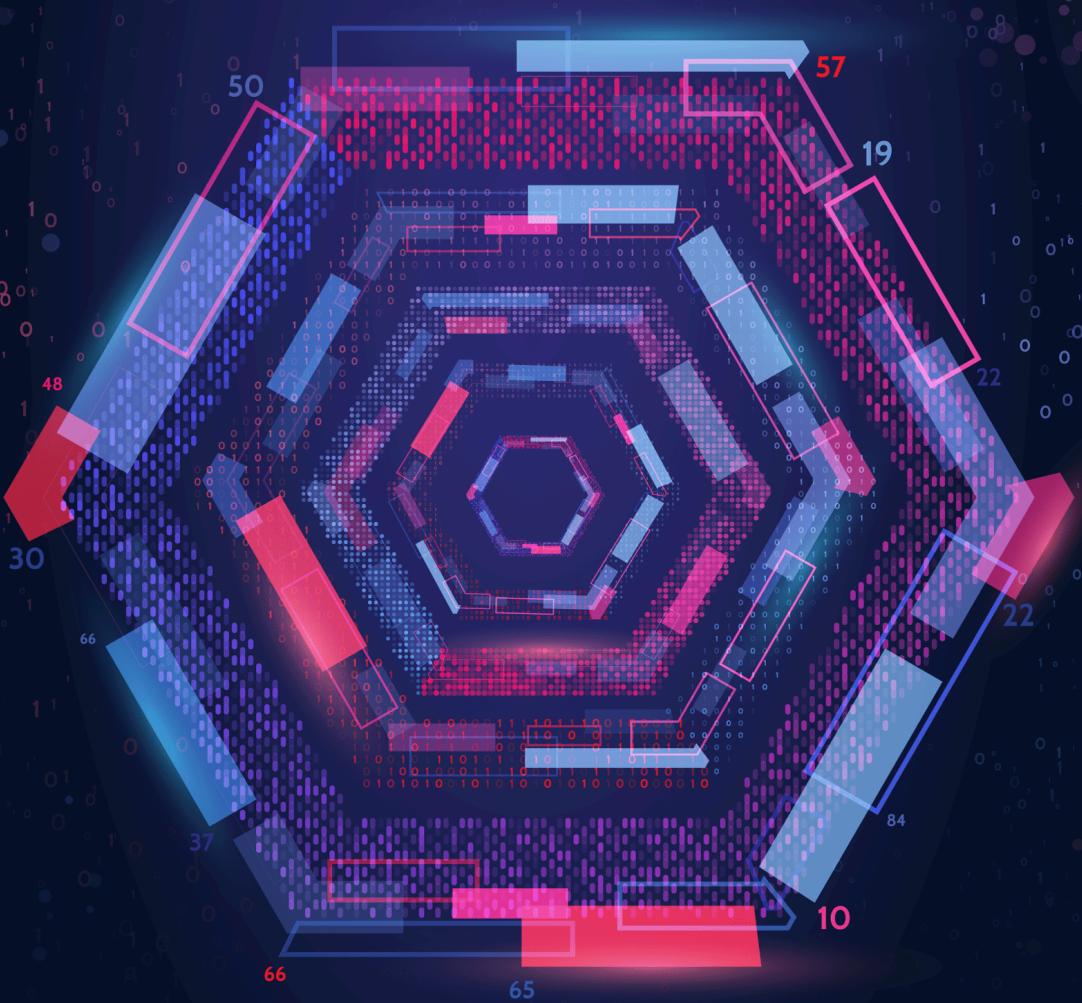


Plongée au cœur des

PATRONS DE CONCEPTION



Alexander Shvets

Plongée au cœur des

PATRONS DE CONCEPTION

v2023-1.17

Acheté par Baptiste Bauer
bbauer02@gmail.com (#73611)

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*¹ et du cours en ligne *Dive Into Refactoring*².



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, 2022

✉ support@refactoring.guru

🖼 Illustrations : Dmitry Zhart

📄 Traduction : David Simon

📝 Révision : Luc Perret

-
1. *Plongée au cœur des patrons de conception*:
<https://refactoring.guru/fr/design-patterns/book>
 2. *Dive Into Refactoring*:
<https://refactoring.guru/fr/refactoring/course>

*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
§ S: Principe de responsabilité unique.....	58
§ O: Principe ouvert/fermé	61
§ L: Principe de substitution de Liskov	65
§ I: Principe de ségrégation des interfaces.....	72
§ D: 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 ancrages. À la fin de chaque chapitre, une liste de liens présentant les relations entre ce patron et d'autres vous est proposée. Si vousapercevez 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.

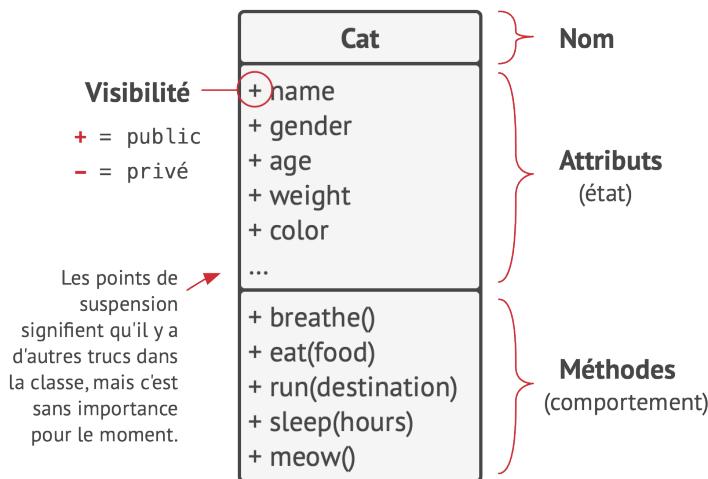
INTRODUCTION À LA POO

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.

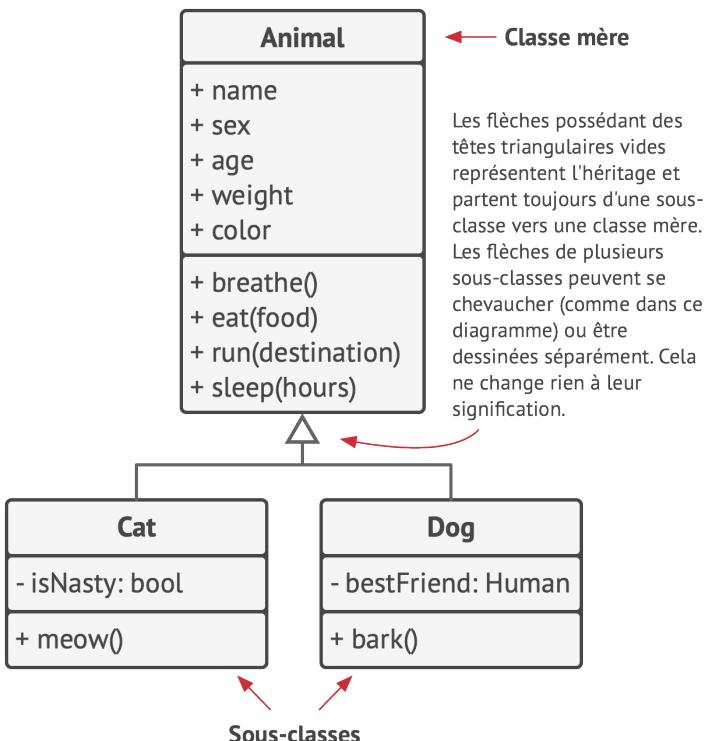
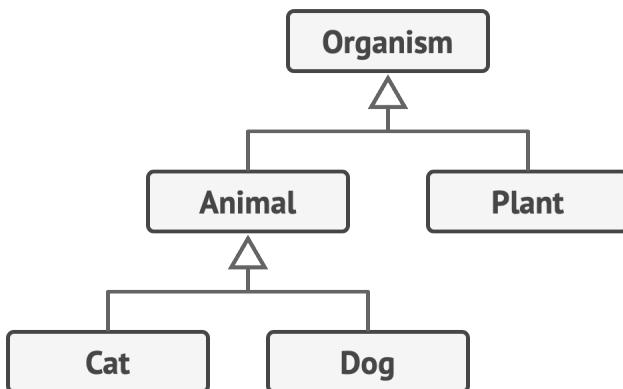


Diagramme UML d'une hiérarchie simple de classes. Toutes les classes de ce diagramme font partie de la hiérarchie de classes `Animal`.

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.

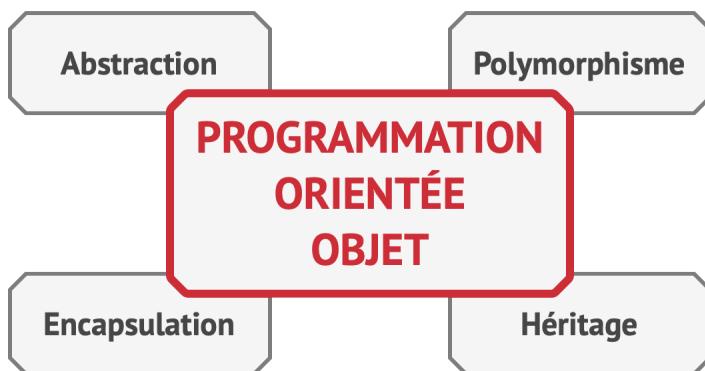


Les classes d'un diagramme UML peuvent être simplifiées s'il est plus important d'afficher leurs relations que leur contenu.

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.

Les piliers de la POO

La programmation orientée objet est basée sur quatre piliers : des concepts qui la différencient des autres paradigmes de la programmation.

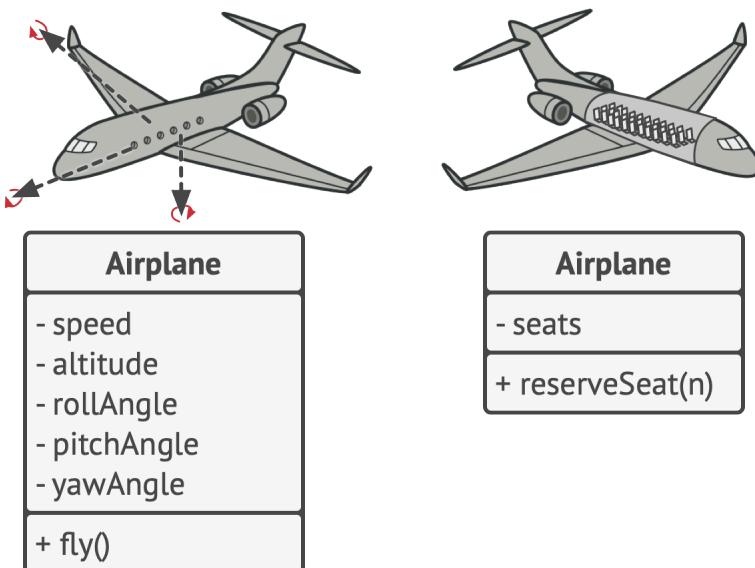


Abstraction

Lorsque vous créez un programme avec la POO, la majeure partie du temps vous allez modéliser les objets en les basant sur des objets du monde réel. Cependant, les objets de votre programme ne représentent pas une copie de l'original à 100 % (et on a rarement besoin que ce soit le cas). Vos objets ne font que *modéliser* les attributs et comportements des objets réels dans un contexte spécifique, et ignorent le reste.

Par exemple, une classe `Avion` pourrait exister dans un simulateur de vol et dans l'application d'une compagnie aérienne qui permet de réserver des places. Mais dans le premier cas, ce

sont les données qui concernent le vol en lui-même que l'on va utiliser, alors que dans le second cas, la classe va plutôt s'occuper des informations concernant l'agencement intérieur et la disponibilité des places.



Différentes modélisations du même objet du monde réel.

L'abstraction est la modélisation d'un objet ou phénomène du monde réel, limité à un contexte spécifique dont on relève tous les détails utiles avec une grande précision, et dont on ignore le reste.

Encapsulation

Pour démarrer le moteur d'une voiture, vous n'avez qu'à tourner une clé ou à appuyer sur un bouton. Vous n'avez pas besoin

de connecter les câbles sous le capot, de faire tourner le vilebrequin et les cylindres, et d'amorcer le cycle d'alimentation du moteur. Ces détails sont cachés sous le capot de la voiture. Vous n'avez qu'une interface toute simple : un interrupteur de démarrage, un volant et des pédales. C'est un exemple parfait pour illustrer l'**interface** d'un objet - la partie publique d'un objet qui est ouverte aux interactions avec les autres objets.

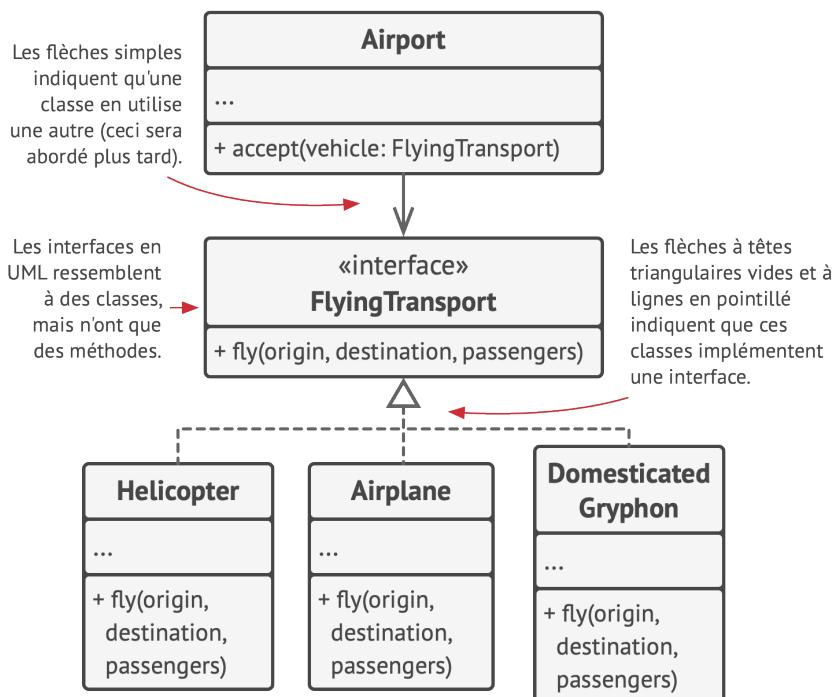
L'encapsulation est la technique qu'un objet utilise pour cacher une partie de son état et de ses comportements aux autres objets, pour ne révéler qu'une interface limitée au reste du programme.

Encapsuler quelque chose signifie qu'on la rend `private`, et donc accessible uniquement aux méthodes de sa propre classe. Il existe un mode légèrement moins restrictif que l'on appelle `protected`, qui rend un membre accessible aux sous-classes.

Les interfaces et les classes/méthodes abstraites de la majorité des langages de programmation sont basées sur les concepts de l'abstraction et de l'encapsulation. Dans les langages de programmation orientés objet modernes, le mécanisme des interfaces (déclaré généralement à l'aide du mot clé `interface` ou `protocol`) vous permet d'établir des contrats d'interaction entre les objets. C'est l'une des raisons pour laquelle les interfaces ne se préoccupent que des comportements des objets, et que vous ne pouvez pas déclarer un attribut dans une interface.

Le mot *interface* représente la partie publique d'un objet, mais il existe également un type `interface` dans la majorité des langages de programmation, ce qui est très perturbant, je vous l'accorde.

Imaginez une interface `TransportAérien` dotée d'une méthode `voler(origine, destination, passagers)`. Lorsque vous concevez un simulateur de transport aérien, vous pouvez mettre une restriction sur la classe `Aéroport`, afin qu'elle ne puisse interagir qu'avec les objets qui implémentent l'interface `TransportAérien`.



Le diagramme UML de plusieurs classes implémentant une interface.

Grâce à cette configuration, vous pouvez être certain que n'importe quel objet passé à un objet aéroport, que ce soit un Avion, un Hélicoptère ou même carrément un GriffonDomestiqué, pourra s'envoler ou atterrir dans ce type d'aéroport.

Vous pouvez modifier l'implémentation de la méthode voler dans ces classes comme bon vous semble. Tant que la signature de la méthode reste identique à celle déclarée dans l'interface, toutes les instances de la classe Aéroport peuvent manipuler vos objets volants sans problème.

Héritage

L'*héritage* permet de construire de nouvelles classes à partir de classes existantes. Il favorise la réutilisation du code. Si vous voulez créer une classe qui est légèrement différente d'une classe existante, vous n'avez pas besoin de dupliquer votre code. À la place, vous pouvez étendre la classe existante dans une sous-classe qui hérite de ses attributs et méthodes et écrire dans cette sous-classe les fonctionnalités supplémentaires.

Les sous-classes qui utilisent l'héritage bénéficient de la même interface que leur classe mère. Vous ne pouvez pas cacher une méthode dans une sous-classe si elle a été déclarée dans la classe mère. Vous devez également implémenter toutes les méthodes abstraites dans vos sous-classes, même si leur présence y paraît illogique.

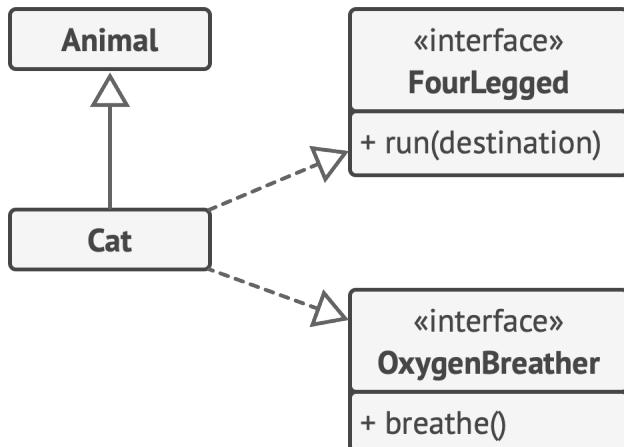


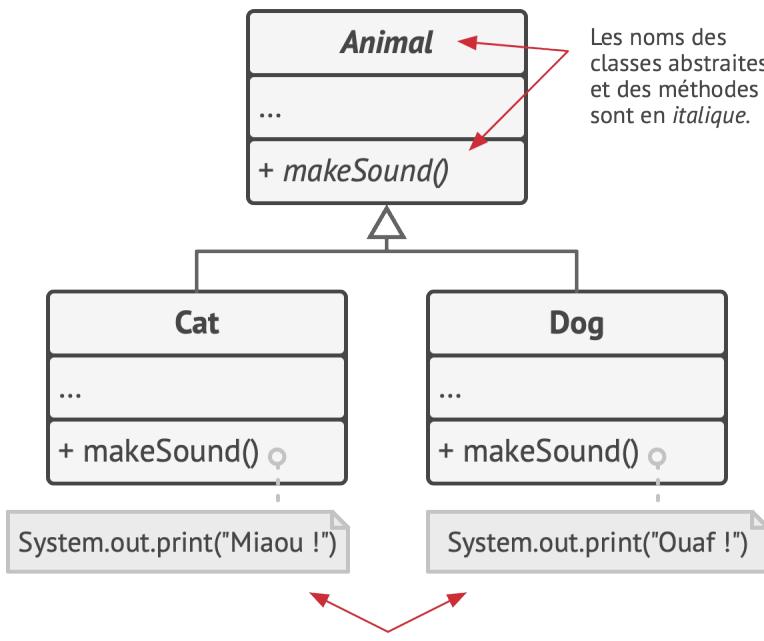
Diagramme UML de l'héritage d'une seule classe contre l'implémentation de plusieurs interfaces en même temps.

Dans la majorité des langages de programmation, une sous-classe ne peut étendre qu'une seule classe mère. Mais dans l'autre sens, une classe peut implémenter autant d'interfaces en même temps qu'elle le veut. Comme je l'ai déjà dit, si une classe mère implémente une interface, toutes ses sous-classes doivent également l'implémenter.

Polymorphisme

Regardons quelques exemples d'animaux. La majorité des Animaux peut émettre des sons. Nous pouvons donc anticiper le fait que toutes les sous-classes vont devoir redéfinir la méthode de base émettreSon afin que chaque classe puisse émettre le son correspondant; nous la déclarons donc immédiatement *abstract*. Ceci nous permet d'ignorer toute implé-

mentation par défaut de la méthode dans la classe mère, mais oblige les sous-classes à posséder leur propre version.



Ce sont des commentaires UML. En général, ils expliquent les détails de l'implémentation des classes ou méthodes données.

Imaginez que vous mettez plusieurs chats et chiens dans un grand sac. Ensuite vous fermez les yeux, et sortez les animaux du sac un par un. Lorsque nous sortons un animal du sac, nous ne pouvons pas dire avec certitude ce que c'est. Cependant, si nous le câlinons assez fort, l'animal va émettre un cri de joie en fonction de sa classe concrète.

```
1 bag = [new Cat(), new Dog()];
2
3 foreach (Animal a : bag)
4     a.makeSound()
5
6 // Miaou !
7 // Ouaf !
```

Le programme ne connaît pas le type concret de l'objet obtenu dans la variable `a`, mais grâce au mécanisme particulier du *polymorphisme*, le programme peut retrouver la sous-classe de l'objet dont la méthode est appelée et lancer le comportement approprié.

Le *polymorphisme* est la capacité d'un programme à détecter la classe d'un objet et à appeler son implémentation, même si son type est inconnu dans le contexte actuel.

Vous pouvez aussi voir le polymorphisme comme la possibilité pour un objet de « prétendre » être autre chose (en général une classe qu'il étend, ou une interface qu'il implémente). Dans notre exemple, les chiens et chats du sac prétendent être des animaux génériques.

Relations entre les objets

Nous avons déjà vu l'*héritage* et l'*implémentation*, mais il y a d'autres types de relations entre objets que nous n'avons pas encore abordés.

Dépendance



Dépendance en UML. Le professeur dépend de son cours.

La *dépendance* est la relation la plus basique, mais aussi la plus faible entre les classes. Deux classes sont dépendantes si certaines modifications apportées à la définition de l'une d'entre elles occasionnent des changements dans une autre classe. Les dépendances sont généralement créées lorsque vous utilisez le nom de classes concrètes dans votre code. Par exemple, lorsque vous mettez des types dans les signatures de vos méthodes, lorsque vous instanciez les objets par un appel de constructeur, etc. Vous pouvez affaiblir ces dépendances en utilisant des interfaces ou des classes abstraites à la place.

En général, un diagramme UML ne montre pas toutes les dépendances, il y en a bien trop dans le code réel. Plutôt que de polluer le diagramme UML avec des dépendances, soyez sélec-

tifs et ne mettez que celles qui sont importantes, en fonction de ce que vous voulez communiquer.

Association



Association en UML. Un professeur communique avec ses élèves.

L'*association* est une relation dans laquelle un objet utilise ou interagit avec un autre objet. Dans un diagramme UML, la relation d'association est représentée par une flèche simple dessinée depuis un objet, vers l'objet qu'il utilise. Si vous apercevez une flèche bidirectionnelle, c'est tout à fait normal. Dans ce cas, la flèche possède une pointe à chacune de ses extrémités. L'association peut être vue comme une sorte de dépendance spécialisée dans laquelle un objet a toujours accès à l'objet avec lequel il interagit, ce qui n'est pas le cas avec une dépendance simple.

En général, on utilise une association pour représenter quelque chose, comme un attribut dans une classe. Le lien est toujours présent : vous pouvez toujours demander à un client ce qu'il souhaite commander. Mais il ne s'agit pas forcément d'un attribut. Si vous modélisez vos classes depuis la perspective d'une interface, cela peut indiquer la présence d'une méthode qui retourne la commande du client.

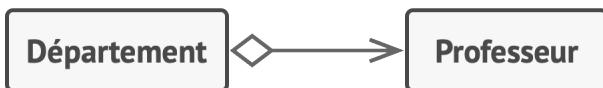
Pour consolider votre compréhension de la différence entre une association et une dépendance, regardons un exemple combiné. Imaginons une classe `Professeur` :

```
1 class Professor is
2   field Student student
3   // ...
4   method teach(Course c) is
5     // ...
6     this.student.remember(c.getKnowledge())
```

Jetons un œil à la méthode `enseigner`. Elle prend un paramètre de la classe `Cours`, qui est ensuite envoyé dans le corps de la méthode. Si quelqu'un modifie la méthode `getConnaissances` de la classe `Cours` (modifier son nom, ajouter des paramètres obligatoires, etc.), notre code va planter. C'est ce que l'on appelle une dépendance.

Maintenant, regardez l'attribut `étudiant` et son utilisation dans la méthode `enseigner`. Nous pouvons affirmer que la classe `Étudiant` est également une dépendance du `Professeur` : si la méthode `seSouvenir` est modifiée, alors le code du `Professeur` va planter. Cependant, puisque l'attribut `étudiant` est toujours accessible à n'importe quelle méthode du `Professeur`, la classe `Étudiant` n'est plus seulement une dépendance, mais une association.

Agrégation



Agrégation en UML. Un département contient des professeurs.

L’*agrégation* est un type spécialisé d’association qui représente « 1 à plusieurs », « plusieurs à plusieurs » ou une relation « agrégat-agrégré » entre plusieurs objets.

En général avec l’agrégation, un objet « a » est un ensemble d’autres objets et sert de conteneur ou de collection. Le composant peut exister sans le conteneur et peut être relié à plusieurs conteneurs en même temps. En UML, on représente la relation d’agrégation avec une ligne et un diamant vide du côté du conteneur (agrégat) et une flèche qui pointe vers le composant (agrégré).

J’en profite pour préciser que nous sommes en train de parler de relations entre objets, mais ce sont bien des relations entre *classes* qui sont représentées en UML. Cela signifie qu’un objet université peut avoir plusieurs départements, même si vous ne voyez qu’un seul « bloc » pour chaque entité dans le diagramme. Il est possible d’indiquer les quantités des deux côtés des relations, mais vous pouvez les omettre si le contexte les rend évidentes.

Composition



Composition en UML. Une université est composée de départements.

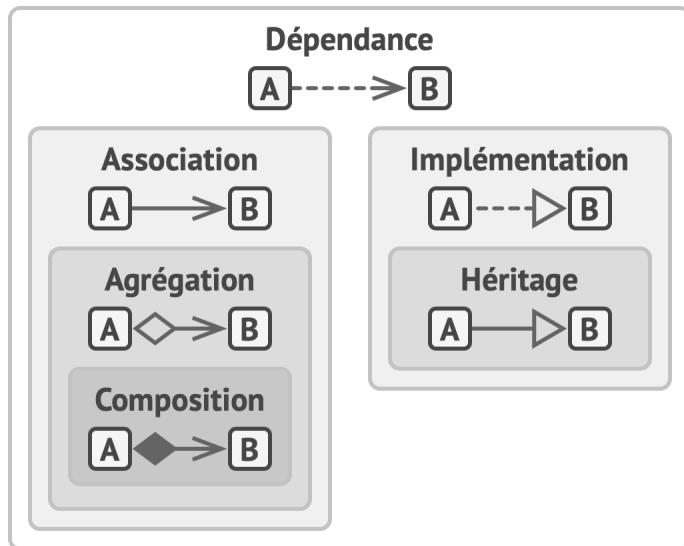
La *composition* est une sorte d'agrégation dans laquelle un objet est composé d'une ou plusieurs instances d'un autre. La différence avec les autres types est que le composant ne peut exister qu'en tant que partie du conteneur. En UML, la relation de composition est dessinée comme l'agrégation, mais le diamant est plein.

Vous noterez que la tendance est d'utiliser le terme « composition » alors que l'on fait référence aussi bien à la composition qu'à l'agrégation. Le parfait exemple est le fameux principe de « préférer la composition à l'héritage ». Ce n'est pas parce que les gens ne sont pas capables de faire la différence, mais simplement que le mot « composition » (composition de l'objet) est plus naturel en anglais, et l'expression a ensuite été littéralement reprise en français.

Vue d'ensemble

Maintenant que nous connaissons tous les types de relations entre les objets, voyons comment ils sont connectés. Ceci devrait répondre à toutes vos questions du genre « Quelle est la différence entre l'agrégation et la composition ? » ou « L'héritage est-il un type de dépendance ? ».

- **Dépendance** : La classe A peut être impactée par les modifications apportées à la classe B.
- **Association** : L'objet A connaît l'objet B. La classe A dépend de B.
- **Agrégation** : L'objet A connaît l'objet B et contient des B. La classe A dépend de B.
- **Composition** : L'objet A connaît l'objet B, contient des B et gère le cycle de vie de B. La classe A dépend de B.
- **Implémentation** : La classe définit des méthodes déclarées dans l'interface B. Les objets A sont traités comme des B. La classe A dépend de B.
- **Héritage** : La classe A hérite de l'interface et de l'implémentation de la classe B, mais elle peut l'étendre. Les objets A peuvent être traités comme des B. La classe A dépend de B.



Les relations entre les objets et les classes : de la plus faible à la plus forte.

INTRODUCTION AUX PATRONS DE CONCEPTION

Qu'est-ce qu'un patron de conception ?

Les **patrons de conception** sont des solutions classiques à des problèmes récurrents de la conception de logiciels. Ce sont des sortes de plans ou de schémas que l'on peut personnaliser afin de résoudre un problème récurrent dans notre code.

Vous ne pouvez pas vous contenter de trouver un patron et de le recopier dans votre programme comme vous le feriez avec des fonctions ou des librairies prêtées à l'emploi. Un patron, ce n'est pas un bout de code spécifique, mais plutôt un concept général pour résoudre un problème précis. Vous pouvez suivre le principe du patron et implémenter une solution qui convient à votre propre programme.

Les patrons sont souvent confondus avec les algorithmes, car ils décrivent tous deux des solutions classiques à des problèmes connus. Un algorithme définit toujours clairement un ensemble d'actions qui va vous mener vers un objectif, alors qu'un patron, c'est la description d'une solution à un plus haut niveau. Le code utilisé pour implémenter un même patron peut être complètement différent s'il est appliqué à deux programmes distincts.

Un algorithme c'est un peu comme une recette de cuisine, ses étapes sont claires et vous guident vers un objectif précis. Un

patron, c'est plutôt comme un plan : vous pouvez voir ses fonctionnalités et les résultats obtenus, mais la manière de l'implémenter vous revient.

⬇ Que trouve-t-on dans un patron de conception ?

La majorité des patrons sont présentés de façon très générale, afin qu'ils soient reproductibles dans tous les contextes. Voici les différentes sections que vous retrouverez habituellement dans la description d'un patron :

- L'**Intention** du patron permet de décrire brièvement le problème et la solution.
- La **Motivation** explique en détail la problématique et la solution offerte par le patron.
- La **Structure** des classes montre les différentes parties du patron et leurs relations.
- L'**Exemple de code** écrit dans un des langages de programmation les plus populaires facilite la compréhension générale de l'idée derrière le patron.

Vous retrouverez toute une liste de détails pratiques dans certains catalogues de patrons : des cas d'utilisation, les étapes de l'implémentation et les liens avec d'autres patrons.

Classification des patrons de conception

Les patrons diffèrent par leur complexité, leur niveau de détails et l'échelle à laquelle ils sont applicables à un système en cours de conception. Je trouve l'analogie faite avec la construction d'une route plutôt parlante : vous pouvez installer des feux de circulation pour renforcer la sécurité d'une intersection ou construire un échangeur à plusieurs niveaux et muni de passages souterrains pour les piétons.

Les patrons les plus basiques et de plus bas niveau sont souvent appelés des *idiomes*. Ils sont généralement conçus pour un langage de programmation spécifique.

Les patrons de conception les plus universels et de plus haut niveau sont des *patrons d'architecture*. Les développeurs peuvent implémenter ces patrons dans à peu près n'importe quel langage. Contrairement aux autres patrons, ils peuvent être utilisés pour concevoir la totalité de l'architecture d'une application.

De plus, tous les patrons de conception peuvent être catégorisés selon leur *intention* ou leur objectif. Ce livre couvre les trois groupes principaux de patrons :

- Les **Patrons de création** fournissent des mécanismes de création d'objets, ce qui augmente la flexibilité et la réutilisation du code.

- Les **Patrons structurels** expliquent comment assembler des objets et des classes en de plus grandes structures, tout en les gardant flexibles et efficaces.
- Les **Patrons comportementaux** mettent en place une communication efficace et répartissent les responsabilités entre les objets.

Qui a inventé les patrons de conception ?

La question est légitime, mais le terme est mal choisi. Les patrons de conception ne sont pas des concepts obscurs et sophistiqués, c'est plutôt l'inverse. Ce sont des solutions classiques à des problèmes connus en conception orientée objet. Lorsqu'une question revient encore et encore dans différents projets, quelqu'un se décide finalement à détailler la solution et à lui donner un nom. C'est souvent comme cela qu'un patron est découvert.

Le concept de patron de conception a d'abord été décrit par Christopher Alexander dans *A Pattern Language: Towns, Buildings, Construction*¹. Le livre invente un « langage » pour concevoir un environnement urbain. Les unités de ce langage sont des patrons. Ils peuvent indiquer la hauteur attendue des fenêtres, le nombre d'étages qu'un bâtiment devrait avoir, la taille des espaces verts d'un quartier, etc.

1. *A Pattern Language: Towns, Buildings, Construction:*
<https://refactoring.guru/fr/pattern-language-book>

Cette idée a été reprise par quatre auteurs : Erich Gamma, John Vlissides, Ralph Johnson, et Richard Helm. En 1994, ils ont publié *Design Patterns : Catalogue de modèles de conception réutilisables*¹, dans lequel ils ont appliqué ce concept de patrons à la programmation. Le livre contient 23 patrons qui résolvent différents problèmes de conception orientée objet et est très rapidement devenu un best-seller. Son nom était un peu trop long et au cours du temps, il a naturellement été remplacé par « the book by the Gang of Four » (le livre du gang des quatre), puis encore raccourci en « the GoF book ».

Depuis ce jour, des dizaines de nouveaux patrons de conception pour la programmation orientée objet ont été découverts. L'utilisation des patrons est devenue tellement populaire dans les autres domaines de la programmation, que de nombreux patrons leur sont spécifiques et ne s'appliquent pas à la programmation orientée objet.

-
1. Design Patterns : Catalogue de modèles de conception réutilisables :
<https://refactoring.guru/fr/gof-book>

Pourquoi devrais-je apprendre les patrons ?

En vérité, il est possible de travailler comme développeur pendant de nombreuses années sans avoir jamais appris un seul patron. C'est le cas pour de nombreuses personnes. Mais même si vous faites partie de ces personnes, il est fort probable que vous les utilisiez sans vous en rendre compte. Pourquoi devriez-vous donc prendre le temps de les apprendre ?

- Les patrons de conception sont une boîte à outils de **solutions fiables et éprouvées** utilisées en réponse à des problèmes classiques de la conception de logiciels. Même si vous n'avez jamais rencontré ces problèmes, le fait de connaître les patrons est tout de même utile, car il vous enseigne des techniques qui permettent de résoudre toutes sortes de problèmes en utilisant les principes de la conception orientée objet.
- Les patrons de conception définissent un langage commun que vous et vos collègues pouvez utiliser pour mieux communiquer. Vous pouvez dire : « Oh, tu n'as qu'à utiliser un singleton. » et tout le monde comprendra instantanément ce que vous venez de suggérer. Nul besoin d'expliquer ce qu'est un singleton, si vous connaissez déjà son nom et son principe.

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 : librairies 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 muri 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.

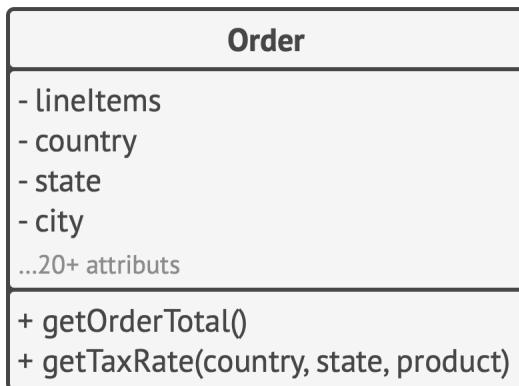
```
1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     total += total * getTaxRate(order.country)
7
8     return total
9
10
10 method getTaxRate(country) is
11     if (country == "US")
12         return 0.07 // 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.

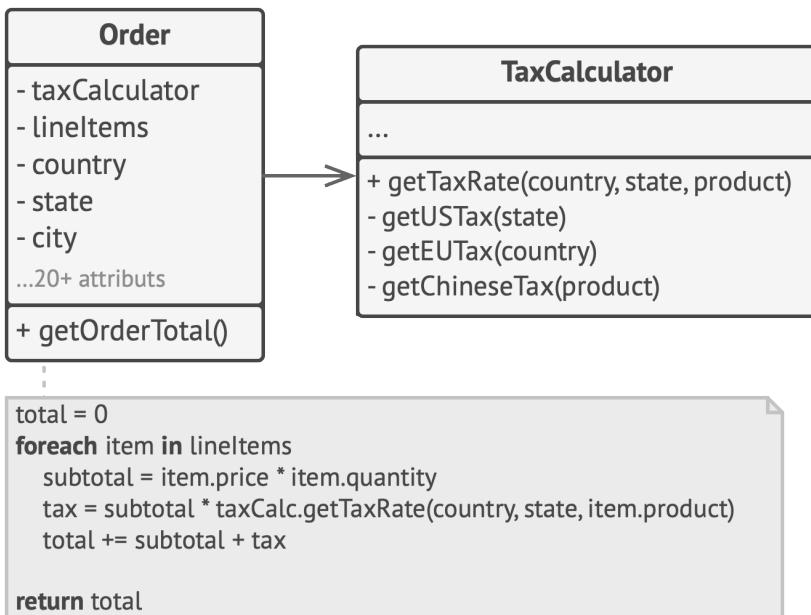
L'encapsulation au niveau des classes

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`.

Programmez avec les interfaces, et non pas avec les implémentations

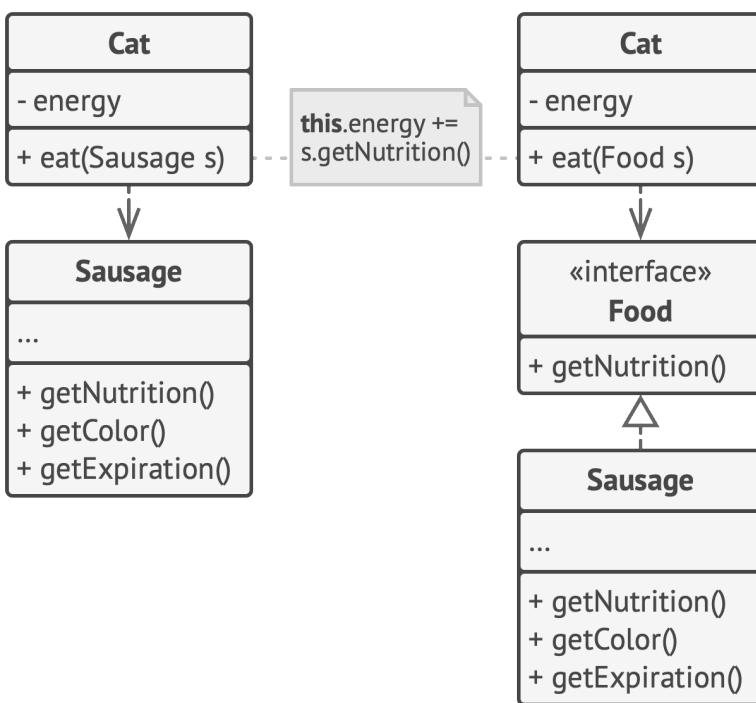
Programmez avec les interfaces, et non pas avec les implémentations. Dépendez des abstractions, pas des classes concrètes.

Vous pouvez qualifier une conception de flexible si vous pouvez facilement l'étendre sans modifier le code existant. Assurons-nous de la véracité de cette affirmation avec un autre exemple de chat. Un `Chat` qui peut manger n'importe quelle nourriture est plus flexible qu'un autre qui ne mange que des saucisses. Vous pouvez tout de même donner des saucisses à manger au premier chat, car c'est un sous-ensemble de « nourriture ». Vous pouvez étendre le menu de ce chat avec n'importe quel type de nourriture.

Lorsque vous voulez faire collaborer deux classes, vous commencez par en rendre une dépendante de l'autre. Mince, même moi, c'est ce que je commence souvent par faire. Cependant, il y a une solution plus flexible pour faire collaborer des objets.

1. Déterminez exactement pourquoi un objet a besoin de l'autre : quelles méthodes exécute-t-il ?
2. Décrivez ces méthodes dans une nouvelle interface ou classe abstraite.

3. Faites implémenter cette interface à la classe qui est une dépendance.
4. Maintenant, rendez la seconde classe dépendante de cette interface, plutôt que de la rendre dépendante de la classe concrète. Vous pouvez toujours la faire manipuler des objets de la classe originale, mais le lien est maintenant beaucoup plus flexible.



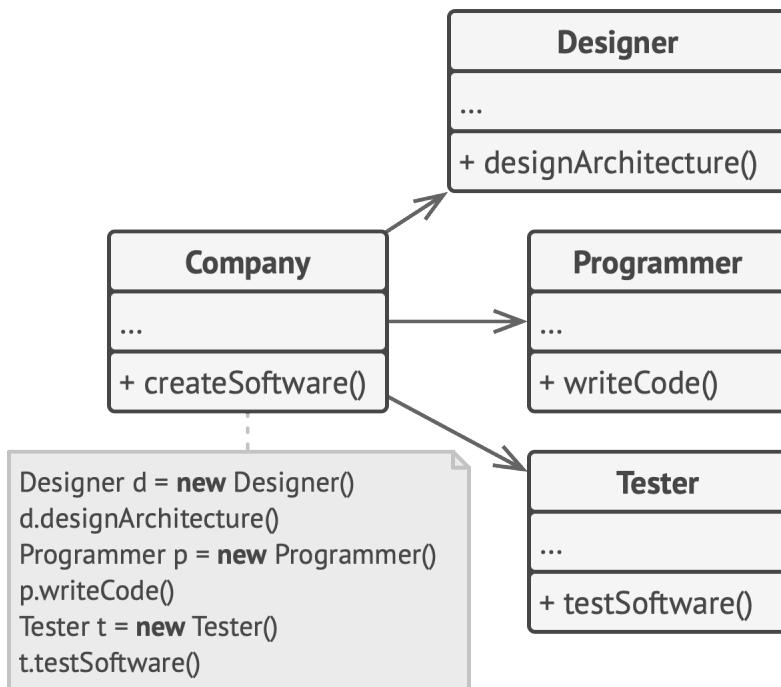
Avant et après avoir extrait l'interface. Le code de droite est plus flexible que celui de gauche, mais également plus compliqué.

Vous ne sentirez pas immédiatement les bénéfices de cette modification et c'est même plutôt le contraire, car le code est devenu plus compliqué qu'avant. Cependant, si vous pen-

sez que vous allez avoir besoin d'un bon point d'extension pour des fonctionnalités supplémentaires ou que d'autres personnes vont vouloir étendre votre code, foncez !

Exemple

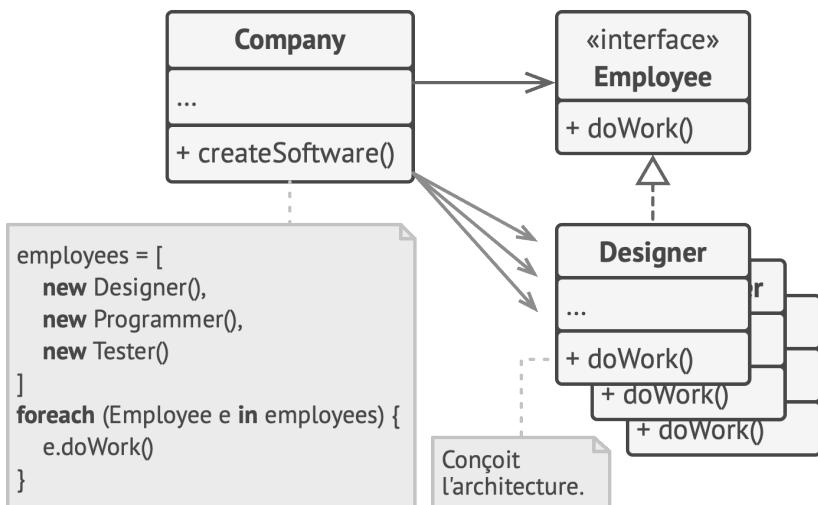
Regardons un autre exemple qui démontre que l'on obtient plus de bénéfices en utilisant des interfaces qu'en dépendant de classes concrètes. Imaginons la création d'un simulateur de sociétés qui éditent des logiciels. Diverses classes représentent les types d'employés.



AVANT : toutes les classes sont fortement couplées.

Au début, la classe `Société` était fortement couplée aux classes concrètes des employés. Malgré la différence dans leurs implementations, nous pouvons généraliser plusieurs méthodes liées au travail et extraire une interface commune pour toutes les classes des employés.

Ensuite, nous pouvons utiliser le polymorphisme à l'intérieur de la classe `Société` et manipuler les divers employés via l'interface `Employé`.

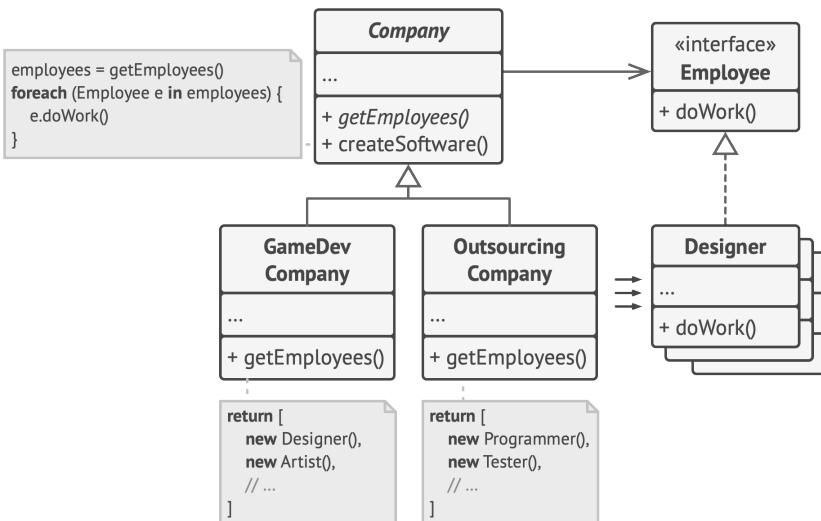


MIEUX : le polymorphisme nous a aidés à simplifier le code, mais le reste de la classe `Société` dépend toujours des classes concrètes des employés.

La classe `Société` est toujours couplée aux classes des employés. Nous allons nous retrouver en mauvaise posture si vous devons ajouter de nouveaux types de sociétés qui hébergent d'autres types d'employés, car nous allons devoir redéfinir

la majeure partie de la classe `Société`, plutôt que de réutiliser son code.

Pour résoudre ce problème, nous pourrions déclarer *abstraite* une méthode pour récupérer des employés. Chaque société concrète va ainsi implémenter cette méthode différemment, et créer uniquement les employés dont elle a besoin.



APRÈS : la méthode principale de la classe `Société` est indépendante des classes concrètes des employés. Les objets employé sont créés dans les sous-classes concrètes de la société.

Après toutes ces modifications, la classe `Société` est devenue indépendante des diverses classes des employés. Vous pouvez maintenant l'étendre et introduire de nouveaux types de sociétés et d'employés, tout en réutilisant une partie du code de la classe de base société. Étendre la classe de base société ne modifie pas le code existant qui en dépend.

Au fait, nous venons juste d'assister à la mise en œuvre d'un patron de conception! C'était un exemple d'utilisation du patron *fabrique*. Ne vous inquiétez pas, nous en discuterons en détail plus tard.

Préférer la composition à l'héritage

L'héritage est la technique la plus simple et intuitive pour réutiliser le code des autres classes. Vous avez deux classes avec le même code. Créez une classe de base commune pour ces deux classes, et déplacez-y le code similaire. Simple comme bonjour !

Malheureusement, l'héritage a aussi ses inconvénients que vous ne détecterez qu'une fois votre programme submergé par d'innombrables classes, et il deviendra difficile d'y modifier quoi que ce soit. Voici une liste de ces problèmes :

- **Une sous-classe ne peut pas réduire la taille de l'interface de la classe mère.** Vous devez implémenter toutes les méthodes abstraites de la classe mère, même si vous ne les utilisez pas.
- **Lorsque vous redéfinissez des méthodes, vous devez vous assurer que leur comportement est compatible avec la classe de base.** C'est un détail plutôt important, car les objets de la sous-classe vont potentiellement être passés à du code qui s'attend à recevoir des objets de la classe mère, ce qui peut mener à des plantages.
- **L'héritage détruit l'encapsulation de la classe mère,** car les détails internes de la classe parent deviennent visibles pour la sous-classe. Il est possible de se retrouver dans la situation inverse, où un développeur expose les détails d'une sous-classe à une classe mère afin de faciliter les futures évolutions.

- **Les sous-classes sont fortement couplées aux classes mères.** La moindre modification de la classe mère peut impacter le fonctionnement de ses sous-classes.
- **Exploiter la réutilisation du code à l'aide de l'héritage peut mener à créer des hiérarchies d'héritage parallèles.** L'héritage ne se fait généralement que sur une seule dimension. Mais dès que vous vous retrouvez avec deux dimensions ou plus, vous devez créer de nombreuses combinaisons de classes, résultant en une hiérarchie de classes ridiculement grande.

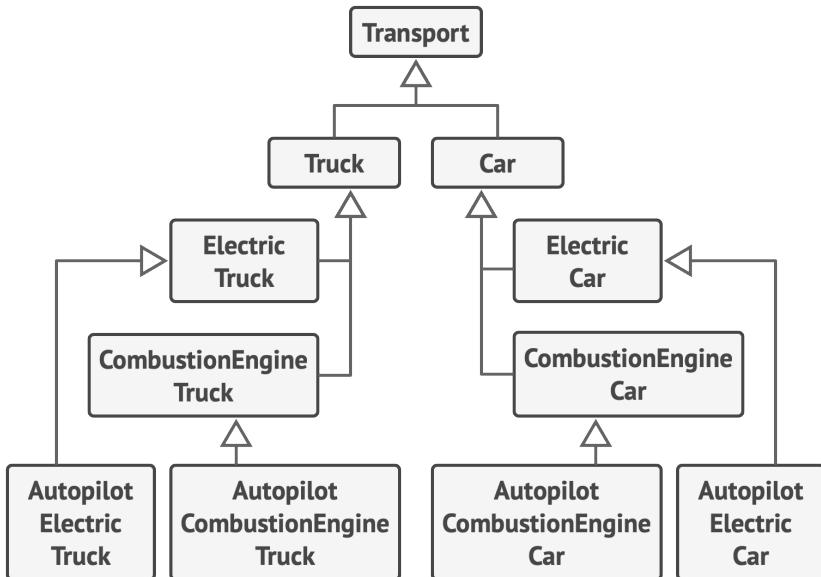
L'héritage a une alternative appelée *composition*. Le premier représente le « est un » de la relation entre classes (une voiture *est un* moyen de transport), et la composition représente le « a un » de la relation (une voiture *a un* moteur).

J'en profite pour préciser que ce principe peut également s'appliquer à l'agrégation - une variante plus souple de la composition, dans laquelle un objet peut avoir une référence sur un autre, mais ne gère pas son cycle de vie. Voici un exemple : une voiture *a un* conducteur, mais il peut utiliser une autre voiture ou marcher *sans la voiture*.

Exemple

Imaginez que vous devez créer une application de catalogue pour un constructeur automobile. La société fabrique des voitures et des camions qui fonctionnent à l'électricité ou avec du

carburant, et tous les modèles se conduisent manuellement ou avec un pilote automatique.

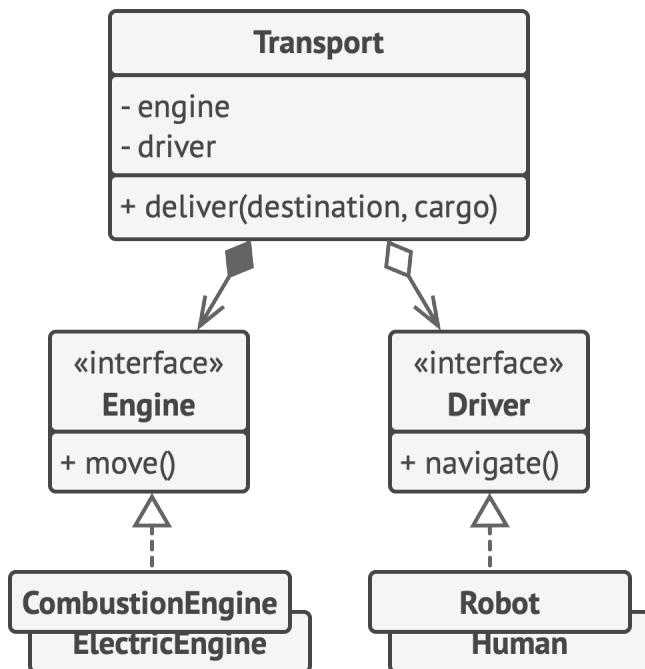


HÉRITAGE : extension de la classe dans plusieurs dimensions (type de transport x type de moteur x type de conduite) qui peut mener à une explosion combinatoire de sous-classes.

Comme vous pouvez le constater, chaque paramètre supplémentaire implique de multiplier le nombre de sous-classes. On retrouve beaucoup de code dupliqué dans les sous-classes, car elles ne peuvent pas étendre 2 classes en même temps.

Vous pouvez résoudre ce problème grâce à la composition. Plutôt que de confier l'implémentation du comportement aux voitures, elles peuvent le déléguer aux autres objets.

Cette technique vous permet même de modifier le comportement lors de l'exécution. Vous pouvez par exemple remplacer un objet moteur associé à un objet voiture, en assignant un moteur différent à la voiture.



COMPOSITION : les différentes « dimensions » de la fonctionnalité sont extraites dans leurs propres hiérarchies de classes.

Cette structure de classe ressemble de près au patron de conception *stratégie* que nous aborderons plus tard.

Principes SOLID

Maintenant que vous connaissez les principes de base de la conception, étudions les cinq qui font partie des principes SOLID. Robert Martin les a présentés dans son livre *Agile Software Development, Principles, Patterns, and Practices*¹.

SOLID est une mnémonique pour cinq principes de conception qui rendent les logiciels plus facilement compréhensibles, flexibles et maintenables.

Comme tout dans la vie, utiliser ces principes n'importe comment peut causer plus de mal que de bien. Appliquer ces principes à l'architecture d'un programme peut le rendre plus compliqué qu'il ne devrait l'être. Je ne pense pas qu'il existe un logiciel à succès dans lequel tous ces principes sont appliqués en même temps. Il faut essayer de s'en rapprocher au maximum, mais toujours rester pragmatique et ne pas prendre tout ce qui est écrit ici comme parole d'évangile.

1. Agile Software Development, Principles, Patterns, and Practices :
<https://refactoring.guru/fr/principles-book>

S

Principe de responsabilité unique Single Responsibility Principle

Une classe ne devrait être modifiée que pour une seule raison.

Essayez de faire en sorte qu'une classe ne soit responsable que d'une partie d'une fonctionnalité de votre logiciel, et encapsulez (vous pouvez aussi dire *cachez à l'intérieur*) cette responsabilité entièrement dans la classe.

L'objectif principal de ce principe est de diminuer la complexité. Vous n'avez pas besoin d'inventer un concept sophistiqué pour un programme doté d'environ 200 lignes de code. Écrivez une dizaine de jolies méthodes et tout ira pour le mieux.

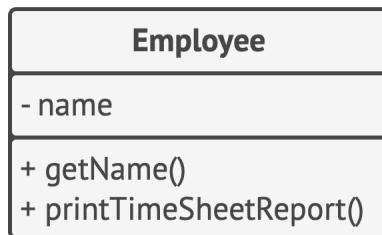
Les vrais problèmes font surface avec l'évolution et les modifications de votre programme. Au bout d'un moment, certaines classes deviendront tellement grosses que vous ne vous souviendrez plus de leurs détails. Naviguer dans le code va devenir fastidieux et vous devrez parcourir des classes entières ou même tout le programme pour trouver ce que vous cherchez. Le nombre d'entités dans le programme va complètement submerger votre cerveau, et vous sentirez que vous perdez le contrôle du code.

Mais ce n'est pas tout : si une classe s'occupe de trop de choses à la fois, vous devrez la modifier à la moindre évolution. Vous risquez ainsi de provoquer des bugs dans des fonctionnalités que vous ne vouliez même pas modifier.

Si vous éprouvez des difficultés à vous concentrer sur des aspects spécifiques du programme, souvenez-vous du principe de responsabilité unique et vérifiez s'il est grand temps de découper certaines classes en plusieurs parties.

Exemple

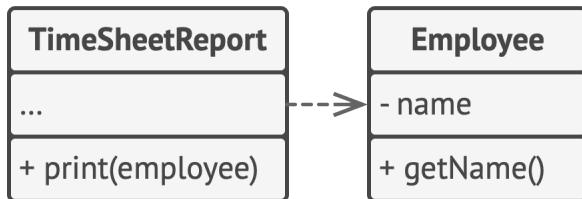
La classe `Employé` a plusieurs raisons d'être modifiée. La première est directement liée à la fonction principale de la classe : gérer les données des employés. Mais il y a une autre raison : le format du rapport des pointages est susceptible de changer, ce qui vous demande de modifier le code de la classe.



AVANT : la classe contient plusieurs comportements différents.

Résolvez le problème en déplaçant le comportement concernant l'impression des rapports des pointages dans une classe

séparée. Ce changement vous permet de déplacer tout ce qui concerne les rapports dans cette nouvelle classe.



APRÈS : le comportement supplémentaire se retrouve dans sa propre classe.

O Principe ouvert/fermé pen/Closed Principle

Les classes doivent être ouvertes à l'extension, mais fermées à la modification.

L'idée principale de ce principe est d'éviter de créer des bugs dans du code existant lorsque vous ajoutez de nouvelles fonctionnalités.

Une classe est *ouverte* si vous pouvez l'étendre, créer une sous-classe ou faire n'importe quoi d'autre avec (ajouter de nouveaux attributs ou méthodes, redéfinir le comportement de base, etc.). Certains langages de programmation vous permettent de limiter l'extension d'une classe à l'aide de mots-clés, comme `final`. Avec ceci, la classe n'est plus ouverte. Une classe est *fermée* (on peut également dire *complète*) si elle est prête à 100 % à être utilisée par d'autres classes (son interface est clairement définie et ne sera plus modifiée dans le futur).

Lorsque j'ai découvert ce principe, je ne l'avais pas très bien compris, car les mots *ouvert* et *fermé* semblent mutuellement exclusifs. Mais pour ce principe, une classe peut être à la fois ouverte (à l'extension) et fermée (à la modification).

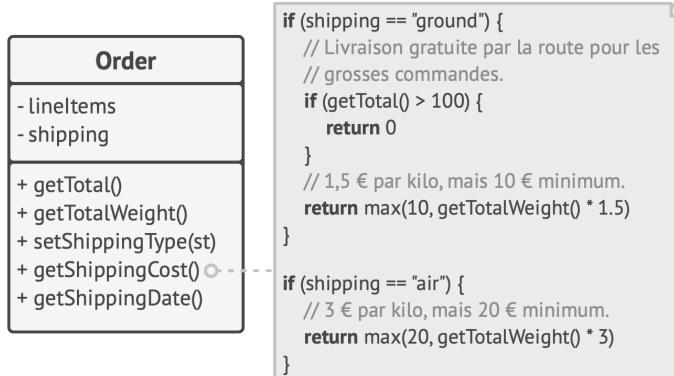
Si une classe a déjà été développée, testée, validée et incluse dans un framework ou utilisée dans une application, toucher au code peut être risqué. Plutôt que de modifier la classe directement, vous pouvez créer une sous-classe et redéfinir les parties de classe ori-

ginale dont vous voulez modifier le comportement. Non seulement vous parviendrez au résultat attendu, mais en plus vous ne causerez pas de problème pour les clients existants de la classe originale.

Ce principe n'est pas forcément valable pour toutes les raisons que vous pourriez avoir de modifier une classe : si vous trouvez un bug dans une classe, corrigez-le. Ne créez pas une sous-classe juste pour cela ! Une classe enfant ne doit pas être responsable des problèmes de ses parents.

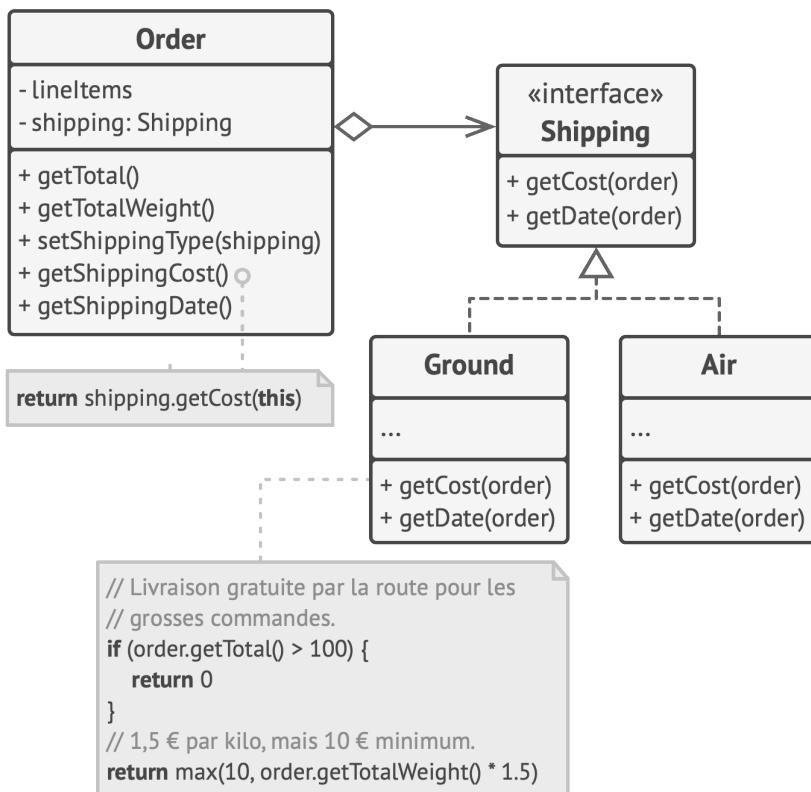
Exemple

Vous travaillez sur une application de boutique en ligne avec une classe `Commande` qui calcule les coûts d'expédition, et toutes les méthodes de ces coûts sont codées en dur dans la classe. Pour ajouter une nouvelle méthode d'expédition, vous allez devoir modifier la classe `Commande` et vous risquez de provoquer des plantages.



AVANT : vous avez modifié la classe `Commande` chaque fois que vous avez ajouté une méthode d'expédition à l'application.

Vous pouvez résoudre ce problème en utilisant le patron de conception *stratégie*. Commencez par extraire les méthodes d'expédition et les mettre dans des classes séparées qui implémentent la même interface.



APRÈS : ajouter une nouvelle méthode d'expédition ne vous oblige pas à effectuer des modifications dans les classes existantes.

À présent, si vous voulez implémenter de nouvelles méthodes d'expédition, vous pouvez dériver une nouvelle classe de l'interface `Expédition` sans avoir à toucher au code de la classe `Commande`. Le code client de la classe `Commande` associera les

commandes à un objet expédition de la nouvelle classe lorsque l'utilisateur choisit sa méthode d'expédition dans l'interface.

En bonus, et en concordance avec le *principe de responsabilité unique*, cette solution vous permet de déplacer le calcul du délai de livraison dans des classes plus pertinentes.

L

Principe de substitution de Liskov Liskov Substitution Principle¹

Lorsque vous étendez une classe, rappelez-vous que vous devez être en mesure de passer des objets de la sous-classe à la place des objets de la classe mère sans faire planter le code.

Cela signifie que les sous-classes restent compatibles avec le comportement de la classe mère. Lorsque vous redéfinissez une méthode, étendez le comportement de base plutôt que de la remplacer complètement avec autre chose.

Le principe de substitution est un ensemble de vérifications qui aide à prédire si une classe va rester compatible avec du code qui fonctionnait auparavant avec les objets de la classe mère. Ce concept est critique lorsque vous développez des bibliothèques et des frameworks parce que vos classes vont être utilisées par d'autres personnes, et vous ne pourrez pas accéder directement à leur code, ni le modifier.

Contrairement aux autres principes de conception qui sont ouverts à l'interprétation, le principe de substitution impose des prérequis aux sous-classes, et plus spécifiquement à leurs méthodes. Regardons cette liste en détail.

-
1. Ce principe a été baptisé d'après Barbara Liskov, qui l'a défini en 1987 dans ses travaux *Data abstraction and hierarchy* : <https://refactoring.guru/liskov/dah>

- **Les types des paramètres de la méthode d'une sous-classe doivent correspondre ou être plus abstraits que les types des paramètres dans la méthode de la classe mère.** Cela vous perturbe ? Prenons un exemple.
 - Prenons une classe avec une méthode qui nourrit les chats : `nourrir(Chat c)`. Le code client passe toujours des chats à cette méthode.
 - **Bon :** Prenons l'hypothèse où vous avez créé une sous-classe qui a redéfini la méthode pour nourrir n'importe quel type d'animal (une classe mère de chat) `nourrir(Animal c)`. Si vous passez un objet de la sous-classe à la place d'un objet de la classe mère au code client, tout devrait se passer pour le mieux. La méthode peut nourrir tous les animaux, et elle peut aussi nourrir les chats passés par le client.
 - **Mauvais :** Vous avez créé une autre sous-classe et limité la méthode pour qu'elle ne nourrisse que les chats Bengal (une sous-classe de chat). `nourrir(ChatBengal c)`. Que va-t-il arriver au code client si vous l'associez à ce genre d'objet plutôt qu'avec la classe originale ? Puisque cette méthode ne peut nourrir qu'une race spécifique de chats, elle ne prendra pas en charge les chats génériques passés par le client, faisant planter la fonctionnalité.
- **Le type de retour dans une méthode d'une sous-classe doit correspondre ou être un sous-type du type de retour de la méthode de la classe mère.** Comme vous pouvez le constater, les pré-

quis pour les types de retour d'une méthode sont l'inverse des types des paramètres.

- Disons que vous avez une classe avec une méthode `acheterChat(): Chat`. Le code client s'attend à recevoir n'importe quel chat lorsqu'il appelle cette méthode.
- **Bon** : Une sous-classe redéfinit la méthode : `acheterChat(): ChatBengal`. Le client récupère un chat Bengal, qui est un chat, donc tout va bien.
- **Mauvais** : Une sous-classe redéfinit la méthode : `acheterChat(): Animal`. À présent le code plante puisqu'il reçoit un animal générique inconnu (un alligator ? Un ours ?) qui ne rentre pas dans la structure conçue pour un chat.

Un autre contre-exemple nous vient du monde des systèmes de typage statiques : la méthode de base retourne une chaîne de caractères, mais la méthode redéfinie renvoie un nombre.

- **Une méthode dans une sous-classe ne devrait pas lever les types d'exceptions que la méthode de base n'est pas censée lever.** En d'autres mots, les types d'exceptions doivent correspondre à ceux que la méthode de base est déjà capable de lever ou en être des *sous-types*. Cette règle vient du fait que les blocs `try-catch` dans le code client ciblent les types spécifiques d'exceptions que la méthode de base est susceptible de lever. Par conséquent, une exception inattendue pourrait se

glisser à travers les lignes de défense du code client et faire planter l'application.

Dans la majorité des langages de programmation modernes et plus particulièrement dans ceux qui sont typés statiques (Java, C# et d'autres), ces règles sont gravées dans le langage. Vous ne pourrez pas compiler un programme qui ne les respecte pas.

- **Une sous-classe ne devrait pas renforcer des préconditions.**
Prenons une méthode de base qui a un paramètre de type `int`. Si une sous-classe redéfinit cette méthode et demande que la valeur du paramètre passé à la méthode soit positive (en levant une exception si la valeur est négative), cela renforce les préconditions. Le code client fonctionnait très bien lorsqu'il passait des nombres négatifs à la méthode, mais maintenant il plante si on lui donne un objet de cette sous-classe.
- **Une sous-classe ne devrait pas affaiblir des postconditions.**
Prenons une classe avec une méthode qui fonctionne avec une base de données. Une méthode de cette classe est censée toujours fermer les connexions ouvertes à la base de données lorsqu'elle renvoie une valeur.

Vous avez créé une sous-classe et l'avez modifiée afin qu'elle laisse les connexions ouvertes pour pouvoir les réutiliser, mais le client n'a probablement aucune idée de vos intentions. Il s'attend à ce que toutes les méthodes ferment les connexions,

alors il pourrait très bien décider de fermer le programme juste après avoir appelé la méthode, polluant le système avec une connexion fantôme à la base de données.

- **Les invariants d'une classe mère doivent être préservés.** C'est probablement la règle la moins formalisée de toutes. Les *invariants* sont des conditions qui permettent de donner tout son sens à un objet. Par exemple, les invariants d'un chat sont qu'ils ont quatre pattes, une queue, la possibilité de miauler, etc. Ce qui n'est pas très clair avec les invariants, c'est qu'ils peuvent être définis explicitement sous la forme de contrats avec l'interface ou d'un ensemble d'assertions à l'intérieur des méthodes, mais ils peuvent aussi être sous-entendus via certains tests unitaires ou via les caractéristiques présumées ou anticipées par le code client.

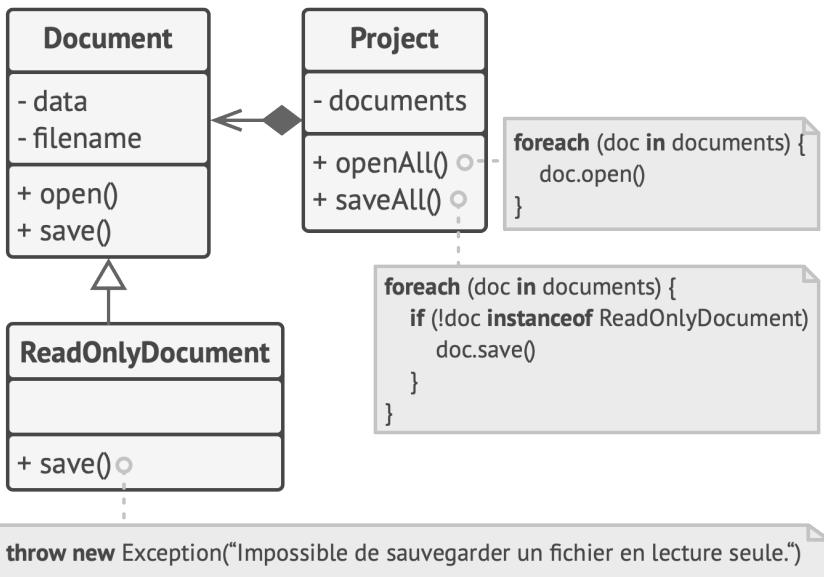
La règle sur les invariants est la plus facile à enfreindre, car vous pourriez omettre certains des invariants d'une classe complexe ou mal les comprendre. Par conséquent, la manière la plus sécurisée d'étendre une classe est de créer de nouveaux attributs et méthodes, et de ne pas toucher aux membres existants de sa classe mère. Bien sûr, ce n'est pas toujours réalisable dans la vraie vie.

- **Une sous-classe ne doit jamais modifier les valeurs des attributs privés d'une classe mère.** *Quoi? Comment serait-ce possible?* Il se trouve que certains langages de programmation vous laissent accéder aux membres privés d'une classe à l'aide

de mécanismes de réflexion. D'autres langages (Python, JavaScript) ne protègent pas du tout les membres privés.

Exemple

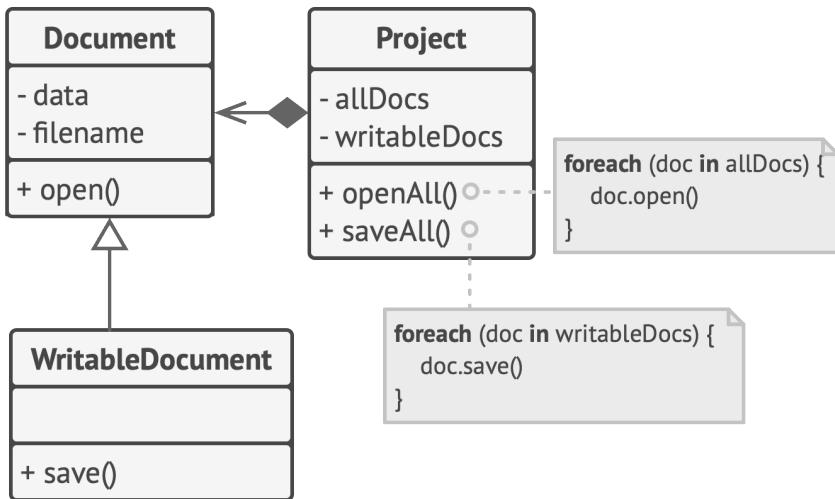
Regardons ensemble l'exemple d'une hiérarchie de classes de documents qui enfreignent le principe de substitution.



AVANT : cela n'a aucun sens de vouloir sauvegarder un document en lecture seule, la sous-classe tente donc de résoudre cette situation en réinitialisant le comportement de base de la méthode redéfinie.

La méthode `sauvegarder` de la sous-classe `DocumentEnLectureSeule` lève une exception si elle est appelée. La méthode de base ne possède pas cette restriction. Cela signifie que le code client va planter si nous ne vérifions pas le type du document avant de le sauvegarder.

Ce code enfreint le principe ouvert/fermé, car le code client devient dépendant des classes concrètes des documents. Si vous voulez introduire une nouvelle sous-classe de documents, vous devez modifier le code client pour la prendre en charge.



APRÈS : le problème est résolu en faisant de la classe documentEnLectureSeule, la classe de base de la hiérarchie.

Vous pouvez résoudre ce problème en repensant la conception de la hiérarchie de classes : une sous-classe peut étendre le comportement d'une classe mère. Le document en lecture seule devient donc la classe de base de la hiérarchie. Le document ouvert en écriture est maintenant une sous-classe qui étend la classe de base et ajoute le comportement de sauvegarde.

I Principe de ségrégation des interfaces **Interface Segregation Principle**

Les clients ne devraient pas être forcés à dépendre de méthodes qu'ils n'utilisent pas.

Essayez de rendre vos interfaces aussi étroites que possible afin de ne pas obliger les classes des clients à implémenter des comportements dont elles n'ont pas besoin.

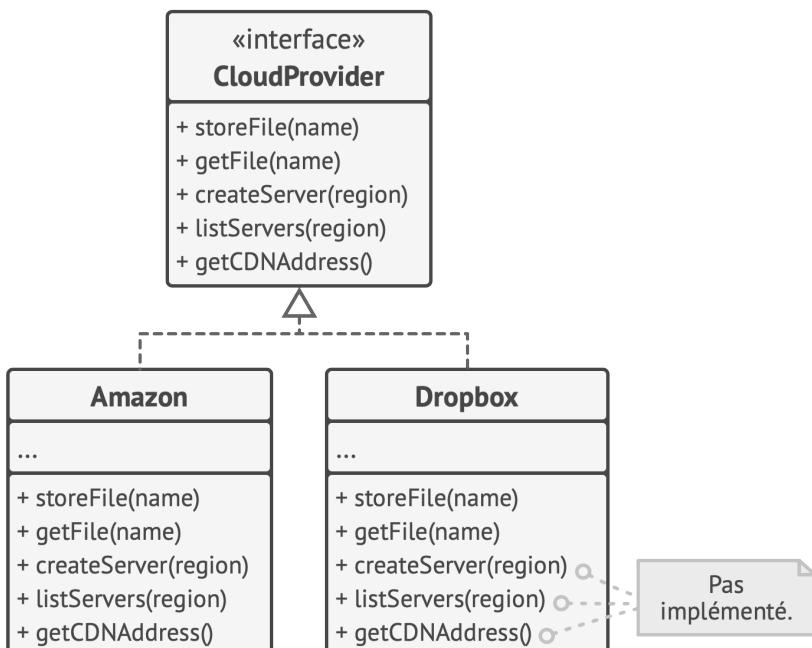
Selon le principe de ségrégation des interfaces, vous devriez découper vos « grosses » interfaces pour les rendre plus précises et spécifiques. Les clients ne doivent implémenter que les méthodes dont ils ont vraiment besoin. Sinon, une modification apportée dans une « grosse » interface va même faire planter les clients qui n'utilisent pas les méthodes modifiées.

L'héritage permet à une classe de n'avoir qu'une seule classe mère, mais elle ne limite pas le nombre d'interfaces qu'une même classe peut implémenter en même temps. Il n'y a donc nul besoin d'entasser des tonnes de méthodes qui n'ont aucun rapport dans une même interface. Découpez-les en plusieurs interfaces plus spécifiques : une classe pourra toutes les implémenter si besoin, mais certaines classes se porteront très bien avec une seule interface.

Exemple

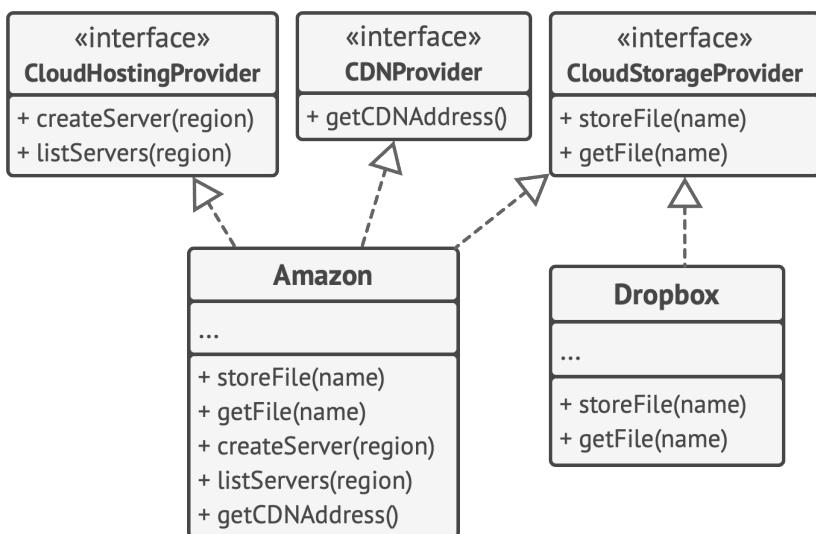
Imaginez-vous en train de créer une bibliothèque pour faciliter l'intégration d'applications de divers fournisseurs de cloud. Dans la première version, elle ne prenait en charge que le cloud d'Amazon (tous ses services et fonctionnalités).

À cette époque, vous pensiez que tous les fournisseurs de clouds utilisaient globalement les mêmes fonctionnalités qu'Amazon. Mais lorsque vous vous êtes attaqués à la prise en charge des autres fournisseurs, vous vous êtes rendu compte que les interfaces de la bibliothèque étaient bien trop larges. Certaines méthodes décrivent des fonctionnalités que les autres fournisseurs ne proposent pas.



AVANT : tous les clients ne remplissent pas les prérequis de cette énorme interface.

Vous pouvez toujours implémenter ces méthodes et y positionner des bouchons, mais ce n'est pas une très jolie solution. La meilleure approche serait de découper cette interface en plusieurs morceaux. Les classes qui implémente l'interface originale peuvent dorénavant implémenter plusieurs interfaces plus spécifiques. Les autres classes peuvent implémenter seulement les interfaces dotées des méthodes qui les intéressent.



APRÈS : une interface géante est découpée en petites interfaces plus spécifiques.

Comme pour les autres principes, le risque est d'aller trop loin. Ne divisez pas une interface qui est déjà très spécifique. Rappelez-vous que plus vous créez d'interfaces, plus votre code devient complexe. Gardez un certain équilibre.

D

Principe d'inversion des dépendances Dependency Inversion Principle

Les classes de haut niveau ne devraient pas dépendre des classes de bas niveau. Elles devraient dépendre toutes les deux d'abstractions. Une abstraction ne doit pas dépendre des détails. Les détails doivent dépendre de l'abstraction.

Généralement, lorsque vous concevez un logiciel, vous pouvez faire la distinction entre deux niveaux de classes.

- Les **classes de bas niveau** implémentent des traitements basiques comme utiliser le disque, transférer des données sur un réseau, se connecter à une base de données, etc.
- Les **classes de haut niveau** contiennent la logique métier qui indique aux classes de bas niveau ce qu'elles doivent faire.

Parfois les développeurs commencent par concevoir les classes de bas niveau et passent ensuite aux classes de haut niveau. C'est quelque chose d'assez commun lorsque vous développez le prototype d'un nouveau système : vous ne savez pas trop ce que vous allez pouvoir faire à haut niveau, car le code de bas niveau n'est pas très clair ou pas encore implémenté. Avec une telle approche, les classes de la logique métier ont tendance à devenir dépendantes des classes primitives de bas niveau.

Le principe d'inversion des dépendances propose de changer le sens de cette dépendance.

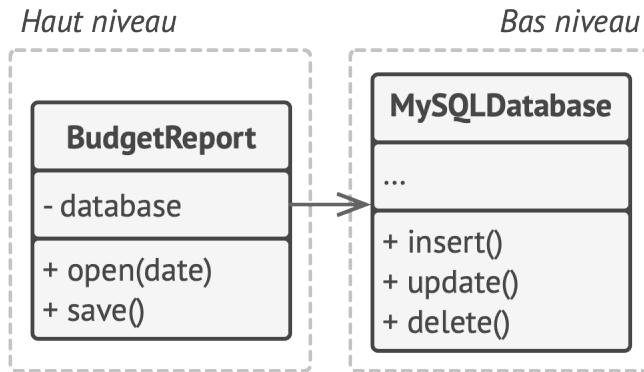
1. Pour commencer, vous devez décrire les interfaces pour les traitements de bas niveau dont les classes de haut niveau vont avoir besoin, de préférence en utilisant les termes de la logique métier. Par exemple, la logique métier doit appeler une méthode `ouvrirRapport(fichier)` plutôt qu'une suite de méthodes `ouvrirFichier(x)`, `lireOctets(n)`, `fermerFichier(x)`. Ces interfaces font partie du haut niveau.
2. Maintenant, vous pouvez rendre les classes de haut niveau dépendantes de ces interfaces, plutôt que de les rendre dépendantes des classes concrètes de bas niveau. Cette dépendance sera bien plus faible que celle d'origine.
3. Une fois que les classes de bas niveau implémentent ces interfaces, elles deviennent dépendantes de la logique métier, inversant la direction de la dépendance originale.

Le principe d'inversion des dépendances est souvent utilisé en corrélation avec le *principe ouvert/fermé* : vous pouvez étendre les classes de bas niveau pour utiliser différentes classes de la logique métier sans avoir à modifier les classes existantes.

Exemple

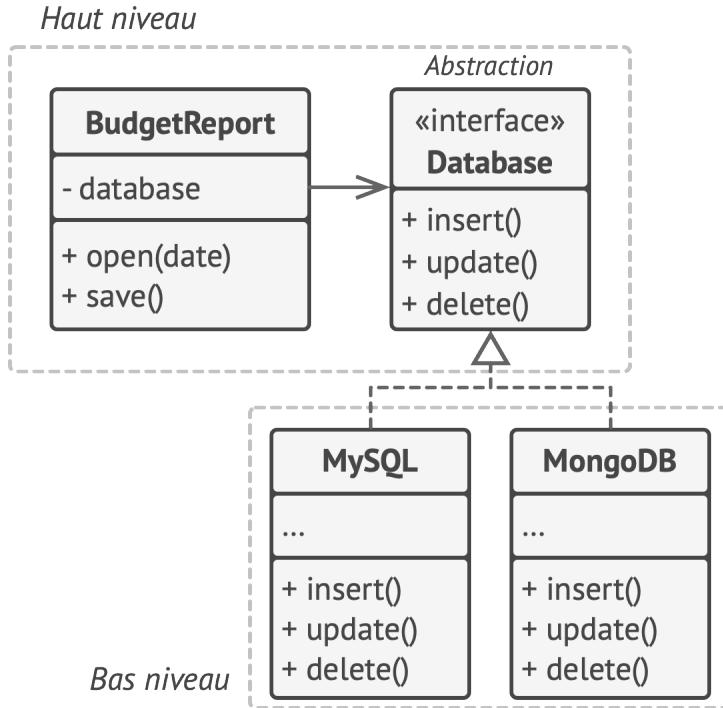
Dans cet exemple, la classe de haut niveau qui établit les rapports du budget utilise une classe de base de données de bas niveau pour lire et écrire ses données. Ce qui signifie que la modification de la classe de bas niveau - par exemple la livrai-

son d'une nouvelle version du serveur de la base de données - va impacter la classe de haut niveau, qui n'est pas censée s'occuper des détails du stockage des données.



AVANT : une classe de haut niveau dépend d'une classe de bas niveau.

Vous pouvez régler ce problème en créant une interface de haut niveau qui décrit les traitements de lecture/écriture et faire en sorte que la classe des rapports utilise l'interface plutôt que la classe de bas niveau. Vous pouvez ensuite modifier ou étendre la classe de bas niveau originale et la faire implémenter la nouvelle interface de lecture/écriture déclarée par la logique métier.



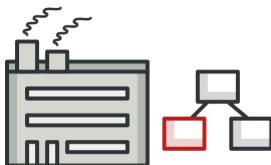
APRÈS : les classes de bas niveau dépendent d'une abstraction de haut niveau.

Par conséquent, la direction de la dépendance originale a été inversée : les classes de bas niveau sont maintenant dépendantes des abstractions de haut niveau.

CATALOGUE DES PATRONS DE CONCEPTION

Patrons de création

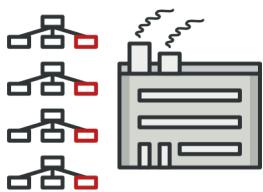
Les patrons de création fournissent des mécanismes de création d'objets qui augmentent la flexibilité et la réutilisation du code.



Fabrique

Factory Method

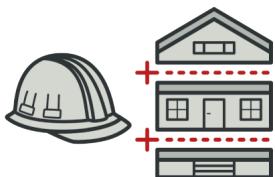
Définit une interface pour la création d'objets dans une classe mère, mais délègue aux sous-classes le choix des types d'objets à créer.



Fabrique abstraite

Abstract Factory

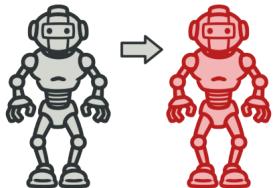
Permet de créer des familles d'objets apparentés sans préciser leur classe concrète.



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

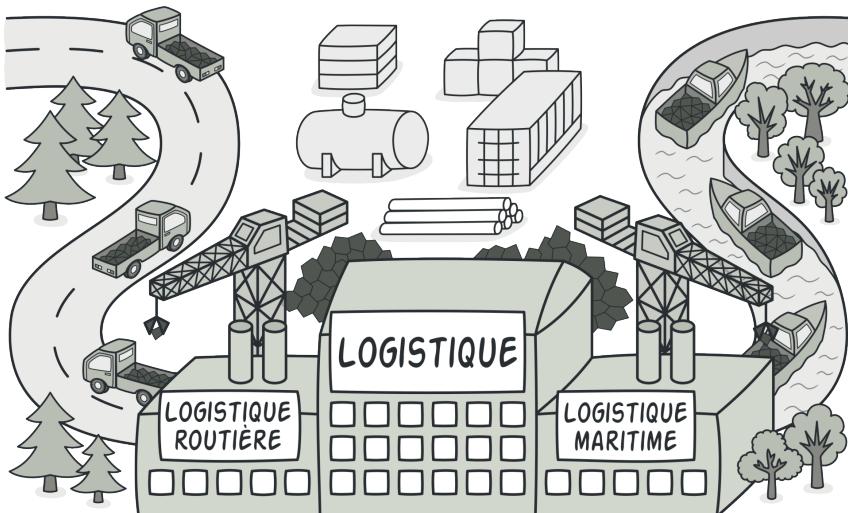
Permet de créer de nouveaux objets à partir d'objets existants sans rendre le code dépendant de leur classe.



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

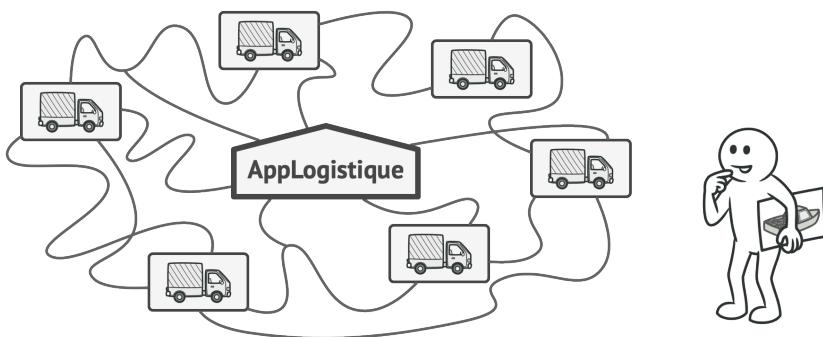
Alias : Constructeur virtuel, Factory Method

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.



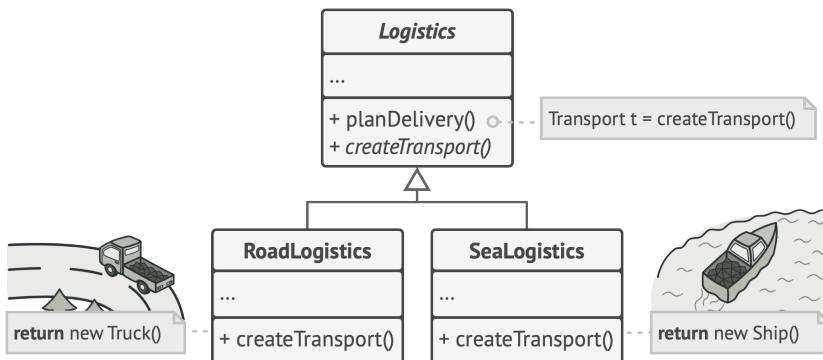
L'ajout d'une nouvelle classe au programme ne s'avère pas si simple que cela si le reste du code est déjà couplé aux classes existantes.

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*.

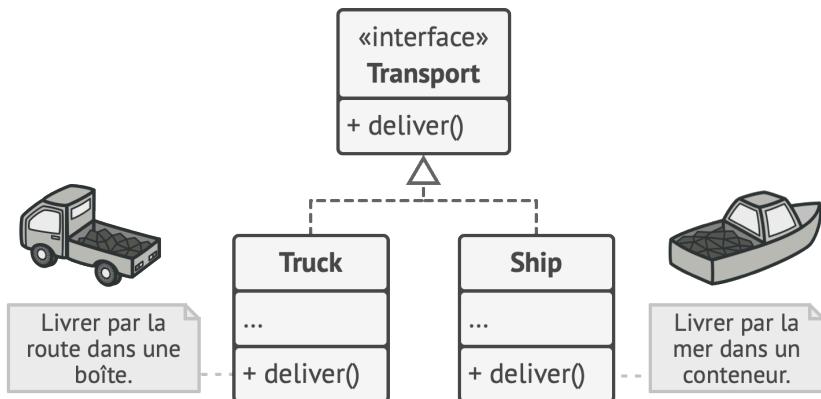


Les sous-classes peuvent modifier les classes des objets retournés par la méthode fabrique.

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

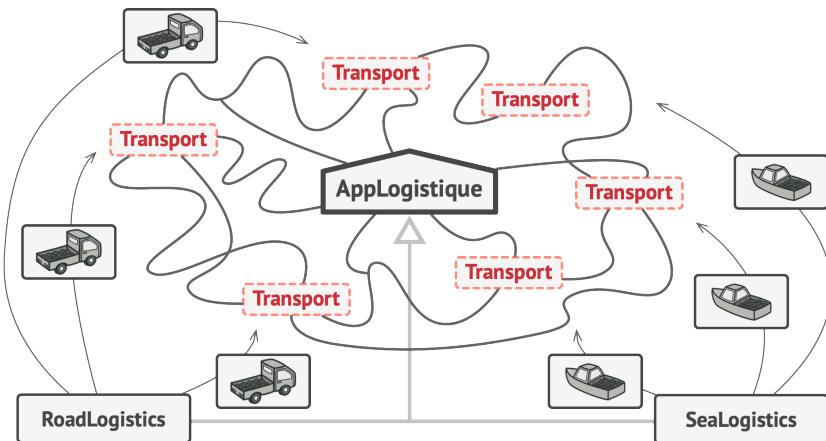
la méthode fabrique dans la sous-classe et changer la classe des produits créés par la méthode.

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.

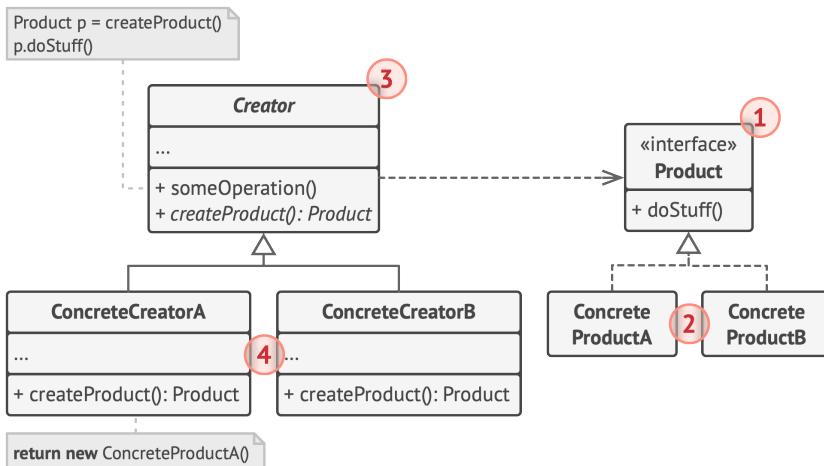
Par exemple, les classes `Camion` et `Bateau` doivent toutes les deux implémenter l'interface `Transport`, qui déclare une méthode `livrer`. Chaque classe implémente cette méthode à sa façon : les camions livrent par la route et les bateaux livrent par la mer. La méthode fabrique de la classe `LogistiqueRoute`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

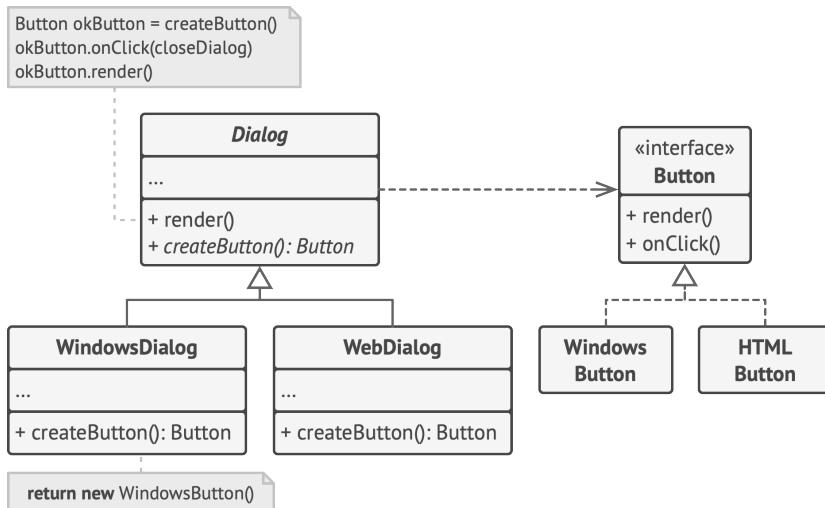
Il faut bien comprendre que malgré son nom, la création de produits n'est **pas** la responsabilité principale du créateur. La classe créateur a en général déjà un fonctionnement propre lié à la nature de ses produits. La fabrique aide à découpler cette logique des produits concrets. C'est un peu comme 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 pourrons 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()
```

83 `dialog.render()`

💡 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 CourierAérien n'utilise que des Avions et la classe CourierTerrestre peut utiliser à la fois des Camions et des Trains. Vous pouvez créer une nouvelle sous-classe (CourierFerroviaire 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 CourierTerrestre pour désigner le type de produit qu'elle veut recevoir.

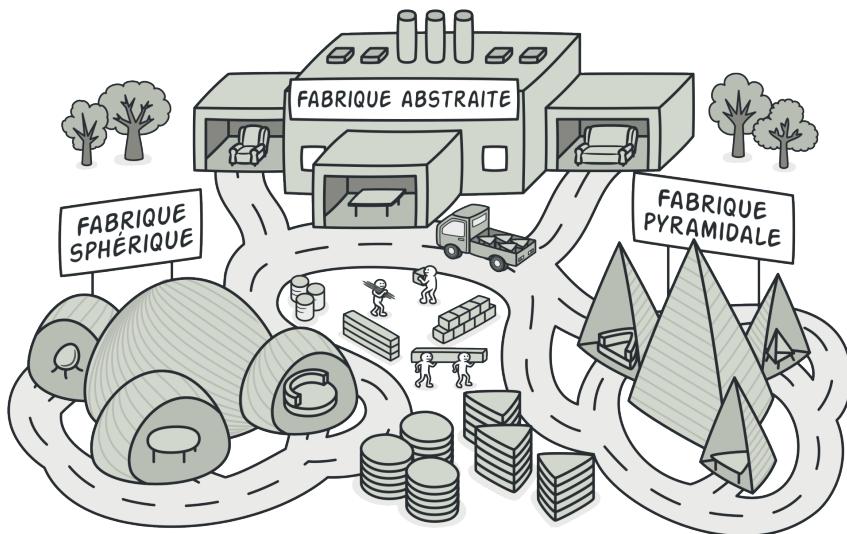
6. Si après tous ces changements la méthode fabrique de base est devenue complètement vide, vous pouvez la rendre abstraite. S'il reste encore quelques lignes, vous pouvez y laisser un comportement par défaut.

ΔιΔ 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*.



FABRIQUE ABSTRAITE

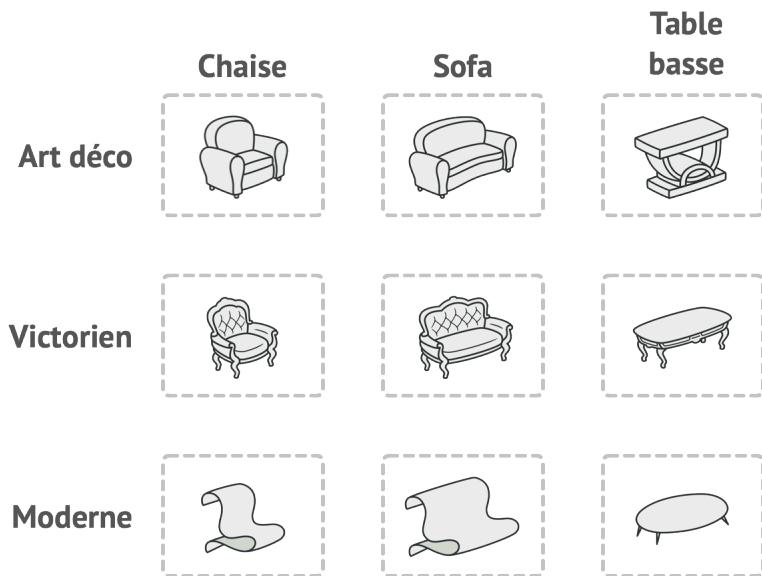
Alias : Abstract Factory

Fabrique abstraite est un patron de conception qui permet de créer des familles d'objets apparentés sans préciser leur classe concrète.

(:() Problème

Imaginons la création d'un simulateur pour un magasin de meubles. Votre code contient les classes suivantes :

1. Une famille de produits appartenant à un même thème : `Chaise` + `Sofa` + `TableBasse`.
2. Plusieurs variantes de cette famille. Par exemple, les produits `Chaise` + `Sofa` + `TableBasse` sont disponibles dans les variantes suivantes : `Moderne`, `Victorien`, `ArtDéco`.



Familles de produits et leurs variantes.

Vous devez trouver une solution pour créer des objets individuels (des meubles) et les faire correspondre à d'autres objets

de la même famille. Les clients sont agacés lorsqu'ils reçoivent des meubles qui ne se marient pas.



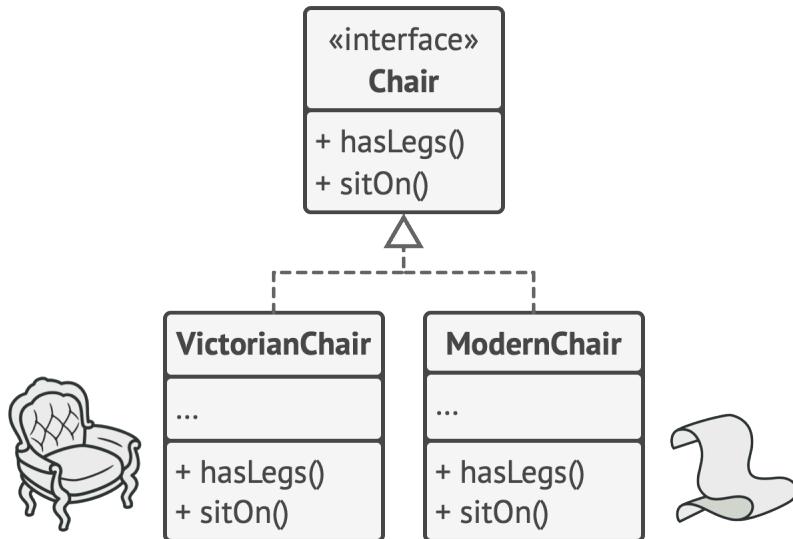
Un sofa moderne ne se marie pas avec des chaises de style victorien.

De plus, vous n'avez pas envie de réécrire votre code chaque fois que vous ajoutez de nouvelles familles de produits à votre programme. Les vendeurs de meubles alimentent régulièrement leurs catalogues et il n'est pas envisageable de restructurer le code à chaque mise à jour.

😊 Solution

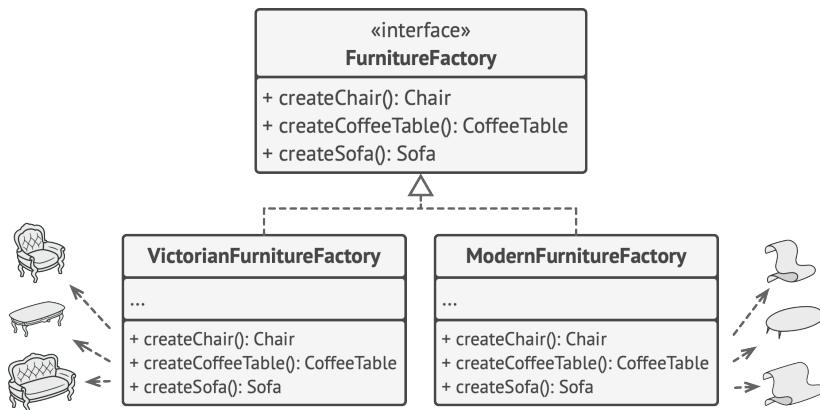
La première chose que propose la fabrique abstraite est de déclarer explicitement des interfaces pour chaque produit de la famille de produits (dans notre cas : chaise, sofa, table basse). Toutes les autres variantes de produits peuvent ensuite se servir de ces interfaces. Par exemple, toutes les variantes de chaises peuvent implémenter l'interface `Chaise` ;

toutes les variantes de tables basses peuvent implémenter `TableBasse`, etc.



Toutes les variantes du même objet peuvent être rangées dans la même hiérarchie de classe.

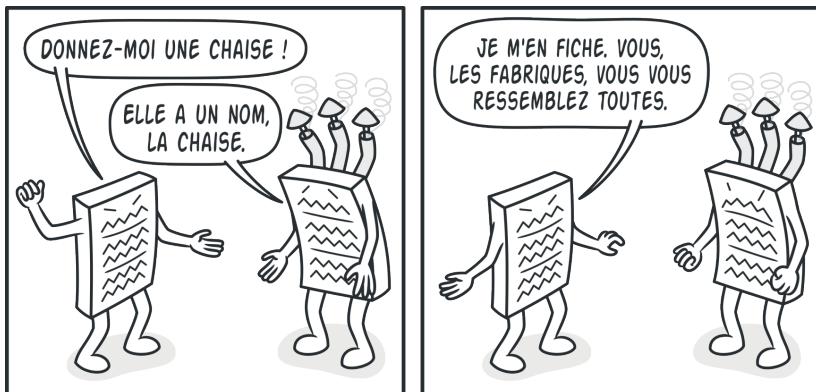
La prochaine étape est de déclarer la *fabrique abstraite* – une interface armée d'une liste de méthodes de création pour toutes les familles de produits (par exemple : `créerChaise`, `créerSofa` et `créerTableBasse`). Ces méthodes doivent renvoyer tous les types de produits **abstraits** des interfaces que nous avons créées précédemment : `Chaise`, `Sofa`, `TableBasse`, etc.



Chaque fabrique concrète correspond à une variante d'un produit spécifique.

Mais que deviennent les variantes des produits ? Pour chaque variante d'une famille de produits, nous créons une classe fabrique qui implémente l'interface `FabriqueAbstraite`. Une fabrique est une classe qui retourne un certain type de produits. Par exemple, la `FabriqueMeubleModerne` peut uniquement créer des `ChaiseModerne`, des `SofaModerne` et des `TableBasseModerne`.

Le code client travaille simultanément avec les interfaces abstraites respectives des fabriques et des produits. Nous pouvons ainsi changer le type de fabrique passé au code client et la variante de produit qu'il reçoit, sans avoir à le modifier.

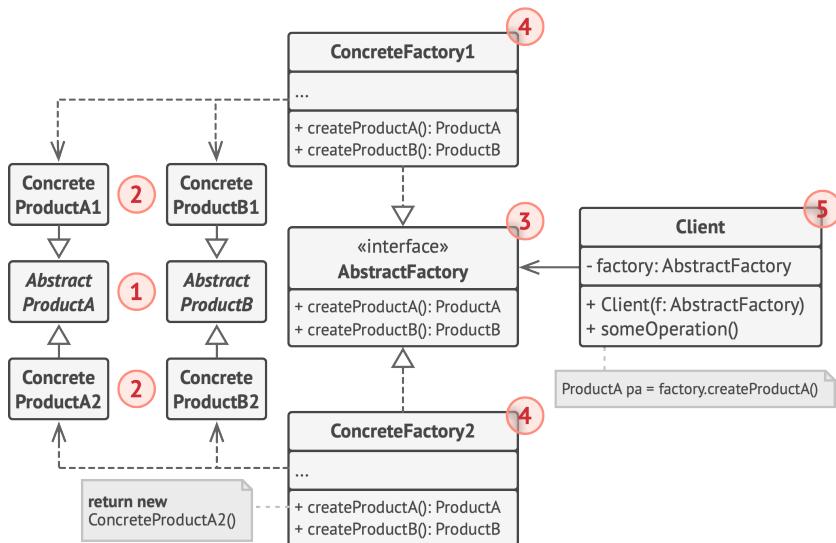


Le client n'a pas besoin de connaître la classe concrète des fabriques qu'il utilise.

Imaginons un cas où le client désire une fabrique qui peut produire une chaise. Le client n'a pas à se préoccuper de la classe de la fabrique ni du type de chaise qu'il va obtenir. Il doit traiter les chaises de la même manière, que ce soit un modèle de style moderne ou victorien, en utilisant l'interface abstraite `Chaise`. Grâce à cette approche, le client ne sait qu'une seule chose à propos de la chaise, c'est qu'elle implémente la méthode `sasseoir`. De plus, quelle que soit la variante de la chaise renvoyée, elle correspondra systématiquement au type de sofa ou de table basse produit par le même objet fabrique.

Il ne reste plus qu'un point à éclaircir : si le client n'est lié qu'aux interfaces abstraites, comment les objets de la fabrique sont-ils créés ? En général, l'application crée un objet fabrique concret lors de l'initialisation. Mais avant cela, l'application doit choisir le type de la fabrique en fonction de la configuration ou des paramètres d'environnement.

Structure



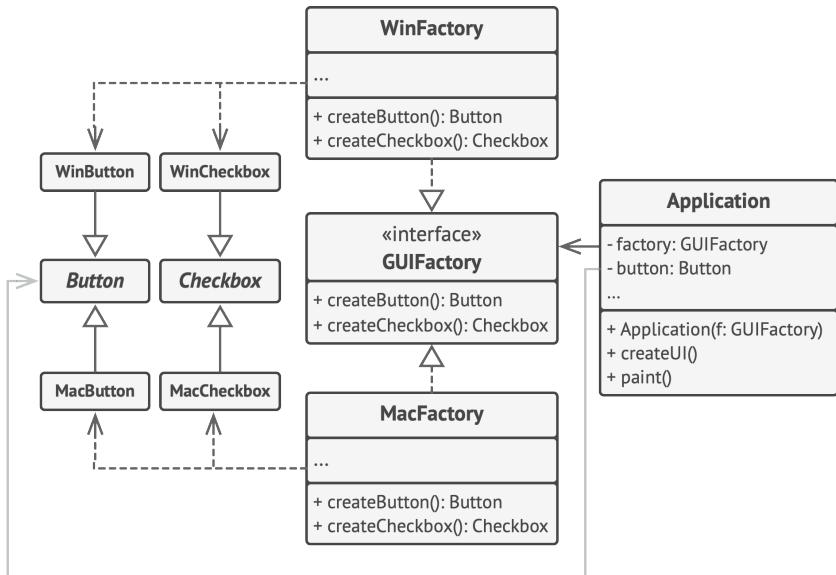
1. Les **Produits Abstraits** déclarent une interface pour un ensemble d'objets distincts mais apparentés, qui forment une famille de produits.
2. Les **Produits Concrets** sont des implémentations des produits abstraits groupés par variantes. Chaque produit abstrait (chaise/sofa) doit être implémenté dans toutes les variantes (victorien/moderne).
3. L'interface **Fabrique Abstraite** déclare un ensemble de méthodes pour créer chacun des produits abstraits.
4. Les **Fabriques Concètes** implémentent les opérations de création d'objets de la fabrique abstraite. Chaque fabrique concrète

correspond à une variante spécifique de produits et ne crée que ces variantes.

5. Bien que les fabriques concrètes instancient des produits concrets, leurs méthodes de création ont une valeur de retour correspondant aux produits *abstraits*. Le code client qui sollicite une fabrique est ainsi isolé de la variante du produit obtenu. Le **client** peut travailler avec n'importe quelle variante de fabrique ou produit, tant qu'il interagit avec les interfaces abstraites.

Pseudo-code

Cet exemple montre comment la **Fabrique abstraite** peut être utilisée pour créer des éléments d'une UI multiplateforme sans coupler le code client avec les classes concrètes de l'UI, tout en gardant une cohérence entre les éléments créés et le système d'exploitation sélectionné.



Exemple des classes d'une UI multiplateforme

Tous les éléments de l'UI d'une application multiplateforme sont censés avoir un comportement identique quel que soit le système d'exploitation, mais leur apparence peut légèrement varier. Vous devez faire en sorte que les éléments de l'interface correspondent bien au style du système d'exploitation. Vous ne voudriez pas vous retrouver avec des boutons macOS dans Windows.

L'interface de la fabrique abstraite déclare un ensemble de méthodes de création que le code client peut utiliser pour produire différents types d'éléments de l'UI. Chaque fabrique concrète correspond à un système d'exploitation particulier et crée ses propres éléments de l'UI en fonction de ce système d'exploitation.

Le fonctionnement est le suivant : lorsqu'une application est exécutée, elle vérifie le système d'exploitation utilisé et exploite cette information pour créer un objet de la fabrique qui y correspond. Cette fabrique est ensuite utilisée pour générer tout le reste des éléments de l'UI, ce qui évite les erreurs.

Grâce à cette approche, tant qu'il utilise leurs interfaces abstraites, le code client ne dépend plus des classes concrètes de la fabrique et des éléments de l'UI. Cela lui permet également d'exploiter les nouveaux éléments de l'UI ou les fabriques que vous pourriez ajouter dans le futur.

Nous n'avons ainsi plus besoin de modifier le code client à chaque nouvel élément de l'interface utilisateur que vous voulez ajouter dans votre application. Vous devrez simplement créer une nouvelle classe fabrique produisant ces éléments et apporter quelques modifications au code d'initialisation afin qu'il choisisse la classe appropriée.

```
1 // L'interface de la fabrique abstraite déclare un ensemble de
2 // méthodes qui retournent différents produits abstraits. Ces
3 // produits font partie de ce que l'on appelle une famille et
4 // sont reliés par un concept ou un thème de haut niveau. Les
5 // produits d'une famille sont généralement capables de
6 // collaborer les uns avec les autres. Une famille de produits
7 // peut avoir plusieurs variantes, mais les produits d'une
8 // variante ne sont pas compatibles avec les produits d'une
9 // autre.
10 interface GUIFactory is
```

```
11     method createButton():Button
12     method createCheckbox():Checkbox
13
14
15 // Les fabriques concrètes construisent une famille de produits
16 // qui appartiennent à une seule variante. La fabrique garantit
17 // la compatibilité des produits qui en sortent. Les signatures
18 // des méthodes des fabriques concrètes renvoient un produit
19 // abstrait, et à l'intérieur de la méthode, un produit concret
20 // est instancié.
21 class WinFactory implements GUIFactory {
22     method createButton():Button {
23         return new WinButton()
24     }
25     method createCheckbox():Checkbox {
26         return new WinCheckbox()
27     }
28 // Chaque fabrique concrète possède une variante de produit
29 // correspondante.
30 class MacFactory implements GUIFactory {
31     method createButton():Button {
32         return new MacButton()
33     }
34     method createCheckbox():Checkbox {
35         return new MacCheckbox()
36     }
37 // Chaque produit d'une famille de produits doit avoir une
38 // interface de base. Toutes les variantes du produit doivent
39 // implémenter cette interface.
40 interface Button {
41     method paint()
42 // Les produits concrets sont créés par les fabriques concrètes
```

```
43 // correspondantes.
44 class WinButton implements Button is
45     method paint() is
46         // Affiche un bouton avec le style Windows.
47
48 class MacButton implements Button is
49     method paint() is
50         // Affiche un bouton avec le style macOS.
51
52 // Voici l'interface de base d'un autre produit. Tous les
53 // produits peuvent interagir les uns avec les autres, mais les
54 // interactions adéquates devraient être implémentées de
55 // préférence entre les produits d'une même variante concrète.
56 interface Checkbox is
57     method paint()
58
59 class WinCheckbox implements Checkbox is
60     method paint() is
61         // Affiche une case à cocher avec le style Windows.
62
63 class MacCheckbox implements Checkbox is
64     method paint() is
65         // Affiche une case à cocher avec le style macOS.
66
67
68 // Le code client travaille uniquement avec les types abstraits
69 // des fabriques et des produits : FabriqueGUI, Bouton et
70 // CaseÀCocher. Ceci vous permet d'envoyer n'importe quelle
71 // sous-classe de fabrique ou de produit au code client sans
72 // toucher à son code.
73 class Application is
74     private field factory: GUIFactory
```

```

75  private field button: Button
76  constructor Application(factory: GUIFactory) is
77      this.factory = factory
78  method createUI() is
79      this.button = factory.createButton()
80  method paint() is
81      button.paint()
82
83
84 // L'application choisit le type de la fabrique en fonction de
85 // la configuration actuelle ou des paramètres d'environnement,
86 // et la crée à l'exécution (en général lors de la phase
87 // d'initialisation).
88 class ApplicationConfigurator is
89     method main() is
90         config = readApplicationConfigFile()
91
92         if (config.OS == "Windows") then
93             factory = new WinFactory()
94         else if (config.OS == "Mac") then
95             factory = new MacFactory()
96         else
97             throw new Exception("Error! Unknown operating system.")
98
99         Application app = new Application(factory)

```

💡 Possibilités d'application

- ⌚ Utilisez la fabrique abstraite si votre code a besoin de manipuler des produits d'un même thème, mais que vous ne voulez pas qu'il dépende des classes concrètes de ces produits –

soit vous ne les connaissez pas encore, soit vous voulez juste rendre votre code évolutif.

- ⚡ La fabrique abstraite fournit une interface qui permet de créer des objets pour chaque classe de la famille de produits. Tant que votre code utilise cette interface pour créer ses objets, il prendra systématiquement les bonnes variantes des produits disponibles dans votre application.
- ⚡ Étudiez la possibilité d'utiliser la fabrique abstraite lorsqu'une classe se retrouve avec un ensemble de patrons Fabrique qui occultent sa fonction principale.
- ⚡ Dans un programme bien conçu *chaque classe n'a qu'une seule responsabilité*. Lorsqu'une classe gère plusieurs types de produits, déplacer les méthodes fabrique dans des fabriques individuelles ou dans une implémentation à part entière d'une fabrique abstraite est souvent plus pratique.

Mise en œuvre

1. Établissez une matrice de vos différents types de produits et leurs variantes.
2. Déclarez des interfaces abstraites pour tous vos types de produits. Mettez en place vos produits concrets dans des classes qui implémentent ces interfaces.

3. Déclarez une interface abstraite de la fabrique avec un ensemble de méthodes de création pour tous les produits abstraits.
4. Implémentez une classe fabrique concrète pour chaque variante des produits.
5. Insérez le code de l'initialisation de la fabrique quelque part dans votre application. Il doit instancier une des fabriques concrètes en fonction de la configuration de l'application ou de l'environnement d'exécution. Passez cette fabrique à toutes les classes qui construisent des produits.
6. Parcourez votre code et repérez tous les appels aux constructeurs des produits. Remplacez-les par des appels à la méthode de création appropriée de la fabrique.

Avantages et inconvénients

- ✓ Vous êtes assurés que les produits d'une fabrique sont compatibles entre eux.
- ✓ Vous découpez le code client des produits concrets.
- ✓ *Principe de responsabilité unique.* Vous pouvez déplacer tout le code de création des produits au même endroit, pour une meilleure maintenabilité.
- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouvelles variantes de produits sans endommager l'existant.

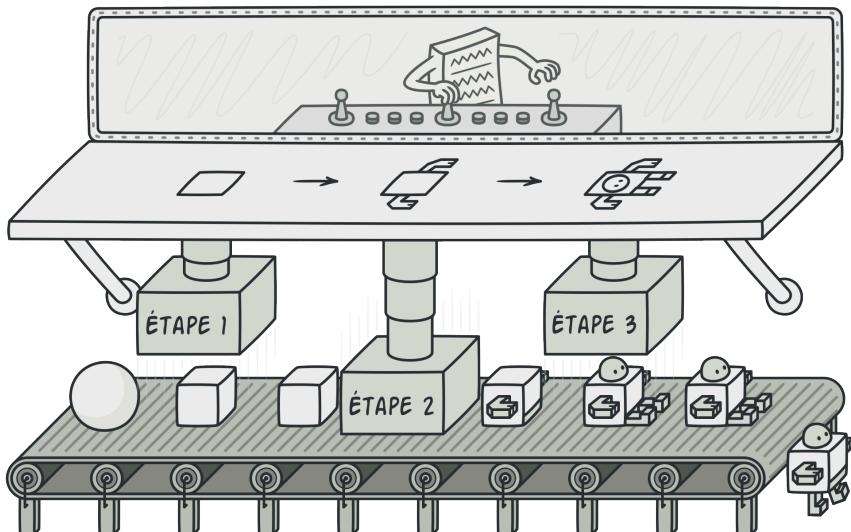
- ✗ Le code peut devenir plus complexe que nécessaire, car ce patron de conception impose l'ajout de nouvelles classes et interfaces.

↔ 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).
- Le **Monteur** se concentre sur la construction d'objets complexes étape par étape. La **Fabrique abstraite** se spécialise dans la création de familles d'objets associés. La *fabrique abstraite* retourne le produit immédiatement, alors que le *monteur* vous permet de lancer des étapes supplémentaires avant de récupérer le produit.
- Les classes **Fabrique abstraite** sont souvent basées sur un ensemble de **Fabriques**, mais vous pouvez également utiliser le **Prototype** pour écrire leurs méthodes.
- La **Fabrique abstraite** peut remplacer la **Façade** si vous voulez simplement cacher au code client la création des objets du sous-système.
- Vous pouvez utiliser la **Fabrique abstraite** avec le **Pont**. Ce couple est très utile quand les abstractions définies par le *pont* ne fonctionnent qu'avec certaines implémentations spé-

cifiques. Dans ce cas, la *fabrique abstraite* peut encapsuler ces relations et cacher la complexité au code client.

- Les **Fabriques abstraites**, **Monteurs** et **Prototypes** peuvent tous être implémentés comme des **Singlenton**s.



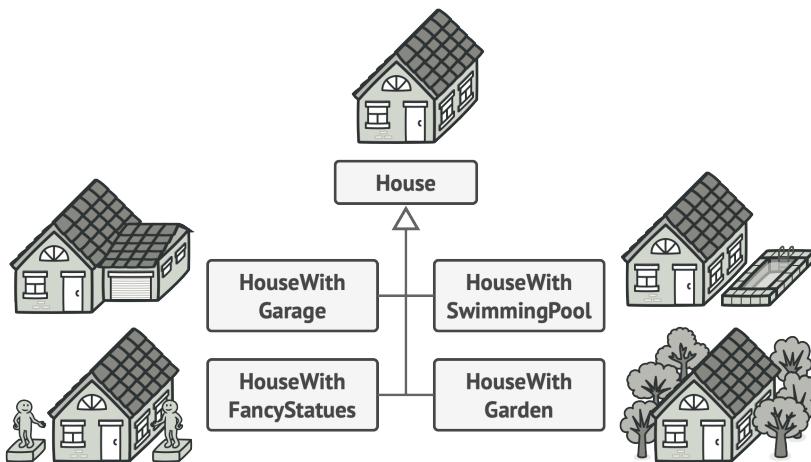
MONTEUR

Alias : Builder

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.

(:() Problème

Imaginez un objet complexe qui nécessite une initialisation fastidieuse, composée de plusieurs parties avec de nombreux champs et objets imbriqués. Le code d'initialisation va se retrouver dans un constructeur, enterré sous une pile monstrueuse de paramètres, ou encore pire : réparti un peu partout dans le code client.

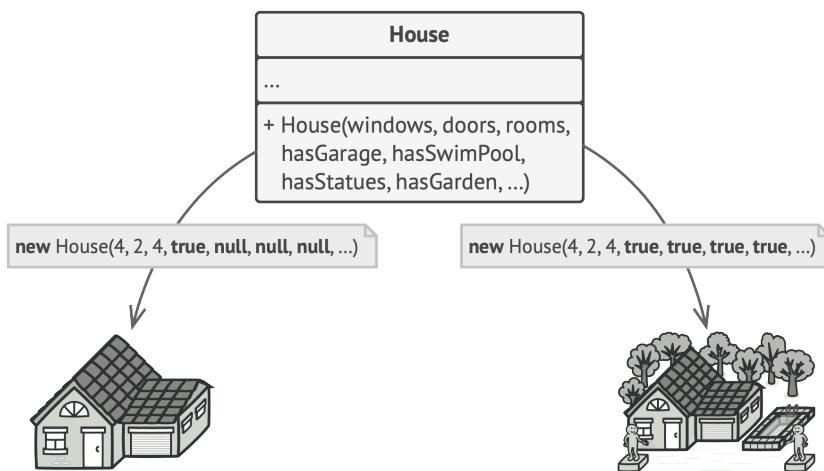


Créer une sous-classe pour chaque configuration possible d'un objet risque de rendre le programme trop complexe.

Réfléchissons à la manière de modéliser un objet `Maison`. Pour fabriquer une maison de base, vous devez construire quatre murs et un sol, installer une porte, poser quelques fenêtres et bâtir un toit. Mais comment procéder si vous voulez une plus grande maison avec plus de lumière, un peu de terrain et autres commodités (un système de chauffage, de la plomberie et des câbles électriques) ?

La solution la plus simple est d'étendre la classe de base `Maison` et de créer un ensemble de sous-classes pour couvrir toutes les combinaisons de paramètres. Mais au bout d'un certain temps, vous allez vous retrouver avec un nombre considérable de sous-classes. Le moindre paramètre supplémentaire comme le style du porche par exemple, va encore plus développer la hiérarchie.

Voici une autre approche qui n'implique pas de générer des sous-classes : vous pouvez créer un constructeur géant dans la classe de base `Maison` avec tous les paramètres contrôlant l'objet maison. Cette solution élimine le besoin de sous-classes, mais entraîne un autre problème.



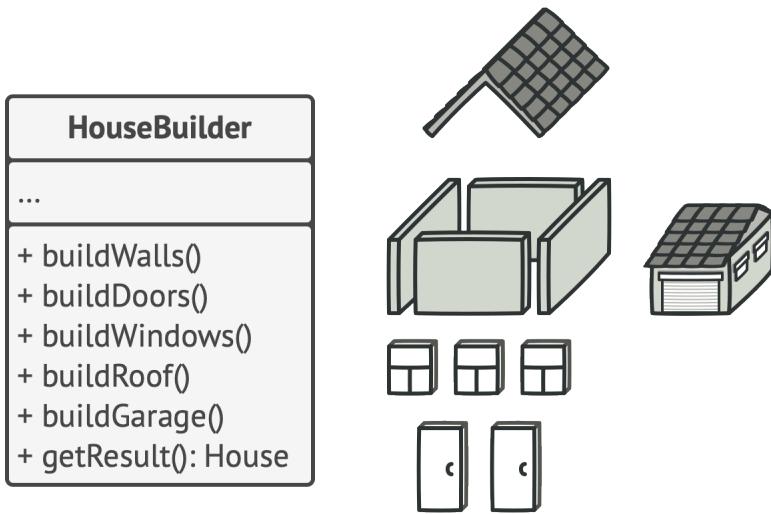
Un constructeur qui possède de nombreux paramètres a ses inconvénients : ces derniers ne sont pas toujours tous utilisés.

Dans la majorité des cas, la plupart des paramètres resteront inutilisés, rendant l'appel au constructeur assez hideux. Par

exemple, le paramètre recensant les piscines se révèle inutile neuf fois sur dix, car peu de maisons en sont équipées.

😊 Solution

Le patron de conception monteur propose d'extraire le code du constructeur d'objet de sa classe et de le déplacer dans des objets distincts appelés *monteurs*.



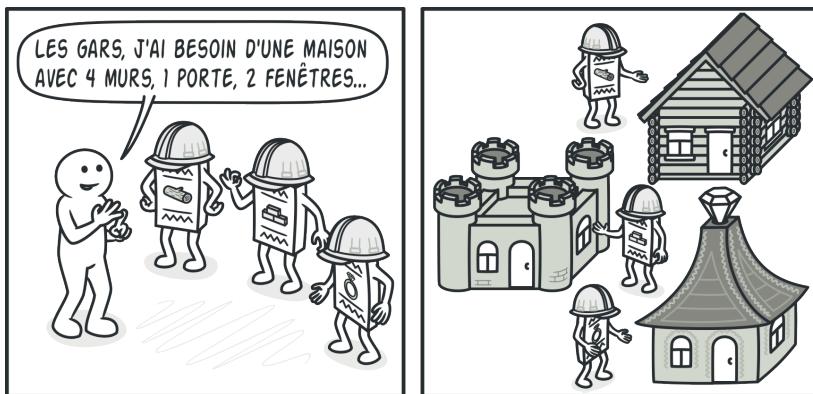
Le patron de conception monteur permet de construire des objets complexes étape par étape. Le monteur empêche les autres objets d'accéder au produit pendant sa construction.

Il organise la construction de l'objet à l'aide d'une série d'étapes (`construireMurs`, `construirePorte`, etc.). Pour créer un objet, vous allez effectuer une séquence d'étapes dans un objet monteur. Le gros avantage, c'est que vous n'avez pas be-

soin d'appeler toutes les étapes, mais seulement celles nécessaires à la création de la configuration particulière d'un objet.

Certaines étapes de la construction peuvent demander des implémentations variables en fonction des différentes représentations du produit. Par exemple, les murs d'une cabane peuvent être en bois, mais ceux d'un château seront en pierre.

Dans ce cas, vous pouvez créer plusieurs monteurs qui implémentent le même ensemble d'étapes de construction, mais d'une manière différente. Vous pouvez ensuite utiliser ces monteurs dans le processus de construction (c'est-à-dire une succession d'appels ordonnés des étapes) pour créer différents types d'objets.



Ces monteurs exécutent la même tâche, mais de manière différente.

Prenons un autre exemple : un premier monteur qui fabrique tout à partir de bois et de verre, un deuxième qui utilise de la pierre et du fer et un troisième qui se sert d'or et de dia-

mants. En appelant les mêmes étapes, vous pouvez construire une maison avec le premier, un petit château avec le deuxième et un palais grâce au troisième. Mais tout ceci ne peut fonctionner que si le code client qui appelle les étapes de la construction peut interagir avec les monteurs via une interface commune.

Directeur (Director)

Vous pouvez aller encore plus loin en prenant tous les appels aux étapes utilisées pour construire un produit, et en les mettant dans une classe séparée que l'on nomme *directeur*. La classe directeur va définir l'ordre d'exécution des différentes étapes et le monteur fournit les implémentations de ces étapes.

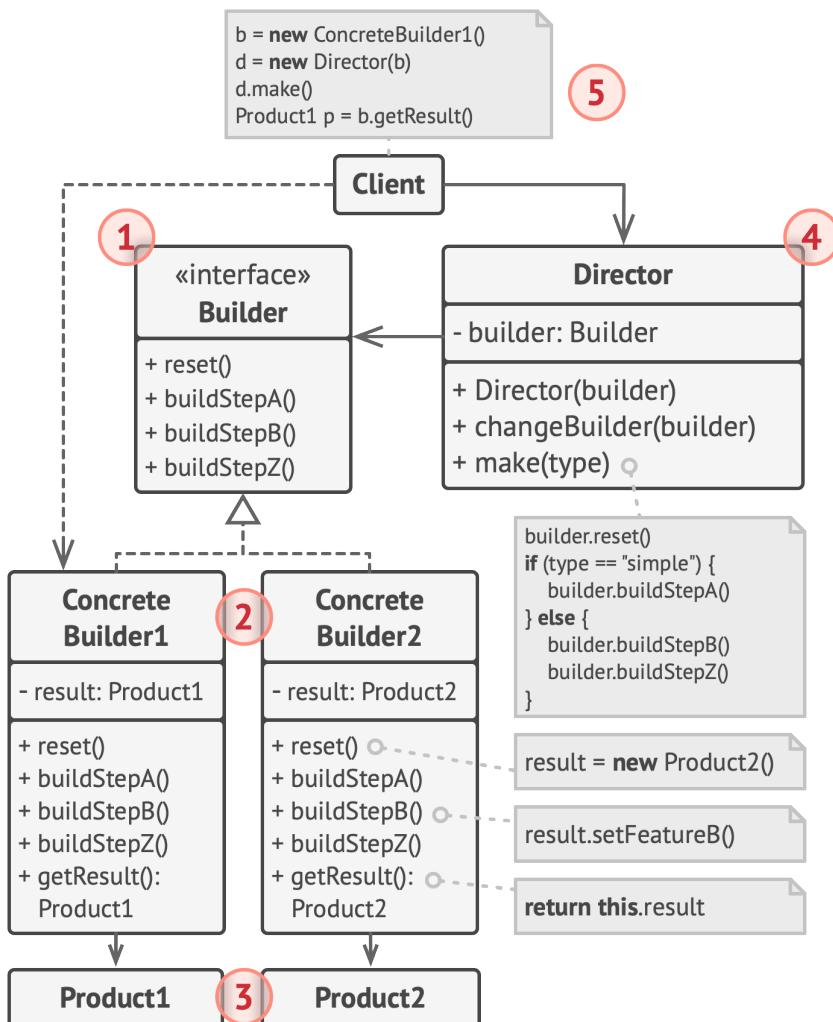


Le directeur connaît les étapes à suivre pour construire un produit fonctionnel.

La classe directeur n'est pas obligatoire. Vous pouvez toujours appeler les étapes de construction dans un ordre spécifique depuis le code client. Cependant, la classe directeur se révèle idéale pour y placer les routines de construction et pouvoir les réutiliser ensuite dans votre programme.

De plus, le directeur cache au client les détails de la construction du produit. Le client doit juste associer un monteur avec un directeur, lancer la construction via le directeur, puis récupérer le résultat auprès du monteur.

Structure

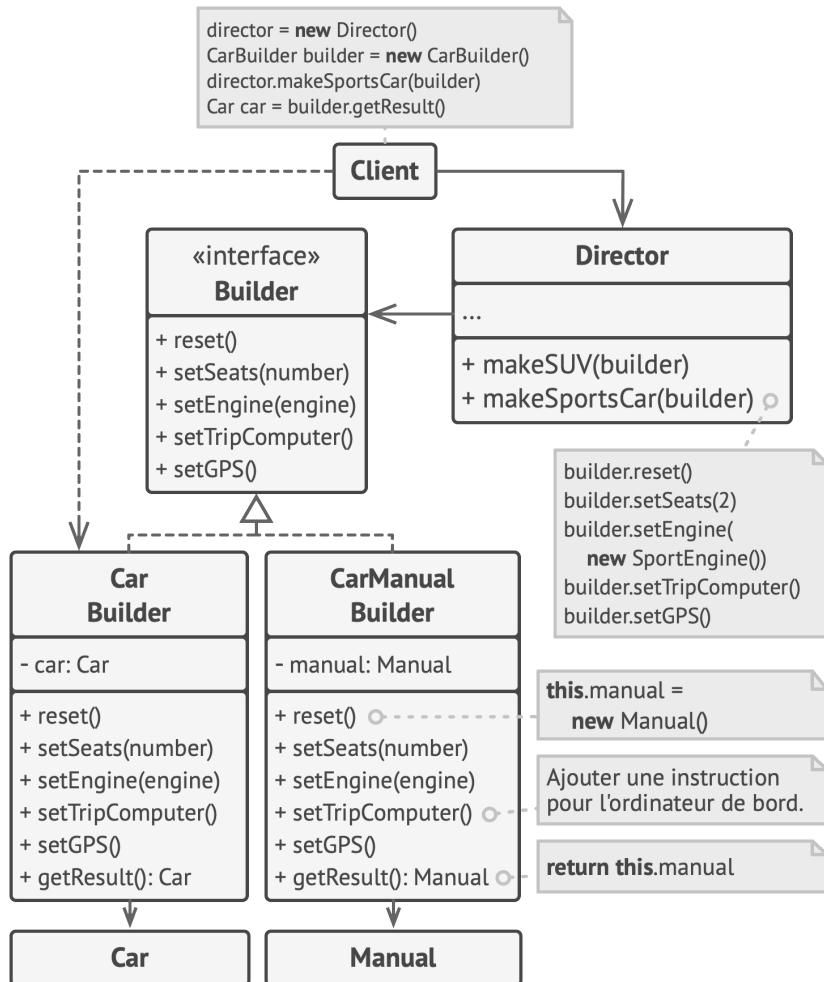


1. L'interface du **Monteur** déclare les étapes communes de la construction du produit entre tous les monteurs.

2. Les **Monteurs Concrets** fournissent différentes implémentations des étapes de la construction. Ils peuvent créer des produits qui ne reprennent pas l'interface commune.
3. Les **Produits** sont les résultats retournés. Les produits construits par les différents monteurs ne sont pas obligés d'appartenir à la même hiérarchie de classes ni d'avoir la même interface.
4. Le **Directeur** indique l'ordonnancement des étapes de construction et offre la possibilité de créer et de réutiliser des configurations spécifiques pour les produits.
5. Le **Client** doit associer l'un des monteurs au directeur. En général cette association n'est réalisée qu'une seule fois, grâce aux paramètres du constructeur du directeur. Pour toute construction ultérieure, le directeur utilise l'objet monteur. En guise d'alternative, le client peut passer l'objet monteur à la méthode de production du directeur. Dans ce cas, vous pouvez utiliser un monteur différent chaque fois que vous lancez une production avec le directeur.

Pseudo-code

Voici un exemple qui montre comment un **Monteur** peut réutiliser le même code de construction d'objet pour assembler différents types de produits comme des voitures, et créer leurs manuels respectifs.



Un exemple de construction de voitures étape par étape et le manuel d'utilisation qui correspond à leur modèle.

Une voiture est un objet complexe. Elle peut être fabriquée de cent manières différentes. Plutôt que d'encombrer la classe Voiture avec un énorme constructeur, nous avons extrait le code dans une classe monteur séparée pour la voiture. Cette

classe est composée d'un ensemble de méthodes pour configurer les différentes parties d'une voiture.

Si le code client veut assembler un modèle spécial de voiture bénéficiant d'un réglage de précision, il peut s'adresser directement au monteur. Il peut également déléguer l'assemblage à la classe directeur qui connaît le processus de fabrication à indiquer au monteur pour les modèles de voitures les plus populaires.

Ce qui suit va peut-être vous choquer, mais chaque voiture doit posséder son propre manuel (franchement, qui les lit?). Le manuel décrit toutes les fonctionnalités de la voiture et les détails vont varier selon les modèles. Il semble donc approprié de réutiliser un processus de construction existant pour les voitures et leurs manuels. Bien entendu, la création d'un manuel et d'une voiture sont deux procédés complètement différents et nous devons concevoir des monteurs spécialisés pour les manuels. Cette classe va implémenter les mêmes méthodes de construction que sa cousine assembleuse de voitures, mais au lieu de fabriquer des pièces de voitures, elle se contente de les décrire. Nous pouvons construire une voiture ou un manuel en passant ces monteurs au même objet directeur.

L'étape finale consiste à récupérer l'objet qui en résulte. Même si une voiture en métal et un manuel en papier sont directement liés, ce sont deux choses complètement différentes. Nous ne pouvons pas insérer une méthode pour récupérer les

résultats dans le directeur sans le coupler aux classes concrètes du produit. Par conséquent, c'est le monteur qui effectue le travail qui nous donne ensuite le résultat.

```
1 // L'utilisation du patron de conception monteur n'est
2 // conseillée que si vos produits sont complexes et nécessitent
3 // une configuration étendue. Bien qu'ils n'aient pas la même
4 // interface, les deux produits suivants sont liés.
5 class Car is
6     // Une voiture est équipée d'un GPS, d'un ordinateur de bord
7     // et d'un certain nombre de sièges. Les différents modèles
8     // de voitures (sport, SUV, cabriolet) ont différentes
9     // fonctionnalités installées ou activées.
10
11 class Manual is
12     // Chaque voiture doit avoir un manuel d'utilisation qui
13     // correspond à sa configuration et décrit toutes ses
14     // fonctionnalités.
15
16
17 // L'interface du monteur contient des méthodes spécialisées
18 // pour créer les différentes parties des objets du produit.
19 interface Builder is
20     method reset()
21     method setSeats(...)
22     method setEngine(...)
23     method setTripComputer(...)
24     method setGPS(...)
25
26 // Les classes concrètes du monteur suivent l'interface monteur
27 // et procurent des implémentations spécifiques pour les étapes
```

```
28 // de la fabrication. Votre programme peut contenir plusieurs
29 // variantes de monteurs, chacune avec sa propre implémentation.
30 class CarBuilder implements Builder is
31     private field car:Car
32
33     // Une instance monteur fraîchement créée doit contenir un
34     // objet produit vide qu'elle va ensuite assembler.
35     constructor CarBuilder() is
36         this.reset()
37
38     // La méthode reset nettoie l'objet qui est construit.
39     method reset() is
40         this.car = new Car()
41
42     // Toutes les étapes de la fabrication manipulent la même
43     // instance du produit.
44     method setSeats(...) is
45         // Configure le nombre de sièges dans la voiture.
46
47     method setEngine(...) is
48         // Installe un moteur donné.
49
50     method setTripComputer(...) is
51         // Installe un ordinateur de bord.
52
53     method setGPS(...) is
54         // Installe un système de géolocalisation.
55
56     // Les monteurs concrets sont censés fournir leurs propres
57     // méthodes pour récupérer les résultats. Ce fonctionnement
58     // permet aux différents types de monteurs de créer des
59     // produits entièrement différents qui ne suivent pas la
```

```
60 // même interface. Par conséquent, les méthodes ne peuvent
61 // pas être déclarées dans l'interface du monteur (en tout
62 // cas pas dans un langage de programmation doté d'un
63 // système de typage statique).
64 //
65 // En général, après avoir retourné le résultat final au
66 // client, une instance de monteur doit être prête à lancer
67 // la fabrication d'un autre produit. C'est pour cette
68 // raison que l'on appelle généralement la méthode reset à
69 // la fin du corps de la méthode `getProduct`. Mais ce
70 // comportement n'est pas obligatoire et votre monteur peut
71 // attendre que le code client lance un appel explicite à un
72 // reset pour vous débarrasser du résultat précédent.
73 method getProduct():Car is
74     product = this.car
75     this.reset()
76     return product
77
78 // Contrairement aux autres patrons de création, le monteur vous
79 // permet de fabriquer des produits qui ne suivent pas la même
80 // interface.
81 class CarManualBuilder implements Builder is
82     private field manual:Manual
83
84     constructor CarManualBuilder() is
85         this.reset()
86
87     method reset() is
88         this.manual = new Manual()
89
90     method setSeats(...) is
91         // Fonctionnalités des sièges dans le manuel.
```

```
92
93     method setEngine(...) is
94         // Ajoute les informations concernant le moteur.
95
96     method setTripComputer(...) is
97         // Ajoute les instructions concernant l'ordinateur de
98         // bord.
99
100    method setGPS(...) is
101        // Ajoute les instructions concernant le GPS.
102
103    method getProduct():Manual is
104        // Retourne le manuel et remet à zéro le monteur
105        // (reset).
106
107
108    // Le directeur a pour seule responsabilité l'ordonnancement des
109    // étapes de la fabrication. Il se révèle utile lorsque vous
110    // fabriquez des produits avec un ordre précis ou dans une
111    // configuration particulière. À proprement parler, la classe
112    // Directeur n'est pas obligatoire, car le client peut manipuler
113    // les monteurs directement.
114    class Director is
115        // Le directeur manipule n'importe quelle instance de
116        // monteur que le code client lui envoie. Ainsi, le code
117        // client peut modifier le type final du produit qui vient
118        // d'être assemblé. Le directeur peut fabriquer plusieurs
119        // variantes de produits en utilisant les mêmes étapes de
120        // fabrication.
121    method constructSportsCar(builder: Builder) is
122        builder.reset()
123        builder.setSeats(2)
```

```
124     builder.setEngine(new SportEngine())
125     builder.setTripComputer(true)
126     builder.setGPS(true)
127
128     method constructSUV(builder: Builder) is
129         // ...
130
131
132 // Le code client crée un objet monteur, le passe au directeur
133 // et lance le processus de fabrication. Le résultat final est
134 // récupéré dans l'objet monteur.
135 class Application is
136
137     method makeCar() is
138         director = new Director()
139
140         CarBuilder builder = new CarBuilder()
141         director.constructSportsCar(builder)
142         Car car = builder.getProduct()
143
144         CarManualBuilder builder = new CarManualBuilder()
145         director.constructSportsCar(builder)
146
147         // Le produit final est souvent récupéré depuis un objet
148         // monteur, car le directeur est indépendant des
149         // monteurs et des produits concrets et il ne les voit
150         // donc pas.
151         Manual manual = builder.getProduct()
```

Possibilités d'application

-  Utilisez le patron de conception monteur afin de vous débarrasser d'un « constructeur télescopique ».
-  Prenons un constructeur avec dix paramètres facultatifs. Faire un appel à cette monstruosité n'est pas très pratique : vous surchargez le constructeur avec plusieurs petites versions, mais avec moins de paramètres. Ces constructeurs font toujours référence au constructeur principal en donnant des valeurs par défaut aux paramètres optionnels.

```

1 class Pizza {
2     Pizza(int size) { ... }
3     Pizza(int size, boolean cheese) { ... }
4     Pizza(int size, boolean cheese, boolean pepperoni) { ... }
5     // ...

```

Cette monstruosité ne peut être créée que dans certains langages qui permettent la surcharge tels que le C# ou le Java.

Le monteur vous permet de créer des objets étape par étape, en utilisant uniquement celles qui sont nécessaires. Après avoir implémenté le patron, vous n'avez plus besoin d'entasser les paramètres dans vos constructeurs.

-  Utilisez le monteur pour rendre votre code capable de créer différentes représentations de produits (par exemple des maisons en pierre et en bois).

- ⚡ Le monteur est utile lorsque les étapes de la construction des différentes représentations du produit se ressemblent (seuls quelques détails diffèrent).

L'interface de base du monteur définit toutes les étapes possibles de la construction. Les monteurs concrets implémentent les étapes pour construire les représentations particulières du produit. Le directeur quant à lui s'occupe de gérer l'ordonnancement des différentes étapes de construction.

- ⚠ Utilisez le monteur afin de construire une arborescence composite ou d'autres objets complexes.

- ⚡ Le monteur vous permet de construire des produits étape par étape. Vous pouvez déléguer l'exécution de certaines étapes sans endommager le produit final. Vous pouvez même appeler les étapes de manière récursive, ce qui est très pratique dans le cas de la construction d'un objet composite.

Le monteur ne met pas à disposition le produit tant que les étapes de la construction ne sont pas terminées. Par conséquent, le code client ne récupérera jamais un résultat incomplet.

📋 Mise en œuvre

1. Assurez-vous de définir clairement les étapes communes de la construction pour toutes les représentations de produits disponibles, sinon vous ne pourrez pas mettre en place le patron.

2. Déclarez ces étapes dans l'interface du monteur.
3. Pour chaque représentation du produit, créez une classe concrète monteur puis implémentez ses étapes de construction.

N'oubliez pas de mettre en place une méthode pour récupérer le résultat de la construction. Cette méthode ne peut pas être déclarée dans l'interface du monteur, car les différents monteurs peuvent fabriquer des produits qui n'ont pas forcément une interface commune. Nous ne pouvons pas connaître à l'avance le type de retour d'une telle méthode. En revanche, si vos produits appartiennent à une hiérarchie unique, la méthode qui récupère le résultat peut être ajoutée à l'interface de base sans problème.

4. Réfléchissez à la création d'une classe directeur. Elle peut encapsuler différents processus de construction d'un produit en utilisant le même objet monteur.
5. Le code client crée les objets monteur et directeur. Le client doit passer un objet monteur au directeur avant le début de la construction. En général, il ne le fait qu'une seule fois, via les paramètres du constructeur du directeur. Le directeur utilise l'objet monteur pour toute construction ultérieure. Nous pourrions également passer le monteur directement à la méthode de construction du directeur.
6. Le résultat de la construction peut être obtenu directement à partir du directeur, si tous les produits sont branchés sur la

même interface. Sinon, le client doit aller directement chercher le résultat auprès du monteur.

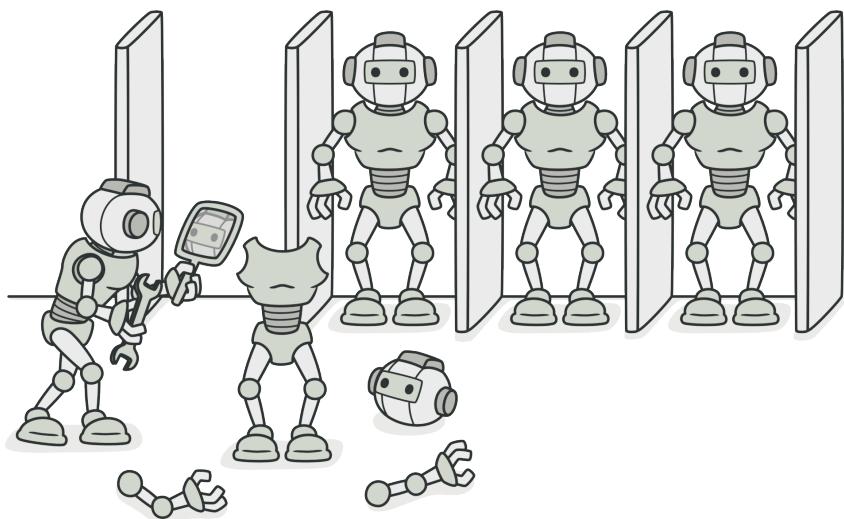
⚠️ Avantages et inconvénients

- ✓ Vous pouvez construire les objets étape par étape et les déléguer ou les exécuter récursivement.
- ✓ Vous pouvez réutiliser le même code de construction lorsque vous construisez différentes représentations des produits.
- ✓ *Principe de responsabilité unique.* Vous pouvez découpler le code complexe de la construction et la logique métier du produit.
- ✗ Le monteur nécessite de créer beaucoup nouvelles classes, ce qui accroît la complexité générale du code.

➡️ 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).
- Le **Monteur** se concentre sur la construction d'objets complexes étape par étape. La **Fabrique abstraite** se spécialise dans la création de familles d'objets associés. La *fabrique abstraite* retourne le produit immédiatement, alors que le *monteur* vous permet de lancer des étapes supplémentaires avant de récupérer le produit.

- Vous pouvez utiliser le **Monteur** lorsque vous créez des arbres **Composites** complexes, car vous pouvez programmer les étapes de la construction récursivement.
- Vous pouvez combiner le **Monteur** avec le **Pont** : la classe directeur joue le rôle de l'abstraction, et les différents *monteurs* prennent le rôle des implémentations.
- Les **Fabriques abstraites**, **Monteurs** et **Prototypes** peuvent tous être implémentés comme des **Singlenton**s.



PROTOTYPE

Alias : Clone

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

(:() Problème

Nous voulons une copie exacte d'un objet. Comment vous y prendriez-vous? Tout d'abord, vous devez créer un nouvel objet de cette même classe. Ensuite, vous devez parcourir tous les attributs de l'objet d'origine et copier leurs valeurs dans le nouvel objet.

Super! Mais il y a un hic. Certains objets ne peuvent pas être copiés ainsi, car leurs attributs peuvent être privés et ne pas être visibles en dehors de l'objet.



Copier un objet « depuis l'extérieur » n'est pas toujours possible.

Un autre problème se profile avec l'approche directe. Étant donné que vous devez connaître la classe de l'objet pour le dupliquer, votre code devient dépendant de cette classe. Même si cette dépendance ne vous effraie pas, il y a un autre hic. Parfois, vous ne connaissez que l'interface implémentée par l'objet, mais pas sa classe concrète. Si le paramètre d'une mé-

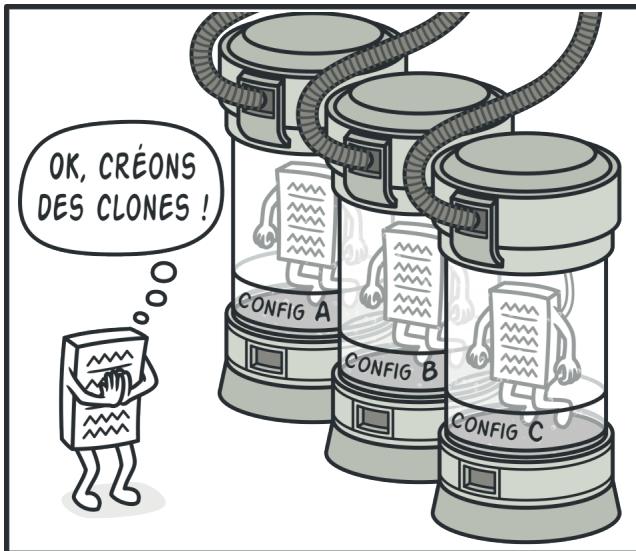
thode accepte tous les objets qui implémentent une interface donnée, vous pouvez rencontrer des problèmes.

Solution

Le patron de conception prototype délègue le processus de clonage aux objets qui vont être copiés. Il déclare une interface commune pour tous les objets qui pourront être clonés. Cette interface vous permet de cloner un objet sans coupler votre code à la classe de cet objet. En général, une telle interface ne contient qu'une seule méthode `clone`.

L'implémentation de cette méthode se ressemble dans toutes les classes. Elle crée un objet de la classe actuelle et reporte les valeurs des attributs de l'objet d'origine dans le nouveau. Vous pouvez également copier des attributs privés, car la plupart des langages de programmation autorisent l'accès à des objets d'une même classe.

Un objet qui peut être cloné est appelé un *prototype*. Lorsque vos objets possèdent des dizaines d'attributs et des centaines de configurations possibles, le clonage est une alternative au sous-classage.

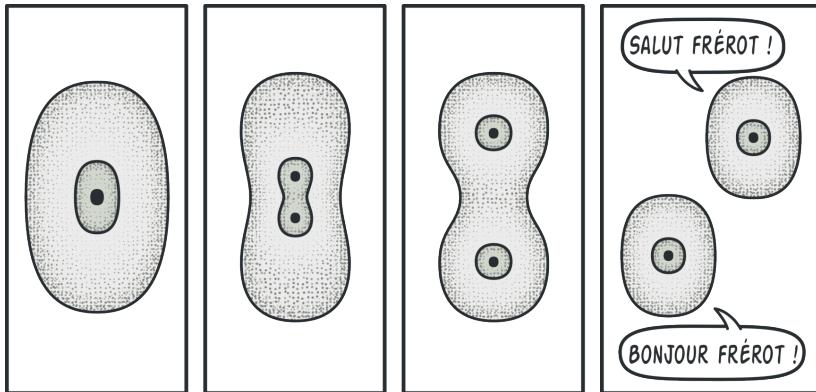


Les prototypes préconstruits sont une alternative au sous-classage.

Leur fonctionnement est le suivant : vous créez un ensemble d'objets configurés différemment. Dès que vous avez besoin de l'un de ces objets, vous clonez un prototype plutôt que d'en construire un nouveau.

Analogie

Dans la vie réelle, des prototypes sont utilisés pour exécuter des tests avant de lancer les chaînes de production d'un produit. Dans ce cas toutefois, les prototypes ne font pas partie intégrante de la fabrication, ils ne jouent qu'un rôle passif.

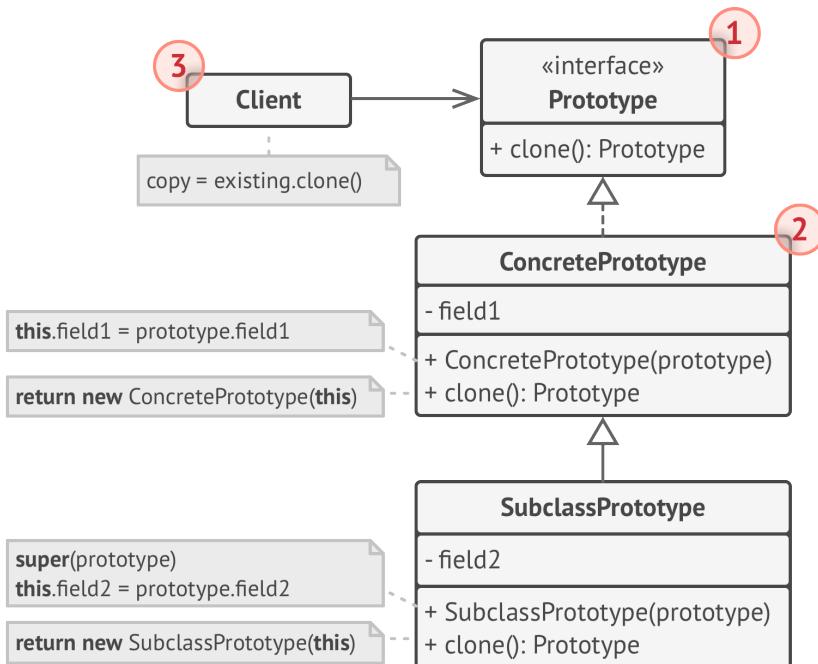


La division cellulaire.

Puisque les prototypes industriels ne se clonent pas eux-mêmes, étudions le processus de la mitose d'une cellule (petit rappel de vos cours de biologie). Une fois la division mitotique terminée, une paire de cellules identiques est créée. La cellule originale sert de prototype et joue un rôle actif dans la création de la copie.

Structure

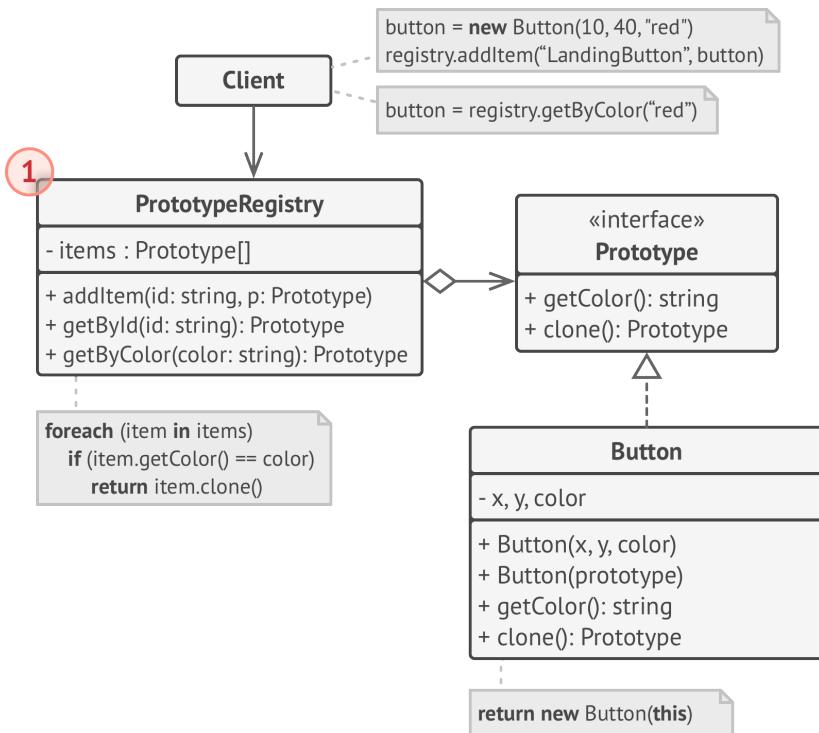
Implémentation de base



1. L'interface **Prototype** déclare les méthodes de clonage. Dans la majorité des cas, il n'y a qu'une seule méthode `clone`.
2. La classe **Prototype Concret** implémente la méthode de clonage. En plus de copier les données de l'objet original dans le clone, cette méthode peut gérer des cas particuliers du processus de clonage, comme des objets imbriqués, démêler les dépendances récursives, etc.

- Le **Client** peut produire une copie de n'importe quel objet implémentant l'interface prototype.

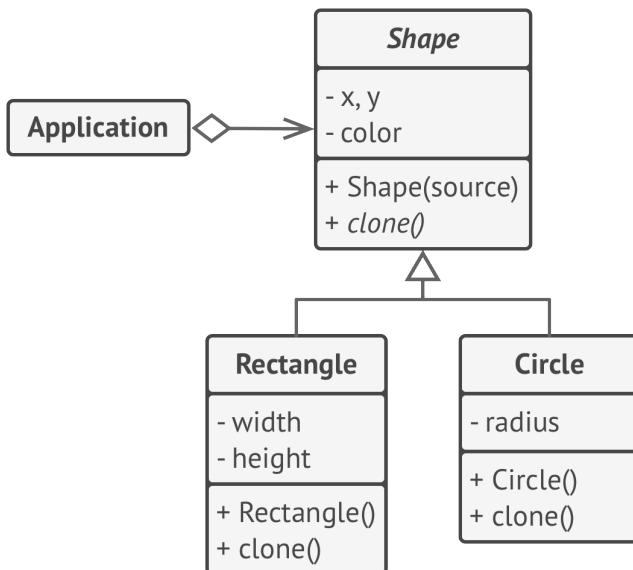
Implémentation du registre de prototypes



- Le **Registre de Prototypes** facilite l'accès aux prototypes fréquemment utilisés. Il stocke un ensemble d'objets préconstruits qui ont déjà été copiés. Le registre de prototypes le plus simple est une table de hachage (hash map) `nom → prototype`. Si vous voulez de meilleurs critères de recherche qu'un simple nom, construisez une version plus robuste du registre.

Pseudo-code

Dans cet exemple, le **Prototype** produit des copies exactes d'objets géométriques, sans coupler le code à leurs classes.



Cloner un ensemble d'objets qui appartiennent à une hiérarchie de classes.

Toutes les formes implémentent la même interface, et cette dernière fournit une méthode de clonage. Une sous-classe peut appeler la méthode de clonage de son parent avant de copier les valeurs de ses propres attributs dans l'objet.

```

1 // Prototype de base.
2 abstract class Shape is
3     field X: int
4     field Y: int
  
```

```
5   field color: string
6
7   // Un constructeur normal.
8   constructor Shape() is
9     // ...
10
11  // Le constructeur du prototype. Un nouvel objet est
12  // initialisé avec les valeurs de l'objet existant.
13  constructor Shape(source: Shape) is
14    this()
15    this.X = source.X
16    this.Y = source.Y
17    this.color = source.color
18
19  // Le traitement de clonage retourne l'une des sous-classes
20  // Forme (Shape)
21  abstract method clone():Shape
22
23
24  // Prototype concret. La méthode de clonage crée un nouvel objet
25  // et le passe au constructeur. Il garde une référence à un
26  // nouveau clone tant que le constructeur n'a pas terminé. Il
27  // est donc impossible de se retrouver avec un clone incomplet
28  // et les résultats du clonage sont toujours fiables.
29  class Rectangle extends Shape is
30    field width: int
31    field height: int
32
33  constructor Rectangle(source: Rectangle) is
34    // Un appel au constructeur parent est requis pour
35    // copier les attributs privés définis dans la classe
36    // mère.
```

```
37     super(source)
38     this.width = source.width
39     this.height = source.height
40
41     method clone():Shape is
42         return new Rectangle(this)
43
44
45 class Circle extends Shape is
46     field radius: int
47
48     constructor Circle(source: Circle) is
49         super(source)
50         this.radius = source.radius
51
52     method clone():Shape is
53         return new Circle(this)
54
55
56 // Quelque part dans le code client.
57 class Application is
58     field shapes: array of Shape
59
60     constructor Application() is
61         Circle circle = new Circle()
62         circle.X = 10
63         circle.Y = 10
64         circle.radius = 20
65         shapes.add(circle)
66
67         Circle anotherCircle = circle.clone()
68         shapes.add(anotherCircle)
```

```
69      // La variable `autreCercle` contient la copie exacte de
70      // l'objet `Cercle`.
71
72      Rectangle rectangle = new Rectangle()
73      rectangle.width = 10
74      rectangle.height = 20
75      shapes.add(rectangle)
76
77  method businessLogic() is
78      // Le prototype est très pratique, car il vous permet de
79      // créer une copie d'un objet en ignorant tout de son
80      // type.
81      Array shapesCopy = new Array of Shapes.
82
83      // Par exemple, nous ne savons pas exactement quels
84      // éléments figurent dans le tableau des formes. Nous
85      // savons juste que ce sont tous des formes. Grâce au
86      // polymorphisme, lorsque nous appelons la méthode
87      // `clone` sur une forme, le programme vérifie sa classe
88      // et lance la méthode clone appropriée de cette classe.
89      // C'est pour cela que nous fabriquons des clones,
90      // plutôt qu'un ensemble d'objets forme.
91      foreach (s in shapes) do
92          shapesCopy.add(s.clone())
93
94      // Le tableau `CopiesFormes` (shapesCopy) contient les
95      // copies exactes du tableau des `formes`.
```

Possibilités d'application

-  **Utilisez le prototype si vous ne voulez pas que votre code dépende des classes concrètes des objets que vous clonez.**
-  Vous rencontrerez ce cas si votre code manipule des objets obtenus via l'interface d'une application externe. Il arrive que votre code ne puisse pas se baser sur les classes concrètes de ces objets car elles sont inconnues.

Le prototype procure une interface générale au code client et lui permet de travailler avec tous les objets clonables. Cette interface rend le code client indépendant des classes concrètes des objets qu'il clone.

-  **Utilisez ce patron si vous voulez réduire le nombre de sous-classes quand elles ne diffèrent que dans leur manière d'initialiser leurs objets respectifs. Ces sous-classes pourraient avoir été prévues pour créer des objets similaires dans des configurations spécifiques.**
-  Le patron de conception prototype vous permet d'utiliser un ensemble d'objets préconstruits (des prototypes) de différentes manières.

Plutôt que d'instancier une sous-classe qui correspond à une configuration précise, le client peut se contenter de chercher le prototype approprié et de le cloner.

Mise en œuvre

1. Créez l'interface du prototype et déclarez-y la méthode `clone`, mais si vous avez une hiérarchie de classes existante, ajoutez juste la méthode à toutes ses classes.
2. Une classe prototype doit définir un autre constructeur capable de prendre un objet de cette classe comme paramètre. Le constructeur doit copier les valeurs des attributs définis dans la classe de l'objet dans l'instance qui vient d'être créée. Si une sous-classe est concernée, vous devez appeler le constructeur du parent pour gérer le clonage de ses attributs privés.

Si le langage de programmation que vous utilisez ne gère pas la surcharge, vous pouvez définir une méthode spéciale pour copier les données de l'objet. Faire tout ceci à l'intérieur du constructeur est plus pratique, car il retourne le résultat directement après avoir appelé l'opérateur `new`.

3. La méthode de clonage ne contient qu'une seule ligne de code : un opérateur `new` avec la version prototype du constructeur. Notez bien que chaque classe doit redéfinir la méthode de clonage et utiliser le nom de sa propre classe avec l'opérateur `new`. Si vous ne le faites pas, la méthode de clonage sera capable de produire des objets de la classe mère.

4. Vous pouvez également créer un registre centralisé des prototypes pour garder un catalogue de prototypes fréquemment utilisés.

Ce registre peut être implémenté avec une classe fabrique, ou mis dans une classe prototype de base avec une méthode statique qui récupère le prototype. Cette méthode ira chercher un prototype en fonction du critère de recherche que le code client passe à la méthode. Ce critère peut être une chaîne de caractères ou un ensemble de paramètres de recherche. Dès que le prototype recherché est trouvé, le registre crée un clone et le retourne au client.

Pour finir, remplacez tous les appels directs aux constructeurs des sous-classes par des appels à la méthode fabrique du registre de prototypes.

Avantages et inconvénients

- ✓ Vous pouvez cloner les objets sans les coupler avec leurs classes concrètes.
- ✓ Vous clonez des prototypes préconstruits et vous pouvez vous débarrasser du code d'initialisation redondant.
- ✓ Vous pouvez créer des objets complexes plus facilement.
- ✓ Vous obtenez une alternative à l'héritage pour gérer les modèles de configuration d'objets complexes.

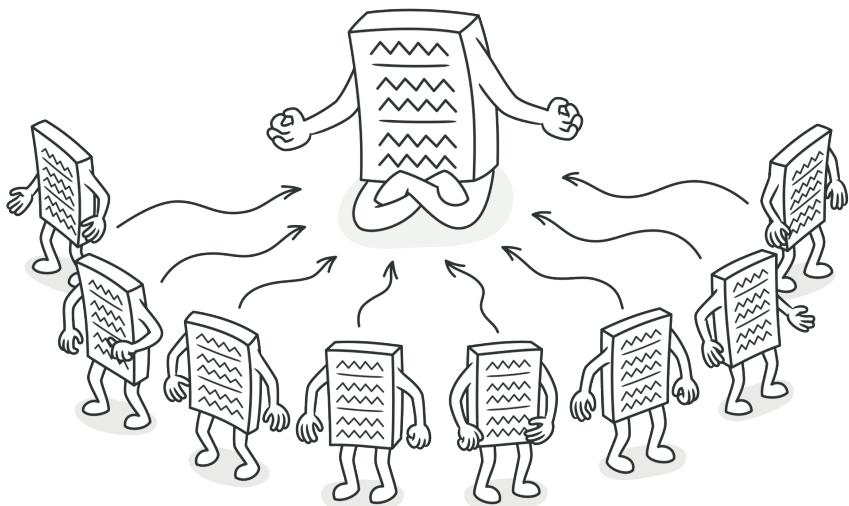
- ✗ Cloner des objets complexes dotés de références circulaires peut se révéler très difficile.

↔ 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.
- Le **Prototype** se révèle utile lorsque vous voulez sauvegarder des copies de **Commandes** dans l'historique.
- Les conceptions qui reposent énormément sur le **Composite** et le **Décorateur** tirent des avantages à utiliser le **Prototype**. Il vous permet de cloner les structures complexes plutôt que de les reconstruire à partir de rien.
- 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.
- Parfois le **Prototype** peut venir remplacer le **Memento** et proposer une solution plus simple. Cela n'est possible que si

l'objet (l'état que vous voulez stocker dans l'historique) est assez direct et ne possède pas de liens vers des ressources externes, ou que les liens sont faciles à recréer.

- Les **Fabriques abstraites**, **Monteurs** et **Prototypes** peuvent tous être implémentés comme des **Singletons**.



SINGLETON

Singleton est un patron de conception de création qui garantit que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.

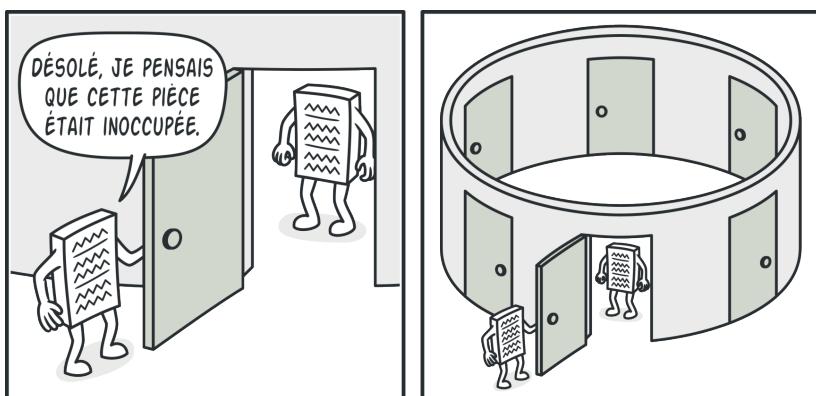
(:() Problème

Le singleton règle deux problèmes à la fois, mais ne respecte pas le *principe de responsabilité unique*.

1. **Il garantit l'unicité d'une instance pour une classe.** Pour quelle raison voudrait-on maîtriser le nombre d'instances d'une classe ? En général, cette situation se présente lorsque l'on veut contrôler l'accès à une ressource partagée – une base de données ou un fichier par exemple.

Son fonctionnement est le suivant : vous créez un objet, mais après un certain temps, vous décidez d'en créer un autre. Plutôt que de vous retrouver avec un objet flambant neuf, vous récupérez celui qui existe déjà.

Vous noterez qu'il est impossible d'implémenter ce comportement avec un constructeur normal, puisqu'un constructeur **doit** théoriquement toujours retourner un nouvel objet.



Les clients ne se rendent pas forcément compte qu'ils travaillent toujours avec le même objet.

- Il fournit un point d'accès global à cette instance. Vous rappelez-vous ces variables globales que vous (bon, d'accord : moi) avez utilisées pour stocker des objets essentiels ? Elles sont très pratiques mais peu fiables, puisque n'importe quelle partie du code peut potentiellement écraser leur contenu et faire planter l'application.

Le singleton vous permet d'accéder à l'objet n'importe où dans le programme, telle une variable globale. Cependant, il protège son instance et l'empêche d'être modifiée.

Un autre aspect majeur vient se glisser dans l'équation : le code qui résout le problème numéro 1 ne doit pas se retrouver éparpillé dans tout le programme. En effet, on préférera tout mettre dans une même classe, surtout si le reste du code repose dessus.

Aujourd'hui, le singleton est devenu très populaire et le terme *singleton* est même parfois employé pour une entité ne résolvant qu'un seul des problèmes listés.

Solution

Toute mise en place d'un singleton est constituée des deux étapes suivantes :

- Rendre le constructeur par défaut privé afin d'empêcher les autres objets d'utiliser l'opérateur `new` avec la classe du singleton.

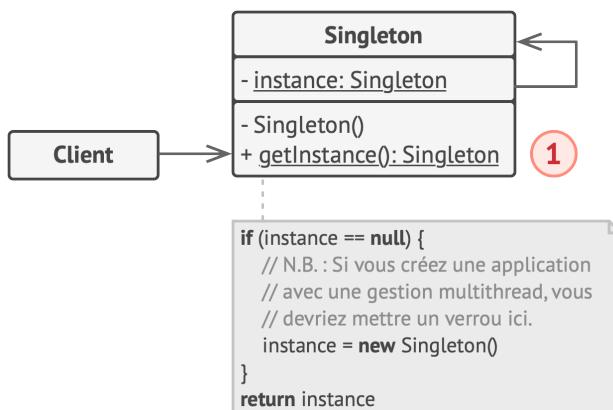
- Mettre en place une méthode de création statique qui se comporte comme un constructeur. En coulisse, cette méthode appelle le constructeur privé pour créer un objet et le sauvegarde dans un attribut statique. Tous les appels ultérieurs à cette méthode retournent l'objet en cache.

Si votre code a accès à la classe du singleton, alors il peut appeler sa méthode statique. À chaque appel de cette méthode, c'est toujours le même objet qui est retourné.

Analogie

Prenons le gouvernement, qui est un excellent exemple de singleton. Un pays ne peut avoir qu'un seul gouvernement officiel. Quels que soient les individus qui composent un gouvernement, l'appellation « Le gouvernement de X » est un point global d'accès qui identifie le groupe de personnes au pouvoir.

Structure



1. La classe **Singleton** déclare la méthode statique `getInstance` qui retourne la même instance de sa propre classe.

Le code client ne doit pas avoir de visibilité sur le constructeur du singleton. Seule la méthode `getInstance` doit permettre l'accès à l'objet du singleton.

Pseudo-code

Dans cet exemple, la classe de la connexion à la base de données est le **Singleton**. Cette classe n'a pas de constructeur public, vous ne pouvez y accéder que grâce à la méthode `getInstance`. Cette méthode met en cache le premier objet créé puis retourne ce même objet lors des appels ultérieurs.

```
1 // La classe baseDeDonnées définit la méthode `getInstance` qui
2 // permet aux clients d'accéder à la même instance de la
3 // connexion à la base de données dans tout le programme.
4 class Database is
5     // L'attribut qui stocke l'instance du singleton doit être
6     // 'static'.
7     private static field instance: Database
8
9     // Le constructeur du singleton doit toujours être privé
10    // afin d'empêcher les appels à l'opérateur `new`.
11    private constructor Database() is
12        // Code d'initialisation (la connexion au serveur de la
13        // base de données par exemple).
14        // ...
15
```

```
16 // La méthode statique qui contrôle l'accès à l'instance du
17 // singleton.
18 public static method getInstance() is
19     if (Database.instance == null) then
20         acquireThreadLock() and then
21             // Ce thread attend la levée du verrou (lock) le
22             // temps de s'assurer que l'instance n'a pas
23             // déjà été initialisée dans un autre thread.
24     if (Database.instance == null) then
25         Database.instance = new Database()
26     return Database.instance
27
28 // Pour finir, tout singleton doit définir de la logique
29 // métier qui peut être exécutée dans sa propre instance.
30 public method query(sql) is
31     // Par exemple, toutes les requêtes sur la base de
32     // données d'une application passent par cette méthode.
33     // Par conséquent, vous pouvez définir le code des
34     // limitations ou de la mise en cache ici.
35     // ...
36
37 class Application is
38     method main() is
39         Database foo = Database.getInstance()
40         foo.query("SELECT ...")
41         // ...
42         Database bar = Database.getInstance()
43         bar.query("SELECT ...")
44         // La variable `bar` contiendra le même objet que la
45         // variable `foo`.
```

💡 Possibilités d'application

- ⚡ Utilisez le singleton lorsque l'une de vos classes ne doit fournir qu'une seule instance à tous ses clients. Par exemple, une base de données partagée entre toutes les parties d'un programme.
- ⚡ La méthode spéciale de création devient le seul moyen de fabriquer des objets pour la classe, car le singleton désactive les autres. Cette méthode crée un objet ou retourne l'objet existant s'il a déjà été créé.
- ⚡ Utilisez le singleton lorsque vous voulez un contrôle absolu sur vos variables globales.
- ⚡ Contrairement aux variables globales, le singleton garantit l'unicité de l'instance de la classe. Seule la classe singleton peut remplacer l'instance mise en cache.

Vous pouvez moduler le nombre d'instances du singleton comme vous le voulez. Vous devez juste apporter une modification dans le corps de la méthode `getInstance`.

📋 Mise en œuvre

1. Ajoutez un attribut statique à la classe pour stocker l'instance du singleton.
2. Déclarez une méthode de création publique et statique pour récupérer l'instance du singleton.

3. Implémentez une « instantiation paresseuse » (lazy initialization) à l'intérieur de la méthode statique. Elle devrait créer un nouvel objet lors du premier appel et le stocker dans l'attribut statique. La méthode doit retourner cette instance lors de tous les appels suivants.
4. Rendez privé le constructeur de la classe. La méthode statique de la classe doit être la seule à pouvoir appeler le constructeur.
5. Parcourez le code client et remplacez les appels directs au constructeur du singleton par des appels à la méthode statique.

Διδικτικά Avantages et inconvénients

- ✓ Vous garantissez l'unicité de l'instance de la classe.
- ✓ Vous obtenez un point d'accès global à cette instance.
- ✓ l'objet du singleton est uniquement initialisé la première fois qu'il est appelé.
- ✗ Ne respecte pas le *principe de responsabilité unique*. Ce patron résout deux problèmes à la fois.
- ✗ Le singleton peut masquer une mauvaise conception ; il se peut, par exemple, que les composants aient trop de visibilité les uns envers les autres.

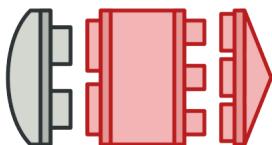
- ✖ Il doit bénéficier d'un traitement spécial pour fonctionner dans un environnement multithread afin que le singleton ne se retrouve pas en plusieurs exemplaires.
- ✖ Les tests unitaires du code client peuvent se révéler difficiles, car de nombreux frameworks reposent sur l'héritage lorsqu'ils créent des objets fictifs. Étant donné que le constructeur de la classe du singleton est privé et que redéfinir la méthode statique est impossible dans la majorité des langages, vous allez devoir être créatif pour reproduire un singleton fictif. Ou ne pas faire de tests. Ou ne pas utiliser de singleton.

➡ Liens avec les autres patrons

- Une classe **Façade** peut souvent être transformée en **Singleton**, car un seul objet façade est en général suffisant.
- Le **Poids mouche** ressemble au **Singleton** si vous parvenez à compiler tous les états partagés des objets en un seul objet poids mouche. Mais ces patrons de conception ont deux différences fondamentales :
 1. Il ne devrait y avoir qu'une seule instance de singleton, mais une classe *poids mouche* peut avoir plusieurs instances avec différents états intrinsèques.
 2. L'objet *singleton* peut être modifiable alors que les objets poids mouche ne sont pas modifiables.
- Les **Fabriques abstraites**, **Monteurs** et **Prototypes** peuvent tous être implémentés comme des **Singlenton**.

Patrons structurels

Les patrons structurels vous guident pour assembler des objets et des classes en de plus grandes structures tout en gardant celles-ci flexibles et efficaces.



Adaptateur

Adapter

Permet de faire collaborer des objets ayant des interfaces normalement incompatibles.



Pont

Bridge

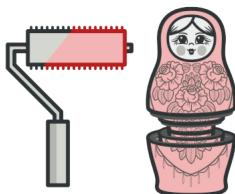
Permet de séparer une grosse classe ou un ensemble de classes connexes en deux hiérarchies – abstraction et implémentation – qui peuvent évoluer indépendamment l'une de l'autre.



Composite

Composite

Permet d'agencer les objets dans des arborescences afin de pouvoir traiter celles-ci comme des objets individuels.



Décorateur

Decorator

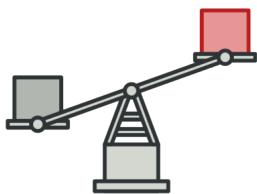
Permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements.



Facade

Facade

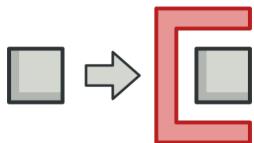
Procure une interface qui offre un accès simplifié à une librairie, un framework ou à n'importe quel ensemble complexe de classes.



Poids mouche

Flyweight

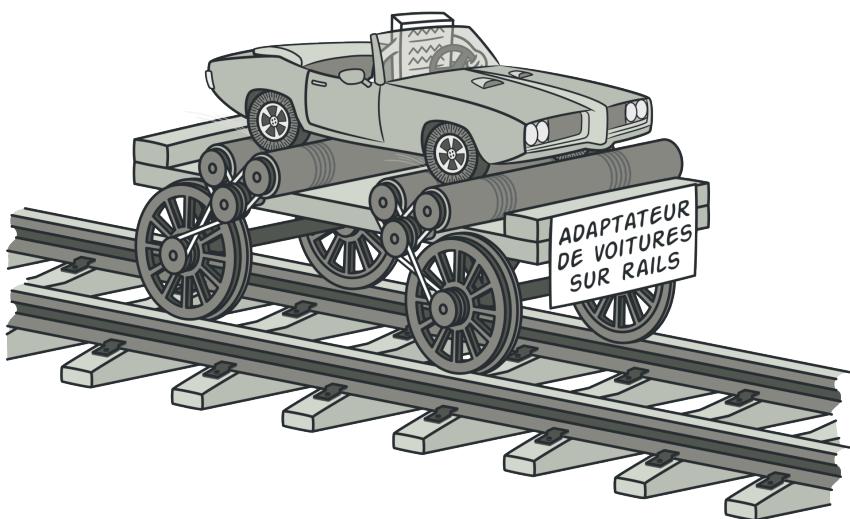
Permet de stocker plus d'objets dans la RAM en partageant les états similaires entre de multiples objets, plutôt que de stocker les données dans chaque objet.



Procuration

Proxy

Permet de fournir un substitut d'un objet. Une procuration donne le contrôle sur l'objet original, vous permettant d'effectuer des manipulations avant ou après que la demande ne lui parvienne.



ADAPTATEUR

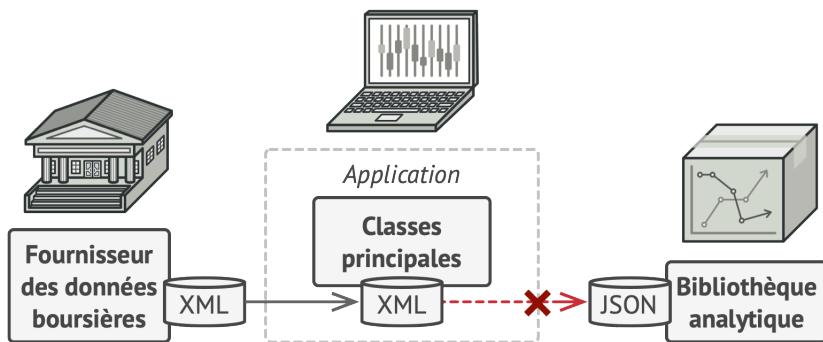
Alias : Wrapper, Adapter

L'**Adaptateur** est un patron de conception structurel qui permet de faire collaborer des objets ayant des interfaces normalement incompatibles.

(:() Problème

Imaginez que vous êtes en train de créer une application de surveillance du marché boursier. L'application télécharge des données de la bourse depuis diverses sources au format XML et affiche ensuite de jolis graphiques et diagrammes destinés à l'utilisateur.

Après un certain temps, vous décidez d'améliorer l'application en intégrant une librairie d'analyse externe. Mais il y a un hic ! Cette librairie ne fonctionne qu'avec des données au format JSON.



Vous ne pouvez pas utiliser la librairie telle qu'elle est actuellement, car elle attend des données incompatibles avec votre application.

Vous pourriez modifier la librairie afin qu'elle accepte du XML, mais vous risquez de faire planter d'autres parties de code qui utilisent déjà cette librairie. Ou alors, vous n'avez tout simplement pas accès au code source de la librairie, rendant la tâche impossible.

Solution

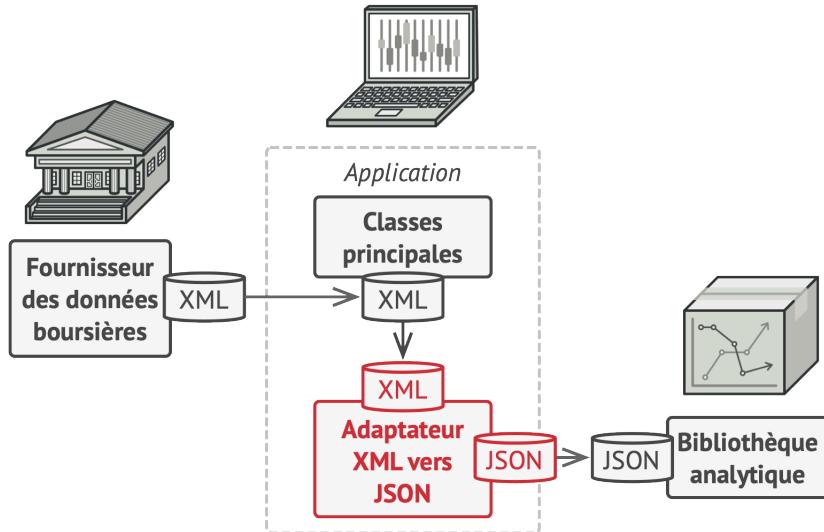
Vous créez un *adaptateur*. C'est un objet spécial qui convertit l'interface d'un objet afin qu'un autre objet puisse le comprendre.

Un adaptateur encapsule un des objets afin de masquer la complexité de la conversion, exécutée à l'ombre des regards. L'objet encapsulé n'a pas conscience de ce que fait l'adaptateur. Par exemple, vous pouvez encapsuler un objet qui calcule en mètres et en kilomètres avec un adaptateur qui effectue la conversion de toutes les données en unités impériales comme les pieds et les milles.

Les adaptateurs peuvent non seulement effectuer des conversions dans différents formats, mais ils peuvent également aider différentes interfaces à collaborer. Le fonctionnement de l'adaptateur est le suivant :

1. L'adaptateur prend une interface compatible avec un des objets existants.
2. L'objet existant peut appeler les méthodes de l'adaptateur via cette interface en toute sécurité.
3. Lorsque l'adaptateur reçoit un appel, il passe la requête au second objet dans un format et dans un ordre qu'il peut interpréter.

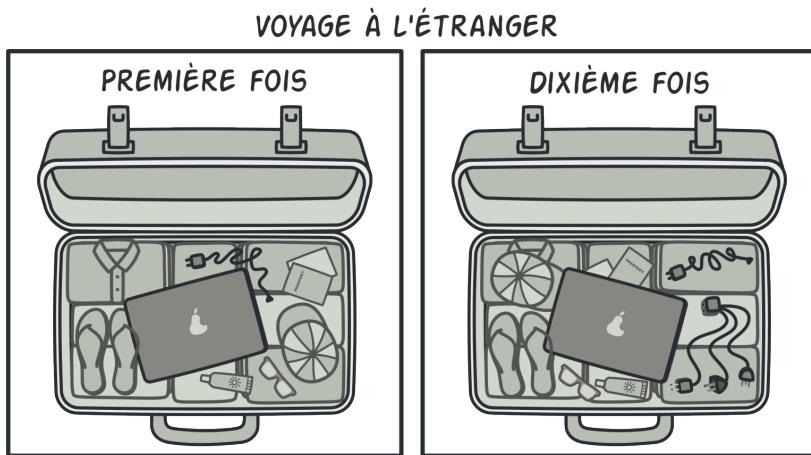
Il est même parfois possible de créer un adaptateur qui peut convertir dans les deux sens !



Retournons à notre application de surveillance du marché boursier. Pour résoudre le problème des formats incompatibles, vous pouvez créer des adaptateurs XML vers JSON pour chaque classe de la librairie que notre code veut utiliser. Vous n'avez plus qu'à ajuster votre code pour communiquer avec la librairie à l'aide de ces adaptateurs. Lorsqu'un adaptateur reçoit un appel, il convertit les données XML en une structure JSON. Il renvoie ensuite l'appel à la méthode appropriée dans un objet d'analyse encapsulé.

🚗 Analogie

Si vous voyagez aux États-Unis pour la première fois, vous allez avoir une petite surprise lorsque vous allez essayer de brancher votre ordinateur portable.



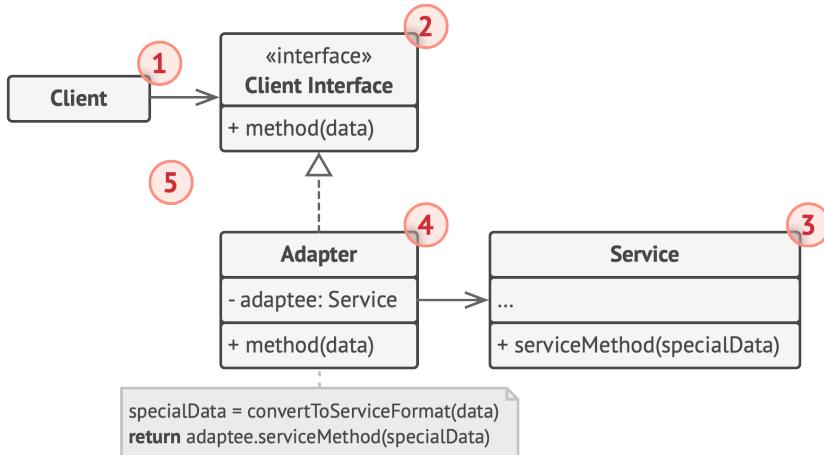
Une valise avant et après un voyage à l'étranger.

Les câbles et prises de courant sont différents dans les autres pays : les câbles français ne rentrent pas dans les prises américaines. Ce problème peut être résolu en utilisant un adaptateur qui accepte un câble européen d'un côté et une prise américaine de l'autre.

Structure

Adaptateur d'objets

Cette implémentation a recours au principe de composition : l'adaptateur implémente l'interface d'un objet et en encapsule un autre. Elle peut être utilisée dans tous les langages de programmation classiques.

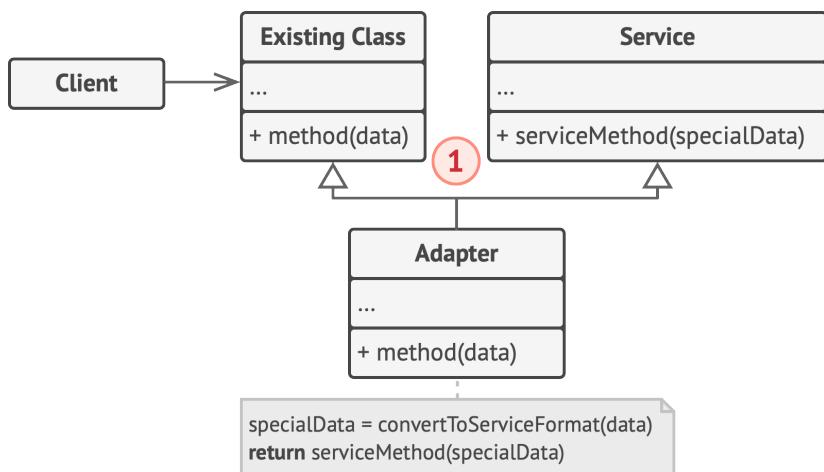


1. Le **Client** est une classe qui contient la logique métier du programme.
2. L'**Interface Client** décrit un protocole que les autres classes doivent implémenter afin de pouvoir collaborer avec le code client.
3. Le **Service** représente une classe que l'on veut utiliser (souvent une application externe ou héritée). Le client ne peut pas l'utiliser directement, car son interface n'est pas compatible.
4. L'**Adaptateur** est une classe qui peut interagir à la fois avec le client et le service : il implémente l'interface client et encapsule l'objet service. L'adaptateur reçoit des appels du client via l'interface client et les convertit en appels à l'objet du service encapsulé, dans un format qu'il peut gérer.

- Le code client n'est pas couplé avec la classe de l'adaptateur concret tant qu'il se contente d'utiliser l'interface du client. Grâce à cela, vous pouvez ajouter de nouveaux types d'adaptateurs dans le programme sans modifier le code client existant. Ce fonctionnement se révèle très pratique si l'interface d'une classe d'un service est modifiée ou remplacée : créez juste une nouvelle classe adaptateur sans toucher au code client.

Adaptateur de classe

Cette implémentation utilise l'héritage : l'adaptateur hérite de l'interface des deux objets en même temps. Vous remarquerez que cette approche ne peut être mise en place que si le langage de programmation gère l'héritage multiple, comme le C++.

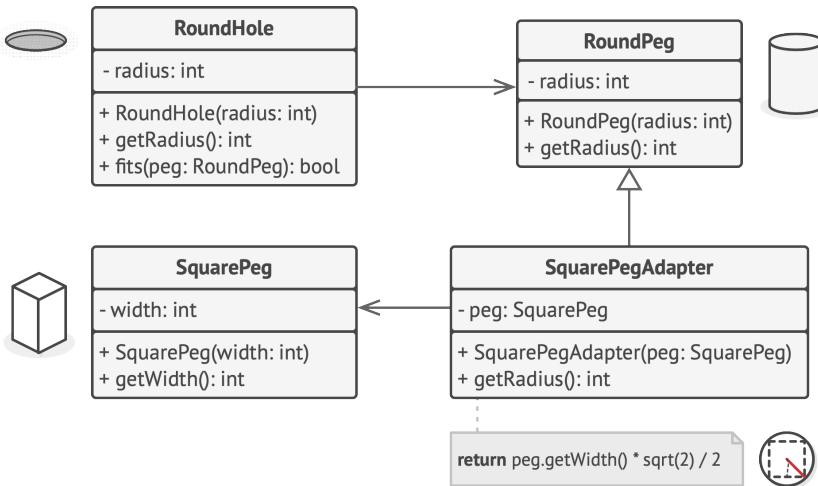


- L'**Adaptateur de Classe** n'a pas besoin d'encapsuler des objets, car il hérite des comportements du client et du service. La to-

talité de l'adaptation se déroule à l'intérieur des méthodes redéfinies. Cet adaptateur peut remplacer une classe existante du client.

Pseudo-code

Voici un exemple d'utilisation du patron de conception **Adaptateur** qui résout le problème classique de la pièce carrée à insérer dans le trou rond.



Adapter des pièces carrées avec des trous ronds.

L'adaptateur se fait passer pour un cylindre, avec un rayon égal à la moitié de la diagonale du carré (en d'autres termes, le rayon minimal du trou pour accueillir la pièce carrée).

```
1 // Prenons deux classes avec des interfaces compatibles :
2 // TrouRond (RoundHole) et PièceRonde (RoundPeg).
3 class RoundHole is
4     constructor RoundHole(radius) { ... }
5
6     method getRadius() is
7         // Retourne le rayon du trou.
8
9     method fits(peg: RoundPeg) is
10        return this.getRadius() >= peg.radius()
11
12 class RoundPeg is
13     constructor RoundPeg(radius) { ... }
14
15     method getRadius() is
16         // Retourne le rayon de la pièce.
17
18
19 // Mais une classe est incompatible : PièceCarrée (SquarePeg).
20 class SquarePeg is
21     constructor SquarePeg(width) { ... }
22
23     method getWidth() is
24         // Retourne la largeur de la pièce carrée.
25
26
27 // Une classe adaptateur permet de faire rentrer des pièces
28 // carrées dans des trous ronds. Elle étend la classe pièceRonde
29 // pour permettre aux objets adaptateur de se comporter comme
30 // des pièces rondes.
31 class SquarePegAdapter extends RoundPeg is
```

```
32 // En réalité, l'adaptateur contient une instance de la
33 // classe pièceCarrée.
34 private field peg: SquarePeg
35
36 constructor SquarePegAdapter(peg: SquarePeg) is
37     this.peg = peg
38
39 method getRadius() is
40     // L'adaptateur se fait passer pour une pièce ronde avec
41     // un rayon qui pourrait faire rentrer la pièce carrée
42     // emballée par l'adaptateur.
43     return peg.getWidth() * Math.sqrt(2) / 2
44
45
46 // Quelque part dans le code client.
47 hole = new RoundHole(5)
48 rpeg = new RoundPeg(5)
49 hole.fits(rpeg) // true
50
51 small_sqpeg = new SquarePeg(5)
52 large_sqpeg = new SquarePeg(10)
53 hole.fits(small_sqpeg) // Ça ne compilera pas (types incompatibles).
54
55 small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
56 large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
57 hole.fits(small_sqpeg_adapter) // true
58 hole.fits(large_sqpeg_adapter) // false
```

Possibilités d'application

-  Utilisez l'adaptateur de classe si vous avez besoin d'une classe existante, mais que son interface est incompatible avec votre code.
-  L'adaptateur permet de créer une classe faisant office de couche intermédiaire. Cette couche sert de convertisseur entre votre code et une classe héritée ou externe, ou n'importe quelle classe avec une interface incongrue.
-  Mettez en place l'adaptateur si vous désirez réutiliser plusieurs sous-classes existantes à qui il manque des fonctionnalités communes qui ne peuvent pas être remontées dans la classe mère.
-  Vous pourriez étendre chaque sous-classe et mettre la fonctionnalité manquante dans les nouvelles sous-classes. En revanche, vous allez devoir dupliquer le code dans ces nouvelles classes, ce qui n'est pas terrible.

Pour une solution un peu plus élégante, vous pouvez mettre la fonctionnalité manquante dans une classe adaptateur. Ensuite, encapsulez les objets avec les fonctionnalités manquantes à l'intérieur de l'adaptateur, les rendant disponibles dynamiquement. Pour que cela fonctionne, les classes ciblées doivent implémenter une interface commune, et l'attribut de l'adaptateur doit implémenter cette interface. Cette solution se rapproche du patron de conception Décorateur.



Mise en œuvre

1. Assurez-vous d'avoir au moins deux classes avec des interfaces incompatibles :
 - Une classe *service* dont vous voulez vous servir, mais que vous ne pouvez pas modifier (application externe, héritée ou dotée d'un grand nombre de dépendances).
 - Une ou plusieurs classes *client* qui pourraient bénéficier de l'utilisation de la classe service.
2. Déclarez l'interface client et décrivez la manière dont les clients vont communiquer avec le service.
3. Créez la classe adaptateur et faites-la implémenter l'interface client. Laissez les méthodes vides pour le moment.
4. Ajoutez un attribut à la classe adaptateur pour y mettre une référence vers l'objet service. En général on initialise cet attribut à l'aide du constructeur, mais il est parfois plus pratique de l'envoyer à l'adaptateur au moment de l'appel de ses méthodes.
5. Implémentez toutes les méthodes de l'interface client une par une dans la classe adaptateur. L'adaptateur doit déléguer le gros du travail à l'objet service et ne s'occuper que de l'interface ou de la conversion du format des données.

6. Les clients doivent utiliser l'adaptateur en passant par l'interface client. Vous pouvez ainsi modifier ou étendre les adaptateurs sans toucher au code client.

⚠️ Avantages et inconvénients

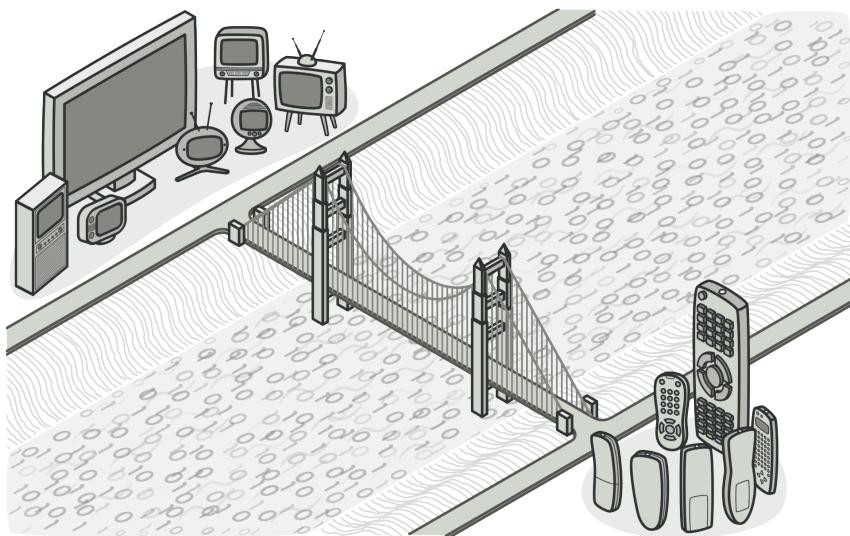
- ✓ *Principe de responsabilité unique.* Vous découpez l'interface ou le code de conversion des données, de la logique métier du programme.
- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouveaux types d'adaptateurs dans le programme sans modifier le code client existant. Ces adaptateurs doivent forcément passer par l'interface du client.
- ✗ La complexité générale du code augmente, car vous devez créer un ensemble de nouvelles classes et interfaces. Parfois, il est plus simple de modifier la classe du service afin de la faire correspondre avec votre code.

➡️ Liens avec les autres patrons

- L'**Adaptateur** fournit une interface complètement différente pour accéder à un objet existant. En revanche, avec le modèle du **Décorateur**, l'interface reste la même ou est étendue. De plus, *Décorateur* supporte la composition récursive, ce qui n'est pas possible lorsque vous utilisez *Adaptateur*.
- Avec **Adaptateur**, vous accédez à un objet existant via une interface différente. Avec **Procuration**, l'interface reste la même.

Avec **Décorateur**, vous accédez à l'objet via une interface améliorée.

- La **Façade** définit une nouvelle interface pour les objets existants, alors que l'**Adaptateur** essaye de rendre l'interface existante utilisable. *L'adaptateur* emballe généralement un seul objet alors que la *façade* s'utilise pour un sous-système complet d'objets.
- Le **Pont**, l'**État**, la **Stratégie** (et dans une certaine mesure l'**Adaptateur**) ont des structures très similaires. En effet, ces patrons sont basés sur la composition, qui délègue les tâches aux autres objets. Cependant, ils résolvent différents problèmes. Un patron n'est pas juste une recette qui vous aide à structurer votre code d'une certaine manière. C'est aussi une façon de communiquer aux autres développeurs le problème qu'il résout.



PONT

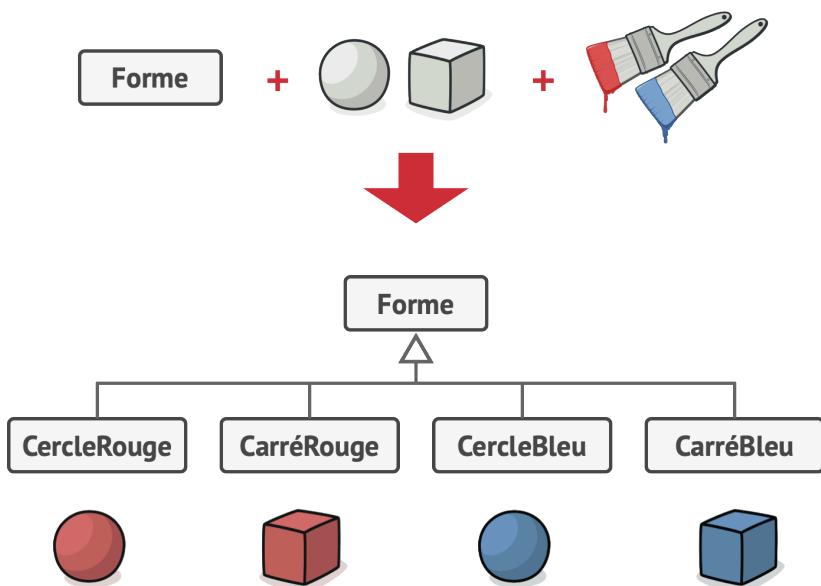
Alias : Bridge

Le **Pont** est un patron de conception structurel qui permet de séparer une grosse classe ou un ensemble de classes connexes en deux hiérarchies – abstraction et implémentation – qui peuvent évoluer indépendamment l'une de l'autre.

Problème

Abstraction? Implémentation? Ces termes vous donnent des frissons? Rassurez-vous, nous allons prendre un exemple simple.

Prenons une classe de `Forme` géométrique avec les sous-classes suivantes : `Cercle` et `Carré`. Vous voulez étendre cette hiérarchie de classes pour incorporer des couleurs, vous créez donc des sous-classes de formes : `Rouge` et `Bleu`. Mais vous avez déjà deux sous-classes, vous devez donc créer quatre combinaisons comme par exemple `CercleBleu` et `CarréRouge`.



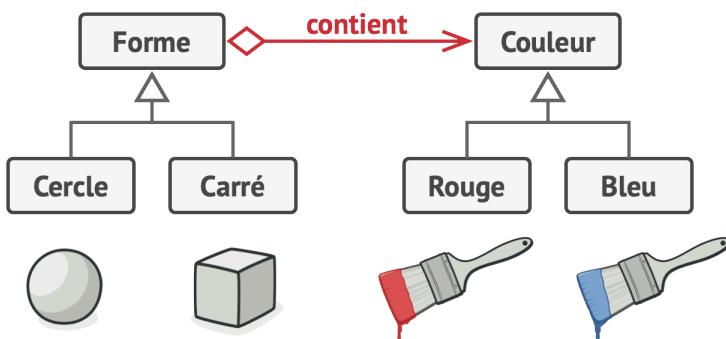
Le nombre de combinaisons augmente exponentiellement.

Ajouter de nouvelles formes et couleurs va augmenter la taille de la hiérarchie exponentiellement. Par exemple, pour ajouter une forme triangle, vous devez créer deux nouvelles sous-classes : une par couleur. Ensuite, ajouter une couleur demandera trois nouvelles sous-classes : une pour chaque forme. Si l'on continue ainsi, la situation ne fait qu'empirer.

😊 Solution

Nous rencontrons ce problème, car nous essayons d'étendre les classes forme dans deux dimensions indépendantes : la forme et la couleur. C'est un problème classique causé par l'héritage.

Le pont tente de résoudre ce problème en utilisant la composition à la place de l'héritage. Pour ce faire, vous insérez une des dimensions dans une hiérarchie de classes séparée afin que la classe originale puisse référencer un objet de cette nouvelle hiérarchie, plutôt que de réunir tous les états et comportements à l'intérieur d'une même classe.



Vous évitez l'explosion de la hiérarchie de classes en la transformant en plusieurs hiérarchies connexes.

Nous allons appliquer ce procédé et récupérer le code concernant les couleurs dans sa propre classe avec deux sous-classes : `Rouge` et `Bleu`. La classe `Forme` est ensuite dotée d'un attribut qui référence l'un des objets couleur. La forme peut maintenant déléguer tous les traitements concernant la couleur de l'objet. Cette référence agira comme un pont entre les classes `Forme` et `Couleur`. Dorénavant, l'ajout de nouvelles couleurs ne bouleversera plus la hiérarchie des formes et inversement.

Abstraction et implémentation

Le livre du GoF¹ ajoute les termes *abstraction* et *implémentation* à la définition du pont. Trop académiques, ces termes rendent le patron plus compliqué qu'il ne l'est en réalité. En gardant en tête l'exemple avec les formes et couleurs, déchiffrons la signification de ces termes barbares.

L'*abstraction* (aussi appelée *interface*) est une couche de contrôle de haut niveau pour une entité. Cette couche n'est pas censée effectuer de traitements toute seule. Elle doit déléguer le travail à la couche *implémentation* (appelée également *plateforme*).

1. « La bande des quatre » (Gang of Four/GoF) est le surnom donné aux quatre auteurs du livre sur les patrons de conception : *Design patterns. Catalogue de modèles de conception réutilisables* <https://refactoring.guru/fr/gof-book>.

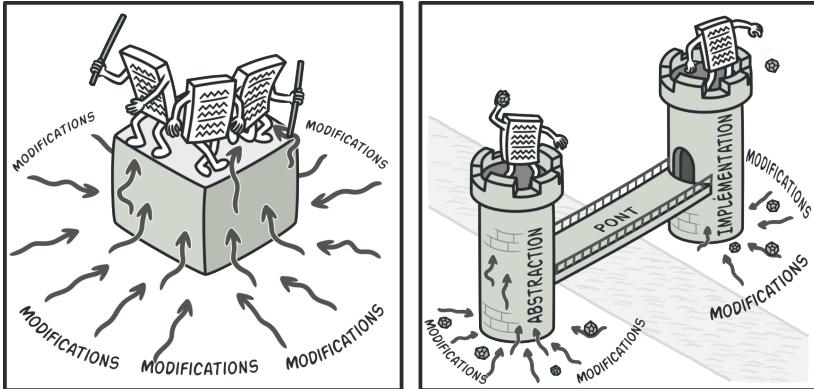
Notez bien qu'il ne s'agit pas des *interfaces* ou des *classes abstraites* d'un langage de programmation.

Si l'on prend un logiciel comme exemple concret, l'interface utilisateur graphique (GUI) prend le rôle de l'abstraction et le code du système d'exploitation (API) prend le rôle de l'implémentation que la couche GUI appelle en réponse aux interactions de l'utilisateur.

D'une manière générale, vous pouvez développer un tel programme en deux parties indépendantes :

- Disposer de plusieurs GUI différentes (on lance celle qui est adaptée à l'utilisateur, client ou admin).
- Gérer plusieurs API différentes (pouvoir exécuter l'application sous Windows, Linux et macOS).

Dans le pire des cas, cette application pourrait ressembler à un énorme plat de spaghetti avec des centaines de conditions connectant différents types de GUI et d'API, réparties un peu partout à travers le code.

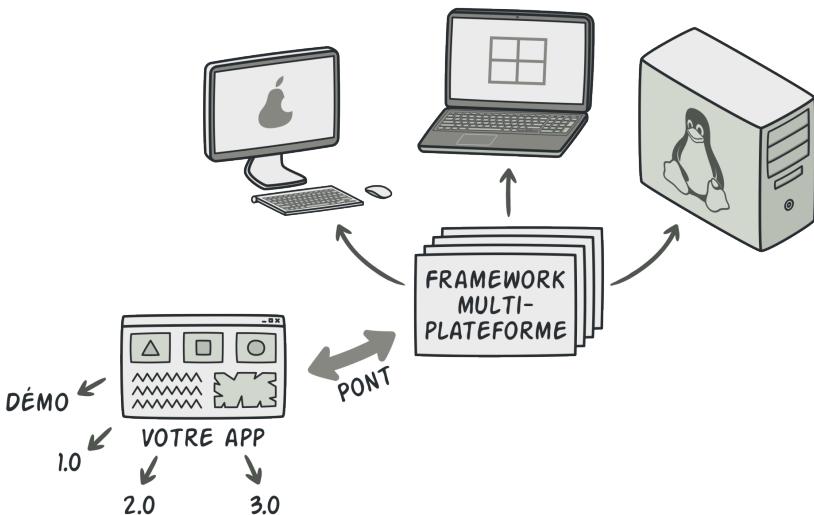


La moindre modification dans une architecture monolithique peut se révéler compliquée, car vous devez comprendre la totalité du code. Vous pouvez facilement effectuer des modifications dans de plus petits modules bien définis.

Vous pouvez mettre de l'ordre dans ce chaos en rangeant le code concernant les combinaisons spécifiques de l'interface/plateforme dans des classes séparées, mais vous découvrirez rapidement que ces classes sont *nombreuses*. La hiérarchie des classes croît rapidement, car l'ajout d'une nouvelle GUI ou l'intégration d'une nouvelle API requièrent la création de classes supplémentaires.

Tentons de résoudre ce problème avec le pont. Il nous propose de mettre les classes dans deux hiérarchies :

- Abstraction : la couche GUI de l'application.
- Implémentation : les API des systèmes d'exploitation.

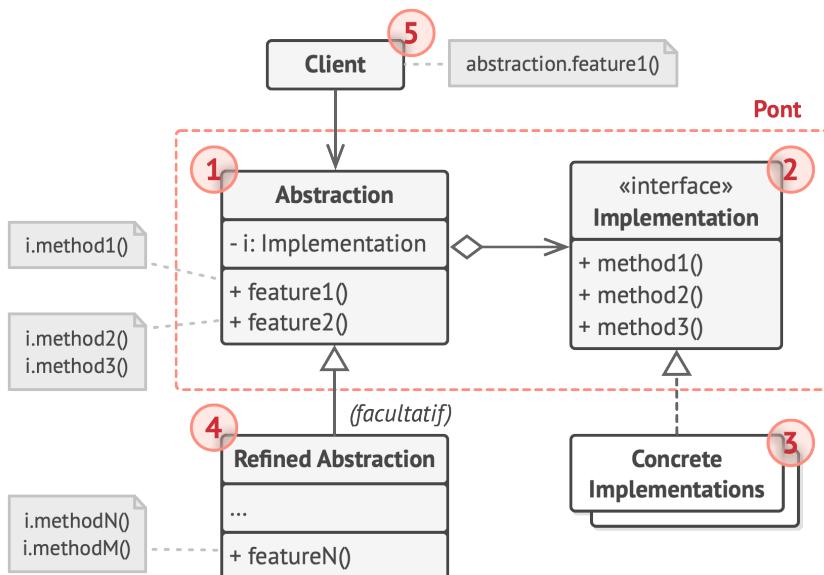


Une des techniques pour organiser une application multiplateforme.

L'objet abstraction contrôle l'apparence de l'application et délégue la partie métier à l'objet d'implémentation correspondant. Les implémentations sont interchangeables tant qu'elles implémentent la même interface, ce qui permet à la même GUI de fonctionner aussi bien sous Windows que sous Linux.

Grâce à cela, vous pouvez modifier les classes de la GUI sans toucher aux classes des API. De plus, adapter le code pour gérer un autre système d'exploitation ne requiert que l'ajout d'une sous-classe dans la hiérarchie de l'implémentation.

Structure



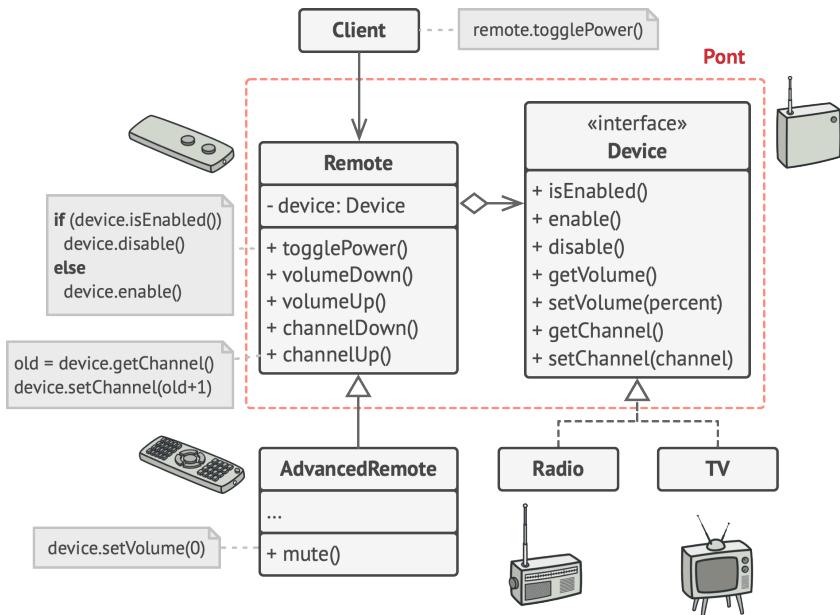
1. L'**Abstraction** offre une logique de contrôle de haut niveau. Elle compte sur l'objet de l'implémentation pour s'occuper des tâches de bas niveau.
2. L'**Implémentation** déclare une interface commune pour toutes les implémentations concrètes. L'abstraction ne peut communiquer avec les objets de l'implémentation que grâce aux méthodes qui y sont déclarées.

L'abstraction peut contenir les mêmes méthodes que l'implémentation, mais en général l'abstraction déclare des comportements complexes qui reposent sur une grande variété d'opérations primitives déclarées par l'implémentation.

3. Les **Implémentations Concrètes** contiennent du code spécialisé pour les plateformes.
4. L'**Abstraction Fine** procure des variantes pour la logique de contrôle. Tout comme leur parent, elles travaillent avec différentes implémentations en passant par l'interface d'implémentation principale.
5. En général, le **Client** ne veut interagir qu'avec l'abstraction, mais c'est son rôle de faire correspondre l'objet d'abstraction avec un des objets d'implémentation.

Pseudo-code

Cet exemple montre comment le **Pont** aide à diviser le code monolithique d'une application qui gère les appareils et leurs télécommandes. Les `Appareils` prennent le rôle de l'implémentation et les `Télécommandes` font office d'abstraction.



La hiérarchie de la classe originale est divisée en deux parties : appareils et télécommandes.

La classe de base télécommande déclare un attribut de référence qui la lie avec un objet appareil. Toutes les télécommandes utilisent l'interface principale des appareils, ce qui leur permet de fonctionner avec tous les types d'appareils.

Vous pouvez faire évoluer les télécommandes indépendamment des appareils, vous devez juste créer une nouvelle sous-classe de télécommande. Par exemple, une télécommande basique pourrait juste avoir deux boutons, mais vous pouvez lui rajouter des fonctionnalités comme une batterie supplémentaire ou un écran tactile.

Le code client établit le lien entre le type de télécommande désiré et un appareil spécifique en passant par le constructeur de la télécommande.

```
1 // L'« abstraction » définit l'interface pour la partie
2 // « télécommande » des deux hiérarchies de classes. Elle garde
3 // une référence sur un objet de la hiérarchie de
4 // l'« implémentation » et lui délègue les tâches.
5 class RemoteControl is
6     protected field device: Device
7     constructor RemoteControl(device: Device) is
8         this.device = device
9     method togglePower() is
10        if (device.isEnabled()) then
11            device.disable()
12        else
13            device.enable()
14     method volumeDown() is
15        device.setVolume(device.getVolume() - 10)
16     method volumeUp() is
17        device.setVolume(device.getVolume() + 10)
18     method channelDown() is
19        device.setChannel(device.getChannel() - 1)
20     method channelUp() is
21        device.setChannel(device.getChannel() + 1)
22
23
24 // Vous pouvez étendre les classes de la hiérarchie de
25 // l'abstraction indépendamment des classes appareil.
26 class AdvancedRemoteControl extends RemoteControl is
27     method mute() is
```

```
28     device.setVolume(0)
29
30
31 // L'interface de l'implémentation déclare les méthodes communes
32 // à toutes les classes concrètes de l'implémentation. Elle n'a
33 // pas besoin de correspondre à l'interface de l'abstraction. En
34 // fait, les deux interfaces peuvent être complètement
35 // différentes. En général, l'interface de l'implémentation ne
36 // fournit que des opérations primitives, alors que
37 // l'abstraction définit des opérations de plus haut niveau
38 // basées sur ces primitives.
39 interface Device is
40     method isEnabled()
41     method enable()
42     method disable()
43     method getVolume()
44     method setVolume(percent)
45     method getChannel()
46     method setChannel(channel)
47
48
49 // Tous les appareils suivent la même interface.
50 class Tv implements Device is
51     // ...
52
53 class Radio implements Device is
54     // ...
55
56
57 // Quelque part dans le code client.
58 tv = new Tv()
59 remote = new RemoteControl(tv)
```

```
60 remote.togglePower()  
61  
62 radio = new Radio()  
63 remote = new AdvancedRemoteControl(radio)
```

💡 Possibilités d'application

- ⚡ Utilisez le pont dans les situations où vous souhaitez diviser et organiser une classe monolithique composée de plusieurs variantes d'une fonctionnalité (par exemple, si la classe fonctionne avec différents serveurs de base de données).
- ⚡ Plus une classe grandit, plus il est difficile de comprendre son fonctionnement et plus les modifications prennent du temps. Les modifications apportées à l'une des variantes de la fonctionnalité vont demander des changements dans toute la classe, ce qui risque de créer des erreurs ou de provoquer des effets de bord critiques.

Le pont vous permet de diviser la classe monolithique en plusieurs hiérarchies de classes. Ensuite, les classes d'une hiérarchie peuvent être modifiées indépendamment des classes des autres hiérarchies. La maintenance du code devient ainsi plus simple et minimise les risques de bugs.

- ⚡ Utilisez le pont si vous voulez étendre une classe dans plusieurs dimensions orthogonales (indépendantes).

- ⚡ Le pont vous propose de construire une hiérarchie de classes séparée pour chaque dimension. La classe d'origine délègue la tâche aux objets de ces hiérarchies plutôt que de tout faire par elle-même.

- ⚡ Utilisez ce patron si vous voulez être en mesure de changer d'implémentation dès le lancement de l'application.

- ⚡ Grâce à ce patron, l'objet de l'implémentation peut être déplacé à l'intérieur de l'abstraction. Cette manipulation n'est pas obligatoire, mais elle est aussi simple à mettre en place que de donner une valeur à un attribut.

J'en profite pour vous informer que ce dernier point pousse souvent les développeurs à confondre le pont et la Stratégie. Rappelez-vous bien qu'un patron n'est pas seulement une manière de structurer vos classes, c'est aussi un moyen de communiquer votre intention ou de répondre à un problème.

⌚ Mise en œuvre

1. Identifiez les dimensions orthogonales de vos classes. Ces concepts indépendants peuvent représenter les couples suivants : abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.

2. Déterminez les opérations que le client veut utiliser et définissez-les dans la classe d'abstraction de base.

3. Établissez la liste des opérations disponibles sur toutes les plateformes. L'abstraction a besoin de certaines de ces opérations : déclarez-les dans l'interface de l'implémentation principale.
4. Créez des classes d'implémentations concrètes pour chaque plateforme de votre domaine, et assurez-vous qu'elles implémentent l'interface de l'implémentation.
5. Ajoutez un attribut de référence pour le type de l'implémentation à l'intérieur de la classe abstraction. Cette dernière délègue la majorité des tâches à l'objet implémentation référencé dans cet attribut.
6. Si vous avez plusieurs variantes de logique de haut niveau, créez des abstractions fines pour chacune d'entre elles en étendant la classe de base abstraction.
7. Le code client doit en principe passer un objet d'implémentation au constructeur de l'abstraction afin d'associer les deux. Ensuite, le client n'a plus besoin de s'occuper de l'implémentation et peut se contenter de travailler uniquement avec l'abstraction.

Avantages et inconvénients

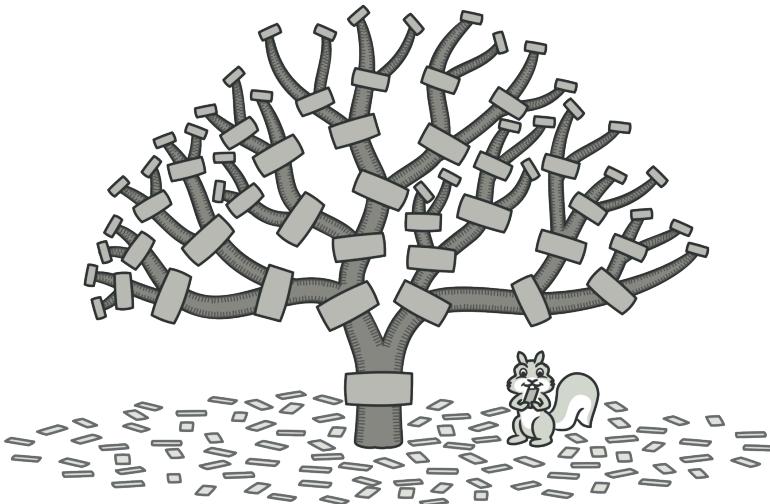
- ✓ Vous pouvez créer des classes et des applications multiplate-formes.

- ✓ Le code client manipule des abstractions de haut niveau. Il n'est pas dépendant des détails de la plateforme.
- ✓ *Principe ouvert/fermé*. Vous pouvez introduire de nouvelles abstractions et implémentations indépendamment les unes des autres.
- ✓ *Principe de responsabilité unique*. Vous pouvez vous concentrer sur la logique de haut niveau dans l'abstraction, et sur les détails de la plateforme dans l'implémentation.
- ✗ Le code va devenir plus compliqué si vous introduisez ce patron dans une classe très cohésive.

↔ Liens avec les autres patrons

- Le **Pont** est habituellement mis en place durant la conception, ce qui vous permet de développer les différentes parties de l'application indépendamment. L'**adaptateur** quant à lui est plus souvent utilisé dans une application existante pour permettre à des classes normalement incompatibles de fonctionner ensemble.
- Le **Pont**, **l'État**, la **Stratégie** (et dans une certaine mesure **l'Adaptateur**) ont des structures très similaires. En effet, ces patrons sont basés sur la composition, qui délègue les tâches aux autres objets. Cependant, ils résolvent différents problèmes. Un patron n'est pas juste une recette qui vous aide à structurer votre code d'une certaine manière. C'est aussi une façon de communiquer aux autres développeurs le problème qu'il résout.

- Vous pouvez utiliser la **Fabrique abstraite** avec le **Pont**. Ce couple est très utile quand les abstractions définies par le *pont* ne fonctionnent qu'avec certaines implémentations spécifiques. Dans ce cas, la *fabrique abstraite* peut encapsuler ces relations et cacher la complexité au code client.
- Vous pouvez combiner le **Monteur** avec le **Pont** : la classe directeur joue le rôle de l'abstraction, et les différents *monteurs* prennent le rôle des implémentations.



COMPOSITE

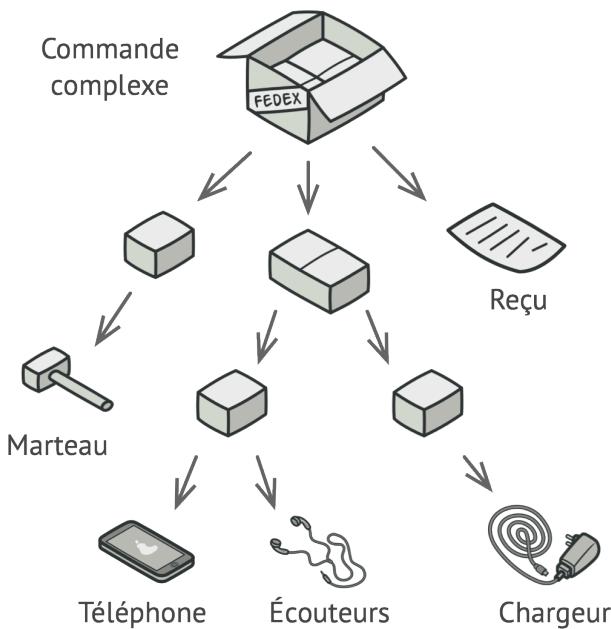
Alias : Arbre d'objets

Composite est un patron de conception structurel qui permet d'agencer les objets dans des arborescences afin de pouvoir traiter celles-ci comme des objets individuels.

(:() Problème

L'utilisation de ce patron doit être réservée aux applications dont la structure principale peut être représentée sous la forme d'une arborescence.

Prenons les deux objets suivants : `Produits` et `Boîtes`. Une boîte peut contenir plusieurs produits ainsi qu'un certain nombre de boîtes plus petites. Ces petites boîtes peuvent également contenir quelques produits ou même d'autres boîtes encore plus petites, et ainsi de suite.



Une commande peut contenir divers produits empaquetés à l'intérieur de boîtes, elles-mêmes rangées dans de plus grosses boîtes, etc. La structure complète ressemble à un arbre inversé.

Vous décidez de mettre au point un système de commandes qui utilise ces classes. Les commandes peuvent être composées de produits simples sans emballage, d'autres boîtes remplies de produits... et d'autres boîtes. Comment allez-vous déterminer le coût total d'une telle commande ?

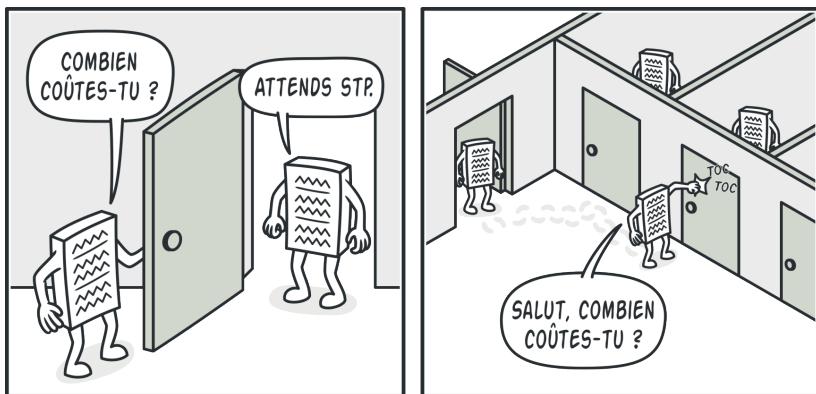
Vous pouvez tenter l'approche directe qui consiste à déballer toutes les boîtes, prendre chaque produit et en faire la somme pour obtenir le total. Ce mode de calcul peut facilement se mettre en place dans le monde réel mais dans un programme, ce n'est pas aussi simple que de créer une boucle. Il faut connaître à l'avance la classe des `Produits` et des `Boîtes` que l'on parcourt, le niveau d'imbrication des boîtes ainsi que d'autres détails. Tout ceci rend l'approche directe assez compliquée et même parfois impossible.

😊 Solution

Le patron de conception composite vous propose de manipuler les `Produits` et les `Boîtes` à l'aide d'une interface qui déclare une méthode de calcul du prix total.

Comment cette méthode peut-elle fonctionner ? Pour un produit, on retourne simplement son prix. Pour une boîte, on parcourt chacun de ses objets, on leur demande leur prix, puis on retourne un total pour la boîte. Si l'un de ces objets est une boîte plus petite, cette dernière va aussi parcourir son propre contenu et ainsi de suite, jusqu'à ce que tous les prix aient été

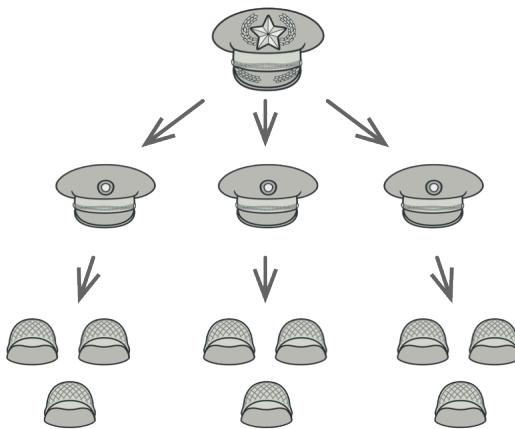
calculés. Une boîte peut même ajouter des frais supplémentaires, comme le prix de l'emballage.



Le patron de conception composite emploie une méthode récursive afin de parcourir tous les composants d'une arborescence.

La cerise sur le gâteau est que vous n'avez même pas besoin de connaître la classe concrète des objets de l'arborescence. Vous n'avez pas besoin de savoir si un objet est un produit tout simple ou une boîte sophistiquée, vous les manipulez de la même manière grâce à une interface commune. Lorsque vous faites appel à une méthode, les objets s'occupent de faire transiter la requête en descendant vers les feuilles de l'arbre.

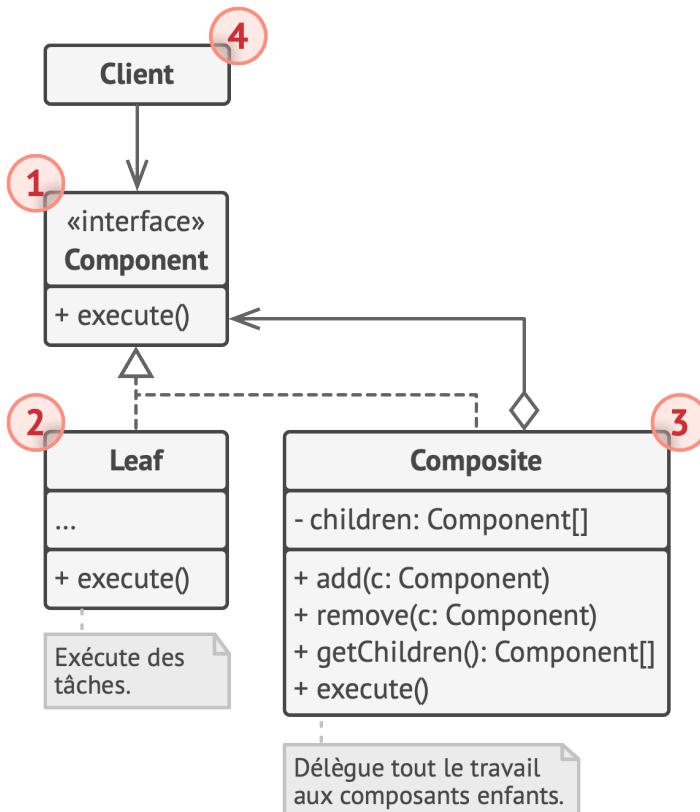
🚗 Analogie



Un exemple de structure militaire.

En général, les armées d'un pays sont structurées en hiérarchies. Une armée comporte plusieurs divisions, une division est composée de brigades, une brigade est composée de compagnies, qui peuvent elles-mêmes être divisées en escouades. Pour finir, une escouade est un petit groupe de soldats. Les ordres sont donnés au sommet de la hiérarchie et passés au niveau directement inférieur à chaque soldat, qui sait quoi en faire.

Structure



1. L'interface **Composant** décrit les opérations communes aux objets simples et complexes de l'arborescence.
2. Une **Feuille** est un élément de base d'une branche qui n'a pas de sous-élément.

En général les feuilles font le plus gros du travail, car elles n'ont personne à qui le déléguer.

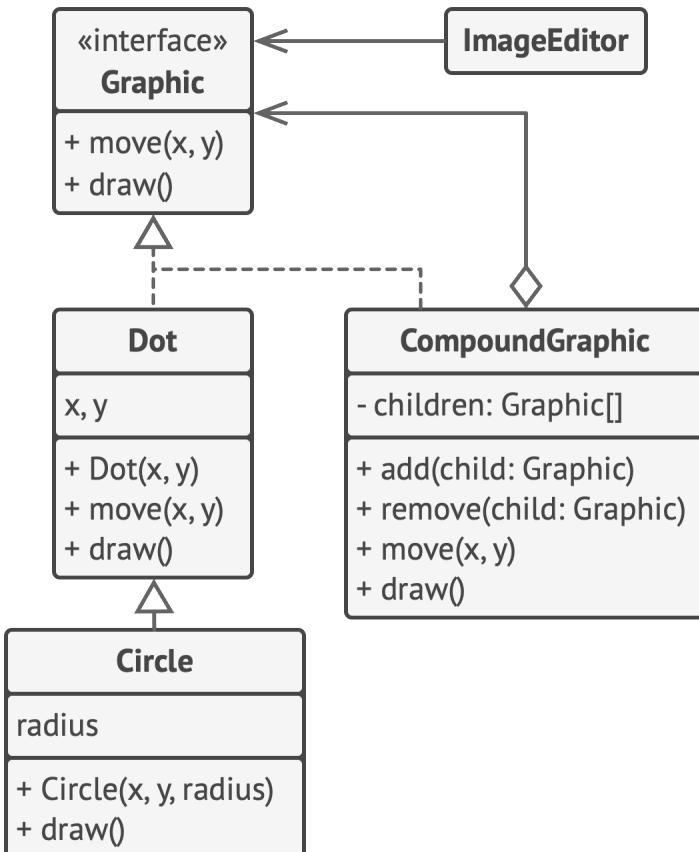
3. Le **Conteneur** (alias *composite*) est un élément composé de sous-éléments : des feuilles ou d'autres conteneurs. Un conteneur ne connaît pas les classes de ses enfants. Il passe par l'interface composant pour interagir avec ses sous-éléments.

Lorsqu'il reçoit une requête, un conteneur délègue la tâche à ses sous-éléments, traite les résultats intermédiaires, puis renvoie le résultat final au client.

4. Le **Client** manipule les éléments depuis l'interface composant, ce qui lui permet de fonctionner de la même manière pour les éléments simples et complexes de l'arborescence.

Pseudo-code

Dans cet exemple, le patron de conception **Composite** nous permet de gérer des imbriques de formes géométriques dans un éditeur graphique.



Exemple de l'éditeur des formes géométriques.

La classe **CompositionGraphique** est un conteneur doté d'un certain nombre de formes, incluant même des formes composées. Une forme composée possède les mêmes méthodes qu'une forme simple. Mais plutôt que de tout gérer toute seule, une forme composée envoie une requête récursive à tous ses enfants et « totalise » le résultat.

Le code client manipule ces formes en passant par l'interface commune. De ce fait, le client ne sait jamais s'il est en train de

manipuler une forme ou une composition. Le client peut manipuler des structures d'objets très complexes sans jamais être couplé avec les classes concrètes de cette structure.

```
1 // L'interface du composant déclare des opérations communes pour
2 // les objets simples et complexes d'une composition.
3 interface Graphic is
4     method move(x, y)
5     method draw()
6
7 // La classe feuille représente les objets finaux d'une
8 // composition. Un objet feuille ne peut pas avoir de sous-
9 // objets. En général, ce sont les feuilles qui lancent les
10 // traitements. Les objets composite ne font que déléguer le
11 // travail à leurs sous-composants.
12 class Dot implements Graphic is
13     field x, y
14
15     constructor Dot(x, y) { ... }
16
17     method move(x, y) is
18         this.x += x, this.y += y
19
20     method draw() is
21         // Dessine un point aux coordonnées X et Y.
22
23 // Toutes les classes composant peuvent étendre d'autres
24 // composants.
25 class Circle extends Dot is
26     field radius
27
```

```
28  constructor Circle(x, y, radius) { ... }
29
30  method draw() is
31      // Trace un cercle de rayon R aux coordonnées X et Y.
32
33  // La classe composite représente les composants complexes qui
34  // peuvent avoir des enfants. Les objets composite délèguent
35  // généralement les tâches à leurs enfants et « additionnent »
36  // ensuite le résultat.
37 class CompoundGraphic implements Graphic is
38     field children: array of Graphic
39
40     // Un composite peut ajouter ou retirer d'autres composants
41     // (simples ou complexes) de la liste de ses enfants.
42     method add(child: Graphic) is
43         // Ajoute un enfant au tableau d'enfants.
44
45     method remove(child: Graphic) is
46         // Retire un enfant du tableau d'enfants.
47
48     method move(x, y) is
49         foreach (child in children) do
50             child.move(x, y)
51
52     // Un composite exécute sa logique principale d'une certaine
53     // manière : il parcourt récursivement tous ses enfants,
54     // puis récupère et additionne leurs résultats. L'objet est
55     // entièrement parcouru, car les enfants du composite
56     // passent ces appels à leurs propres enfants et ainsi de
57     // suite.
58     method draw() is
59         // 1. Pour chaque composant enfant :
```

```
60      //      – Dessine le composant.  
61      //      – Met à jour le rectangle de délimitation.  
62      // 2. Dessine un rectangle en pointillé en utilisant les  
63      // coordonnées de la délimitation.  
64  
65  
66  // Le code client manipule les composants grâce à leur interface  
67  // de base. Ainsi, le code client peut aussi bien prendre en  
68  // charge les composants simples que les complexes.  
69  class ImageEditor is  
70      field all: CompoundGraphic  
71  
72  method load() is  
73      all = new CompoundGraphic()  
74      all.add(new Dot(1, 2))  
75      all.add(new Circle(5, 3, 10))  
76      // ...  
77  
78  // Combine les composants sélectionnés en un seul composant  
79  // complexe.  
80  method groupSelected(components: array of Graphic) is  
81      group = new CompoundGraphic()  
82      foreach (component in components) do  
83          group.add(component)  
84          all.remove(component)  
85      all.add(group)  
86      // Tous les composants vont être dessinés.  
87      all.draw()
```

Possibilités d'application

-  Utilisez le composite si vous devez gérer une structure d'objets qui ressemble à une arborescence.
-  Le patron de conception composite vous propose deux éléments de base qui partagent la même interface : des feuilles simples et des conteneurs complexes. Un conteneur peut être composé de feuilles et d'autres conteneurs. Grâce à cela, vous pouvez construire une structure récursive composée d'objets imbriqués qui ressemble à un arbre.
-  Utilisez ce patron si vous voulez que le client interagisse avec les éléments simples aussi bien que complexes de façon uniforme.
-  Tous les éléments définis dans le patron composite partagent une interface commune. En utilisant cette interface, le client n'a pas besoin de connaître la classe concrète des objets qu'il manipule.

Mise en œuvre

1. Assurez-vous que votre application possède bien la forme d'une arborescence. Décomposez-la en conteneurs et en éléments simples. Rappelez-vous que les conteneurs doivent pouvoir accueillir des éléments simples et d'autres conteneurs.

2. Déclarez l'interface composant avec une liste de méthodes qui fonctionnent à la fois avec les composants simples et complexes.
3. Créez une classe feuille pour représenter les éléments simples. Un même programme peut avoir plusieurs classes feuille différentes.
4. Créez une classe conteneur pour représenter les éléments complexes. Fournissez un attribut de type tableau à cette classe, afin de stocker les références aux sous-éléments. Ce tableau doit pouvoir stocker les feuilles et les conteneurs, assurez-vous donc qu'il est bien déclaré avec le type d'interface du composant.

Tout en implémentant les méthodes de l'interface composant, gardez en tête que les conteneurs sont censés déléguer la majeure partie du travail à leurs sous-éléments.

5. Enfin, définissez des méthodes pour ajouter ou retirer des éléments enfants du conteneur.

Elles peuvent être placées à l'intérieur de l'interface composant, mais ceci ne respecte pas le *principe de ségrégation des interfaces*, car la classe feuille contiendra des méthodes vides. L'avantage est que le client pourra traiter ces éléments uniformément, même lorsqu'il construit l'arborescence.

⚠️ Avantages et inconvénients

- ✓ Vous pouvez travailler dans des structures arborescentes complexes plus facilement en utilisant les avantages du polymorphisme et de la récursivité.
- ✓ *Principe ouvert/fermé.* Vous pouvez introduire de nouveaux types d'éléments dans l'application qui pourront directement être intégrés dans l'arborescence, sans avoir à réécrire l'existant.
- ✗ Vous rencontrerez parfois des difficultés pour définir une interface commune à certaines classes dont les fonctionnalités sont trop différentes. Dans certains scénarios, vous devez créer une interface composant bien trop générique, rendant le fonctionnement difficile à comprendre.

➡️ Liens avec les autres patrons

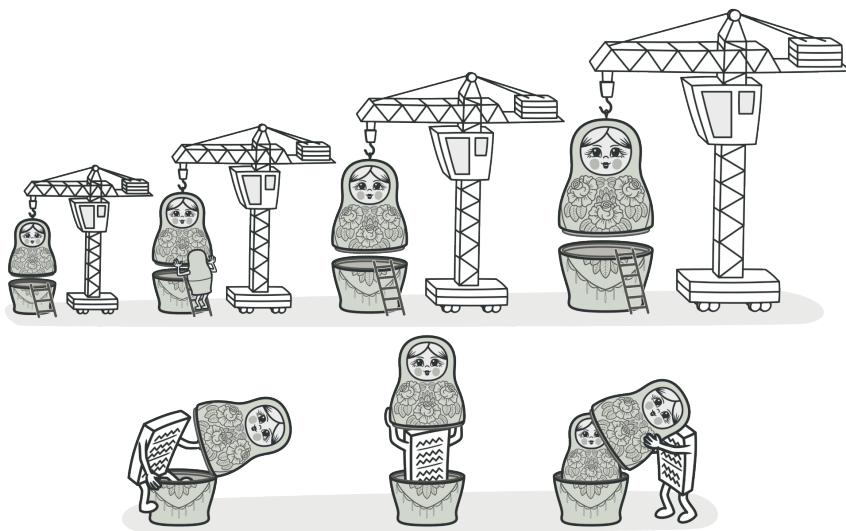
- Vous pouvez utiliser le **Monteur** lorsque vous créez des arbres **Composites** complexes, car vous pouvez programmer les étapes de la construction récursivement.
- La **Chaîne de Responsabilité** est souvent utilisée en conjonction avec le **Composite**. Dans ce cas, lorsqu'une feuille reçoit une demande, elle la passe le long de la chaîne de ses composants parent, jusqu'à la racine de l'arborescence.
- Vous pouvez utiliser les **Itérateurs** pour parcourir des arbres **Composites**.

- Vous pouvez utiliser le **Visiteur** pour lancer une opération sur un arbre **Composite** entier.
- Vous pouvez transformer des nœuds de feuilles de l'arbre **Composite** en **Poids mouches** et les partager pour économiser de la RAM.
- Le **Composite** et le **Décorateur** ont des diagrammes de structure similaires puisqu'ils reposent sur la composition récursive pour organiser un nombre variable d'objets.

Un *décorateur* est comme un *composite*, mais avec un seul composant enfant. Il y a une autre différence importante : Le *décorateur* ajoute des responsabilités supplémentaires à l'objet emballé, alors que le *composite* se contente d'« additionner » les résultats de ses enfants.

Mais ces patrons de conception peuvent également coopérer : vous pouvez utiliser le *décorateur* pour étendre le comportement d'un objet spécifique d'un arbre *Composite*.

- Les conceptions qui reposent énormément sur le **Composite** et le **Décorateur** tirent des avantages à utiliser le **Prototype**. Il vous permet de cloner les structures complexes plutôt que de les reconstruire à partir de rien.



DÉCORATEUR

Alias : Emballeur, Empaqueteur, Wrapper, Decorator

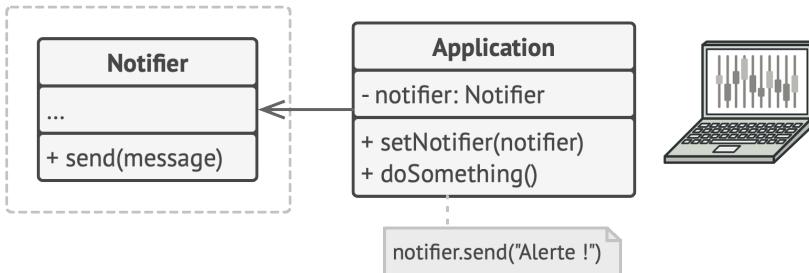
Décorateur est un patron de conception structurel qui permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements.

(:() Problème

Imaginez que vous travaillez sur une librairie qui permet aux programmes d'envoyer des notifications à leurs utilisateurs lorsque des événements importants se produisent.

La version initiale de la librairie était basée sur la classe `Notificateur` qui n'avait que quelques attributs, un constructeur et une unique méthode `envoyer`. La méthode pouvait prendre un message en paramètre et l'envoyait à une liste d'e-mails à l'aide du constructeur du notificateur. Une application externe qui jouait le rôle du client devait créer et configurer l'objet notificateur une première fois, puis l'utiliser lorsqu'un événement important se produisait.

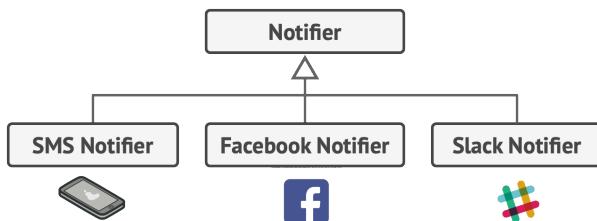
Bibliothèque de notification



Un programme peut utiliser la classe notificateur afin d'envoyer des notifications au sujet d'événements importants à une liste prédéfinie d'adresses e-mail.

Au bout d'un certain temps, vous vous rendez compte que les utilisateurs de la librairie veulent plus que les notifications qu'ils reçoivent sur leur boîte mail. Ils sont nombreux à vou-

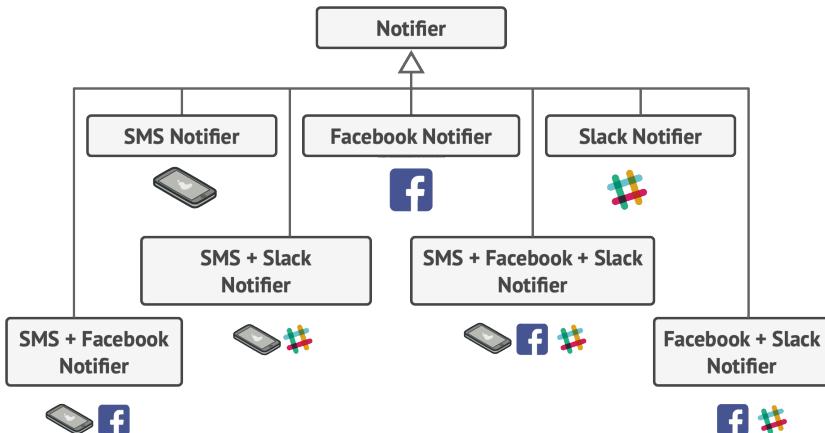
loir recevoir des SMS lorsque leurs applications rencontrent des problèmes critiques. Certains voudraient être prévenus sur Facebook et les employés de certaines entreprises adoreraient recevoir des notifications Slack.



Chaque type de notification est implémenté dans une sous-classe du notificateur.

Cela n'a pas l'air bien difficile ! Vous avez étendu la classe `Notificateur` et ajouté des méthodes de notification supplémentaires dans de nouvelles sous-classes. la classe de notification désirée et utiliser cette instance pour toutes les autres notifications, mais quelqu'un a remis en question ce fonctionnement en vous affirmant qu'il aimerait bien utiliser plusieurs types de notifications simultanément. Si votre maison prend feu, vous avez très certainement envie d'en être informé par tous les moyens possibles.

Vous avez tenté de résoudre ce problème en créant des sous-classes spéciales qui combinent plusieurs méthodes de notification dans une seule classe. Mais il s'avère que cette approche va non seulement gonfler le code de la librairie, mais aussi celui du client.



Explosion combinatoire des sous-classes.

Vous devez structurer vos classes de notification différemment pour ne pas trop les multiplier, sinon vous allez vous retrouver dans le livre Guinness des records.

😊 Solution

La première chose qui nous vient à l'esprit lorsque l'on veut modifier le comportement d'un objet, c'est d'étendre sa classe. Cependant, voici quelques mises en garde concernant l'héritage :

- L'héritage est statique. Vous ne pouvez pas modifier le comportement d'un objet au moment de l'exécution. Vous ne pouvez que remplacer la totalité de l'objet par un autre, généré grâce à une sous-classe différente.

- Les sous-classes ne peuvent avoir qu'un seul parent. Dans la majorité des langages de programmation, vous ne pouvez hériter que d'une seule classe à la fois.

Une des solutions pour contourner ce problème consiste à utiliser l'*aggrégation* ou la *composition*¹ à la place de l'*héritage*. Ces alternatives fonctionnent à peu près de la même façon : un objet *possède* une référence à un autre objet et lui délègue une partie de son travail. Avec l'*héritage*, l'objet *est capable de faire le travail tout seul en héritant du comportement de sa classe mère*.

Grâce à cette nouvelle approche, il est facile de remplacer l'objet référencé par un autre, ce qui modifie le comportement du conteneur au moment de l'exécution. Un objet peut utiliser le comportement de diverses classes, posséder des références à de nombreux objets et leur déléguer toutes sortes de tâches.



Héritage contre Agrégation.

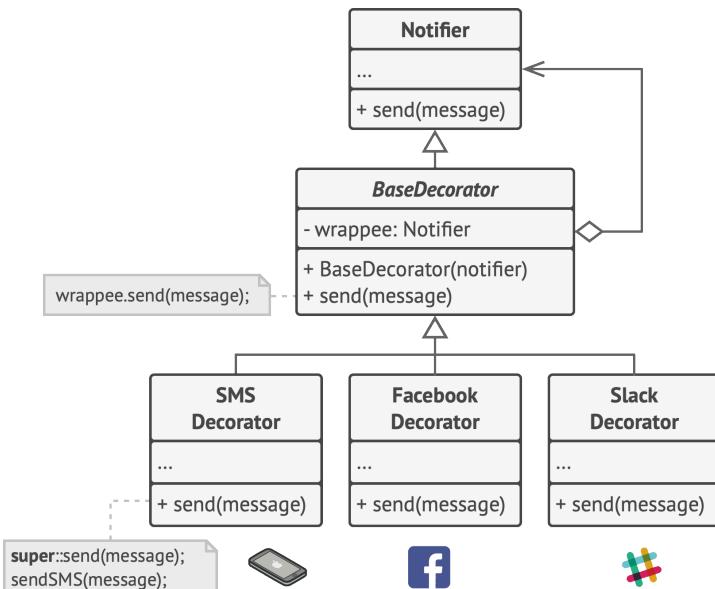
-
1. *Aggrégation* : l'objet A contient les objets B ; B peut exister sans A.
Composition : l'objet A est composé d'objets B ; A gère le cycle de vie de B ; B ne peut pas exister sans A.

L'agrégation et la composition sont des principes clés dans le décorateur, tout comme dans de nombreux autres patrons. Sur ces belles paroles, revenons à nos moutons.

Le décorateur est également appelé « emballeur » ou « empaqueteur ». Ces surnoms révèlent l'idée générale derrière le concept. Un *emballeur* est un objet qui peut être lié par un objet *cible*. L'*emballeur* possède le même ensemble de méthodes que la cible et lui délègue toutes les demandes qu'il reçoit. Il peut exécuter un traitement et modifier le résultat avant ou après avoir envoyé sa demande à la cible.

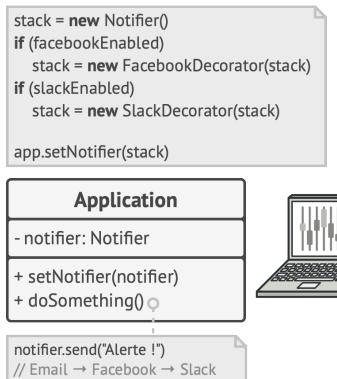
À quel moment un emballeur devient-il réellement un décorateur ? Comme je l'ai déjà dit, un emballeur implémente la même interface que l'objet emballé. Du point de vue du client, ces objets sont identiques. L'attribut de la référence de l'*emballeur* doit pouvoir accueillir n'importe quel objet qui implémente cette interface. Vous pouvez ainsi utiliser plusieurs emballeurs sur un seul objet et lui attribuer les comportements de plusieurs emballeurs en même temps.

Reprendons notre exemple et mettons-y une notification par e-mail dans la classe de base `Notificateur`, mais transformons toutes les autres méthodes de notification en décorateurs.



Des méthodes de notification deviennent des décorateurs.

Le code client doit emballer un objet basique Notificateur pour le transformer en un ensemble de décorateurs adapté aux préférences du client. Les objets qui en résultent sont empilés.

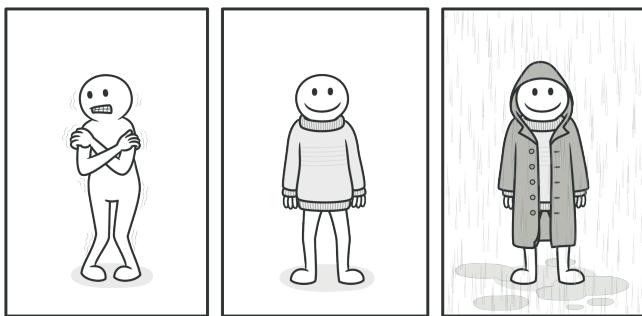


Les applications peuvent configurer des piles complexes de décorateurs pour les notifications.

Le client traite le dernier objet de la pile. Les décorateurs implémentent tous la même interface (le notificateur de base). Le reste du code client manipule indifféremment l'objet notificateur original et l'objet décoré.

Nous pouvons utiliser cette technique pour tous les autres comportements, comme la mise en page des messages ou la création de la liste des destinataires. Le client peut décorer l'objet avec des décorateurs personnalisés tant qu'ils suivent la même interface que les autres.

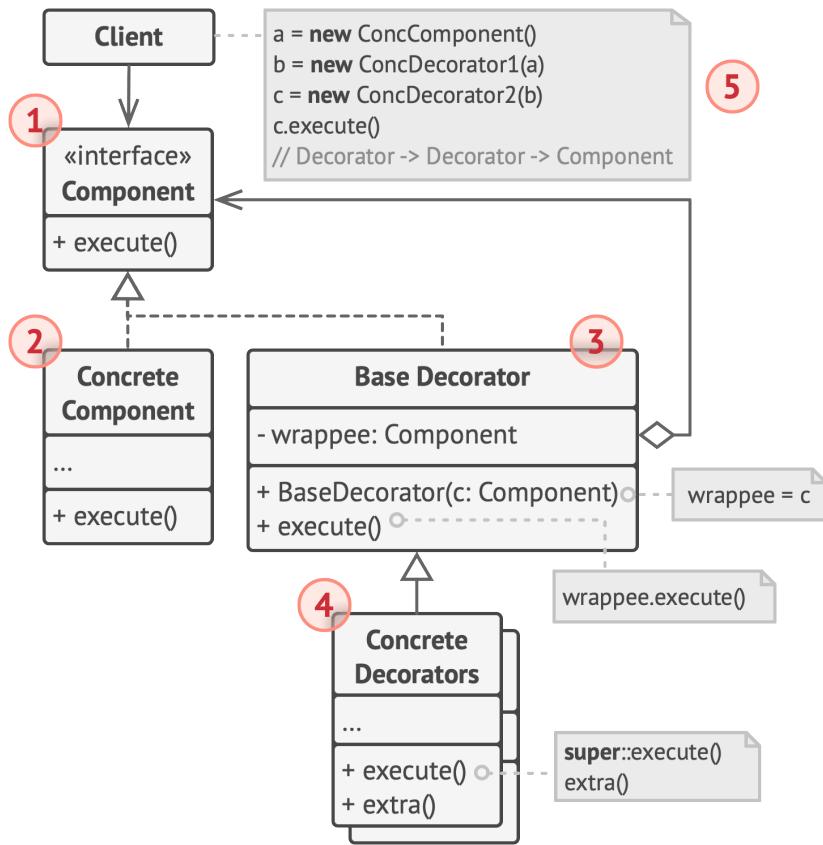
Analogie



Les effets se cumulent si vous portez plusieurs couches de vêtements.

Porter des vêtements est un bon exemple d'utilisation. Si vous avez froid, vous vous enroulez dans un pull. Si vous avez encore froid, vous pouvez porter un blouson par-dessus. S'il pleut, vous enfilez un imperméable. Tous ces vêtements « étendent » votre comportement de base mais ne font pas partie de vous, et vous pouvez facilement enlever un vêtement lorsque vous n'en avez plus besoin.

Structure

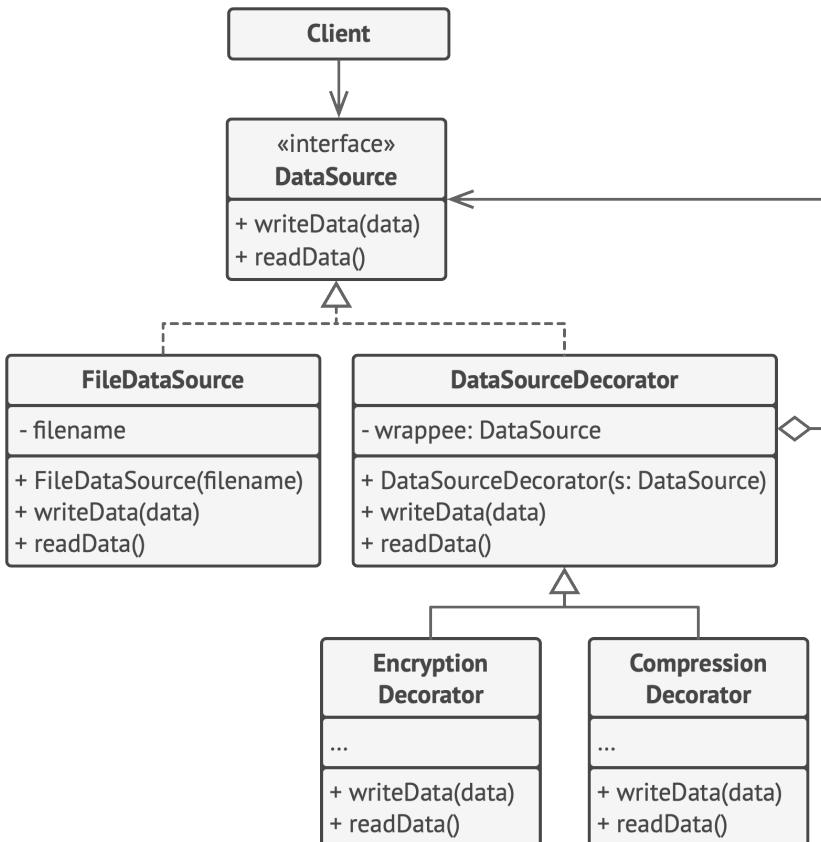


1. Le **Composant** déclare l'interface commune pour les décorateurs et les objets décorés.
 2. Le **Composant Concret** est une classe contenant des objets qui vont être emballés. Il définit le comportement par défaut qui peut être modifié par les décorateurs.

3. Le **Décorateur de Base** possède un attribut pour référencer un objet emballé. L'attribut doit être déclaré avec le type de l'interface du composant afin de contenir à la fois les composants concrets et les décorateurs. Le décorateur de base délègue toutes les opérations à l'objet emballé.
4. Les **Décorateurs Concrets** définissent des comportements supplémentaires qui peuvent être ajoutés dynamiquement aux composants. Les décorateurs concrets redéfinissent les méthodes du décorateur de base et exécutent leur traitement avant ou après l'appel à la méthode du parent.
5. Le **Client** peut emballer les composants dans plusieurs couches de décorateurs, tant qu'il manipule les objets à l'aide de l'interface du composant.

Pseudo-code

Dans cet exemple, le **Décorateur** permet la compression et le chiffrage des données indépendamment du code qui les utilise.



L'exemple de chiffrage et de compression utilisé.

L'application emballe l'objet de la source de données à l'aide de deux décorateurs. Ils modifient la manière dont les données sont lues et écrites sur le disque :

- Les décorateurs chiffrent et compressent les données juste avant qu'elles soient **écrites sur le disque**. La classe d'origine écrit les données chiffrées et protégées dans le fichier sans savoir qu'il y a eu une modification.

- Une fois que les données sont **lues depuis le disque**, elles repassent dans ces mêmes décorateurs qui les décompressent et les déchiffrent.

Les décorateurs et la classe de la source de données implémentent la même interface, ce qui les rend interchangeables aux yeux du code client.

```
1 // L'interface du composant définit les traitements qui peuvent
2 // être modifiés par les décorateurs.
3 interface DataSource is
4     method writeData(data)
5     method readData():data
6
7 // Les composants concrets fournissent des implémentations par
8 // défaut pour les traitements. Plusieurs variantes de ces
9 // classes peuvent se trouver dans un programme.
10 class FileDataSource implements DataSource is
11     constructor FileDataSource(filename) { ... }
12
13     method writeData(data) is
14         // Écrit les données dans le fichier.
15
16     method readData():data is
17         // Lit les données du fichier.
18
19 // La classe de base du décorateur implémente la même interface
20 // que les autres composants. Le but principal de cette classe
21 // est de définir l'interface d'emballage pour tous les
22 // décorateurs concrets. L'implémentation par défaut du code
23 // d'emballage peut comprendre un attribut pour stocker un
```

```
24 // composant emballé et tout le nécessaire pour l'initialiser.
25 class DataSourceDecorator implements DataSource is
26     protected field wrappee: DataSource
27
28     constructor DataSourceDecorator(source: DataSource) is
29         wrappee = source
30
31     // Le décorateur de base délègue tout le travail à l'objet
32     // emballé. Les comportements supplémentaires peuvent être
33     // ajoutés aux décorateurs concrets.
34     method writeData(data) is
35         wrappee.writeData(data)
36
37     // Les décorateurs concrets peuvent appeler l'implémentation
38     // parent du traitement plutôt que d'appeler l'objet emballé
39     // directement. Cette approche simplifie l'extension des
40     // classes décorateur.
41     method readData():data is
42         return wrappee.readData()
43
44     // Les décorateurs concrets doivent appeler les méthodes de
45     // l'objet emballé, mais ils peuvent ajouter quelque chose au
46     // résultat. Les décorateurs peuvent exécuter ce comportement
47     // supplémentaire avant ou après l'appel à l'objet emballé.
48 class EncryptionDecorator extends DataSourceDecorator is
49     method writeData(data) is
50         // 1. Chiffre les données passées.
51         // 2. Envoie les données chiffrées à la méthode emballée
52         // écrireDonnées
53
54     method readData():data is
55         // 1. Récupère les données de la méthode emballée
```

```
56     // lireDonnées.  
57     // 2. La déchiffre si besoin.  
58     // 3. Retourne le résultat.  
59  
60 // Vous pouvez emballer les objets dans plusieurs couches de  
61 // décorateurs.  
62 class CompressionDecorator extends DataSourceDecorator is  
63     method writeData(data) is  
64         // 1. Comprime les données passées.  
65         // 2. Envoie les données compressées à la méthode  
66         // emballée écrireDonnées  
67  
68     method readData():data is  
69         // 1. Récupère les données de la méthode emballée  
70         // lireDonnées.  
71         // 2. La décomprime si besoin.  
72         // 3. Retourne le résultat.  
73  
74  
75 // Option 1 : un exemple simple de l'assemblage d'un décorateur.  
76 class Application is  
77     method dumbUsageExample() is  
78         source = new FileDataSource("somefile.dat")  
79         source.writeData(salaryRecords)  
80         // Le fichier cible a été écrit avec des données brutes.  
81  
82         source = new CompressionDecorator(source)  
83         source.writeData(salaryRecords)  
84         // Le fichier cible a été écrit avec des données  
85         // compressées.  
86  
87         source = new EncryptionDecorator(source)
```

```
88     // La variable source contient à présent :
89     // Chiffrage > Compression > FileDataSource.
90     source.writeData(salaryRecords)
91     // Le fichier cible a été écrit avec des données
92     // compressées et chiffrées.
93
94
95 // Option 2 : un code client qui utilise une source de données
96 // externe. Les objets responsablePaye (SalaryManager) ne
97 // s'encombrent pas des détails concernant le stockage des
98 // données. Ils utilisent une source de données pré configurée
99 // reçue par le configurateur de l'application.
100 class SalaryManager is
101     field source: DataSource
102
103     constructor SalaryManager(source: DataSource) { ... }
104
105     method load() is
106         return source.readData()
107
108     method save() is
109         source.writeData(salaryRecords)
110     // ...d'autres méthodes utiles...
111
112
113 // L'application peut assembler différentes piles de décorateurs
114 // à l'exécution, en fonction de la configuration ou de
115 // l'environnement.
116 class ApplicationConfigurator is
117     method configurationExample() is
118         source = new FileDataSource("salary.dat")
119         if (enabledEncryption)
```

```
120     source = new EncryptionDecorator(source)
121     if (enabledCompression)
122         source = new CompressionDecorator(source)
123
124     logger = new SalaryManager(source)
125     salary = logger.load()
126 // ...
```

💡 Possibilités d'application

- ⚡ Utilisez le décorateur si vous avez besoin d'ajouter des comportements supplémentaires au moment de l'exécution sans avoir à altérer le code source de ces objets.
- ⚡ Le décorateur vous permet de structurer votre logique métier en couches, de créer un décorateur pour chacune de ces couches et de décorer les objets avec différentes combinaisons au moment de l'exécution. Le code client peut traiter les objets uniformément puisqu'ils implémentent la même interface.
- ⚡ Utilisez ce patron si l'héritage est impossible ou peu logique pour étendre le comportement d'un objet.
- ⚡ De nombreux langages de programmation permettent l'utilisation du mot clé `final` pour interdire l'héritage d'une classe. Le seul moyen d'étendre le comportement d'une telle classe est de l'emballer en utilisant un décorateur.



Mise en œuvre

1. Assurez-vous que votre domaine peut être représenté sous la forme d'un composant principal recouvert par plusieurs couches facultatives.
2. Déterminez les méthodes communes entre le composant principal et les couches facultatives. Créez l'interface du composant et déclarez-y ces méthodes.
3. Créez une classe concrète pour le composant et définissez son comportement de base.
4. Créez une classe de base décorateur. Elle doit inclure un attribut qui va permettre de stocker la référence à un objet emballé. Cet attribut doit être déclaré avec le type de l'interface du composant, afin de le relier aux composants concrets et aux décorateurs. Le décorateur de base doit déléguer tout le travail à l'objet emballé.
5. Assurez-vous que les classes implémentent l'interface du composant.
6. Créez des décorateurs concrets en les implémentant à partir du décorateur de base. Un décorateur concret doit exécuter son comportement avant ou après l'appel à la méthode de son parent (qui délègue toujours la tâche à l'objet emballé).

7. Le code client doit être responsable de la création des décorateurs et de leur agencement en fonction des besoins du client.

→ Avantages et inconvénients

- ✓ Vous pouvez étendre le comportement d'un objet sans avoir recours à la création d'une nouvelle sous-classe.
- ✓ Vous pouvez ajouter ou retirer dynamiquement des responsabilités à un objet au moment de l'exécution.
- ✓ Vous pouvez combiner plusieurs comportements en emballant un objet dans plusieurs décorateurs.
- ✓ *Principe de responsabilité unique.* Vous pouvez découper une classe monolithique qui implémente plusieurs comportements différents en plusieurs petits morceaux.
- ✗ Retirer un emballeur spécifique de la pile n'est pas chose aisée.
- ✗ Il n'est pas non plus aisé de mettre en place un décorateur dont le comportement ne varie pas en fonction de sa position dans la pile.
- ✗ Le code de configuration initial des couches peut avoir l'air assez moche.

↔ Liens avec les autres patrons

- L'**Adaptateur** fournit une interface complètement différente pour accéder à un objet existant. En revanche, avec le modèle du **Décorateur**, l'interface reste la même ou est étendue. De

plus, *Décorateur* supporte la composition récursive, ce qui n'est pas possible lorsque vous utilisez *Adaptateur*.

- Avec **Adaptateur**, vous accédez à un objet existant via une interface différente. Avec **Procuration**, l'interface reste la même. Avec **Décorateur**, vous accédez à l'objet via une interface améliorée.
- La **Chaîne de Responsabilité** et le **Décorateur** ont des structures de classes très similaires. Ils reposent tous deux sur la composition récursive pour passer les appels à une suite d'objets. Ils possèdent cependant plusieurs différences majeures.

Les handlers de la *chaîne de responsabilité* peuvent lancer des opérations arbitraires indépendantes l'une de l'autre. Ils peuvent également décider de ne pas propager la demande plus loin dans la chaîne. Les *décorateurs* quant à eux peuvent étendre le comportement de l'objet sans perturber leur fonctionnement avec l'interface de base. De plus, les décorateurs n'ont pas le droit d'interrompre la demande.

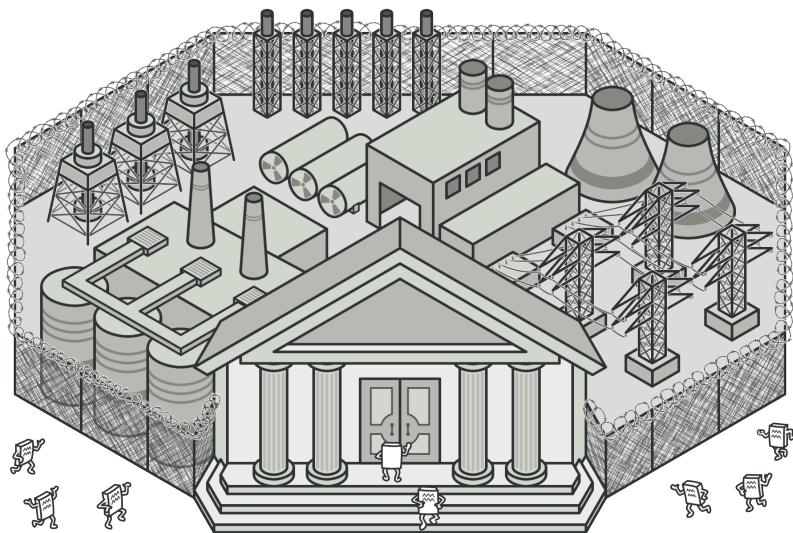
- Le **Composite** et le **Décorateur** ont des diagrammes de structure similaires puisqu'ils reposent sur la composition récursive pour organiser un nombre variable d'objets.

Un *décorateur* est comme un *composite*, mais avec un seul composant enfant. Il y a une autre différence importante : Le *décorateur* ajoute des responsabilités supplémentaires à l'objet

emballé, alors que le *composite* se contente d'« additionner » les résultats de ses enfants.

Mais ces patrons de conception peuvent également coopérer : vous pouvez utiliser le *décorateur* pour étendre le comportement d'un objet spécifique d'un arbre *Composite*.

- Les conceptions qui reposent énormément sur le **Composite** et le **Décorateur** tirent des avantages à utiliser le **Prototype**. Il vous permet de cloner les structures complexes plutôt que de les reconstruire à partir de rien.
- Le **Décorateur** vous permet de changer la peau d'un objet, alors que la **Stratégie** vous permet de changer ses tripes.
- Le **Décorateur** et la **Procuration** ont des structures similaires, mais des intentions différentes. Ces deux patrons sont bâtis sur le principe de la composition, où un objet est censé déléguer certains traitements à un autre. La différence est qu'en principe, la *procuration* gère elle-même le cycle de vie de son objet service, alors que la composition des *décorateurs* est toujours contrôlée par le client.



FAÇADE

Alias : Facade

Façade est un patron de conception structurel qui procure une interface offrant un accès simplifié à une librairie, un framework ou à n'importe quel ensemble complexe de classes.

Problème

Imaginez que vous devez adapter votre code pour manipuler un ensemble d'objets qui appartiennent à une librairie ou à un framework assez sophistiqué. D'ordinaire, vous initialisez tous ces objets en premier, gardez la trace des dépendances et appelez les méthodes dans le bon ordre, etc.

Par conséquent, la logique métier de vos classes devient fortement couplée avec les détails de l'implémentation des classes externes, rendant cette logique difficile à comprendre et à maintenir.

Solution

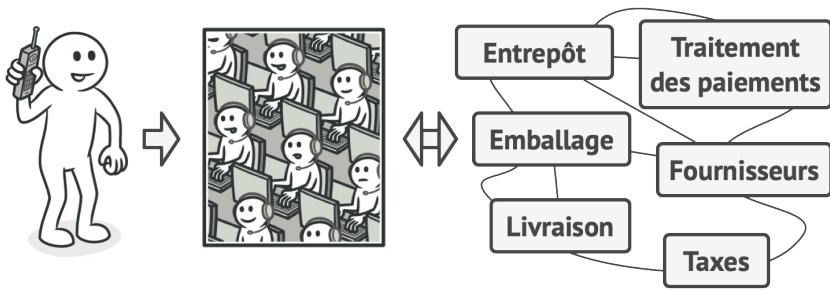
Une façade est une classe qui procure une interface simple vers un sous-système complexe de parties mobiles. Les fonctionnalités proposées par la façade seront plus limitées que si vous interagissiez directement avec le sous-système, mais vous pouvez vous contenter de n'inclure que les fonctionnalités qui intéressent votre client.

Une façade se révèle très pratique si votre application n'a besoin que d'une partie des fonctionnalités d'une librairie sophistiquée parmi les nombreuses qu'elle propose.

Par exemple, une application qui envoie des petites vidéos de chats comiques sur des réseaux sociaux peut potentiellement utiliser une librairie de conversion vidéo professionnelle.

Mais la seule chose dont vous ayez réellement besoin, c'est d'une classe dotée d'une méthode `encoder(fichier, format)`. Après avoir créé cette classe et intégré la librairie de conversion vidéo, votre façade est opérationnelle.

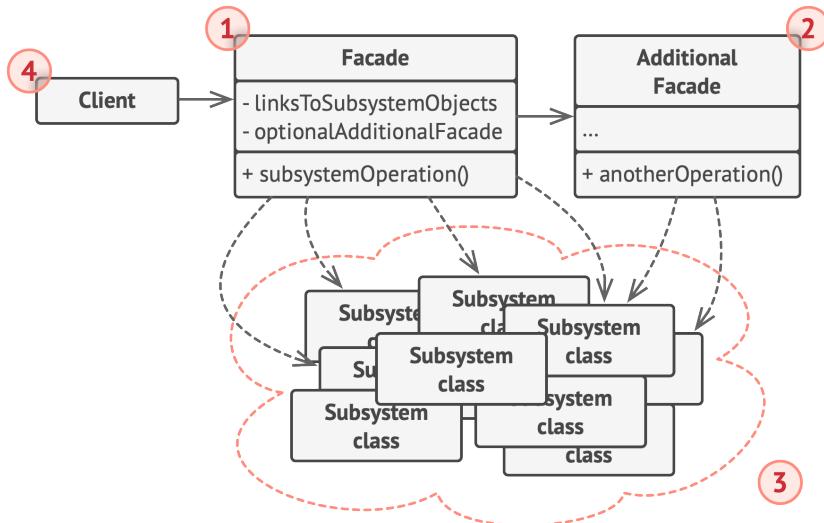
🚗 Analogie



Passer une commande au téléphone.

Lorsque vous téléphenez à un magasin pour commander, un opérateur joue le rôle de la façade pour tous ses services. L'opérateur vous sert d'interface vocale avec le système de commandes, les moyens de paiement et les différents services de livraison.

Structure



1. La **Façade** procure un accès pratique aux différentes parties des fonctionnalités du sous-système. Elle sait où diriger les requêtes du client et comment utiliser les différentes parties mobiles.
2. Une classe **Façade Supplémentaire** peut être créée pour éviter de polluer une autre façade avec des fonctionnalités qui pourraient la rendre trop complexe. Les façades supplémentaires peuvent être utilisées à la fois par le client et par les autres façades.
3. Le **Sous-système Complex**e est composé de dizaines d'objets variés. Pour leur donner une réelle utilité, vous devez plonger au cœur des détails de l'implémentation du sous-système,

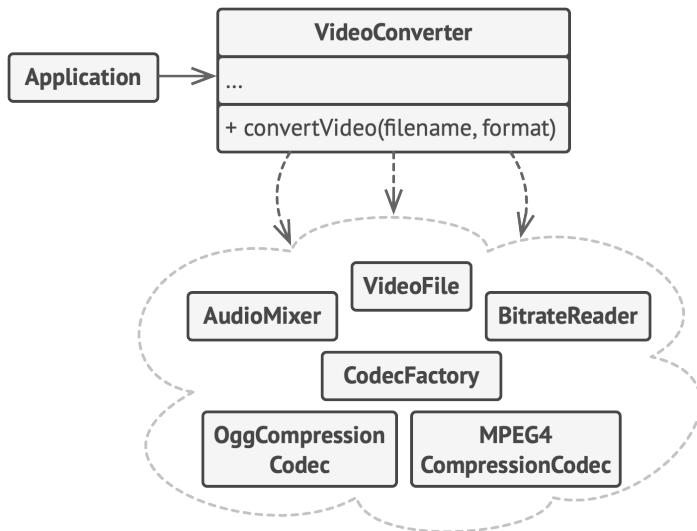
comme initialiser les objets dans le bon ordre et leur fournir les données dans le bon format.

Les classes du sous-système ne sont pas conscientes de l'existence de la façade. Elles opèrent et interagissent directement à l'intérieur de leur propre système.

- Le **Client** passe par la façade plutôt que d'appeler les objets du sous-système directement.

Pseudo-code

Dans cet exemple, la **Façade** simplifie l'interaction avec un framework complexe de conversion vidéo.



Utilisation d'une classe façade dans un exemple d'isolation de dépendances multiples.

Plutôt que d'adapter directement la base de votre code à des dizaines de classes, vous créez une façade qui permet d'envelopper cette fonctionnalité et de la cacher du reste du code. Cette structure permet de minimiser le travail nécessaire aux futures évolutions du framework, ou au remplacement de ce dernier par un autre. Vous pouvez vous contenter de modifier l'implémentation des méthodes de la façade.

```
1 // Voici quelques classes d'un framework complexe de conversion
2 // vidéo externe. Nous n'avons pas la main sur le code, nous ne
3 // pouvons donc pas le simplifier.
4 class VideoFile
5 // ...
6
7 class OggCompressionCodec
8 // ...
9
10 class MPEG4CompressionCodec
11 // ...
12
13 class CodecFactory
14 // ...
15
16 class BitrateReader
17 // ...
18
19 class AudioMixer
20 // ...
21
22
23 // Nous créons une classe façade pour cacher la complexité du
```

```

24 // framework derrière une interface simple. C'est un compromis
25 // entre fonctionnalité et simplicité.
26 class VideoConverter is
27   method convert(filename, format):File is
28     file = new VideoFile(filename)
29     sourceCodec = (new CodecFactory).extract(file)
30     if (format == "mp4")
31       destinationCodec = new MPEG4CompressionCodec()
32     else
33       destinationCodec = new OggCompressionCodec()
34     buffer = BitrateReader.read(filename, sourceCodec)
35     result = BitrateReader.convert(buffer, destinationCodec)
36     result = (new AudioMixer()).fix(result)
37   return new File(result)
38
39 // Les classes application ne dépendent pas d'un milliard de
40 // classes fournies par un framework complexe. Si vous décidez
41 // de changer de framework, vous avez seulement besoin de
42 // réécrire la classe façade.
43 class Application is
44   method main() is
45     convertor = new VideoConverter()
46     mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
47     mp4.save()

```

💡 Possibilités d'application

 Utilisez la façade si vous avez besoin d'une interface limitée mais directe à un sous-système complexe.

⚡ Souvent, les sous-systèmes gagnent en complexité avec le temps et mettre en place un patron de conception implique de créer de nouvelles classes. Dans de nombreux contextes, un sous-système peut se révéler plus souple et facile à réutiliser, mais la quantité de travail pour configurer et écrire le code de base ne fait que croître. La façade tente de remédier à ce problème en fournissant un raccourci aux fonctionnalités du sous-système qui sont adaptées au besoin du client.

💡 Utilisez la façade si vous voulez structurer un sous-système en plusieurs couches.

⚡ Créez des façades pour définir des points d'entrée à chaque niveau d'un sous-système. Vous pouvez réduire le couplage entre plusieurs sous-systèmes en les obligeant à communiquer au travers de façades.

Reprendons notre exemple de framework de conversion vidéo. Il peut être désassemblé en deux couches : tout ce qui concerne la vidéo d'un côté et l'audio de l'autre. Créez une façade pour chaque couche. Vous pouvez utiliser ce genre de façades pour faire communiquer les classes de ces couches ensemble. Cette approche peut fortement ressembler au patron de conception Médiateur.

📋 Mise en œuvre

1. Vérifiez s'il est possible de fournir une interface plus simple que ce qu'un sous-système vous propose. Si cette interface

peut découpler le code client des nombreuses classes du sous-système, vous êtes sur la bonne voie !

2. Déclarez et implémentez cette interface en tant que nouvelle façade. Elle doit rediriger les appels du code client aux objets appropriés du sous-système. Elle doit également être responsable de l'initialisation du sous-système et gérer son cycle de vie, sauf si le code client s'en occupe déjà.
3. Pour exploiter au maximum les avantages de la façade, toute communication du code client avec le sous-système doit passer par elle. Cette manipulation protège le code client de tout effet de bord que des modifications apportées au sous-système pourraient provoquer. Si un sous-système est mis à jour, vous n'aurez besoin que de modifier le code de la façade.
4. Si la façade devient **trop grande**, vous devriez envisager de transférer une partie de ses comportements vers une nouvelle façade plus spécialisée.

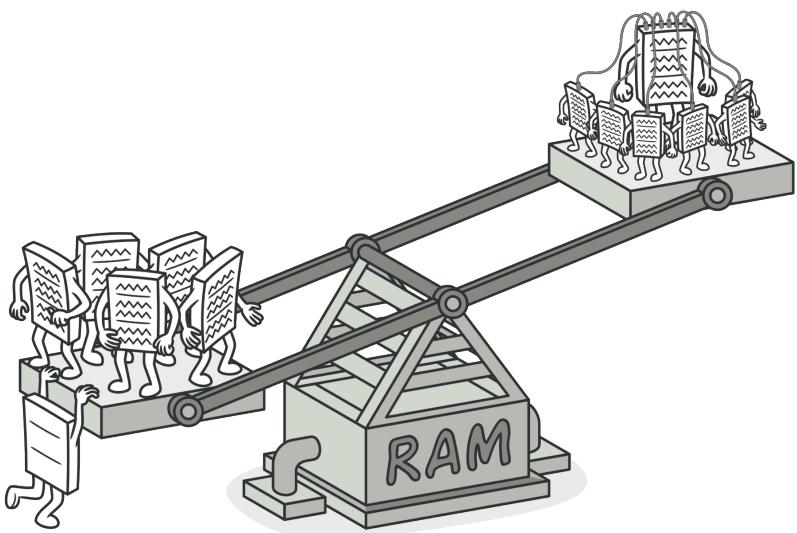
⚠️ Avantages et inconvénients

- ✓ Vous pouvez isoler votre code de la complexité d'un sous-système.
- ✗ Une façade peut devenir **un objet omniscient** couplé à toutes les classes d'une application.

↔ Liens avec les autres patrons

- La **Façade** définit une nouvelle interface pour les objets existants, alors que l'**Adaptateur** essaye de rendre l'interface existante utilisable. L'*adaptateur* emballé généralement un seul objet alors que la *façade* s'utilise pour un sous-système complet d'objets.
- La **Fabrique abstraite** peut remplacer la **Façade** si vous voulez simplement cacher au code client la création des objets du sous-système.
- Le **Poids mouche** vous montre comment créer plein de petits objets alors que la **Façade** permet de créer un seul objet qui représente un sous-système complet.
- La **Façade** et le **Médiateur** ont des rôles similaires : ils essayent de faire collaborer des classes étroitement liées.
 - La *façade* définit une interface simplifiée pour un sous-système d'objets, mais elle n'ajoute pas de nouvelles fonctionnalités. Le sous-système lui-même n'a pas connaissance de la façade. Les objets situés à l'intérieur du sous-système peuvent communiquer directement.
 - Le *médiateur* centralise la communication entre les composants du système. Les composants ne voient que l'objet médiateur et ne communiquent pas directement.

- Une classe **Façade** peut souvent être transformée en **Singleton**, car un seul objet façade est en général suffisant.
- La **Façade** et la **Procuration** ont une similarité : ils mettent en tampon une entité complexe et l'initialisent individuellement. Contrairement à la *façade*, la *procuration* implémente la même interface que son objet service, ce qui les rend interchangeables.



POIDS MOUCHE

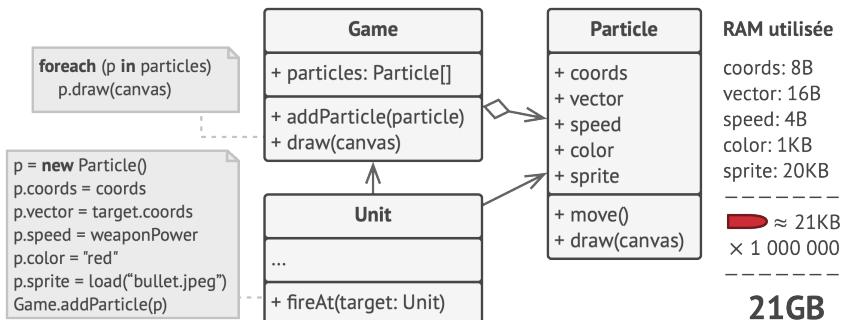
Alias : Poids plume, Flyweight

Poids mouche est un patron de conception structurel qui permet de stocker plus d'objets dans la RAM en partageant les états similaires entre de multiples objets, plutôt que de stocker les données dans chaque objet.

(:() Problème

Pour vous détendre après de longues heures de travail, vous décidez de créer un jeu vidéo tout simple dans lequel les joueurs peuvent se déplacer sur une carte et se tirer dessus. Vous choisissez de mettre en place un système réaliste de particules et d'en faire l'une des caractéristiques les plus importantes du jeu. De grandes quantités de balles, missiles et shrapnels projetés par des explosions vont être envoyés dans toutes les directions et offrir au joueur une expérience palpitante.

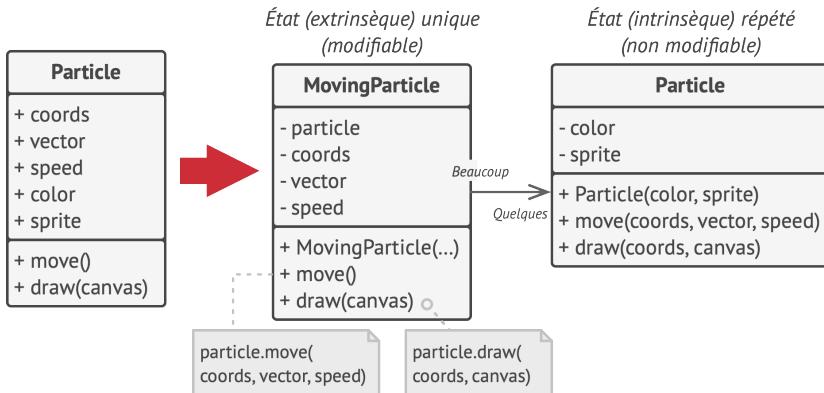
Une fois la programmation terminée, vous lancez le dernier commit, assemblez le jeu, puis l'envoyez à votre ami afin qu'il le teste. Le jeu fonctionnait parfaitement sur votre machine, mais votre ami n'a pas pu jouer bien longtemps : son ordinateur plante systématiquement au bout de quelques minutes. Après plusieurs heures à écumer les logs de debug, vous découvrez que le plantage survient à cause d'une quantité de RAM insuffisante. Il semblerait que la machine de votre ami soit bien moins puissante que la vôtre, le problème se produit donc rapidement sur sa machine.



Il semblerait que le problème soit causé par votre système de particules. Chaque particule (une balle, un missile ou un morceau de shrapnel) est représentée par un objet séparé qui contient de nombreuses données. Lorsque l'action bat son plein sur l'écran du joueur, les nouvelles particules ne peuvent plus tenir dans la RAM et le programme plante.

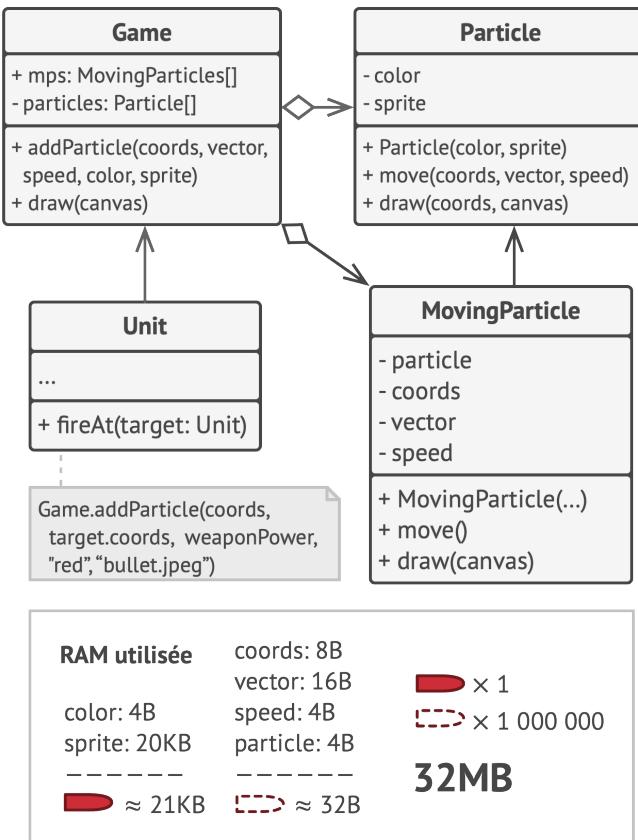
😊 Solution

En inspectant la classe `Particule` de plus près, vous pourrez remarquer que les attributs de couleur et de sprite consomment beaucoup plus de mémoire que les autres attributs. Le pire est que ces deux attributs stockent des données quasi identiques dans toutes les particules. Par exemple, toutes les balles ont la même couleur et le même sprite.



Certaines parties de l'état d'une particule, comme les coordonnées, le mouvement vectoriel et la vitesse sont uniques pour chaque particule. Après tout, les valeurs de ces attributs changent constamment. Ces données représentent le contexte évolutif de la particule, alors que la couleur et le sprite restent constants.

Les données constantes d'un objet forment ce qu'on appelle en général l'*état intrinsèque*. Elles vivent à l'intérieur de l'objet ; les autres objets peuvent seulement les lire, mais pas les modifier. Le reste des données de l'objet sont souvent modifiées « depuis l'extérieur » par d'autres objets et constituent l'*état extrinsèque*.



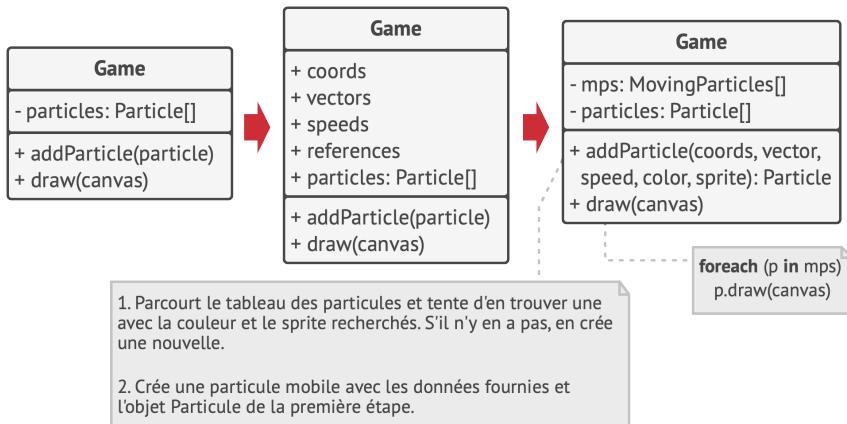
Le patron de conception poids mouche propose de ne plus stocker tous les états extrinsèques à l'intérieur de l'objet. À la place, vous devriez passer cet état à des méthodes spécialisées qui en dépendent. Seul l'état intrinsèque va rester à l'intérieur de l'objet, vous permettant de le réutiliser dans différents contextes. Vous n'aurez besoin que de quelques-uns de ces objets, car ils diffèrent uniquement dans l'état intrinsèque. Ce dernier possède bien moins de variantes que l'état extrinsèque.

Revenons à notre jeu. Si nous extrayons l'état extrinsèque de notre classe particule, trois objets seraient suffisants pour représenter toutes les particules du jeu : une balle, un missile et un morceau de shrapnel. Vous l'avez probablement déjà deviné, un objet qui ne stocke que l'état intrinsèque est appelé poids mouche.

Stockage d'état extrinsèque

Où allons-nous pouvoir mettre l'état extrinsèque ? Il faut bien qu'une classe l'accueille, non ? En général, il se retrouve dans l'objet du contenant, celui qui agrège les objets avant la mise en place du patron.

Dans notre cas, c'est l'objet principal `Jeu` qui stocke toutes les particules dans l'attribut `particules`. Pour déplacer l'état extrinsèque dans cette classe, vous devez créer plusieurs attributs de type tableau pour stocker les coordonnées, les vecteurs et la vitesse de chaque particule. Mais ce n'est pas tout. Vous allez devoir utiliser un autre tableau pour stocker les références à un poids mouche spécifique qui représente la particule. Ces tableaux doivent être synchronisés afin de pouvoir accéder à toutes les données d'une particule en utilisant le même index.



Il existe une solution plus élégante qui consiste à créer une classe contexte séparée, qui va stocker l'état extrinsèque avec une référence à l'objet poids mouche. Cette approche ne nécessite qu'un seul tableau dans la classe du contenant.

Attendez une minute ! Au tout début, il y avait de nombreux objets contextuels, en avons-nous encore besoin ? Techniquelement, oui. Mais ces objets sont bien plus petits qu'avant. Les attributs qui demandent le plus de mémoire ont été déplacés dans quelques objets poids mouche. À présent, des milliers de petits objets contextuels vont pouvoir réutiliser un seul gros objet poids mouche, plutôt que de stocker des milliers de copies de ses données.

Poids mouche et objet immuable

Étant donné que le même objet poids mouche peut être utilisé dans différents contextes, vous devez vous assurer que son état ne peut pas être modifié. Un poids mouche doit initialiser

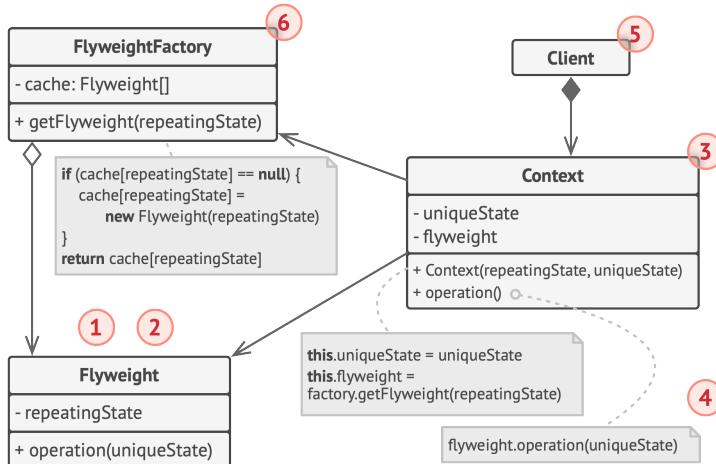
son état une seule fois grâce aux paramètres de son constructeur. Il ne doit pas laisser les autres objets accéder à quelque setter ou attribut public que ce soit.

Fabrique poids mouche

Pour accéder plus facilement aux différents poids mouches, vous pouvez créer une méthode fabrique qui gère une réserve d'objets poids mouche existants. La méthode prend en paramètre l'état intrinsèque du poids mouche demandé par le client, cherche s'il en existe un qui correspond à cet état, et le retourne s'il le trouve. Sinon, il crée un nouveau poids mouche et l'ajoute à la réserve.

Cette méthode peut être placée à différents endroits. Il paraît logique de l'écrire dans le contenant du poids mouche, mais vous pouvez aussi bien créer une nouvelle classe fabrique. Vous pouvez également rendre la méthode fabrique statique et la mettre dans une classe poids mouche.

Structure

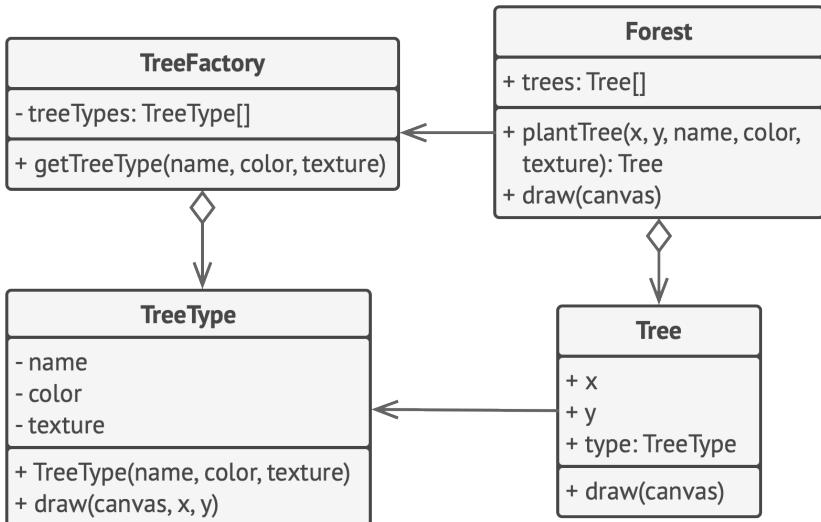


1. Le patron de conception poids mouche ne permet que de faire de l'optimisation. Avant de le mettre en place, assurez-vous que votre programme souffre bien d'un problème de RAM insuffisante lié à un nombre trop important d'objets similaires en mémoire, et que d'autres solutions ne sont pas envisageables.
2. Le **Poids mouche** contient la part de l'état de l'objet original qui peut être partagée entre plusieurs objets. Le même objet poids mouche peut être utilisé dans de nombreux contextes. L'état stocké à l'intérieur d'un poids mouche est *intrinsèque*. L'état passé aux méthodes du poids mouche est *extrinsèque*.
3. La classe **Contexte** détient l'état extrinsèque, qui est unique dans chaque objet original. Lorsqu'un contexte est relié avec un objet poids mouche, il est identique à l'objet original.

4. En général, le comportement de l'objet original reste dans la classe **poids mouche**. Dans ce cas, tout appel à une méthode du poids mouche doit également passer en paramètre les parties de l'état extrinsèque appropriées. Vous pouvez déplacer le comportement dans la classe **contexte**, ce qui permet ensuite de manipuler le poids mouche associé comme n'importe quelles données d'un objet.
5. Le **Client** calcule et stocke l'état extrinsèque des poids mouches. Du point de vue du client, un poids mouche est un objet modèle qui peut être configuré à l'exécution en passant des données contextuelles en paramètre de ses méthodes.
6. La **Fabrique du Poids mouche** gère une réserve de poids mouche existants. Si une fabrique est présente, les clients ne créent pas de poids mouche directement. À la place, ils appellent la fabrique et lui passent les morceaux de l'état intrinsèque du poids mouche désiré. La fabrique parcourt les poids mouches existants et en retourne un s'il correspond aux critères de recherche. Elle en crée un nouveau si elle n'en a pas trouvé.

Pseudo-code

Dans l'exemple suivant, le **Poids mouche** diminue l'utilisation de la mémoire lorsqu'il affiche les millions d'arbres d'un canevas.



Le patron récupère l'état intrinsèque qui se répète dans une classe principale `Arbre` et la stocke dans une classe `TypeArbre`.

À présent, on ne stocke plus les mêmes données dans différents objets. On les met dans quelques poids mouche et on les associe avec les `Arbres` correspondants qui jouent le rôle du contexte. Le code client crée de nouveaux objets arbre en utilisant la fabrique du poids mouche. Cette dernière encapsule la complexité de la recherche du bon objet et le réutilise si possible.

```

1 // La classe poids mouche contient une partie de l'état d'un
2 // arbre. Ces attributs stockent des valeurs uniques pour chaque
3 // arbre. Par exemple, vous n'y trouverez pas les coordonnées de
4 // l'arbre, mais plutôt les textures et les couleurs partagées

```

```
5 // entre plusieurs arbres. Comme ces données sont souvent très
6 // GROSSES, vous gâcheriez beaucoup de mémoire en stockant
7 // chacune d'entre elles dans chaque arbre. À la place, nous
8 // pouvons extraire la texture, la couleur et d'autres données
9 // redondantes dans un objet séparé que de nombreux arbres vont
10 // référencer.
11 class TreeType is
12     field name
13     field color
14     field texture
15     constructor TreeType(name, color, texture) { ... }
16     method draw(canvas, x, y) is
17         // 1. Crée un bitmap d'une couleur, structure et d'un
18         // type donnés.
19         // 2. Dessine le bitmap sur le canevas aux coordonnées X
20         // et Y.
21
22 // La fabrique de poids mouche décide si elle veut réutiliser un
23 // poids mouche existant ou si elle veut en créer un nouveau.
24 class TreeFactory is
25     static field treeTypes: collection of tree types
26     static method getTreeType(name, color, texture) is
27         type = treeTypes.find(name, color, texture)
28         if (type == null)
29             type = new TreeType(name, color, texture)
30             treeTypes.add(type)
31         return type
32
33 // L'objet contextuel contient la partie extrinsèque de l'état
34 // de l'arbre. Une application peut en créer des milliards, car
35 // ils sont petits : ils sont seulement constitués de deux
36 // coordonnées entières et d'un attribut pour une référence.
```

```

37 class Tree is
38     field x,y
39     field type: TreeType
40     constructor Tree(x, y, type) { ... }
41     method draw(canvas) is
42         type.draw(canvas, this.x, this.y)
43
44 // Les classes arbre et Forêt sont les clients du poids mouche.
45 // Vous pouvez les fusionner si vous n'avez pas l'intention
46 // d'ajouter d'autres évolutions à la classe arbre.
47 class Forest is
48     field trees: collection of Trees
49
50     method plantTree(x, y, name, color, texture) is
51         type = TreeFactory.getType(name, color, texture)
52         tree = new Tree(x, y, type)
53         trees.add(tree)
54
55     method draw(canvas) is
56         foreach (tree in trees) do
57             tree.draw(canvas)

```

Possibilités d'application

-  Utilisez le poids mouche uniquement si votre programme génère de nombreux objets et consomme trop de RAM.
-  Les bénéfices apportés par ce patron varient énormément en fonction de son utilisation et de son contexte. Il est surtout utile dans les cas suivants :

- Votre application demande un grand nombre d'objets similaires.
- La RAM de l'appareil ciblé est saturée.
- Les objets contiennent des états identiques qui peuvent être extraits et partagés entre plusieurs objets.

Mise en œuvre

1. Divisez les attributs d'une classe en deux parties pour les transformer en poids mouche :
 - L'état intrinsèque : les attributs qui contiennent des données invariables, dupliquées dans de nombreux objets.
 - L'état extrinsèque : les attributs qui contiennent des données contextuelles uniques à chaque objet.
2. Laissez les attributs de l'état intrinsèque dans la classe, mais empêchez leur modification. Leur valeur initiale ne doit être modifiable qu'à l'intérieur de leur constructeur.
3. Parcourez toutes les méthodes qui utilisent les attributs de l'état extrinsèque. Pour chacun de ces attributs, introduisez un nouveau paramètre que vous utiliserez à la place.
4. Il est envisageable de créer une fabrique pour gérer une réserve de poids mouches. Elle devra vérifier l'existence du poids mouche avant d'en créer un nouveau. Une fois que la fabrique est en place, les clients doivent obligatoirement passer par

elle pour récupérer des poids mouches. Ils doivent donner une description du poids mouche désiré en passant leur état intrinsèque à la fabrique.

5. Le client doit stocker ou calculer les valeurs de l'état extrinsèque (contexte) pour appeler des méthodes du poids mouche. Il est parfois plus pratique de déplacer l'état extrinsèque et l'attribut qui référence le poids mouche dans une classe contexte séparée.

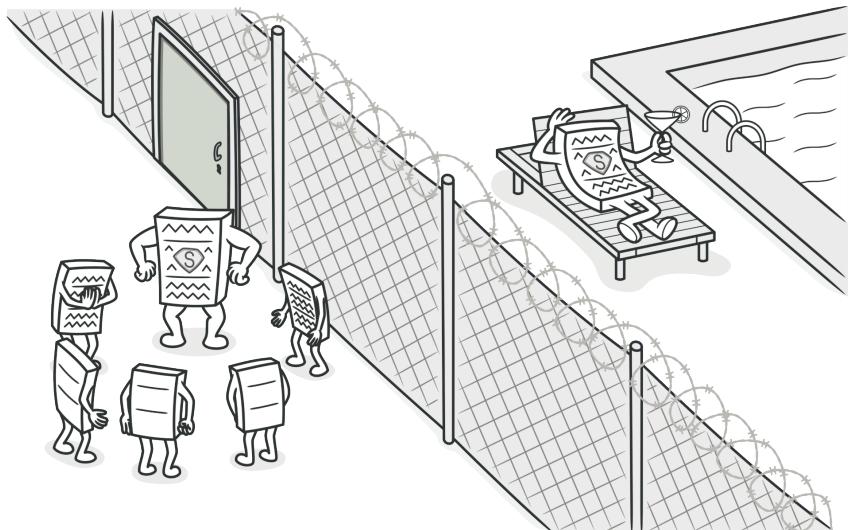
Avantages et inconvénients

- ✓ Vous économisez beaucoup de RAM si votre programme est noyé par des tonnes d'objets similaires.
- ✗ Vous allez gagner en RAM mais perdre en cycles microprocesseur (CPU) si les données du contexte sont recalculées lors de chaque appel d'une méthode poids mouche.
- ✗ Le code devient beaucoup plus compliqué. Les développeurs qui rejoignent l'équipe vont se demander pourquoi les états d'une entité ont été séparés de cette manière.

Liens avec les autres patrons

- Vous pouvez transformer des nœuds de feuilles de l'arbre **Composite** en **Poids mouches** et les partager pour économiser de la RAM.

- Le **Poids mouche** vous montre comment créer plein de petits objets alors que la **Façade** permet de créer un seul objet qui représente un sous-système complet.
- Le **Poids mouche** ressemble au **Singleton** si vous parvenez à compiler tous les états partagés des objets en un seul objet poids mouche. Mais ces patrons de conception ont deux différences fondamentales :
 1. Il ne devrait y avoir qu'une seule instance de singleton, mais une classe *poids mouche* peut avoir plusieurs instances avec différents états intrinsèques.
 2. L'objet *singleton* peut être modifiable alors que les objets poids mouche ne sont pas modifiables.



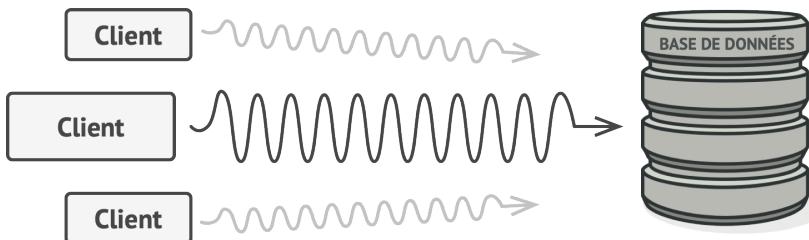
PROCURATION

Alias : Proxy

La **Procuration** est un patron de conception structurel qui vous permet d'utiliser un substitut pour un objet. Elle donne le contrôle sur l'objet original, vous permettant d'effectuer des manipulations avant ou après que la demande ne lui parvienne.

(:() Problème

Pourquoi vouloir maîtriser l'accès à un objet? Voici un exemple : vous avez un énorme objet qui consomme beaucoup de ressources, mais vous n'en avez besoin que de temps en temps.



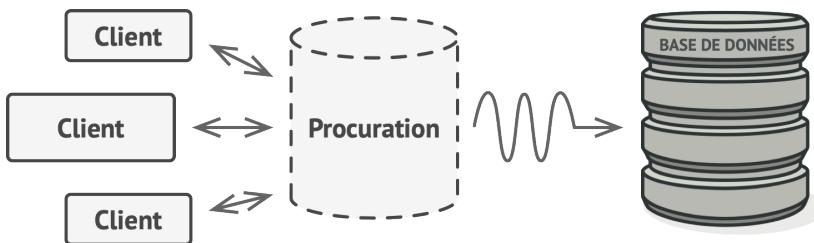
Les requêtes sur la base de données sont parfois très lentes.

Vous pouvez recourir à l'instanciation paresseuse : créer l'objet uniquement lorsque vous en avez besoin. Vous devrez implémenter une initialisation différée dans tous les clients pour l'objet concerné. Malheureusement, vous allez vous retrouver avec beaucoup de code dupliqué.

Dans le meilleur des mondes, nous voudrions mettre ce code directement dans la classe de l'objet, mais ce n'est pas toujours possible. La classe pourrait par exemple faire partie d'une application externe non modifiable.

😊 Solution

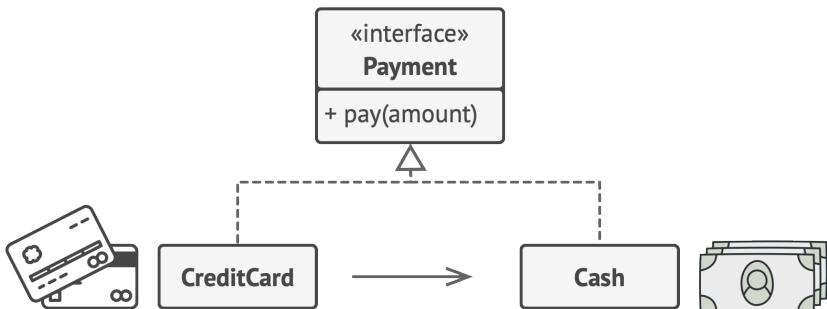
Ce patron de conception vous propose de créer une classe procuration qui a la même interface que l'objet du service original. Vous passez ensuite l'objet procuration à tous les clients de l'objet original. Lors de la réception d'une demande d'un client, la procuration crée l'objet du service original et lui délègue la tâche.



La procuration se déguise en objet base de données. Elle peut gérer l'instanciation paresseuse et mettre en cache le résultat sans que le client ou que l'objet de la base de données ne le remarque.

Mais quel est son intérêt? Si vous voulez lancer un traitement avant ou après avoir appliqué la logique principale de la classe, la procuration peut intervenir sans modifier cette dernière. Elle implémente la même interface que la classe originale, et peut donc être passée en paramètre à n'importe quel client qui attend l'objet du service original.

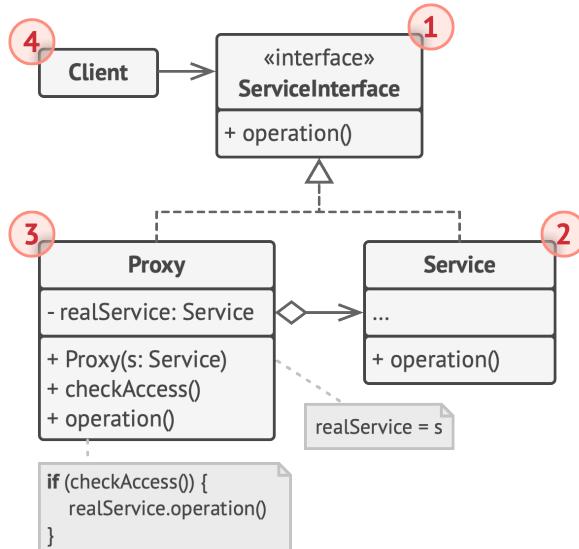
🚗 Analogie



Une carte de crédit peut être utilisée au même titre que de l'argent liquide.

Une carte de crédit est une procuration pour un compte bancaire, qui est une procuration pour une liasse de billets. Ils implémentent la même interface : tous deux peuvent être utilisés pour effectuer un paiement. Le consommateur apprécie grandement, car il n'a pas besoin de garder une grosse somme en liquide sur lui. Le commerçant est également très heureux, car les transactions sont versées électroniquement vers son compte en banque, sans courir le risque d'égarer son dépôt ou de se le faire voler sur le chemin de la banque.

Structure



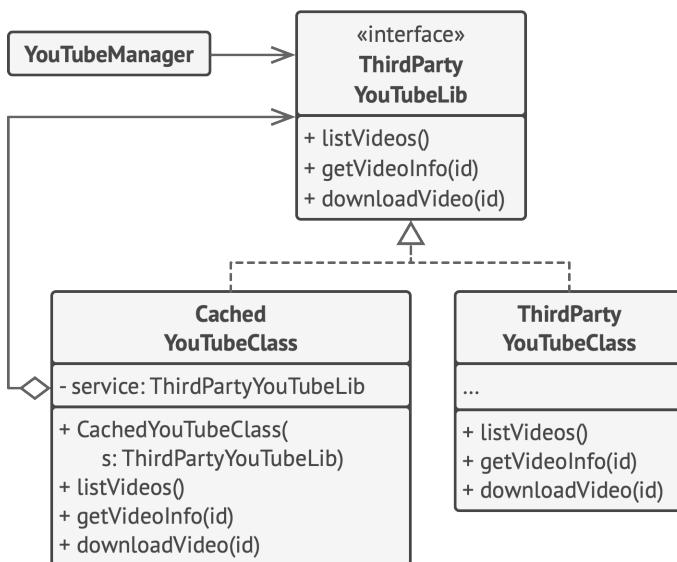
1. L'**Interface Service** déclare l'interface du service. La procuration doit implémenter cette interface afin de pouvoir se déguiser en objet du service.
 2. Le **Service** est une classe qui fournit la logique métier dont vous voulez vous servir.
 3. La **Procuration** est une classe dotée d'un attribut qui pointe vers un objet service. Une fois que la procuration a lancé tous ses traitements (instanciation paresseuse, historisation des logs, vérification des droits, mise en cache, etc.), elle envoie la demande à l'objet du service.

En général, les procurations gèrent le cycle de vie de leurs objets service.

- Le **Client** passe par la même interface pour travailler avec les services et les procurations. Il est ainsi possible de passer une procuration à n'importe quel code qui attend un objet service.

Pseudo-code

L'exemple suivant montre comment le patron de conception **Procuration** nous aide à utiliser l'instanciation paresseuse et la mise en cache pour intégrer une librairie YouTube externe.



Résultats obtenus pour la mise en cache d'un service à l'aide d'une procuration.

La librairie nous fournit une classe pour télécharger des vidéos. Malheureusement, elle n'est pas très efficace. Si l'application client demande une même vidéo à plusieurs reprises, la librairie va la télécharger encore et encore, plutôt que de la mettre en cache après la première utilisation.

La classe procuration implémente la même interface que l'outil de téléchargement original et délègue tout le travail à ce dernier. En revanche, elle garde la trace de tous les fichiers téléchargés et retourne les données du cache lorsque l'application fait plusieurs fois appel à la même vidéo.

```
1 // L'interface d'un service distant
2 interface ThirdPartyYouTubeLib is
3     method listVideos()
4     method getVideoInfo(id)
5     method downloadVideo(id)
6
7 // La méthode d'implémentation concrète d'un connecteur de
8 // service. Les méthodes de cette classe peuvent demander des
9 // informations à YouTube. La vitesse de la demande dépend de la
10 // connexion Internet de l'utilisateur, ainsi que de celle de
11 // YouTube. L'application sera plus lente si plusieurs demandes
12 // sont envoyées en même temps, même si elles recherchent les
13 // mêmes informations.
14 class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
15     method listVideos() is
16         // Envoie une demande via une API à YouTube.
17
18     method getVideoInfo(id) is
```

```
19     // Récupère les métadonnées d'une vidéo.  
20  
21     method downloadVideo(id) is  
22         // Télécharge un fichier vidéo sur YouTube.  
23  
24     // Pour économiser de la bande passante, nous pouvons mettre en  
25     // cache les résultats des demandes et les mettre de côté  
26     // temporairement. Mais vous n'allez pas forcément pouvoir  
27     // écrire ce code directement dans la classe du service. Elle  
28     // pourrait provenir d'une librairie externe ou avoir été  
29     // définie comme `final`. Pour cette raison, nous écrivons le  
30     // code de la mise en cache dans une nouvelle classe procuration  
31     // qui implémente la même interface que la classe du service.  
32     // Elle délègue les tâches à l'objet du service lorsque les  
33     // demandes doivent vraiment être envoyées.  
34 class CachedYouTubeClass implements ThirdPartyYouTubeLib is  
35     private field service: ThirdPartyYouTubeLib  
36     private field listCache, videoCache  
37     field needReset  
38  
39     constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) is  
40         this.service = service  
41  
42     method listVideos() is  
43         if (listCache == null || needReset)  
44             listCache = service.listVideos()  
45         return listCache  
46  
47     method getVideoInfo(id) is  
48         if (videoCache == null || needReset)  
49             videoCache = service.getVideoInfo(id)  
50         return videoCache
```

```
51
52     method downloadVideo(id) is
53         if (!downloadExists(id) || needReset)
54             service.downloadVideo(id)
55
56 // La classe GUI qui fonctionnait auparavant avec un objet
57 // Service n'a pas besoin d'être modifiée tant qu'elle interagit
58 // avec lui par le biais d'une interface. Nous pouvons passer en
59 // toute sécurité un objet procuration à la place d'un objet du
60 // service, car ils implémentent la même interface.
61 class YouTubeManager is
62     protected field service: ThirdPartyYouTubeLib
63
64     constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
65         this.service = service
66
67     method renderVideoPage(id) is
68         info = service.getVideoInfo(id)
69         // Affiche la page de la vidéo.
70
71     method renderListPanel() is
72         list = service.listVideos()
73         // Affiche la liste des miniatures des vidéos.
74
75     method reactOnUserInput() is
76         renderVideoPage()
77         renderListPanel()
78
79 // L'application peut configurer les procurations à la volée.
80 class Application is
81     method init() is
82         aYouTubeService = new ThirdPartyYouTubeClass()
```

```
83     aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)
84     manager = new YouTubeManager(aYouTubeProxy)
85     manager.reactOnUserInput()
```

💡 Possibilités d'application

La procuration possède de nombreux usages. Passons en revue les plus populaires.

- ⚡ **Instanciation paresseuse (procuration virtuelle).** À utiliser lorsque l'objet du service est très consommateur en ressources système car il est actif en permanence, mais vous n'en avez pas tout le temps besoin.
- ⚡ Vous pouvez différer l'initialisation de l'objet plutôt que de le créer au lancement de l'application.
- ⚡ **Vérification des droits (procuration de protection).** Si vous avez besoin de limiter l'accès de vos clients à l'objet du service, par exemple si vos objets sont des parties cruciales d'un système d'exploitation et que les clients sont différentes applications lancées (dont certaines sont malveillantes).
- ⚡ La procuration peut ne transmettre une demande à l'objet du service que si les identifiants du client remplissent certains critères.

 **Lancement local d'un service distant (procuration à distance).**
L'objet du service se situe sur un serveur distant.

 Dans ce cas, la procuration envoie la demande par le réseau en s'occupant de gérer tous les détails compliqués de la gestion du réseau.

 **Demande de logs (procuration de logs).** Pour garder un historique des demandes passées auprès de l'objet du service.

 La procuration peut garder la trace de toutes les demandes passées au service.

 **Mettre en cache les résultats des demandes (procuration de cache).** Si vous voulez mettre en cache les résultats des demandes faites au client et gérer le cycle de vie de ce cache, principalement lorsque les résultats sont imposants.

 La procuration peut gérer la mise en cache des demandes récurrentes qui retournent toujours le même résultat. Les paramètres des demandes peuvent servir de clé.

 **Référencement intelligent.** Si vous voulez vous débarrasser d'un objet très consommateur en ressources lorsqu'aucun client ne l'utilise.

 La procuration peut garder la trace des clients qui ont récupéré une référence vers l'objet du service ou vers ses résultats. De

temps en temps, la procuration peut faire le tour des clients et vérifier s'ils sont toujours actifs. Si la liste des clients est vide, la procuration peut supprimer l'objet du service afin de libérer des ressources.

Elle peut également savoir si le client avait modifié l'objet du service. Les objets non modifiés peuvent ensuite être réutilisés par les autres clients.

Mise en œuvre

1. Si le service n'a pas encore d'interface, créez-en une pour rendre la procuration et les objets du service interchangeables. Il n'est pas toujours possible d'extraire l'interface de la classe du service, car vous devez effectuer des modifications pour que tous les clients du service utilisent cette interface. Heureusement, nous avons un plan B. Transformez la procuration en sous-classe de la classe service, afin qu'elle hérite de l'interface du service.
2. Créez la classe procuration. Elle doit inclure un attribut qui stocke la référence au service. En général, les procurations créent et gèrent le cycle de vie de leurs services. En de rares occasions, un service est passé à la procuration par le biais d'un constructeur du client.
3. Mettez en place les méthodes de la procuration et leur fonctionnement. Dans la plupart des cas, la procuration doit dé-

léguer le travail à l'objet du service après avoir lancé ses traitements.

4. Réfléchissez à l'implémentation d'une méthode de création qui décide si votre client doit utiliser directement le service ou passer par la procuration. Il peut s'agir d'une méthode statique toute simple ou d'une classe procuration avec une méthode fabrique complète.
5. Envisagez également l'implémentation d'une instantiation paresseuse pour l'objet du service.

Avantages et inconvénients

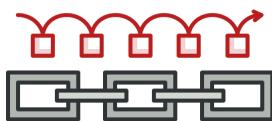
- ✓ Vous pouvez contrôler l'objet du service sans que le client ne s'en aperçoive.
- ✓ Vous pouvez gérer le cycle de vie de l'objet du service si les clients ne s'en occupent pas.
- ✓ La procuration fonctionne même si l'objet du service n'est pas prêt ou pas disponible.
- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouvelles procurations sans toucher au service ou aux clients.
- ✗ Le code peut devenir plus complexe puisque vous devez y introduire de nombreuses classes.
- ✗ La réponse du service peut mettre plus de temps à arriver.

➡ Liens avec les autres patrons

- Avec **Adaptateur**, vous accédez à un objet existant via une interface différente. Avec **Procuration**, l'interface reste la même. Avec **Décorateur**, vous accédez à l'objet via une interface améliorée.
- La **Façade** et la **Procuration** ont une similarité : ils mettent en tampon une entité complexe et l'initialisent individuellement. Contrairement à la *façade*, la *procuration* implémente la même interface que son objet service, ce qui les rend interchangeables.
- Le **Décorateur** et la **Procuration** ont des structures similaires, mais des intentions différentes. Ces deux patrons sont bâtis sur le principe de la composition, où un objet est censé déléguer certains traitements à un autre. La différence est qu'en principe, la *procuration* gère elle-même le cycle de vie de son objet service, alors que la composition des *décorateurs* est toujours contrôlée par le client.

Patrons comportementaux

Les patrons comportementaux s'occupent des algorithmes et de la répartition des responsabilités entre les objets.



Chaîne de responsabilité

Chain of Responsibility

Permet de faire circuler une demande dans une chaîne de handlers. Lorsqu'un handler reçoit une demande, il décide de la traiter ou de l'envoyer au handler suivant de la chaîne.



Commande

Command

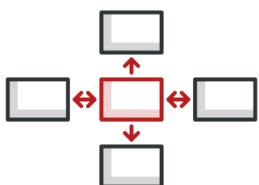
Prend une action à effectuer et la transforme en un objet autonome qui contient tous les détails de cette action. Cette transformation permet de paramétriser des méthodes avec différentes actions, planifier leur exécution, les mettre dans une file d'attente ou d'annuler des opérations effectuées.



Itérateur

Iterator

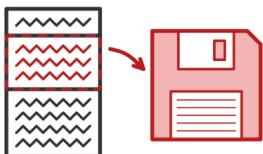
Permet de parcourir les éléments d'une collection sans révéler sa représentation interne (liste, pile, arbre, etc.).



Médiateur

Mediator

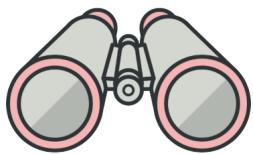
Permet de diminuer les dépendances chaotiques entre les objets. Ce patron restreint les communications directes entre les objets et les force à collaborer uniquement via un objet médiateur.



Memento

Memento

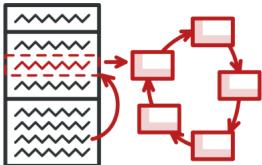
Permet de sauvegarder et de rétablir l'état précédent d'un objet sans révéler les détails de son implémentation.



Observateur

Observer

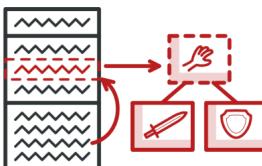
Permet de mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.



État

State

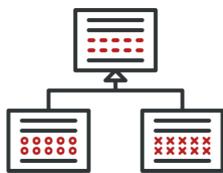
Modifie le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change de classe.



Stratégie

Strategy

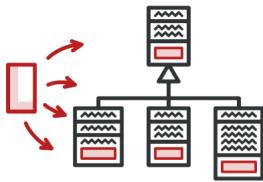
Permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables.



Patron de méthode

Template Method

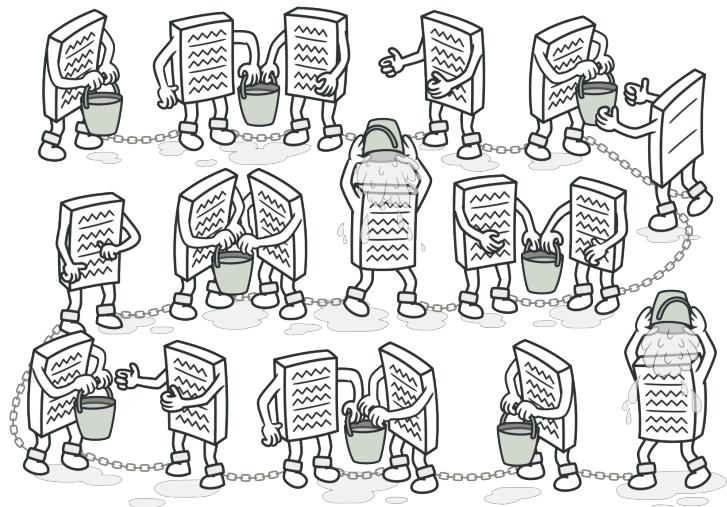
Permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure.



Visiteur

Visitor

Permet de séparer les algorithmes et les objets sur lesquels ils opèrent.



CHAÎNE DE RESPONSABILITÉ

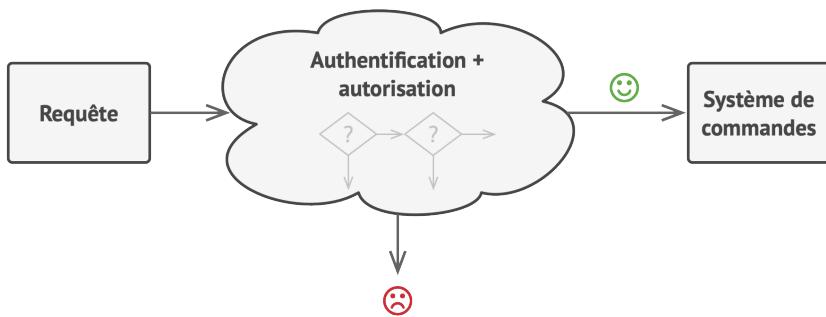
Alias : CoR, Chaîne de commande, Chain of Responsibility

Chaîne de responsabilité est un patron de conception comportemental qui permet de faire circuler des demandes dans une chaîne de handlers. Lorsqu'un handler reçoit une demande, il décide de la traiter ou de l'envoyer au handler suivant de la chaîne.

(:() Problème

Imaginez que vous travaillez sur un système de commandes en ligne. Vous voulez restreindre l'accès au système pour que seuls les utilisateurs authentifiés puissent créer des commandes. Les utilisateurs qui ont les autorisations administratives doivent avoir un accès total aux commandes.

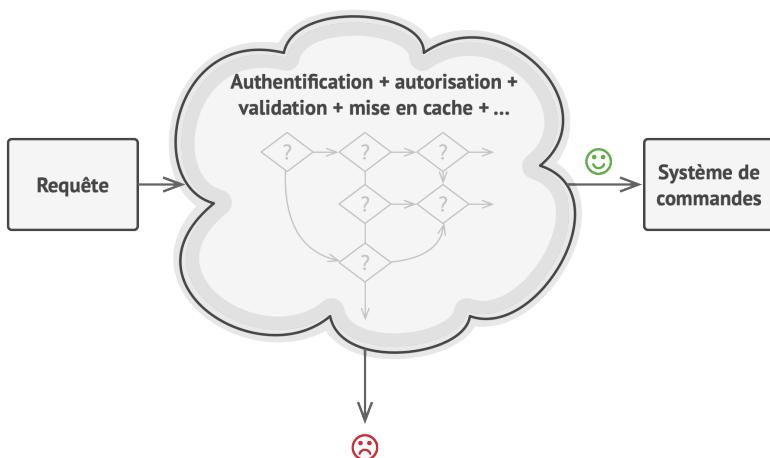
Après un travail de préparation, vous vous rendez compte que ces étapes doivent être exécutées dans un ordre précis. L'application peut essayer d'authentifier un utilisateur auprès du système lorsqu'il reçoit une demande qui contient ses identifiants. Mais si ces derniers ne sont pas corrects et que l'authentification échoue, ce n'est pas la peine de lancer d'autres vérifications.



La demande doit d'abord passer par une série de vérifications avant que le système de commandes ne prenne le relais.

Pendant les mois qui ont suivi, vous avez mis en place plusieurs vérifications supplémentaires.

- Un de vos collègues vous a fait remarquer qu'il n'était pas très prudent d'envoyer des données brutes directement dans le système de commandes. Vous avez donc ajouté une étape de validation supplémentaire pour purger les données de la demande.
- Plus tard, quelqu'un a découvert que le système de mot de passe était vulnérable aux attaques par force brute. Vous avez ajouté une étape qui filtre les échecs répétés provenant de la même adresse IP pour y remédier.
- Quelqu'un d'autre vous a suggéré d'améliorer la vitesse du système en envoyant les résultats directement depuis le cache, lorsque des demandes répétées retournent les mêmes résultats. Vous avez donc implémenté une autre étape qui laisse la demande passer si le système ne trouve pas la réponse correspondante dans le cache.



Plus le code grossit et plus il devient moche et désordonné.

Le code des vérifications – qui n'était déjà pas très ordonné au départ – a grossi au fur et à mesure de l'ajout de nouvelles fonctionnalités. La modification d'une vérification affecte parfois les autres. Le pire dans tout cela, c'est qu'en voulant réutiliser les vérifications existantes pour protéger les autres composants du système, vous avez été obligé de dupliquer du code, car ces composants n'avaient pas besoin de toutes les étapes.

Le système est devenu très difficile à comprendre et cher à maintenir. Vous vous êtes débattu avec le code pendant un moment, jusqu'au jour où vous avez décidé de tout refaire.

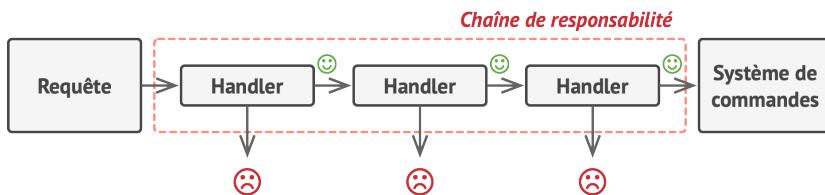
Solution

Tout comme plusieurs patrons de conception comportementaux, la **Chaîne de Responsabilité** repose sur la transformation de comportements particuliers en objets autonomes que l'on appelle *handlers*. Dans notre cas, chaque étape doit être extraite de sa propre classe avec une seule méthode qui effectue la vérification. La demande est passée en paramètre de la méthode avec toutes ses données.

Le patron vous propose de relier ces handlers par une chaîne. Chaque handler stocke une référence vers le prochain handler de la chaîne dans l'un de ses attributs. En plus de traiter la demande, les handlers la font passer plus loin dans la chaîne. La demande fait le tour de la chaîne jusqu'à ce que tous les handlers aient eu l'occasion de la traiter.

Le mieux dans tout cela, c'est qu'un handler peut décider de ne pas envoyer la demande plus loin dans la chaîne et de mettre fin à son traitement.

Dans notre exemple du système de commandes, un handler effectue le traitement et décide s'il doit envoyer la demande plus loin dans la chaîne. Si la commande contient les bonnes données, les handlers peuvent exécuter leur traitement, qu'il s'agisse de l'authentification ou de la mise en cache.

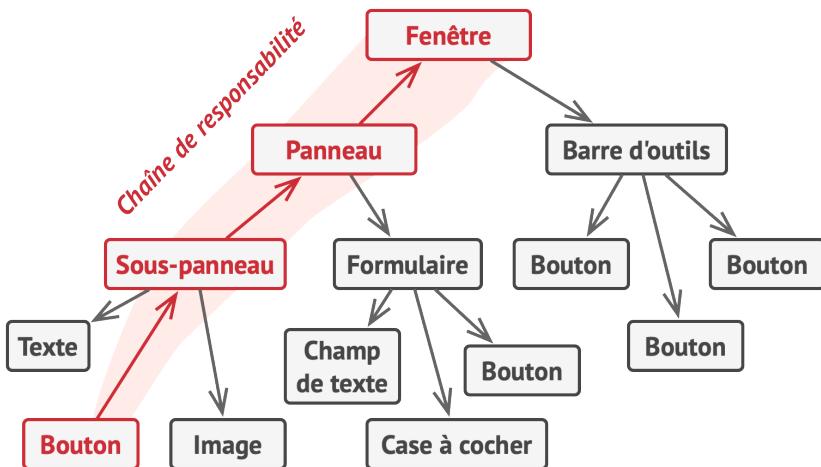


Les handlers forment une chaîne.

Il existe une approche légèrement différente (un peu plus canonique) dans laquelle un handler décide s'il traite la demande dès sa réception. S'il peut la traiter, la demande n'ira pas plus loin. Dans ce cas de figure, un seul handler s'occupera de traiter la demande (ou aucun). C'est une approche classique utilisée dans les piles d'éléments d'une interface graphique (GUI).

Par exemple, lorsqu'un utilisateur clique sur un bouton, l'événement est propagé à travers la chaîne des éléments de la GUI. Cette chaîne débute par le bouton, continue avec ses conteneurs (les formulaires ou les panneaux) et se termine avec la fenêtre principale de l'application. L'événement est traité par

le premier élément de la chaîne qui est en mesure de s'en occuper. Cet exemple est particulièrement intéressant, car il démontre qu'une chaîne peut toujours être extraite depuis une arborescence.



Une chaîne peut être construite à partir de la branche d'une arborescence.

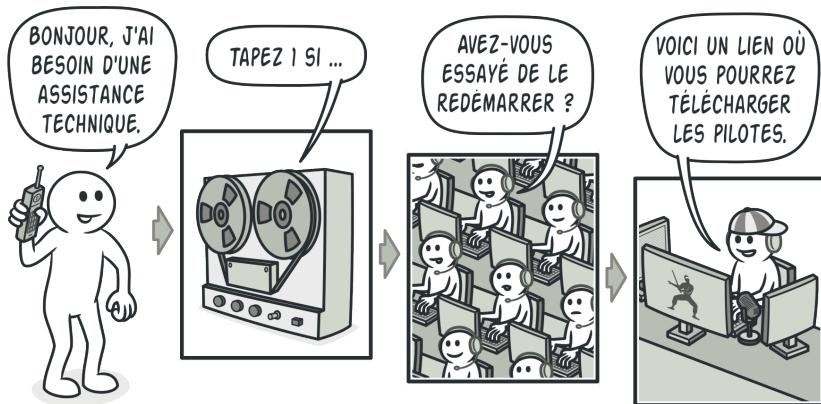
Les classes handler doivent toutes implémenter la même interface. Chaque handler concret ne se préoccupe que de l'existence d'une méthode `traiter` chez le handler suivant. Ainsi, vous pouvez créer vos chaînes à l'exécution et utiliser divers handlers sans coupler votre code à leurs classes concrètes.

🚗 Analogie

Vous venez juste d'installer un nouveau composant sur votre ordinateur. Comme vous êtes un geek, vous avez installé plusieurs systèmes d'exploitation. Vous essayez de tous les démarrer pour voir si votre matériel est bien pris en compte.

Windows détecte votre nouveau composant et l'active automatiquement. En revanche, votre petit chouchou Linux refuse de le faire fonctionner. Avec un soupçon d'espoir, vousappelez le support technique dont le numéro est indiqué sur la boîte.

Votre premier interlocuteur n'est autre que la voix robotique du répondeur. Il vous propose neuf solutions à des problèmes classiques, mais aucun ne vous concerne. Au bout d'un moment, le robot vous redirige vers un opérateur humain.



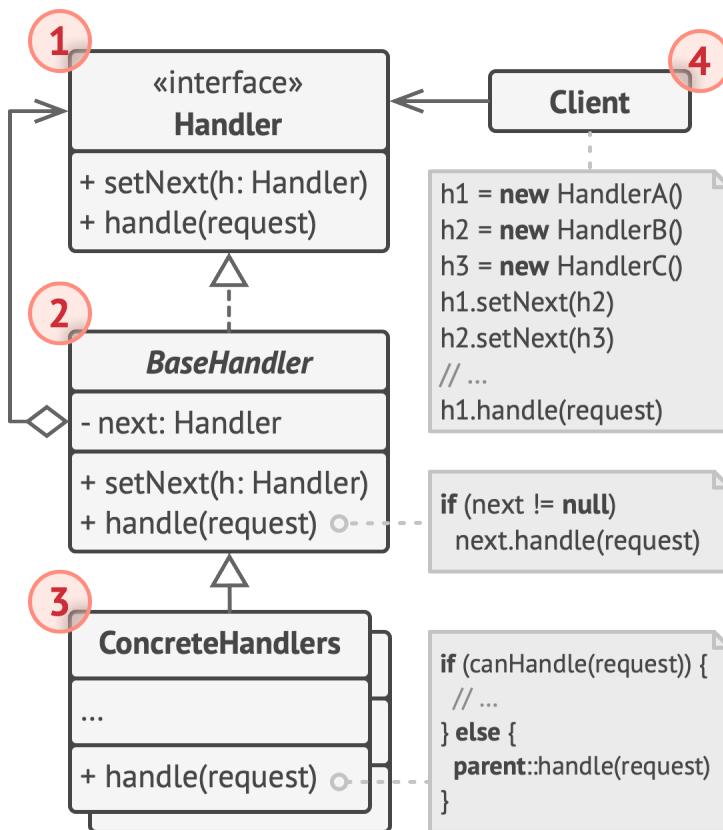
Un appel au support technique passe par plusieurs opérateurs.

Malheureusement, ce dernier ne vous répond rien de bien intéressant. Il ne cesse de répéter des extraits de son manuel et ignore vos commentaires. Après avoir entendu « avez-vous essayé de redémarrer votre ordinateur ? » pour la dixième fois, vous demandez à parler avec un vrai technicien.

Finalement, l'opérateur vous envoie vers un de leurs techniciens qui était probablement assis tout seul depuis des heures

dans la salle des serveurs, située quelque part dans la cave sombre des bureaux de leur entreprise, et attendait avec impatience de pouvoir parler à quelqu'un. Le technicien vous indique un lien de téléchargement pour récupérer les bons pilotes et la marche à suivre pour les installer sur Linux. Enfin, la solution ! Vous mettez fin à l'appel, fou de joie !

Structure



1. Le **Handler** déclare une interface commune pour tous les handlers concrets. En général, il ne comporte qu'une seule méthode pour gérer les demandes, mais il peut parfois en contenir une autre pour désigner le prochain handler de la chaîne.
2. Le **Handler de Base** est une classe facultative dans laquelle le code commun à tous les handlers peut être écrit.

En général, cette classe définit un attribut qui pointe vers le prochain handler. Les clients peuvent assembler une chaîne en passant un handler au constructeur ou au setter du handler précédent. La classe peut également implémenter le comportement par défaut d'un handler : il s'assure de l'existence du prochain handler, puis lui délègue le travail.

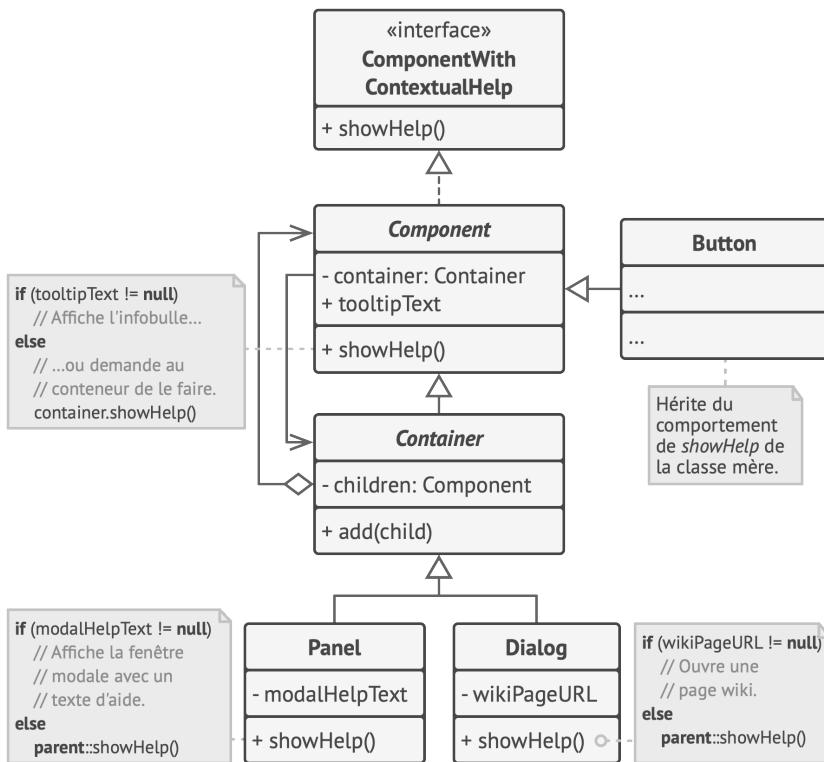
3. Les **Handlers Concrets** contiennent le code qui traite les demandes. Lors de la réception d'une demande, chaque handler décide s'il doit la traiter et s'il doit l'envoyer plus loin dans la chaîne.

Les handlers sont généralement autonomes et non modifiables, et n'accepteront qu'une seule fois les données nécessaires par le biais du constructeur.

4. Le **Client** peut créer les chaînes juste une fois ou les assembler dynamiquement en fonction de la logique métier. Notez bien que la demande initiale n'est pas obligatoirement envoyée au premier handler de la chaîne.

Pseudo-code

Dans cet exemple, la **Chaîne de responsabilité** est chargée d'afficher l'aide contextuelle pour les éléments actifs de la GUI.

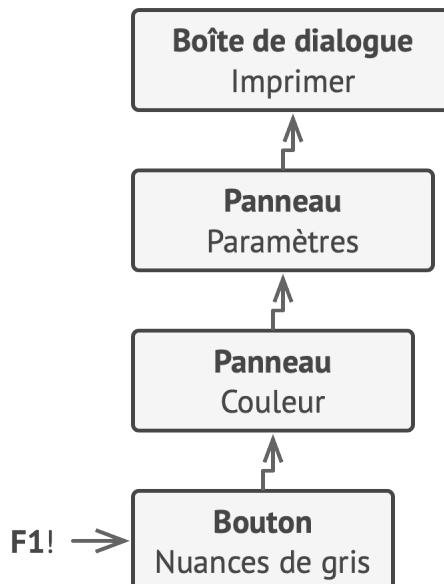


Les classes de la GUI sont construites à l'aide du patron composite. Chaque élément est relié à son conteneur. À n'importe quel moment, vous pouvez bâtir une chaîne d'éléments qui commence par l'élément lui-même et parcourt tous ses conteneurs.

La GUI de l'application prend généralement la forme d'une arborescence. Par exemple, la classe `Dialogue` qui s'occupe du rendu de la fenêtre principale de l'application, est la racine de

l'arbre. La boîte de dialogue contient des Panneaux, qui peuvent eux-mêmes être composés d'autres panneaux ou d'éléments simples de plus bas niveau comme des Boutons et des ChampsTexte.

Un composant simple peut afficher brièvement des infobulles contextuelles si son texte d'aide a été configuré. Les composants plus complexes ont leur propre manière d'afficher l'aide contextuelle. Ils peuvent par exemple consulter l'aperçu du manuel ou ouvrir une page dans un navigateur.



Voici comment une demande d'aide parcourt les objets de la GUI.

Si un utilisateur positionne le pointeur de sa souris sur un élément et appuie sur la touche F1, l'application détecte le composant situé sous le pointeur et lui envoie une demande d'aide.

La demande remonte vers la surface en parcourant tous les conteneurs jusqu'à ce qu'elle atteigne un élément qui peut afficher les informations de l'aide.

```
1 // L'interface du handler déclare une méthode pour exécuter la
2 // demande.
3 interface ComponentWithContextualHelp is
4     method showHelp()
5
6
7 // La classe de base des composants simples.
8 abstract class Component implements ComponentWithContextualHelp is
9     field tooltipText: string
10
11    // Le conteneur du composant agit comme le prochain maillon
12    // de la chaîne des handlers.
13    protected field container: Container
14
15    // Le composant affiche une infobulle si un texte d'aide y
16    // est associé. Sinon, il envoie l'appel vers le conteneur
17    // (s'il existe).
18    method showHelp() is
19        if (tooltipText != null)
20            // Affiche l'infobulle.
21        else
22            container.showHelp()
23
24
25 // Les conteneurs peuvent avoir des composants simples et
26 // d'autres conteneurs enfants. Les liens de la chaîne sont
27 // établis ici. La classe hérite du comportement de la méthode
```

```
28 // montrerAide (showHelp) de son parent.
29 abstract class Container extends Component is
30     protected field children: array of Component
31
32     method add(child) is
33         children.add(child)
34         child.container = this
35
36
37 // Les composants primitifs peuvent se contenter de l'aide par
38 // défaut...
39 class Button extends Component is
40     // ...
41
42 // Mais les composants complexes peuvent redéfinir
43 // l'implémentation par défaut. Si le texte d'aide ne peut être
44 // fourni d'une autre manière, le composant peut toujours
45 // appeler l'implémentation de base (se référer à la classe
46 // Composant).
47 class Panel extends Container is
48     field modalHelpText: string
49
50     method showHelp() is
51         if (modalHelpText != null)
52             // Affiche une fenêtre modale avec le texte d'aide.
53         else
54             super.showHelp()
55
56 // ...idem qu'au-dessus...
57 class Dialog extends Container is
58     field wikiPageURL: string
59
```

```
60  method showHelp() is
61      if (wikiPageURL != null)
62          // Ouvre la page wiki d'aide.
63      else
64          super.showHelp()
65
66
67 // Code client.
68 class Application is
69     // Chaque application configure la chaîne différemment.
70     method createUI() is
71         dialog = new Dialog("Budget Reports")
72         dialog.wikiPageURL = "http://..."
73         panel = new Panel(0, 0, 400, 800)
74         panel.modalHelpText = "This panel does..."
75         ok = new Button(250, 760, 50, 20, "OK")
76         ok.tooltipText = "This is an OK button that..."
77         cancel = new Button(320, 760, 50, 20, "Cancel")
78         // ...
79         panel.add(ok)
80         panel.add(cancel)
81         dialog.add(panel)
82
83 // Imaginez ce qui se passe ici.
84 method onF1KeyPress() is
85     component = this.getComponentAtMouseCoords()
86     component.showHelp()
```

Possibilités d'application

-  Utilisez la chaîne de responsabilité quand votre programme doit traiter des types de demandes variées de différentes manières, mais que leur type exact et leur ordre dans la chaîne ne sont pas connus à l'avance.
-  Ce patron vous permet de former une chaîne avec les handlers et d'interroger chacun d'entre eux lors de la réception de la demande afin de savoir s'ils peuvent la traiter. Chaque handler a ainsi l'opportunité de traiter la demande.
-  Utilisez ce patron si vos handlers doivent absolument respecter un ordre donné.
 -  Comme vous pouvez définir les liens entre les handlers de la chaîne, les demandes la parcourront selon l'ordonnancement que vous avez configuré.
 -  Utilisez la chaîne de responsabilité si l'ensemble des handlers et leur ordre dans la chaîne peuvent changer lors de l'exécution.
 -  Si vous fournissez des setters à un attribut à l'intérieur d'une classe handler, vous serez en mesure d'ajouter, de retirer ou d'ordonner dynamiquement les handlers.



Mise en œuvre

1. Déclarez l'interface du handler et la méthode qui gère les demandes.

Déterminez la manière dont le client passera les données de la demande dans la méthode. La manière la plus flexible consiste à convertir la demande en objet et à le passer en paramètre de la méthode.

2. Pour éviter la duplication du code de base dans les handlers concrets, il peut être utile de créer une classe abstraite de base pour le handler, dérivée de l'interface handler.

Cette classe doit posséder un attribut qui est une référence vers le prochain handler de la chaîne et vous devriez envisager de la rendre non modifiable. Mais si vous prévoyez de modifier les chaînes lors de l'exécution, vous devez définir des setters pour changer la valeur de l'attribut qui stocke les références.

Vous pouvez également mettre en place le comportement par défaut des méthodes des handlers qui consiste à envoyer la demande au prochain objet (s'il en reste). Les handlers concrets peuvent utiliser ce comportement en faisant appel à la méthode de leur parent.

3. Créez les sous-classes des handlers concrets une par une et mettez en place leurs traitements. Chaque handler doit prendre deux décisions lors de la réception d'une demande :

- Traiter ou non la demande.
 - Passer ou non la demande au handler suivant de la chaîne.
4. Le client doit assembler lui-même les chaînes ou recevoir des chaînes pré construites via d'autres objets. Dans le dernier cas, vous devez mettre en place des fabriques pour assembler des chaînes selon la configuration ou selon les paramètres d'environnement.
5. Le client peut déclencher n'importe quel élément de la chaîne, pas forcément le premier. La demande continuera de parcourir la chaîne jusqu'à ce qu'un handler refuse de la laisser continuer, ou jusqu'à ce que l'on atteigne la fin de la chaîne.
6. La chaîne étant dynamique, le client doit être capable de gérer les scénarios suivants :
- La chaîne peut être composée d'un unique lien.
 - Certaines demandes n'iront pas jusqu'au bout de la chaîne.
 - Certaines demandes atteindront la fin de la chaîne sans être traitées.

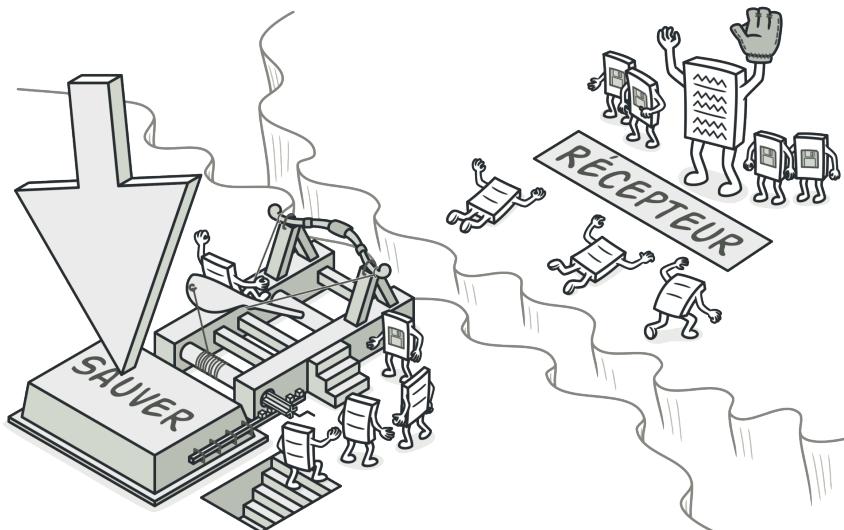
Avantages et inconvénients

- ✓ Vous pouvez contrôler l'ordre des traitements de la demande.
- ✓ *Principe de responsabilité unique.* Vous pouvez découpler les classes qui appellent des traitements, de celles qui les exécutent.

- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouveaux handlers dans le programme sans toucher au code client existant.
- ✗ Il se peut que certaines demandes ne soient pas traitées.

↔ Liens avec les autres patrons

- La Chaîne de responsabilité, la Commande, le Médiateur et l'Observateur proposent différentes solutions pour associer les demandeurs et les récepteurs.
 - La *chaîne de responsabilité* envoie une demande ordonnée qui est passée tout au long d'une chaîne dynamique de récepteurs potentiels, jusqu'à ce que l'un d'entre eux décide de la traiter.
 - La *commande* établit des connexions unidirectionnelles entre les demandeurs et les récepteurs.
 - Le *médiateur* élimine les liens directs entre les demandeurs et les récepteurs, et les force à communiquer indirectement via un objet médiateur.
 - L'*observateur* permet aux récepteurs de s'inscrire et de se désinscrire dynamiquement à la réception des demandes.



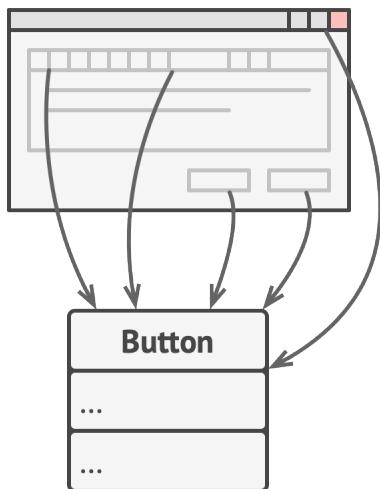
COMMAND

Alias : Action, Translations, Command

Commande est un patron de conception comportemental qui prend une action à effectuer et la transforme en un objet autonome qui contient tous les détails de cette action. Cette transformation permet de paramétriser des méthodes avec différentes actions, planifier leur exécution, les mettre dans une file d'attente ou d'annuler des opérations effectuées.

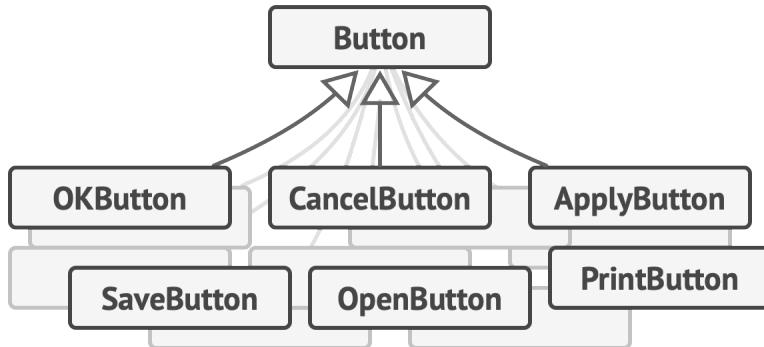
(:() Problème

Imaginez-vous en train de développer un nouvel éditeur de texte. Vous travaillez actuellement sur la création d'une barre d'outils équipée d'un ensemble de boutons qui permettent d'effectuer diverses opérations dans l'éditeur. Vous avez créé une belle classe `Bouton` qui peut être utilisée pour les boutons de la barre d'outils ou pour les boutons génériques des différentes boîtes de dialogue.



Tous les boutons de l'application sont dérivés de la même classe.

Ces boutons ont beau se ressembler, ils sont censés effectuer des actions différentes. Où va-t-on bien pouvoir mettre le code des actions que ces différents boutons déclenchent? Le plus simple est de créer des tonnes de sous-classes où les boutons sont utilisés. Ces sous-classes vont contenir le code exécuté lors d'un clic sur un bouton.



De nombreuses sous-classes de boutons. Tout semble aller pour le mieux.

Mais bientôt, vous allez remarquer que cette approche comporte de sérieux défauts. Le premier que nous allons aborder ici, est le fait que vous avez créé énormément de sous-classes. Lors de chaque modification de la classe `Bouton`, vous risquez d'ajouter de nouveaux bugs dans le code. Pour faire simple, votre GUI (interface graphique utilisateur) est devenue un peu trop dépendante du code volatile de la logique métier.



Plusieurs classes implémentent la même fonctionnalité.

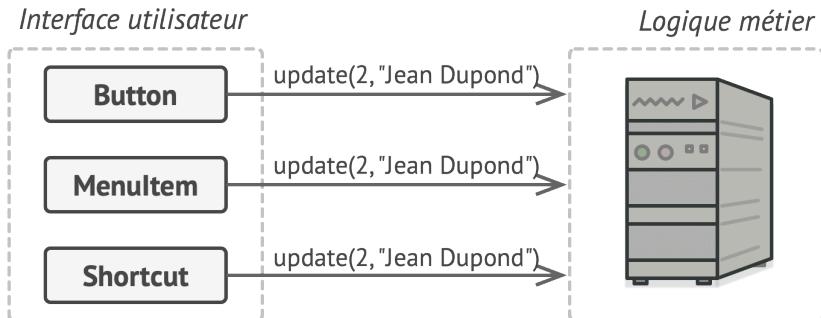
Abordons à présent le défaut majeur. Certaines opérations, comme copier-coller du texte, doivent être appelées depuis différents endroits. Par exemple, un utilisateur peut cliquer sur un petit bouton « copier » de la barre d'outils, copier via le menu contextuel ou encore faire `Ctrl+C` sur son clavier.

À l'époque où notre application n'avait qu'une barre d'outils, cela ne posait pas de problème de mettre l'implémentation des différentes actions dans la sous-classe du bouton. Vous pouviez écrire le code de la copie du texte dans la sous-classe `BoutonCopier` sans problème. Mais lorsque vous implémentez des menus contextuels, des raccourcis ou autres fonctionnalités du même ordre, vous devez dupliquer le code de ces actions dans plusieurs classes ou rendre les menus dépendants des boutons, ce qui est encore pire.

Solution

Un logiciel bien conçu repose souvent sur le *principe de séparation des préoccupations*, qui est en général obtenu en décomposant l'application en plusieurs couches. Le cas le plus classique consiste à avoir une couche pour la GUI et une autre pour la logique métier. La couche GUI a la responsabilité d'afficher une belle image à l'écran, de détecter les actions de l'utilisateur et de l'application, et de les répercuter à l'écran. Mais en ce qui concerne l'exécution des traitements importants comme calculer la trajectoire vers la lune ou générer les rapports annuels, la couche GUI délègue le travail à la couche métier.

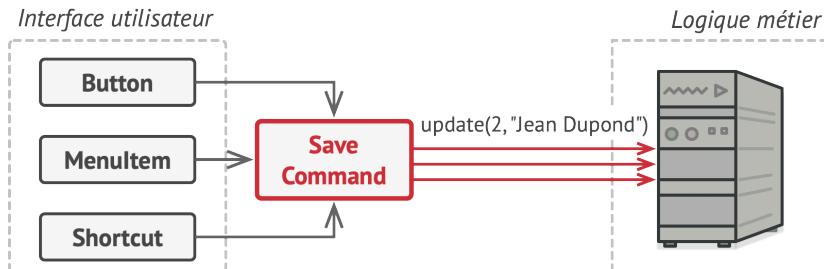
Dans le code, on se retrouve avec un objet GUI qui appelle une méthode de l'objet de la logique métier et lui passe des paramètres. Ce processus est souvent décrit comme un objet envoyant une *demande* à un autre.



Les objets de la GUI peuvent accéder directement aux objets de la couche métier.

Le patron de conception commande propose de ne pas envoyer ces demandes directement depuis les objets de la GUI. À la place, vous devez extraire le détail de ces demandes (un appel vers l'objet, le nom de la méthode et la liste des paramètres) dans une classe *commande* séparée avec une unique méthode qui déclenche cette demande.

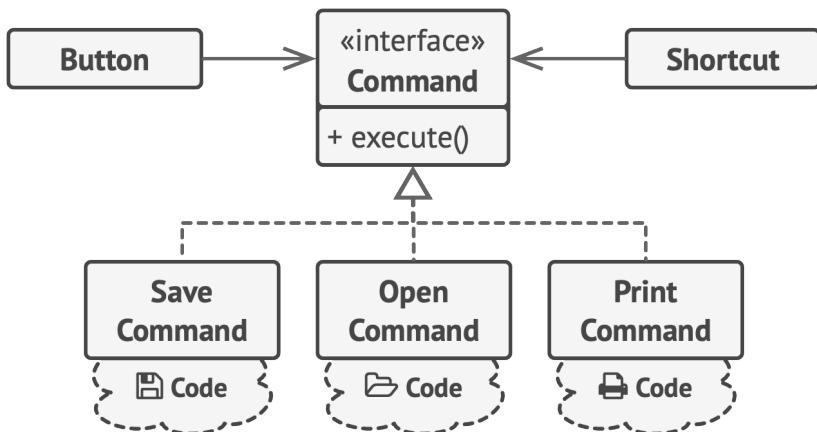
Les objets commande servent de lien entre les diverses GUI et les objets métier. À partir de maintenant, l'objet GUI ne saura plus quel objet métier reçoit la demande et comment il la traite. L'objet GUI déclenche juste la commande, et cette dernière se charge de gérer les détails.



Accéder à la couche métier via une commande.

La prochaine étape consiste à implémenter la même interface pour toutes vos commandes. En général, elle ne possède qu'une seule méthode d'exécution et ne prend aucun paramètre. Cette interface vous permet d'utiliser diverses commandes avec le même demandeur (invoker), sans avoir à la coupler avec les classes concrètes des commandes. En bonus, vous pouvez dorénavant échanger les objets de la commande liés au demandeur, ce qui permet de modifier le comportement du demandeur lors de l'exécution.

Vous vous êtes peut-être rendu compte qu'il manquait une pièce du puzzle : les paramètres de la demande. Un objet GUI devrait les fournir à la couche métier. Mais comme la méthode d'exécution de la commande ne possède aucun paramètre, comment allons-nous nous en sortir ? En fait, la commande doit être pré configurée avec ces données ou être capable de les récupérer par elle-même.



Les objets de la GUI déléguent le travail aux commandes.

Revenons à notre éditeur de texte. Après avoir mis en place le patron de conception commande, nous n'avons plus besoin de toutes ces sous-classes pour implémenter le comportement des différents clics. Vous pouvez vous contenter de mettre un seul attribut dans la classe de base **Bouton** qui conserve une référence vers un objet commande et lui permet de lancer cette commande lors d'un clic.

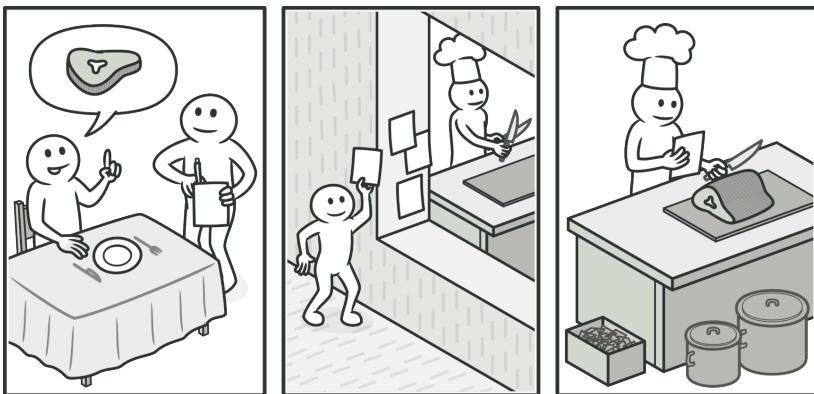
Vous allez implémenter plusieurs classes commande pour chaque opération possible, et allez les relier avec les boutons en fonction du comportement attendu.

Vous pouvez implémenter tous les autres éléments de la GUI comme des menus, raccourcis ou des boîtes de dialogue complètes de la même manière. Ils seront reliés à une commande qui est exécutée lorsqu'un utilisateur interagit avec l'élément de la GUI. Vous l'avez probablement deviné, les éléments qui

déclenchent les mêmes actions seront reliés aux mêmes commandes afin d'éviter la duplication de code.

Les commandes forment ainsi une couche intermédiaire très pratique et réduisent le couplage entre la GUI et les couches métier. Et nous n'avons abordé qu'une infime partie des bénéfices que le patron de conception commande peut apporter.

🚗 Analogie



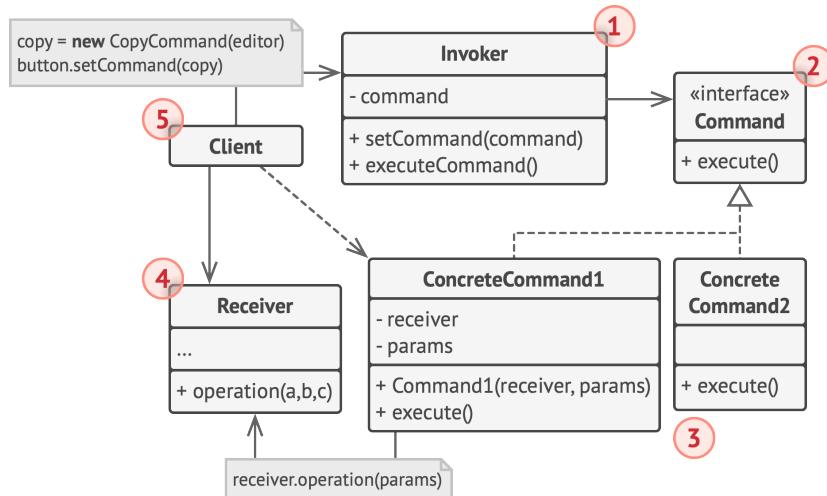
Passer une commande au restaurant.

Après une longue balade en ville, vous trouvez un restaurant sympa et vous vous asseyez à une table près de la fenêtre. Un serveur aimable se rapproche et prend votre commande en l'écrivant sur un bout de papier. Il se rend dans la cuisine et colle la commande sur le mur. Au bout d'un moment, la commande arrive au chef, qui la lit et prépare le plat. Le cuisinier place le repas sur un plateau avec la commande. Le serveur dé-

couvre le plateau, vérifie la commande pour s'assurer que tout est conforme, et l'apporte à votre table.

Le bout de papier joue le rôle de la commande. Il reste dans une file d'attente jusqu'à ce que le chef soit prêt à la servir. Il peut commencer à cuisiner directement, car la commande contient toutes les informations permettant au chef de préparer le plat. C'est mieux que de perdre du temps à venir vous demander les détails de la commande.

Structure



1. La classe **Demandeur (invoker)** a la responsabilité de l'envoi des demandes. Elle doit inclure un attribut qui stocke la référence à un objet commande. Le demandeur déclenche la commande plutôt que de l'envoyer directement au récepteur. Gardez bien en tête que le demandeur n'est pas responsable de la créa-

tion de l'objet commande. En général, il s'agit d'une commande créée auparavant dans le constructeur du client.

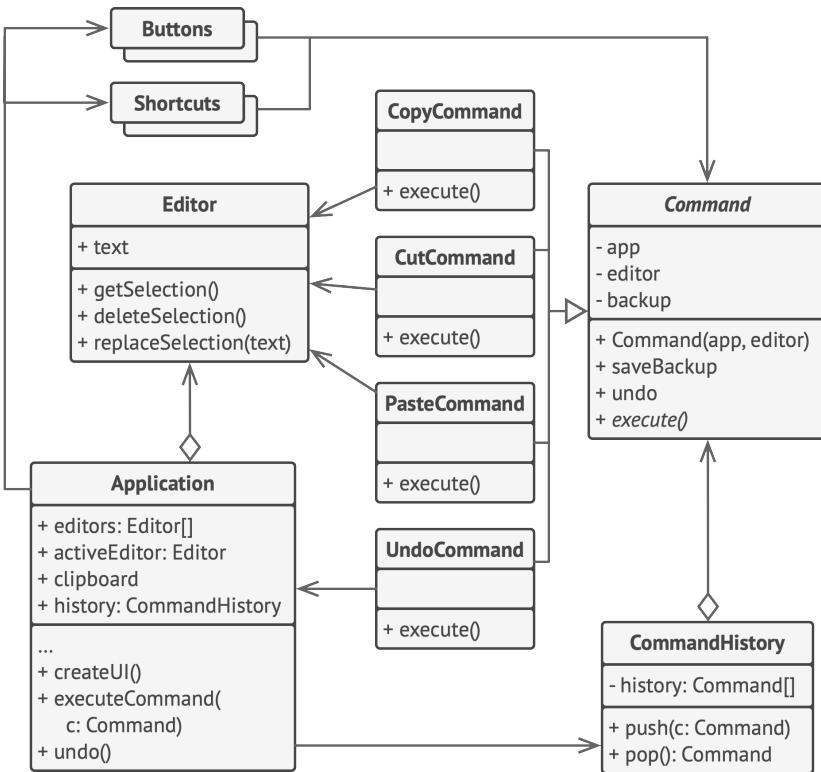
2. L'interface **Commande** déclare habituellement une seule méthode pour exécuter la commande.
3. Les **Commandes Concètes** implémentent plusieurs types de demandes. Une commande concrète n'est pas censée accomplir la tâche d'elle-même. Elle transmet l'appel aux objets de la logique métier. Mais pour simplifier le code, ces classes peuvent être fusionnées.

Les paramètres nécessaires à l'exécution d'une méthode sur un objet récepteur peuvent être déclarés comme des attributs de la commande concrète. Vous pouvez rendre les objets de la commande non modifiables en autorisant uniquement linitialisation de ces attributs dans le constructeur.

4. La classe **Récepteur** porte la logique métier. À peu près nimporte quel objet peut prendre le rôle de récepteur. La majorité des commandes se contentent de passer la demande au récepteur et ce dernier se charge du traitement.
5. Le **Client** crée et configure les objets de la commande concrète. Il doit passer tous les paramètres de la demande (dont linstance du récepteur) dans le constructeur de la commande. Ensuite, la commande qui en résulte peut être associée avec un ou plusieurs demandeurs.

Pseudo-code

Dans cet exemple, le patron de conception **Commande** aide à tracer l'historique des traitements et est en mesure d'annuler les modifications effectuées par un traitement.



Actions pouvant être annulées dans un éditeur de texte.

Les commandes qui modifient l'état de l'éditeur (par exemple couper et coller) font une sauvegarde de son état avant de lancer le traitement associé à la commande. Une fois que la commande a été exécutée, elle est placée dans un historique

de commandes (une pile d'objets commande) avec une sauvegarde de l'état actuel de l'éditeur. Si l'utilisateur a besoin d'annuller un traitement, l'application peut prendre la commande la plus récente de l'historique, lire la sauvegarde associée à l'état de l'éditeur et la restaurer.

Le code client (éléments de la GUI, historique des commandes, etc.) n'est pas couplé avec les classes de la commande concrète, car il interagit avec les commandes via l'interface commande. Cette approche vous permet d'ajouter de nouvelles commandes dans l'application sans endommager l'existant.

```
1 // La classe de base commande définit une interface commune pour
2 // toutes les commandes concrètes.
3 abstract class Command is
4     protected field app: Application
5     protected field editor: Editor
6     protected field backup: text
7
8     constructor Command(app: Application, editor: Editor) is
9         this.app = app
10        this.editor = editor
11
12    // Fait une sauvegarde de l'état de l'éditeur.
13    method saveBackup() is
14        backup = editor.text
15
16    // Rétablit l'état de l'éditeur.
17    method undo() is
18        editor.text = backup
```

```
19
20 // La méthode d'exécution est abstraite pour forcer les
21 // commandes concrètes à fournir leurs propres
22 // implémentations. La méthode doit retourner un booléen qui
23 // indique si la commande a modifié l'état de l'éditeur.
24 abstract method execute()
25
26
27 // Les commandes concrètes sont écrites ici.
28 class CopyCommand extends Command is
29     // La commande pour copier n'est pas sauvegardée dans
30     // l'historique, car elle ne modifie pas l'état de
31     // l'éditeur.
32     method execute() is
33         app.clipboard = editor.getSelection()
34         return false
35
36 class CutCommand extends Command is
37     // La commande 'couper' change l'état de l'éditeur, elle
38     // doit donc être sauvegardée dans l'historique. Elle sera
39     // sauvegardée tant que la méthode renvoie vrai.
40     method execute() is
41         saveBackup()
42         app.clipboard = editor.getSelection()
43         editor.deleteSelection()
44         return true
45
46 class PasteCommand extends Command is
47     method execute() is
48         saveBackup()
49         editor.replaceSelection(app.clipboard)
50         return true
```

```
51
52 // Le traitement 'annuler' est aussi une commande.
53 class UndoCommand extends Command is
54     method execute() is
55         app.undo()
56         return false
57
58
59 // L'historique principal des commandes est juste une pile.
60 class CommandHistory is
61     private field history: array of Command
62
63     // Dernier entré...
64     method push(c: Command) is
65         // Pousse la commande à la fin du tableau de
66         // l'historique.
67
68     // ...premier sorti.
69     method pop():Command is
70         // Récupère la commande la plus récente de l'historique.
71
72
73 // La classe éditeur possède des fonctionnalités de traitement
74 // de texte. Elle joue le rôle du récepteur : toutes les
75 // commandes finissent par déléguer l'exécution aux méthodes de
76 // l'éditeur.
77 class Editor is
78     field text: string
79
80     method getSelection() is
81         // Retourne le texte sélectionné.
82
```

```
83     method deleteSelection() is
84         // Supprime le texte sélectionné.
85
86     method replaceSelection(text) is
87         // Insère le contenu du presse-papiers à la position
88         // actuelle.
89
90
91     // La classe application configure les relations de l'objet.
92     // Elle agit comme un demandeur : lorsque quelque chose doit
93     // être fait, elle crée un objet commande et lance son
94     // traitement.
95     class Application is
96         field clipboard: string
97         field editors: array of Editors
98         field activeEditor: Editor
99         field history: CommandHistory
100
101    // Le code qui assigne les commandes à l'objet UI pourrait
102    // ressembler à ceci.
103    method createUI() is
104        // ...
105        copy = function() { executeCommand(
106            new CopyCommand(this, activeEditor)) }
107        copyButton.setCommand(copy)
108        shortcuts.onKeyPress("Ctrl+C", copy)
109
110        cut = function() { executeCommand(
111            new CutCommand(this, activeEditor)) }
112        cutButton.setCommand(cut)
113        shortcuts.onKeyPress("Ctrl+X", cut)
114
```

```

115     paste = function() { executeCommand(
116         new PasteCommand(this, activeEditor)) }
117     pasteButton.setCommand(paste)
118     shortcuts.onKeyPress("Ctrl+V", paste)
119
120     undo = function() { executeCommand(
121         new UndoCommand(this, activeEditor)) }
122     undoButton.setCommand(undo)
123     shortcuts.onKeyPress("Ctrl+Z", undo)
124
125     // Lance une commande et vérifie si elle doit être ajoutée à
126     // l'historique.
127     method executeCommand(command) is
128         if (command.execute)
129             history.push(command)
130
131     // Prend la commande la plus récente dans l'historique et
132     // lance sa méthode 'annuler'. Vous remarquerez que nous ne
133     // connaissons pas la classe de la commande. Nous n'avons
134     // pas besoin de la connaître puisque la commande sait
135     // comment annuler sa propre action.
136     method undo() is
137         command = history.pop()
138         if (command != null)
139             command.undo()

```

⌚ Possibilités d'application

 Utilisez le patron de conception commande lorsque vous voulez paramétrer des objets avec des traitements.

⚡ Le patron de conception commande peut transformer l'appel d'une méthode en un objet autonome. Cette approche permet des utilisations très intéressantes : vous pouvez passer des commandes dans les paramètres d'une méthode, les stocker à l'intérieur d'objets, modifier les liens des commandes à l'exécution, etc.

Voici un exemple : vous développez un composant de GUI (un menu contextuel par exemple) et vous voulez que vos utilisateurs puissent configurer des éléments de menu qui déclenchent des traitements lorsqu'un utilisateur final clique dessus.

💡 Utilisez ce patron si vous voulez mettre des traitements dans une file d'attente, programmer leur déclenchement, ou les exécuter à distance.

⚡ Comme pour tous les objets, une commande peut être sérialisée, ce qui veut dire qu'on la convertit en une chaîne de caractères qui peut facilement être écrite dans un fichier ou dans une base de données. La chaîne de caractère pourra ensuite être restaurée et utilisée par l'objet de la commande originale. Vous pouvez ainsi différer et planifier l'exécution de la commande. Mais ce n'est pas tout ! De la même manière, vous pouvez mettre une commande dans une file d'attente, l'historiser ou l'envoyer par le réseau.

💡 Utilisez le patron de conception commande quand vous voulez implémenter des opérations réversibles.

- ⚡ Parmi les nombreuses techniques qui permettent d'implémenter la fonctionnalité annuler/rétablissement, le patron de conception commande est probablement le plus populaire.

Pour pouvoir annuler des traitements, vous devez mettre en place un historique des actions effectuées. L'historique des commandes est une pile qui contient tous les objets des commandes exécutées avec l'état de l'application correspondant.

Cette méthode possède deux défauts. Premièrement, l'état d'une application n'est pas facile à sauvegarder, car une partie peut être privée. Ce problème peut être résolu grâce à l'intervention du patron de conception **Memento**.

Deuxièmement, la sauvegarde de l'état risque de consommer beaucoup de RAM. Par conséquent, vous pouvez utiliser une autre alternative : plutôt que de rétablir l'état précédent, la commande effectue l'inverse de la modification, ce qui est malheureusement parfois impossible ou difficile à mettre en place.

Mise en œuvre

1. Déclarez l'interface commande avec une seule méthode d'exécution.
2. Commencez à récupérer les demandes et à les mettre dans les classes concrètes Commande qui implémentent l'interface commande. Chaque classe doit comporter un ensemble d'attri-

buts qui permettent de stocker les paramètres des demandes, avec une référence à l'objet initial du récepteur. Ces valeurs doivent toutes être initialisées dans le constructeur de la commande.

3. Identifiez les classes qui vont jouer le rôle de *Demandeurs*. Créez les attributs de ces classes qui vont stocker les commandes. Les demandeurs doivent uniquement communiquer avec leurs commandes via l'interface commande. En général, ils ne créent pas les objets commande, c'est le code client qui s'en charge.
4. Plutôt que d'envoyer la demande directement au récepteur, modifiez les demandeurs afin qu'ils exécutent la commande.
5. Le client doit initialiser les objets dans l'ordre suivant :
 - Créer les récepteurs.
 - Créer les commandes et les associer avec les récepteurs si besoin.
 - Créer les demandeurs et les associer avec les commandes.

Avantages et inconvénients

- ✓ *Principe de responsabilité unique*. Vous pouvez découpler les classes qui appellent des traitements, de celles qui les exécutent.

- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouvelles commandes dans le programme sans endommager le code client existant.
- ✓ Vous pouvez mettre en place une fonctionnalité annuler-rétablir.
- ✓ Vous pouvez différer l'exécution de vos traitements.
- ✓ Vous pouvez assembler plusieurs commandes simples en une seule plus complexe.
- ✗ Le code peut devenir plus compliqué, car vous créez une nouvelle couche entre les demandeurs et les récepteurs.

↔ Liens avec les autres patrons

- La **Chaîne de responsabilité**, la **Commande**, le **Médiateur** et l'**Observateur** proposent différentes solutions pour associer les demandeurs et les récepteurs.
 - La *chaîne de responsabilité* envoie une demande ordonnée qui est passée tout au long d'une chaîne dynamique de récepteurs potentiels, jusqu'à ce que l'un d'entre eux décide de la traiter.
 - La *commande* établit des connexions unidirectionnelles entre les demandeurs et les récepteurs.
 - Le *médiateur* élimine les liens directs entre les demandeurs et les récepteurs, et les force à communiquer indirectement via un objet médiateur.

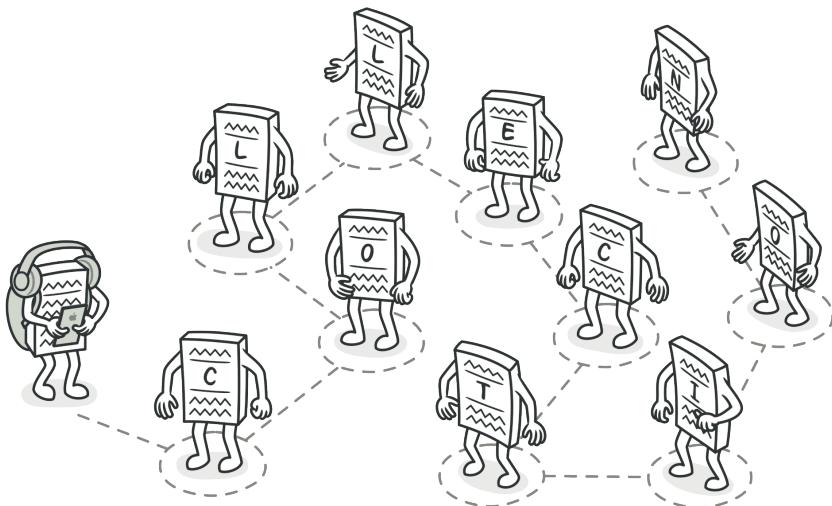
- *L'observateur* permet aux récepteurs de s'inscrire et de se désinscrire dynamiquement à la réception des demandes.
- Les handlers dans la **Chaîne de Responsabilité** peuvent être implémentés comme des **Commandes**. Dans ce cas, vous pouvez lancer beaucoup de traitements différents sur le même objet contexte, représenté par une demande.

Vous pouvez cependant utiliser une autre approche dans laquelle la demande en elle-même est un objet *commande*. Vous pouvez ainsi exécuter le même traitement dans une série de différents contextes connectés par une chaîne.

- Vous pouvez utiliser la **Commande** et le **Memento** ensemble pour implémenter la fonctionnalité « annuler ». Dans ce cas, les commandes ont la responsabilité d'exécuter les divers traitements sur un objet cible. Les mémentos sauvegardent l'état de cet objet juste avant le lancement de la commande.
- La **Commande** et la **Stratégie** peuvent se ressembler, car vous les utilisez toutes les deux pour paramétrier un objet avec une action. Cependant, ces patrons ont des intentions très différentes.
 - Vous pouvez utiliser la *commande* pour convertir un traitement en un objet. Les paramètres du traitement deviennent des attributs de cet objet. La conversion vous permet de différer le lancement du traitement, le mettre dans une file

d'attente, stocker l'historique des commandes, envoyer les commandes à des services distants, etc.

- La *stratégie* quant à elle, décrit généralement différentes manières de faire la même chose et vous laisse permuter entre ces algorithmes à l'intérieur d'une unique classe contexte.
- Le **Prototype** se révèle utile lorsque vous voulez sauvegarder des copies de **Commandes** dans l'historique.
- Vous pouvez traiter le **Visiteur** comme une version plus puissante du patron de conception **Commande**. Ses objets peuvent lancer des traitements sur divers objets dans différentes classes.



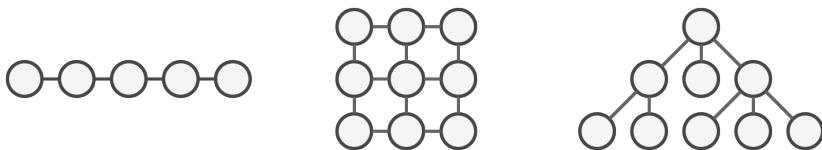
ITÉRATEUR

Alias : Iterator

Itérateur est un patron de conception comportemental qui permet de parcourir les éléments d'une collection sans révéler sa représentation interne (liste, pile, arbre, etc.).

(:() Problème

Les collections ne servent que de conteneur pour un groupe d'objets, mais elles demeurent l'un des types de données les plus utilisés en programmation.



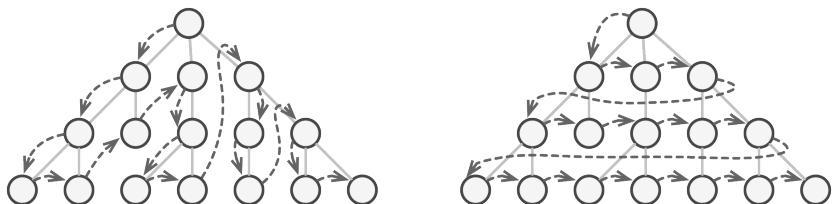
Différents types de collections.

La majorité des collections stockent leurs éléments dans de simples listes, mais certaines d'entre elles sont basées sur les piles, les arbres, les graphes ou d'autres structures complexes de données.

Quelle que soit sa structure, une collection doit fournir un moyen d'accéder à ses éléments pour permettre au code de les utiliser. Elle doit donner la possibilité de parcourir tous ses éléments sans passer plusieurs fois par les mêmes.

À première vue, cela semble simple pour les collections qui ressemblent à une liste. Vous lancez juste une boucle sur tous les éléments. Mais comment faites-vous pour parcourir séquentiellement les éléments d'une structure complexe de données comme un arbre ? Un jour donné, vous allez peut-être vous en sortir avec un parcours en profondeur. Mais le lendemain, vous allez avoir besoin d'un parcours en largeur. La

semaine d'après, vous allez avoir besoin d'autre chose pour établir un parcours aléatoire des éléments de l'arbre.



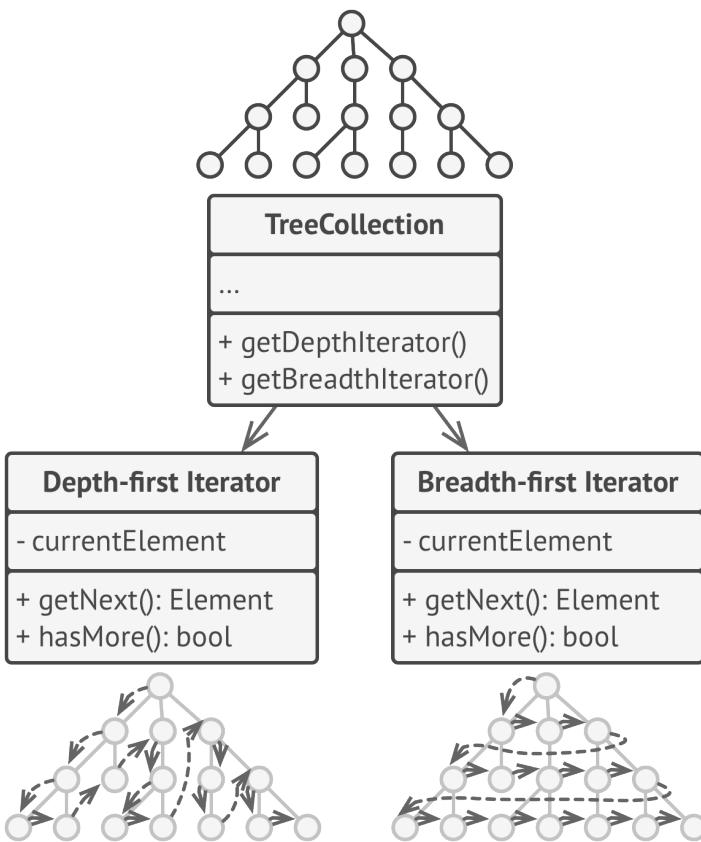
Il y a plusieurs manières de parcourir une même collection.

Plus vous ajoutez d'algorithmes différents pour parcourir votre collection et plus vous masquez sa responsabilité principale : stocker efficacement les données. De plus, certains algorithmes sont dédiés à un usage spécifique. Il serait donc bizarre de les ajouter au comportement d'une collection générique.

Le code client qui va se servir de ces collections se fiche probablement de la manière dont elles stockent leurs éléments. Cependant, ces collections fournissent différentes manières d'accéder à leurs éléments, vous couplez donc forcément votre code aux classes de ces collections.

☺ Solution

Le but du patron de conception itérateur est d'extraire le comportement qui permet de parcourir une collection et de le mettre dans un objet que l'on nomme *itérateur*.



Les itérateurs implémentent différents algorithmes de parcours. Plusieurs itérateurs peuvent parcourir une même collection simultanément.

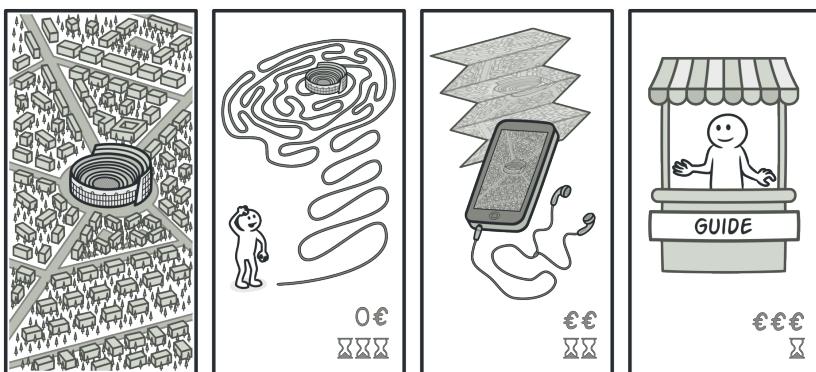
En plus d'implémenter l'algorithme de parcours, un objet itérateur encapsule tous les détails comme la position actuelle et le nombre d'éléments restants avant d'atteindre la fin. Grâce à cela, plusieurs itérateurs peuvent parcourir une même collection simultanément et indépendamment les uns des autres.

En général, les itérateurs fournissent une méthode principale pour récupérer les éléments d'une collection. Le client peut

appeler la méthode `en continu` jusqu'à ce qu'elle ne retourne plus rien, ce qui signifie que l'itérateur a parcouru tous les éléments.

Les itérateurs doivent tous implémenter la même interface. Le code client est ainsi compatible avec tous les types de collections et tous les algorithmes de parcours, tant que l'itérateur adéquat existe. Si vous voulez effectuer un parcours un peu spécial dans une collection, il vous suffit de créer une nouvelle classe itérateur, sans toucher à la collection ni au client.

Analogie



Differentes manières de parcourir Rome.

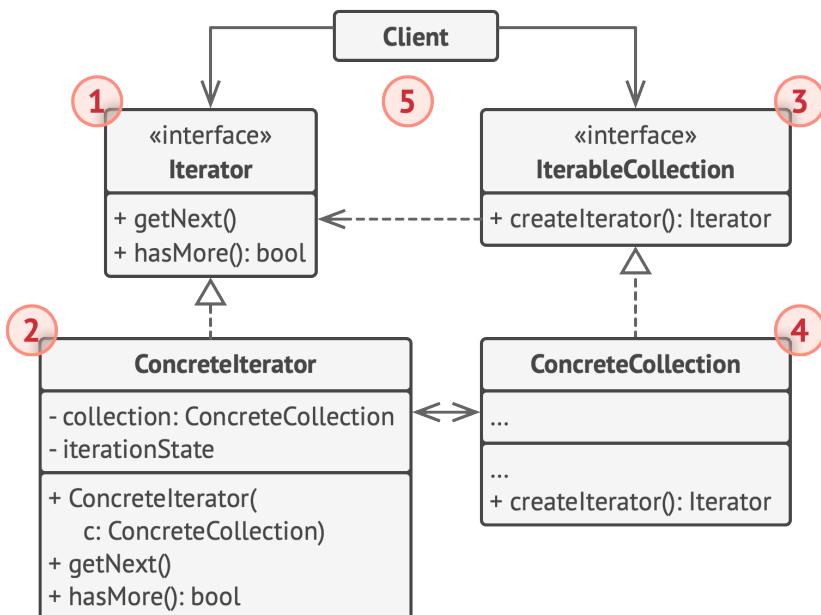
Vous projetez de visiter Rome pendant quelques jours et voulez voir ses principaux sites et attractions. Mais une fois sur place, vous risquez de perdre beaucoup de temps à tourner en rond et même de ne pas réussir à trouver le Colisée.

Vous pourriez acheter une application de guide virtuel sur votre smartphone qui vous permettrait de trouver votre chemin. C'est pratique et peu onéreux, et vous pourriez visiter des sites intéressants à volonté.

Vous pourriez également dépenser une partie de votre budget pour engager un guide local qui connaît la ville comme sa poche. Il adapterait votre séjour en fonction de vos goûts et de vos attentes, vous ferait visiter toutes les attractions et vous raconterait des histoires passionnantes. Ce serait vraiment génial, mais malheureusement, beaucoup plus cher.

Toutes ces possibilités – des directions aléatoires que vous prenez, l'application sur smartphone ou le guide humain – sont des itérateurs sur une vaste collection de sites et d'attractions dans Rome.

Structure



1. L'interface **Itérateur** déclare les opérations nécessaires au parcours d'une collection : récupérer le prochain élément, donner la position actuelle, recommencer la boucle depuis le début, etc.
2. Les **Itérateurs Concrets** implémentent les algorithmes qui servent au parcours d'une collection. L'objet itérateur doit garder la trace du parcours actuel. Grâce à cela, plusieurs itérateurs peuvent parcourir la même collection de manière indépendante.
3. L'interface **Collection** déclare une ou plusieurs méthodes pour récupérer des itérateurs compatibles avec la collection. Le type

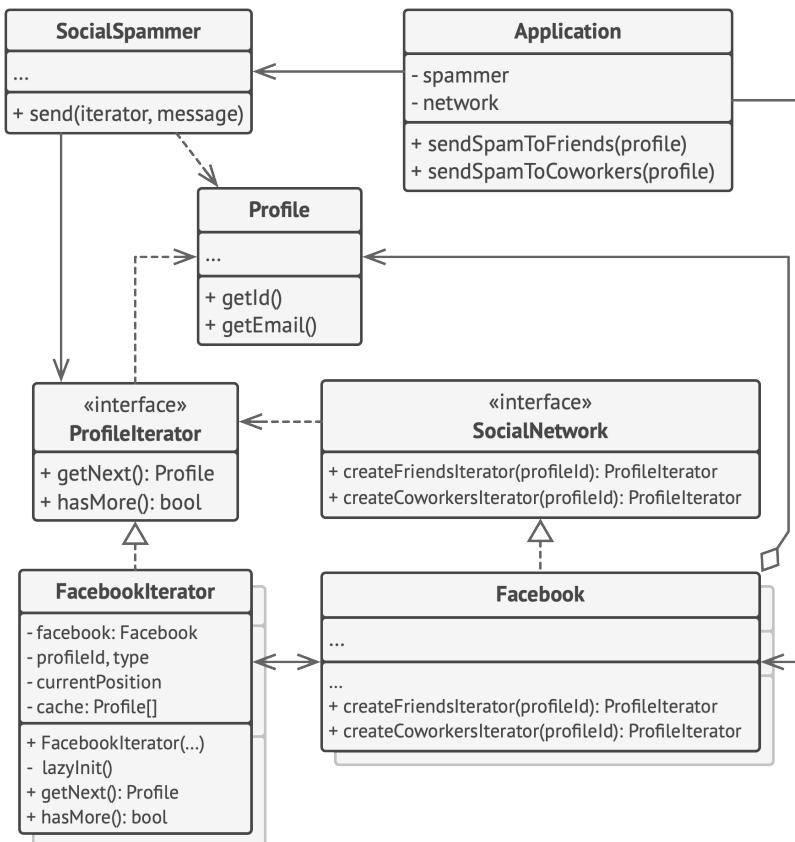
de retour de la méthode doit être l'interface de l'itérateur afin de permettre aux collections concrètes de renvoyer tous types d'itérateurs.

4. Les **Collections Concètes** renvoient les nouvelles instances de la classe concrète de l'itérateur lorsque le client en demande une. Vous vous demandez peut-être où l'on doit mettre le reste du code de la collection. Ne vous inquiétez pas, il devrait être dans la même classe. Ces détails ne sont pas très importants pour ce patron de conception, je me permets donc de les omettre.
5. Le **Client** manipule les collections et les itérateurs grâce à leurs interfaces. Ceci permet de ne pas coupler le client avec les classes concrètes et d'utiliser différents itérateurs et collections avec le même code client.

En général, les clients ne créent pas les itérateurs, ils les récupèrent auprès des collections. Mais dans certains cas, le client peut en créer un directement. Le client peut par exemple définir son propre itérateur spécial.

Pseudo-code

Dans cet exemple, le patron de conception **Itérateur** va servir à parcourir un type spécial de collection qui encapsule l'accès au graphe social de Facebook. Cette collection procure plusieurs itérateurs qui parcourent les profils de différentes manières.



Un exemple d'itération sur les profils sociaux.

L’itérateur des ‘amis’ peut être utilisé pour parcourir les amis d’un profil donné. L’itérateur des ‘collègues’ fait la même chose, sauf qu’il ignore les amis qui ne travaillent pas dans la même entreprise que la personne sélectionnée. Ces deux itérateurs implémentent une interface commune qui permet aux clients d’aller chercher les profils sans tenir compte des détails de l’implémentation, comme l’identification ou l’envoi de requêtes REST.

Le code client n'est pas couplé aux classes concrètes, car il manipule les collections et les itérateurs en passant par leurs interfaces. Si vous décidez de connecter votre application à un réseau social, vous devez simplement fournir de nouvelles classes de collections et d'itérateurs sans toucher au code existant.

```
1 // L'interface collection doit déclarer une fabrique pour créer
2 // des itérateurs. Si vous avez besoin de plusieurs types
3 // d'itérations dans votre programme, vous pouvez déclarer
4 // plusieurs méthodes.
5 interface SocialNetwork is
6     method createFriendsIterator(profileId):ProfileIterator
7     method createCoworkersIterator(profileId):ProfileIterator
8
9
10 // Chaque collection concrète est couplée à un ensemble de
11 // classes concrètes Itérateur qu'elle retourne. Mais le client
12 // en est indépendant, car la signature de ces méthodes renvoie
13 // des interfaces itérateur.
14 class Facebook implements SocialNetwork is
15     // ...Le code principal de la collection devrait être écrit
16     // ici...
17
18     // Code de création de l'itérateur.
19     method createFriendsIterator(profileId) is
20         return new FacebookIterator(this, profileId, "friends")
21     method createCoworkersIterator(profileId) is
22         return new FacebookIterator(this, profileId, "coworkers")
23
24
```

```
25 // L'interface commune à tous les itérateurs.  
26 interface ProfileIterator is  
27     method getNext():Profile  
28     method hasMore():bool  
29  
30  
31 // La classe concrète Itérateur.  
32 class FacebookIterator implements ProfileIterator is  
33     // L'itérateur a besoin d'une référence à la collection  
34     // qu'il parcourt.  
35     private field facebook: Facebook  
36     private field profileId, type: string  
37  
38     // Un objet itérateur parcourt la collection indépendamment  
39     // des autres itérateurs. Il doit donc stocker l'état de  
40     // l'itération.  
41     private field currentPosition  
42     private field cache: array of Profile  
43  
44     constructor FacebookIterator/facebook, profileId, type) is  
45         this.facebook = facebook  
46         this.profileId = profileId  
47         this.type = type  
48  
49     private method lazyInit() is  
50         if (cache == null)  
51             cache = facebook.socialGraphRequest(profileId, type)  
52  
53     // Chaque classe concrète Itérateur a sa propre  
54     // implémentation de l'interface commune Itérateur.  
55     method getNext() is  
56         if (hasMore())
```

```
57     result = cache[currentPosition]
58     currentPosition++
59     return result
60
61     method hasMore() is
62         lazyInit()
63         return currentPosition < cache.length
64
65
66 // Voici une astuce très pratique : vous pouvez passer un
67 // itérateur à une classe du client, plutôt que de lui donner
68 // accès à une collection complète. Cette manipulation vous
69 // permet de ne pas exposer la collection au client.
70 //
71 // Elle vous permet également de modifier la façon dont le
72 // client fonctionne avec les collections lors de son exécution,
73 // en lui passant un itérateur différent. Cette manipulation
74 // serait impossible si le code client était couplé aux classes
75 // concrètes de l'itérateur.
76 class SocialSpammer is
77     method send(iterator: ProfileIterator, message: string) is
78         while (iterator.hasMore())
79             profile = iterator.getNext()
80             System.sendEmail(profile.getEmail(), message)
81
82
83 // La classe application configure les collections et les
84 // itérateurs, puis les envoie au code client.
85 class Application is
86     field network: SocialNetwork
87     field spammer: SocialSpammer
88
```

```
89  method config() is
90      if working with Facebook
91          this.network = new Facebook()
92      if working with LinkedIn
93          this.network = new LinkedIn()
94      this.spammer = new SocialSpammer()
95
96  method sendSpamToFriends(profile) is
97      iterator = network.createFriendsIterator(profile.getId())
98      spammer.send(iterator, "Very important message")
99
100 method sendSpamToCoworkers(profile) is
101     iterator = network.createCoworkersIterator(profile.getId())
102     spammer.send(iterator, "Very important message")
```

💡 Possibilités d'application

- 💡 Utilisez le patron de conception itérateur quand la structure de données de votre collection est trop complexe et que vous voulez cacher ces détails aux clients (parce que c'est pratique ou pour des raisons de sécurité).
- ⚡ L'itérateur encapsule le détail du fonctionnement de la structure complexe de données et passe plusieurs méthodes simples au client qui lui permettent d'accéder aux éléments de la collection. En plus d'être très pratique pour le client, ce fonctionnement permet de protéger la collection d'actions maladroites ou malicieuses que le client pourrait entreprendre s'il travaillait directement avec la collection.

Utilisez l'itérateur pour que le code du parcours soit le moins dupliqué possible dans votre application.

 Un algorithme qui permet une itération complexe sur la collection a tendance à être très volumineux. Si on le place à l'intérieur de la logique de l'application, cela va masquer la responsabilité du code original et le rendre moins maintenable. Pour rendre le code encore plus simple et propre, vous pouvez écrire l'algorithme de parcours dans leur itérateur.

Utilisez ce patron pour permettre à votre code de naviguer dans des structures de données complexes ou inconnues.

 Ce patron procure des interfaces génériques pour les collections et les itérateurs. Si votre code utilise bien ces interfaces, il va pouvoir manipuler toutes les collections (et leurs itérateurs) qui les implémentent.

Mise en œuvre

1. Déclarez l'interface itérateur. Elle doit comporter au moins une méthode qui récupère le prochain élément de la collection. Mais pour qu'elle soit vraiment utile, vous devriez en ajouter d'autres, par exemple récupérer l'élément précédent, connaître la position actuelle ou vérifier la fin de l'itération.
2. Déclarez l'interface de la collection et écrivez une méthode pour récupérer les itérateurs. Le type de la valeur de retour

doit être l'interface de l'itérateur. Vous pouvez déclarer des itérateurs supplémentaires si vous pensez en utiliser d'autres.

3. Mettez en place les classes concrètes des itérateurs pour les collections que vous allez parcourir. Un objet itérateur ne peut être lié qu'à une seule instance d'une collection. En général, ce lien est établi dans le constructeur de l'itérateur.
4. Implémentez l'interface de la collection dans les classes de collection. Son but est de fournir un raccourci pour le client qui crée les itérateurs spécifiques à chaque classe de la collection. L'objet collection doit s'envoyer lui-même au constructeur de l'itérateur pour établir un lien entre eux.
5. Écumez le code et remplacez toutes les occurrences de parcours de collection par vos itérateurs. Le client va chercher un nouvel itérateur chaque fois qu'il parcourt les éléments d'une collection.

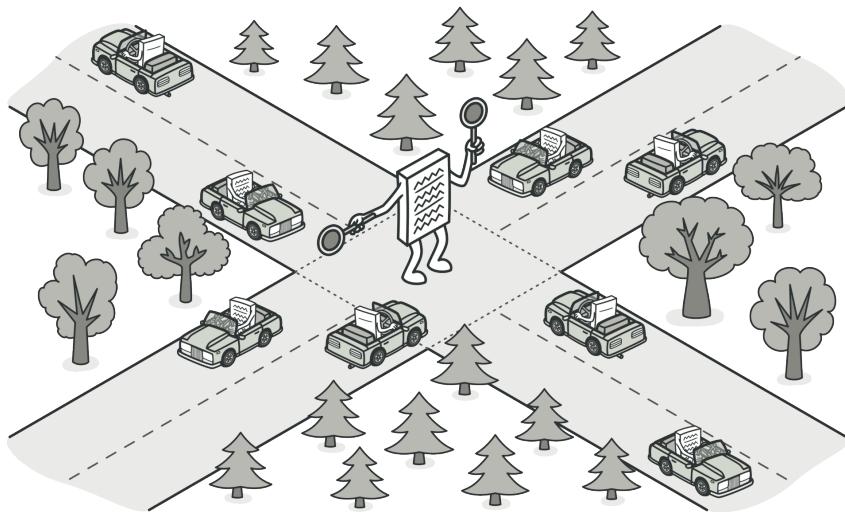
Avantages et inconvénients

- ✓ *Principe de responsabilité unique.* Vous allez nettoyer le code client et les collections en déplaçant les algorithmes de parcours – souvent très lourds – dans des classes séparées.
- ✓ *Principe ouvert/fermé.* Vous pouvez implémenter de nouveaux types de collections et d'itérateurs et les utiliser avec le code existant sans rien endommager.

- ✓ Vous pouvez parcourir une même collection avec plusieurs itérateurs simultanément, car chacun possède son propre état d'itération.
- ✓ Pour cette même raison, vous pouvez arrêter une itération et la reprendre quand vous le souhaitez.
- ✗ L'utilisation de ce patron est exagérée si votre application ne se sert que de collections simples.
- ✗ Les itérateurs sont parfois moins efficaces que certaines collections spécialisées.

↔ Liens avec les autres patrons

- Vous pouvez utiliser les **Itérateurs** pour parcourir des arbres **Composites**.
- 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.
- Vous pouvez utiliser le **Memento** avec l'**Itérateur** pour récupérer l'état actuel de l'itération et rétablir sa valeur plus tard si besoin.
- Vous pouvez utiliser le **Visiteur** avec l'**Itérateur** pour parcourir une structure de données complexe et lancer un traitement sur ses éléments, même si elles ont des classes différentes.



MÉDIATEUR

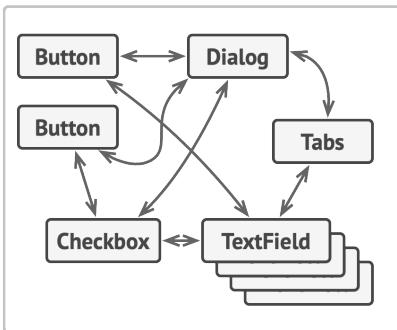
Alias : Intermédiaire, Contrôleur, Mediator

Médiateur est un patron de conception comportemental qui diminue les dépendances chaotiques entre les objets. Il restreint les communications directes entre les objets et les force à collaborer uniquement via un objet médiateur.

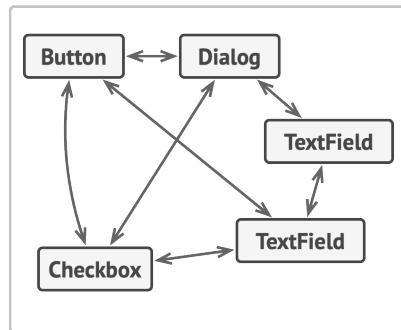
Problème

Prenons une boîte de dialogue qui crée et modifie des profils client. Elle comporte plusieurs contrôles de formulaires comme des champs de texte, des cases à cocher, des boutons, etc.

Boîte de dialogue du profil



Boîte de dialogue de l'identification



Les dépendances entre les éléments de l'interface utilisateur peuvent devenir chaotiques avec l'évolution de l'application.

Certains éléments du formulaire peuvent interagir. Par exemple, en cochant la case « J'ai un chien », vous allez révéler un champ de texte caché qui vous permet d'écrire le nom du chien. Vous pourriez également avoir un bouton Envoyer qui doit valider les valeurs de tous les champs avant de sauvegarder les données.



Les éléments peuvent avoir beaucoup de liens. De plus, la modification de certains éléments peut influer sur d'autres.

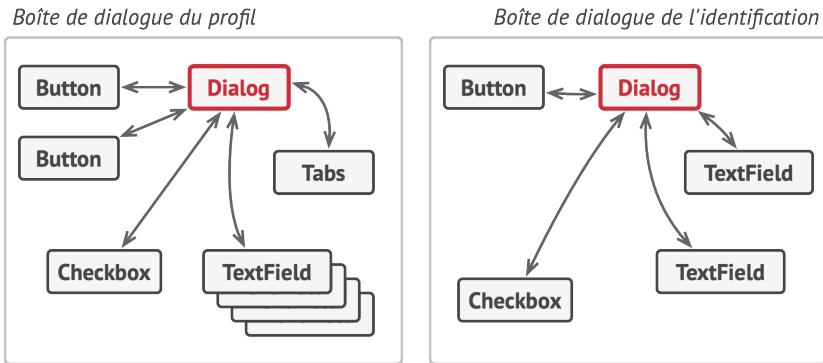
En écrivant directement la logique dans le code des éléments du formulaire, vous rendez les classes de ces éléments bien plus difficiles à réutiliser dans l'application. Par exemple, vous ne pourrez pas utiliser la classe de la case à cocher dans un autre formulaire, car elle est couplée avec le champ de texte du chien. Vous êtes par conséquent obligé d'utiliser toutes les classes du formulaire du profil si vous voulez vous servir de l'une d'entre elles.

😊 Solution

Le patron de conception médiateur vous propose de mettre fin à toutes les communications directes entre les composants et de rendre ces derniers indépendants les uns des autres. À la place, ces composants collaborent indirectement en utilisant un objet spécial médiateur qui redirige les appels vers les composants appropriés. Ainsi, les composants ne reposent que sur une seule classe médiateur plutôt que d'être couplés à de nombreux collègues.

Dans notre exemple qui porte sur l'édition du formulaire d'un profil, la classe dialogue peut prendre le rôle du médiateur.

Elle connaît déjà probablement tous ses sous-éléments, vous n'avez donc pas besoin d'y ajouter des dépendances.



Les éléments de l'UI doivent communiquer directement avec l'objet médiateur.

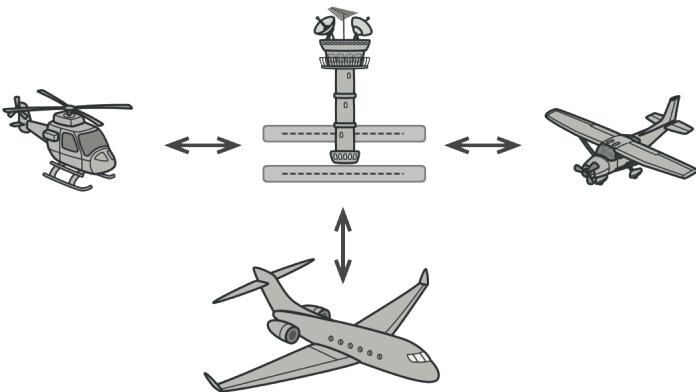
Le changement le plus important intervient sur les éléments du formulaire. Prenons le bouton Envoyer. Avant, chaque fois qu'un utilisateur appuyait sur ce bouton, ce dernier devait valider les valeurs des éléments individuels du formulaire. À présent, il se contente d'informer la classe dialogue lorsqu'un clic a lieu. Après avoir reçu cette notification, la classe dialogue effectue la validation ou délègue la tâche aux différents éléments. Le bouton n'est ainsi couplé qu'avec la classe dialogue, au lieu d'être relié à un paquet d'éléments.

Vous pouvez pousser le vice plus loin et diminuer encore plus cette dépendance en extrayant l'interface commune pour toutes ces boîtes de dialogue. L'interface devrait déclarer la méthode de notification que tous les éléments du formulaire utilisent pour prévenir la classe dialogue des événements qui

les affectent. Notre bouton Envoyer devrait dorénavant pouvoir manipuler n'importe quelle boîte de dialogue qui implémente cette interface.

Ainsi, le patron de conception médiateur vous permet d'envelopper une toile complexe de relations entre divers objets à l'intérieur d'un simple objet médiateur. Moins une classe a de dépendances et plus il est facile de la modifier, de l'étendre ou de la réutiliser.

Analogie



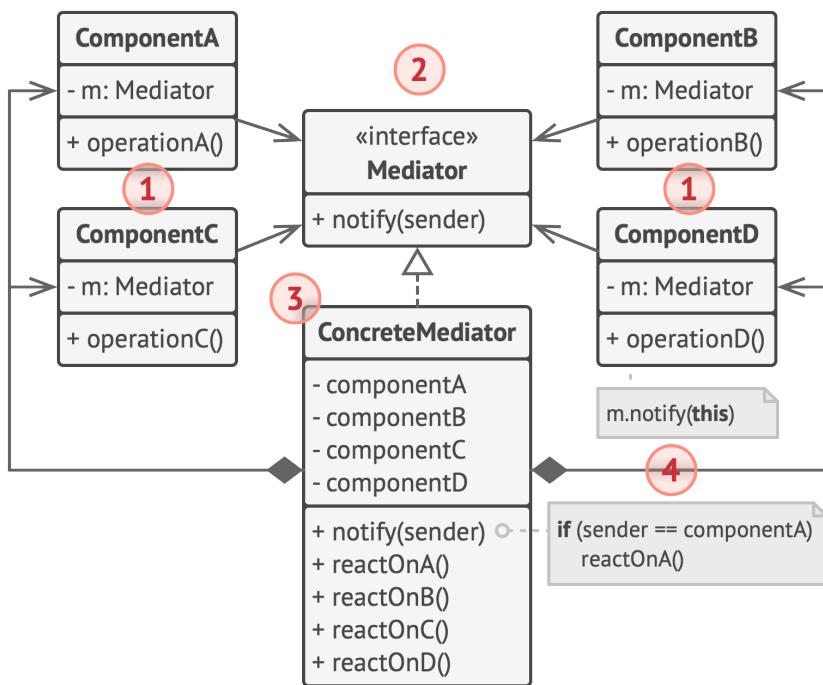
Les pilotes d'avion ne communiquent pas directement ensemble pour déterminer qui sera le prochain à atterrir. Toute communication passe par la tour de contrôle.

Les pilotes qui vont décoller ou atterrir sur la piste ne communiquent pas ensemble directement. Ils s'adressent à un contrôleur aérien qui est assis dans une grande tour près de la piste d'atterrissage. Sans lui, les pilotes seraient obligés de savoir si d'autres avions sont à proximité et décider des priorités d'at-

terrissage avec un ensemble de pilotes. Les accidents d'avion augmenteraient probablement beaucoup.

La tour n'a pas besoin de contrôler la totalité d'un vol. Elle n'est là que pour faire respecter des contraintes au départ et à l'arrivée, pour faire en sorte que les pilotes n'aient pas trop de paramètres à gérer.

Structure



1. Les classes **Composant** contiennent de la logique métier. Chaque composant possède une référence vers un médiateur, déclaré avec le type de l'interface médiateur. Le composant ne connaît pas la classe du médiateur, vous pouvez ainsi réutiliser

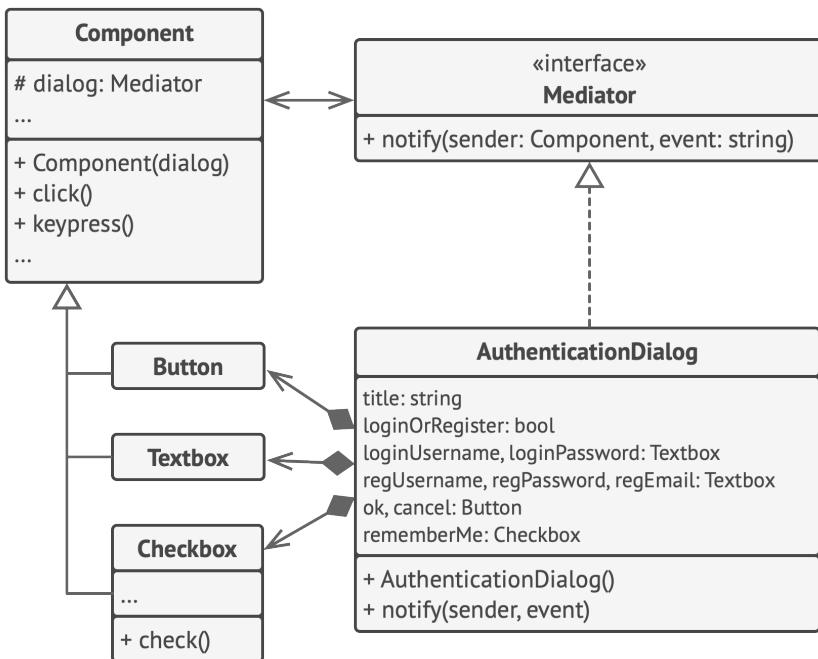
les mêmes composants dans d'autres programmes en les liant à un autre médiateur.

2. L'interface **Médiateur** déclare des méthodes pour communiquer avec les composants et n'est généralement dotée que d'une seule méthode de notification. Les composants peuvent passer n'importe quel contexte en paramètre de cette méthode (ce qui inclut leurs propres objets), mais en évitant de provoquer un couplage entre un composant récepteur et la classe du demandeur.
3. Les **Médiateurs Concrets** encapsulent les relations entre les divers composants. Les médiateurs concrets gardent souvent des références vers les composants qu'ils gèrent et s'occupent même parfois de leur cycle de vie.
4. Les composants ne doivent pas avoir de visibilité sur les autres composants. S'il arrive quelque chose d'important à un composant, il doit seulement en prévenir le médiateur. Quand ce dernier reçoit la notification, il doit pouvoir facilement identifier le demandeur, ce qui peut suffire pour déterminer le composant qui doit être déclenché en retour.

De son point de vue, le composant ne voit qu'une boîte noire. Le demandeur ne sait pas qui va se charger de sa demande, et le récepteur ignore qui l'a envoyée.

Pseudo-code

Dans cet exemple, le **Médiateur** vous aide à éliminer les dépendances mutuelles entre différentes classes de l'UI (buttons, cases à cocher et libellés de texte).



Structure des classes des boîtes de dialogue de l'UI.

Un élément déclenché par un utilisateur ne communique pas directement avec les autres éléments, même si c'est l'impression qui s'en dégage. À la place, l'élément n'a besoin que de prévenir son médiateur qu'un événement a eu lieu, en lui donnant les informations contextuelles avec la notification.

Dans cet exemple, la boîte de dialogue d'identification prend le rôle du médiateur. Elle sait comment les éléments concrets sont censés collaborer et facilite leur communication indirecte. Lorsque la boîte de dialogue reçoit une notification d'événement, elle sait quel élément est concerné et redirige l'appel en conséquence.

```
1 // L'interface médiateur déclare une méthode qui permet aux
2 // composants d'avertir le médiateur au sujet de divers
3 // événements. Le médiateur peut réagir à ces événements et
4 // passer les appels aux autres composants.
5 interface Mediator is
6     method notify(sender: Component, event: string)
7
8
9 // La classe concrète Médiateur. Les interconnexions entre les
10 // composants individuels ont été démêlées et transférées dans
11 // le médiateur.
12 class AuthenticationDialog implements Mediator is
13     private field title: string
14     private field loginOrRegisterChkBx: Checkbox
15     private field loginUsername, loginPassword: Textbox
16     private field registrationUsername, registrationPassword,
17             registrationEmail: Textbox
18     private field okBtn, cancelBtn: Button
19
20 constructor AuthenticationDialog() is
21     // Crée les composants et passe le médiateur actuel dans
22     // leurs constructeurs pour établir les liens.
23
24 // Le médiateur est averti si un composant est affecté par
```

```
25 // quoi que ce soit. Lorsqu'un médiateur reçoit une
26 // notification, il peut lancer un traitement ou envoyer la
27 // demande à un autre composant.
28 method notify(sender, event) is
29     if (sender == loginOrRegisterChkBx and event == "check")
30         if (loginOrRegisterChkBx.checked)
31             title = "Log in"
32             // 1. Affiche les composants du formulaire
33             // d'identification.
34             // 2. Cache les composants du formulaire
35             // d'inscription.
36     else
37         title = "Register"
38         // 1. Affiche les composants du formulaire
39         // d'inscription.
40         // 2. Cache les composants du formulaire
41         // d'identification.
42
43     if (sender == okBtn && event == "click")
44         if (loginOrRegister.checked)
45             // Essaye de trouver un utilisateur à l'aide de
46             // ses identifiants.
47             if (!found)
48                 // Montre un message d'erreur au-dessus du
49                 // champ identifiant.
50     else
51         // 1. Crée un compte utilisateur en utilisant
52         // les données saisies dans les champs
53         // d'inscription.
54         // 2. Connecte cet utilisateur.
55         // ...
56
```

```
57
58 // Les composants communiquent avec un médiateur en utilisant
59 // son interface. Grâce à cela, vous pouvez utiliser les mêmes
60 // composants dans d'autres contextes en les associant avec
61 // d'autres objets médiateur.
62 class Component is
63     field dialog: Mediator
64
65 constructor Component(dialog) is
66     this.dialog = dialog
67
68 method click() is
69     dialog.notify(this, "click")
70
71 method keypress() is
72     dialog.notify(this, "keypress")
73
74 // Les composants concrets ne communiquent pas ensemble. Leur
75 // unique canal de communication leur sert à envoyer des
76 // notifications au médiateur.
77 class Button extends Component is
78     // ...
79
80 class Textbox extends Component is
81     // ...
82
83 class Checkbox extends Component is
84     method check() is
85         dialog.notify(this, "check")
86     // ...
```

Possibilités d'application

-  Utilisez ce patron si vous rencontrez des difficultés pour modifier certaines classes trop fortement couplées avec d'autres.
-  Le médiateur vous permet d'extraire toutes les relations entre les classes dans une classe séparée, en isolant les modifications appliquées à un composant spécifique du reste des composants.
-  Utilisez ce patron quand vous ne pouvez pas réutiliser un composant ailleurs, car il est trop dépendant des autres composants.
-  Après avoir mis en place le médiateur, les composants individuels n'ont plus de visibilité sur les autres composants. Ils peuvent toujours communiquer ensemble mais indirectement, via un objet médiateur. Pour réutiliser un composant dans une application différente, vous n'avez besoin que de lui fournir une classe médiateur.
-  Utilisez le médiateur lorsque vous créez des tonnes de sous-classes pour les composants, juste pour pouvoir bénéficier de leur comportement de base dans différents contextes.
-  Toutes les relations entre composants se trouvent à l'intérieur d'un médiateur. De nouvelles classes médiateur peuvent donc facilement définir de nouveaux moyens de collaboration pour ces composants sans avoir à les modifier.



Mise en œuvre

1. Identifiez des classes qui sont fortement couplées et qui pourraient bénéficier de plus d'indépendance (pour une maintenance plus aisée ou pour faciliter la réutilisation de ces classes).
2. Déclarez l'interface médiateur et écrivez le protocole de communication voulu entre les médiateurs et les différents composants. Dans la plupart des cas, une seule méthode est suffisante pour recevoir les notifications des composants.

Cette interface est cruciale si vous voulez pouvoir réutiliser les classes des composants dans différents contextes. Tant que le composant communique avec le médiateur en passant par l'interface générique, vous pouvez relier le composant avec une implémentation différente du médiateur.

3. Implémentez la classe concrète du médiateur. Faites en sorte que cette classe garde des références vers tous les composants qu'elle gère. Elle en tirera de gros bénéfices.
4. Vous pouvez encore aller plus loin en rendant le médiateur responsable de la création et de la destruction des objets composant. Le médiateur ressemblera ainsi à une fabrique ou à une façade.
5. Les composants doivent avoir une référence vers l'objet médiateur. La connexion est souvent établie dans le constructeur

du composant, dans lequel un objet médiateur est passé en paramètre.

6. Modifiez le code des composants afin qu'ils appellent la méthode de notification du médiateur plutôt que des méthodes écrites dans d'autres composants. Mettez tout le code qui appelle les autres composants dans la classe médiateur, puis appelez-le lorsque le médiateur reçoit des notifications de ce composant.

Avantages et inconvénients

- ✓ *Principe de responsabilité unique.* Vous pouvez mettre les communications entre les différents composants au même endroit, rendant le code plus facile à comprendre et à maintenir.
- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouveaux médiateurs sans avoir à modifier les composants déjà en place.
- ✓ Vous diminuez le couplage entre les différents composants d'un programme.
- ✓ Vous pouvez réutiliser les composants individuels plus facilement.
- ✗ Avec le temps, un médiateur peut évoluer en Objet Omniscent.

↔ Liens avec les autres patrons

- La **Chaîne de responsabilité**, la **Commande**, le **Médiateur** et l'**Observateur** proposent différentes solutions pour associer les demandeurs et les récepteurs.
 - La *chaîne de responsabilité* envoie une demande ordonnée qui est passée tout au long d'une chaîne dynamique de récepteurs potentiels, jusqu'à ce que l'un d'entre eux décide de la traiter.
 - La *commande* établit des connexions unidirectionnelles entre les demandeurs et les récepteurs.
 - Le *médiateur* élimine les liens directs entre les demandeurs et les récepteurs, et les force à communiquer indirectement via un objet médiateur.
 - L'*observateur* permet aux récepteurs de s'inscrire et de se désinscrire dynamiquement à la réception des demandes.
- La **Façade** et le **Médiateur** ont des rôles similaires : ils essayent de faire collaborer des classes étroitement liées.
 - La *façade* définit une interface simplifiée pour un sous-système d'objets, mais elle n'ajoute pas de nouvelles fonctionnalités. Le sous-système lui-même n'a pas connaissance de la façade. Les objets situés à l'intérieur du sous-système peuvent communiquer directement.
 - Le *médiateur* centralise la communication entre les composants du système. Les composants ne voient que l'objet médiateur et ne communiquent pas directement.

- La différence entre le **Médiateur** et l'**Observateur** est souvent très fine. Dans la majorité des cas, vous pouvez implémenter l'un ou l'autre, mais parfois vous pouvez les utiliser simultanément. Regardons comment faire.

Le but principal du *médiateur* est d'éliminer les dépendances mutuelles entre un ensemble de composants du système. À la place, ces composants peuvent devenir dépendants d'un unique objet médiateur. Le but de l'*observateur* est d'établir des connexions dynamiques à sens unique entre les objets, où certains objets peuvent être les subordonnés d'autres objets.

Il existe une implémentation populaire du *médiateur* qui repose sur l'*observateur*. L'objet médiateur joue le rôle du diffuseur et les composants agissent comme des souscripteurs qui s'inscrivent et se désinscrivent des événements du médiateur. Lorsque ce type de conception est mis en place, le *médiateur* ressemble de près à l'*observateur*.

Si vous êtes un peu perdu, rappelez-vous qu'il y a plusieurs manières d'implémenter le médiateur. Par exemple, vous pouvez associer de manière permanente tous les composants au même objet médiateur. Cette implémentation ne ressemblera pas à l'*observateur*, mais sera tout de même une instance du patron de conception médiateur.

Maintenant, imaginez un programme dont tous les composants sont devenus des diffuseurs, permettant des connexions dynamiques les uns avec les autres. Nous n'aurons pas d'objet médiateur centralisé, seulement un ensemble d'observateurs distribués.



MÉMENTO

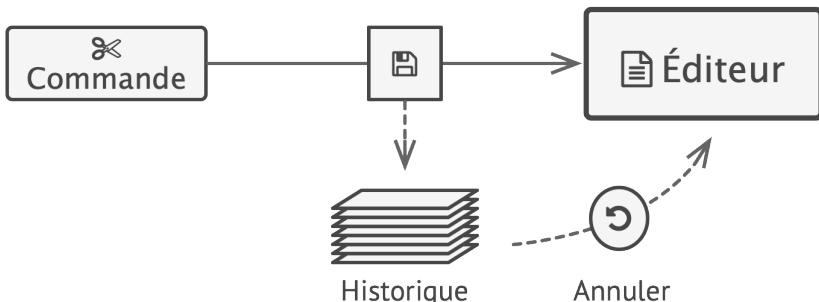
Alias : Jeton, Memento

Mémento est un patron de conception comportemental qui permet de sauvegarder et de rétablir l'état précédent d'un objet sans révéler les détails de son implémentation.

(:() Problème

Imaginez que vous êtes en train de créer un éditeur de texte. En plus de pouvoir écrire du texte, votre éditeur permet de le formater, d'insérer des images alignées, etc.

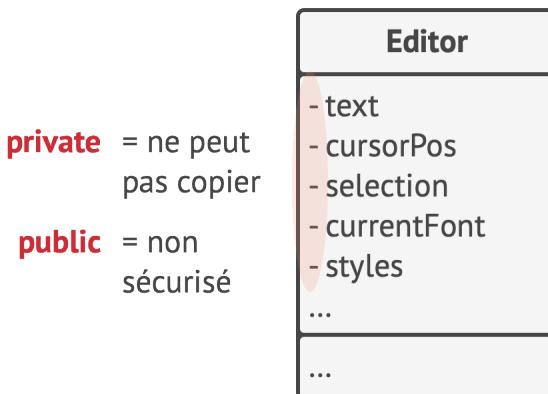
Au bout d'un moment, vous décidez d'ajouter une fonctionnalité qui permet aux utilisateurs d'annuler une action sur le texte. Cette fonctionnalité est devenue si classique au fil des années, que les utilisateurs s'attendent systématiquement à en disposer. Vous choisissez une approche directe pour la mettre en place. Avant d'effectuer une action, l'application enregistre l'état de tous les objets et les sauvegarde. Plus tard, lorsqu'un utilisateur décide d'annuler une action, l'application récupère la dernière sauvegarde de l'historique et s'en sert pour restaurer l'état de tous les objets.



Avant d'effectuer une action, l'application prend un instantané (snapshot) du dernier état des objets. Il peut être utilisé plus tard pour rétablir les objets dans leur ancien état.

Prenons un moment pour réfléchir à ces photos. Comment va-t-on s'y prendre pour les créer? Vous allez probablement devoir parcourir tous les attributs d'un objet et copier leurs valeurs quelque part. Cependant, cela ne fonctionnera que si le contenu de l'objet ne possède pas trop de restrictions. Malheureusement, la plupart des objets ne se laissent pas approcher si facilement et cachent les données importantes dans des attributs privés.

Laissons de côté ce problème pour le moment et considérons que nos objets se comportent en bons hippies : ils préfèrent avoir des relations ouvertes et garder leur état public. Même si cette approche nous permet de produire des instantanés de l'état des objets à volonté, de sérieux problèmes demeurent.



Comment faire une copie de l'état privé d'un objet?

Dans le futur, vous pourriez décider de refactoriser certaines classes de l'éditeur, ou d'ajouter ou de supprimer certains attributs. Cela semble facile à première vue, mais vous allez de-

voir également modifier les classes qui ont la responsabilité de créer la copie de l'état des objets concernés.

Mais ce n'est pas tout ! Regardons un peu du côté des « photos » de l'état de l'éditeur. Quel genre de données contiennent-elles ? On doit au moins avoir accès aux coordonnées du curseur de la souris, à la position de la barre de défilement, etc. Pour prendre une photo, vous devez récupérer ces valeurs et les mettre dans un conteneur.

Vous allez très certainement stocker un paquet de ces conteneurs dans une liste qui représente l'historique. Ces conteneurs seront probablement les objets d'une classe. Cette classe possèdera très peu de méthodes, mais aura beaucoup d'attributs qui répliquent l'état de l'éditeur. Pour permettre à d'autres objets de lire et d'écrire les données d'une photo, vous allez devoir rendre ses attributs publics, ce qui exposerait les états de l'éditeur, qu'ils soient privés ou non. Les autres classes deviendraient dépendantes au moindre changement apporté à la classe photo. Si nous rendons ses attributs et méthodes privés, ces changements ne seraient de toute façon pas répercutés sur les autres classes.

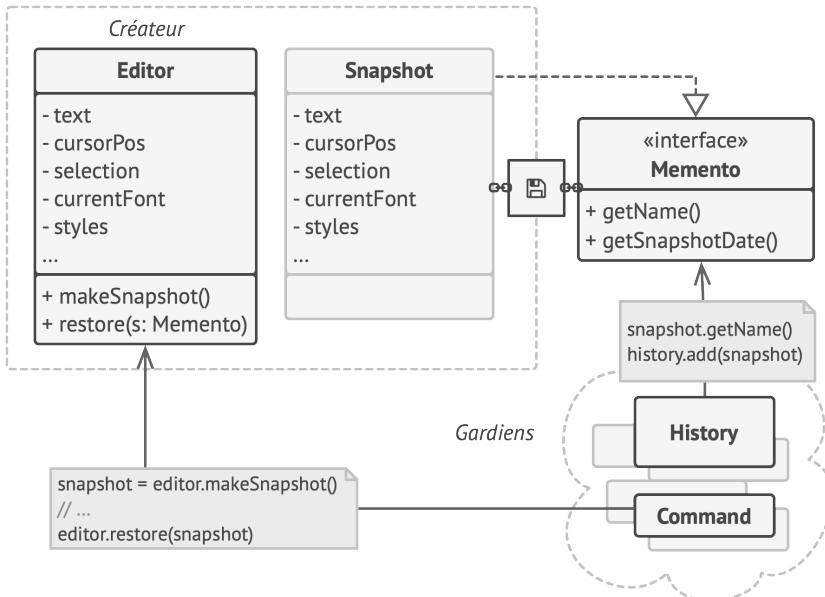
Il semblerait que nous sommes dans une impasse : soit on expose tous les détails d'une classe, ce qui les rend trop fragiles, soit on restreint l'accès à leur état, ce qui empêche de prendre des instantanés. Dispose-t-on d'une autre technique pour implémenter un « annuler » ?

Solution

Tous les problèmes que nous rencontrons sont provoqués par une mauvaise encapsulation. Certains objets essayent d'en faire plus que ce qu'ils sont supposés faire. Pour récupérer les données dont ils ont besoin pour effectuer une action, ils en-vahissent l'espace personnel des autres objets, plutôt que de leur demander d'exécuter l'action eux-mêmes.

Le memento délègue la création des états des photos à leur propriétaire, l'objet *créateur* (originator). Plutôt que d'essayer de copier l'état de l'éditeur depuis « l'extérieur », la classe de l'éditeur de texte peut prendre la photo elle-même, car elle a accès à son propre état.

Ce patron propose de stocker la copie de l'état de l'objet dans un objet spécial appelé *memento*. Son contenu n'est accessible que pour l'objet qui l'a créé. Les autres objets peuvent communiquer avec les méméntos via une interface limitée qui leur permet de récupérer certaines métadonnées de la photo (date de création, nom de l'action effectuée, etc.), mais pas l'état de l'objet original contenu dans la photo.



Le créateur a un accès total au memento, alors que le gardien ne peut que consulter les métadonnées.

Une telle stratégie vous permet de stocker des mémentos à l'intérieur d'autres objets que l'on appelle *gardiens* (caretakers). Le gardien ne manipule le memento qu'en passant par l'interface limitée. Il ne peut donc pas modifier les états qui y sont stockés. De son côté, le créateur rétablit les états qu'il désire, car il a accès à tous les attributs du memento.

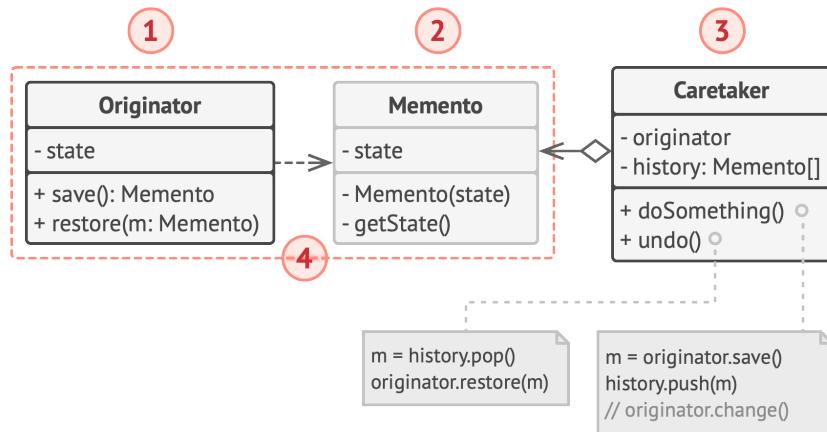
Dans l'exemple de notre éditeur de texte, nous pouvons créer une classe historique séparée qui prend le rôle du gardien. La pile de mémentos stockée dans le gardien va grandir chaque fois que l'éditeur va effectuer une action. Vous pouvez même afficher cette pile dans l'interface utilisateur, afin de montrer les dernières opérations effectuées par un utilisateur.

Lorsqu'un utilisateur veut annuler une action, l'historique prend le memento le plus récent de la pile et l'envoie à l'éditeur en demandant un rollback. Comme l'éditeur possède un accès total au memento, il change lui-même son état en copiant les valeurs du memento.

Structure

Implémentation basée sur des classes imbriquées

L'implémentation classique du patron repose sur le principe des classes imbriquées, disponibles dans de nombreux langages de programmation populaires (comme le C++, C# et Java).



1. La classe **Créateur** (Originator) peut prendre des instantanés de son propre état, et le restaurer à volonté.

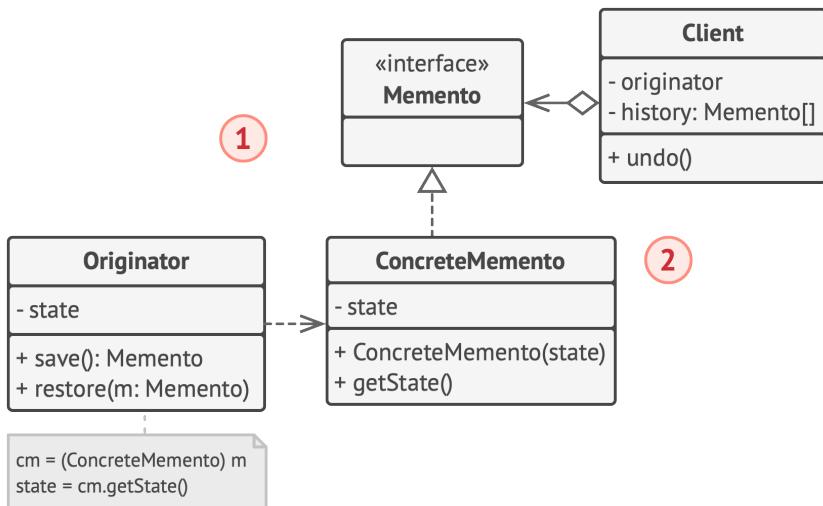
2. Le **Mémento** est un objet de valeur qui joue le rôle d'un instantané d'un état du créateur. En général, le memento n'est pas modifiable et écrit ses données une seule fois dans son constructeur.
3. Le **Gardien** sait « quand » et « pourquoi » il doit photographier l'état du créateur, mais il sait également quand l'état doit être restauré.

Le gardien peut conserver la trace de l'historique du créateur en enregistrant une pile de mémentos. Lorsque le créateur veut revenir dans le passé, le gardien récupère l'élément du haut de la pile et l'envoie à la méthode de restauration du créateur.

4. Dans cette implémentation, la classe memento est imbriquée à l'intérieur du créateur. Ceci permet au créateur d'accéder aux attributs et méthodes du memento, même s'ils sont privés. Le gardien quant à lui n'a qu'un accès limité aux attributs et méthodes du memento : on le laisse entasser les mémentos dans la pile, mais il ne peut pas les modifier.

Implémentation basée sur une interface intermédiaire

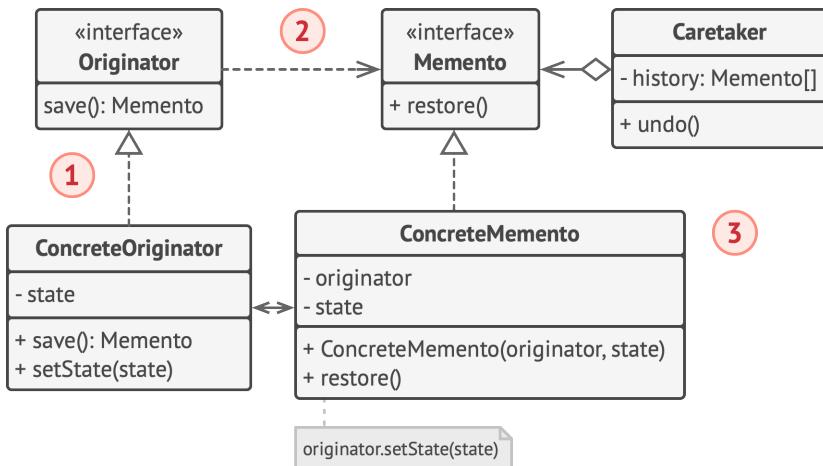
Voici une autre possibilité très pratique pour les langages qui ne gèrent pas les classes imbriquées (oui PHP, je te regarde).



1. En l'absence de classes imbriquées, vous pouvez restreindre l'accès aux attributs du memento en établissant une convention : les gardiens ne vont pouvoir manipuler un memento qu'à travers une interface intermédiaire déclarée explicitement. Cette interface ne déclare que des méthodes liées aux métadonnées du memento.
2. De leur côté, les créateurs peuvent manipuler directement les objets memento, accéder à leurs attributs et méthodes déclarés dans la classe memento. L'inconvénient de cette technique est que vous devez déclarer tous les membres du memento en public.

Implémentation avec une encapsulation encore plus stricte

Si vous ne voulez vraiment pas que les autres classes accèdent au créateur en passant par le memento, vous pouvez vous y prendre différemment.

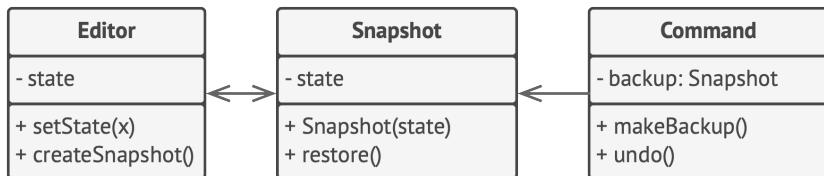


1. Cette implémentation permet de gérer plusieurs créateurs et plusieurs mémentos. Chaque créateur manipule sa propre classe memento. Les créateurs et les mémentos n'exposent pas leur état aux autres.
2. Il est désormais explicité que les gardiens ne peuvent pas modifier l'état stocké dans le memento. De plus, la classe gardien n'est plus couplée avec le créateur, car la méthode de restauration est maintenant définie dans la classe memento.
3. Chaque memento devient lié au créateur qui l'a produite. Le créateur s'envoie lui-même et toutes les valeurs de son état

au constructeur du memento. Grâce à l'étroite proximité de ces deux classes, un memento peut restaurer l'état de son créateur, tant que ce dernier a bien défini les setters appropriés.

Pseudo-code

Cet exemple utilise le patron **Commande** en plus du memento pour stocker les photos de l'état de l'éditeur de texte complexe, et rétablit un état précédent lorsqu'on le lui demande.



Sauvegarder des photos de l'état de l'éditeur de texte.

Les objets commande prennent le rôle de gardiens. Ils vont chercher le memento de l'éditeur avant de lancer les actions déclenchées par les commandes. Lorsqu'un utilisateur veut annuler l'action la plus récente, l'éditeur peut se servir du memento stocké dans cette commande pour revenir à son état précédent.

La classe memento ne déclare aucun attribut public, ni de getters et de setters. Aucun objet ne peut modifier son contenu. Les mémentos sont reliés à l'objet éditeur qui les a créés. Le memento peut aussi restaurer l'état de l'éditeur qui lui est associé, en passant les données dans les setters de l'objet éditeur. Comme les mémentos sont liés à des objets

éditeur spécifiques, votre application peut gérer plusieurs fenêtres avec des éditeurs indépendants et une pile centralisée d'états à rétablir.

```
1 // Le créateur conserve des données importantes qui varient
2 // parfois avec le temps. Il définit également une méthode pour
3 // sauvegarder son état dans un memento, et une autre méthode
4 // pour rétablir cet état.
5 class Editor is
6     private field text, curX, curY, selectionWidth
7
8     method setText(text) is
9         this.text = text
10
11    method setCursor(x, y) is
12        this.curX = x
13        this.curY = y
14
15    method setSelectionWidth(width) is
16        this.selectionWidth = width
17
18    // Sauvegarde l'état actuel dans un memento.
19    method createSnapshot():Snapshot is
20        // Le memento est un objet non modifiable. C'est pour
21        // cela que le créateur passe son état dans les
22        // paramètres du constructeur du memento.
23        return new Snapshot(this, text, curX, curY, selectionWidth)
24
25    // La classe memento conserve un ancien état de l'éditeur.
26    class Snapshot is
27        private field editor: Editor
```

```
28 private field text, curX, curY, selectionWidth
29
30 constructor Snapshot(editor, text, curX, curY, selectionWidth) is
31     this.editor = editor
32     this.text = text
33     this.curX = x
34     this.curY = y
35     this.selectionWidth = selectionWidth
36
37 // Un objet memento peut être utilisé pour rétablir un
38 // ancien état de l'éditeur.
39 method restore() is
40     editor.setText(text)
41     editor.setCursor(curX, curY)
42     editor.setSelectionWidth(selectionWidth)
43
44 // Un objet commande peut prendre le rôle du gardien. Dans ce
45 // cas, un memento est affecté à la commande juste avant que
46 // l'état du créateur ne change. Lorsqu'une demande de
47 // restauration arrive, il rétablit l'état du créateur à partir
48 // du memento.
49 class Command is
50     private field backup: Snapshot
51
52     method makeBackup() is
53         backup = editor.createSnapshot()
54
55     method undo() is
56         if (backup != null)
57             backup.restore()
58     // ...
```

Possibilités d'application

-  Utilisez le memento si vous voulez prendre des photos de l'état d'un objet afin de pouvoir rétablir un de ses états précédents.
-  Le memento vous permet de faire des copies complètes de l'état d'un objet (attributs privés inclus), et les stocke en dehors de l'objet. Ce patron est surtout connu pour l'utilisation de la fonctionnalité annuler, mais est également indispensable pour les transactions (par exemple, pour permettre le rollback d'une opération en erreur).
-  Utilisez ce patron lorsque vous ne pouvez pas accéder directement aux attributs/getters/setters d'un objet sans enfreindre les principes de l'encapsulation.
-  Le memento rend l'objet responsable de lui-même, ce qui sécurise les données de l'état de l'objet. Aucun autre objet ne peut lire les données de la photo, l'objet original est donc complètement sécurisé.

Mise en œuvre

1. Déterminez la classe qui jouera le rôle du créateur. Il est important de savoir si le programme va utiliser un seul objet central ou plusieurs petits objets.
2. Créez la classe memento. Déclarez un à un l'ensemble des attributs qui recopient les attributs de la classe créateur.

3. Rendez la classe memento non modifiable. Un memento ne doit écrire ses données qu'une seule fois via son constructeur. Sa classe ne doit pas avoir de setters.
4. Si votre langage de programmation accepte les classes imbriquées, mettez le memento à l'intérieur du créateur. Sinon, extrayez une interface vide depuis la classe du memento et obligez les autres objets à utiliser cette interface pour y accéder. Vous pouvez ajouter des opérations sur les métadonnées dans l'interface, mais rien qui ne puisse exposer l'état du créateur.
5. Ajoutez une méthode qui crée des mémentos à la classe créateur. Le créateur doit passer son état dans un ou plusieurs paramètres du constructeur du memento.

Le type de la valeur de retour de la méthode doit être l'interface extraite dans l'étape précédente (si vous avez suivi cette étape). La méthode qui crée le memento doit directement manipuler la classe memento.

6. Écrivez la méthode qui permet de rétablir l'état du créateur dans sa propre classe. Il doit prendre un objet memento en paramètre. Si vous avez extrait une interface lors de l'une des étapes précédentes, c'est le type de la valeur de retour que cette méthode doit accepter. Dans ce cas, vous devez caster cet objet dans la classe memento, car le créateur a besoin d'un accès total à cet objet.

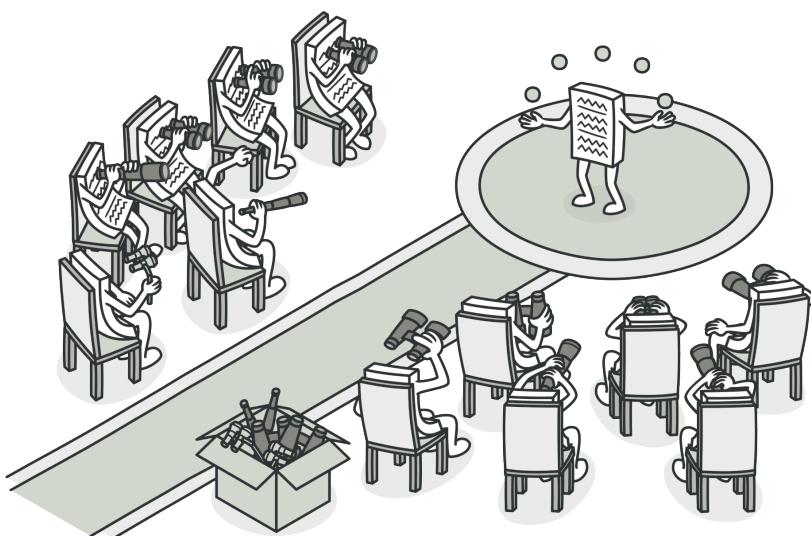
7. Le gardien, que ce soit une commande, un historique ou n'importe quoi d'autre, doit savoir quand demander de nouveaux mémentos au créateur, comment les stocker et quand restaurer le créateur à l'aide d'un memento donné.
8. Le lien entre les gardiens et les créateurs peut être déplacé dans la classe memento. Dans ce cas, chaque memento doit être associé à son propre créateur. La méthode de restauration doit également être déplacée dans la classe memento. Mais faites-le uniquement si la classe memento est imbriquée dans son créateur, ou si la classe du créateur fournit assez de setters pour redéfinir son état.

Avantages et inconvénients

- ✓ Vous pouvez prendre des instantanés de l'état d'un objet tout en respectant son encapsulation.
- ✓ Vous pouvez simplifier le code du créateur en laissant le gardien gérer l'historique de l'état du créateur.
- ✗ L'application pourrait consommer beaucoup de RAM si les clients créent des mémentos trop souvent.
- ✗ Les gardiens doivent garder la trace du cycle de vie des créateurs pour pouvoir détruire les mémentos obsolètes.
- ✗ La majorité des langages de programmation dynamiques comme le PHP, Python ou le JavaScript ne peuvent pas garantir que l'état sauvegardé dans le memento ne sera pas modifié.

↔ Liens avec les autres patrons

- Vous pouvez utiliser la **Commande** et le **Mémento** ensemble pour implémenter la fonctionnalité « annuler ». Dans ce cas, les commandes ont la responsabilité d'exécuter les divers traitements sur un objet cible. Les mémentos sauvegardent l'état de cet objet juste avant le lancement de la commande.
- Vous pouvez utiliser le **Mémento** avec l'**Itérateur** pour récupérer l'état actuel de l'itération et rétablir sa valeur plus tard si besoin.
- Parfois le **Prototype** peut venir remplacer le **Mémento** et proposer une solution plus simple. Cela n'est possible que si l'objet (l'état que vous voulez stocker dans l'historique) est assez direct et ne possède pas de liens vers des ressources externes, ou que les liens sont faciles à recréer.



OBSERVATEUR

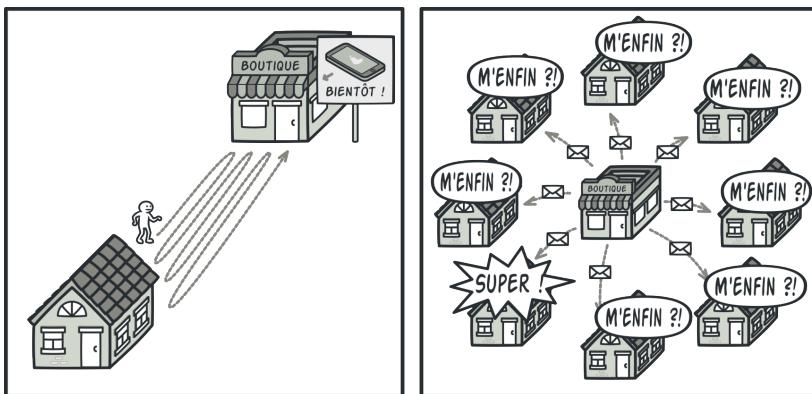
Alias : Dépendants, Diffusion–Souscription, Observer

L'**Observateur** est un patron de conception comportemental qui permet de mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.

(:() Problème

Imaginez que vous avez deux types d'objets : un Client et un Magasin. Le client s'intéresse à une marque spécifique d'un produit (disons que c'est un nouveau modèle d'iPhone) qui sera bientôt disponible dans la boutique.

Le client pourrait se rendre sur place tous les jours et vérifier la disponibilité du produit. Mais comme le produit n'est pas encore prêt, ses allées et venues seraient inutiles.



Se rendre au magasin ou envoyer du spam.

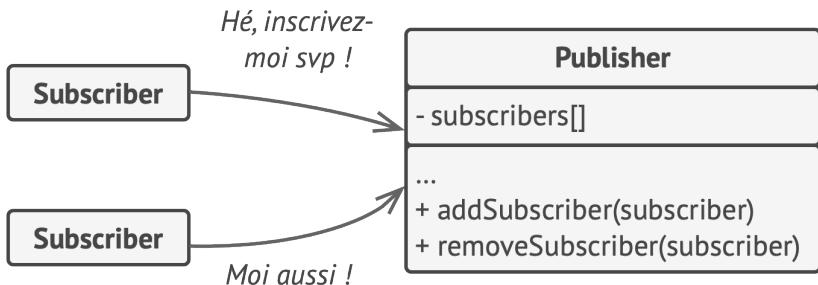
À la place, le magasin pourrait envoyer des tonnes d'e-mails (ce qui peut être vu comme du spam) à leurs clients chaque fois qu'un nouveau produit est disponible. Cette solution économiserait bien des voyages à leurs clients. En contrepartie, le magasin risque de se mettre à dos ceux qui ne sont pas intéressés par les nouveaux produits.

Nous nous retrouvons dans une situation conflictuelle. Soit les clients perdent leur temps à venir vérifier la disponibilité des produits, soit le magasin gâche des ressources pour prévenir des clients qui ne sont pas concernés.

Solution

L'objet que l'on veut suivre est en général appelé *sujet*, mais comme il va envoyer des notifications pour prévenir les autres objets dès qu'il est modifié, nous l'appellerons *diffuseur* (publisher). Tous les objets qui veulent suivre les modifications apportées au diffuseur sont appelés des *souscripteurs* (subscribers).

Le patron de conception Observateur vous propose d'ajouter un mécanisme de souscription à la classe diffuseur pour permettre aux objets individuels de s'inscrire ou se désinscrire de ce diffuseur. Pas d'inquiétude ! Ce n'est pas si compliqué que cela en a l'air. En réalité, ce mécanisme est composé 1) d'un tableau d'attributs qui stocke une liste de références vers les objets souscripteur et 2) de plusieurs méthodes publiques qui permettent d'ajouter ou de supprimer des souscripteurs de cette liste.

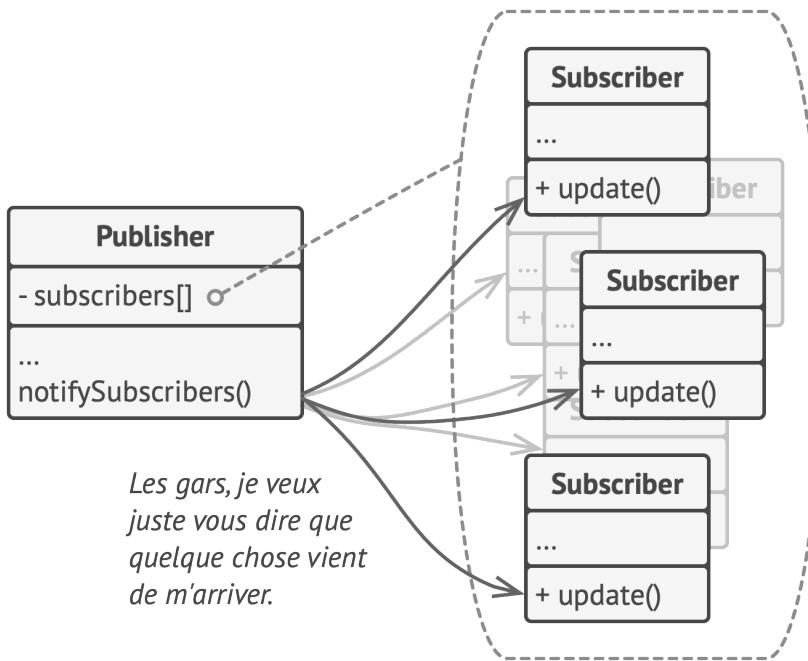


Un mécanisme de souscription qui permet aux objets individuels de s'inscrire aux notifications des événements.

Quand un événement important arrive au diffuseur, il fait le tour de ses souscripteurs et appelle la méthode de notification sur leurs objets.

Les applications peuvent comporter des dizaines de classes souscripteur différentes qui veulent être tenues au courant des événements qui affectent une même classe diffuseur. Vous n'avez sûrement pas envie de coupler le diffuseur à toutes ces classes. De plus, certaines ne seront peut-être pas connues à l'avance, dans les cas où votre classe diffuseur est censée pouvoir être utilisée par d'autres personnes.

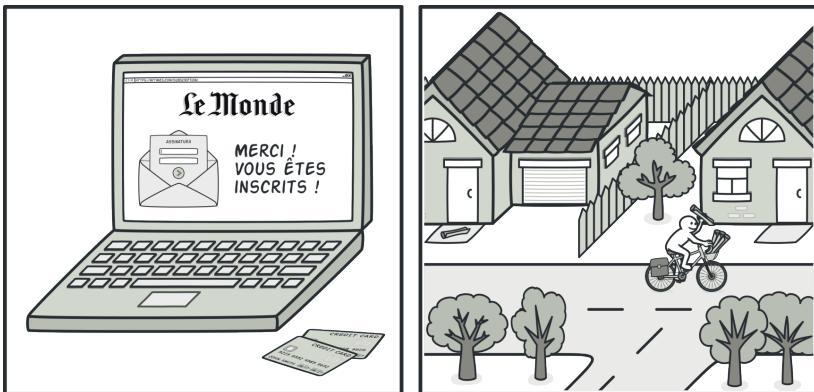
C'est pourquoi il est crucial que tous les souscripteurs implémentent la même interface et qu'elle soit le seul moyen utilisé par le diffuseur pour communiquer avec eux. Elle doit déclarer la méthode de notification avec un ensemble de paramètres que le diffuseur peut utiliser pour envoyer des données contextuelles avec la notification.



Le diffuseur envoie des notifications aux souscripteurs en appelant la méthode de notification spécifique sur leurs objets.

De plus, les diffuseurs doivent tous suivre la même interface si votre application en comporte plusieurs types et que vous voulez que vos souscripteurs soient tous compatibles avec eux. Cette interface doit contenir quelques méthodes de souscription et elle doit permettre aux souscripteurs d'observer les états du diffuseur sans le coupler avec leurs classes concrètes.

Analogie

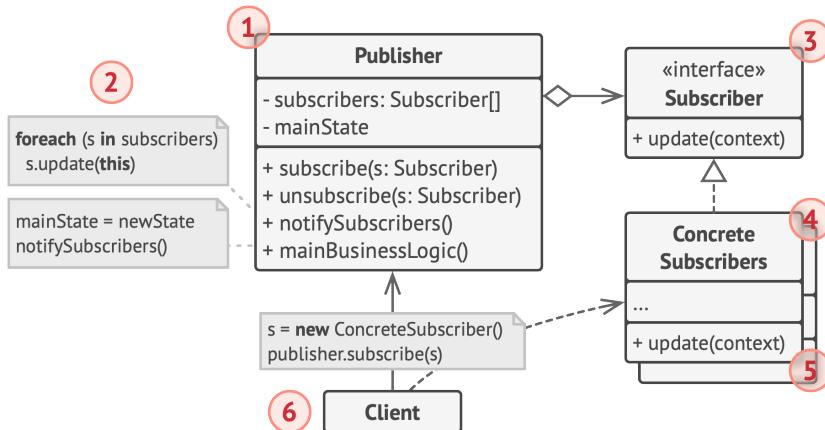


Abonnement aux magazines et aux journaux.

Lorsque vous vous inscrivez à un journal ou à un magazine, vous n'avez plus besoin de vous rendre en magasin pour vérifier si le dernier numéro est sorti. À la place, le diffuseur vous envoie directement les nouveaux numéros dans votre boîte aux lettres dès qu'ils le publient, ou même parfois en avance.

Le diffuseur garde une liste de souscripteurs et connaît les magazines qui les intéressent. S'ils ne souhaitent plus recevoir les nouveaux numéros, les souscripteurs peuvent quitter la liste à n'importe quel moment.

Structure

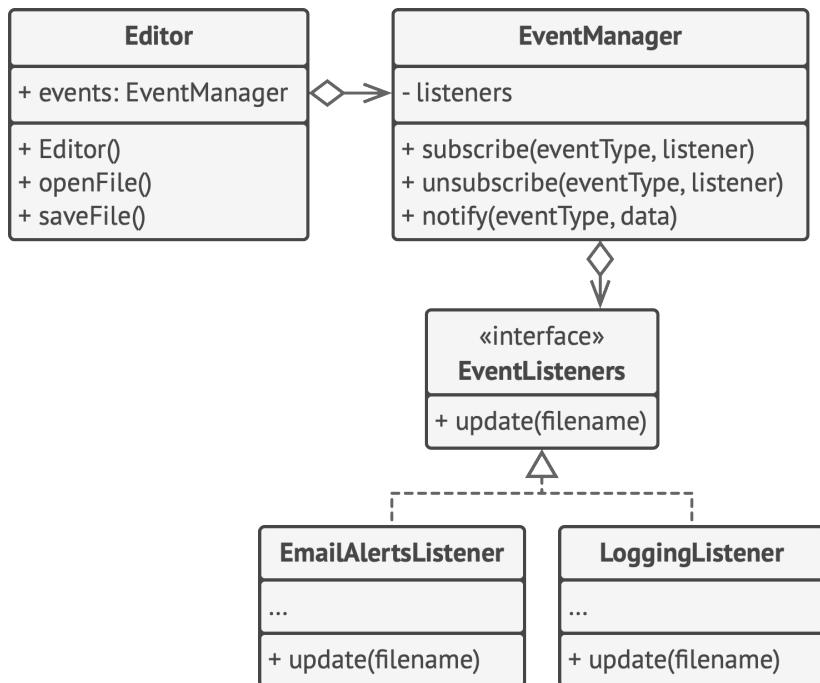


1. Le **Diffuseur** envoie des événements intéressants à d'autres objets. Ces événements se produisent quand le diffuseur change d'état ou exécute certains comportements. Le diffuseur possède une infrastructure d'inscription qui permet aux nouveaux souscripteurs de rejoindre la liste et aux souscripteurs actuels de la quitter.
2. Quand un nouvel événement survient, le diffuseur parcourt la liste d'inscriptions et appelle la méthode de notification déclarée dans l'interface des souscripteurs sur chaque objet souscripteur.
3. L'interface **Souscripteur** déclare les méthodes de notification. Dans la majorité des cas, il n'y a qu'une seule méthode `update`. Elle peut prendre plusieurs paramètres pour que le diffuseur leur envoie plus de détails concernant la modification.

4. Les **Souscripteurs Concrets** exécutent certaines actions en réponse aux notifications envoyées par le diffuseur. Toutes ces classes doivent implémenter la même interface pour ne pas coupler le diffuseur avec leurs classes concrètes.
5. En général, les souscripteurs ont besoin de détails à propos du contexte afin d'exécuter correctement la mise à jour. C'est pour cela que les diffuseurs passent souvent des données du contexte en paramètre de la méthode de notification. Le diffuseur peut même s'envoyer lui-même en paramètre et laisser les souscripteurs récupérer directement les données nécessaires.
6. Le **Client** crée des objets diffuseur et Souscripteur séparément et inscrit les souscripteurs aux mises à jour du diffuseur.

Pseudo-code

Dans cet exemple, le patron de conception **Observateur** permet à l'objet éditeur de texte d'avertir d'autres objets de service des changements de son état.



Envoyer des notifications aux objets au sujet d'événements qui affectent d'autres objets.

La liste des souscripteurs est compilée dynamiquement : les objets peuvent commencer ou arrêter de suivre les notifications lors du lancement de l'application, en fonction du comportement voulu.

Dans cette implémentation, la classe éditeur ne gère pas la liste des souscripteurs toute seule. Elle délègue la tâche à l'objet spécial assistant dont c'est la seule tâche. Vous pouvez moduler cet objet afin qu'il serve de centrale de distribution, transformant n'importe quel objet en diffuseur.

Tant que les classes diffuseur passent toutes par la même interface pour communiquer avec les souscripteurs, l'ajout de nouveaux souscripteurs ne nécessite aucun changement dans les classes diffuseur.

```
1 // La classe de base diffuseur contient le code pour
2 // l'inscription et les méthodes de notification.
3 class EventManager is
4     private field listeners: hash map of event types and listeners
5
6     method subscribe(eventType, listener) is
7         listeners.add(eventType, listener)
8
9     method unsubscribe(eventType, listener) is
10        listeners.remove(eventType, listener)
11
12    method notify(eventType, data) is
13        foreach (listener in listeners.of(eventType)) do
14            listener.update(data)
15
16 // Le diffuseur concret abrite de la logique métier dédiée à
17 // certains souscripteurs. Nous pourrions dériver cette classe
18 // depuis le diffuseur de base, mais ce n'est pas toujours
19 // possible, car le diffuseur concret pourrait déjà être une
20 // sous-classe. Dans ce cas, vous pouvez utiliser la composition
21 // pour effectuer le lien avec la logique de souscription, comme
22 // effectué ci-dessous :
23 class Editor is
24     public field events: EventManager
25     private field file: File
26
```

```
27 constructor Editor() is
28     events = new EventManager()
29
30     // Les méthodes de la logique métier peuvent prévenir les
31     // souscripteurs de toute modification.
32 method openFile(path) is
33     this.file = new File(path)
34     events.notify("open", file.name)
35
36 method saveFile() is
37     file.write()
38     events.notify("save", file.name)
39
40     // ...
41
42
43     // Voici l'interface des souscripteurs. Si votre langage de
44     // programmation prend en charge les types fonctionnels, vous
45     // pouvez remplacer toute la hiérarchie des souscripteurs par un
46     // ensemble de fonctions.
47 interface EventListener is
48     method update(filename)
49
50     // Les souscripteurs concrets réagissent aux mises à jour de
51     // leur diffuseur.
52 class LoggingListener implements EventListener is
53     private field log: File
54     private field message: string
55
56     constructor LoggingListener(log_filename, message) is
57         this.log = new File(log_filename)
58         this.message = message
```

```
59
60     method update(filename) is
61         log.write(replace('%s',filename,message))
62
63 class EmailAlertsListener implements EventListener is
64     private field email: string
65     private field message: string
66
67 constructor EmailAlertsListener(email, message) is
68     this.email = email
69     this.message = message
70
71 method update(filename) is
72     system.email(email, replace('%s',filename,message))
73
74
75 // Une application peut configurer des diffuseurs et des
76 // souscripteurs à l'exécution.
77 class Application is
78     method config() is
79         editor = new Editor()
80
81         logger = new LoggingListener(
82             "/path/to/log.txt",
83             "Someone has opened the file: %s")
84         editor.events.subscribe("open", logger)
85
86         emailAlerts = new EmailAlertsListener(
87             "admin@example.com",
88             "Someone has changed the file: %s")
89         editor.events.subscribe("save", emailAlerts)
```

Possibilités d'application

 Utilisez le patron de conception Observateur quand des modifications de l'état d'un objet peuvent en impacter d'autres, et que l'ensemble des objets n'est pas connu à l'avance ou qu'il change dynamiquement.

 Ce problème est souvent rencontré lorsque l'on travaille sur des classes d'une interface utilisateur graphique. Par exemple, si vous créez des classes bouton personnalisées et que vous voulez que les clients puissent y ajouter du code déclenché par le clic d'un utilisateur.

L'observateur permet à tous les objets qui suivent l'interface souscripteur de s'inscrire aux notifications des événements des objets diffuseur. Vous pouvez ajouter le mécanisme de souscription à tous vos boutons et laisser les clients mettre leur code personnalisé dans des classes souscripteur personnalisées.

 Utilisez ce patron quand certains objets de votre application doivent en suivre d'autres, mais seulement pendant un certain temps ou dans des cas spécifiques.

 La liste d'inscription est dynamique, les souscripteurs peuvent donc rejoindre ou quitter la liste quand ils le désirent.



Mise en œuvre

1. Passez en revue votre logique métier et découpez-la en deux parties : la fonctionnalité principale (indépendante du reste du code) prendra le rôle du diffuseur ; le reste va être transformé en classes souscripteur.
2. Déclarez l'interface souscripteur. Elle doit au moins contenir une méthode `update`.
3. Déclarez l'interface diffuseur et écrivez des méthodes qui permettent d'ajouter et de retirer des objets souscripteur de la liste. Rappelez-vous que les diffuseurs doivent manipuler les souscripteurs uniquement en passant par l'interface des souscripteurs.
4. Décidez où vous allez mettre la liste de souscription ainsi que l'implémentation des méthodes de souscription. En général, ce code est presque identique pour tous les types de diffuseurs. Le mieux est donc de le mettre dans une classe abstraite directement dérivée de l'interface diffuseur. Les diffuseurs concrets étendent cette classe et héritent du comportement de la souscription.

Si vous implémentez ce patron dans une hiérarchie de classes existante, pensez à la composition : mettez la logique de souscription dans un objet séparé et faites en sorte que les diffuseurs l'utilisent.

5. Créez les classes concrètes Diffuseur. Chaque fois que quelque chose d'important se produit chez un diffuseur, il doit prévenir ses souscripteurs.
6. Implémentez les méthodes de notifications (`update`) dans les classes concrètes Souscripteur. La majorité des souscripteurs va vouloir des données du contexte qui concernent l'événement en question. Vous pouvez les envoyer en paramètre de la méthode de notification.

Mais il y a une autre possibilité. Lors de la réception d'une notification, le souscripteur peut aller chercher les données directement dans la notification. Dans ce cas, le diffuseur doit s'envoyer lui-même en paramètre de la méthode `update`. On peut également lier un diffuseur à un souscripteur de manière permanente dans le constructeur, mais cette possibilité est moins flexible.

7. Le client doit créer tous les souscripteurs nécessaires et les enregistrer auprès des diffuseurs.

⚠️ Avantages et inconvénients

- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouvelles classes souscripteur sans avoir à modifier le code du diffuseur (et inversement si vous avez une interface diffuseur).
- ✓ Vous pouvez établir des relations entre les objets lors du lancement de l'application.

- ✗ Les souscripteurs sont avertis dans un ordre aléatoire.

↔ Liens avec les autres patrons

- La Chaîne de responsabilité, la Commande, le Médiateur et l'Observateur proposent différentes solutions pour associer les demandeurs et les récepteurs.
 - La *chaîne de responsabilité* envoie une demande ordonnée qui est passée tout au long d'une chaîne dynamique de récepteurs potentiels, jusqu'à ce que l'un d'entre eux décide de la traiter.
 - La *commande* établit des connexions unidirectionnelles entre les demandeurs et les récepteurs.
 - Le *médiateur* élimine les liens directs entre les demandeurs et les récepteurs, et les force à communiquer indirectement via un objet médiateur.
 - L'*observateur* permet aux récepteurs de s'inscrire et de se désinscrire dynamiquement à la réception des demandes.
- La différence entre le Médiateur et l'Observateur est souvent très fine. Dans la majorité des cas, vous pouvez implémenter l'un ou l'autre, mais parfois vous pouvez les utiliser simultanément. Regardons comment faire.

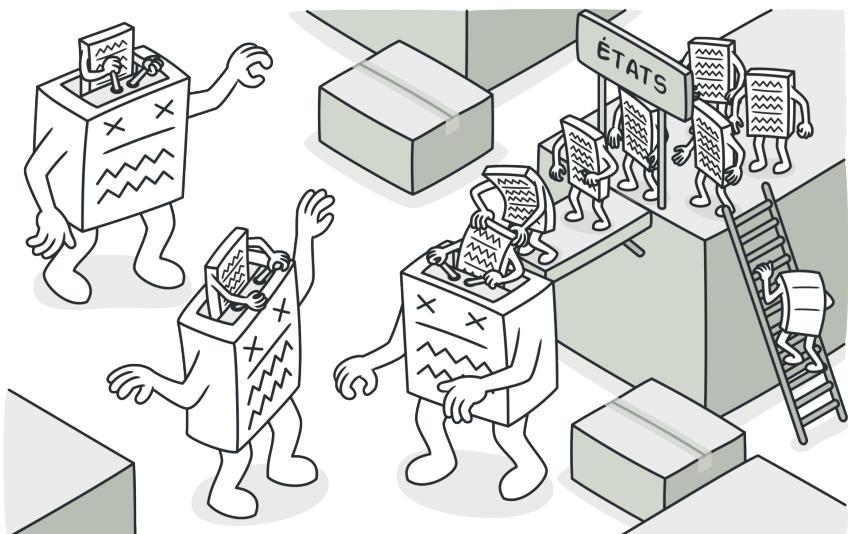
Le but principal du *médiateur* est d'éliminer les dépendances mutuelles entre un ensemble de composants du système. À la place, ces composants peuvent devenir dépendants d'un

unique objet médiateur. Le but de l'*observateur* est d'établir des connexions dynamiques à sens unique entre les objets, où certains objets peuvent être les subordonnés d'autres objets.

Il existe une implémentation populaire du *médiateur* qui repose sur l'*observateur*. L'objet médiateur joue le rôle du diffuseur et les composants agissent comme des souscripteurs qui s'inscrivent et se désinscrivent des événements du médiateur. Lorsque ce type de conception est mis en place, le *médiateur* ressemble de près à l'*observateur*.

Si vous êtes un peu perdu, rappelez-vous qu'il y a plusieurs manières d'implémenter le médiateur. Par exemple, vous pouvez associer de manière permanente tous les composants au même objet médiateur. Cette implémentation ne ressemblera pas à l'*observateur*, mais sera tout de même une instance du patron de conception médiateur.

Maintenant, imaginez un programme dont tous les composants sont devenus des diffuseurs, permettant des connexions dynamiques les uns avec les autres. Nous n'aurons pas d'objet médiateur centralisé, seulement un ensemble d'observateurs distribués.



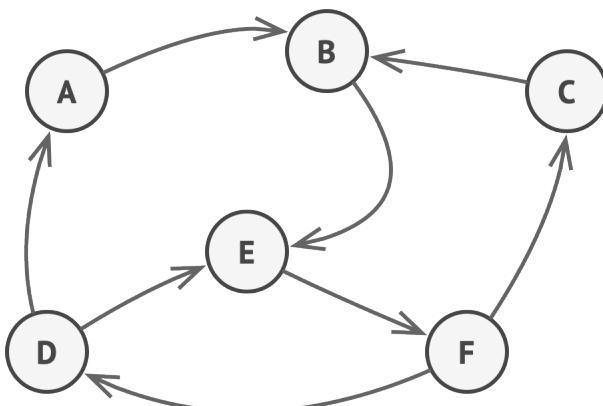
ÉTAT

Alias : State

État est un patron de conception comportemental qui permet de modifier le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change de classe.

(:() Problème

Le patron de conception est très proche du concept de l'*Automate fini*¹.



Automate fini.

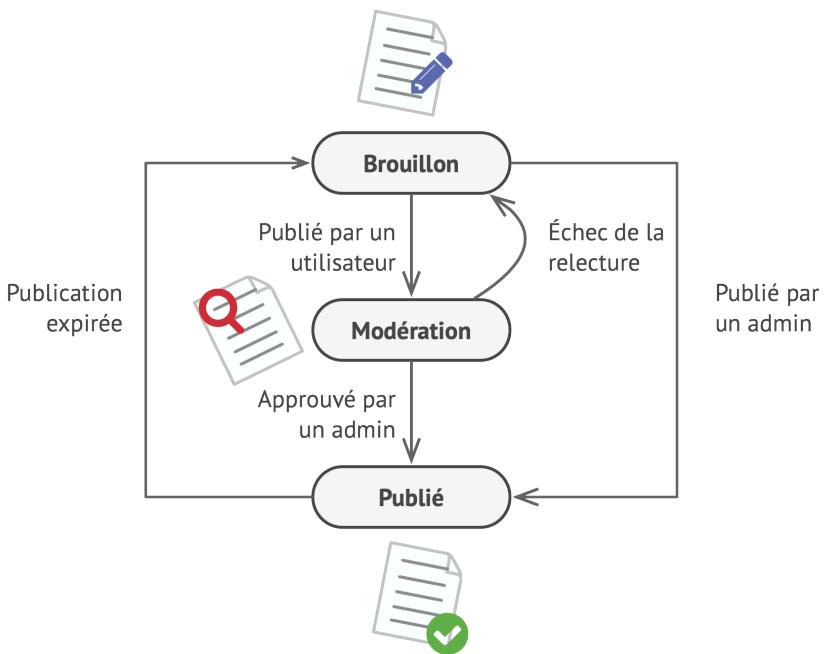
Le principe repose sur le fait qu'un programme possède un nombre *fini* d'états. Le programme se comporte différemment selon son état et peut en changer instantanément. En revanche, selon l'état dans lequel il se trouve, certains états ne lui sont pas accessibles. Ces règles de changement d'état sont appelées *transitions*. Elles sont également finies et prédéterminées.

Vous pouvez appliquer cette approche aux objets. Imaginons une classe `Document`. Un document peut être dans l'un des trois états suivants : `Brouillon` (draft), `Modération` et `Publié`.

1. Automate fini: <https://refactoring.guru/fr/fsm>

La méthode `publier` du document fonctionne un peu différemment en fonction de son état :

- Dans `Brouillon`, elle passe le document en modération.
- Dans `Modération`, elle rend le document public si l'utilisateur actuel est un administrateur.
- Dans `Publié`, elle ne fait rien du tout.



Les états et transitions possibles d'un objet document.

Les automates sont généralement implémentés avec beaucoup d'opérateurs conditionnels (`if` ou `switch`) qui choisissent le comportement approprié en fonction de l'état actuel de l'objet. Cet « état » se limite souvent à un ensemble de valeurs

dans les attributs de l'objet. Même si vous n'avez jamais entendu parler des automates finis, vous avez probablement déjà implémenté un état au moins une fois. La structure du code suivant vous dit-elle quelque chose ?

```
1 class Document is
2     field state: string
3     // ...
4     method publish() is
5         switch (state)
6             "draft":
7                 state = "moderation"
8                 break
9             "moderation":
10                if (currentUser.role == "admin")
11                    state = "published"
12                    break
13                "published":
14                    // Do nothing.
15                    break
16        // ...
```

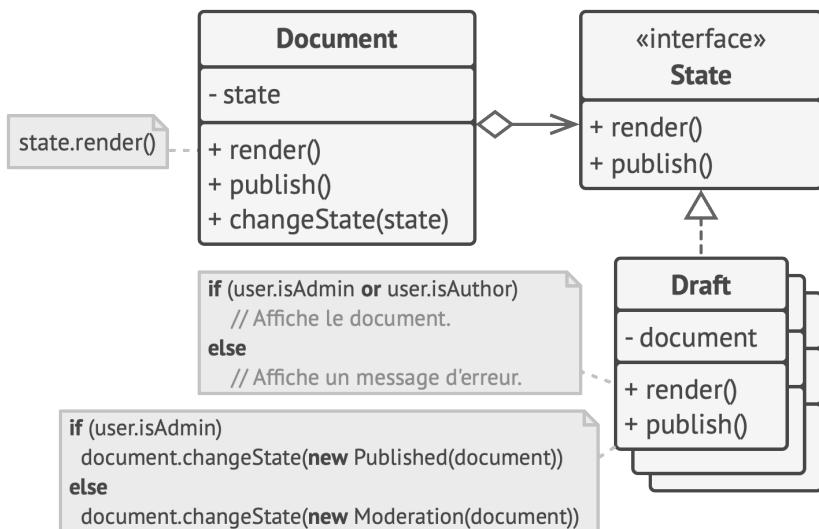
La plus grosse faiblesse de l'automate fini devient visible lorsque l'on commence à ajouter de plus en plus d'états et de comportements qui en sont dépendants à la classe `Document`. La majorité des méthodes va contenir dénormes blocs de conditions qui vont choisir le comportement d'une méthode en fonction de l'état actuel. Ce genre de code est très difficile à maintenir, car tout changement dans la logique de transi-

tion demande de modifier les états conditionnels dans chaque méthode.

Plus le projet évolue et plus cette faiblesse s'aggrave. Il est très difficile de prédire tous les états et transitions possibles lors de la phase de conception. Un automate fini doté d'un nombre limité de conditions peut se transformer en un bazar pas possible au bout d'un certain temps.

😊 Solution

Le patron de conception état propose de créer de nouvelles classes pour tous les états possibles d'un objet et d'extraire les comportements liés aux états dans ces classes.



Document délègue la tâche à un objet état.

Plutôt que d'implémenter tous les comportements de lui-même, l'objet original que l'on nomme *contexte*, stocke une référence vers un des objets état qui représente son état actuel. Il délègue tout ce qui concerne la manipulation des états à cet objet.

Pour faire passer le contexte dans un autre état, remplacez l'objet état par un autre qui représente son nouvel état. Vous ne pourrez le faire que si toutes les classes suivent la même interface et si le contexte utilise cette dernière pour manipuler ces objets.

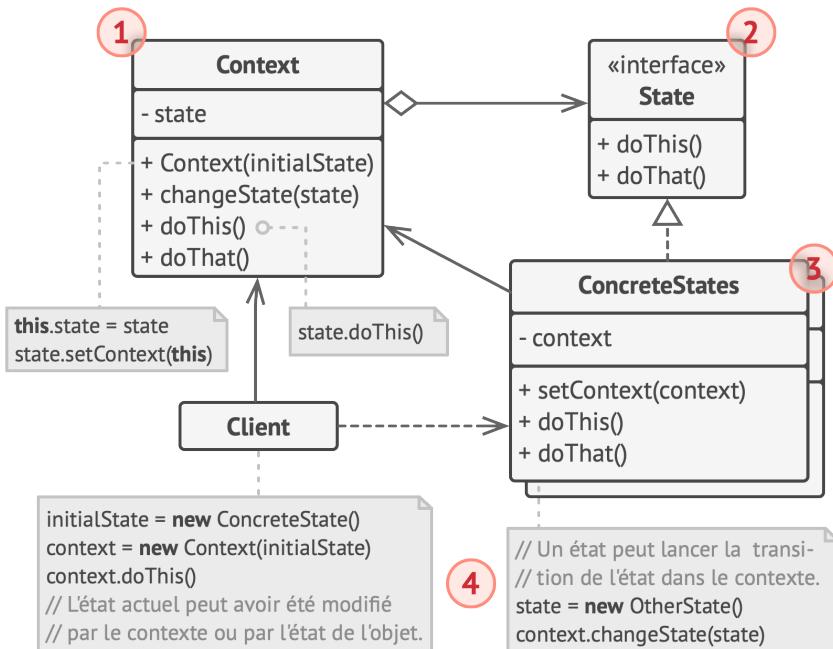
Cette structure ressemble de près au patron de conception **Stratégie**, mais il y a une différence majeure. Dans le patron de conception état, les états ont de la visibilité entre eux et peuvent lancer les transitions d'un état à l'autre, alors que les stratégies ne peuvent pas se voir.

Analogie

Les boutons de votre smartphone fonctionnent différemment selon l'état de l'appareil :

- Si le téléphone est déverrouillé, appuyer sur des boutons lance différentes fonctionnalités.
- Si le téléphone est verrouillé, appuyer sur n'importe quel bouton envoie sur l'écran de déverrouillage.
- Si la batterie du téléphone est faible, appuyer sur n'importe quel bouton montre l'écran de charge.

Structure



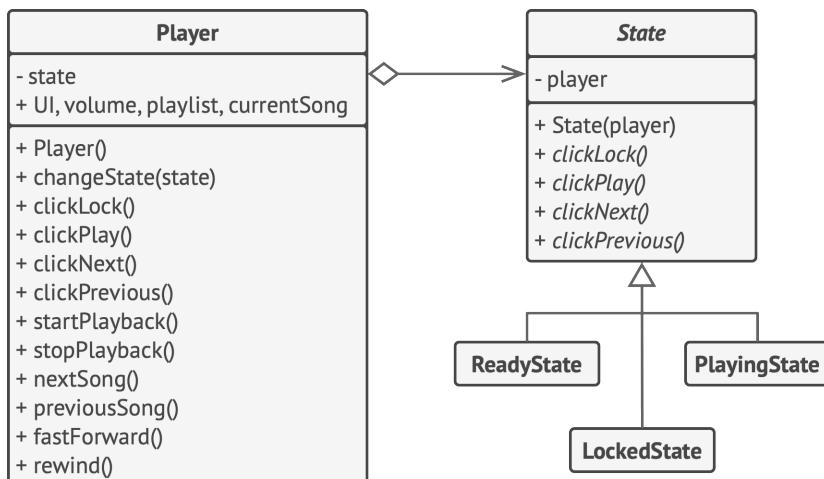
1. Le **Contexte** stocke une référence vers un des objets concrets État et lui délègue toutes les tâches concernant les états. Il utilise l'interface état pour communiquer avec l'objet état. Il expose un setter pour lui passer un nouvel état.
2. L'interface **État** déclare les méthodes spécifiques aux états. Ces méthodes doivent fonctionner avec tous les états concrets : des méthodes inutiles qui ne sont jamais appelées à l'intérieur de vos états sont à proscrire.
3. Les **États Concrets** fournissent leurs propres implémentations aux méthodes qui agissent sur les états. Pour éviter d'écrire le

même code dans les différents états, vous pouvez créer des classes abstraites intermédiaires qui encapsulent les comportements identiques.

Les états peuvent garder une référence vers le contexte. Grâce à cette référence, l'état peut récupérer des informations depuis le contexte et lancer des transitions.

4. Le contexte et les états concrets peuvent modifier le prochain état du contexte et lancer une transition en remplaçant l'état lié au contexte.

Pseudo-code



Un exemple de modification du comportement de l'objet effectué à l'aide d'objets état.

Dans cet exemple, le patron de conception **État** permet aux touches du lecteur multimédia d'avoir un comportement relatif à l'état actuel de la lecture.

L'objet principal du lecteur est toujours associé à un objet état qui effectue la majeure partie du travail pour le lecteur. Certaines actions remplacent l'état actuel du lecteur par un autre, modifiant sa manière de réagir aux interactions de l'utilisateur.

```
1 // La classe lecteurAudio prend le rôle du contexte. Elle
2 // maintient également une référence vers l'instance de l'une
3 // des classes état qui représente l'état actuel du lecteur
4 // audio.
5 class AudioPlayer is
6     field state: State
7     field UI, volume, playlist, currentSong
8
9     constructor AudioPlayer() is
10        this.state = new ReadyState(this)
11
12        // Le contexte délègue la gestion des interventions de
13        // l'utilisateur à un objet état. Le résultat va
14        // évidemment dépendre de l'état actuel, puisque chaque
15        // état réagit différemment aux manipulations des
16        // utilisateurs.
17        UI = new UserInterface()
18        UI.lockButton.onClick(this.clickLock)
19        UI.playButton.onClick(this.clickPlay)
20        UI.nextButton.onClick(this.clickNext)
21        UI.prevButton.onClick(this.clickPrevious)
22
```

```
23 // Les autres objets doivent pouvoir changer l'état du
24 // lecteur audio.
25 method changeState(state: State) is
26     this.state = state
27
28 // Les méthodes de l'UI déléguent l'exécution à l'état
29 // actuel.
30 method clickLock() is
31     state.clickLock()
32 method clickPlay() is
33     state.clickPlay()
34 method clickNext() is
35     state.clickNext()
36 method clickPrevious() is
37     state.clickPrevious()
38
39 // Un état peut appeler les méthodes d'un service sur le
40 // contexte.
41 method startPlayback() is
42     ...
43 method stopPlayback() is
44     ...
45 method nextSong() is
46     ...
47 method previousSong() is
48     ...
49 method fastForward(time) is
50     ...
51 method rewind(time) is
52     ...
53
54
```

```
55 // La classe de base état déclare des méthodes que tous les
56 // états concrets doivent obligatoirement implémenter et fournit
57 // aussi une référence arrière vers l'objet du contexte associé
58 // à l'état. Les états peuvent utiliser cette référence arrière
59 // pour permuter l'état du contexte.
60 abstract class State is
61     protected field player: AudioPlayer
62
63     // Le contexte s'envoie lui-même au constructeur de l'état,
64     // permettant de donner un coup de pouce à l'état pour
65     // récupérer des données contextuelles si nécessaire.
66     constructor State(player) is
67         this.player = player
68
69     abstract method clickLock()
70     abstract method clickPlay()
71     abstract method clickNext()
72     abstract method clickPrevious()
73
74
75     // Les états concrets implémentent différents comportements
76     // associés à un état du contexte.
77     class LockedState extends State is
78
79         // Lorsque vous déverrouillez un lecteur verrouillé, il peut
80         // prendre l'un des deux états.
81         method clickLock() is
82             if (player.playing)
83                 player.changeState(new PlayingState(player))
84             else
85                 player.changeState(new ReadyState(player))
86
```

```
87     method clickPlay() is
88         // Verrouillé, ne rien faire.
89
90     method clickNext() is
91         // Verrouillé, ne rien faire.
92
93     method clickPrevious() is
94         // Verrouillé, ne rien faire.
95
96
97     // Ils peuvent également déclencher les transitions de l'état
98     // dans le contexte.
99
100    class ReadyState extends State is
101        method clickLock() is
102            player.changeState(new LockedState(player))
103
104        method clickPlay() is
105            player.startPlayback()
106            player.changeState(new PlayingState(player))
107
108        method clickNext() is
109            player.nextSong()
110
111        method clickPrevious() is
112            player.previousSong()
113
114    class PlayingState extends State is
115        method clickLock() is
116            player.changeState(new LockedState(player))
117
118        method clickPlay() is
```

```
119     player.stopPlayback()  
120     player.changeState(new ReadyState(player))  
121  
122     method clickNext() is  
123         if (event.doubleclick)  
124             player.nextSong()  
125         else  
126             player.fastForward(5)  
127  
128     method clickPrevious() is  
129         if (event.doubleclick)  
130             player.previous()  
131         else  
132             player.rewind(5)
```

💡 Possibilités d'application

- ⌚ Utilisez le patron de conception état lorsque le comportement de l'un de vos objets varie en fonction de son état, qu'il y a beaucoup d'états différents et que ce code change souvent.
- ⚡ Ce patron vous propose d'extraire tout le code lié aux états et de le mettre dans des classes distinctes. Ceci vous permet d'ajouter de nouveaux états ou de modifier ceux qui existent indépendamment des autres, et de réduire les coûts de maintenance.

-  **Utilisez ce patron si l'une de vos classes est polluée par d'énormes blocs conditionnels qui modifient le comportement de la classe en fonction de la valeur de ses attributs.**
-  Le patron de conception état vous permet d'extraire des branches de ces conditions et de les transformer en méthodes dans les classes état. Tout en faisant vos modifications, vous pouvez retirer les attributs temporaires et les méthodes qui gèrent les changements d'état du code de votre classe principale.
-  **Utilisez ce patron de conception si vous avez trop de code dupliqué dans des états et transitions similaires de votre automate.**
-  Le patron de conception état vous permet d'assembler des hiérarchies de classes état et de réduire la duplication de code en regroupant le code commun dans des classes de base abstraites.

Mise en œuvre

1. Choisissez la classe qui va prendre le rôle du contexte. Cette classe peut déjà exister et posséder du code qui gère les états, mais vous pouvez en créer une nouvelle si ce code est réparti dans plusieurs classes.

2. Déclarez l'interface état. Vous pourriez très bien vous contenter de recopier toutes les méthodes déclarées dans le contexte, mais ne reprenez que celles qui concernent les états.
3. Pour chaque état, créez une classe qui dérive de l'interface état. Parcourez ensuite les méthodes du contexte pour identifier le code qui concerne cet état et recopiez-le dans votre nouvelle classe.

En effectuant cette manipulation, vous pourriez tomber sur des membres privés dans le contexte. Il y a plusieurs moyens de contournement :

- Rendez ces attributs ou ces méthodes publics.
 - Transformez le comportement que vous extrayez en méthode publique que vous mettez dans le contexte, puis appelez-la depuis la classe état. Ce n'est pas le plus esthétique, mais vous pourrez revenir dessus plus tard.
 - Imbriquez les classes état dans la classe contexte si votre langage de programmation le permet.
4. Dans votre classe contexte, ajoutez un attribut qui référence le type de l'interface état et un setter public qui permet de redéfinir la valeur de cet attribut.
 5. Parcourez à nouveau les méthodes du contexte et remplacez les conditions concernant les états par des appels aux méthodes correspondantes de l'objet état.

6. Pour changer l'état du contexte, créez une instance de l'une des classes état et passez-la au contexte. Ceci peut être fait à l'intérieur du contexte, dans les différents états ou dans le client. Où qu'elle soit, cette classe devient dépendante de la classe concrète État qu'elle instancie.

Avantages et inconvénients

- ✓ *Principe de responsabilité unique.* Organisez le code lié aux différents états dans des classes séparées.
- ✓ *Principe ouvert/fermé.* Ajoutez de nouveaux états sans modifier les classes état ou le contexte existants.
- ✓ Simplifiez le code du contexte en éliminant les gros blocs conditionnels de l'automate.
- ✗ L'utilisation de ce patron est un peu exagérée si votre automate n'a que quelques états ou qu'il y a peu de transitions.

Liens avec les autres patrons

- Le Pont, l'État, la Stratégie (et dans une certaine mesure l'Adaptateur) ont des structures très similaires. En effet, ces patrons sont basés sur la composition, qui délègue les tâches aux autres objets. Cependant, ils résolvent différents problèmes. Un patron n'est pas juste une recette qui vous aide à structurer votre code d'une certaine manière. C'est aussi une façon de communiquer aux autres développeurs le problème qu'il résout.

- L'**État** peut être considéré comme une extension de la **Stratégie**. Ces deux patrons de conception sont basés sur la composition : ils changent le comportement du contexte en déléguant certaines tâches aux objets assistant. La *stratégie* rend ces objets complètement indépendants sans aucune visibilité l'un sur l'autre. Cependant, l'*état* n'impose pas de restrictions sur les dépendances entre les états concrets, et leur laisse modifier l'état du contexte à volonté.



STRATÉGIE

Alias : Politique, Strategy

Stratégie est un patron de conception comportemental qui permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables.

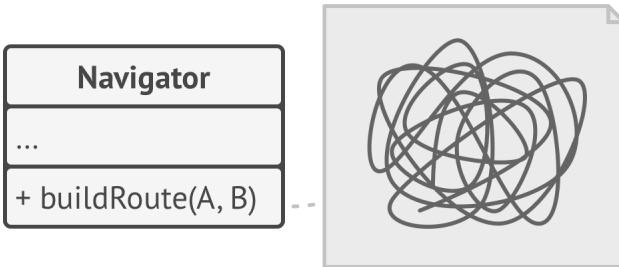
(:() Problème

Un beau jour, vous avez décidé de créer une application de navigation pour les voyageurs occasionnels. Vous l'avez développé avec une superbe carte comme fonctionnalité principale, qui aide les utilisateurs à s'orienter rapidement dans n'importe quelle ville.

La fonctionnalité la plus demandée était la planification d'itinéraire. Un utilisateur devrait pouvoir entrer une adresse et le chemin le plus rapide pour arriver à destination s'afficherait sur la carte.

La première version de l'application ne pouvait tracer des itinéraires que sur les routes. Les automobilistes étaient comblés. Mais apparemment, certaines personnes préfèrent utiliser d'autres moyens de locomotion pendant leurs vacances. Vous avez ajouté la possibilité de créer des trajets à pied dans la version suivante. Juste après cela, vous avez ajouté la possibilité d'utiliser les transports en commun dans les itinéraires.

Mais tout ceci n'était que le début. Vous avez continué en adaptant l'application pour les cyclistes, et plus tard, ajouté la possibilité de construire les itinéraires en passant par les attractions touristiques de la ville.



Le code du navigateur grossit à vue d'œil.

L'application a beau avoir très bien marché d'un point de vue financier, vous vous êtes arraché les cheveux sur le côté technique. Chaque fois que vous ajoutiez un nouvel algorithme pour tracer les itinéraires, la classe principale Navigateur doublait de taille. À un moment donné, la bête n'était plus possible à maintenir.

Que ce soit pour corriger un petit problème ou pour ajuster les scores des rues, la moindre touche apportée aux algorithmes impactait la totalité de la classe, augmentant les chances de créer des bugs dans du code qui fonctionnait très bien.

De plus, travailler en équipe n'était plus efficace. Les membres de votre équipe embauchés juste après la sortie et le succès de votre application se plaignaient de passer trop de temps à résoudre des problèmes de fusion. Ajouter une nouvelle fonctionnalité vous demandait de modifier une classe énorme, créant des conflits dans le code produit par les autres développeurs.

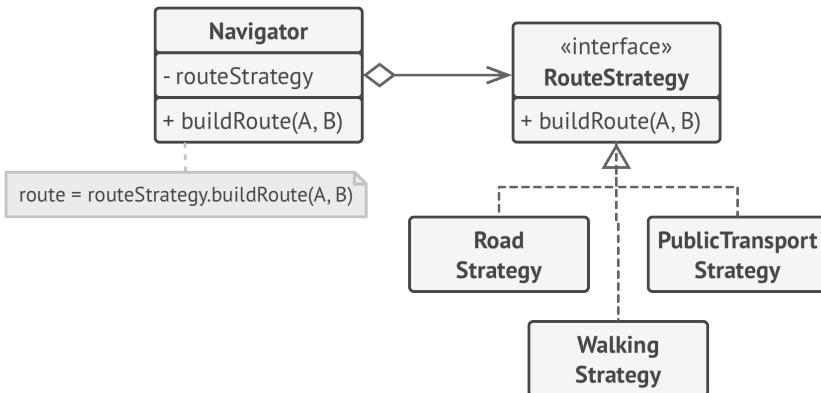
Solution

Le patron de conception stratégie vous propose de prendre une classe dotée d'un comportement spécifique mais qui l'exécute de différentes façons, et de décomposer ses algorithmes en classes séparées appelées *stratégies*.

La classe originale (le *contexte*) doit avoir un attribut qui garde une référence vers une des stratégies. Plutôt que de s'occuper de la tâche, le contexte la délègue à l'objet stratégie associé.

Le contexte n'a pas la responsabilité de la sélection de l'algorithme adapté, c'est le client qui lui envoie la stratégie. En fait, le contexte n'y connaît pas grand-chose en stratégies, c'est l'interface générique qui lui permet de les utiliser. Elle n'expose qu'une seule méthode pour déclencher l'algorithme encapsulé à l'intérieur de la stratégie sélectionnée.

Le contexte devient indépendant des stratégies concrètes. Vous pouvez ainsi modifier des algorithmes ou en ajouter de nouveaux sans toucher au code du contexte ou aux autres stratégies.

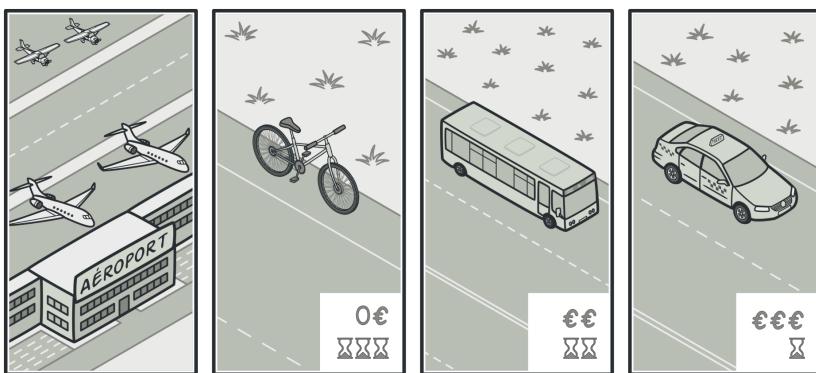


Stratégies de planification d'itinéraire.

Dans notre application de navigation, chaque algorithme d'itinéraire peut être extrait de sa propre classe avec une seule méthode `tracerItinéraire`. La méthode accepte une origine et une destination, puis retourne une liste de points de passage.

Quand bien même les différentes classes itinéraire ne donneraient pas un résultat identique avec les mêmes paramètres, la classe navigateur principale ne se préoccupe pas de l'algorithme sélectionné, car sa fonction première est d'afficher les points de passage sur la carte. La classe navigateur possède une méthode pour changer la stratégie d'itinéraire active afin que ses clients (les boutons de l'interface utilisateur par exemple) puissent remplacer le comportement sélectionné par un autre.

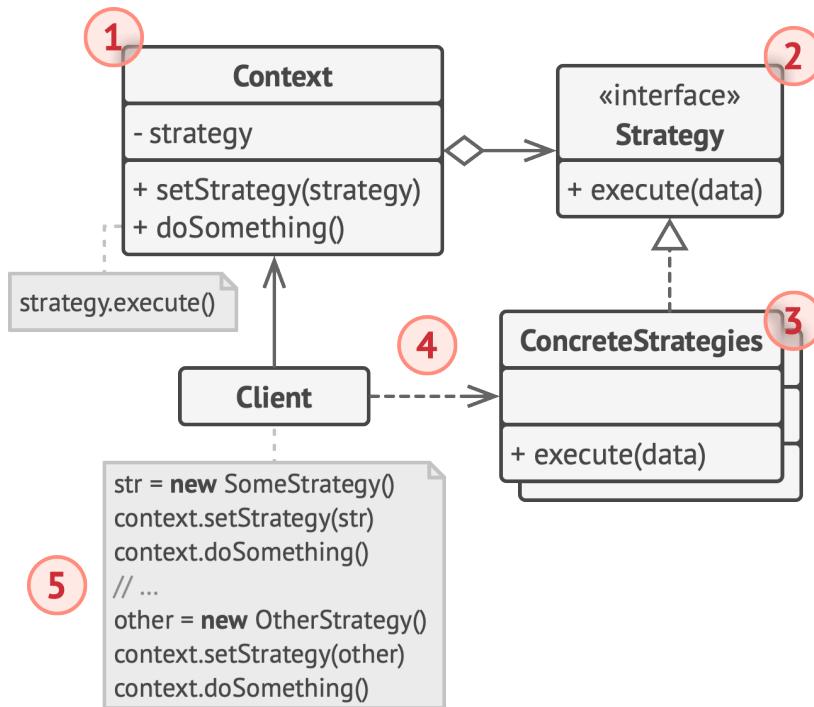
🚗 Analogie



Différentes stratégies pour se rendre à l'aéroport.

Imaginez que vous devez vous rendre à l'aéroport. Vous pouvez prendre le bus, appeler un taxi ou enfourcher votre vélo. Ce sont vos stratégies de transport. Vous pouvez sélectionner une de ces stratégies en fonction de certains facteurs, comme le budget ou les contraintes de temps.

Structure



1. Le **Contexte** garde une référence vers une des stratégies concrètes et communique avec cet objet uniquement au travers de l'interface stratégie.
2. L'interface **Stratégie** est commune à toutes les stratégies concrètes. Elle déclare une méthode que le contexte utilise pour exécuter une stratégie.
3. Les **Stratégies Concrètes** implémentent différentes variantes d'algorithme utilisées par le contexte.

4. Chaque fois qu'il veut lancer un algorithme, le contexte appelle la méthode d'exécution de l'objet stratégie associé. Le contexte ne sait pas comment la stratégie fonctionne ni comment l'algorithme est lancé.
5. Le **Client** crée un objet spécifique Stratégie et le passe au contexte. Le contexte expose un setter qui permet aux clients de remplacer la stratégie associée au contexte lors de l'exécution.

Pseudo-code

Dans cet exemple, le contexte utilise plusieurs **Stratégies** pour lancer diverses opérations arithmétiques.

```
1 // L'interface stratégie déclare les traitements communs à
2 // toutes les versions supportées de l'algorithme. Le contexte
3 // utilise cette interface pour appeler l'algorithme défini par
4 // les stratégies concrètes.
5 interface Strategy is
6     method execute(a, b)
7
8     // Les stratégies concrètes implémentent l'interface de base
9     // Stratégie et hébergent l'algorithme. L'interface les rend
10    // interchangeables dans le contexte.
11 class ConcreteStrategyAdd implements Strategy is
12     method execute(a, b) is
13         return a + b
14
15 class ConcreteStrategySubtract implements Strategy is
16     method execute(a, b) is
```

```
17     return a - b
18
19 class ConcreteStrategyMultiply implements Strategy is
20     method execute(a, b) is
21         return a * b
22
23 // Le contexte définit l'interface dont les clients ont besoin.
24 class Context is
25     // Le contexte maintient une référence à l'un des objets
26     // Stratégie. Le contexte ne connaît pas la classe concrète
27     // de la stratégie. Il doit manipuler toutes les stratégies
28     // via l'interface stratégie.
29     private strategy: Strategy
30
31     // En général, le contexte accepte une stratégie en
32     // paramètre du constructeur et fournit un setter pour
33     // permettre de changer de stratégie lors de l'exécution.
34     method setStrategy(Strategy strategy) is
35         this.strategy = strategy
36
37     // Le contexte délègue certaines tâches à l'objet stratégie,
38     // plutôt que d'implémenter plusieurs versions de
39     // l'algorithme.
40     method executeStrategy(int a, int b) is
41         return strategy.execute(a, b)
42
43
44     // Le code client choisit une stratégie concrète et la passe au
45     // contexte. Le client doit connaître les différences entre les
46     // stratégies afin de faire le bon choix.
47 class ExampleApplication is
48     method main() is
```

```
49     Create context object.  
50  
51     Read first number.  
52     Read last number.  
53     Read the desired action from user input.  
54  
55     if (action == addition) then  
56         context.setStrategy(new ConcreteStrategyAdd())  
57  
58     if (action == subtraction) then  
59         context.setStrategy(new ConcreteStrategySubtract())  
60  
61     if (action == multiplication) then  
62         context.setStrategy(new ConcreteStrategyMultiply())  
63  
64     result = context.executeStrategy(First number, Second number)  
65  
66     Print result.
```

💡 Possibilités d'application

- ⚡ Utilisez le patron de conception stratégie si vous voulez avoir différentes variantes d'un algorithme à l'intérieur d'un objet à disposition, et pouvoir passer d'un algorithme à l'autre lors de l'exécution.
- ⚡ Ce patron vous permet de modifier indirectement le comportement de l'objet lors de l'exécution, en l'associant avec différents sous-objets qui peuvent accomplir des sous-tâches spécifiques de différentes manières.

- ⚡ Utilisez la stratégie si vous avez beaucoup de classes dont la seule différence est leur façon d'exécuter un comportement.
 - ⚡ Ce patron vous permet d'extraire des variantes d'un comportement dans une hiérarchie de classes séparées et de combiner les classes originales dans une seule, évitant de dupliquer du code.
- ⚡ Utilisez la stratégie pour isoler la logique métier d'une classe, de l'implémentation des algorithmes dont les détails ne sont pas forcément importants pour le contexte.
 - ⚡ Ce patron vous permet de séparer le code, les données internes et les dépendances des divers algorithmes du reste du code. Une interface toute simple permet aux clients d'exécuter les algorithmes et d'en changer lors de l'exécution.
- ⚡ Utilisez ce patron si votre classe possède un gros bloc conditionnel qui choisit entre différentes variantes du même algorithme.
 - ⚡ La stratégie vous débarrasse de toutes ces conditions en extrayant tous les algorithmes dans des classes séparées, et ces dernières implémentent toutes la même interface. L'objet original délègue l'exécution à l'un de ces objets, au lieu d'implémenter toutes les variantes de l'algorithme.

Mise en œuvre

1. Dans la classe contexte, identifiez un algorithme qui varie souvent. Il peut s'agir d'un gros bloc conditionnel qui sélectionne une variante du même algorithme lors de l'exécution.
2. Déclarez l'interface stratégie commune à toutes les variantes de l'algorithme.
3. Extrayez tous les algorithmes un par un et mettez-les dans leurs propres classes. Elles doivent toutes implémenter l'interface stratégie.
4. Ajoutez un attribut pour garder une référence vers un objet stratégie dans la classe contexte. Créez un setter pour modifier le contenu de cet attribut. Le contexte ne doit manipuler l'objet stratégie qu'au travers de l'interface stratégie. Le contexte peut définir une interface qui laisse la stratégie accéder à ses données.
5. Les clients d'un contexte doivent l'associer avec une stratégie adaptée au comportement attendu.

Avantages et inconvénients

- ✓ Vous pouvez permuter l'algorithme utilisé à l'intérieur d'un objet à l'exécution.
- ✓ Vous pouvez séparer les détails de l'implémentation d'un algorithme et le code qui l'utilise.

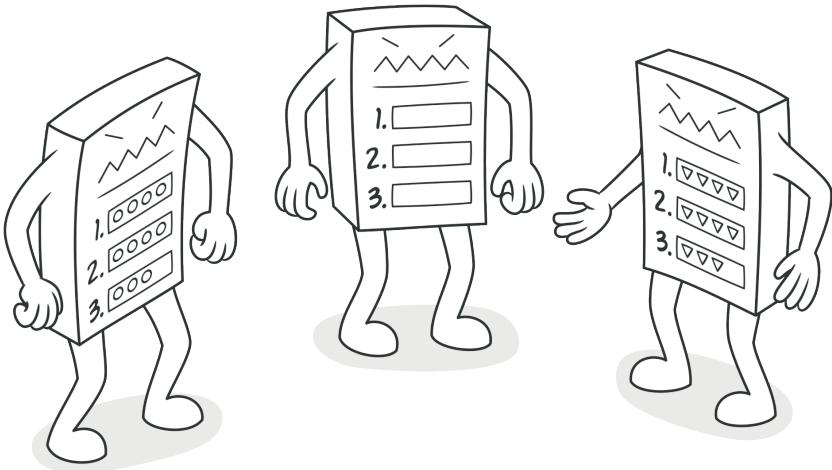
- ✓ Vous pouvez remplacer l'héritage par la composition.
- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter de nouvelles stratégies sans avoir à modifier le contexte.
- ✗ Si vous n'avez que quelques algorithmes qui ne varient pas beaucoup, nul besoin de rendre votre programme plus compliqué avec les nouvelles classes et interfaces qui accompagnent la mise en place du patron.
- ✗ Les clients doivent pouvoir comparer les différentes stratégies et choisir la bonne.
- ✗ De nombreux langages de programmation modernes gèrent les types fonctionnels et vous permettent d'implémenter différentes versions d'un algorithme à l'intérieur d'un ensemble de fonctions anonymes. Vous pouvez ensuite utiliser ces fonctions exactement comme vous le feriez pour des objets stratégie, sans encombrer votre code avec des classes et interfaces supplémentaires.

➡ Liens avec les autres patrons

- Le **Pont**, **l'État**, la **Stratégie** (et dans une certaine mesure **l'Adaptateur**) ont des structures très similaires. En effet, ces patrons sont basés sur la composition, qui délègue les tâches aux autres objets. Cependant, ils résolvent différents problèmes. Un patron n'est pas juste une recette qui vous aide à structurer votre code d'une certaine manière. C'est aussi une façon de communiquer aux autres développeurs le problème qu'il résout.

- La **Commande** et la **Stratégie** peuvent se ressembler, car vous les utilisez toutes les deux pour paramétrer un objet avec une action. Cependant, ces patrons ont des intentions très différentes.
 - Vous pouvez utiliser la *commande* pour convertir un traitement en un objet. Les paramètres du traitement deviennent des attributs de cet objet. La conversion vous permet de différer le lancement du traitement, le mettre dans une file d'attente, stocker l'historique des commandes, envoyer les commandes à des services distants, etc.
 - La *stratégie* quant à elle, décrit généralement différentes manières de faire la même chose et vous laisse permuter entre ces algorithmes à l'intérieur d'une unique classe contexte.
- Le **Décorateur** vous permet de changer la peau d'un objet, alors que la **Stratégie** vous permet de changer ses tripes.
- Le **Patron de méthode** est basé sur l'héritage : il vous laisse modifier certaines parties d'un algorithme en les étendant dans les sous-classes. La **Stratégie** est basée sur la composition : vous pouvez modifier certaines parties du comportement de l'objet en lui fournissant différentes stratégies qui correspondent à ce comportement. Le *patron de méthode* agit au niveau de la classe, il est donc statique. La *stratégie* agit au niveau de l'objet et vous laisse permuter les comportements à l'exécution.

- L'**État** peut être considéré comme une extension de la **Stratégie**. Ces deux patrons de conception sont basés sur la composition : ils changent le comportement du contexte en déléguant certaines tâches aux objets assistant. La *stratégie* rend ces objets complètement indépendants sans aucune visibilité l'un sur l'autre. Cependant, l'*état* n'impose pas de restrictions sur les dépendances entre les états concrets, et leur laisse modifier l'état du contexte à volonté.



PATRON DE MÉTHODE

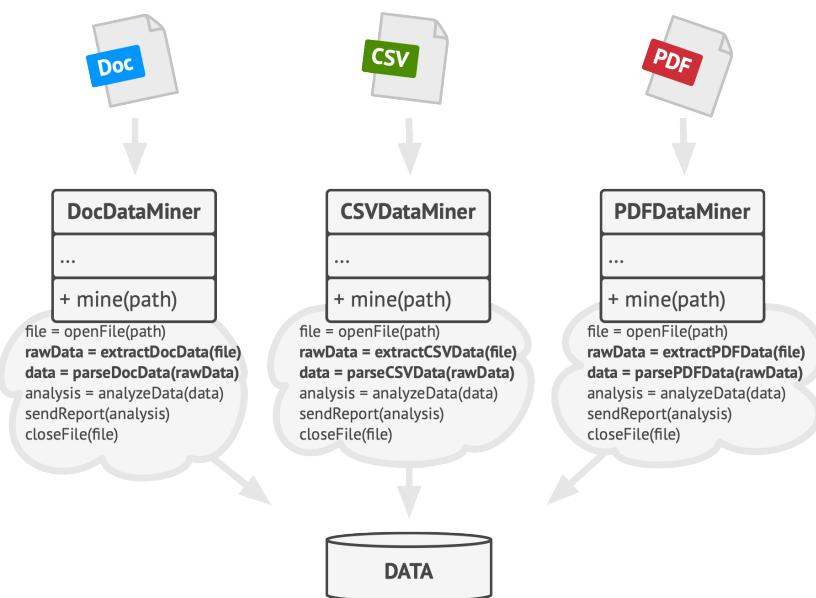
Alias : Méthode socle, Template Method

Patron de Méthode est un patron de conception comportemental qui permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure.

(:() Problème

Imaginez que vous êtes en train de créer une application de data mining (exploration de données) qui analyse les documents d'une entreprise. Les utilisateurs alimentent l'application avec différents formats (PDF, DOC, CSV) et celle-ci tente de récupérer les données utiles dans un format uniforme.

La première version de l'application ne fonctionnait qu'avec les fichiers DOC. Dans la version suivante, les fichiers CSV étaient acceptés. Un mois plus tard, vous lui avez « appris » à récupérer les données des fichiers PDF.



Les classes de data mining contiennent beaucoup de code dupliqué.

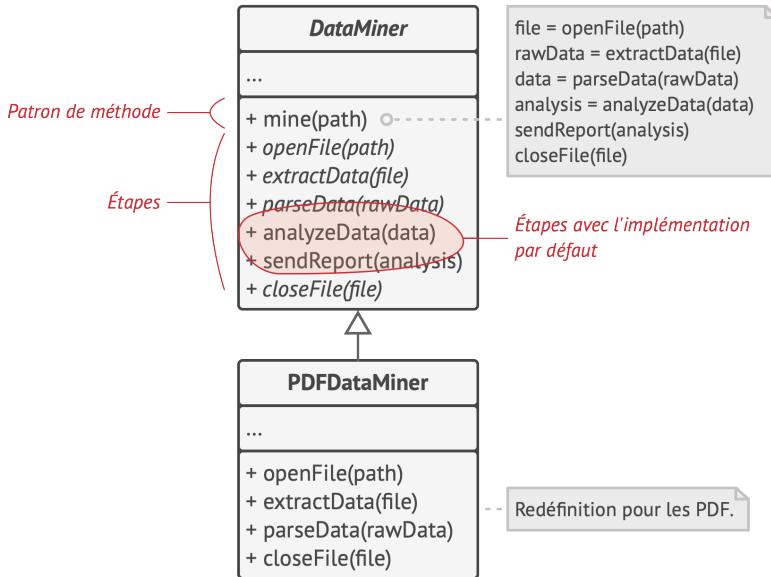
Au bout d'un moment, vous remarquez que ces trois classes comportent beaucoup de code similaire. Bien que le code qui gère les différents formats de données soit complètement différent d'une classe à l'autre, celui qui traite et analyse les données est presque identique. Ne serait-ce pas super de se débarrasser de tout ce code dupliqué tout en laissant la structure de l'algorithme intact ?

Un autre problème se pose avec le code client qui utilise ces classes. Il y a de gros blocs conditionnels qui choisissent un comportement en fonction de la classe de l'objet traité. Si ces trois classes avaient une interface commune (ou une classe de base), vous pourriez enlever toutes les conditions du code client et utiliser le polymorphisme lorsque vous appelez les méthodes sur un objet à traiter.

Solution

Le patron de méthode vous propose de découper un algorithme en une série d'étapes, de transformer ces étapes en méthodes et de mettre l'ensemble des appels à ces méthodes dans une seule méthode socle, le *patron de méthode*. Les étapes peuvent être **abstraites** ou avoir une implémentation par défaut. Pour utiliser l'algorithme, le client doit fournir sa propre sous-classe, implémenter toutes les étapes abstraites et redéfinir certaines d'entre elles si besoin (mais pas la méthode socle elle-même).

Mettons ceci en application dans notre logiciel de data mining. Nous pouvons créer une classe de base pour les trois algorithmes d'analyse syntaxique (parsing). Cette classe définit une méthode socle, composée d'une série d'appels à plusieurs étapes qui traitent les documents.



La méthode socle scinde l'algorithme en plusieurs étapes, permettant aux sous-classes de les redéfinir, mais on ne redéfinit pas la méthode socle elle-même.

En premier lieu, nous pouvons déclarer toutes les étapes abstraites afin de forcer les sous-classes à définir leur propre implémentation pour ces méthodes. Dans notre cas, les sous-classes ont déjà les implementations nécessaires. Nous devons donc uniquement ajuster les signatures des méthodes en les faisant correspondre à celles de la classe mère.

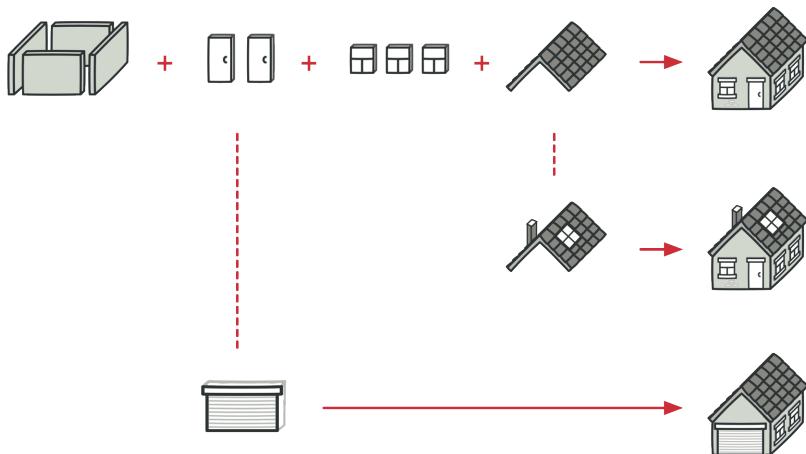
À présent, voyons ce que l'on peut faire pour se débarrasser du code dupliqué. Il semblerait que le code pour ouvrir/fermer les fichiers et récupérer/parser les données est différent pour chaque format de données, il n'y a donc aucune raison de toucher à ces méthodes. En revanche, les autres étapes (analyser les données brutes et établir les rapports) se ressemblent de près et leur implémentation peut donc être déplacée dans la classe de base, où les sous-classes se partagent le code.

Comme vous pouvez le voir, nous avons deux types d'étapes :

- les *opérations abstraites* qui doivent être implémentées dans chaque sous-classe.
- les *opérations facultatives* qui possèdent déjà une implémentation par défaut, mais peuvent être redéfinies si besoin.

Toutefois, un autre type d'étape existe : les *crochets* (hooks). Un crochet est une étape facultative dont le corps de méthode est laissé vide. Un patron de méthode peut fonctionner même si un crochet n'est pas redéfini. En général, les crochets sont placés avant ou après les étapes cruciales des algorithmes et procurent aux sous-classes des points d'extension supplémentaires.

Analogie

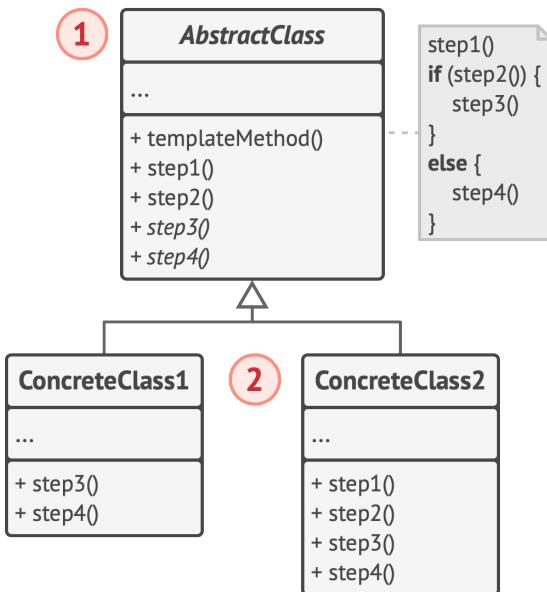


Un plan architectural typique peut être légèrement remanié pour mieux répondre aux besoins d'un client.

Le patron de méthode peut être utilisé pour construire des maisons en série. Le plan architectural pour construire une maison standard peut être doté de plusieurs points d'extensions qui permettent à un propriétaire potentiel d'ajuster certains détails dans la maison.

Chaque étape de construction (poser les fondations, poser la charpente, monter les murs, installer la plomberie pour l'eau et les câbles pour l'électricité, etc.) peut être légèrement modifiée pour différencier un peu la maison des autres.

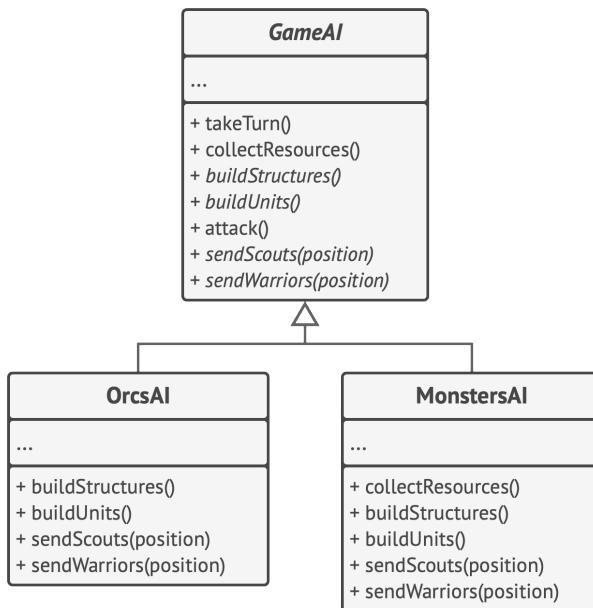
Structure



1. La **Classe Abstraite** déclare des méthodes (qui représentent les étapes d'un algorithme) et la méthode `patronDeMéthode` qui appelle toutes ces méthodes dans un ordre spécifique. Les étapes peuvent être déclarées abstraites ou posséder une implémentation par défaut.
2. Les **Classes Concrètes** peuvent redéfinir toutes les étapes, mais pas `patronDeMéthode`.

Pseudo-code

Dans cet exemple, le patron de conception **Patron de Méthode** fournit un squelette pour les branches de l'IA (intelligence artificielle) d'un jeu vidéo de stratégie simple.



Les classes d'IA dans un jeu vidéo simple.

Toutes les races du jeu ont à peu près les mêmes unités et bâtiments. Vous pouvez donc réutiliser la même structure d'IA pour les différentes races, tout en personnalisant certains détails. Grâce à cette approche, vous pouvez redéfinir l'IA des orcs pour la rendre plus agressive, obtenir des humains plus défensifs, et faire en sorte que les monstres ne puissent pas construire de bâtiments. Pour ajouter une autre race au jeu, vous devez simplement créer une nouvelle sous-classe d'IA et

redéfinir les méthodes par défaut déclarées dans la classe de base de l'IA.

```
1 // La classe abstraite définit un patron de méthode qui contient
2 // le squelette d'un algorithme. Ce dernier est généralement
3 // composé d'appels à des opérations primitives abstraites. Les
4 // sous-classes concrètes implémentent ces opérations, mais ne
5 // touchent pas au patron de méthode.
6 class GameAI is
7     // Le patron de méthode définit le squelette d'un
8     // algorithme.
9     method turn() is
10         collectResources()
11         buildStructures()
12         buildUnits()
13         attack()
14
15     // Certaines étapes peuvent être implémentées directement
16     // dans une classe de base.
17     method collectResources() is
18         foreach (s in this.builtStructures) do
19             s.collect()
20
21     // Et certaines peuvent être abstraites.
22     abstract method buildStructures()
23     abstract method buildUnits()
24
25     // Une classe peut avoir plusieurs patrons de méthode.
26     method attack() is
27         enemy = closestEnemy()
28         if (enemy == null)
```

```
29         sendScouts(map.center)
30     else
31         sendWarriors(enemy.position)
32
33     abstract method sendScouts(position)
34     abstract method sendWarriors(position)
35
36 // Les classes concrètes doivent implémenter toutes les
37 // opérations abstraites de la classe de base, mais elles ne
38 // doivent pas redéfinir le patron de méthode.
39 class OrcsAI extends GameAI is
40     method buildStructures() is
41         if (there are some resources) then
42             // Construit des fermes, des casernes, puis une
43             // forteresse.
44
45     method buildUnits() is
46         if (there are plenty of resources) then
47             if (there are no scouts)
48                 // Construit un ouvrier et l'ajoute au groupe
49                 // d'éclaireurs.
50             else
51                 // Construit un combattant et l'ajoute au groupe
52                 // des guerriers.
53
54     // ...
55
56     method sendScouts(position) is
57         if (scouts.length > 0) then
58             // Envoie les éclaireurs à la position.
59
60     method sendWarriors(position) is
```

```
61     if (warriors.length > 5) then
62         // Envoie les guerriers à la position.
63
64 // Les sous-classes peuvent redéfinir certaines opérations avec
65 // une implémentation par défaut.
66 class MonstersAI extends GameAI is
67     method collectResources() is
68         // Les monstres ne récoltent pas de ressources.
69
70     method buildStructures() is
71         // Les monstres ne fabriquent pas de bâtiments.
72
73     method buildUnits() is
74         // Les monstres ne construisent pas d'unités.
```

💡 Possibilités d'application

- ⚡ Utilisez le patron de méthode si vous voulez que vos clients puissent étendre des étapes spécifiques d'un algorithme, mais pas l'algorithme entier ou sa structure.
- ⚡ Le patron de méthode vous permet de transformer un algorithme monolithique en une série d'étapes individuelles qui peuvent être facilement étendues par des sous-classes, tout en gardant intacte la structure établie dans une classe mère.
- ⚡ Utilisez ce patron si vous avez plusieurs classes qui contiennent des algorithmes presque identiques, avec seulement quelques différences mineures. Par conséquent, vous risquez-

riez de devoir retoucher toutes les classes lorsque vous modifiez l'algorithme.

- ⚡ Lorsque vous transformez un tel algorithme en un patron de méthode, vous pouvez également remonter les étapes dotées d'implémentations similaires dans la classe mère, afin d'éviter la duplication de code. Vous pouvez laisser le reste du code dans les sous-classes.

Mise en œuvre

1. Analysez l'algorithme ciblé pour voir si vous pouvez le décomposer en étapes. Déterminez les étapes communes à toutes les sous-classes et celles qui sont uniques.
2. Créez une classe de base abstraite et déclarez le patron de méthode et un ensemble de méthodes abstraites pour représenter les opérations de l'algorithme. Faites une ébauche de la structure de l'algorithme dans ce patron de méthode en appelant les opérations correspondantes. Rendez ce patron `final` pour empêcher les sous-classes de la redéfinir.
3. Cela ne pose aucun problème si toutes les opérations sont abstraites, mais une implémentation par défaut bénéficierait à certaines opérations. Les sous-classes n'ont pas besoin d'implémenter ces méthodes.
4. Pensez à ajouter des crochets entre les étapes cruciales de votre algorithme.

5. Pour chaque variante de l'algorithme, créez une nouvelle sous-classe. Elle *doit* implémenter toutes les opérations abstraites, mais *peut* également redéfinir les opérations facultatives.

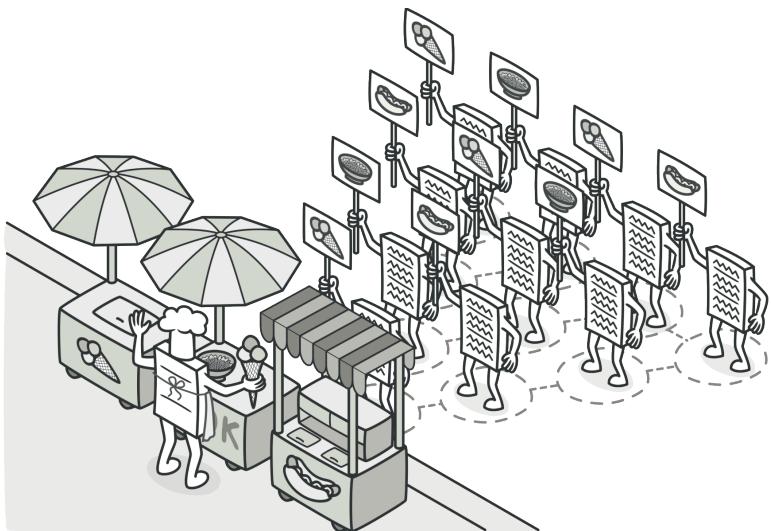
⚠️ Avantages et inconvénients

- ✓ Vous permettez aux clients de redéfinir certaines parties d'un grand algorithme. Elles sont ainsi moins affectées par les modifications apportées aux autres parties de l'algorithme.
- ✓ Vous pouvez remonter le code dupliqué dans la classe mère.
- ✗ Certains clients peuvent être limités à cause du squelette de l'algorithme.
- ✗ Vous ne respectez pas le *Principe de substitution de Liskov*, si vous supprimez l'implémentation d'une étape par défaut dans une sous-classe.
- ✗ Plus vous avez d'étapes, plus le patron de méthode devient difficile à maintenir.

➡️ Liens avec les autres patrons

- La **Fabrique** est une spécialisation du **Patron de méthode**. Une *fabrique* peut aussi faire office d'étape dans un grand *patron de méthode*.
- Le **Patron de méthode** est basé sur l'héritage : il vous laisse modifier certaines parties d'un algorithme en les étendant dans les sous-classes. La **Stratégie** est basée sur la composi-

tion : vous pouvez modifier certaines parties du comportement de l'objet en lui fournissant différentes stratégies qui correspondent à ce comportement. Le *patron de méthode* agit au niveau de la classe, il est donc statique. La *stratégie* agit au niveau de l'objet et vous laisse permuter les comportements à l'exécution.



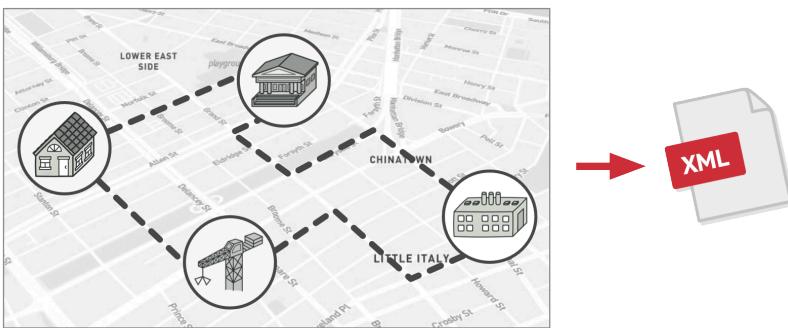
VISITEUR

Alias : Visitor

Visiteur est un patron de conception comportemental qui vous permet de séparer les algorithmes et les objets sur lesquels ils opèrent.

(:() Problème

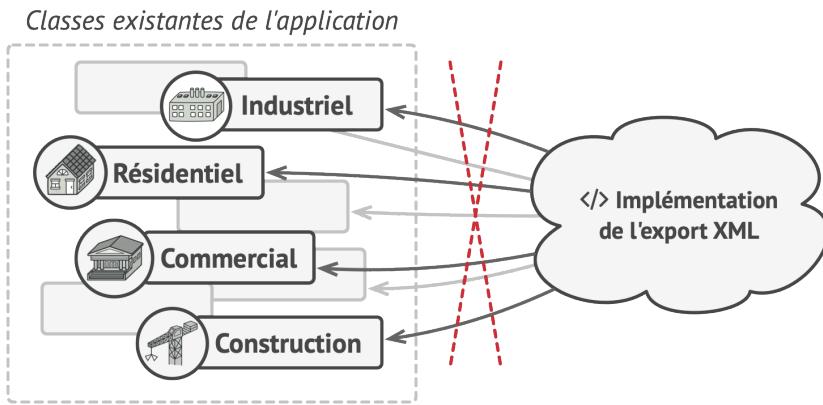
Imaginez que votre équipe développe une application avec des informations géographiques qui prennent la forme d'un graphe géant. Chaque nœud du graphe peut représenter une entité complexe comme une ville, mais aussi des choses plus spécifiques comme des usines, des sites touristiques, etc. Les nœuds sont interconnectés s'il est possible de les relier. Dans le code, chaque type de nœud est représenté par sa propre classe et chaque nœud spécifique est un objet.



Export du graphe en XML.

Le jour vient où vous devez vous attaquer à la mise en place de l'export du graphe au format XML. À première vue, cela semble assez simple. Vous avez prévu de mettre en place une méthode d'export pour chaque classe nœud, puis vous exécutez la méthode sur chaque nœud du graphe en le parcourant récursivement. La solution était simple et élégante : grâce au polymorphisme, vous n'avez pas couplé le code qui appellait la méthode d'exportation aux classes concrètes des nœuds.

Malheureusement, l'architecte du système vous a interdit de modifier les classes nœud existantes. Sa décision était basée sur le fait que le code était déjà en production et que vos modifications pourraient causer des bugs.



La méthode d'export XML devait être ajoutée dans toutes les classes nœud, ce qui risquait de compromettre l'intégrité du code de l'application.

De plus, il a remis en question la pertinence de placer le code d'export XML à l'intérieur des nœuds. Le rôle principal de ces classes est de manipuler les données géographiques, ce code serait perçu comme un intrus.

Il a avancé un autre argument contre votre modification. Il semblait très probable qu'une fois cette fonctionnalité en place, une personne du service marketing demande un export dans un format différent ou d'autres trucs bizarres. Cela vous obligerait à modifier une fois de plus ces petites classes fragiles.

Solution

Le patron de conception visiteur vous propose de placer ce nouveau comportement dans une classe séparée que l'on appelle *visiteur*, plutôt que de l'intégrer dans des classes existantes. L'objet qui devait lancer ce traitement à l'origine est maintenant passé en paramètre des méthodes du visiteur, ce qui permet à la méthode d'avoir accès à toutes les données nécessaires qui se trouvent à l'intérieur de l'objet.

Comment fait-on pour que ce comportement puisse être exécuté sur des objets de différentes classes ? Par exemple, dans le cas de notre export XML, l'implémentation sera probablement légèrement différente pour chaque nœud. La classe visiteur va donc avoir besoin d'un ensemble de méthodes et chacune d'entre elles pourra prendre des paramètres de différents types, comme ce qui suit :

```

1 class ExportVisitor implements Visitor {
2     method doForCity(City c) { ... }
3     method doForIndustry(Industry f) { ... }
4     method doForSightSeeing(SightSeeing ss) { ... }
5     // ...

```

Mais comment allons-nous appeler ces méthodes, surtout celles qui gèrent le graphe complet ? Nous ne pouvons pas utiliser le polymorphisme, car ces méthodes ont différentes signatures. Pour sélectionner une méthode qui peut traiter un

objet donné, nous devons vérifier sa classe. On se croirait dans un cauchemar !

```
1 foreach (Node node in graph)
2     if (node instanceof City)
3         exportVisitor.doForCity((City) node)
4     if (node instanceof Industry)
5         exportVisitor.doForIndustry((Industry) node)
6     // ...
7 }
```

Il vous est peut-être venu à l'esprit d'utiliser la surcharge. La surcharge, c'est une technique qui permet de donner le même nom à toutes les méthodes, même si elles n'ont pas des paramètres identiques. Malheureusement, même si l'on suppose que notre langage de programmation nous le permet (le Java et le C# par exemple), cela ne nous sera daucune aide. Comme nous ne connaissons pas la classe d'un nœud à l'avance, le mécanisme de la surcharge ne sera pas capable de déterminer la bonne méthode à exécuter. Il ira systématiquement chercher la méthode qui prend un objet de la classe de base `Nœud`.

Heureusement, le patron de conception visiteur résout ce problème. Il utilise une technique appelée **double répartition** (double dispatch), qui aide à lancer la bonne méthode sans s'encombrer avec des blocs conditionnels. Plutôt que de laisser le client choisir la version de la méthode à appeler, pourquoi ne déléguons-nous pas la décision aux objets que nous passons en paramètre au visiteur ? Comme les objets connais-

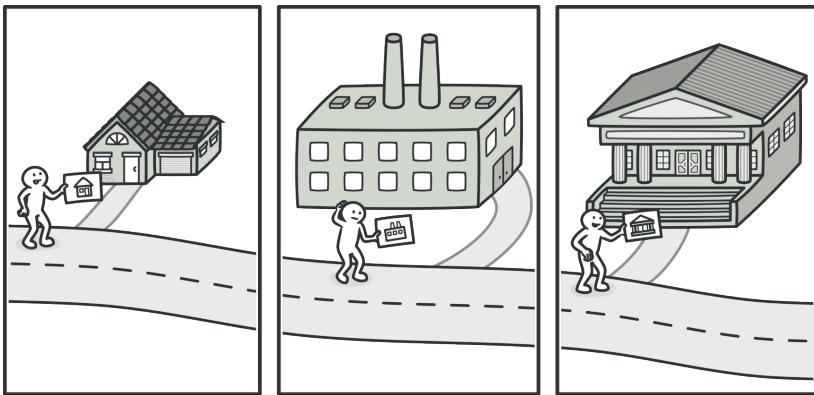
sent leur propre classe, ils seront plus à même de choisir la méthode adaptée au visiteur. Ils « acceptent » un visiteur et lui indiquent la méthode à exécuter.

```
1 // Code client
2 foreach (Node node in graph)
3     node.accept(exportVisitor)
4
5 // City
6 class City is
7     method accept(Visitor v) is
8         v.doForCity(this)
9     // ...
10
11 // Industry
12 class Industry is
13     method accept(Visitor v) is
14         v.doForIndustry(this)
15     // ...
```

J'avoue. Nous avons finalement été obligés de toucher aux classes nœud. Mais cette modification est mineure et elle nous permet d'ajouter de nouveaux comportements sans avoir à re-toucher le code.

À présent, si nous extrayons une interface pour tous les visiteurs, les nœuds existants vont accepter tous les visiteurs ajoutés dans l'application. Pour ajouter un nouveau comportement aux nœuds, il vous suffit d'implémenter une nouvelle classe visiteur.

🚗 Analogie

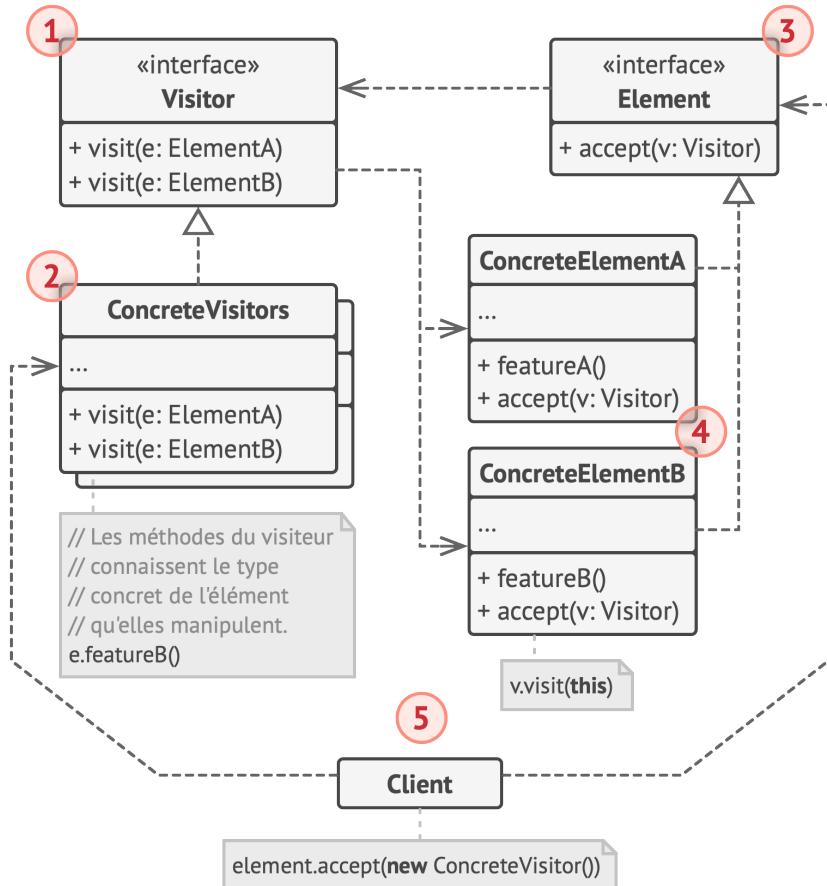


Un bon agent d'assurance est toujours prêt à proposer différents contrats pour différentes organisations.

Imaginez un agent d'assurance expérimenté qui veut absolument acquérir de nouveaux clients. Il peut faire du porte-à-porte dans tous les bâtiments d'un quartier et essayer de vendre des assurances à tous ceux qu'il rencontre. En fonction du type d'entreprise qui occupe le bâtiment, il peut proposer des contrats d'assurance adaptés :

- Si c'est un bâtiment résidentiel, il vend des assurances maladie.
- Si c'est une banque, il vend des assurances contre le vol.
- Si c'est un café, il vend des assurances contre les incendies et les inondations.

Structure

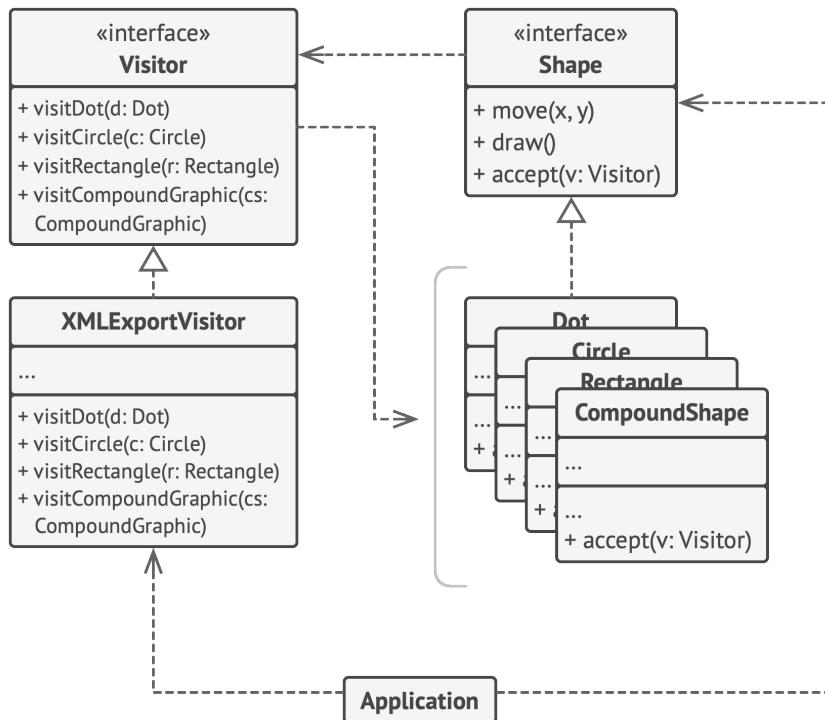


1. L'interface **Visiteur** déclare un ensemble de méthodes de parcours qui peuvent prendre les éléments concrets d'une structure d'objets en paramètre. Ces méthodes peuvent avoir le même nom si le programme est écrit dans un langage qui gère la surcharge, mais le type de ses paramètres sera différent.

2. Chaque **Visiteur Concret** implémente plusieurs versions des mêmes comportements, en fonction des classes des éléments concrets.
3. L'interface **Élément** déclare une méthode qui « accepte » les visiteurs. Cette méthode déclare un paramètre du type de l'interface visiteur.
4. Chaque **Élément Concret** doit implémenter une méthode d'acceptation. Le but de cette méthode est de rediriger l'appel vers la méthode appropriée du visiteur en fonction de la classe de l'élément actuel. Soyez conscient que même si la classe d'un élément de base implémente cette méthode, toutes les sous-classes doivent tout de même la redéfinir et appeler la méthode appropriée sur l'objet visiteur.
5. Le **Client** représente en général une collection ou tout autre objet complexe (par exemple un arbre **Composite**). En général, les clients n'ont pas de visibilité sur les classes des éléments concrets, car ils manipulent les objets de cette collection via une interface abstraite.

Pseudo-code

Dans cet exemple, le **Visiteur** implémente l'export XML dans la hiérarchie de classes des formes géométriques.



Exporter différents types d'objets vers le format XML à l'aide d'un objet visiteur.

```

1 // L'interface élément déclare une méthode `accepter` qui prend
2 // l'interface de base visiteur en argument.
3
4 interface Shape is
5     method move(x, y)
6     method draw()
7     method accept(v: Visitor)
8
9 // Chaque classe concrète Élément doit implémenter la méthode
10 // `accepter` et la faire appeler la méthode du visiteur qui
11 // correspond à la classe de l'élément.
12
13 class Dot implements Shape is
  
```

```
12 // ...
13
14 // Vous remarquerez que nous appelons `visiterPoint`, ce qui
15 // correspond au nom de la classe actuelle. Ainsi, nous
16 // fournissons le nom de la classe de l'élément au visiteur
17 // qui le manipule.
18 method accept(v: Visitor) is
19     v.visitDot(this)
20
21 class Circle implements Shape is
22     // ...
23     method accept(v: Visitor) is
24         v.visitCircle(this)
25
26 class Rectangle implements Shape is
27     // ...
28     method accept(v: Visitor) is
29         v.visitRectangle(this)
30
31 class CompoundShape implements Shape is
32     // ...
33     method accept(v: Visitor) is
34         v.visitCompoundShape(this)
35
36
37 // L'interface visiteur déclare un ensemble de méthodes visiter
38 // qui correspondent aux classes élément. La signature de la
39 // méthode visiter permet au visiteur d'identifier la classe
40 // exacte de l'élément qu'il manipule.
41 interface Visitor is
42     method visitDot(d: Dot)
43     method visitCircle(c: Circle)
```

```
44     method visitRectangle(r: Rectangle)
45     method visitCompoundShape(cs: CompoundShape)
46
47 // Les visiteurs concrets implémentent plusieurs versions du
48 // même algorithme. Ce dernier fonctionne avec toutes les
49 // classes concrètes Élément.
50 //
51 // Vous tirez tous les bénéfices du patron de conception
52 // visiteur lorsque vous l'utilisez avec une structure complexe
53 // d'objets, comme une arborescence. Dans ce cas, il peut être
54 // pratique de stocker certains états intermédiaires de
55 // l'algorithme tout en exécutant les méthodes du visiteur sur
56 // les différents objets de la structure.
57 class XMLExportVisitor implements Visitor is
58     method visitDot(d: Dot) is
59         // Exporte l'ID du point et ses coordonnées.
60
61     method visitCircle(c: Circle) is
62         // Exporte l'ID du cercle, les coordonnées de son centre
63         // et son rayon.
64
65     method visitRectangle(r: Rectangle) is
66         // Exporte l'ID du rectangle, les coordonnées du point
67         // supérieur gauche, ainsi que sa largeur et sa
68         // longueur.
69
70     method visitCompoundShape(cs: CompoundShape) is
71         // Exporte l'ID de la forme, ainsi qu'une liste des ID
72         // de ses enfants.
73
74
75 // Le code client peut lancer des traitements du visiteur sur
```

```
76 // n'importe quel ensemble d'éléments sans connaître leurs
77 // classes concrètes. La méthode accepter envoie un appel au
78 // traitement approprié de l'objet visiteur.
79 class Application is
80     field allShapes: array of Shapes
81
82     method export() is
83         exportVisitor = new XMLExportVisitor()
84
85         foreach (shape in allShapes) do
86             shape.accept(exportVisitor)
```

Si vous voulez savoir pourquoi nous avons besoin d'une méthode `accepter` dans cet exemple, vous pouvez consulter mon article [**Visiteur et Double répartition**](#) qui décrit le sujet en détail.

💡 Possibilités d'application

- ⚡ Utilisez le visiteur lorsque vous voulez lancer des traitements sur les éléments d'un objet ayant une structure complexe (une arborescence par exemple).
- ⚡ Le patron de conception visiteur vous permet de lancer des traitements sur un ensemble d'objets de différentes classes à l'aide d'un objet visiteur qui implémente une variante d'un même traitement pour chaque classe visée.

Utilisez le visiteur pour nettoyer la logique métier de tous les comportements secondaires.

 Ce patron vous permet de spécialiser encore plus les classes principales de votre application, en transférant les autres comportements dans des classes visiteur.

Utilisez le visiteur si un comportement n'est adapté que pour certaines classes d'une hiérarchie de classes, mais pas pour les autres.

 Vous pouvez envoyer ce comportement dans une classe visiteur séparée, implémenter seulement les méthodes de visite qui acceptent les objets des classes concernées et laisser le reste vide.

Mise en œuvre

1. Déclarez l'interface visiteur avec un ensemble de méthodes « visiter »; une pour chaque classe d'élément concret qui existe dans le programme.
2. Déclarez l'interface élément. Si vous avez déjà une hiérarchie de classes élément, ajoutez la méthode abstraite « accepter » à la classe de base de la hiérarchie. Cette méthode doit prendre un Visiteur en paramètre.
3. Implémentez les méthodes d'acceptation dans toutes les classes des éléments concrets. Ces méthodes doivent simple-

ment rediriger l'appel vers la méthode de visite de l'objet visiteur qui correspond à la classe de l'élément actuel.

4. Les classes élément doivent uniquement interagir avec les visiteurs via l'interface visiteur. En revanche, les visiteurs doivent avoir la visibilité sur toutes les classes des éléments concrets, qui sont référencés comme les types des paramètres des méthodes de visite.
5. Pour chaque comportement qui ne peut être écrit à l'intérieur de la hiérarchie des éléments, créez une nouvelle classe concrète Visiteur et implémentez toutes les méthodes de visite.

Vous pourriez vous retrouver dans une situation où le visiteur aura besoin d'un accès aux membres privés d'une classe élément. Dans ce cas, vous pouvez rendre ces attributs ou méthodes publics (ce qui ne respecte pas l'encapsulation de l'élément) ou imbriquer la classe visiteur dans la classe de l'élément. Cette dernière possibilité n'est envisageable que si votre langage de programmation gère les classes imbriquées.

6. Le client doit créer des objets visiteur et les passer dans des éléments via des méthodes « accepter ».

Avantages et inconvénients

- ✓ *Principe ouvert/fermé.* Vous pouvez ajouter un nouveau comportement qui acceptera les objets de différentes classes sans les modifier.
- ✓ *Principe de responsabilité unique.* Vous pouvez déplacer plusieurs versions du même comportement dans une seule classe.

- ✓ Un objet visiteur peut accumuler des informations utiles en manipulant différents objets. Cela peut se révéler pratique si vous voulez parcourir une structure complexe d'objets comme un arbre, et lancer le traitement du visiteur sur chaque objet de cette structure.
- ✗ Vous devez mettre à jour les visiteurs chaque fois qu'une classe est ajoutée ou retirée de la hiérarchie des éléments.
- ✗ Les visiteurs n'ont parfois pas les accès nécessaires aux attributs ou méthodes privés des éléments qu'ils sont censés manipuler.

↔ Liens avec les autres patrons

- Vous pouvez traiter le **Visiteur** comme une version plus puissante du patron de conception **Commande**. Ses objets peuvent lancer des traitements sur divers objets dans différentes classes.
- Vous pouvez utiliser le **Visiteur** pour lancer une opération sur un arbre **Composite** entier.
- Vous pouvez utiliser le **Visiteur** avec **l'Itérateur** pour parcourir une structure de données complexe et lancer un traitement sur ses éléments, même s'ils ont des classes différentes.

Conclusion

Félicitations ! Vous avez atteint la fin du livre !

Cependant, il existe encore de nombreux patrons de conception. J'espère que ce livre sera votre point de départ dans l'apprentissage des patrons et vous permettra de développer des compétences de conception dignes d'un super-héros.

Voici quelques suggestions pour la suite.

-  N'oubliez pas que vous avez également **accès à l'archive** des échantillons de code que vous pouvez télécharger dans divers langages de programmation.
-  Lisez le livre de Joshua Kerievsky « **Refactoring To Patterns** ».
-  Vous n'y connaissez rien en refactorisation ? **J'ai un cours à vous proposer.**
-  Imprimez vos propres **aide-mémoires pour patrons de conception** et mettez-les en évidence afin de pouvoir les consulter en permanence.
-  **Laissez un commentaire** sur ce livre. J'ai hâte de découvrir votre avis, même si vous avez beaucoup de choses à critiquer 😊