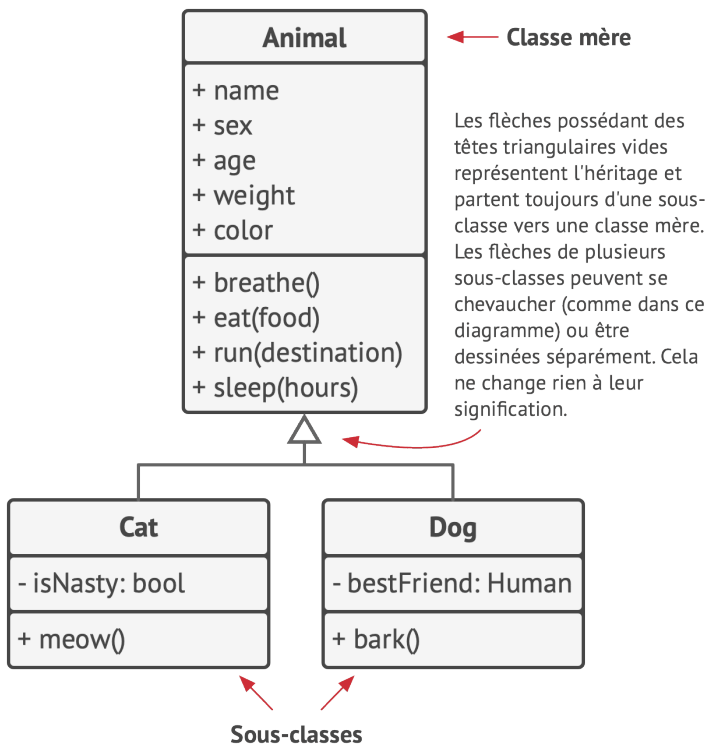
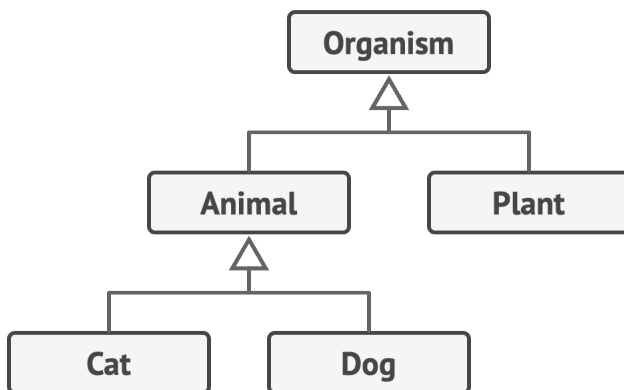


Diagramme UML d'une hiérarchie simple de classes. Toutes les classes de ce diagramme font partie de la hiérarchie de classes `Animal`.

Une classe parent, comme celle que nous venons juste de définir, est appelée une **classe mère** (ou super-classe, ou encore classe de base). Ses enfants sont des **sous-classes** (ou classes dérivées). Les sous-classes héritent de l'état et du comportement de leur parent et ne définissent que les attributs ou les comportements qui diffèrent. Par conséquent, la classe `Chat` posséderait la méthode `miaule`, et la classe `Chien` la méthode `aboie`.

Dans le cas où nous avons d'autres besoins du même ordre, nous pouvons aller encore plus loin et extraire une classe plus générale pour tous les `Organismes` vivants, qui va devenir la classe mère des `Animaux` et des `Plantes`. Cette pyramide s'appelle une **hiérarchie**. Dans une telle hiérarchie, la classe `Chat` hérite de tout ce qu'il y a dans les classes `Animal` et `Organisme`.



Les classes d'un diagramme UML peuvent être simplifiées s'il est plus important d'afficher leurs relations que leur contenu.

Les sous-classes peuvent redéfinir le comportement des méthodes dont elles héritent via leurs classes mères. Une sous-classe peut soit remplacer complètement le comportement par défaut, soit juste l'améliorer en y ajoutant des choses supplémentaires.

21 pages

du livre complet sont absentes de la version démo

PRINCIPES DE CONCEPTION LOGICIELLE

Caractéristiques d'une bonne conception

Avant d'aborder les patrons de conception, discutons du processus de conception de l'architecture d'un logiciel : ce qu'il faut faire et ce qu'il faut éviter.

Réutilisation du code

Les indicateurs les plus importants dans le développement d'un logiciel sont le coût et le temps. Plus le temps de développement est court, et plus vous entrerez tôt sur le marché par rapport à vos concurrents. Des coûts de développement faibles vous permettent d'allouer une plus grande part de votre budget au marketing et de prospecter sur une plus grande échelle.

La réutilisation du code est le moyen le plus classique de réduire les coûts de développement. C'est évident après tout : plutôt que de recommencer à chaque fois depuis le début, pourquoi ne pas utiliser du code existant pour les nouveaux projets ?

Cette idée est géniale sur le papier, mais en réalité, adapter le code à un nouveau contexte demande un certain travail. Les composants fortement couplés, les dépendances aux classes concrètes (au lieu d'interfaces), le code écrit en dur : tout ceci réduit la flexibilité du code et le rend plus difficile à réutiliser.

Ce manque de flexibilité des composants de votre logiciel peut être résolu en utilisant des patrons de conception. De plus, ils vont faciliter la réutilisation de votre code. Ceci peut en revanche rendre les composants plus compliqués.

Voici quelques conseils avisés d'Erich Gamma¹, un des pères fondateurs des patrons de conception, au sujet du rôle des patrons dans la réutilisation du code :

“

Je vois trois niveaux de réutilisation.

Au plus bas niveau, vous réutilisez les classes : bibliothèques de classes, conteneurs, et peut-être quelques « équipes » de classes comme des conteneurs/itérateurs.

Les frameworks se trouvent au plus haut niveau. Ils essayent de distiller vos décisions de conception. Ils identifient les abstractions clés pour résoudre un problème, les représentent dans des classes et définissent les relations entre elles. Par exemple, JUnit est un petit framework. C'est un peu le « Hello, world » des frameworks. `Test`, `TestCase`, `TestSuite` et les relations y sont déjà définies.

Un framework travaille de manière plus globale qu'une classe. Pour vous raccorder à un framework, vous sous-classez un endroit de votre code. Ils utilisent le principe hollywoodien : « ne nous rappelez pas, nous vous rappellerons ». Le framework vous laisse définir votre comportement personnalisé et il vous

1. Erich Gamma on Flexibility and Reuse: <https://refactoring.guru/gamma-interview>

appellera lorsque ce sera votre tour d'agir. C'est la même chose avec JUnit. Il vous appelle lorsqu'il veut lancer votre test, mais tout le reste se déroule à l'intérieur du framework.

Il y a également un niveau intermédiaire. C'est ici que je range les patrons. Les patrons de conception sont plus petits et plus abstraits que les frameworks. Ils permettent réellement de décrire les relations et interactions entre plusieurs classes. Plus vous montez de niveau (les classes, puis les patrons et enfin les frameworks), plus le niveau de réutilisation augmente.

Le gros avantage des patrons par rapport aux frameworks, c'est que la réutilisation du code est beaucoup moins risquée. Construire un framework comporte de gros risques et demande un gros investissement. Les patrons vous permettent de réutiliser vos idées de conception et vos concepts indépendamment du code concret.

”



Extensibilité

Le **changement** est la seule constante dans la vie d'un développeur.

- Vous avez sorti un jeu vidéo sur Windows, mais maintenant les gens veulent une version macOS.
- Vous avez créé un framework de GUI avec des boutons carrés, mais plusieurs mois plus tard, les boutons ronds sont à la mode.

- Vous avez conçu une superbe architecture pour des boutiques en ligne, mais le mois suivant, les clients vous demandent une fonctionnalité qui leur permettrait d'accepter les commandes par téléphone.

Chaque développeur a vécu des dizaines d'histoires similaires. Ceci se produit pour plusieurs raisons.

Tout d'abord, nous comprenons mieux le problème une fois que nous commençons à le résoudre. Bien souvent, une fois que vous avez publié la première version de votre application, vous êtes prêts à la réécrire une nouvelle fois depuis le début, car vous comprenez dorénavant beaucoup mieux plusieurs aspects de la problématique. Vous avez également mûri professionnellement, et maintenant vous trouvez que votre code a mauvaise allure.

Quelque chose que vous ne contrôlez pas a changé. C'est pourquoi de nombreuses équipes de développement s'éloignent de leur idée originale pour faire quelque chose de nouveau. Tous ceux qui se sont basés sur Flash dans une application en ligne ont retravaillé ou migré leur code une fois que les navigateurs ont arrêté sa prise en charge.

La troisième raison est que les poteaux du but se déplacent. Votre client était satisfait de la version actuelle de l'application, mais il voit à présent les « petites » modifications qu'il souhaite, des petits ajouts dont il n'avait jamais parlé lors des sessions originales de planification. Ce ne sont pas des modifi-

cations futiles : votre excellente première version lui a montré qu'il y avait encore plus de possibilités.

Il y a un côté positif : si quelqu'un vous demande de modifier quelque chose dans votre application, cela veut dire qu'elle compte beaucoup pour lui.

C'est la raison pour laquelle tous les développeurs expérimentés essayent d'anticiper de futures modifications lorsqu'ils conçoivent l'architecture d'une application.

Principes de conception

Un logiciel bien conçu, c'est quoi? Comment pouvez-vous l'évaluer? Quelles pratiques devez-vous suivre pour y parvenir? Comment pouvez-vous rendre votre architecture flexible, stable et facile à comprendre?

Ce sont les bonnes questions, mais malheureusement les réponses varient énormément en fonction de l'application que vous voulez développer. Néanmoins, il y a plusieurs principes universels de conception qui pourraient vous aider à répondre à ces questions pour votre projet. La majorité des patrons de conception que vous retrouverez dans ce livre sont basés sur ces principes.

Encapsuler ce qui varie

Identifiez les parties qui varient dans votre application, puis séparez-les de ce qui est statique.

L'objectif de ce principe est de minimiser les effets causés par toute modification.

Imaginez que votre programme est un navire, et les modifications sont des mines qui traînent sous l'eau. Si le navire est touché par une mine, il coule.

Avec ceci en tête, vous pouvez diviser la coque du navire en compartiments indépendants et scellés, qui permettent de limiter les dégâts à un seul compartiment. À présent, si le navire heurte une mine, il ne coulera pas.

De la même façon, vous pouvez isoler les parties du programme qui varient dans des modules indépendants, protégeant le reste du code des effets indésirables. Grâce à cela, vous perdez moins de temps à remettre en état votre programme et à tester et implémenter les modifications. Moins vous passez de temps à effectuer ces modifications, et plus vous en avez pour implémenter de nouvelles fonctionnalités.

L'encapsulation au niveau des méthodes

Imaginons que vous concevez une boutique en ligne. Quelque part dans votre code, il y a une méthode `getTotalCommande` qui calcule le résultat total de la commande, taxes comprises.

Nous pouvons anticiper le fait que le code des taxes aura probablement besoin d'évoluer dans le futur. Le taux de la taxe varie en fonction du pays, de l'état ou même de la ville de résidence du client, et la formule peut changer tout au long de la vie du programme à cause de nouvelles lois. Vous allez donc modifier la méthode `getTotalCommande` assez souvent. Mais le nom de la méthode suggère implicitement qu'elle ne se préoccupe pas de la manière dont ce calcul est effectué.

```
1  method getOrderTotal(order) is
2      total = 0
3      foreach item in order.lineItems
4          total += item.price * item.quantity
5
6      if (order.country == "US")
7          total += total * 0.07 // Taxe américaine
8      else if (order.country == "EU"):
9          total += total * 0.20 // TVA européenne
10
11     return total
```

AVANT : le code de calcul des taxes est mélangé avec le reste du code de la méthode.

Vous pouvez extraire la logique métier du calcul de la taxe, la mettre dans une méthode séparée et la cacher de la méthode originale.

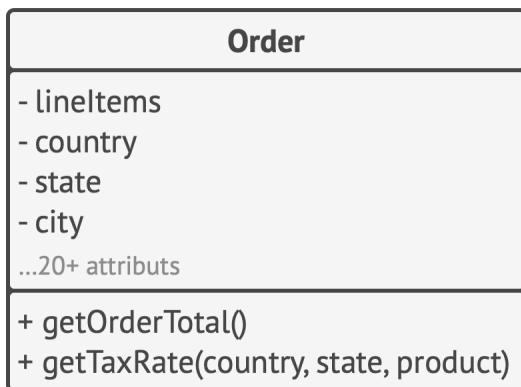
```
1  method getOrderTotal(order) is
2    total = 0
3    foreach item in order.lineItems
4      total += item.price * item.quantity
5
6    total += total * getTaxRate(order.country)
7
8    return total
9
10 method getTaxRate(country) is
11   if (country == "US")
12     return 0.07 // Taxe américaine
13   else if (country == "EU")
14     return 0.20 // TVA européenne
15   else
16     return 0
```

APRÈS : vous pouvez récupérer le taux de la taxe en appelant la méthode concernée.

Les modifications qui concernent la taxe sont isolées dans une seule méthode. De plus, si la logique de calcul devient trop compliquée, vous pouvez maintenant la transférer facilement dans une classe séparée.

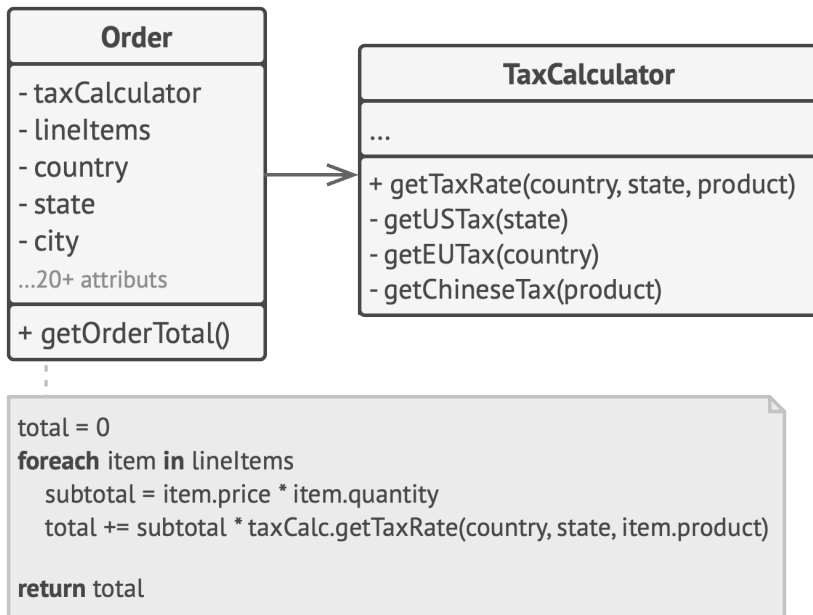
L'encapsulation au niveau des classes

Avec le temps, vous allez ajouter de plus en plus de responsabilités à une méthode qui n'en avait qu'une seule à l'origine. Ces comportements supplémentaires viennent souvent avec leurs propres attributs et méthodes, qui au bout d'un moment, vont masquer la responsabilité principale de la classe englobante. Tout extraire dans une nouvelle classe va vous permettre de tout rendre clair et simple.



AVANT : calcul de la taxe dans la classe **Commande** .

Les objets de la classe `Commande` délèguent toutes les tâches concernant les taxes à un objet spécialisé qui ne s'occupe que de cela.



APRÈS : le calcul de la taxe est caché dans la classe `Commande` .

32 pages

du livre complet sont absentes de la version démo

CATALOGUE DES PATRONS DE CONCEPTION

Patrons de création

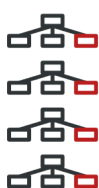
Les patrons de création fournissent des mécanismes de création d'objets qui augmentent la flexibilité et la réutilisation du code.



Fabrique

Factory Method

Définit une interface pour la création d'objets dans une classe mère, mais délègue aux sous-classes le choix des types d'objets à créer.



Fabrique abstraite

Abstract Factory

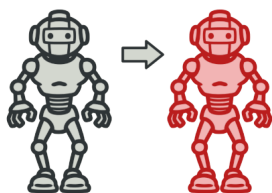
Permet de créer des familles d'objets apparentés sans préciser leur classe concrète.



Monteur

Builder

Permet de construire des objets complexes étape par étape. Ce patron permet de construire différentes variations ou représentations d'un objet en utilisant le même code de construction.



Prototype

Prototype

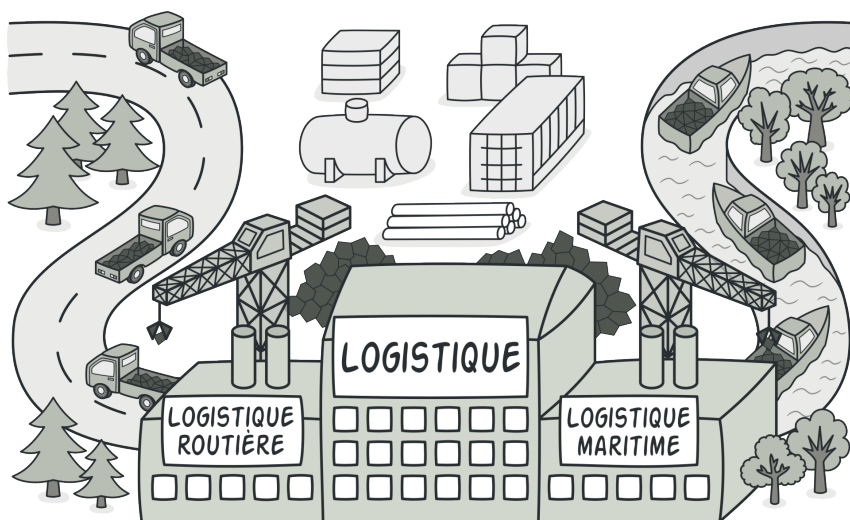
Permet de créer de nouveaux objets à partir d'objets existants sans rendre le code dépendant de leur classe.



Singleton

Singleton

Permet de garantir que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.



FABRIQUE

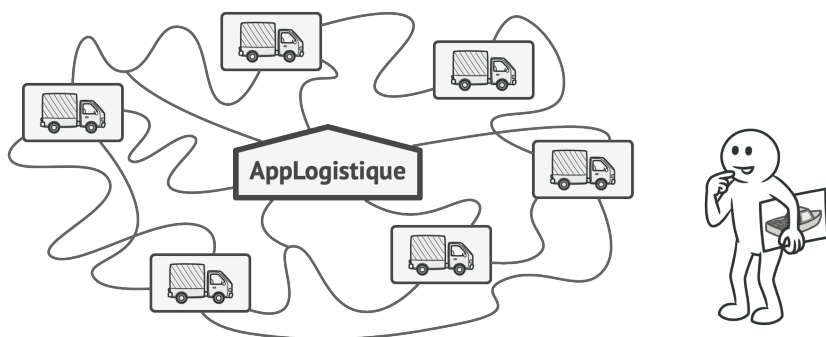
Alias: Constructeur virtuel, Factory Method

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.

🙄 Problème

Imaginez que vous êtes en train de créer une application de gestion logistique. La première version de votre application ne propose que le transport par camion, la majeure partie de votre code est donc située dans la classe `Camion`.

Au bout d'un certain temps, votre application devient populaire et de nombreuses entreprises de transport maritime vous demandent tous les jours d'ajouter la gestion de la logistique maritime dans l'application.



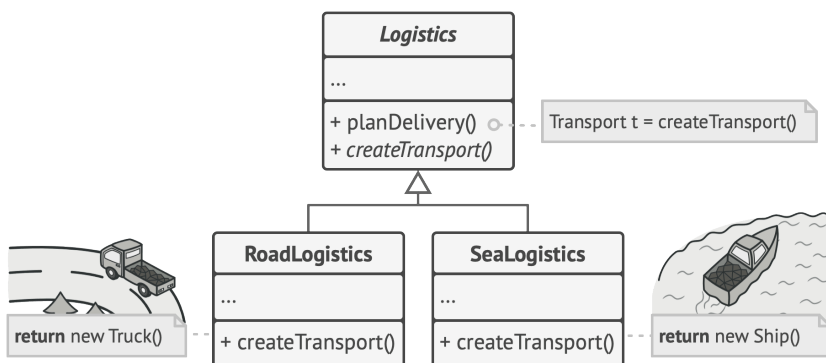
L'ajout d'une nouvelle classe au programme ne s'avère pas si simple que cela si le reste du code est déjà couplé aux classes existantes.

C'est super, n'est-ce pas ? Mais qu'en est-il du code ? La majeure partie est actuellement couplée à la classe `Camion`. Pour pouvoir ajouter des `Bateaux` dans l'application, il faudrait revoir la base du code. De plus, si vous décidez plus tard d'ajouter un autre type de transport dans l'application, il faudra effectuer à nouveau ces changements.

Par conséquent, vous allez vous retrouver avec du code pas très propre, rempli de conditions qui modifient le comportement du programme en fonction de la classe des objets de transport.

😊 Solution

Le patron de conception fabrique vous propose de remplacer les appels directs au constructeur de l'objet (à l'aide de l'opérateur `new`) en appelant une méthode *fabrique* spéciale. Pas d'inquiétude, les objets sont toujours créés avec l'opérateur `new`, mais l'appel se fait à l'intérieur de la méthode fabrique. Les objets qu'elle retourne sont souvent appelés *produits*.

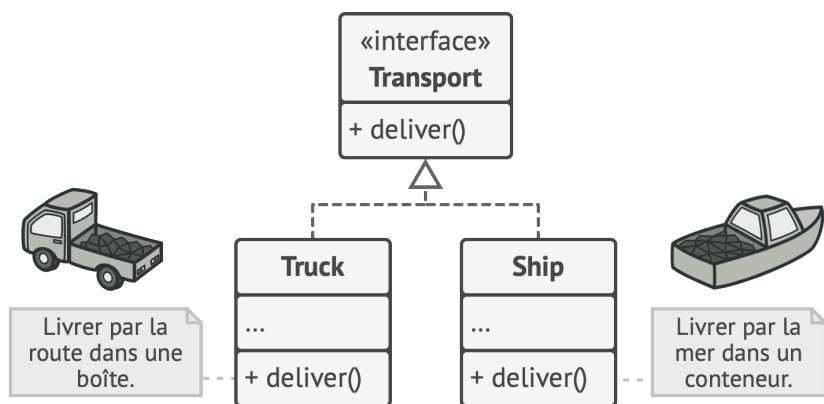


Les sous-classes peuvent modifier les classes des objets retournés par la méthode fabrique.

À première vue, cette modification peut sembler inutile : nous avons juste déplacé l'appel du constructeur dans une autre partie du programme. Mais maintenant, vous pouvez redéfinir

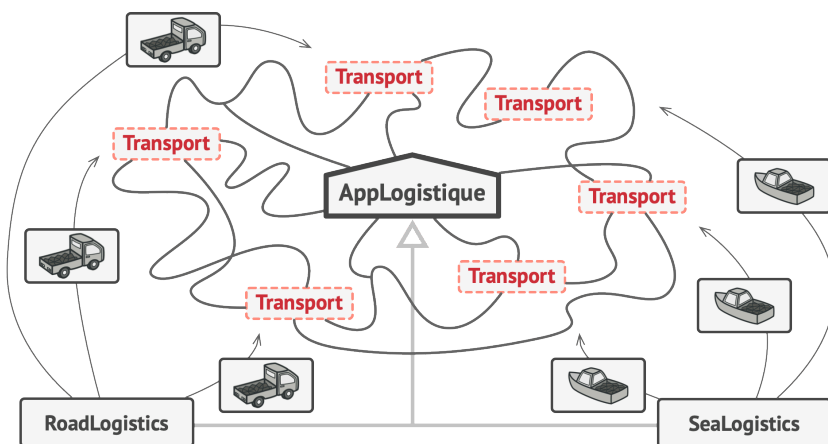
la méthode fabrique dans la sous-classe et changer la classe des produits créés par la méthode.

Il y a tout de même une petite limitation : les sous-classes peuvent retourner des produits différents seulement si les produits ont une classe de base ou une interface commune. De plus, cette interface doit être le type retourné par la méthode fabrique de la classe de base.



Les produits doivent tous implémenter la même interface.

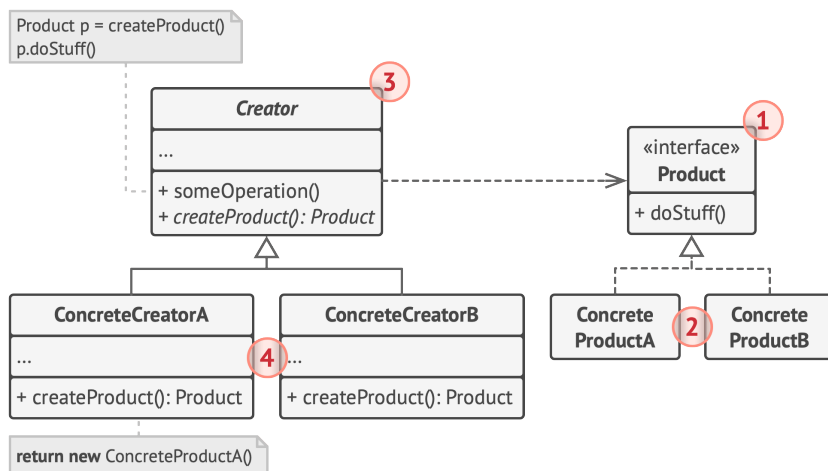
Par exemple, les classes `Camion` et `Bateau` doivent toutes les deux implémenter l'interface `Transport`, qui déclare une méthode `livrer`. Chaque classe implémente cette méthode à sa façon : les camions livrent par la route et les bateaux livrent par la mer. La méthode fabrique de la classe `LogistiqueRoute` retourne des camions, alors que celle de la classe `LogistiqueMer` retourne des bateaux.



Tant que les classes produit implémentent une interface commune, vous pouvez passer leurs objets au code client sans tout faire planter.

Le code qui appelle la méthode fabrique (souvent appelé le code *client*) ne fait pas la distinction entre les différents produits concrets retournés par les sous-classes, il les considère tous comme des `Transports` abstraits. Le client sait que tous les objets transportés sont censés avoir une méthode `livrer`, mais son fonctionnement lui importe peu.

Structure



1. L'interface est déclarée par le **Produit** et est commune à tous les objets qui peuvent être conçus par le créateur et ses sous-classes.
2. Les **Produits Concrets** sont différentes implémentations de l'interface produit.
3. La méthode fabrique est déclarée par la classe **Créateur** et retourne les nouveaux produits. Il est important que son type de retour concorde avec l'interface produit.

Vous pouvez rendre la méthode fabrique abstraite afin d'obliger ses sous-classes à implémenter leur propre version de la méthode ou vous pouvez modifier la méthode fabrique de la classe de base afin qu'elle retourne un type de produit par défaut.

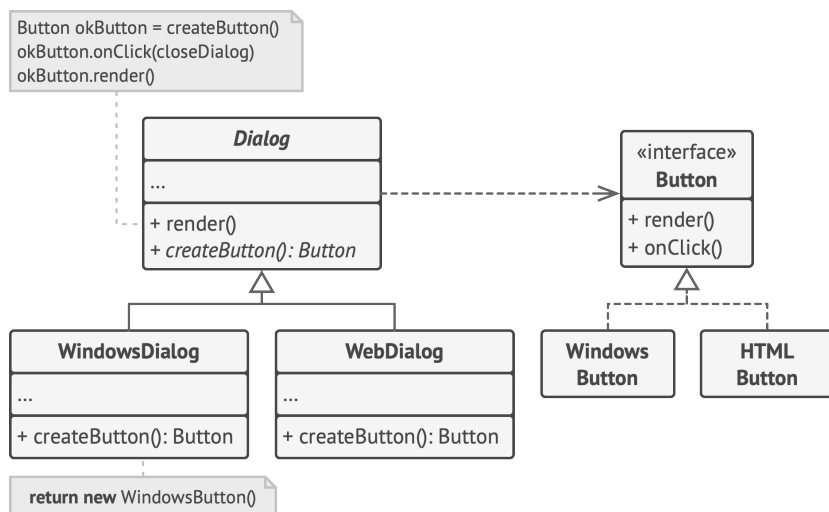
Il faut bien comprendre que malgré son nom, la création de produits n'est **pas** la responsabilité principale du créateur. La classe créateur a en général déjà un fonctionnement propre lié à la nature de ses produits. La fabrique aide à découpler cette logique des produits concrets. C'est un peu comme une grande entreprise de développement de logiciels : elle peut posséder un département spécialisé dans la formation des développeurs, mais son activité principale reste d'écrire du code, pas de produire des développeurs.

4. Les **Créateurs Concrets** redéfinissent la méthode fabrique de la classe de base afin de pouvoir retourner les différents types de produits.

Notez toutefois que la méthode fabrique n'est pas obligée de **créer** tout le temps de nouvelles instances. Elle peut retourner des objets depuis un cache, un réservoir d'objets ou une autre source.

Pseudo-code

Cet exemple montre comment la **fabrique** peut être utilisée pour créer des éléments d'une UI (interface utilisateur) multi-plateforme sans coupler le code client aux classes concrètes de l'UI.



Exemple multiplateforme pour une boîte de dialogue.

La classe de base dialogue utilise différents éléments d'UI pour le rendu de ses fenêtres. Ces éléments peuvent un peu varier en fonction du système d'exploitation, mais ils doivent garder le même comportement. Un bouton sous Windows est aussi un bouton sous Linux.

Lorsque la fabrique entre dans l'équation, il est inutile de réécrire la logique de la boîte de dialogue pour chaque système d'exploitation. Si nous déclarons une méthode fabrique qui crée des boutons à l'intérieur de la classe de base dialogue, nous pourrions par la suite sous-classer cette dernière afin que la méthode fabrique retourne des boutons dotés du style Windows. La sous-classe hérite alors du code de la classe de base dialogue, mais grâce à la fabrique, elle est capable d'afficher à l'écran des boutons à l'allure Windows.

Pour le bon fonctionnement de ce patron, la classe de base dialogue doit utiliser des boutons abstraits représentés par une classe de base ou une interface dont tous les boutons concrets hériteront. Ainsi, quel que soit le type de boutons, le code de la classe dialogue reste fonctionnel.

Bien sûr, cette approche fonctionne également avec les autres éléments de l'UI. Cependant, chaque nouvelle méthode fabrique ajoutée à Dialogue nous rapproche du patron de conception **Fabrique abstraite**. Pas de panique, nous aborderons ce patron plus tard !

```

1  // La classe créateur déclare la méthode fabrique qui doit
2  // renvoyer un objet de la classe produit. Les sous-classes du
3  // créateur fournissent en général une implémentation de cette
4  // méthode.
5  class Dialog is
6      // Le créateur peut également fournir des implémentations
7      // par défaut de la méthode fabrique.
8      abstract method createButton():Button
9
10     // Ne vous laissez pas berner par son nom, la responsabilité
11     // principale du créateur n'est pas de fabriquer des
12     // produits. Il héberge en général de la logique métier qui
13     // concerne les produits retournés par la méthode fabrique.
14     // Les sous-classes peuvent modifier indirectement cette
15     // logique métier en redéfinissant la méthode fabrique et en
16     // lui faisant retourner un type de produit différent.
17     method render() is
18         // Appelle la méthode fabrique pour créer un objet

```



```

19     // Produit.
20     Button okButton = createButton()
21     // Utilise le produit.
22     okButton.onClick(closeDialog)
23     okButton.render()
24
25
26     // Les créateurs concrets redéfinissent la méthode fabrique pour
27     // changer le type du produit qui en résulte.
28     class WindowsDialog extends Dialog is
29         method createButton():Button is
30             return new WindowsButton()
31
32     class WebDialog extends Dialog is
33         method createButton():Button is
34             return new HTMLButton()
35
36
37     // L'interface du produit déclare les traitements que tous les
38     // produits concrets doivent implémenter.
39     interface Button is
40         method render()
41         method onClick(f)
42
43     // Les produits concrets fournissent diverses implémentations de
44     // l'interface du produit.
45     class WindowsButton implements Button is
46         method render(a, b) is
47             // Affiche un bouton avec le style Windows.
48         method onClick(f) is
49             // Attribue un événement sur un clic natif dans un
50             // système d'exploitation.


```


```

51
52 class HTMLButton implements Button is
53     method render(a, b) is
54         // Retourne une représentation HTML d'un bouton.
55     method onClick(f) is
56         // Attribue un événement sur un clic dans un navigateur
57         // Internet.
58
59
60 class Application is
61     field dialog: Dialog
62
63     // L'application choisit un type de créateur en fonction de
64     // la configuration actuelle ou des paramètres
65     // d'environnement.
66     method initialize() is
67         config = readApplicationConfigFile()
68
69         if (config.OS == "Windows") then
70             dialog = new WindowsDialog()
71         else if (config.OS == "Web") then
72             dialog = new WebDialog()
73         else
74             throw new Exception("Error! Unknown operating system.")
75
76     // Le code client manipule une instance d'un créateur
77     // concret, mais uniquement au moyen de son interface de
78     // base. Tant que le client continue de passer par cette
79     // interface pour manipuler le créateur, vous pouvez passer
80     // n'importe quelle sous-classe du créateur.
81     method main() is
82         this.initialize()


```


Possibilités d'application

 **Utilisez la fabrique si vous ne connaissez pas à l'avance les types et dépendances précis des objets que vous allez utiliser dans votre code.**

 La fabrique effectue une séparation entre le code du constructeur et le code qui utilise réellement le produit. Le code du constructeur devient ainsi plus évolutif et indépendant du reste du code.

Par exemple, si vous voulez ajouter un nouveau produit dans l'application, il vous suffit d'ajouter une sous-classe de création et d'y redéfinir la méthode fabrique.

 **Utilisez la fabrique si vous voulez mettre à disposition une librairie ou un framework pour vos utilisateurs avec un moyen d'étendre ses composants internes.**

 L'héritage est probablement le moyen le plus simple pour étendre le comportement par défaut d'une librairie ou d'un framework. Mais comment le framework peut-il savoir qu'il doit utiliser votre sous-classe plutôt qu'un composant standard ?

La solution est de réunir dans une seule méthode fabrique le code qui construit les composants dans le framework, et non

seulement d'étendre ceux-ci, mais de laisser la possibilité de redéfinir la méthode fabrique.

Voyons un exemple d'utilisation. Imaginez la conception d'une application qui utilise un framework d'UI open source. Vous désirez utiliser des boutons ronds, mais le framework ne fournit que des boutons carrés. Vous étendez le `Bouton` standard avec une magnifique sous-classe `BoutonRond`. Mais vous devez à présent expliquer à la classe principale `UIFramework` qu'elle doit utiliser la sous-classe du nouveau bouton plutôt que celle par défaut.

Pour ce faire, vous créez une sous-classe `UIAvecBoutonsRonds` depuis une classe de base du framework et redéfinissez sa méthode `créerBouton`. Même si la méthode de la classe de base retourne des `Boutons`, votre sous-classe renvoie des `BoutonsRonds`. Vous pouvez dorénavant utiliser `UIAvecBoutonsRonds` à la place de `UIFramework`. Et c'est à peu près tout !



Utilisez la fabrique lorsque vous voulez économiser des ressources système en réutilisant des objets au lieu d'en construire de nouveaux.



Le besoin se présente souvent lorsque l'on utilise des objets qui prennent beaucoup de ressources tels que des bases de données, des systèmes de fichiers ou des ressources réseau.

Que faut-il pour réutiliser un objet existant ?

1. Tout d'abord, vous devez créer un moyen de stockage afin de garder la trace de tous les objets créés.
2. Lorsqu'un nouvel objet est demandé, le programme doit chercher un objet libre dans cette réserve.
3. ... et le renvoyer au code client.
4. Si aucun objet n'est disponible, le programme en crée un nouveau (et l'ajoute à la réserve).

Cela représente un paquet de code ! De plus, il faut tout mettre au même endroit afin de ne pas polluer le code avec des doublons.

Il serait probablement plus pratique de l'écrire dans le constructeur de la classe de l'objet que l'on veut réutiliser, mais par définition, un constructeur doit toujours renvoyer de **nouveaux objets**. Il ne peut pas retourner des instances existantes.

C'est pourquoi vous devez disposer d'une méthode non seulement capable de créer de nouveaux objets, mais aussi de réutiliser ceux qui existent déjà. Cela ressemble énormément à un patron de conception fabrique.



Mise en œuvre

1. Implémentez la même interface pour tous les produits. Cette interface doit déclarer des méthodes que tous les produits peuvent avoir en commun.

2. Ajoutez une méthode fabrique vide à l'intérieur de la classe créateur. Le type de retour de la méthode doit correspondre à l'interface commune des produits.
3. Localisez toutes les références aux constructeurs des produits dans le code du créateur. Remplacez-les une par une par des appels à la méthode fabrique et déplacez le code de la création de produits dans la méthode fabrique.

Vous allez peut-être devoir ajouter un paramètre temporaire à la méthode fabrique pour vérifier le type du produit retourné.

À ce stade, le code de la méthode fabrique peut paraître désordonné. Il pourrait même contenir un gros `switch` qui choisit la classe à instancier. Ne vous inquiétez pas, tout va bientôt rentrer dans l'ordre.

4. Pour chaque type de produit listé dans la méthode fabrique, créez une sous-classe de Créateur. Redéfinissez la méthode fabrique dans les sous-classes et récupérez les morceaux de code appropriés de la méthode de base.
5. S'il y a trop de types de produits et peu d'intérêt de créer des sous-classes pour tous, vous pouvez réutiliser le paramètre de contrôle de la classe de base dans les sous-classes.

Imaginons la hiérarchie de classes suivante : la classe de base `Courrier` avec les sous-classes `CourrierAérien` et `CourrierTerrestre` ; les classes de `Transport` sont `Avion` ,

`Camion` et `Train`. La classe `CourrierAérien` n'utilise que des `Avions` et la classe `CourrierTerrestre` peut utiliser à la fois des `Camions` et des `Trains`. Vous pouvez créer une nouvelle sous-classe (`CourrierFerroviaire` par exemple) pour gérer les deux cas, mais il y a une autre possibilité. Le code client peut passer un argument à la méthode fabrique du `CourrierTerrestre` pour désigner le type de produit qu'elle veut recevoir.

6. Si après tous ces changements la méthode fabrique de base est devenue complètement vide, vous pouvez la rendre abstraite. S'il reste encore quelques lignes, vous pouvez y laisser un comportement par défaut.

Avantages et inconvénients

- ✓ Vous désolidarisez le Créateur des produits concrets.
- ✓ *Principe de responsabilité unique.* Vous pouvez déplacer tout le code de création des produits au même endroit, permettant ainsi une meilleure maintenabilité.
- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouveaux types de produits dans le programme sans endommager l'existant.
- ✗ Le code peut devenir plus complexe puisque vous devez introduire de nombreuses sous-classes pour la mise en place du patron. La condition optimale d'intégration du patron dans du code existant se présente lorsque vous avez déjà une hiérarchie existante de classes de création.

⇔ Liens avec les autres patrons

- La **Fabrique** est souvent utilisée dès le début de la conception (moins compliquée et plus personnalisée grâce aux sous-classes) et évolue vers la **Fabrique abstraite**, le **Prototype**, ou le **Monteur** (ce dernier étant plus flexible, mais plus compliqué).
- Les classes **Fabrique abstraite** sont souvent basées sur un ensemble de **Fabriques**, mais vous pouvez également utiliser le **Prototype** pour écrire leurs méthodes.
- Vous pouvez utiliser la **Fabrique** avec l'**Itérateur** pour permettre aux sous-classes des collections de renvoyer différents types d'itérateurs compatibles avec les collections.
- Le **Prototype** n'est pas basé sur l'héritage, il n'a donc pas ses désavantages. Mais le *prototype* requiert une initialisation compliquée pour l'objet cloné. La **Fabrique** est basée sur l'héritage, mais n'a pas besoin d'une étape d'initialisation.
- La **Fabrique** est une spécialisation du **Patron de méthode**. Une *fabrique* peut aussi faire office d'étape dans un grand *patron de méthode*.

346 pages

du livre complet sont absentes de la version démo